PostgreSQL 7.0.3 Documentation

The PostgreSQL Development Team

PostgreSQL 7.0.3 Documentation

by The PostgreSQL Development Team

Covering v7.0 for general release

PostgreSQL is Copyright © 1996-2000 by PostgreSQL Inc.

Table of Contents

Summary xii
I. User's Guide 1
1. Introduction 4
1.1. What is Postgres? 4
1.2. A Short History of Postgres 4
1.3. About This Release 6
1.4. Resources 6
1.5. Terminology 7
1.6. Notation 8
1.7. Problem Reporting Guidelines 8
1.8. Y2K Statement 11
1.9. Copyrights and Trademarks 12
2. SQL Syntax 13
2.1. Key Words 13
2.2. Comments 17
2.3. Names 17
2.4. Constants 17
2.5. Fields and Columns 19
2.6. Operators 19
2.7. Expressions 19
3. Data Types 22
3.1. Numeric Types 23
3.2. Monetary Type 24
3.3. Character Types 25
3.4. Date/Time Types 25
3.5. Boolean Type 30
3.6. Geometric Types 31
3.7. IP Version 4 Networks and Host Addresses 33
4. Operators 35
4.1. Lexical Precedence 35
4.2. General Operators 36
4.3. Numerical Operators 36
4.4. Geometric Operators 37
4.5. Time Interval Operators 38
4.6. IP V4 CIDR Operators 38
4.7. IP V4 INET Operators 39
5. Functions 40
5.1. SQL Functions 40
5.2. Mathematical Functions 40
5.3. String Functions 41
5.4. Date/Time Functions 42
5.5. Formatting Functions 43
5.6. Geometric Functions 47
5.7. IP V4 Functions 48
6. Type Conversion 49
6.1. Overview 49
6.2. Operators 50
6.3. Functions 52
6.4. Query Targets 54
6.5. UNION Queries 55
7. Indices and Keys 57
7.1. Keys 58
7.2. Partial Indices 59
8. Arrays 61
9. Inheritance 63

II. Administrator's Guide 312
22. Ports 315
22.1. Currently Supported Platforms 315
22.2. Unsupported Platforms 316
23. Configuration Options 318
23.1. Parameters for Configuration (configure) 318
23.2. Parameters for Building (make) 319
23.3. Locale Support 320
23.4. Kerberos Authentication 321
24. System Layout 323
25. Installation 325
25.1. Before you start 325
25.2. Installation Procedure 325
26. Installation on Win32 332
26.1. Building the libraries 332
26.2. Installing the libraries 332
26.3. Using the libraries 332
27. Runtime Environment 333
27.1. Using Postgres from Unix 333
27.2. Starting postmaster 333
27.3. Using pg_options 333
28. Security 338
28.1. User Authentication 338
28.2. User Names and Groups 340
28.3. Access Control 341
28.4. Functions and Rules 341
28.5. Secure TCP/IP Connection 342
29. Adding and Deleting Users 343
30. Disk Management 344
30.1. Alternate Locations 344
31. Managing a Database 346
31.1. Creating a Database 346
31.2. Accessing a Database 346
31.3. Destroying a Database 347
31.4. Backup and Restore 347
32. Troubleshooting 349
32.1. Postmaster Startup Failures 349
32.2. Client Connection Problems 349
32.3. Debugging Messages 350
33. Database Recovery 353
34. Regression Test 354
34.1. Regression Environment 354
34.2. Regression Test Procedure 355
34.3. Regression Analysis 356
34.4. Platform-specific comparison files 358
35. Release Notes 359
35.1. Release 7.0.3 359
III. Programmer's Guide 406
37. Architecture 409
37.1. Postgres Architectural Concepts 409
38. Extending SQL: An Overview 415
38.1. How Extensibility Works 415
38.2. The Postgres Type System 415
38.3. About the Postgres System Catalogs 415
39. Extending SQL: Functions 418
39.1. Query Language (SQL) Functions 418
39.2. Procedural Language Functions 421
39.3. Internal Functions 421

39.4. Compiled (C) Language Functions 421
39.5. Function Overloading 427
40. Extending SQL: Types 429
40.1. User-Defined Types 429
41. Extending SQL: Operators 431
41.1. Operator Optimization Information 431
42. Extending SQL: Aggregates 436
43. The Postgres Rule System 438
43.1. What is a Querytree? 438
43.2. Views and the Rule System 439
43.3. Rules on INSERT, UPDATE and DELETE 448
43.4. Rules and Permissions 458
43.5. Rules versus Triggers 458
44. Interfacing Extensions To Indices 462
45. Index Cost Estimation Functions 468
46. GiST Indices 471
47. Linking Dynamically-Loaded Functions 473
47.1. Linux 473
47.2. DEC OSF/1 474
47.3. SunOS 4.x, Solaris 2.x and HP-UX 474
48. Triggers 476
48.1. Trigger Creation 476
48.2. Interaction with the Trigger Manager 477
48.3. Visibility of Data Changes 479
48.4. Examples 479
49. Server Programming Interface 483
49.1. Interface Functions 483
49.2. Interface Support Functions 491
49.3. Memory Management 503
49.4. Visibility of Data Changes 504
49.5. Examples 504
50. Procedural Languages 507
50.1. Installing Procedural Languages 507
IV. Interfaces 509
51. Functions 512
52. Large Objects 513
52.1. Historical Note 513
52.2. Implementation Features 513
52.3. Interfaces 513
52.4. Built in registered functions 515
52.5. Accessing Large Objects from LIBPQ 515
52.6. Sample Program 515
53. ecpg - Embedded SQL in C 520
53.1. Why Embedded SQL? 520
53.2. The Concept 520
53.3. How To Use ecpg 520
53.4. Limitations 523
53.5. Porting From Other RDBMS Packages 523
53.6. Installation 524
53.7. For the Developer 524
54. libpq - C Library 530
54.1. Database Connection Functions 530
54.2. Query Execution Functions 535
54.3. Asynchronous Query Processing 539
54.4. Fast Path 541
54.5. Asynchronous Notification 542
54.6. Functions Associated with the COPY Command 543
54.7. libpq Tracing Functions 545

54.8. libpq Control Functions 545
54.9. Environment Variables 545
54.10. Threading Behavior 546
54.11. Sample Programs 546
55. libpq - C++ Binding Library 555
55.1. Control and Initialization 555
55.2. libpq++ Classes 556
55.3. Database Connection Functions 556
55.4. Query Execution Functions 557
55.5. Asynchronous Notification 560
55.6. Functions Associated with the COPY Command 560
56. pgtcl - TCL Binding Library 562
56.1. Commands 562
56.2. Examples 562
56.3. pgtcl Command Reference Information 563
57. libpgeasy - Simplified C Binding Library 582
58. ODBC Interface 583
58.1. Background 583
58.2. Windows Applications 583
58.3. Unix Installation 584
58.4. Configuration Files 587
58.5. ApplixWare 588
59. JDBC Interface 592
59.1. Building the JDBC Interface 592
59.2. Preparing the Database for JDBC 592
59.3. Using the Driver 593
59.4. Importing JDBC 593
59.5. Loading the Driver 593
59.6. Connecting to the Database 594
59.7. Issuing a Query and Processing the Result 594
59.8. Performing Updates 595
59.9. Closing the Connection 595
59.10. Using Large Objects 595
59.11. Postgres Extensions to the JDBC API 596
59.12. Further Reading 635
60. Lisp Programming Interface 636
V. Developer's Guide 637
61. Postgres Source Code 639
61.1. Formatting 639
62. Overview of PostgreSQL Internals 640
62.1. The Path of a Query 640
62.2. How Connections are Established 640
62.3. The Parser Stage 641
62.4. The Postgres Rule System 643
62.5. Planner/Optimizer 644
62.6. Executor 645
63. pg_options 647
64. Genetic Query Optimization in Database Systems 650
64.1. Query Handling as a Complex Optimization Problem 650
64.2. Genetic Algorithms (GA) 650
64.3. Genetic Query Optimization (GEQO) in Postgres 651
64.4. Future Implementation Tasks for Postgres GEQO 652
65. Frontend/Backend Protocol 653
65.1. Overview 653
65.2. Protocol 653
65.3. Message Data Types 658
65.4. Message Formats 658
66. Postgres Signals 665

67. gcc Default Optimizations 667
68. Backend Interface 668
68.1. BKI File Format 668
68.2. General Commands 668
68.3. Macro Commands 669
68.4. Debugging Commands 669
68.5. Example 670
69. Page Files 671
69.1. Page Structure 671
69.2. Files 672
69.3. Bugs 672
VI. Tutorial 673
70. SQL 675
70.1. The Relational Data Model 675
70.2. Relational Data Model Formalities 676
70.3. Operations in the Relational Data Model 677
70.4. The SQL Language 680
71. Architecture 692
71.1. Postgres Architectural Concepts 692
72. Getting Started 694
72.1. Setting Up Your Environment 694
72.2. Starting the Interactive Monitor (psql) 694
72.3. Managing a Database 695
73. The Query Language 698
73.1. Interactive Monitor 698
73.2. Concepts 698
73.3. Creating a New Class 698
73.4. Populating a Class with Instances 699
73.5. Querying a Class 699
73.6. Redirecting SELECT Queries 700
73.7. Joins Between Classes 700
73.8. Updates 701
73.9. Deletions 701
73.10. Using Aggregate Functions 701
74. Advanced Postgres SQL Features 703
74.1. Inheritance 703
74.2. Non-Atomic Values 704
74.3. More Advanced Features 705
VII. Appendices 706
UG1. Date/Time Support 708
UG1.1. Time Zones 708
UG1.2. History of Units 711
DG1. The CVS Repository 713
DG1.1. CVS Tree Organization 713
DG1.2. Getting The Source Via Anonymous CVS 714
DG1.3. Getting The Source Via CVSup 715
DG2. Documentation 720
DG2.1. Documentation Roadmap 720
DG2.2. The Documentation Project 720
DG2.3. Documentation Sources 721
DG2.4. Building Documentation 723
DG2.5. Manpages 724
DG2.6. Hardcopy Generation for v7.0 724
DG2.7. Toolsets 726
DG2.8. Alternate Toolsets 733
Bibliography 734

List of Figures

24.1. Postgres file layout 323
37.1. How a connection is established 412
38.1. The major Postgres system catalogs 416
71.1. How a connection is established 692

List of Tables

3.1. Data Types 22
3.2. Constants 23
3.3. Numerics 23
3.4. Money 25
3.5. Characters 25
3.6. Specialty Characters 25
3.7. Date/Time 25
3.8. Date Inputs 26
3.9. Month Abbreviations 26
3.10. Day of Week Abbreviations 27
3.11. Time Inputs 27
3.12. Time With Time Zone Inputs 28
3.13. Time Zone Inputs 28
3.14. Constants 29
3.15. Styles 29
3.16. Date Order 29
3.17. Booleans 31
3.18. Geometrics 31
3.19. IPV4 34
3.20. PostgresIP Types Examples 34
4.1. Operator Ordering (decreasing precedence) 35
4.2. Operators 36
4.3. Operators 36
4.4. Operators 37
4.5. Operators 38
4.6. Operators 38
4.7. Operators 39
5.1. SQL Functions 40
5.2. Mathematical Functions 40
5.3. Transcendental Mathematical Functions 41
5.4. SQL92 String Functions 41
5.5. String Functions 41
5.6. Date/Time Functions 42
5.7. Formatting Functions 43
5.8. Templates for date/time conversions 43
5.9. Suffixes for templates for date/time to_char() 45
5.10. Templates for to_char(*numeric*) 45
5.11. `to_char` Examples 46
5.12. Geometric Functions 47
5.13. Geometric Type Conversion Functions 47
5.14. Geometric Upgrade Functions 48
5.15. PostgresIP V4 Functions 48
13.1. Isolation Levels 80
19.1. Contents of a binary copy file 111
22.1. Supported Platforms 315
22.2. Unsupported Platforms 316
22.3. Incompatibles 317
23.1. Kerberos 322
34.1. Kerberos 355
38.1. Catalogs 416
39.1. Equivalent C Types 422
44.1. Indices 462
44.2. B-tree 463
56.1. `pgtcl` Commands 562
66.1. Signals 665

69.1. Page Layout 671
UG1.1. Time Zones 708
UG1.2. Australian Time Zones 710
DG2.1. Postgres Documentation Products 720
DG2.2. Indent Formatting for Table of Contents 726

List of Examples

19.1. Example of a circular rewrite rule combination. 134
27.1. pg_options File 334
62.1. A Simple Select 641
70.1. The Suppliers and Parts Database 676
70.2. An Inner Join 678
70.3. A Query Using Relational Algebra 679
70.4. Simple Query with Qualification 681
70.5. Aggregates 683
70.6. Aggregates 683
70.7. Having 685
70.8. Subselect 685
70.9. Union, Intersect, Except 686
70.10. Table Creation 687
70.11. Create Index 688

Summary

Postgres, developed originally in the UC Berkeley Computer Science Department, pioneered many of the object-relational concepts now becoming available in some commercial databases. It provides SQL92/SQL3 language support, transaction integrity, and type extensibility. PostgreSQL is an open-source descendant of this original Berkeley code.

Part I. User's Guide

Information for users.

Table of Contents

1. Introduction 4
    1.1. What is Postgres? 4
    1.2. A Short History of Postgres 4
        1.2.1. The Berkeley Postgres Project 4
        1.2.2. Postgres95 5
        1.2.3. PostgreSQL 6
    1.3. About This Release 6
    1.4. Resources 6
    1.5. Terminology 7
    1.6. Notation 8
    1.7. Problem Reporting Guidelines 8
        1.7.1. Identifying Bugs 9
        1.7.2. What to report 9
        1.7.3. Where to report bugs 11
    1.8. Y2K Statement 11
    1.9. Copyrights and Trademarks 12
2. SQL Syntax 13
    2.1. Key Words 13
        2.1.1. Reserved Key Words 13
        2.1.2. Non-reserved Keywords 15
    2.2. Comments 17
    2.3. Names 17
    2.4. Constants 17
        2.4.1. String Constants 17
        2.4.2. Integer Constants 17
        2.4.3. Floating Point Constants 18
        2.4.4. Constants of Postgres User-Defined Types 18
        2.4.5. Array constants 18
    2.5. Fields and Columns 19
        2.5.1. Fields 19
        2.5.2. Columns 19
    2.6. Operators 19
    2.7. Expressions 19
        2.7.1. Parameters 20
        2.7.2. Functional Expressions 20
        2.7.3. Aggregate Expressions 20
        2.7.4. Target List 21
        2.7.5. Qualification 21
        2.7.6. From List 21
3. Data Types 22
    3.1. Numeric Types 23
        3.1.1. The Serial Type 24
    3.2. Monetary Type 24
    3.3. Character Types 25
    3.4. Date/Time Types 25
        3.4.1. Date/Time Input 26
        3.4.2. Date/Time Output 29
        3.4.3. Time Zones 30
        3.4.4. Internals 30
    3.5. Boolean Type 30
    3.6. Geometric Types 31
        3.6.1. Point 31
        3.6.2. Line Segment 31
        3.6.3. Box 32
        3.6.4. Path 32

3.6.5. Polygon 33
3.6.6. Circle 33
3.7. IP Version 4 Networks and Host Addresses 33
3.7.1. CIDR 34
3.7.2. `inet` 34
4. Operators 35
4.1. Lexical Precedence 35
4.2. General Operators 36
4.3. Numerical Operators 36
4.4. Geometric Operators 37
4.5. Time Interval Operators 38
4.6. IP V4 CIDR Operators 38
4.7. IP V4 INET Operators 39
5. Functions 40
5.1. SQL Functions 40
5.2. Mathematical Functions 40
5.3. String Functions 41
5.4. Date/Time Functions 42
5.5. Formatting Functions 43
5.6. Geometric Functions 47
5.7. IP V4 Functions 48
6. Type Conversion 49
6.1. Overview 49
6.1.1. Guidelines 50
6.2. Operators 50
6.2.1. Conversion Procedure 50
6.2.2. Examples 51
6.3. Functions 52
6.3.1. Examples 53
6.4. Query Targets 54
6.4.1. Examples 54
6.5. UNION Queries 55
6.5.1. Examples 55
7. Indices and Keys 57
7.1. Keys 58
7.2. Partial Indices 59
8. Arrays 61
9. Inheritance 63

Chapter 1. Introduction

This document is the user manual for the PostgreSQL[1] database management system, originally developed at the University of California at Berkeley. PostgreSQL is based on Postgres release 4.2[2]. The Postgres project, led by Professor Michael Stonebraker, was sponsored by the Defense Advanced Research Projects Agency (DARPA), the Army Research Office (ARO), the National Science Foundation (NSF), and ESL, Inc.

1.1. What is Postgres?

Traditional relational database management systems (DBMSs) support a data model consisting of a collection of named relations, containing attributes of a specific type. In current commercial systems, possible types include floating point numbers, integers, character strings, money, and dates. It is commonly recognized that this model is inadequate for future data processing applications. The relational model successfully replaced previous models in part because of its "Spartan simplicity". However, as mentioned, this simplicity often makes the implementation of certain applications very difficult. Postgres offers substantial additional power by incorporating the following four additional basic concepts in such a way that users can easily extend the system:

classes
inheritance
types
functions

Other features provide additional power and flexibility:

constraints
triggers
rules
transaction integrity

These features put Postgres into the category of databases referred to as *object-relational*. Note that this is distinct from those referred to as *object-oriented*, which in general are not as well suited to supporting the traditional relational database languages. So, although Postgres has some object-oriented features, it is firmly in the relational database world. In fact, some commercial databases have recently incorporated features pioneered by Postgres.

1.2. A Short History of Postgres

The Object-Relational Database Management System now known as PostgreSQL (and briefly called Postgres95) is derived from the Postgres package written at Berkeley. With over a decade of development behind it, PostgreSQL is the most advanced open-source database available anywhere, offering multi-version concurrency control, supporting almost all SQL constructs (including subselects, transactions, and user-defined types and functions), and having a wide range of language bindings available (including C, C++, Java, perl, tcl, and python).

1.2.1. The Berkeley Postgres Project

Implementation of the Postgres DBMS began in 1986. The initial concepts for the system were presented in The Design of Postgres and the definition of the initial data model appeared in The Postgres Data Model . The design of the rule system at that time was described in The Design of the Postgres Rules System. The rationale and architecture of the storage manager were detailed in The Postgres Storage System .

[1] http://postgresql.org/
[2] http://s2k-ftp.CS.Berkeley.EDU:8000/postgres/postgres.html

Postgres has undergone several major releases since then. The first "demoware" system became operational in 1987 and was shown at the 1988 ACM-SIGMOD Conference. We released Version 1, described in The Implementation of Postgres , to a few external users in June 1989. In response to a critique of the first rule system (A Commentary on the Postgres Rules System), the rule system was redesigned (On Rules, Procedures, Caching and Views in Database Systems) and Version 2 was released in June 1990 with the new rule system. Version 3 appeared in 1991 and added support for multiple storage managers, an improved query executor, and a rewritten rewrite rule system. For the most part, releases until Postgres95 (see below) focused on portability and reliability.

Postgres has been used to implement many different research and production applications. These include: a financial data analysis system, a jet engine performance monitoring package, an asteroid tracking database, a medical information database, and several geographic information systems. Postgres has also been used as an educational tool at several universities. Finally, Illustra Information Technologies[3] (since merged into Informix[4]) picked up the code and commercialized it. Postgres became the primary data manager for the Sequoia 2000[5] scientific computing project in late 1992.

The size of the external user community nearly doubled during 1993. It became increasingly obvious that maintenance of the prototype code and support was taking up large amounts of time that should have been devoted to database research. In an effort to reduce this support burden, the project officially ended with Version 4.2.

1.2.2. Postgres95

In 1994, Andrew Yu[6] and Jolly Chen[7] added a SQL language interpreter to Postgres. Postgres95 was subsequently released to the Web to find its own way in the world as an open-source descendant of the original Postgres Berkeley code.

Postgres95 code was completely ANSI C and trimmed in size by 25%. Many internal changes improved performance and maintainability. Postgres95 v1.0.x ran about 30-50% faster on the Wisconsin Benchmark compared to Postgres v4.2. Apart from bug fixes, these were the major enhancements:

- The query language Postquel was replaced with SQL (implemented in the server). Subqueries were not supported until PostgreSQL (see below), but they could be imitated in Postgres95 with user-defined SQL functions. Aggregates were re-implemented. Support for the GROUP BY query clause was also added. The `libpq` interface remained available for C programs.

- In addition to the monitor program, a new program (psql) was provided for interactive SQL queries using GNU `readline`.

- A new front-end library, `libpgtcl`, supported Tcl-based clients. A sample shell, pgtclsh, provided new Tcl commands to interface tcl programs with the Postgres95 backend.

- The large object interface was overhauled. The Inversion large objects were the only mechanism for storing large objects. (The Inversion file system was removed.)

- The instance-level rule system was removed. Rules were still available as rewrite rules.

- A short tutorial introducing regular SQL features as well as those of Postgres95 was distributed with the source code.

- GNU make (instead of BSD make) was used for the build. Also, Postgres95 could be compiled with an unpatched gcc (data alignment of doubles was fixed).

[3] http://www.illustra.com/
[4] http://www.informix.com/
[5] http://www.sdsc.edu/0/Parts_Collabs/S2K/s2k_home.html
[6] mailto:ayu@informix.com
[7] http://http.cs.berkeley.edu/~jolly/

1.2.3. PostgreSQL

By 1996, it became clear that the name "Postgres95" would not stand the test of time. We chose a new name, PostgreSQL, to reflect the relationship between the original Postgres and the more recent versions with SQL capability. At the same time, we set the version numbering to start at 6.0, putting the numbers back into the sequence originally begun by the Postgres Project.

The emphasis during development of Postgres95 was on identifying and understanding existing problems in the backend code. With PostgreSQL, the emphasis has shifted to augmenting features and capabilities, although work continues in all areas.

Major enhancements in PostgreSQL include:

- Table-level locking has been replaced with multi-version concurrency control, which allows readers to continue reading consistent data during writer activity and enables hot backups from pg_dump while the database stays available for queries.
- Important backend features, including subselects, defaults, constraints, and triggers, have been implemented.
- Additional SQL92-compliant language features have been added, including primary keys, quoted identifiers, literal string type coercion, type casting, and binary and hexadecimal integer input.
- Built-in types have been improved, including new wide-range date/time types and additional geometric type support.
- Overall backend code speed has been increased by approximately 20-40%, and backend startup time has decreased 80% since v6.0 was released.

1.3. About This Release

PostgreSQL is available without cost. This manual describes version 7.0 of PostgreSQL.

We will use Postgres to mean the version distributed as PostgreSQL.

Check the Administrator's Guide for a list of currently supported machines. In general, Postgres is portable to any Unix/Posix-compatible system with full libc library support.

1.4. Resources

This manual set is organized into several parts:

Tutorial

An introduction for new users. Does not cover advanced features.

User's Guide

General information for users, including available commands and data types.

Programmer's Guide

Advanced information for application programmers. Topics include type and function extensibility, library interfaces, and application design issues.

Administrator's Guide

Installation and management information. List of supported machines.

Developer's Guide

Information for Postgres developers. This is intended for those who are contributing to the Postgres project; application development information should appear in the *Programmer's Guide*. Currently included in the *Programmer's Guide*.

Reference Manual

Detailed reference information on command syntax. Currently included in the *User's Guide*.

In addition to this manual set, there are other resources to help you with Postgres installation and use:

man pages

The man pages have general information on command syntax.

FAQs

The Frequently Asked Questions (FAQ) documents address both general issues and some platform-specific issues.

READMEs

README files are available for some contributed packages.

Web Site

The Postgres[8] web site might have some information not appearing in the distribution. There is a mhonarc catalog of mailing list traffic which is a rich resource for many topics.

Mailing Lists

The pgsql-general[9] (archive[10]) mailing list is a good place to have user questions answered. Other mailing lists are available; consult the Info Central section of the PostgreSQL web site for details.

Yourself!

Postgres is an open source product. As such, it depends on the user community for ongoing support. As you begin to use Postgres, you will rely on others for help, either through the documentation or through the mailing lists. Consider contributing your knowledge back. If you learn something which is not in the documentation, write it up and contribute it. If you add features to the code, contribute it.

Even those without a lot of experience can provide corrections and minor changes in the documentation, and that is a good way to start. The pgsql-docs[11] (archive[12]) mailing list is the place to get going.

1.5. Terminology

In the following documentation, *site* may be interpreted as the host machine on which Postgres is installed. Since it is possible to install more than one set of Postgres databases on a single host, this term more precisely denotes any particular set of installed Postgres binaries and databases.

The Postgres *superuser* is the user named `postgres` who owns the Postgres binaries and database files. As the database superuser, all protection mechanisms may be bypassed and any data accessed

[8] http://www.postgresql.org
[9] mailto:pgsql-general@postgresql.org
[10] http://www.postgresql.org/mhonarc/pgsql-general/
[11] mailto:pgsql-docs@postgresql.org
[12] http://www.PostgreSQL.ORG/mhonarc/pgsql-docs/

arbitrarily. In addition, the Postgres superuser is allowed to execute some support programs which are generally not available to all users. Note that the Postgres superuser is *not* the same as the Unix superuser (which will be referred to as *root*). The superuser should have a non-zero user identifier (*UID*) for security reasons.

The *database administrator* or DBA, is the person who is responsible for installing Postgres with mechanisms to enforce a security policy for a site. The DBA can add new users by the method described below and maintain a set of template databases for use by createdb.

The postmaster is the process that acts as a clearing-house for requests to the Postgres system. Frontend applications connect to the postmaster, which keeps tracks of any system errors and communication between the backend processes. The postmaster can take several command-line arguments to tune its behavior. However, supplying arguments is necessary only if you intend to run multiple sites or a non-default site.

The Postgres backend (the actual executable program postgres) may be executed directly from the user shell by the Postgres super-user (with the database name as an argument). However, doing this bypasses the shared buffer pool and lock table associated with a postmaster/site, therefore this is not recommended in a multiuser site.

1.6. Notation

"`...`" or `/usr/local/pgsql/` at the front of a file name is used to represent the path to the Postgres superuser's home directory.

In a command synopsis, brackets ("`[`" and "`]`") indicate an optional phrase or keyword. Anything in braces ("`{`" and "`}`") and containing vertical bars ("`|`") indicates that you must choose one.

In examples, parentheses ("`(`" and "`)`") are used to group boolean expressions. "`|`" is the boolean operator OR.

Examples will show commands executed from various accounts and programs. Commands executed from the root account will be preceeded with ">". Commands executed from the Postgres superuser account will be preceeded with "`%`", while commands executed from an unprivileged user's account will be preceeded with "`$`". SQL commands will be preceeded with "=>" or will have no leading prompt, depending on the context.

Note

At the time of writing (Postgres v7.0) the notation for flagging commands is not universally consistant throughout the documentation set. Please report problems to the Documentation Mailing List[13].

1.7. Problem Reporting Guidelines

When you encounter a problem in PostgreSQL we want to hear about it. Your bug reports are an important part in making PostgreSQL more reliable because even the utmost care cannot guarantee that every part of PostgreSQL will work on every platform under every circumstance.

The following suggestions are intended to assist you in forming bug reports that can be handled in an effective fashion. No one is required to follow them but it tends to be to everyone's advantage.

We cannot promise to fix every bug right away. If the bug is obvious, critical, or affects a lot of users, chances are good that someone will look into it. It could also happen that we tell you to update to a newer version to see if the bug happens there. Or we might decide that the bug cannot be fixed

[13] mailto:docs@postgresql.org

before some major rewrite we might be planning is done. Or perhaps it's simply too hard and there are more important things on the agenda. If you need help immediately, consider obtaining a commercial support contract.

1.7.1. Identifying Bugs

Before you ask "Is this a bug?", please read and re-read the documentation to verify that you can really do whatever it is you are trying. If it is not clear from the documentation whether you can do something or not, please report that too; it's a bug in the documentation. If it turns out that the program does something different from what the documentation says, that's a bug. That might include, but is not limited to, the following circumstances:

- A program terminates with a fatal signal or an operating system error message that would point to a problem in the program (a counterexample might be a "disk full" message, since that must be fixed outside of Postgres).
- A program produces the wrong output for any given input.
- A program refuses to accept valid input.
- A program accepts invalid input without a notice or error message.
- PostgreSQL fails to compile, build, or install according to the instructions on supported platforms.

Here "`program`" refers to any executable, not only the backend server.

Being slow or resource-hogging is not necessarily a bug. Read the documentation or ask on one of the mailing lists for help in tuning your applications. Failing to comply to SQL is not a bug unless compliance for the specific feature is explicitly claimed.

Before you continue, check on the TODO list and in the FAQ to see if your bug is already known. If you can't decode the information on the TODO list, report your problem. The least we can do is make the TODO list clearer.

1.7.2. What to report

The most important thing to remember about bug reporting is to state all the facts and only facts. Do not speculate what you think went wrong, what "it seemed to do", or which part of the program has a fault. If you are not familiar with the implementation you would probably guess wrong and not help us a bit. And even if you are, educated explanations are a great supplement to but no substitute for facts. If we are going to fix the bug we still have to see it happen for ourselves first. Reporting the bare facts is relatively straightforward (you can probably copy and paste them from the screen) but all too often important details are left out because someone thought it doesn't matter or the report would be understood anyway.

The following items should be contained in every bug report:

- The exact sequence of steps *from program startup* necessary to reproduce the problem. This should be self-contained; it is not enough to send in a bare select statement without the preceeding create table and insert statements, if the output should depend on the data in the tables. We do not have the time to decode your database schema, and if we are supposed to make up our own data we would probably miss the problem. The best format for a test case for query-language related problems is a file that can be run through the psql frontend that shows the problem. (Be sure to not have anything in your `~/.psqlrc` startup file.) You are encouraged to minimize the size of your example, but this is not absolutely necessary. If the bug is reproduceable, we'll find it either way.

  If your application uses some other client interface, such as PHP, then please try to isolate the offending queries. We probably won't set up a web server to reproduce your problem. In any case

remember to provide the exact input files, do not guess that the problem happens for "large files" or "mid-size databases", etc. since this information is too inexact to be of use.

- The output you got. Please do not say that it "didn't work" or "failed". If there is an error message, show it, even if you don't understand it. If the program terminates with an operating system error, say which. If nothing at all happens, say so. Even if the result of your test case is a program crash or otherwise obvious it might not happen on our platform. The easiest thing is to copy the output from the terminal, if possible.

  > **Note**
  >
  > In case of fatal errors, the error message provided by the client might not contain all the information available. In that case, also look at the output of the database server. If you do not keep your server output, this would be a good time to start doing so.

- The output you expected is very important to state. If you just write "This command gives me that output." or "This is not what I expected.", we might run it ourselves, scan the output, and think it looks okay and is exactly what we expected. We shouldn't have to spend the time to decode the exact semantics behind your commands. Especially refrain from merely saying that "This is not what SQL says/Oracle does." Digging out the correct behavior from SQL is not a fun undertaking, nor do we all know how all the other relational databases out there behave. (If your problem is a program crash you can obviously omit this item.)

- Any command line options and other startup options, including concerned environment variables or configuration files that you changed from the default. Again, be exact. If you are using a pre-packaged distribution that starts the database server at boot time, you should try to find out how that is done.

- Anything you did at all differently from the installation instructions.

- The PostgreSQL version. You can run the command `SELECT version();` to find out what version you are currently running. If this function does not exist, say so, then we know that your version is old enough. If you can't start up the server or a client, look into the README file in the source directory or at the name of your distribution file or package name. If your version is older than 7.0 we will almost certainly tell you to upgrade. There are tons of bug fixes in each new version, that's why we write them.

  If you run a pre-packaged version, such as RPMs, say so, including any subversion the package may have. If you are talking about a CVS snapshot, mention that, including its date and time.

- Platform information. This includes the kernel name and version, C library, processor, memory information. In most cases it is sufficient to report the vendor and version, but do not assume everyone knows what exactly "Debian" contains or that everyone runs on Pentiums. If you have installation problems information about compilers, make, etc. is also necessary.

Do not be afraid if your bug report becomes rather lengthy. That is a fact of life. It's better to report everything the first time than us having to squeeze the facts out of you. On the other hand, if your input files are huge, it is fair to ask first whether somebody is interested in looking into it.

Do not spend all your time to figure out which changes in the input make the problem go away. This will probably not help solving it. If it turns out that the bug can't be fixed right away, you will still have time to find and share your work around. Also, once again, do not waste your time guessing why the bug exists. We'll find that out soon enough.

When writing a bug report, please choose non-confusing terminology. The software package as such is called "PostgreSQL", sometimes "Postgres" for short. (Sometimes the abbreviation "Pgsql" is used but don't do that.) When you are specifically talking about the backend server, mention that, don't just say "Postgres crashes". The interactive frontend is called "psql" and is for all intends and purposes completely separate from the backend.

1.7.3. Where to report bugs

In general, send bug reports to the bug report mailing list[14]. You are invited to find a descriptive subject for your email message, perhaps parts of the error message.

Do not send bug reports to any of the user mailing lists, such as the SQL language mailing list[15] or the general topics mailing list[16]. These mailing lists are for answering user questions and their subscribers normally do not wish to receive bug reports. More importantly, they are unlikely to fix them.

Also, please do *not* send reports to the developers' mailing list[17]. This list is for discussing the development of PostgreSQL and it would be nice if we could keep the bug reports separate. We might choose to take up a discussion about your bug report on it, if the bug needs more review.

If you have a problem with the documentation, send email to the documentation mailing list[18]. Mention the document, chapter, and sections in your problem report.

If your bug is a portability problem on a non-supported platform, send mail to the porting issues mail list[19], so we (and you) can work on porting PostgreSQL to your platform.

Note

Due to the unfortunate amount of spam going around, all of the above email addresses are closed mailing lists. That is, you need to be subscribed to them in order to be allowed to post. If you simply want to send mail but do not want to receive list traffic, you can subscribe to the special pgsql-loophole mailing list, which allows you to post to all PostgreSQL mailing lists without receiving any messages. Send email to pgsql-loophole-request@postgresql.org[20] to subscribe.

1.8. Y2K Statement

Author

Written by Thomas Lockhart[21] on 1998-10-22. Updated 2000-03-31.

The PostgreSQL Global Development Team provides the Postgres software code tree as a public service, without warranty and without liability for it's behavior or performance. However, at the time of writing:

- The author of this statement, a volunteer on the Postgres support team since November, 1996, is not aware of any problems in the Postgres code base related to time transitions around Jan 1, 2000 (Y2K).
- The author of this statement is not aware of any reports of Y2K problems uncovered in regression testing or in other field use of recent or current versions of Postgres. We might have expected to hear about problems if they existed, given the installed base and the active participation of users on the support mailing lists.
- To the best of the author's knowledge, the assumptions Postgres makes about dates specified with a two-digit year are documented in the current User's Guide[22] in the chapter on data types. For two-

[14] mailto:pgsql-bugs@postgresql.org
[15] mailto:pgsql-sql@postgresql.org
[16] mailto:pgsql-general@postgresql.org
[17] mailto:pgsql-hackers@postgresql.org
[18] mailto:pgsql-docs@postgresql.org
[19] mailto:pgsql-ports@postgresql.org
[20] mailto:pgsql-loophole-request@postgresql.org
[21] mailto:lockhart@alumni.caltech.edu
[22] http://www.postgresql.org/docs/user/datatype.htm

digit years, the significant transition year is 1970, not 2000; e.g. "`70-01-01`" is interpreted as 1970-01-01, whereas "`69-01-01`" is interpreted as 2069-01-01.

- Any Y2K problems in the underlying OS related to obtaining "the current time" may propagate into apparent Y2K problems in Postgres.

Refer to The Gnu Project[23] and The Perl Institute[24] for further discussion of Y2K issues, particularly as it relates to open source, no fee software.

1.9. Copyrights and Trademarks

PostgreSQL is Copyright © 1996-2000 by PostgreSQL Inc. and is distributed under the terms of the Berkeley license.

Postgres95 is Copyright © 1994-5 by the Regents of the University of California. Permission to use, copy, modify, and distribute this software and its documentation for any purpose, without fee, and without a written agreement is hereby granted, provided that the above copyright notice and this paragraph and the following two paragraphs appear in all copies.

In no event shall the University of California be liable to any party for direct, indirect, special, incidental, or consequential damages, including lost profits, arising out of the use of this software and its documentation, even if the University of California has been advised of the possibility of such damage.

The University of California specifically disclaims any warranties, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. The software provided hereunder is on an "as-is" basis, and the University of California has no obligations to provide maintainance, support, updates, enhancements, or modifications.

All trademarks are the property of their respective owners.

[23] http://www.gnu.org/software/year2000.html
[24] http://language.perl.com/news/y2k.html

Chapter 2. SQL Syntax

A description of the general syntax of SQL.

SQL manipulates sets of data. The language is composed of various *key words*. Arithmetic and procedural expressions are allowed. We will cover these topics in this chapter; subsequent chapters will include details on data types, functions, and operators.

2.1. Key Words

SQL92 defines *key words* for the language which have specific meaning. Some key words are *reserved*, which indicates that they are restricted to appear in only certain contexts. Other key words are *not restricted*, which indicates that in certain contexts they have a specific meaning but are not otherwise constrained.

Postgres implements an extended subset of the SQL92 and SQL3 languages. Some language elements are not as restricted in this implementation as is called for in the language standards, in part due to the extensibility features of Postgres.

Information on SQL92 and SQL3 key words is derived from Date and Darwen, 1997 .

2.1.1. Reserved Key Words

SQL92 and SQL3 have *reserved key words* which are not allowed as identifiers and not allowed in any usage other than as fundamental tokens in SQL statements. Postgres has additional key words which have similar restrictions. In particular, these key words are not allowed as column or table names, though in some cases they are allowed to be column labels (i.e. in AS clauses).

Tip

Any string can be specified as an identifier if surrounded by double quotes (“like this!”). Some care is required since such an identifier will be case sensitive and will retain embedded whitespace and most other special characters.

The following are Postgres reserved words which are neither SQL92 nor SQL3 reserved words. These are allowed to be present as column labels, but not as identifiers:

```
ABORT ANALYZE
BINARY
CLUSTER CONSTRAINT COPY
DO
EXPLAIN EXTEND
LISTEN LOAD LOCK
MOVE
NEW NONE NOTIFY
OFFSET
RESET
SETOF SHOW
UNLISTEN UNTIL
VACUUM VERBOSE
```

The following are Postgres reserved words which are also SQL92 or SQL3 reserved words, and which are allowed to be present as column labels, but not as identifiers:

```
ALL ANY ASC BETWEEN BIT BOTH
CASE CAST CHAR CHARACTER CHECK COALESCE COLLATE COLUMN
 CONSTRAINT CROSS CURRENT CURRENT_DATE CURRENT_TIME
```

```
 CURRENT_TIMESTAMP CURRENT_USER
DEC DECIMAL DEFAULT DESC DISTINCT
ELSE END EXCEPT EXISTS EXTRACT
FALSE FLOAT FOR FOREIGN FROM FULL
GLOBAL GROUP
HAVING
IN INNER INTERSECT INTO IS
JOIN
LEADING LEFT LIKE LOCAL
NATURAL NCHAR NOT NULL NULLIF NUMERIC
ON OR ORDER OUTER OVERLAPS
POSITION PRECISION PRIMARY PUBLIC
REFERENCES RIGHT
SELECT SESSION_USER SOME SUBSTRING
TABLE THEN TO TRANSACTION TRIM TRUE
UNION UNIQUE USER
VARCHAR
WHEN WHERE
```

The following are Postgres reserved words which are also SQL92 or SQL3 reserved words:

```
ADD ALTER AND AS
BEGIN BY
CASCADE CLOSE COMMIT CREATE CURSOR
DECLARE DEFAULT DELETE DESC DISTINCT DROP
EXECUTE EXISTS EXTRACT
FETCH FLOAT FOR FROM FULL
GRANT
HAVING
IN INNER INSERT INTERVAL INTO IS
JOIN
LEADING LEFT LIKE LOCAL
NAMES NATIONAL NATURAL NCHAR NO NOT NULL
ON OR OUTER
PARTIAL PRIMARY PRIVILEGES PROCEDURE PUBLIC
REFERENCES REVOKE RIGHT ROLLBACK
SELECT SET SUBSTRING
TO TRAILING TRIM
UNION UNIQUE UPDATE USING
VALUES VARCHAR VARYING VIEW
WHERE WITH WORK
```

The following are SQL92 reserved key words which are not Postgres reserved key words, but which if used as function names are always translated into the function `CHAR_LENGTH`:

```
CHARACTER_LENGTH
```

The following are SQL92 or SQL3 reserved key words which are not Postgres reserved key words, but if used as type names are always translated into an alternate, native type:

```
BOOLEAN DOUBLE FLOAT INT INTEGER INTERVAL REAL SMALLINT
```

The following are not keywords of any kind, but when used in the context of a type name are translated into a native Postgres type, and when used in the context of a function name are translated into a native function:

```
DATETIME TIMESPAN
```

(translated to `TIMESTAMP` and `INTERVAL`, respectively). This feature is intended to help with transitioning to v7.0, and will be removed in the next full release (likely v7.1).

The following are either SQL92 or SQL3 reserved key words which are not key words in Postgres. These have no proscribed usage in Postgres at the time of writing (v7.0) but may become reserved key words in the future:

Note

Some of these key words represent functions in SQL92. These functions are defined in Postgres, but the parser does not consider the names to be key words and they are allowed in other contexts.

```
ALLOCATE ARE ASSERTION AT AUTHORIZATION AVG
BIT_LENGTH
CASCADED CATALOG CHAR_LENGTH CHARACTER_LENGTH COLLATION
 CONNECT CONNECTION CONTINUE CONVERT CORRESPONDING COUNT
 CURRENT_SESSION
DATE DEALLOCATE DEC DESCRIBE DESCRIPTOR
 DIAGNOSTICS DISCONNECT DOMAIN
ESCAPE EXCEPT EXCEPTION EXEC EXTERNAL
FIRST FOUND
GET GO GOTO
IDENTITY INDICATOR INPUT INTERSECT
LAST LOWER
MAX MIN MODULE
OCTET_LENGTH OPEN OUTPUT OVERLAPS
PREPARE PRESERVE
ROWS
SCHEMA SECTION SESSION SIZE SOME
 SQL SQLCODE SQLERROR SQLSTATE SUM SYSTEM_USER
TEMPORARY TRANSLATE TRANSLATION
UNKNOWN UPPER USAGE
VALUE
WHENEVER WRITE
```

2.1.2. Non-reserved Keywords

SQL92 and SQL3 have *non-reserved keywords* which have a prescribed meaning in the language but which are also allowed as identifiers. Postgres has additional keywords which allow similar unrestricted usage. In particular, these keywords are allowed as column or table names.

The following are Postgres non-reserved key words which are neither SQL92 nor SQL3 non-reserved key words:

```
ACCESS AFTER AGGREGATE
BACKWARD BEFORE
CACHE COMMENT CREATEDB CREATEUSER CYCLE
DATABASE DELIMITERS
EACH ENCODING EXCLUSIVE
FORCE FORWARD FUNCTION
HANDLER
INCREMENT INDEX INHERITS INSENSITIVE INSTEAD ISNULL
LANCOMPILER LOCATION
```

```
MAXVALUE MINVALUE MODE
NOCREATEDB NOCREATEUSER NOTHING NOTIFY NOTNULL
OIDS OPERATOR
PASSWORD PROCEDURAL
RECIPE REINDEX RENAME RETURNS ROW RULE
SEQUENCE SERIAL SHARE START STATEMENT STDIN STDOUT
TEMP TRUSTED
UNLISTEN UNTIL
VALID VERSION
```

The following are Postgres non-reserved key words which are SQL92 or SQL3 reserved key words:

```
ABSOLUTE ACTION
CONSTRAINTS
DAY DEFERRABLE DEFERRED
HOUR
IMMEDIATE INITIALLY INSENSITIVE ISOLATION
KEY
LANGUAGE LEVEL
MATCH MINUTE MONTH
NEXT
OF ONLY OPTION
PENDANT PRIOR PRIVILEGES
READ RELATIVE RESTRICT
SCROLL SECOND
TIME TIMESTAMP TIMEZONE_HOUR TIMEZONE_MINUTE TRIGGER
YEAR
ZONE
```

The following are Postgres non-reserved key words which are also either SQL92 or SQL3 non-reserved key words:

```
COMMITTED SERIALIZABLE TYPE
```

The following are either SQL92 or SQL3 non-reserved key words which are not key words of any kind in Postgres:

```
ADA
C CATALOG_NAME CHARACTER_SET_CATALOG CHARACTER_SET_NAME
 CHARACTER_SET_SCHEMA CLASS_ORIGIN COBOL COLLATION_CATALOG
 COLLATION_NAME COLLATION_SCHEMA COLUMN_NAME
 COMMAND_FUNCTION CONDITION_NUMBER
 CONNECTION_NAME CONSTRAINT_CATALOG CONSTRAINT_NAME
 CONSTRAINT_SCHEMA CURSOR_NAME
DATA DATE_TIME_INTERVAL_CODE DATE_TIME_INTERVAL_PRECISION
 DYNAMIC_FUNCTION
FORTRAN
LENGTH
MESSAGE_LENGTH MESSAGE_OCTET_LENGTH MORE MUMPS
NAME NULLABLE NUMBER
PAD PASCAL PLI
REPEATABLE RETURNED_LENGTH RETURNED_OCTET_LENGTH
 RETURNED_SQLSTATE ROW_COUNT
SCALE SCHEMA_NAME SERVER_NAME SPACE SUBCLASS_ORIGIN
TABLE_NAME
UNCOMMITTED UNNAMED
```

2.2. Comments

A *comment* is an arbitrary sequence of characters beginning with double dashes and extending to the end of the line, e.g.:

```
-- This is a standard SQL comment
```

We also support C-style block comments, e.g.:

```
/* multi
   line
   comment
 */
```

A comment beginning with "/*" extends to the first occurrence of "*/".

2.3. Names

Names in SQL must begin with a letter (`a-z`) or underscore (_). Subsequent characters in a name can be letters, digits (`0-9`), or underscores. The system uses no more than NAMEDATALEN-1 characters of a name; longer names can be written in queries, but they will be truncated. By default, NAMEDATALEN is 32 so the maximum name length is 31 (but at the time the system is built, NAMEDATALEN can be changed in src/include/postgres_ext.h).

Names containing other characters may be formed by surrounding them with double quotes ("). For example, table or column names may contain otherwise disallowed characters such as spaces, ampersands, etc. if quoted. Quoting a name also makes it case-sensitive, whereas unquoted names are always folded to lower case. For example, the names `FOO`, `foo` and `"foo"` are considered the same by Postgres, but `"Foo"` is a different name.

Double quotes can also be used to protect a name that would otherwise be taken to be an SQL keyword. For example, `IN` is a keyword but `"IN"` is a name.

2.4. Constants

There are three *implicitly typed constants* for use in Postgres: strings, integers, and floating point numbers. Constants can also be specified with explicit types, which can enable more accurate representation and more efficient handling by the backend. The implicit constants are described below; explicit constants are discussed afterwards.

2.4.1. String Constants

Strings in SQL are arbitrary sequences of ASCII characters bounded by single quotes ("'", e.g. `'This is a string'`). SQL92 allows single quotes to be embedded in strings by typing two adjacent single quotes (e.g. `'Dianne''s horse'`). In Postgres single quotes may alternatively be escaped with a backslash ("\", e.g. `'Dianne\'s horse'`). To include a backslash in a string constant, type two backslashes. Non-printing characters may also be embedded within strings by prepending them with a backslash (e.g. `'\`*`tab`*`'`).

2.4.2. Integer Constants

Integer constants in SQL are collection of ASCII digits with no decimal point. Legal values range from -2147483648 to +2147483647. This will vary depending on the operating system and host machine.

Note that larger integers can be specified for `int8` by using SQL92 string notation or Postgres type notation:

```
int8 '4000000000'  -- string style
'4000000000'::int8 -- Postgres (historical) style
```

2.4.3. Floating Point Constants

Floating point constants consist of an integer part, a decimal point, and a fraction part or scientific notation of the following format:

```
{dig}.{dig} [e [+-] {dig}]
```

where *dig* is one or more digits. You must include at least one *dig* after the period and after the [+-] if you use those options. An exponent with a missing mantissa has a mantissa of 1 inserted. There may be no extra characters embedded in the string.

Floating point constaints are of type `float8`. `float4` can be specified explicitly by using SQL92 string notation or Postgres type notation:

```
float4 '1.23'  -- string style
'1.23'::float4 -- Postgres (historical) style
```

2.4.4. Constants of Postgres User-Defined Types

A constant of an *arbitrary* type can be entered using any one of the following notations:

```
type 'string'
'string'::type
CAST 'string' AS type
```

The value inside the string is passed to the input conversion routine for the type called *type*. The result is a constant of the indicated type. The explicit typecast may be omitted if there is no ambiguity as to the type the constant must be, in which case it is automatically coerced.

2.4.5. Array constants

Array constants are arrays of any Postgres type, including other arrays, string constants, etc. The general format of an array constant is the following:

```
{val1delimval2delim}
```

where *delim* is the delimiter for the type stored in the `pg_type` class. (For built-in types, this is the comma character (","). An example of an array constant is

```
{{1,2,3},{4,5,6},{7,8,9}}
```

This constant is a two-dimensional, 3 by 3 array consisting of three sub-arrays of integers.

Individual array elements can and should be placed between quotation marks whenever possible to avoid ambiguity problems with respect to leading white space.

2.5. Fields and Columns

2.5.1. Fields

A *field* is either an attribute of a given class or one of the following:

oid

stands for the unique identifier of an instance which is added by Postgres to all instances automatically. Oids are not reused and are 32 bit quantities.

xmin

The identity of the inserting transaction.

xmax

The identity of the deleting transaction.

cmin

The command identifier within the transaction.

cmax

The identity of the deleting command.

For further information on these fields consult Stonebraker, Hanson, Hong, 1987 . Times are represented internally as instances of the `abstime` data type. Transaction and command identifiers are 32 bit quantities. Transactions are assigned sequentially starting at 512.

2.5.2. Columns

A *column* is a construct of the form:

```
instance{.composite_field}.field `['number`]'
```

instance identifies a particular class and can be thought of as standing for the instances of that class. An instance variable is either a class name, a surrogate for a class defined by means of a FROM clause, or the keyword NEW or CURRENT. NEW and CURRENT can only appear in the action portion of a rule, while other instance variables can be used in any SQL statement. *composite_field* is a field of of one of the Postgres composite types, while successive composite fields address attributes in the class(s) to which the composite field evaluates. Lastly, *field* is a normal (base type) field in the class(s) last addressed. If *field* is of type `array`, then the optional *number* designator indicates a specific element in the array. If no number is indicated, then all array elements are returned.

2.6. Operators

Any built-in system, or user-defined operator may be used in SQL. For the list of built-in and system operators consult Operators. For a list of user-defined operators consult your system administrator or run a query on the `pg_operator` class. Parentheses may be used for arbitrary grouping of operators in expressions.

2.7. Expressions

SQL92 allows *expressions* to transform data in tables. Expressions may contain operators (see Operators for more details) and functions (Functions has more information).

An expression is one of the following:

(a_expr)
constant
attribute
a_expr binary_operator a_expr
a_expr right_unary_operator
left_unary_operator a_expr
parameter
functional expression
aggregate expression

We have already discussed constants and attributes. The three kinds of operator expressions indicate respectively binary (infix), right-unary (suffix) and left-unary (prefix) operators. The following sections discuss the remaining options.

2.7.1. Parameters

A *parameter* is used to indicate a parameter in a SQL function. Typically this is used in SQL function definition statement. The form of a parameter is:

$number

For example, consider the definition of a function, `dept`, as

```
CREATE FUNCTION dept (name)
 RETURNS dept
 AS 'select * from
     dept where name=$1'
 LANGUAGE 'sql';
```

2.7.2. Functional Expressions

A *functional expression* is the name of a legal SQL function, followed by its argument list enclosed in parentheses:

function (*a_expr* [, *a_expr* ...])

For example, the following computes the square root of an employee salary:

```
sqrt(emp.salary)
```

2.7.3. Aggregate Expressions

An *aggregate expression* represents the application of an aggregate function across the rows selected by a query. An aggregate function reduces multiple inputs to a single output value, such as the sum or average of the inputs. The syntax of an aggregate expression is one of the following:

aggregate_name (*expression*)
aggregate_name (ALL *expression*)
aggregate_name (DISTINCT *expression*)
aggregate_name (*)
where *aggregate_name* is a previously defined aggregate, and *expression* is any expression that doesn't itself contain an aggregate expression.

The first form of aggregate expression invokes the aggregate across all input rows for which the given expression yields a non-null value. The second form is the same as the first, since ALL is the default. The third form invokes the aggregate for all distinct non-null values of the expression found in the input rows. The last form invokes the aggregate once for each input row regardless of null or non-null values; since no particular input value is specified, it is generally only useful for the count() aggregate.

For example, count(*) yields the total number of input rows; count(f1) yields the number of input rows in which f1 is non-null; count(distinct f1) yields the number of distinct non-null values of f1.

2.7.4. Target List

A *target list* is a parenthesized, comma-separated list of one or more elements, each of which must be of the form:

```
a_expr [ AS result_attname ]
```

where *result_attname* is the name of the attribute to be created (or an already existing attribute name in the case of update statements.) If *result_attname* is not present, then *a_expr* must contain only one attribute name which is assumed to be the name of the result field. In Postgres default naming is only used if *a_expr* is an attribute.

2.7.5. Qualification

A *qualification* consists of any number of clauses connected by the logical operators:

NOT
AND
OR
A clause is an *a_expr* that evaluates to a `boolean` over a set of instances.

2.7.6. From List

The *from list* is a comma-separated list of *from expressions*. Each "from expression" is of the form:

```
[ class_reference ] instance_variable
 {, [ class_ref ] instance_variable... }
```

where *class_reference* is of the form

```
class_name [ * ]
```

The "from expression" defines one or more instance variables to range over the class indicated in *class_reference*. One can also request the instance variable to range over all classes that are beneath the indicated class in the inheritance hierarchy by postpending the designator asterisk ("*").

Chapter 3. Data Types

Describes the built-in data types available in Postgres.

Postgres has a rich set of native data types available to users. Users may add new types to Postgres using the `CREATE TYPE` command.

In the context of data types, the following sections will discuss SQL standards compliance, porting issues, and usage. Some Postgres types correspond directly to SQL92-compatible types. In other cases, data types defined by SQL92 syntax are mapped directly into native Postgres types. Many of the built-in types have obvious external formats. However, several types are either unique to Postgres, such as open and closed paths, or have several possibilities for formats, such as the date and time types.

Table 3.1. Postgres Data Types

Postgres Type	SQL92 or SQL3 Type	Description
bool	boolean	logical boolean (true/false)
box		rectangular box in 2D plane
char(n)	character(n)	fixed-length character string
cidr		IP version 4 network or host address
circle		circle in 2D plane
date	date	calendar date without time of day
decimal	decimal(p,s)	exact numeric for p <= 9, s = 0
float4	float(*p*), *p* < 7	floating-point number with precision *p*
float8	float(*p*), 7 <= *p* < 16	floating-point number with precision *p*
inet		IP version 4 network or host address
int2	smallint	signed two-byte integer
int4	int, integer	signed 4-byte integer
int8		signed 8-byte integer
interval	interval	general-use time span
line		infinite line in 2D plane
lseg		line segment in 2D plane
money	decimal(9,2)	US-style currency
numeric	numeric(p,s)	exact numeric for p == 9, s = 0
path		open and closed geometric path in 2D plane
point		geometric point in 2D plane
polygon		closed geometric path in 2D plane
serial		unique id for indexing and cross-reference
time	time	time of day
timetz	time with time zone	time of day, including time zone
timestamp	timestamp with time zone	date/time

Postgres Type	SQL92 or SQL3 Type	Description
varchar(n)	character varying(n)	variable-length character string

Note

The `cidr` and `inet` types are designed to handle any IP type but only ipv4 is handled in the current implementation. Everything here that talks about ipv4 will apply to ipv6 in a future release.

Table 3.2. Postgres Function Constants

Postgres Function	SQL92 Constant	Description
getpgusername()	current_user	user name in current session
date('now')	current_date	date of current transaction
time('now')	current_time	time of current transaction
timestamp('now')	current_timestamp	date and time of current transaction

Postgres has features at the forefront of ORDBMS development. In addition to SQL3 conformance, substantial portions of SQL92 are also supported. Although we strive for SQL92 compliance, there are some aspects of the standard which are ill considered and which should not live through subsequent standards. Postgres will not make great efforts to conform to these features; however, these tend to apply in little-used or obsure cases, and a typical user is not likely to run into them.

Most of the input and output functions corresponding to the base types (e.g., integers and floating point numbers) do some error-checking. Some of the operators and functions (e.g., addition and multiplication) do not perform run-time error-checking in the interests of improving execution speed. On some systems, for example, the numeric operators for some data types may silently underflow or overflow.

Some of the input and output functions are not invertible. That is, the result of an output function may lose precision when compared to the original input.

Note

Floating point numbers are allowed to retain most of the intrinsic precision of the type (typically 15 digits for doubles, 6 digits for 4-byte floats). Other types with underlying floating point fields (e.g. geometric types) carry similar precision.

3.1. Numeric Types

Numeric types consist of two- and four-byte integers, four- and eight-byte floating point numbers and fixed-precision decimals.

Table 3.3. Postgres Numeric Types

Numeric Type	Storage	Description	Range
decimal	variable	User-specified precision	~8000 digits
float4	4 bytes	Variable-precision	6 decimal places
float8	8 bytes	Variable-precision	15 decimal places
int2	2 bytes	Fixed-precision	-32768 to +32767

Numeric Type	Storage	Description	Range
int4	4 bytes	Usual choice for fixed-precision	-2147483648 to +2147483647
int8	8 bytes	Very large range fixed-precision	+/- > 18 decimal places
numeric	variable	User-specified precision	no limit
serial	4 bytes	Identifer or cross-reference	0 to +2147483647

The numeric types have a full set of corresponding arithmetic operators and functions. Refer to Numerical Operators and Mathematical Functions for more information.

The `int8` type may not be available on all platforms since it relies on compiler support for this.

3.1.1. The Serial Type

The `serial` type is a special-case type constructed by Postgres from other existing components. It is typically used to create unique identifiers for table entries. In the current implementation, specifying

```
CREATE TABLE tablename (colname SERIAL);
```

is equivalent to specifying:

```
CREATE SEQUENCE tablename_colname_seq;
CREATE TABLE tablename
    (colname INT4 DEFAULT nextval('tablename_colname_seq');
CREATE UNIQUE INDEX tablename_colname_key on tablename (colname);
```

Caution

The implicit sequence created for the `serial` type will *not* be automatically removed when the table is dropped.

Implicit sequences supporting the `serial` are not automatically dropped when a table containing a serial type is dropped. So, the following commands executed in order will likely fail:

```
CREATE TABLE tablename (colname SERIAL);
DROP TABLE tablename;
CREATE TABLE tablename (colname SERIAL);
```

The sequence will remain in the database until explicitly dropped using `DROP SEQUENCE`.

3.2. Monetary Type

Obsolete Type

The `money` is now deprecated. Use `numeric` or `decimal` instead. The money type may become a locale-aware layer over the numeric type in a future release.

The `money` type supports US-style currency with fixed decimal point representation. If Postgres is compiled with USE_LOCALE then the money type should use the monetary conventions defined for *locale(7)*.

Table 3.4. Postgres Monetary Types

Monetary Type	Storage	Description	Range
money	4 bytes	Fixed-precision	-21474836.48 to +21474836.47

`numeric` will replace the money type, and should be preferred.

3.3. Character Types

SQL92 defines two primary character types: `char` and `varchar`. Postgres supports these types, in addition to the more general `text` type, which unlike `varchar` does not require an explicit declared upper limit on the size of the field.

Table 3.5. Postgres Character Types

Character Type	Storage	Recommendation	Description
char	1 byte	SQL92-compatible	Single character
char(n)	(4+n) bytes	SQL92-compatible	Fixed-length blank padded
text	(4+x) bytes	Best choice	Variable-length
varchar(n)	(4+n) bytes	SQL92-compatible	Variable-length with limit

There is one other fixed-length character type in Postgres. The `name` type only has one purpose and that is for storage of internal catalog names. It is not intended for use by the general user. Its length is currently defined as 32 bytes (31 characters plus terminator) but should be reference using NAMEDATALEN. The length is set at compile time (and is therefore adjustable for special uses); the default maximum length may change in a future release.

Table 3.6. Postgres Specialty Character Type

Character Type	Storage	Description
name	32 bytes	Thirty-one character internal type

3.4. Date/Time Types

Postgres supports the full set of SQL date and time types.

Table 3.7. Postgres Date/Time Types

Type	Description	Storage	Earliest	Latest	Resolution
`timestamp`	both date and time	8 bytes	4713 BC	AD 1465001	1 microsec / 14 digits
`timestamp with time zone`	date and time with time zone	8 bytes	1903 AD	2037 AD	1 microsec / 14 digits
`interval`	for time intervals	12 bytes	-178000000 years	178000000 years	1 mircosecond
`date`	dates only	4 bytes	4713 BC	32767 AD	1 day
`time`	times of day only	4 bytes	00:00:00.00	23:59:59.99	1 microsecond

Type	Description	Storage	Earliest	Latest	Resolution
`time with time zone`	times of day only	4 bytes	00:00:00.00+12	23:59:59.99-12	1 microsecond

Note

To ensure compatibility to earlier versions of Postgres we also continue to provide `datetime` (equivalent to `timestamp`) and `timespan` (equivalent to `interval`), however support for these is now restricted to having an implicit translation to `timestamp` and `interval`. The types `abstime` and `reltime` are lower precision types which are used internally. You are discouraged from using any of these types in new applications and are encouraged to move any old ones over when appropriate. Any or all of these internal types might disappear in a future release.

3.4.1. Date/Time Input

Date and time input is accepted in almost any reasonable format, including ISO-8601, SQL-compatible, traditional Postgres, and others. The ordering of month and day in date input can be ambiguous, therefore a setting exists to specify how it should be interpreted in ambiguous cases. The command `SET DateStyle TO 'US'` or `SET DateStyle TO 'NonEuropean'` specifies the variant "month before day", the command `SET DateStyle TO 'European'` sets the variant "day before month". The `ISO` style is the default but this default can be changed at compile time or at run time.

See Date/Time Support for the exact parsing rules of date/time input and for the recognized time zones.

Remember that any date or time input needs to be enclosed into single quotes, like text strings.

3.4.1.1. date

The following are possible inputs for the `date` type.

Table 3.8. Postgres Date Input

Example	Description
January 8, 1999	Unambiguous
1999-01-08	ISO-8601 format, preferred
1/8/1999	US; read as August 1 in European mode
8/1/1999	European; read as August 1 in US mode
1/18/1999	US; read as January 18 in any mode
19990108	ISO-8601 year, month, day
990108	ISO-8601 year, month, day
1999.008	Year and day of year
99008	Year and day of year
January 8, 99 BC	Year 99 before the Common Era

Table 3.9. Postgres Month Abbreviations

Month	Abbreviations
April	Apr
August	Aug

Month	Abbreviations
December	Dec
February	Feb
January	Jan
July	Jul
June	Jun
March	Mar
November	Nov
October	Oct
September	Sep, Sept

Note

The month `May` has no explicit abbreviation, for obvious reasons.

Table 3.10. Postgres Day of Week Abbreviations

Day	Abbreviation
Sunday	Sun
Monday	Mon
Tuesday	Tue, Tues
Wednesday	Wed, Weds
Thursday	Thu, Thur, Thurs
Friday	Fri
Saturday	Sat

3.4.1.2. time

The following are valid `time` inputs.

Table 3.11. Postgres Time Input

Example	Description
04:05:06.789	ISO-8601
04:05:06	ISO-8601
04:05	ISO-8601
040506	ISO-8601
04:05 AM	Same as 04:05; AM does not affect value
04:05 PM	Same as 16:05; input hour must be <= 12
z	Same as 00:00:00
zulu	Same as 00:00:00
allballs	Same as 00:00:00

3.4.1.3. time with time zone

This type is defined by SQL92, but the definition exhibits fundamental deficiencies which renders the type nearly useless. In most cases, a combination of `date`, `time`, and `timestamp` should provide a complete range of date/time functionality required by any application.

`time with time zone` accepts all input also legal for the `time` type, appended with a legal time zone, as follows:

Table 3.12. Postgres Time With Time Zone Input

Example	Description
04:05:06.789-8	ISO-8601
04:05:06-08:00	ISO-8601
04:05-08:00	ISO-8601
040506-08	ISO-8601

Refer to Postgres Time Zone Input for more examples of time zones.

3.4.1.4. timestamp

Valid input for the `timestamp` type consists of a concatenation of a date and a time, followed by an optional `AD` or `BC`, followed by an optional time zone. (See below.) Thus

```
1999-01-08 04:05:06 -8:00
```

is a valid `timestamp` value, which is ISO-compliant. In addition, the wide-spread format

```
January 8 04:05:06 1999 PST
```

is supported.

Table 3.13. Postgres Time Zone Input

Time Zone	Description
PST	Pacific Standard Time
-8:00	ISO-8601 offset for PST
-800	ISO-8601 offset for PST
-8	ISO-8601 offset for PST

3.4.1.5. interval

`intervals` can be specified with the following syntax:

```
  Quantity Unit [Quantity Unit...] [Direction]
@ Quantity Unit [Direction]
```

where: `Quantity` is ..., `-1`, `0`, `1`, `2`, ...; `Unit` is `second`, `minute`, `hour`, `day`, `week`, `month`, `year`, `decade`, `century`, `millennium`, or abbreviations or plurals of these units; `Direction` can be `ago` or empty.

3.4.1.6. Special values

The following SQL-compatible functions can be used as date or time input for the corresponding datatype: `CURRENT_DATE`, `CURRENT_TIME`, `CURRENT_TIMESTAMP`.

Postgres also supports several special constants for convenience.

Table 3.14. Postgres Special Date/Time Constants

Constant	Description
current	Current transaction time, deferred
epoch	1970-01-01 00:00:00+00 (Unix system time zero)
infinity	Later than other valid times
-infinity	Earlier than other valid times
invalid	Illegal entry
now	Current transaction time
today	Midnight today
tomorrow	Midnight tomorrow
yesterday	Midnight yesterday

`'now'` is resolved when the value is inserted, `'current'` is resolved everytime the value is retrieved. So you probably want to use `'now'` in most applications. (Of course you *really* want to use `CURRENT_TIMESTAMP`, which is equivalent to `'now'`.)

3.4.2. Date/Time Output

Output formats can be set to one of the four styles ISO-8601, SQL (Ingres), traditional Postgres, and German, using the `SET DateStyle`. The default is the ISO format.

Table 3.15. Postgres Date/Time Output Styles

Style Specification	Description	Example
'ISO'	ISO-8601 standard	1997-12-17 07:37:16-08
'SQL'	Traditional style	12/17/1997 07:37:16.00 PST
'Postgres'	Original style	Wed Dec 17 07:37:16 1997 PST
'German'	Regional style	17.12.1997 07:37:16.00 PST

The output of the `date` and `time` styles is of course only the date or time part in accordance with the above examples.

The SQL style has European and non-European (US) variants, which determines whether month follows day or vica versa. (See also above at Date/Time Input, how this setting affects interpretation of input values.)

Table 3.16. Postgres Date Order Conventions

Style Specification	Description	Example
European	*day*/*month*/*year*	17/12/1997 15:37:16.00 MET
US	*month*/*day*/*year*	12/17/1997 07:37:16.00 PST

`interval` output looks like the input format, except that units like `week` or `century` are converted to years and days. In ISO mode the output looks like

```
[ Quantity Units [ ... ] ] [ Days ] Hours:Minutes [ ago ]
```

There are several ways to affect the appearance of date/time types:

- The `PGDATESTYLE` environment variable used by the backend directly on postmaster startup.
- The `PGDATESTYLE` environment variable used by the frontend libpq on session startup.

- `SET DATESTYLE` SQL command.

3.4.3. Time Zones

Postgres endeavors to be compatible with SQL92 definitions for typical usage. However, the SQL92 standard has an odd mix of date and time types and capabilities. Two obvious problems are:

- Although the `date` type does not have an associated time zone, the `time` type can or does.
- The default time zone is specified as a constant integer offset from GMT/UTC.

Time zones in the real world can have no meaning unless associated with a date as well as a time since the offset may vary through the year with daylight savings time boundaries.

To address these difficulties, Postgres associates time zones only with date and time types which contain both date and time, and assumes local time for any type containing only date or time. Further, time zone support is derived from the underlying operating system time zone capabilities, and hence can handle daylight savings time and other expected behavior.

Postgres obtains time zone support from the underlying operating system for dates between 1902 and 2038 (near the typical date limits for Unix-style systems). Outside of this range, all dates are assumed to be specified and used in Universal Coordinated Time (UTC).

All dates and times are stored internally in Universal UTC, alternately known as Greenwich Mean Time (GMT). Times are converted to local time on the database server before being sent to the client frontend, hence by default are in the server time zone.

There are several ways to affect the time zone behavior:

- The TZ environment variable used by the backend directly on postmaster startup as the default time zone.
- The PGTZ environment variable set at the client used by libpq to send time zone information to the backend upon connection.
- The SQL command `SET TIME ZONE` sets the time zone for the session.

If an invalid time zone is specified, the time zone becomes GMT (on most systems anyway).

Note

If the compiler option USE_AUSTRALIAN_RULES is set then `EST` refers to Australia Eastern Std Time, which has an offset of +10:00 hours from UTC.

3.4.4. Internals

Postgres uses Julian dates for all date/time calculations. They have the nice property of correctly predicting/calculating any date more recent than 4713BC to far into the future, using the assumption that the length of the year is 365.2425 days.

Date conventions before the 19th century make for interesting reading, but are not consistant enough to warrant coding into a date/time handler.

3.5. Boolean Type

Postgres supports `bool` as the SQL3 boolean type. `bool` can have one of only two states: 'true' or 'false'. A third state, 'unknown', is not implemented and is not suggested in SQL3; NULL is an effective substitute. `bool` can be used in any boolean expression, and boolean expressions always evaluate to a result compatible with this type.

`bool` uses 1 byte of storage.

Table 3.17. Postgres Boolean Type

State	Output	Input
True	't'	TRUE, 't', 'true', 'y', 'yes', '1'
False	'f'	FALSE, 'f', 'false', 'n', 'no', '0'

3.6. Geometric Types

Geometric types represent two-dimensional spatial objects. The most fundamental type, the point, forms the basis for all of the other types.

Table 3.18. Postgres Geometric Types

Geometric Type	Storage	Representation	Description
point	16 bytes	(x,y)	Point in space
line	32 bytes	((x1,y1),(x2,y2))	Infinite line
lseg	32 bytes	((x1,y1),(x2,y2))	Finite line segment
box	32 bytes	((x1,y1),(x2,y2))	Rectangular box
path	4+32n bytes	((x1,y1),...)	Closed path (similar to polygon)
path	4+32n bytes	[(x1,y1),...]	Open path
polygon	4+32n bytes	((x1,y1),...)	Polygon (similar to closed path)
circle	24 bytes	<(x,y),r>	Circle (center and radius)

A rich set of functions and operators is available to perform various geometric operations such as scaling, translation, rotation, and determining intersections.

3.6.1. Point

Points are the fundamental two-dimensional building block for geometric types.

`point` is specified using the following syntax:

```
( x , y )
  x , y
```

where the arguments are

x

The x-axis coordinate as a floating point number.

y

The y-axis coordinate as a floating point number.

3.6.2. Line Segment

Line segments (`lseg`) are represented by pairs of points.

`lseg` is specified using the following syntax:

```
( ( x1 , y1 ) , ( x2 , y2 ) )
  ( x1 , y1 ) , ( x2 , y2 )
    x1 , y1   ,   x2 , y2
```

where the arguments are

(*x1*,*y1*)
(*x2*,*y2*)

The endpoints of the line segment.

3.6.3. Box

Boxes are represented by pairs of points which are opposite corners of the box.

`box` is specified using the following syntax:

```
( ( x1 , y1 ) , ( x2 , y2 ) )
  ( x1 , y1 ) , ( x2 , y2 )
    x1 , y1   ,   x2 , y2
```

where the arguments are

(*x1*,*y1*)
(*x2*,*y2*)

Opposite corners of the box.

Boxes are output using the first syntax. The corners are reordered on input to store the lower left corner first and the upper right corner last. Other corners of the box can be entered, but the lower left and upper right corners are determined from the input and stored.

3.6.4. Path

Paths are represented by connected sets of points. Paths can be "open", where the first and last points in the set are not connected, and "closed", where the first and last point are connected. Functions `popen(p)` and `pclose(p)` are supplied to force a path to be open or closed, and functions `isopen(p)` and `isclosed(p)` are supplied to test for either type in a query.

`path` is specified using the following syntax:

```
( ( x1 , y1 ) , ... , ( xn , yn ) )
[ ( x1 , y1 ) , ... , ( xn , yn ) ]
  ( x1 , y1 ) , ... , ( xn , yn )
  ( x1 , y1   , ... ,   xn , yn )
    x1 , y1   , ... ,   xn , yn
```

where the arguments are

(*x*,*y*)

Endpoints of the line segments comprising the path. A leading square bracket ("[") indicates an open path, while a leading parenthesis ("(") indicates a closed path.

Paths are output using the first syntax. Note that Postgres versions prior to v6.1 used a format for paths which had a single leading parenthesis, a "closed" flag, an integer count of the number of points, then

the list of points followed by a closing parenthesis. The built-in function `upgradepath` is supplied to convert paths dumped and reloaded from pre-v6.1 databases.

3.6.5. Polygon

Polygons are represented by sets of points. Polygons should probably be considered equivalent to closed paths, but are stored differently and have their own set of support routines.

`polygon` is specified using the following syntax:

```
( ( x1 , y1 ) , ... , ( xn , yn ) )
  ( x1 , y1 ) , ... , ( xn , yn )
  ( x1 , y1   , ... ,   xn , yn )
    x1 , y1   , ... ,   xn , yn
```

where the arguments are

(*x*,*y*)

Endpoints of the line segments comprising the boundary of the polygon.

Polygons are output using the first syntax. Note that Postgres versions prior to v6.1 used a format for polygons which had a single leading parenthesis, the list of x-axis coordinates, the list of y-axis coordinates, followed by a closing parenthesis. The built-in function `upgradepoly` is supplied to convert polygons dumped and reloaded from pre-v6.1 databases.

3.6.6. Circle

Circles are represented by a center point and a radius.

`circle` is specified using the following syntax:

```
< ( x , y ) , r >
( ( x , y ) , r )
  ( x , y ) , r
    x , y   , r
```

where the arguments are

(*x*,*y*)

Center of the circle.

r

Radius of the circle.

Circles are output using the first syntax.

3.7. IP Version 4 Networks and Host Addresses

The `cidr` type stores networks specified in CIDR (Classless Inter-Domain Routing) notation. The `inet` type stores hosts and networks in CIDR notation using a simple variation in representation to represent simple host TCP/IP addresses.

Table 3.19. PostgresIP Version 4 Types

IPV4 Type	Storage	Description	Range
cidr	variable	CIDR networks	Valid IPV4 CIDR blocks
inet	variable	nets and hosts	Valid IPV4 CIDR blocks

3.7.1. CIDR

The `cidr` type holds a CIDR network. The format for specifying classless networks is *`x.x.x.x/y`* where *`x.x.x.x`* is the network and */`y`* is the number of bits in the netmask. If */`y`* omitted, it is calculated using assumptions from the older classfull naming system except that it is extended to include at least all of the octets in the input.

Here are some examples:

Table 3.20. PostgresIP Types Examples

CIDR Input	CIDR Displayed
192.168.1	192.168.1/24
192.168	192.168.0/24
128.1	128.1/16
128	128.0/16
128.1.2	128.1.2/24
10.1.2	10.1.2/24
10.1	10.1/16
10	10/8

3.7.2. `inet`

The `inet` type is designed to hold, in one field, all of the information about a host including the CIDR-style subnet that it is in. Note that if you want to store proper CIDR networks, you should use the `cidr` type. The `inet` type is similar to the `cidr` type except that the bits in the host part can be non-zero. Functions exist to extract the various elements of the field.

The input format for this function is *`x.x.x.x/y`* where *`x.x.x.x`* is an internet host and *`y`* is the number of bits in the netmask. If the */`y`* part is left off, it is treated as `/32`. On output, the */`y`* part is not printed if it is `/32`. This allows the type to be used as a straight host type by just leaving off the bits part.

Chapter 4. Operators

Describes the built-in operators available in Postgres.

Postgres provides a large number of built-in operators on system types. These operators are declared in the system catalog `pg_operator`. Every entry in `pg_operator` includes the name of the procedure that implements the operator and the class OIDs of the input and output types.

To view all variations of the "`||`" string concatenation operator, try

```
    SELECT oprleft, oprright, oprresult, oprcode
    FROM pg_operator WHERE oprname = '||';

oprleft|oprright|oprresult|oprcode
-------+--------+---------+-------
     25|      25|       25|textcat
   1042|    1042|     1042|textcat
   1043|    1043|     1043|textcat
(3 rows)
```

Users may invoke operators using the operator name, as in:

```
select * from emp where salary < 40000;
```

Alternatively, users may call the functions that implement the operators directly. In this case, the query above would be expressed as:

```
select * from emp where int4lt(salary, 40000);
```

psql has a command (`\dd`) to show these operators.

4.1. Lexical Precedence

Operators have a precedence which is currently hardcoded into the parser. Most operators have the same precedence and are left-associative. This may lead to non-intuitive behavior; for example the boolean operators "<" and ">" have a different precedence than the boolean operators "<=" and ">=".

Table 4.1. Operator Ordering (decreasing precedence)

Element	Precedence	Description
UNION	left	SQL select construct
::		Postgres typecasting
[ ]	left	array delimiters
.	left	table/column delimiter
-	right	unary minus
:	right	exponentiation
|	left	start of interval
* / %	left	multiplication, division, modulo
+ -	left	addition, subtraction
IS		test for TRUE, FALSE, NULL
ISNULL		test for NULL

Element	Precedence	Description
NOTNULL		test for NOT NULL
(all other operators)		native and user-defined
IN		set membership
BETWEEN		containment
OVERLAPS		time interval overlap
LIKE		string pattern matching
< >		inequality
=	right	equality
NOT	right	logical negation
AND	left	logical intersection
OR	left	logical union

4.2. General Operators

The operators listed here are defined for a number of native data types, ranging from numeric types to data/time types.

Table 4.2. Postgres Operators

Operator	Description	Usage
<	Less than?	1 < 2
<=	Less than or equal to?	1 <= 2
<>	Not equal?	1 <> 2
=	Equal?	1 = 1
>	Greater than?	2 > 1
>=	Greater than or equal to?	2 >= 1
||	Concatenate strings	'Postgre' || 'SQL'
!!=	NOT IN	3 !!= i
~~	LIKE	'scrappy,marc,hermit' ~~ '%scrappy%'
!~~	NOT LIKE	'bruce' !~~ '%al%'
~	Match (regex), case sensitive	'thomas' ~ '.*thomas.*'
~*	Match (regex), case insensitive	'thomas' ~* '.*Thomas.*'
!~	Does not match (regex), case sensitive	'thomas' !~ '.*Thomas.*'
!~*	Does not match (regex), case insensitive	'thomas' !~* '.*vadim.*'

4.3. Numerical Operators

Table 4.3. Postgres Numerical Operators

Operator	Description	Usage
!	Factorial	3 !
!!	Factorial (left operator)	!! 3

Operator	Description	Usage
%	Modulo	5 % 4
%	Truncate	% 4.5
*	Multiplication	2 * 3
+	Addition	2 + 3
-	Subtraction	2 - 3
/	Division	4 / 2
:	Natural Exponentiation	: 3.0
@	Absolute value	@ -5.0
^	Exponentiation	2.0 ^ 3.0
|/	Square root	|/ 25.0
||/	Cube root	||/ 27.0

Note

Two operators, ":" and ";", are now deprecated and will be removed in the next release. Use the equivalent functions exp() and ln() instead.

4.4. Geometric Operators

Table 4.4. Postgres Geometric Operators

Operator	Description	Usage
+	Translation	'((0,0),(1,1))'::box + '(2.0,0)'::point
-	Translation	'((0,0),(1,1))'::box - '(2.0,0)'::point
*	Scaling/rotation	'((0,0),(1,1))'::box * '(2.0,0)'::point
/	Scaling/rotation	'((0,0),(2,2))'::box / '(2.0,0)'::point
#	Intersection	'((1,-1),(-1,1))' # '((1,1),(-1,-1))'
#	Number of points in polygon	# '((1,0),(0,1),(-1,0))'
##	Point of closest proximity	'(0,0)'::point ## '((2,0),(0,2))'::lseg
&&	Overlaps?	'((0,0),(1,1))'::box && '((0,0),(2,2))'::box
&<	Overlaps to left?	'((0,0),(1,1))'::box &< '((0,0),(2,2))'::box
&>	Overlaps to right?	'((0,0),(3,3))'::box &> '((0,0),(2,2))'::box
<->	Distance between	'((0,0),1)'::circle <-> '((5,0),1)'::circle
<<	Left of?	'((0,0),1)'::circle << '((5,0),1)'::circle
<^	Is below?	'((0,0),1)'::circle <^ '((0,5),1)'::circle

Operator	Description	Usage
>>	Is right of?	'((5,0),1)'::circle >> '((0,0),1)'::circle
>^	Is above?	'((0,5),1)'::circle >^ '((0,0),1)'::circle
?#	Intersects or overlaps	'((-1,0),(1,0))'::lseg ?# '((-2,-2),(2,2))'::box;
?-	Is horizontal?	'(1,0)'::point ?- '(0,0)'::point
?-|	Is perpendicular?	'((0,0),(0,1))'::lseg ?-| '((0,0),(1,0))'::lseg
@-@	Length or circumference	@-@ '((0,0),(1,0))'::path
?|	Is vertical?	'(0,1)'::point ?| '(0,0)'::point
?||	Is parallel?	'((-1,0),(1,0))'::lseg ?|| '((-1,2),(1,2))'::lseg
@	Contained or on	'(1,1)'::point @ '((0,0),2)'::circle
@@	Center of	@@ '((0,0),10)'::circle
~=	Same as	'((0,0),(1,1))'::polygon ~= '((1,1),(0,0))'::polygon

4.5. Time Interval Operators

The time interval data type `tinterval` is a legacy from the original date/time types and is not as well supported as the more modern types. There are several operators for this type.

Table 4.5. Postgres Time Interval Operators

Operator	Description	Usage
#<	Interval less than?	
#<=	Interval less than or equal to?	
#<>	Interval not equal?	
#=	Interval equal?	
#>	Interval greater than?	
#>=	Interval greater than or equal to?	
<#>	Convert to time interval	
<<	Interval less than?	
|	Start of interval	
~=	Same as	
<?>	Time inside interval?	

4.6. IP V4 CIDR Operators

Table 4.6. PostgresIP V4 CIDR Operators

Operator	Description	Usage
<	Less than	'192.168.1.5'::cidr < '192.168.1.6'::cidr

Operator	Description	Usage
<=	Less than or equal	'192.168.1.5'::cidr <= '192.168.1.5'::cidr
=	Equals	'192.168.1.5'::cidr = '192.168.1.5'::cidr
>=	Greater or equal	'192.168.1.5'::cidr >= '192.168.1.5'::cidr
>	Greater	'192.168.1.5'::cidr > '192.168.1.4'::cidr
<>	Not equal	'192.168.1.5'::cidr <> '192.168.1.4'::cidr
<<	is contained within	'192.168.1.5'::cidr << '192.168.1/24'::cidr
<<=	is contained within or equals	'192.168.1/24'::cidr <<= '192.168.1/24'::cidr
>>	contains	'192.168.1/24'::cidr >> '192.168.1.5'::cidr
>>=	contains or equals	'192.168.1/24'::cidr >>= '192.168.1/24'::cidr

4.7. IP V4 INET Operators

Table 4.7. PostgresIP V4 INET Operators

Operator	Description	Usage
<	Less than	'192.168.1.5'::inet < '192.168.1.6'::inet
<=	Less than or equal	'192.168.1.5'::inet <= '192.168.1.5'::inet
=	Equals	'192.168.1.5'::inet = '192.168.1.5'::inet
>=	Greater or equal	'192.168.1.5'::inet >= '192.168.1.5'::inet
>	Greater	'192.168.1.5'::inet > '192.168.1.4'::inet
<>	Not equal	'192.168.1.5'::inet <> '192.168.1.4'::inet
<<	is contained within	'192.168.1.5'::inet << '192.168.1/24'::inet
<<=	is contained within or equals	'192.168.1/24'::inet <<= '192.168.1/24'::inet
>>	contains	'192.168.1/24'::inet >> '192.168.1.5'::inet
>>=	contains or equals	'192.168.1/24'::inet >>= '192.168.1/24'::inet

Chapter 5. Functions

Describes the built-in functions available in Postgres.

Many data types have functions available for conversion to other related types. In addition, there are some type-specific functions. Some functions are also available through operators and may be documented as operators only.

5.1. SQL Functions

SQL functions are constructs defined by the SQL92 standard which have function-like syntax but which can not be implemented as simple functions.

Table 5.1. SQL Functions

Function	Returns	Description	Example
COALESCE(*list*)	non-NULL	return first non-NULL value in list	COALESCE(rle, c2 + 5, 0)
NULLIF(*input*,*value*)	*input* or NULL	return NULL if *input* = *value*, else *input*	NULLIF(c1, 'N/A')
CASE WHEN *expr* THEN *expr* [...] ELSE *expr* END	*expr*	return expression for first true WHEN clause	CASE WHEN c1 = 1 THEN 'match' ELSE 'no match' END

5.2. Mathematical Functions

Table 5.2. Mathematical Functions

Function	Returns	Description	Example
abs(float8)	float8	absolute value	abs(-17.4)
degrees(float8)	float8	radians to degrees	degrees(0.5)
exp(float8)	float8	raise e to the specified exponent	exp(2.0)
ln(float8)	float8	natural logarithm	ln(2.0)
log(float8)	float8	base 10 logarithm	log(2.0)
pi()	float8	fundamental constant	pi()
pow(float8,float8)	float8	raise a number to the specified exponent	pow(2.0, 16.0)
radians(float8)	float8	degrees to radians	radians(45.0)
round(float8)	float8	round to nearest integer	round(42.4)
sqrt(float8)	float8	square root	sqrt(2.0)
cbrt(float8)	float8	cube root	cbrt(27.0)
trunc(float8)	float8	truncate (towards zero)	trunc(42.4)
float(int)	float8	convert integer to floating point	float(2)
float4(int)	float4	convert integer to floating point	float4(2)
integer(float)	int	convert floating point to integer	integer(2.0)

Most of the functions listed for FLOAT8 are also available for type NUMERIC.

Table 5.3. Transcendental Mathematical Functions

Function	Returns	Description	Example
acos(float8)	float8	arccosine	acos(10.0)
asin(float8)	float8	arcsine	asin(10.0)
atan(float8)	float8	arctangent	atan(10.0)
atan2(float8,float8)	float8	arctangent	atan2(10.0,20.0)
cos(float8)	float8	cosine	cos(0.4)
cot(float8)	float8	cotangent	cot(20.0)
sin(float8)	float8	sine	cos(0.4)
tan(float8)	float8	tangent	tan(0.4)

5.3. String Functions

SQL92 defines string functions with specific syntax. Some of these are implemented using other Postgres functions. The supported string types for SQL92 are `char`, `varchar`, and `text`.

Table 5.4. SQL92 String Functions

Function	Returns	Description	Example
char_length(string)	int4	length of string	char_length('jose')
character_length(string)	int4	length of string	char_length('jose')
lower(string)	string	convert string to lower case	lower('TOM')
octet_length(string)	int4	storage length of string	octet_length('jose')
position(string in string)	int4	location of specified substring	position('o' in 'Tom')
substring(string [from int] [for int])	string	extract specified substring	substring('Tom' from 2 for 2)
trim([leading|trailing|both] [string] from string)	string	trim characters from string	trim(both 'x' from 'xTomx')
upper(text)	text	convert text to upper case	upper('tom')

Many additional string functions are available for text, varchar(), and char() types. Some are used internally to implement the SQL92 string functions listed above.

Table 5.5. String Functions

Function	Returns	Description	Example
char(text)	char	convert text to char type	char('text string')
char(varchar)	char	convert varchar to char type	char(varchar 'varchar string')
initcap(text)	text	first letter of each word to upper case	initcap('thomas')
lpad(text,int,text)	text	left pad string to specified length	lpad('hi',4,'??')

Function	Returns	Description	Example
ltrim(text,text)	text	left trim characters from text	ltrim('xxxxtrim','x')
textpos(text,text)	text	locate specified substring	position('high','ig')
rpad(text,int,text)	text	right pad string to specified length	rpad('hi',4,'x')
rtrim(text,text)	text	right trim characters from text	rtrim('trimxxxx','x')
substr(text,int[,int])	text	extract specified substring	substr('hi there',3,5)
text(char)	text	convert char to text type	text('char string')
text(varchar)	text	convert varchar to text type	text(varchar 'varchar string')
translate(text,from,to)	text	convert character in string	translate('12345', '1', 'a')
varchar(char)	varchar	convert char to varchar type	varchar('char string')
varchar(text)	varchar	convert text to varchar type	varchar('text string')

Most functions explicitly defined for text will work for char() and varchar() arguments.

5.4. Date/Time Functions

The date/time functions provide a powerful set of tools for manipulating various date/time types.

Table 5.6. Date/Time Functions

Function	Returns	Description	Example
abstime(timestamp)	abstime	convert to abstime	abstime(timestamp 'now')
age(timestamp)	interval	preserve months and years	age(timestamp '1957-06-13')
age(timestamp,timestamp)	interval	preserve months and years	age('now', timestamp '1957-06-13')
date_part(text,timestamp)	float8	portion of date	date_part('dow',timestamp 'now')
date_part(text,interval)	float8	portion of time	date_part('hour',interval '4 hrs 3 mins')
date_trunc(text,timestamp)	timestamp	truncate date	date_trunc('month',abstime 'now')
interval(reltime)	interval	convert to interval	interval(reltime '4 hours')
isfinite(timestamp)	bool	a finite time?	isfinite(timestamp 'now')
isfinite(interval)	bool	a finite time?	isfinite(interval '4 hrs')
reltime(interval)	reltime	convert to reltime	reltime(interval '4 hrs')
timestamp(date)	timestamp	convert to timestamp	timestamp(date 'today')

Function	Returns	Description	Example
timestamp(date,time)	timestamp	convert to timestamp	timestamp(timestamp '1998-02-24',time '23:07');
to_char(timestamp,text)	text	convert to string	to_char(timestamp '1998-02-24','DD');

For the `date_part` and `date_trunc` functions, arguments can be `` `year' ``, `` `month' ``, `` `day' ``, `` `hour' ``, `` `minute' ``, and `` `second' ``, as well as the more specialized quantities `` `decade' ``, `` `century' ``, `` `millennium' ``, `` `millisecond' ``, and `` `microsecond' ``. `date_part` allows `` `dow' `` to return day of week, `` `week' `` to return the ISO-defined week of year, and `` `epoch' `` to return seconds since 1970 (for `timestamp`) or `` `epoch' `` to return total elapsed seconds (for `interval`).

5.5. Formatting Functions

Author

Written by Karel Zak[1] on 2000-01-24.

The Postgres formatting functions provide a powerful set of tools for converting various datetypes (date/time, int, float, numeric) to formatted strings and for converting from formatted strings to specific datetypes.

Note

The second argument for all formatting functions is a template to be used for the conversion.

Table 5.7. Formatting Functions

Function	Returns	Description	Example
to_char(timestamp, text)	text	convert timestamp to string	to_char(timestamp 'now','HH12:MI:SS')
to_char(int, text)	text	convert int4/int8 to string	to_char(125, '999')
to_char(float, text)	text	convert float4/float8 to string	to_char(125.8, '999D9')
to_char(numeric, text)	text	convert numeric to string	to_char(numeric '-125.8', '999D99S')
to_date(text, text)	date	convert string to date	to_date('05 Dec 2000', 'DD Mon YYYY')
to_timestamp(text, text)	date	convert string to timestamp	to_timestamp('05 Dec 2000', 'DD Mon YYYY')
to_number(text, text)	numeric	convert string to numeric	to_number('12,454.8-', '99G999D9S')

Table 5.8. Templates for date/time conversions

Template	Description
HH	hour of day (01-12)

[1] mailto:zakkr@zf.jcu.cz

Template	Description
HH12	hour of day (01-12)
HH24	hour of day (00-23)
MI	minute (00-59)
SS	second (00-59)
SSSS	seconds past midnight (0-86399)
AM or A.M. or PM or P.M.	meridian indicator (upper case)
am or a.m. or pm or p.m.	meridian indicator (lower case)
Y,YYY	year (4 and more digits) with comma
YYYY	year (4 and more digits)
YYY	last 3 digits of year
YY	last 2 digits of year
Y	last digit of year
BC or B.C. or AD or A.D.	year indicator (upper case)
bc or b.c. or ad or a.d.	year indicator (lower case)
MONTH	full upper case month name (9 chars)
Month	full mixed case month name (9 chars)
month	full lower case month name (9 chars)
MON	upper case abbreviated month name (3 chars)
Mon	abbreviated mixed case month name (3 chars)
mon	abbreviated lower case month name (3 chars)
MM	month (01-12)
DAY	full upper case day name (9 chars)
Day	full mixed case day name (9 chars)
day	full lower case day name (9 chars)
DY	abbreviated upper case day name (3 chars)
Dy	abbreviated mixed case day name (3 chars)
dy	abbreviated lower case day name (3 chars)
DDD	day of year (001-366)
DD	day of month (01-31)
D	day of week (1-7; SUN=1)
W	week of month
WW	week number of year
CC	century (2 digits)
J	Julian Day (days since January 1, 4712 BC)
Q	quarter
RM	month in Roman Numerals (I-XII; I=JAN) - upper case
rm	month in Roman Numerals (I-XII; I=JAN) - lower case

All templates allow the use of prefix and suffix modifiers. Modifiers are always valid for use in templates. The prefix '`FX`' is a global modifier only.

Table 5.9. Suffixes for templates for date/time to_char()

Suffix	Description	Example
FM	fill mode prefix	FMMonth
TH	upper ordinal number suffix	DDTH
th	lower ordinal number suffix	DDTH
FX	FiXed format global option (see below)	FX Month DD Day
SP	spell mode (not yet implemented)	DDSP

Usage notes:

- `to_timestamp` and `to_date` skip blank space if the `FX` option is not used. `FX` must be specified as the first item in the template.
- Backslash ("\") must be specified with a double backslash ("\\"); for example `'\\HH\\MI\\SS'`.
- A double quote ("") between quotation marks is skipped and is not parsed. If you want to write a double quote to output you must preceed it with a double backslash (`'\\"`), for example `'\\"YYYY Month\\"'`.
- `to_char` supports text without a leading double quote (""), but any string between a quotation marks is rapidly handled and you are guaranteed that it will not be interpreted as a template keyword (example: `'"Hello Year: "YYYY'`).

Table 5.10. Templates for to_char(*numeric*)

Template	Description
9	value with the specified number of digits
0	value with leading zeros
. (period)	decimal point
, (comma)	group (thousand) separator
PR	negative value in angle brackets
S	negative value with minus sign (use locales)
L	currency symbol (use locales)
D	decimal point (use locales)
G	group separator (use locales)
MI	minus sign on specified position (if number < 0)
PL	plus sign on specified position (if number > 0)
SG	plus/minus sign on specified position
RN	roman numeral (input between 1 and 3999)
TH or th	convert to ordinal number
V	Shift *n* digits (see notes)
EEEE	science numbers. Now not supported.

Usage notes:

- A sign formatted using 'SG', 'PL' or 'MI' is not an anchor in the number; for example, to_char(-12, 'S9999') produces `'  -12'`, but to_char(-12, 'MI9999') produces `'-  12'`. The Oracle implementation does not allow the use of `MI` ahead of `9`, but rather requires that `9` preceeds `MI`.

- `PL`, `SG`, and `TH` are Postgres extensions.
- `9` specifies a value with the same number of digits as there are `9`s. If a digit is not available use blank space.
- `TH` does not convert values less than zero and does not convert decimal numbers. `TH` is a Postgres extension.
- `V` effectively multiplies the input values by `10^n`, where *n* is the number of digits following `V`. `to_char` does not support the use of `V` combined with a decimal point (e.g. "99.9V99" is not allowed).

Table 5.11. `to_char` Examples

Input	Output
to_char(now(),'Day, HH12:MI:SS')	'Tuesday , 05:39:18'
to_char(now(),'FMDay, HH12:MI:SS')	'Tuesday, 05:39:18'
to_char(-0.1,'99.99')	' -.10'
to_char(-0.1,'FM9.99')	'-.1'
to_char(0.1,'0.9')	' 0.1'
to_char(12,'9990999.9')	' 0012.0'
to_char(12,'FM9990999.9')	'0012'
to_char(485,'999')	' 485'
to_char(-485,'999')	'-485'
to_char(485,'9 9 9')	' 4 8 5'
to_char(1485,'9,999')	' 1,485'
to_char(1485,'9G999')	' 1 485'
to_char(148.5,'999.999')	' 148.500'
to_char(148.5,'999D999')	' 148,500'
to_char(3148.5,'9G999D999')	' 3 148,500'
to_char(-485,'999S')	'485-'
to_char(-485,'999MI')	'485-'
to_char(485,'999MI')	'485'
to_char(485,'PL999')	'+485'
to_char(485,'SG999')	'+485'
to_char(-485,'SG999')	'-485'
to_char(-485,'9SG99')	'4-85'
to_char(-485,'999PR')	'<485>'
to_char(485,'L999')	'DM 485
to_char(485,'RN')	' CDLXXXV'
to_char(485,'FMRN')	'CDLXXXV'
to_char(5.2,'FMRN')	V
to_char(482,'999th')	' 482nd'
to_char(485, '"Good number:"999')	'Good number: 485'
to_char(485.8,'"Pre-decimal:"999" Post-decimal:" .999')	'Pre-decimal: 485 Post-decimal: .800'
to_char(12,'99V999')	' 12000'

Input	Output
to_char(12.4,'99V999')	' 12400'
to_char(12.45, '99V9')	' 125'

5.6. Geometric Functions

The geometric types point, box, lseg, line, path, polygon, and circle have a large set of native support functions.

Table 5.12. Geometric Functions

Function	Returns	Description	Example
area(object)	float8	area of item	area(box '((0,0),(1,1))')
box(box,box)	box	intersection box	box(box '((0,0),(1,1))',box '((0.5,0.5),(2,2))')
center(object)	point	center of item	center(box '((0,0),(1,2))')
diameter(circle)	float8	diameter of circle	diameter(circle '((0,0),2.0)')
height(box)	float8	vertical size of box	height(box '((0,0),(1,1))')
isclosed(path)	bool	a closed path?	isclosed(path '((0,0),(1,1),(2,0))')
isopen(path)	bool	an open path?	isopen(path '[(0,0),(1,1),(2,0)]')
length(object)	float8	length of item	length(path '((-1,0),(1,0))')
pclose(path)	path	convert path to closed	popen(path '[(0,0),(1,1),(2,0)]')
npoint(path)	int4	number of points	npoints(path '[(0,0),(1,1),(2,0)]')
popen(path)	path	convert path to open path	popen(path '((0,0),(1,1),(2,0))')
radius(circle)	float8	radius of circle	radius(circle '((0,0),2.0)')
width(box)	float8	horizontal size	width(box '((0,0),(1,1))')

Table 5.13. Geometric Type Conversion Functions

Function	Returns	Description	Example
box(circle)	box	circle to box	box('((0,0),2.0)'::circle)
box(point,point)	box	points to box	box('(0,0)'::point,'(1,1)'::point)
box(polygon)	box	polygon to box	box('((0,0),(1,1),(2,0))'::polygon)
circle(box)	circle	to circle	circle('((0,0),(1,1))'::box)
circle(point,float8)	circle	point to circle	circle('(0,0)'::point,2.0)

Function	Returns	Description	Example
lseg(box)	lseg	box diagonal to lseg	lseg('((-1,0),(1,0))'::box)
lseg(point,point)	lseg	points to lseg	lseg('(-1,0)'::point,'(1,0)'::point)
path(polygon)	point	polygon to path	path('((0,0),(1,1),(2,0))'::polygon)
point(circle)	point	center	point('((0,0),2.0)'::circle)
point(lseg,lseg)	point	intersection	point('((-1,0),(1,0))'::lseg, '((-2,-2),(2,2))'::lseg)
point(polygon)	point	center	point('((0,0),(1,1),(2,0))'::polygon)
polygon(box)	polygon	12 point polygon	polygon('((0,0),(1,1))'::box)
polygon(circle)	polygon	12-point polygon	polygon('((0,0),2.0)'::circle)
polygon(*npts*,circle)	polygon	*npts* polygon	polygon(12,'((0,0),2.0)'::circle)
polygon(path)	polygon	path to polygon	polygon('((0,0),(1,1),(2,0))'::path)

Table 5.14. Geometric Upgrade Functions

Function	Returns	Description	Example
isoldpath(path)	path	test path for pre-v6.1 form	isoldpath('(1,3,0,0,1,1,2,0)'::path)
revertpoly(polygon)	polygon	to pre-v6.1	revertpoly('((0,0),(1,1),(2,0))'::polygon)
upgradepath(path)	path	to pre-v6.1	upgradepath('(1,3,0,0,1,1,2,0)'::path)
upgradepoly(polygon)	polygon	to pre-v6.1	upgradepoly('(0,1,2,0,1,0)'::polygon)

5.7. IP V4 Functions

Table 5.15. PostgresIP V4 Functions

Function	Returns	Description	Example
broadcast(cidr)	text	construct broadcast address as text	broadcast('192.168.1.5/24')
broadcast(inet)	text	construct broadcast address as text	broadcast('192.168.1.5/24')
host(inet)	text	extract host address as text	host('192.168.1.5/24')
masklen(cidr)	int4	calculate netmask length	masklen('192.168.1.5/24')
masklen(inet)	int4	calculate netmask length	masklen('192.168.1.5/24')
netmask(inet)	text	construct netmask as text	netmask('192.168.1.5/24')

Chapter 6. Type Conversion

SQL queries can, intentionally or not, require mixing of different data types in the same expression. Postgres has extensive facilities for evaluating mixed-type expressions.

In many cases a user will not need to understand the details of the type conversion mechanism. However, the implicit conversions done by Postgres can affect the apparent results of a query, and these results can be tailored by a user or programmer using *explicit* type coercion.

This chapter introduces the Postgres type conversion mechanisms and conventions. Refer to the relevant sections in the User's Guide and Programmer's Guide for more information on specific data types and allowed functions and operators.

The Programmer's Guide has more details on the exact algorithms used for implicit type conversion and coercion.

6.1. Overview

SQL is a strongly typed language. That is, every data item has an associated data type which determines its behavior and allowed usage. Postgres has an extensible type system which is much more general and flexible than other RDBMS implementations. Hence, most type conversion behavior in Postgres should be governed by general rules rather than by ad-hoc heuristics to allow mixed-type expressions to be meaningful, even with user-defined types.

The Postgres scanner/parser decodes lexical elements into only five fundamental categories: integers, floats, strings, names, and keywords. Most extended types are first tokenized into strings. The SQL language definition allows specifying type names with strings, and this mechanism is used by Postgres to start the parser down the correct path. For example, the query

```
tgl=> SELECT text 'Origin' AS "Label", point '(0,0)' AS "Value";
 Label  | Value
--------+-------
 Origin | (0,0)
(1 row)
```

has two strings, of type `text` and `point`. If a type is not specified, then the placeholder type `unknown` is assigned initially, to be resolved in later stages as described below.

There are four fundamental SQL constructs requiring distinct type conversion rules in the Postgres parser:

Operators

Postgres allows expressions with left- and right-unary (one argument) operators, as well as binary (two argument) operators.

Function calls

Much of the Postgres type system is built around a rich set of functions. Function calls have one or more arguments which, for any specific query, must be matched to the functions available in the system catalog.

Query targets

SQL INSERT statements place the results of query into a table. The expressions in the query must be matched up with, and perhaps converted to, the target columns of the insert.

UNION queries

Since all select results from a UNION SELECT statement must appear in a single set of columns, the types of each SELECT clause must be matched up and converted to a uniform set.

Many of the general type conversion rules use simple conventions built on the Postgres function and operator system tables. There are some heuristics included in the conversion rules to better support conventions for the SQL92 standard native types such as `smallint`, `integer`, and `float`.

The Postgres parser uses the convention that all type conversion functions take a single argument of the source type and are named with the same name as the target type. Any function meeting this criteria is considered to be a valid conversion function, and may be used by the parser as such. This simple assumption gives the parser the power to explore type conversion possibilities without hardcoding, allowing extended user-defined types to use these same features transparently.

An additional heuristic is provided in the parser to allow better guesses at proper behavior for SQL standard types. There are five categories of types defined: boolean, string, numeric, geometric, and user-defined. Each category, with the exception of user-defined, has a "preferred type" which is used to resolve ambiguities in candidates. Each "user-defined" type is its own "preferred type", so ambiguous expressions (those with multiple candidate parsing solutions) with only one user-defined type can resolve to a single best choice, while those with multiple user-defined types will remain ambiguous and throw an error.

Ambiguous expressions which have candidate solutions within only one type category are likely to resolve, while ambiguous expressions with candidates spanning multiple categories are likely to throw an error and ask for clarification from the user.

6.1.1. Guidelines

All type conversion rules are designed with several principles in mind:

- Implicit conversions should never have surprising or unpredictable outcomes.
- User-defined types, of which the parser has no a-priori knowledge, should be "higher" in the type hierarchy. In mixed-type expressions, native types shall always be converted to a user-defined type (of course, only if conversion is necessary).
- User-defined types are not related. Currently, Postgres does not have information available to it on relationships between types, other than hardcoded heuristics for built-in types and implicit relationships based on available functions in the catalog.
- There should be no extra overhead from the parser or executor if a query does not need implicit type conversion. That is, if a query is well formulated and the types already match up, then the query should proceed without spending extra time in the parser and without introducing unnecessary implicit conversion functions into the query.

  Additionally, if a query usually requires an implicit conversion for a function, and if then the user defines an explicit function with the correct argument types, the parser should use this new function and will no longer do the implicit conversion using the old function.

6.2. Operators

6.2.1. Conversion Procedure

Operator Evaluation

1. Check for an exact match in the pg_operator system catalog.

   a. (Optional) If one argument of a binary operator is `unknown`, then assume it is the same type as the other argument.

   b. Reverse the arguments, and look for an exact match with an operator which points to itself as being commutative. If found, then reverse the arguments in the parse tree and use this operator.

2. Look for the best match.

a. (Optional) Make a list of all operators of the same name.

b. If only one operator is in the list, use it if the input type can be coerced, and throw an error if the type cannot be coerced.

c. Keep all operators with the most explicit matches for types. Keep all if there are no explicit matches and move to the next step. If only one candidate remains, use it if the type can be coerced.

d. If any input arguments are "unknown", categorize the input candidates as boolean, numeric, string, geometric, or user-defined. If there is a mix of categories, or more than one user-defined type, throw an error because the correct choice cannot be deduced without more clues. If only one category is present, then assign the "preferred type" to the input column which had been previously "unknown".

e. Choose the candidate with the most exact type matches, and which matches the "preferred type" for each column category from the previous step. If there is still more than one candidate, or if there are none, then throw an error.

6.2.2. Examples

6.2.2.1. Exponentiation Operator

There is only one exponentiation operator defined in the catalog, and it takes `float8` arguments. The scanner assigns an initial type of `int4` to both arguments of this query expression:

```
tgl=> select 2 ^ 3 AS "Exp";
 Exp
-----
   8
(1 row)
```

So the parser does a type conversion on both operands and the query is equivalent to

```
tgl=> select float8(2) ^ float8(3) AS "Exp";
 Exp
-----
   8
(1 row)
```

or

```
tgl=> select 2.0 ^ 3.0 AS "Exp";
 Exp
-----
   8
(1 row)
```

Note

This last form has the least overhead, since no functions are called to do implicit type conversion. This is not an issue for small queries, but may have an impact on the performance of queries involving large tables.

6.2.2.2. String Concatenation

A string-like syntax is used for working with string types as well as for working with complex extended types. Strings with unspecified type are matched with likely operator candidates.

One unspecified argument:

```
tgl=> SELECT text 'abc' || 'def' AS "Text and Unknown";
 Text and Unknown
------------------
 abcdef
(1 row)
```

In this case the parser looks to see if there is an operator taking `text` for both arguments. Since there is, it assumes that the second argument should be interpreted as of type `text`.

Concatenation on unspecified types:

```
tgl=> SELECT 'abc' || 'def' AS "Unspecified";
 Unspecified
-------------
 abcdef
(1 row)
```

In this case there is no initial hint for which type to use, since no types are specified in the query. So, the parser looks for all candidate operators and finds that all arguments for all the candidates are string types. It chooses the "preferred type" for strings, `text`, for this query.

Note

If a user defines a new type and defines an operator "`||`" to work with it, then this query would no longer succeed as written. The parser would now have candidate types from two categories, and could not decide which to use.

6.2.2.3. Factorial

This example illustrates an interesting result. Traditionally, the factorial operator is defined for integers only. The Postgres operator catalog has only one entry for factorial, taking an integer operand. If given a non-integer numeric argument, Postgres will try to convert that argument to an integer for evaluation of the factorial.

```
tgl=> select (4.3 !);
 ?column?
----------
       24
(1 row)
```

Note

Of course, this leads to a mathematically suspect result, since in principle the factorial of a non-integer is not defined. However, the role of a database is not to teach mathematics, but to be a tool for data manipulation. If a user chooses to take the factorial of a floating point number, Postgres will try to oblige.

6.3. Functions

Function Evaluation

1. Check for an exact match in the pg_proc system catalog.
2. Look for the best match.
   a. Make a list of all functions of the same name with the same number of arguments.

b. If only one function is in the list, use it if the input types can be coerced, and throw an error if the types cannot be coerced.

c. Keep all functions with the most explicit matches for types. Keep all if there are no explicit matches and move to the next step. If only one candidate remains, use it if the type can be coerced.

d. If any input arguments are "unknown", categorize the input candidate arguments as boolean, numeric, string, geometric, or user-defined. If there is a mix of categories, or more than one user-defined type, throw an error because the correct choice cannot be deduced without more clues. If only one category is present, then assign the "preferred type" to the input column which had been previously "unknown".

e. Choose the candidate with the most exact type matches, and which matches the "preferred type" for each column category from the previous step. If there is still more than one candidate, or if there are none, then throw an error.

6.3.1. Examples

6.3.1.1. Factorial Function

There is only one factorial function defined in the pg_proc catalog. So the following query automatically converts the `int2` argument to `int4`:

```
tgl=> select int4fac(int2 '4');
 int4fac
---------
      24
(1 row)
```

and is actually transformed by the parser to

```
tgl=> select int4fac(int4(int2 '4'));
 int4fac
---------
      24
(1 row)
```

6.3.1.2. Substring Function

There are two `substr` functions declared in pg_proc. However, only one takes two arguments, of types `text` and `int4`.

If called with a string constant of unspecified type, the type is matched up directly with the only candidate function type:

```
tgl=> select substr('1234', 3);
 substr
--------
     34
(1 row)
```

If the string is declared to be of type `varchar`, as might be the case if it comes from a table, then the parser will try to coerce it to become `text`:

```
tgl=> select substr(varchar '1234', 3);
 substr
--------
```

```
     34
(1 row)
```

which is transformed by the parser to become

```
tgl=> select substr(text(varchar '1234'), 3);
 substr
--------
     34
(1 row)
```

Note

There are some heuristics in the parser to optimize the relationship between the `char`, `varchar`, and `text` types. For this case, `substr` is called directly with the `varchar` string rather than inserting an explicit conversion call.

And, if the function is called with an `int4`, the parser will try to convert that to `text`:

```
tgl=> select substr(1234, 3);
 substr
--------
     34
(1 row)
```

actually executes as

```
tgl=> select substr(text(1234), 3);
 substr
--------
     34
(1 row)
```

6.4. Query Targets

Target Evaluation

1. Check for an exact match with the target.
2. Try to coerce the expression directly to the target type if necessary.
3. If the target is a fixed-length type (e.g. `char` or `varchar` declared with a length) then try to find a sizing function of the same name as the type taking two arguments, the first the type name and the second an integer length.

6.4.1. Examples

6.4.1.1. `varchar` Storage

For a target column declared as `varchar(4)` the following query ensures that the target is sized correctly:

```
tgl=> CREATE TABLE vv (v varchar(4));
CREATE
tgl=> INSERT INTO vv SELECT 'abc' || 'def';
INSERT 392905 1
tgl=> SELECT * FROM vv;
  v
```

```
------
 abcd
(1 row)
```

6.5. UNION Queries

The UNION construct is somewhat different in that it must match up possibly dissimilar types to become a single result set.

UNION Evaluation

1. Check for identical types for all results.
2. Coerce each result from the UNION clauses to match the type of the first SELECT clause or the target column.

6.5.1. Examples

6.5.1.1. Underspecified Types

```
tgl=> SELECT text 'a' AS "Text" UNION SELECT 'b';
 Text
------
 a
 b
(2 rows)
```

6.5.1.2. Simple UNION

```
tgl=> SELECT 1.2 AS "Float8" UNION SELECT 1;
 Float8
--------
      1
    1.2
(2 rows)
```

6.5.1.3. Transposed UNION

The types of the union are forced to match the types of the first/top clause in the union:

```
tgl=> SELECT 1 AS "All integers"
tgl-> UNION SELECT '2.2'::float4
tgl-> UNION SELECT 3.3;
 All integers
--------------
            1
            2
            3
(3 rows)
```

An alternate parser strategy could be to choose the "best" type of the bunch, but this is more difficult because of the nice recursion technique used in the parser. However, the "best" type is used when selecting *into* a table:

```
tgl=> CREATE TABLE ff (f float);
CREATE
```

```
tgl=> INSERT INTO ff
tgl-> SELECT 1
tgl-> UNION SELECT '2.2'::float4
tgl-> UNION SELECT 3.3;
INSERT 0 3
tgl=> SELECT f AS "Floating point" from ff;
  Floating point
------------------
                1
 2.20000004768372
              3.3
(3 rows)
```

Chapter 7. Indices and Keys

Indexes are commonly used to enhance database performance. They should be defined on table columns (or class attributes) which are used as qualifications in repetitive queries. Inappropriate use will result in slower performance, since update and insertion times are increased in the presence of indices.

Indexes may also be used to enforce uniqueness of a table's primary key. When an index is declared UNIQUE, multiple table rows with identical index entries won't be allowed. For this purpose, the goal is ensuring data consistency, not improving performance, so the above caution about inappropriate use doesn't apply.

Two forms of indices may be defined:

- For a *value index*, the key fields for the index are specified as column names; multiple columns can be specified if the index access method supports multi-column indexes.

- For a *functional index*, an index is defined on the result of a function applied to one or more attributes of a single class. This is a single-column index (namely, the function result) even if the function uses more than one input field. Functional indices can be used to obtain fast access to data based on operators that would normally require some transformation to apply them to the base data.

Postgres provides btree, rtree and hash access methods for indices. The btree access method is an implementation of Lehman-Yao high-concurrency btrees. The rtree access method implements standard rtrees using Guttman's quadratic split algorithm. The hash access method is an implementation of Litwin's linear hashing. We mention the algorithms used solely to indicate that all of these access methods are fully dynamic and do not have to be optimized periodically (as is the case with, for example, static hash access methods).

The Postgres query optimizer will consider using a btree index whenever an indexed attribute is involved in a comparison using one of: <, <=, =, >=, >

The Postgres query optimizer will consider using an rtree index whenever an indexed attribute is involved in a comparison using one of: <<, &<, &>, >>, @, ~=, &&

The Postgres query optimizer will consider using a hash index whenever an indexed attribute is involved in a comparison using the = operator.

Currently, only the btree access method supports multi-column indexes. Up to 16 keys may be specified by default (this limit can be altered when building Postgres).

An *operator class* can be specified for each column of an index. The operator class identifies the operators to be used by the index for that column. For example, a btree index on four-byte integers would use the `int4_ops` class; this operator class includes comparison functions for four-byte integers. In practice the default operator class for the field's datatype is usually sufficient. The main point of having operator classes is that for some datatypes, there could be more than one meaningful ordering. For example, we might want to sort a complex-number datatype either by absolute value or by real part. We could do this by defining two operator classes for the datatype and then selecting the proper class when making an index. There are also some operator classes with special purposes:

- The operator classes `box_ops` and `bigbox_ops` both support rtree indices on the `box` datatype. The difference between them is that `bigbox_ops` scales box coordinates down, to avoid floating point exceptions from doing multiplication, addition, and subtraction on very large floating-point coordinates. If the field on which your rectangles lie is about 20,000 units square or larger, you should use `bigbox_ops`.

- The `int24_ops` operator class is useful for constructing indices on int2 data, and doing comparisons against int4 data in query qualifications. Similarly, `int42_ops` support indices on int4 data that is to be compared against int2 data in queries.

The following query shows all defined operator classes:

```
SELECT am.amname AS acc_name,
       opc.opcname AS ops_name,
       opr.oprname AS ops_comp
    FROM pg_am am, pg_amop amop,
         pg_opclass opc, pg_operator opr
    WHERE amop.amopid = am.oid AND
          amop.amopclaid = opc.oid AND
          amop.amopopr = opr.oid
    ORDER BY acc_name, ops_name, ops_comp
```

Use DROP INDEX to remove an index.

7.1. Keys

Author

Written by Herouth Maoz[1] This originally appeared on the User's Mailing List on 1998-03-02 in response to the question: "What is the difference between PRIMARY KEY and UNIQUE constraints?".

```
Subject: Re: [QUESTIONS] PRIMARY KEY | UNIQUE

        What's the difference between:

              PRIMARY KEY(fields,...) and
              UNIQUE (fields,...)

       - Is this an alias?
       - If PRIMARY KEY is already unique, then why
         is there another kind of key named UNIQUE?
```

A primary key is the field(s) used to identify a specific row. For example, Social Security numbers identifying a person.

A simply UNIQUE combination of fields has nothing to do with identifying the row. It's simply an integrity constraint. For example, I have collections of links. Each collection is identified by a unique number, which is the primary key. This key is used in relations.

However, my application requires that each collection will also have a unique name. Why? So that a human being who wants to modify a collection will be able to identify it. It's much harder to know, if you have two collections named "Life Science", the the one tagged 24433 is the one you need, and the one tagged 29882 is not.

So, the user selects the collection by its name. We therefore make sure, within the database, that names are unique. However, no other table in the database relates to the collections table by the collection Name. That would be very inefficient.

Moreover, despite being unique, the collection name does not actually define the collection! For example, if somebody decided to change the name of the collection from "Life Science" to "Biology", it will still be the same collection, only with a different name. As long as the name is unique, that's OK.

So:

- Primary key:

[1] herouth@oumail.openu.ac.il

- Is used for identifying the row and relating to it.
- Is impossible (or hard) to update.
- Should not allow NULLs.

- Unique field(s):
  - Are used as an alternative access to the row.
  - Are updateable, so long as they are kept unique.
  - NULLs are acceptable.

As for why no non-unique keys are defined explicitly in standard SQL syntax? Well, you must understand that indices are implementation-dependent. SQL does not define the implementation, merely the relations between data in the database. Postgres does allow non-unique indices, but indices used to enforce SQL keys are always unique.

Thus, you may query a table by any combination of its columns, despite the fact that you don't have an index on these columns. The indexes are merely an implementational aid which each RDBMS offers you, in order to cause commonly used queries to be done more efficiently. Some RDBMS may give you additional measures, such as keeping a key stored in main memory. They will have a special command, for example

```
CREATE MEMSTORE ON <table> COLUMNS <cols>
```

(this is not an existing command, just an example).

In fact, when you create a primary key or a unique combination of fields, nowhere in the SQL specification does it say that an index is created, nor that the retrieval of data by the key is going to be more efficient than a sequential scan!

So, if you want to use a combination of fields which is not unique as a secondary key, you really don't have to specify anything - just start retrieving by that combination! However, if you want to make the retrieval efficient, you'll have to resort to the means your RDBMS provider gives you - be it an index, my imaginary MEMSTORE command, or an intelligent RDBMS which creates indices without your knowledge based on the fact that you have sent it many queries based on a specific combination of keys... (It learns from experience).

7.2. Partial Indices

Author

This is from a reply to a question on the e-mail list by Paul M. Aoki[2] on 1998-08-11.

A *partial index* is an index built over a subset of a table; the subset is defined by a predicate. Postgres supported partial indices with arbitrary predicates. I believe IBM's DB2 for AS/400 supports partial indices using single-clause predicates.

The main motivation for partial indices is this: if all of the queries you ask that can profitably use an index fall into a certain range, why build an index over the whole table and suffer the associated space/time costs? (There are other reasons too; see Stonebraker, M, 1989b for details.)

The machinery to build, update and query partial indices isn't too bad. The hairy parts are index selection (which indices do I build?) and query optimization (which indices do I use?); i.e., the parts that involve deciding what predicate(s) match the workload/query in some useful way. For those who are into database theory, the problems are basically analogous to the corresponding materialized view problems, albeit with different cost parameters and formulae. These are, in the general case, hard problems for the standard ordinal SQL types; they're super-hard problems with black-box extension types, because the selectivity estimation technology is so crude.

[2] aoki@CS.Berkeley.EDU

Check Stonebraker, M, 1989b, Olson, 1993 , and Seshardri, 1995 for more information.

Chapter 8. Arrays

Note

This must become a chapter on array behavior. Volunteers? - thomas 1998-01-12

Postgres allows attributes of a class to be defined as variable-length multi-dimensional arrays. Arrays of any built-in type or user-defined type can be created. To illustrate their use, we create this class:

```
CREATE TABLE sal_emp (
    name            text,
    pay_by_quarter  int4[],
    schedule        text[][]
);
```

The above query will create a class named *sal_emp* with a *text* string (name), a one-dimensional array of *int4* (pay_by_quarter), which represents the employee's salary by quarter, and a two-dimensional array of *text* (schedule), which represents the employee's weekly schedule. Now we do some *INSERTS*s; note that when appending to an array, we enclose the values within braces and separate them by commas. If you know *C*, this is not unlike the syntax for initializing structures.

```
INSERT INTO sal_emp
    VALUES ('Bill',
    '{10000, 10000, 10000, 10000}',
    '{{"meeting", "lunch"}, {}}');

INSERT INTO sal_emp
    VALUES ('Carol',
    '{20000, 25000, 25000, 25000}',
    '{{"talk", "consult"}, {"meeting"}}');
```

Now, we can run some queries on sal_emp. First, we show how to access a single element of an array at a time. This query retrieves the names of the employees whose pay changed in the second quarter:

```
SELECT name FROM sal_emp WHERE pay_by_quarter[1] <>
 pay_by_quarter[2];

 name
-------
 Carol
(1 row)
```

Postgres uses the "one-based" numbering convention for arrays --- that is, an array of n elements starts with array[1] and ends with array[n].

This query retrieves the third quarter pay of all employees:

```
SELECT pay_by_quarter[3] FROM sal_emp;

 pay_by_quarter
----------------
          10000
          25000
(2 rows)
```

We can also access arbitrary slices of an array, or subarrays. An array slice is denoted by writing "lower subscript : upper subscript" for one or more array dimensions. This query retrieves the first item on Bill's schedule for the first two days of the week:

```
SELECT schedule[1:2][1:1] FROM sal_emp WHERE name = 'Bill';

      schedule
--------------------
 {{"meeting"},{""}}
(1 row)
```

We could also have written

```
SELECT schedule[1:2][1] FROM sal_emp WHERE name = 'Bill';
```

with the same result.

An array value can be replaced completely:

```
UPDATE sal_emp SET pay_by_quarter = '{25000,25000,27000,27000}'
    WHERE name = 'Carol';
```

or updated at a single entry:

```
UPDATE sal_emp SET pay_by_quarter[4] = 15000
    WHERE name = 'Bill';
```

or updated in a slice:

```
UPDATE sal_emp SET pay_by_quarter[1:2] = '{27000,27000}'
    WHERE name = 'Carol';
```

It is not currently possible to resize an array value except by complete replacement; for example, we couldn't change a four- element array value to a five-element value with a single assignment to array[5].

The syntax for CREATE TABLE allows fixed-length arrays to be defined:

```
CREATE TABLE tictactoe (
    squares   int4[3][3]
);
```

However, the current implementation does not enforce the array size limits --- the behavior is the same as for arrays of unspecified length.

Chapter 9. Inheritance

Let's create two classes. The capitals class contains state capitals which are also cities. Naturally, the capitals class should inherit from cities.

```
CREATE TABLE cities (
    name            text,
    population      float,
    altitude        int     -- (in ft)
);

CREATE TABLE capitals (
    state           char(2)
) INHERITS (cities);
```

In this case, an instance of capitals *inherits* all attributes (name, population, and altitude) from its parent, cities. The type of the attribute name is `text`, a native Postgres type for variable length ASCII strings. The type of the attribute population is `float`, a native Postgres type for double precision floating point numbers. State capitals have an extra attribute, state, that shows their state. In Postgres, a class can inherit from zero or more other classes, and a query can reference either all instances of a class or all instances of a class plus all of its descendants.

Note

The inheritance hierarchy is a actually a directed acyclic graph.

```
SELECT name, altitude
    FROM cities
    WHERE altitude > 500;

   name    | altitude
-----------+----------
 Las Vegas |     2174
 Mariposa  |     1953
(2 rows)
```

On the other hand, to find the names of all cities, including state capitals, that are located at an altitude over 500ft, the query is:

```
SELECT c.name, c.altitude
    FROM cities* c
    WHERE c.altitude > 500;
```

which returns:

```
   name    | altitude
-----------+----------
 Las Vegas |     2174
 Mariposa  |     1953
 Madison   |      845
```

Here the "*" after cities indicates that the query should be run over cities and all classes below cities in the inheritance hierarchy. Many of the commands that we have already discussed -- `SELECT`, `UPDATE` and `DELETE` -- support this "*" notation, as do others, like `ALTER TABLE`.

Chapter 10. PL/pgSQL - SQL Procedural Language

PL/pgSQL is a loadable procedural language for the Postgres database system.

This package was originally written by Jan Wieck.

10.1. Overview

The design goals of PL/pgSQL were to create a loadable procedural language that

- can be used to create functions and trigger procedures,
- adds control structures to the SQL language,
- can perform complex computations,
- inherits all user defined types, functions and operators,
- can be defined to be trusted by the server,
- is easy to use.

The PL/pgSQL call handler parses the functions source text and produces an internal binary instruction tree on the first time, the function is called by a backend. The produced bytecode is identified in the call handler by the object ID of the function. This ensures, that changing a function by a DROP/CREATE sequence will take effect without establishing a new database connection.

For all expressions and SQL statements used in the function, the PL/pgSQL bytecode interpreter creates a prepared execution plan using the SPI managers SPI_prepare() and SPI_saveplan() functions. This is done the first time, the individual statement is processed in the PL/pgSQL function. Thus, a function with conditional code that contains many statements for which execution plans would be required, will only prepare and save those plans that are really used during the entire lifetime of the database connection.

Except for input-/output-conversion and calculation functions for user defined types, anything that can be defined in C language functions can also be done with PL/pgSQL. It is possible to create complex conditional computation functions and later use them to define operators or use them in functional indices.

10.2. Description

10.2.1. Structure of PL/pgSQL

The PL/pgSQL language is case insensitive. All keywords and identifiers can be used in mixed upper- and lowercase.

PL/pgSQL is a block oriented language. A block is defined as

```
[<<label>>]
[DECLARE
    declarations]
BEGIN
    statements
```

```
END;
```

There can be any number of subblocks in the statement section of a block. Subblocks can be used to hide variables from outside a block of statements. The variables declared in the declarations section preceding a block are initialized to their default values every time the block is entered, not only once per function call.

It is important not to misunderstand the meaning of BEGIN/END for grouping statements in PL/pgSQL and the database commands for transaction control. Functions and trigger procedures cannot start or commit transactions and Postgres does not have nested transactions.

10.2.2. Comments

There are two types of comments in PL/pgSQL. A double dash '--' starts a comment that extends to the end of the line. A '/*' starts a block comment that extends to the next occurence of '*/'. Block comments cannot be nested, but double dash comments can be enclosed into a block comment and a double dash can hide the block comment delimiters '/*' and '*/'.

10.2.3. Declarations

All variables, rows and records used in a block or it's subblocks must be declared in the declarations section of a block except for the loop variable of a FOR loop iterating over a range of integer values. Parameters given to a PL/pgSQL function are automatically declared with the usual identifiers $n. The declarations have the following syntax:

name [CONSTANT] *type*> [NOT NULL] [DEFAULT | := *value*];

Declares a variable of the specified base type. If the variable is declared as CONSTANT, the value cannot be changed. If NOT NULL is specified, an assignment of a NULL value results in a runtime error. Since the default value of all variables is the SQL NULL value, all variables declared as NOT NULL must also have a default value specified.

The default value is evaluated ever time the function is called. So assigning '*now*' to a variable of type *datetime* causes the variable to have the time of the actual function call, not when the function was precompiled into it's bytecode.

name class%ROWTYPE;

Declares a row with the structure of the given class. Class must be an existing table- or viewname of the database. The fields of the row are accessed in the dot notation. Parameters to a function can be composite types (complete table rows). In that case, the corresponding identifier $n will be a rowtype, but it must be aliased using the ALIAS command described below. Only the user attributes of a table row are accessible in the row, no Oid or other system attributes (hence the row could be from a view and view rows don't have useful system attributes).

The fields of the rowtype inherit the tables fieldsizes or precision for char() etc. data types.

name RECORD;

Records are similar to rowtypes, but they have no predefined structure. They are used in selections and FOR loops to hold one actual database row from a SELECT operation. One and the same record can be used in different selections. Accessing a record or an attempt to assign a value to a record field when there is no actual row in it results in a runtime error.

The NEW and OLD rows in a trigger are given to the procedure as records. This is necessary because in Postgres one and the same trigger procedure can handle trigger events for different tables.

name ALIAS FOR $n;

For better readability of the code it is possible to define an alias for a positional parameter to a function.

This aliasing is required for composite types given as arguments to a function. The dot notation $1.salary as in SQL functions is not allowed in PL/pgSQL.

RENAME *oldname* TO *newname*;

Change the name of a variable, record or row. This is useful if NEW or OLD should be referenced by another name inside a trigger procedure.

10.2.4. Data Types

The type of a varible can be any of the existing basetypes of the database. *type* in the declarations section above is defined as:

- Postgres-basetype
- *variable*%TYPE
- *class.field*%TYPE

variable is the name of a variable, previously declared in the same function, that is visible at this point.

class is the name of an existing table or view where *field* is the name of an attribute.

Using the *class.field*%TYPE causes PL/pgSQL to lookup the attributes definitions at the first call to the function during the lifetime of a backend. Have a table with a char(20) attribute and some PL/pgSQL functions that deal with it's content in local variables. Now someone decides that char(20) isn't enough, dumps the table, drops it, recreates it now with the attribute in question defined as char(40) and restores the data. Ha - he forgot about the funcitons. The computations inside them will truncate the values to 20 characters. But if they are defined using the *class.field*%TYPE declarations, they will automagically handle the size change or if the new table schema defines the attribute as text type.

10.2.5. Expressions

All expressions used in PL/pgSQL statements are processed using the backends executor. Expressions which appear to contain constants may in fact require run-time evaluation (e.g. 'now' for the datetime type) so it is impossible for the PL/pgSQL parser to identify real constant values other than the NULL keyword. All expressions are evaluated internally by executing a query

```
SELECT expression
```

using the SPI manager. In the expression, occurences of variable identifiers are substituted by parameters and the actual values from the variables are passed to the executor in the parameter array. All expressions used in a PL/pgSQL function are only prepared and saved once.

The type checking done by the Postgres main parser has some side effects to the interpretation of constant values. In detail there is a difference between what the two functions

```
CREATE FUNCTION logfunc1 (text) RETURNS datetime AS '
    DECLARE
        logtxt ALIAS FOR $1;
    BEGIN
        INSERT INTO logtable VALUES (logtxt, ''now'');
```

```
        RETURN ''now'';
    END;
' LANGUAGE 'plpgsql';
```

and

```
CREATE FUNCTION logfunc2 (text) RETURNS datetime AS '
    DECLARE
        logtxt ALIAS FOR $1;
        curtime datetime;
    BEGIN
        curtime := ''now'';
        INSERT INTO logtable VALUES (logtxt, curtime);
        RETURN curtime;
    END;
' LANGUAGE 'plpgsql';
```

do. In the case of logfunc1(), the Postgres main parser knows when preparing the plan for the INSERT, that the string 'now' should be interpreted as datetime because the target field of logtable is of that type. Thus, it will make a constant from it at this time and this constant value is then used in all invocations of logfunc1() during the lifetime of the backend. Needless to say that this isn't what the programmer wanted.

In the case of logfunc2(), the Postgres main parser does not know what type 'now' should become and therefor it returns a datatype of text containing the string 'now'. During the assignment to the local variable curtime, the PL/pgSQL interpreter casts this string to the datetime type by calling the text_out() and datetime_in() functions for the conversion.

This type checking done by the Postgres main parser got implemented after PL/pgSQL was nearly done. It is a difference between 6.3 and 6.4 and affects all functions using the prepared plan feature of the SPI manager. Using a local variable in the above manner is currently the only way in PL/pgSQL to get those values interpreted correctly.

If record fields are used in expressions or statements, the data types of fields should not change between calls of one and the same expression. Keep this in mind when writing trigger procedures that handle events for more than one table.

10.2.6. Statements

Anything not understood by the PL/pgSQL parser as specified below will be put into a query and sent down to the database engine to execute. The resulting query should not return any data.

Assignment

An assignment of a value to a variable or row/record field is written as

```
identifier := expression;
```

If the expressions result data type doesn't match the variables data type, or the variable has a size/precision that is known (as for char(20)), the result value will be implicitly casted by the PL/pgSQL bytecode interpreter using the result types output- and the variables type input-functions. Note that this could potentially result in runtime errors generated by the types input functions.

An assignment of a complete selection into a record or row can be done by

```
SELECT  INTO target expressions FROM ...;
```

target can be a record, a row variable or a comma separated list of variables and record-/row-fields.

if a row or a variable list is used as target, the selected values must exactly match the structure of the target(s) or a runtime error occurs. The FROM keyword can be followed by any valid qualification, grouping, sorting etc. that can be given for a SELECT statement.

There is a special variable named FOUND of type bool that can be used immediately after a SELECT INTO to check if an assignment had success.

```
SELECT INTO myrec * FROM EMP WHERE empname = myname;
IF NOT FOUND THEN
    RAISE EXCEPTION ''employee % not found'', myname;
END IF;
```

If the selection returns multiple rows, only the first is moved into the target fields. All others are silently discarded.

Calling another function

All functions defined in a Prostgres database return a value. Thus, the normal way to call a function is to execute a SELECT query or doing an assignment (resulting in a PL/pgSQL internal SELECT). But there are cases where someone isn't interested int the functions result.

```
PERFORM query
```

executes a 'SELECT *query*' over the SPI manager and discards the result. Identifiers like local variables are still substituted into parameters.

Returning from the function

```
RETURN expression
```

The function terminates and the value of *expression* will be returned to the upper executor. The return value of a function cannot be undefined. If control reaches the end of the toplevel block of the function without hitting a RETURN statement, a runtime error will occur.

The expressions result will be automatically casted into the functions return type as described for assignments.

Aborting and messages

As indicated in the above examples there is a RAISE statement that can throw messages into the Postgres elog mechanism.

```
RAISE level format'' [, identifier [...]];
```

Inside the format, "%" is used as a placeholder for the subsequent comma-separated identifiers. Possible levels are DEBUG (silently suppressed in production running databases), NOTICE (written into the database log and forwarded to the client application) and EXCEPTION (written into the database log and aborting the transaction).

Conditionals

```
IF expression THEN
    statements
[ELSE
    statements]
```

```
END IF;
```

The *expression* must return a value that at least can be casted into a boolean type.

Loops

There are multiple types of loops.

```
[<<label>>]
LOOP
    statements
END LOOP;
```

An unconditional loop that must be terminated explicitly by an EXIT statement. The optional label can be used by EXIT statements of nested loops to specify which level of nesting should be terminated.

```
[<<label>>]
WHILE expression LOOP
    statements
END LOOP;
```

A conditional loop that is executed as long as the evaluation of *expression* is true.

```
[<<label>>]
FOR name IN [ REVERSE ] expression .. expression LOOP
    statements
END LOOP;
```

A loop that iterates over a range of integer values. The variable *name* is automatically created as type integer and exists only inside the loop. The two expressions giving the lower and upper bound of the range are evaluated only when entering the loop. The iteration step is always 1.

```
[<<label>>]
FOR record | row IN select_clause LOOP
    statements
END LOOP;
```

The record or row is assigned all the rows resulting from the select clause and the statements executed for each. If the loop is terminated with an EXIT statement, the last assigned row is still accessible after the loop.

```
EXIT [ label ] [ WHEN expression ];
```

If no *label* given, the innermost loop is terminated and the statement following END LOOP is executed next. If *label* is given, it must be the label of the current or an upper level of nested loop blocks. Then the named loop or block is terminated and control continues with the statement after the loops/blocks corresponding END.

10.2.7. Trigger Procedures

PL/pgSQL can be used to define trigger procedures. They are created with the usual CREATE FUNCTION command as a function with no arguments and a return type of OPAQUE.

There are some Postgres specific details in functions used as trigger procedures.

First they have some special variables created automatically in the toplevel blocks declaration section. They are

NEW

Datatype RECORD; variable holding the new database row on INSERT/UPDATE operations on ROW level triggers.

OLD

Datatype RECORD; variable holding the old database row on UPDATE/DELETE operations on ROW level triggers.

TG_NAME

Datatype name; variable that contains the name of the trigger actually fired.

TG_WHEN

Datatype text; a string of either 'BEFORE' or 'AFTER' depending on the triggers definition.

TG_LEVEL

Datatype text; a string of either 'ROW' or 'STATEMENT' depending on the triggers definition.

TG_OP

Datatype text; a string of 'INSERT', 'UPDATE' or 'DELETE' telling for which operation the trigger is actually fired.

TG_RELID

Datatype oid; the object ID of the table that caused the trigger invocation.

TG_RELNAME

Datatype name; the name of the table that caused the trigger invocation.

TG_NARGS

Datatype integer; the number of arguments given to the trigger procedure in the CREATE TRIGGER statement.

TG_ARGV[]

Datatype array of text; the arguments from the CREATE TRIGGER statement. The index counts from 0 and can be given as an expression. Invalid indices (< 0 or >= tg_nargs) result in a NULL value.

Second they must return either NULL or a record/row containing exactly the structure of the table the trigger was fired for. Triggers fired AFTER might always return a NULL value with no effect. Triggers fired BEFORE signal the trigger manager to skip the operation for this actual row when returning NULL. Otherwise, the returned record/row replaces the inserted/updated row in the operation. It is possible to replace single values directly in NEW and return that or to build a complete new record/row to return.

10.2.8. Exceptions

Postgres does not have a very smart exception handling model. Whenever the parser, planner/optimizer or executor decide that a statement cannot be processed any longer, the whole transaction gets aborted and the system jumps back into the mainloop to get the next query from the client application.

It is possible to hook into the error mechanism to notice that this happens. But currently it's impossible to tell what really caused the abort (input/output conversion error, floating point error, parse error). And it is possible that the database backend is in an inconsistent state at this point so returning to the upper executor or issuing more commands might corrupt the whole database. And even if, at this point the information, that the transaction is aborted, is already sent to the client application, so resuming operation does not make any sense.

Thus, the only thing PL/pgSQL currently does when it encounters an abort during execution of a function or trigger procedure is to write some additional DEBUG level log messages telling in which function and where (line number and type of statement) this happened.

10.3. Examples

Here are only a few functions to demonstrate how easy PL/pgSQL functions can be written. For more complex examples the programmer might look at the regression test for PL/pgSQL.

One painful detail of writing functions in PL/pgSQL is the handling of single quotes. The functions source text on CREATE FUNCTION must be a literal string. Single quotes inside of literal strings must be either doubled or quoted with a backslash. We are still looking for an elegant alternative. In the meantime, doubling the single qoutes as in the examples below should be used. Any solution for this in future versions of Postgres will be upward compatible.

10.3.1. Some Simple PL/pgSQL Functions

The following two PL/pgSQL functions are identical to their counterparts from the C language function discussion.

```
CREATE FUNCTION add_one (int4) RETURNS int4 AS '
    BEGIN
        RETURN $1 + 1;
    END;
' LANGUAGE 'plpgsql';


CREATE FUNCTION concat_text (text, text) RETURNS text AS '
    BEGIN
        RETURN $1 || $2;
    END;
' LANGUAGE 'plpgsql';
```

10.3.2. PL/pgSQL Function on Composite Type

Again it is the PL/pgSQL equivalent to the example from The C functions.

```
CREATE FUNCTION c_overpaid (EMP, int4) RETURNS bool AS '
    DECLARE
        emprec ALIAS FOR $1;
        sallim ALIAS FOR $2;
    BEGIN
        IF emprec.salary ISNULL THEN
            RETURN ''f'';
        END IF;
        RETURN emprec.salary > sallim;
    END;
' LANGUAGE 'plpgsql';
```

10.3.3. PL/pgSQL Trigger Procedure

This trigger ensures, that any time a row is inserted or updated in the table, the current username and time are stamped into the row. And it ensures that an employees name is given and that the salary is a positive value.

```
CREATE TABLE emp (
    empname text,
    salary int4,
    last_date datetime,
    last_user name);

CREATE FUNCTION emp_stamp () RETURNS OPAQUE AS '
    BEGIN
        -- Check that empname and salary are given
        IF NEW.empname ISNULL THEN
            RAISE EXCEPTION ''empname cannot be NULL value'';
        END IF;
        IF NEW.salary ISNULL THEN
            RAISE EXCEPTION ''% cannot have NULL salary'',
 NEW.empname;
        END IF;

        -- Who works for us when she must pay for?
        IF NEW.salary < 0 THEN
            RAISE EXCEPTION ''% cannot have a negative salary'',
 NEW.empname;
        END IF;

        -- Remember who changed the payroll when
        NEW.last_date := ''now'';
        NEW.last_user := getpgusername();
        RETURN NEW;
    END;
' LANGUAGE 'plpgsql';

CREATE TRIGGER emp_stamp BEFORE INSERT OR UPDATE ON emp
    FOR EACH ROW EXECUTE PROCEDURE emp_stamp();
```

Chapter 11. PL/Tcl - TCL Procedural Language

PL/Tcl is a loadable procedural language for the Postgres database system that enables the Tcl language to be used to create functions and trigger-procedures.

This package was originally written by Jan Wieck.

11.1. Overview

PL/Tcl offers most of the capabilities a function writer has in the C language, except for some restrictions.

The good restriction is, that everything is executed in a safe Tcl-interpreter. In addition to the limited command set of safe Tcl, only a few commands are available to access the database over SPI and to raise messages via elog(). There is no way to access internals of the database backend or gaining OS-level access under the permissions of the Postgres user ID like in C. Thus, any unprivileged database user may be permitted to use this language.

The other, internal given, restriction is, that Tcl procedures cannot be used to create input-/output-functions for new data types.

The shared object for the PL/Tcl call handler is automatically built and installed in the Postgres library directory if the Tcl/Tk support is specified in the configuration step of the installation procedure.

11.2. Description

11.2.1. Postgres Functions and Tcl Procedure Names

In Postgres, one and the same function name can be used for different functions as long as the number of arguments or their types differ. This would collide with Tcl procedure names. To offer the same flexibility in PL/Tcl, the internal Tcl procedure names contain the object ID of the procedures pg_proc row as part of their name. Thus, different argtype versions of the same Postgres function are different for Tcl too.

11.2.2. Defining Functions in PL/Tcl

To create a function in the PL/Tcl language, use the known syntax

```
CREATE FUNCTION funcname argument-types) RETURNS return-type AS '
    # PL/Tcl function body
' LANGUAGE 'pltcl';
```

When calling this function in a query, the arguments are given as variables $1 ... $n to the Tcl procedure body. So a little max function returning the higher of two int4 values would be created as:

```
CREATE FUNCTION tcl_max (int4, int4) RETURNS int4 AS '
    if {$1 > $2} {return $1}
    return $2
' LANGUAGE 'pltcl';
```

Composite type arguments are given to the procedure as Tcl arrays. The element names in the array are the attribute names of the composite type. If an attribute in the actual row has the NULL value,

it will not appear in the array! Here is an example that defines the overpaid_2 function (as found in the older Postgres documentation) in PL/Tcl

```
CREATE FUNCTION overpaid_2 (EMP) RETURNS bool AS '
    if {200000.0 < $1(salary)} {
        return "t"
    }
    if {$1(age) < 30 && 100000.0 < $1(salary)} {
        return "t"
    }
    return "f"
' LANGUAGE 'pltcl';
```

11.2.3. Global Data in PL/Tcl

Sometimes (especially when using the SPI functions described later) it is useful to have some global status data that is held between two calls to a procedure. All PL/Tcl procedures executed in one backend share the same safe Tcl interpreter. To help protecting PL/Tcl procedures from side effects, an array is made available to each procedure via the upvar command. The global name of this variable is the procedures internal name and the local name is GD.

11.2.4. Trigger Procedures in PL/Tcl

Trigger procedures are defined in Postgres as functions without arguments and a return type of opaque. And so are they in the PL/Tcl language.

The informations from the trigger manager are given to the procedure body in the following variables:

$TG_name

The name of the trigger from the CREATE TRIGGER statement.

$TG_relid

The object ID of the table that caused the trigger procedure to be invoked.

$TG_relatts

A Tcl list of the tables field names prefixed with an empty list element. So looking up an element name in the list with the lsearch Tcl command returns the same positive number starting from 1 as the fields are numbered in the pg_attribute system catalog.

$TG_when

The string BEFORE or AFTER depending on the event of the trigger call.

$TG_level

The string ROW or STATEMENT depending on the event of the trigger call.

$TG_op

The string INSERT, UPDATE or DELETE depending on the event of the trigger call.

$NEW

An array containing the values of the new table row on INSERT/UPDATE actions, or empty on DELETE.

$OLD

An array containing the values of the old table row on UPDATE/DELETE actions, or empty on INSERT.

$GD

The global status data array as described above.

$args

A Tcl list of the arguments to the procedure as given in the CREATE TRIGGER statement. The arguments are also accessible as $1 ... $n in the procedure body.

The return value from a trigger procedure is one of the strings OK or SKIP, or a list as returned by the 'array get' Tcl command. If the return value is OK, the normal operation (INSERT/UPDATE/DELETE) that fired this trigger will take place. Obviously, SKIP tells the trigger manager to silently suppress the operation. The list from 'array get' tells PL/Tcl to return a modified row to the trigger manager that will be inserted instead of the one given in $NEW (INSERT/UPDATE only). Needless to say that all this is only meaningful when the trigger is BEFORE and FOR EACH ROW.

Here's a little example trigger procedure that forces an integer value in a table to keep track of the # of updates that are performed on the row. For new row's inserted, the value is initialized to 0 and then incremented on every update operation:

```
CREATE FUNCTION trigfunc_modcount() RETURNS OPAQUE AS '
    switch $TG_op {
        INSERT {
            set NEW($1) 0
        }
        UPDATE {
            set NEW($1) $OLD($1)
            incr NEW($1)
        }
        default {
            return OK
        }
    }
    return [array get NEW]
' LANGUAGE 'pltcl';

CREATE TABLE mytab (num int4, modcnt int4, desc text);

CREATE TRIGGER trig_mytab_modcount BEFORE INSERT OR UPDATE ON mytab
    FOR EACH ROW EXECUTE PROCEDURE trigfunc_modcount('modcnt');
```

11.2.5. Database Access from PL/Tcl

The following commands are available to access the database from the body of a PL/Tcl procedure:

elog *level msg*

Fire a log message. Possible levels are NOTICE, ERROR, FATAL, DEBUG and NOIND like for the `elog` C function.

quote *string*

Duplicates all occurences of single quote and backslash characters. It should be used when variables are used in the query string given to `spi_exec` or `spi_prepare` (not for the value list on `spi_execp`). Think about a query string like

```
"SELECT '$val' AS ret"
```

where the Tcl variable val actually contains "doesn't". This would result in the final query string

```
"SELECT 'doesn't' AS ret"
```

what would cause a parse error during `spi_exec` or `spi_prepare`. It should contain

```
"SELECT 'doesn''t' AS ret"
```

and has to be written as

```
"SELECT '[ quote $val ]' AS ret"
```

spi_exec ?-count *n*? ?-array *name*? *query* ?*loop-body*?

Call parser/planner/optimizer/executor for query. The optional -count value tells `spi_exec` the maximum number of rows to be processed by the query.

If the query is a SELECT statement and the optional loop-body (a body of Tcl commands like in a foreach statement) is given, it is evaluated for each row selected and behaves like expected on continue/break. The values of selected fields are put into variables named as the column names. So a

```
spi_exec "SELECT count(*) AS cnt FROM pg_proc"
```

will set the variable $cnt to the number of rows in the pg_proc system catalog. If the option -array is given, the column values are stored in the associative array named 'name' indexed by the column name instead of individual variables.

```
spi_exec -array C "SELECT * FROM pg_class" {
    elog DEBUG "have table $C(relname)"
}
```

will print a DEBUG log message for every row of pg_class. The return value of `spi_exec` is the number of rows affected by query as found in the global variable SPI_processed.

spi_prepare *query typelist*

Prepares AND SAVES a query plan for later execution. It is a bit different from the C level SPI_prepare in that the plan is automatically copied to the toplevel memory context. Thus, there is currently no way of preparing a plan without saving it.

If the query references arguments, the type names must be given as a Tcl list. The return value from spi_prepare is a query ID to be used in subsequent calls to spi_execp. See spi_execp for a sample.

spi_exec ?-count *n*? ?-array*name*? ?-nulls*string*? *query* ?*value-list*? ?*loop-body*?

Execute a prepared plan from spi_prepare with variable substitution. The optional -count value tells spi_execp the maximum number of rows to be processed by the query.

The optional value for -nulls is a string of spaces and 'n' characters telling spi_execp which of the values are NULL's. If given, it must have exactly the length of the number of values.

The queryid is the ID returned by the spi_prepare call.

If there was a typelist given to spi_prepare, a Tcl list of values of exactly the same length must be given to spi_execp after the query. If the type list on spi_prepare was empty, this argument must be omitted.

If the query is a SELECT statement, the same as described for spi_exec happens for the loop-body and the variables for the fields selected.

Here's an example for a PL/Tcl function using a prepared plan:

```
CREATE FUNCTION t1_count(int4, int4) RETURNS int4 AS '
    if {![ info exists GD(plan) ]} {
        # prepare the saved plan on the first call
        set GD(plan) [ spi_prepare \\
                "SELECT count(*) AS cnt FROM t1 WHERE num >= \\
$1 AND num <= \\$2" \\
                int4 ]
    }
    spi_execp -count 1 $GD(plan) [ list $1 $2 ]
    return $cnt
' LANGUAGE 'pltcl';
```

Note that each backslash that Tcl should see must be doubled in the query creating the function, since the main parser processes backslashes too on CREATE FUNCTION. Inside the query string given to spi_prepare should really be dollar signs to mark the parameter positions and to not let $1 be substituted by the value given in the first function call.

Modules and the unknown command

PL/Tcl has a special support for things often used. It recognizes two magic tables, pltcl_modules and pltcl_modfuncs. If these exist, the module 'unknown' is loaded into the interpreter right after creation. Whenever an unknown Tcl procedure is called, the unknown proc is asked to check if the procedure is defined in one of the modules. If this is true, the module is loaded on demand. To enable this behavior, the PL/Tcl call handler must be compiled with -DPLTCL_UNKNOWN_SUPPORT set.

There are support scripts to maintain these tables in the modules subdirectory of the PL/Tcl source including the source for the unknown module that must get installed initially.

Chapter 12. PL/perl - Perl Procedural Language

This chapter describes how to compile, install and use PL/Perl.

12.1. Overview

PL/Perl allows you to write functions in the Perl scripting language which may be used in SQL queries as if they were built into Postgres.

The PL/Perl intepreter is a full Perl interpreter. However, certain operations have been disabled in order to increase the security level of the system.

In general, the operations that are restricted are those that interact with the environment. This includes filehandle operations, `require`, and `use` (for external modules).

It should be noted that this security is not absolute. Indeed, several Denial-of-Service attacks are still possible - memory exhaustion and endless loops are two.

12.2. Building and Installing

Assuming that the Postgres source tree is rooted at $PGSRC, then the Pl/perl source code is located in $PGSRC/src/pl/plperl.

To build, simply do the following:

```
cd $PGSRC/src/pl/plperl
perl Makefile.PL
make
```

This will create a shared library file. On a Linux system, it will be named plperl.so. The extension may differ on other systems.

The shared library should then be copied into the lib subdirectory of the postgres installation.

The createlang command is used to install the language into a database. If it is installed into template1, all future databases will have the language installed automatically.

12.3. Using PL/Perl

Assume you have the following table:

```
CREATE TABLE EMPLOYEE (
    name text,
    basesalary int4,
    bonus int4 );
```

In order to get the total compensation (base + bonus) we could define a function as follows:

```
CREATE FUNCTION totalcomp(int4, int4) RETURNS int4
    AS 'return $_[0] + $_[1]'
    LANGUAGE 'plperl';
```

Note that the arguments are passed to the function in `@_` as might be expected. Also, because of the quoting rules for the SQL creating the function, you may find yourself using the extended quoting functions (qq[], q[], qw[]) more often that you are used to.

We may now use our function like so:

```
SELECT name, totalcomp(basesalary, bonus) from employee
```

But, we can also pass entire tuples to our function:

```
CREATE FUNCTION empcomp(employee) returns int4
    AS 'my $emp = shift;
        return $emp->{''basesalary''} + $emp->{''bonus''};'
    LANGUAGE 'plperl';
```

A tuple is passed as a reference to hash. The keys are the names of fields in the tuples. The values are values of the corresponding field in the tuple.

The new function `empcomp` can used like:

```
SELECT name, empcomp(employee) from employee;
```

Chapter 13. Multi-Version Concurrency Control

Multi-Version Concurrency Control (MVCC) is an advanced technique for improving database performance in a multi-user environment. Vadim Mikheev[1] provided the implementation for Postgres.

13.1. Introduction

Unlike most other database systems which use locks for concurrency control, Postgres maintains data consistency by using a multiversion model. This means that while querying a database each transaction sees a snapshot of data (a *database version*) as it was some time ago, regardless of the current state of the underlying data. This protects the transaction from viewing inconsistent data that could be caused by (other) concurrent transaction updates on the same data rows, providing *transaction isolation* for each database session.

The main difference between multiversion and lock models is that in MVCC locks acquired for querying (reading) data don't conflict with locks acquired for writing data and so reading never blocks writing and writing never blocks reading.

13.2. Transaction Isolation

The ANSI/ISO SQL standard defines four levels of transaction isolation in terms of three phenomena that must be prevented between concurrent transactions. These undesirable phenomena are:

dirty reads

A transaction reads data written by concurrent uncommitted transaction.

non-repeatable reads

A transaction re-reads data it has previously read and finds that data has been modified by another committed transaction.

phantom read

A transaction re-executes a query returning a set of rows that satisfy a search condition and finds that additional rows satisfying the condition has been inserted by another committed transaction.

The four isolation levels and the corresponding behaviors are described below.

Table 13.1. Postgres Isolation Levels

	Dirty Read	**Non-Repeatable Read**	**Phantom Read**
Read uncommitted	Possible	Possible	Possible
Read committed	Not possible	Possible	Possible
Repeatable read	Not possible	Not possible	Possible
Serializable	Not possible	Not possible	Not possible

Postgres offers the read committed and serializable isolation levels.

[1] mailto:vadim@krs.ru

13.3. Read Committed Isolation Level

Read Committed is the default isolation level in Postgres. When a transaction runs on this isolation level, a query sees only data committed before the query began and never sees either dirty data or concurrent transaction changes committed during query execution.

If a row returned by a query while executing an `UPDATE` statement (or `DELETE` or `SELECT FOR UPDATE`) is being updated by a concurrent uncommitted transaction then the second transaction that tries to update this row will wait for the other transaction to commit or rollback. In the case of rollback, the waiting transaction can proceed to change the row. In the case of commit (and if the row still exists; i.e. was not deleted by the other transaction), the query will be re-executed for this row to check that new row version satisfies query search condition. If the new row version satisfies the query search condition then row will be updated (or deleted or marked for update).

Note that the results of execution of `SELECT` or `INSERT` (with a query) statements will not be affected by concurrent transactions.

13.4. Serializable Isolation Level

Serializable provides the highest transaction isolation. When a transaction is on the serializable level, a query sees only data committed before the transaction began and never see either dirty data or concurrent transaction changes committed during transaction execution. So, this level emulates serial transaction execution, as if transactions would be executed one after another, serially, rather than concurrently.

If a row returned by query while executing a `UPDATE` (or `DELETE` or `SELECT FOR UPDATE`) statement is being updated by a concurrent uncommitted transaction then the second transaction that tries to update this row will wait for the other transaction to commit or rollback. In the case of rollback, the waiting transaction can proceed to change the row. In the case of a concurrent transaction commit, a serializable transaction will be rolled back with the message

```
ERROR:  Can't serialize access due to concurrent update
```

because a serializable transaction cannot modify rows changed by other transactions after the serializable transaction began.

Note

Note that results of execution of `SELECT` or `INSERT` (with a query) will not be affected by concurrent transactions.

13.5. Locking and Tables

Postgres provides various lock modes to control concurrent access to data in tables. Some of these lock modes are acquired by Postgres automatically before statement execution, while others are provided to be used by applications. All lock modes (except for AccessShareLock) acquired in a transaction are held for the duration of the transaction.

In addition to locks, short-term share/exclusive latches are used to control read/write access to table pages in shared buffer pool. Latches are released immediately after a tuple is fetched or updated.

13.5.1. Table-level locks

AccessShareLock

An internal lock mode acquiring automatically over tables being queried. Postgres releases these locks after statement is done.

Conflicts with AccessExclusiveLock only.

RowShareLock

Acquired by `SELECT FOR UPDATE` and `LOCK TABLE` for `IN ROW SHARE MODE` statements.

Conflicts with ExclusiveLock and AccessExclusiveLock modes.

RowExclusiveLock

Acquired by `UPDATE`, `DELETE`, `INSERT` and `LOCK TABLE` for `IN ROW EXCLUSIVE MODE` statements.

Conflicts with ShareLock, ShareRowExclusiveLock, ExclusiveLock and AccessExclusiveLock modes.

ShareLock

Acquired by `CREATE INDEX` and `LOCK TABLE` table for `IN SHARE MODE` statements.

Conflicts with RowExclusiveLock, ShareRowExclusiveLock, ExclusiveLock and AccessExclusiveLock modes.

ShareRowExclusiveLock

Acquired by `LOCK TABLE` for `IN SHARE ROW EXCLUSIVE MODE` statements.

Conflicts with RowExclusiveLock, ShareLock, ShareRowExclusiveLock, ExclusiveLock and AccessExclusiveLock modes.

ExclusiveLock

Acquired by `LOCK TABLE` table for `IN EXCLUSIVE MODE` statements.

Conflicts with RowShareLock, RowExclusiveLock, ShareLock, ShareRowExclusiveLock, ExclusiveLock and AccessExclusiveLock modes.

AccessExclusiveLock

Acquired by `ALTER TABLE`, `DROP TABLE`, `VACUUM` and `LOCK TABLE` statements.

Conflicts with RowShareLock, RowExclusiveLock, ShareLock, ShareRowExclusiveLock, ExclusiveLock and AccessExclusiveLock modes.

Note

Only AccessExclusiveLock blocks `SELECT` (without `FOR UPDATE`) statement.

13.5.2. Row-level locks

These locks are acquired when internal fields of a row are being updated (or deleted or marked for update). Postgres doesn't remember any information about modified rows in memory and so has no limit to the number of rows locked without lock escalation.

However, take into account that `SELECT FOR UPDATE` will modify selected rows to mark them and so will results in disk writes.

Row-level locks don't affect data querying. They are used to block writers to *the same row* only.

13.6. Locking and Indices

Though Postgres provides unblocking read/write access to table data, unblocked read/write access is not provided for every index access methods implemented in Postgres.

The various index types are handled as follows:

GiST and R-Tree indices

Share/exclusive index-level locks are used for read/write access. Locks are released after statement is done.

Hash indices

Share/exclusive page-level locks are used for read/write access. Locks are released after page is processed.

Page-level locks produces better concurrency than index-level ones but are subject to deadlocks.

Btree

Short-term share/exclusive page-level latches are used for read/write access. Latches are released immediately after the index tuple is inserted/fetched.

Btree indices provide the highest concurrency without deadlock conditions.

13.7. Data consistency checks at the application level

Because readers in Postgres don't lock data, regardless of transaction isolation level, data read by one transaction can be overwritten by another. In the other words, if a row is returned by `SELECT` it doesn't mean that this row really exists at the time it is returned (i.e. sometime after the statement or transaction began) nor that the row is protected from deletion or update by concurrent transactions before the current transaction does a commit or rollback.

To ensure the actual existance of a row and protect it against concurrent updates one must use `SELECT FOR UPDATE` or an appropriate `LOCK TABLE` statement. This should be taken into account when porting applications using serializable mode to Postgres from other environments.

Note

Before version 6.5 Postgres used read-locks and so the above consideration is also the case when upgrading to 6.5 (or higher) from previous Postgres versions.

Chapter 14. Setting Up Your Environment

This section discusses how to set up your own environment so that you can use frontend applications. We assume Postgres has already been successfully installed and started; refer to the Administrator's Guide and the installation notes for how to install Postgres.

Postgres is a client/server application. As a user, you only need access to the client portions of the installation (an example of a client application is the interactive monitor psql). For simplicity, we will assume that Postgres has been installed in the directory `/usr/local/pgsql`. Therefore, wherever you see the directory `/usr/local/pgsql` you should substitute the name of the directory where Postgres is actually installed. All Postgres commands are installed in the directory `/usr/local/pgsql/bin`. Therefore, you should add this directory to your shell command path. If you use a variant of the Berkeley C shell, such as csh or tcsh, you would add

```
set path = ( /usr/local/pgsql/bin path )
```

in the `.login` file in your home directory. If you use a variant of the Bourne shell, such as sh, ksh, or bash, then you would add

```
$ PATH=/usr/local/pgsql/bin:$PATH
$ export PATH
```

to the `.profile` file in your home directory. From now on, we will assume that you have added the Postgres bin directory to your path. In addition, we will make frequent reference to "setting a shell variable" or "setting an environment variable" throughout this document. If you did not fully understand the last paragraph on modifying your search path, you should consult the Unix manual pages that describe your shell before going any further.

If your site administrator has not set things up in the default way, you may have some more work to do. For example, if the database server machine is a remote machine, you will need to set the PGHOST environment variable to the name of the database server machine. The environment variable PGPORT may also have to be set. The bottom line is this: if you try to start an application program and it complains that it cannot connect to the postmaster, you should immediately consult your site administrator to make sure that your environment is properly set up.

Chapter 15. Managing a Database

Note

This section is currently a thinly disguised copy of the Tutorial. Needs to be augmented. - thomas 1998-01-12

Although the *site administrator* is responsible for overall management of the Postgres installation, some databases within the installation may be managed by another person, designated the *database administrator*. This assignment of responsibilities occurs when a database is created. A user may be assigned explicit privileges to create databases and/or to create new users. A user assigned both privileges can perform most administrative task within Postgres, but will not by default have the same operating system privileges as the site administrator.

The Database Administrator's Guide covers these topics in more detail.

15.1. Database Creation

Databases are created by the `create database` issued from within Postgres. createdb is a command-line utility provided to give the same functionality from outside Postgres.

The Postgres backend must be running for either method to succeed, and the user issuing the command must be the Postgres *superuser* or have been assigned database creation privileges by the superuser.

To create a new database named `mydb` from the command line, type

```
% createdb mydb
```

and to do the same from within psql type

```
=> CREATE DATABASE mydb;
```

If you do not have the privileges required to create a database, you will see the following:

```
ERROR:  CREATE DATABASE: Permission denied.
```

Postgres allows you to create any number of databases at a given site and you automatically become the database administrator of the database you just created. Database names must have an alphabetic first character and are limited to 32 characters in length.

15.2. Alternate Database Locations

It is possible to create a database in a location other than the default location for the installation. Remember that all database access actually occurs through the database backend, so that any location specified must be accessible by the backend.

Alternate database locations are created and referenced by an environment variable which gives the absolute path to the intended storage location. This environment variable must have been defined before the backend was started and the location it points to must be writable by the postgres administrator account. Consult with the site administrator regarding preconfigured alternate database locations. Any valid environment variable name may be used to reference an alternate location, although using variable names with a prefix of `PGDATA` is recommended to avoid confusion and conflict with other variables.

Note

In previous versions of Postgres, it was also permissable to use an absolute path name to specify an alternate storage location. Although the environment variable style of specification

is to be preferred since it allows the site administrator more flexibility in managing disk storage, it is also possible to use an absolute path to specify an alternate location. The administrator's guide discusses how to enable this feature.

For security and integrity reasons, any path or environment variable specified has some additional path fields appended. Alternate database locations must be prepared by running initlocation.

To create a data storage area using the environment variable `PGDATA2` (for this example set to `/alt/postgres`), ensure that `/alt/postgres` already exists and is writable by the Postgres administrator account. Then, from the command line, type

```
% initlocation PGDATA2
Creating Postgres database system directory /alt/postgres/data
Creating Postgres database system directory /alt/postgres/data/base
```

To create a database in the alternate storage area `PGDATA2` from the command line, use the following command:

```
% createdb -D PGDATA2 mydb
```

and to do the same from within psql type

```
=> CREATE DATABASE mydb WITH LOCATION = 'PGDATA2';
```

If you do not have the privileges required to create a database, you will see the following:

```
ERROR:  CREATE DATABASE: permission denied
```

If the specified location does not exist or the database backend does not have permission to access it or to write to directories under it, you will see the following:

```
ERROR:  The database path '/no/where' is invalid. This may be due
 to a character that is not allowed or because the chosen path
 isn't permitted for databases.
```

15.3. Accessing a Database

Once you have constructed a database, you can access it by:

- running the PostgreSQL interactive terminal psql which allows you to interactively enter, edit, and execute SQL commands.
- writing a C program using the LIBPQ subroutine library. This allows you to submit SQL commands from C and get answers and status messages back to your program. This interface is discussed further in *The PostgreSQL Programmer's Guide*.

You might want to start up psql, to try out the examples in this manual. It can be activated for the mydb database by typing the command:

```
% psql mydb
```

You will be greeted with the following message:

```
Welcome to psql, the PostgreSQL interactive terminal.

Type:  \copyright for distribution terms
       \h for help with SQL commands
       \? for help on internal slash commands
       \g or terminate with semicolon to execute query
       \q to quit
```

```
mydb=>
```

This prompt indicates that psql is listening to you and that you can type SQL queries into a workspace maintained by the terminal monitor. The psql program responds to escape codes that begin with the backslash character, "\". For example, you can get help on the syntax of various PostgreSQL SQL commands by typing:

```
mydb=> \h
```

Once you have finished entering your queries into the workspace, you can pass the contents of the workspace to the Postgres server by typing:

```
mydb=> \g
```

This tells the server to process the query. If you terminate your query with a semicolon, the "\g" is not necessary. psql will automatically process semicolon terminated queries. To read queries from a file, say myFile, instead of entering them interactively, type:

```
mydb=> \i fileName
```

To get out of psql and return to Unix, type

```
mydb=> \q
```

and psql will quit and return you to your command shell. (For more escape codes, type `\?` at the psql prompt.) White space (i.e., spaces, tabs and newlines) may be used freely in SQL queries. Single-line comments are denoted by "`--`". Everything after the dashes up to the end of the line is ignored. Multiple-line comments, and comments within a line, are denoted by "`/* ... */`".

15.3.1. Database Privileges

15.3.2. Table Privileges

TBD

15.4. Destroying a Database

If you are the owner of the database mydb, you can destroy it using the following Unix command:

```
% dropdb mydb
```

This action physically removes all of the Unix files associated with the database and cannot be undone, so this should only be done with a great deal of forethought.

Chapter 16. Disk Storage

This section needs to be written. Some information is in the FAQ. Volunteers? - thomas 1998-01-11

Chapter 17. Understanding Performance

Query performance can be affected by many things. Some of these can be manipulated by the user, while others are fundamental to the underlying design of the system.

Some performance issues, such as index creation and bulk data loading, are covered elsewhere. This chapter will discuss the `EXPLAIN` command, and will show how the details of a query can affect the query plan, and hence overall performance.

17.1. Using `EXPLAIN`

Author

Written by Tom Lane, from e-mail dated 2000-03-27.

Plan-reading is an art that deserves a tutorial, and I haven't had time to write one. Here is some quick & dirty explanation.

The numbers that are currently quoted by EXPLAIN are:

- Estimated startup cost (time expended before output scan can start, eg, time to do the sorting in a SORT node).
- Estimated total cost (if all tuples are retrieved, which they may not be --- LIMIT will stop short of paying the total cost, for example).
- Estimated number of rows output by this plan node.
- Estimated average width (in bytes) of rows output by this plan node.

The costs are measured in units of disk page fetches. (CPU effort estimates are converted into disk-page units using some fairly arbitrary fudge-factors. See the `SET` reference page if you want to experiment with these.) It's important to note that the cost of an upper-level node includes the cost of all its child nodes. It's also important to realize that the cost only reflects things that the planner/optimizer cares about. In particular, the cost does not consider the time spent transmitting result tuples to the frontend --- which could be a pretty dominant factor in the true elapsed time, but the planner ignores it because it cannot change it by altering the plan. (Every correct plan will output the same tuple set, we trust.)

Rows output is a little tricky because it is *not* the number of rows processed/scanned by the query --- it is usually less, reflecting the estimated selectivity of any WHERE-clause constraints that are being applied at this node.

Average width is pretty bogus because the thing really doesn't have any idea of the average length of variable-length columns. I'm thinking about improving that in the future, but it may not be worth the trouble, because the width isn't used for very much.

Here are some examples (using the regress test database after a vacuum analyze, and almost-7.0 sources):

```
regression=# explain select * from tenk1;
NOTICE:  QUERY PLAN:

Seq Scan on tenk1  (cost=0.00..333.00 rows=10000 width=148)
```

This is about as straightforward as it gets. If you do

```
select * from pg_class where relname = 'tenk1';
```

you'll find out that tenk1 has 233 disk pages and 10000 tuples. So the cost is estimated at 233 block reads, defined as 1.0 apiece, plus 10000 * cpu_tuple_cost which is currently 0.01 (try `show cpu_tuple_cost`).

Now let's modify the query to add a qualification clause:

```
regression=# explain select * from tenk1 where unique1 < 1000;
NOTICE:  QUERY PLAN:

Seq Scan on tenk1  (cost=0.00..358.00 rows=1000 width=148)
```

The estimate of output rows has gone down because of the WHERE clause. (The uncannily accurate estimate is just because tenk1 is a particularly simple case --- the unique1 column has 10000 distinct values ranging from 0 to 9999, so the estimator's linear interpolation between min and max column values is dead-on.) However, the scan will still have to visit all 10000 rows, so the cost hasn't decreased; in fact it has gone up a bit to reflect the extra CPU time spent checking the WHERE condition.

Modify the query to restrict the qualification even more:

```
regression=# explain select * from tenk1 where unique1 < 100;
NOTICE:  QUERY PLAN:

Index Scan using tenk1_unique1 on tenk1  (cost=0.00..89.35 rows=100
 width=148)
```

and you will see that if we make the WHERE condition selective enough, the planner will eventually decide that an indexscan is cheaper than a sequential scan. This plan will only have to visit 100 tuples because of the index, so it wins despite the fact that each individual fetch is expensive.

Add another condition to the qualification:

```
regression=# explain select * from tenk1 where unique1 < 100 and
regression-# stringu1 = 'xxx';
NOTICE:  QUERY PLAN:

Index Scan using tenk1_unique1 on tenk1  (cost=0.00..89.60 rows=1
 width=148)
```

The added clause "stringu1 = 'xxx'" reduces the output-rows estimate, but not the cost because we still have to visit the same set of tuples.

Let's try joining two tables, using the fields we have been discussing:

```
regression=# explain select * from tenk1 t1, tenk2 t2 where
 t1.unique1 < 100
regression-# and t1.unique2 = t2.unique2;
NOTICE:  QUERY PLAN:

Nested Loop  (cost=0.00..144.07 rows=100 width=296)
  ->  Index Scan using tenk1_unique1 on tenk1 t1
              (cost=0.00..89.35 rows=100 width=148)
  ->  Index Scan using tenk2_unique2 on tenk2 t2
              (cost=0.00..0.53 rows=1 width=148)
```

In this nested-loop join, the outer scan is the same indexscan we had in the example before last, and so its cost and row count are the same because we are applying the "unique1 < 100" WHERE clause at that node. The "t1.unique2 = t2.unique2" clause isn't relevant yet, so it doesn't affect the outer scan's row count. For the inner scan, the current outer-scan tuple's unique2 value is plugged into the inner indexscan to produce an indexqual like "t2.unique2 = *constant*". So we get the same inner-scan plan and costs that we'd get from, say, "explain select * from tenk2 where unique2 = 42". The loop node's costs are then set on the basis of the outer scan's cost, plus one repetition of the inner scan for each outer tuple (100 * 0.53, here), plus a little CPU time for join processing.

In this example the loop's output row count is the same as the product of the two scans' row counts, but that's not true in general, because in general you can have WHERE clauses that mention both relations and so can only be applied at the join point, not to either input scan. For example, if we added "WHERE ... AND t1.hundred < t2.hundred", that'd decrease the output row count of the join node, but not change either input scan.

We can look at variant plans by forcing the planner to disregard whatever strategy it thought was the winner (a pretty crude tool, but it's what we've got at the moment):

```
regression=# set enable_nestloop = off;
SET VARIABLE
regression=# explain select * from tenk1 t1, tenk2 t2 where
 t1.unique1 < 100
regression-# and t1.unique2 = t2.unique2;
NOTICE:  QUERY PLAN:

Hash Join  (cost=89.60..574.10 rows=100 width=296)
  ->  Seq Scan on tenk2 t2
              (cost=0.00..333.00 rows=10000 width=148)
  ->  Hash  (cost=89.35..89.35 rows=100 width=148)
        ->  Index Scan using tenk1_unique1 on tenk1 t1
              (cost=0.00..89.35 rows=100 width=148)
```

This plan proposes to extract the 100 interesting rows of tenk1 using ye same olde indexscan, stash them into an in-memory hash table, and then do a sequential scan of tenk2, probing into the hash table for possible matches of "t1.unique2 = t2.unique2" at each tenk2 tuple. The cost to read tenk1 and set up the hash table is entirely startup cost for the hash join, since we won't get any tuples out until we can start reading tenk2. The total time estimate for the join also includes a pretty hefty charge for CPU time to probe the hash table 10000 times. Note, however, that we are NOT charging 10000 times 89.35; the hash table setup is only done once in this plan type.

Chapter 18. Populating a Database

Author

Written by Tom Lane, from an e-mail message dated 1999-12-05.

One may need to do a large number of table insertions when first populating a database. Here are some tips and techniques for making that as efficient as possible.

18.1. Disable Auto-commit

Turn off auto-commit and just do one commit at the end. Otherwise Postgres is doing a lot of work for each record added. In general when you are doing bulk inserts, you want to turn off some of the database features to gain speed.

18.2. Use COPY FROM

Use `COPY FROM STDIN` to load all the records in one command, instead of a series of INSERT commands. This reduces parsing, planning, etc overhead a great deal. If you do this then it's not necessary to fool around with autocommit.

18.3. Remove Indices

If you are loading a freshly created table, the fastest way is to create the table, bulk-load with COPY, then create any indexes needed for the table. Creating an index on pre-existing data is quicker than updating it incrementally as each record is loaded.

If you are augmenting an existing table, you can `DROP INDEX`, load the table, then recreate the index. Of course, the database performance for other users may be adversely affected during the time that the index is missing.

Chapter 19. SQL Commands

This is reference information for the SQL commands supported by Postgres.

Name

ABORT — Aborts the current transaction

Synopsis

<refsynopsisdivinfo>1999-07-20</refsynopsisdivinfo>

```
ABORT [ WORK | TRANSACTION ]
```

Inputs

None.

Outputs

`ROLLBACK`

Message returned if successful.

`NOTICE: ROLLBACK: no transaction in progress`

If there is not any transaction currently in progress.

Description

`ABORT` rolls back the current transaction and causes all the updates made by the transaction to be discarded. This command is identical in behavior to the SQL92 command `ROLLBACK`, and is present only for historical reasons.

Notes

Use `COMMIT` to successfully terminate a transaction.

Usage

To abort all changes:

```
ABORT WORK;
```

Compatibility

SQL92

This command is a Postgres extension present for historical reasons. `ROLLBACK` is the SQL92 equivalent command.

Name

ALTER GROUP — Add users to a group, remove users from a group

Synopsis

<refsynopsisdivinfo>2000-01-14</refsynopsisdivinfo>

```
ALTER GROUP name ADD USER username [, ... ]
ALTER GROUP name DROP USER username [, ... ]
```

Inputs

name

The name of the group to modify.

username

Users which are to be added or removed from the group. The user names must exist.

Outputs

`ALTER GROUP`

Message returned if the alteration was successful.

Description

`ALTER GROUP` is used to change add users to a group or remove them from a group. Only database superusers can use this command. Adding a user to a group does not create the user. Similarly, removing a user from a group does not drop the user itself.

Use CREATE GROUP to create a new group and DROP GROUP to remove a group.

Usage

Add users to a group:

```
ALTER GROUP staff ADD USER karl, john
```

Remove a user from a group

```
ALTER GROUP workers DROP USER beth
```

Compatibility

SQL92

There is no `ALTER GROUP` statement in SQL92. The concept of roles is similar.

Name

ALTER TABLE — Modifies table properties

Synopsis

<refsynopsisdivinfo>1999-07-20</refsynopsisdivinfo>

```
ALTER TABLE table [ * ]
    ADD [ COLUMN ] column type
ALTER TABLE table [ * ]
    ALTER [ COLUMN ] column { SET DEFAULT value | DROP DEFAULT }
ALTER TABLE table [ * ]
    RENAME [ COLUMN ] column TO newcolumn
ALTER TABLE table
    RENAME TO newtable
ALTER TABLE table
    ADD table constraint definition
```

Inputs

table

The name of an existing table to alter.

column

Name of a new or existing column.

type

Type of the new column.

newcolumn

New name for an existing column.

newtable

New name for the table.

table constraint definition

New table constraint for the table

Outputs

ALTER

Message returned from column or table renaming.

ERROR

Message returned if table or column is not available.

Description

ALTER TABLE changes the definition of an existing table. The ADD COLUMN form adds a new column to the table using the same syntax as CREATE TABLE . The ALTER COLUMN form allows you to set or remove the default for the column. Note that defaults only apply to newly inserted rows. The RENAME clause causes the name of a table or column to change without changing any of the data contained in the affected table. Thus, the table or column will remain of the same type and size

after this command is executed. The ADD *table constraint definition* clause adds a new constraint to the table using the same syntax as CREATE TABLE .

You must own the table in order to change its schema.

Notes

The keyword COLUMN is noise and can be omitted.

"*" following a name of a table indicates that the statement should be run over that table and all tables below it in the inheritance hierarchy; by default, the attribute will not be added to or renamed in any of the subclasses. This should always be done when adding or modifying an attribute in a superclass. If it is not, queries on the inheritance hierarchy such as

```
SELECT NewColumn FROM SuperClass*
```

will not work because the subclasses will be missing an attribute found in the superclass.

In the current implementation, default and constraint clauses for the new column will be ignored. You can use the SET DEFAULT form of ALTER TABLE to set the default later. (You will also have to update the already existing rows to the new default value, using UPDATE .)

In the current implementation, only FOREIGN KEY constraints can be added to a table. To create or remove a unique constraint, create a unique index (see CREATE INDEX). To add check constraints you need to recreate and reload the table, using other parameters to the CREATE TABLE command.

You must own the class in order to change its schema. Renaming any part of the schema of a system catalog is not permitted. The *PostgreSQL User's Guide* has further information on inheritance.

Refer to CREATE TABLE for a further description of valid arguments.

Usage

To add a column of type VARCHAR to a table:

```
ALTER TABLE distributors ADD COLUMN address VARCHAR(30);
```

To rename an existing column:

```
ALTER TABLE distributors RENAME COLUMN address TO city;
```

To rename an existing table:

```
ALTER TABLE distributors RENAME TO suppliers;
```

To add a foreign key constraint to a table:

```
ALTER TABLE distributors ADD CONSTRAINT distfk FOREIGN KEY
 (address) REFERENCES addresses(address) MATCH FULL
```

Compatibility

SQL92

The ADD COLUMN form is compliant with the exception that it does not support defaults and constraints, as explained above. The ALTER COLUMN form is in full compliance.

SQL92 specifies some additional capabilities for `ALTER TABLE` statement which are not yet directly supported by Postgres:

```
ALTER TABLE table DROP CONSTRAINT constraint { RESTRICT |
 CASCADE }
```

Removes a table constraint (such as a check constraint, unique constraint, or foreign key constraint). To remove a unique constraint, drop a unique index, To remove other kinds of constraints you need to recreate and reload the table, using other parameters to the CREATE TABLE command.

For example, to drop any constraints on a table `distributors`:

```
CREATE TABLE temp AS SELECT * FROM distributors;
DROP TABLE distributors;
CREATE TABLE distributors AS SELECT * FROM temp;
DROP TABLE temp;
```

```
ALTER TABLE table DROP [ COLUMN ] column { RESTRICT |
 CASCADE }
```

Removes a column from a table. Currently, to remove an existing column the table must be recreated and reloaded:

```
CREATE TABLE temp AS SELECT did, city FROM distributors;
DROP TABLE distributors;
CREATE TABLE distributors (
    did      DECIMAL(3)  DEFAULT 1,
    name     VARCHAR(40) NOT NULL,
);
INSERT INTO distributors SELECT * FROM temp;
DROP TABLE temp;
```

The clauses to rename columns and tables are Postgres extensions from SQL92.

Name

ALTER USER — Modifies user account information

Synopsis

<refsynopsisdivinfo>1999-07-20</refsynopsisdivinfo>

```
ALTER USER username
    [ WITH PASSWORD 'password' ]
    [ CREATEDB | NOCREATEDB ] [ CREATEUSER | NOCREATEUSER ]
    [ VALID UNTIL 'abstime' ]
```

Inputs

username

The name of the user whose details are to be altered.

password

The new password to be used for this account.

CREATEDB
NOCREATEDB

These clauses define a user's ability to create databases. If CREATEDB is specified, the user being defined will be allowed to create his own databases. Using NOCREATEDB will deny a user the ability to create databases.

CREATEUSER
NOCREATEUSER

These clauses determine whether a user will be permitted to create new users himself. This option will also make the user a superuser who can override all access restrictions.

abstime

The date (and, optionally, the time) at which this user's password is to expire.

Outputs

`ALTER USER`

Message returned if the alteration was successful.

`ERROR: ALTER USER: user "username" does not exist`

Error message returned if the specified user is not known to the database.

Description

`ALTER USER` is used to change the attributes of a user's Postgres account. Only a database superuser can change privileges and password expiration with this command. Ordinary users can only change their own password.

Use CREATE USER to create a new user and DROP USER to remove a user.

Usage

Change a user password:

```
ALTER USER davide WITH PASSWORD 'hu8jmn3';
```

Change a user's valid until date

```
ALTER USER manuel VALID UNTIL 'Jan 31 2030';
```

Change a user's valid until date, specifying that his authorisation should expire at midday on 4th May 1998 using the time zone which is one hour ahead of UTC

```
ALTER USER chris VALID UNTIL 'May 4 12:00:00 1998 +1';
```

Give a user the ability to create other users and new databases.

```
ALTER USER miriam CREATEUSER CREATEDB;
```

Compatibility

SQL92

There is no `ALTER USER` statement in SQL92. The standard leaves the definition of users to the implementation.

Name

BEGIN — Begins a transaction in chained mode

Synopsis

<refsynopsisdivinfo>1999-07-20</refsynopsisdivinfo>

```
BEGIN [ WORK | TRANSACTION ]
```

Inputs

WORK
TRANSACTION

Optional keywords. They have no effect.

Outputs

`BEGIN`

This signifies that a new transaction has been started.

`NOTICE: BEGIN: already a transaction in progress`

This indicates that a transaction was already in progress. The current transaction is not affected.

Description

By default, Postgres executes transactions in *unchained mode* (also known as "autocommit" in other database systems). In other words, each user statement is executed in its own transaction and a commit is implicitly performed at the end of the statement (if execution was successful, otherwise a rollback is done). `BEGIN` initiates a user transaction in chained mode, i.e. all user statements after `BEGIN` command will be executed in a single transaction until an explicit COMMIT , ROLLBACK , or execution abort. Statements in chained mode are executed much faster, because transaction start/commit requires significant CPU and disk activity. Execution of multiple statements inside a transaction is also required for consistency when changing several related tables.

The default transaction isolation level in Postgres is READ COMMITTED, where queries inside the transaction see only changes committed before query execution. So, you have to use `SET TRANSACTION ISOLATION LEVEL SERIALIZABLE` just after `BEGIN` if you need more rigorous transaction isolation. In SERIALIZABLE mode queries will see only changes committed before the entire transaction began (actually, before execution of the first DML statement in a serializable transaction).

If the transaction is committed, Postgres will ensure either that all updates are done or else that none of them are done. Transactions have the standard ACID (atomic, consistent, isolatable, and durable) property.

Notes

Refer to LOCK for further information about locking tables inside a transaction.

Use COMMIT or ROLLBACK to terminate a transaction.

Usage

To begin a user transaction:

```
BEGIN WORK;
```

Compatibility

SQL92

`BEGIN` is a Postgres language extension. There is no explicit `BEGIN` command in SQL92; transaction initiation is always implicit and it terminates either with a `COMMIT` or `ROLLBACK` statement.

Note

Many relational database systems offer an autocommit feature as a convenience.

Incidentally, the `BEGIN` keyword is used for a different purpose in embedded SQL. You are advised to be careful about the transaction semantics when porting database applications.

SQL92 also requires SERIALIZABLE to be the default transaction isolation level.

Name

CLOSE — Close a cursor

Synopsis

<refsynopsisdivinfo>1999-07-20</refsynopsisdivinfo>

```
CLOSE cursor
```

Inputs

cursor

The name of an open cursor to close.

Outputs

`CLOSE`

Message returned if the cursor is successfully closed.

`NOTICE PerformPortalClose: portal "`*cursor*`" not found`

This warning is given if *cursor* is not declared or has already been closed.

Description

`CLOSE` frees the resources associated with an open cursor. After the cursor is closed, no subsequent operations are allowed on it. A cursor should be closed when it is no longer needed.

An implicit close is executed for every open cursor when a transaction is terminated by `COMMIT` or `ROLLBACK`.

Notes

Postgres does not have an explicit `OPEN` cursor statement; a cursor is considered open when it is declared. Use the `DECLARE` statement to declare a cursor.

Usage

Close the cursor liahona:

```
CLOSE liahona;
```

Compatibility

SQL92

`CLOSE` is fully compatible with SQL92.

Name

CLUSTER — Gives storage clustering advice to the server

Synopsis

<refsynopsisdivinfo>1999-07-20</refsynopsisdivinfo>

```
CLUSTER indexname ON table
```

Inputs

indexname

The name of an index.

table

The name of a table.

Outputs

CLUSTER

The clustering was done successfully.

```
ERROR: relation <tablerelation_number> inherits "table"
```

```
ERROR: Relation table does not exist!
```

Description

CLUSTER instructs Postgres to cluster the class specified by *table* approximately based on the index specified by *indexname*. The index must already have been defined on *classname*.

When a class is clustered, it is physically reordered based on the index information. The clustering is static. In other words, as the class is updated, the changes are not clustered. No attempt is made to keep new instances or updated tuples clustered. If one wishes, one can recluster manually by issuing the command again.

Notes

The table is actually copied to a temporary table in index order, then renamed back to the original name. For this reason, all grant permissions and other indexes are lost when clustering is performed.

In cases where you are accessing single rows randomly within a table, the actual order of the data in the heap table is unimportant. However, if you tend to access some data more than others, and there is an index that groups them together, you will benefit from using CLUSTER.

Another place where CLUSTER is helpful is in cases where you use an index to pull out several rows from a table. If you are requesting a range of indexed values from a table, or a single indexed value that has multiple rows that match, CLUSTER will help because once the index identifies the heap page for the first row that matches, all other rows that match are probably already on the same heap page, saving disk accesses and speeding up the query.

There are two ways to cluster data. The first is with the CLUSTER command, which reorders the original table with the ordering of the index you specify. This can be slow on large tables because the

rows are fetched from the heap in index order, and if the heap table is unordered, the entries are on random pages, so there is one disk page retrieved for every row moved. Postgres has a cache, but the majority of a big table will not fit in the cache.

Another way to cluster data is to use

```
SELECT columnlist INTO TABLE newtable
     FROM table ORDER BY columnlist
```

which uses the Postgres sorting code in the ORDER BY clause to match the index, and which is much faster for unordered data. You then drop the old table, use `ALTER TABLE/RENAME` to rename *`temp`* to the old name, and recreate any indexes. The only problem is that OIDs will not be preserved. From then on, `CLUSTER` should be fast because most of the heap data has already been ordered, and the existing index is used.

Usage

Cluster the employees relation on the basis of its salary attribute

```
CLUSTER emp_ind ON emp;
```

Compatibility

SQL92

There is no `CLUSTER` statement in SQL92.

Name

COMMENT — Add comment to an object

Synopsis

<refsynopsisdivinfo>1999-07-20</refsynopsisdivinfo>

```
COMMENT ON
[
  [ DATABASE | INDEX | RULE | SEQUENCE | TABLE | TYPE | VIEW ]
  object_name |
  COLUMN table_name.column_name|
  AGGREGATE agg_name agg_type|
  FUNCTION func_name (arg1, arg2, ...)|
  OPERATOR op (leftoperand_type rightoperand_type) |
  TRIGGER trigger_name ON table_name
] IS 'text'
```

Inputs

object_name, *table_name*, *column_name*, *agg_name*, *func_name*, *op*, *trigger_name*

The name of the object to be be commented.

text

The comment to add.

Outputs

COMMENT

Message returned if the table is successfully commented.

Description

COMMENT adds a comment to an object that can be easily retrieved with psql's *\dd* command. To remove a comment, use NULL. Comments are automatically dropped when the object is dropped.

Usage

Comment the table mytable:

```
COMMENT ON mytable IS 'This is my table.';
```

Some more examples:

```
COMMENT ON DATABASE my_database IS 'Development Database';
COMMENT ON INDEX my_index IS 'Enforces uniqueness on employee id';
COMMENT ON RULE my_rule IS 'Logs UPDATES of employee records';
COMMENT ON SEQUENCE my_sequence IS 'Used to generate primary keys';
COMMENT ON TABLE my_table IS 'Employee Information';
COMMENT ON TYPE my_type IS 'Complex Number support';
COMMENT ON VIEW my_view IS 'View of departmental costs';
COMMENT ON COLUMN my_table.my_field IS 'Employee ID number';
```

```
COMMENT ON AGGREGATE my_aggregate float8 IS 'Computes sample
 variance';
COMMENT ON FUNCTION my_function (datetime) IS 'Returns Roman
 Numeral';
COMMENT ON OPERATOR ^ (text, text) IS 'Performs intersection of two
 text';
COMMENT ON TRIGGER my_trigger ON my_table IS 'Used for R.I.';
```

Compatibility

SQL92

There is no `COMMENT` in SQL92.

Name

COMMIT — Commits the current transaction

Synopsis

<refsynopsisdivinfo>1999-07-20</refsynopsisdivinfo>

```
COMMIT [ WORK | TRANSACTION ]
```

Inputs

WORK
TRANSACTION

Optional keywords. They have no effect.

Outputs

`COMMIT`

Message returned if the transaction is successfully committed.

```
NOTICE: COMMIT: no transaction in progress
```

If there is no transaction in progress.

Description

`COMMIT` commits the current transaction. All changes made by the transaction become visible to others and are guaranteed to be durable if a crash occurs.

Notes

The keywords WORK and TRANSACTION are noise and can be omitted.

Use ROLLBACK to abort a transaction.

Usage

To make all changes permanent:

```
COMMIT WORK;
```

Compatibility

SQL92

SQL92 only specifies the two forms `COMMIT` and `COMMIT WORK`. Otherwise full compatibility.

Name

COPY — Copies data between files and tables

Synopsis

<refsynopsisdivinfo>1999-12-11</refsynopsisdivinfo>

```
COPY [ BINARY ] table [ WITH OIDS ]
    FROM { 'filename' | stdin }
    [ [USING] DELIMITERS 'delimiter' ]
    [ WITH NULL AS 'null string' ]
COPY [ BINARY ] table [ WITH OIDS ]
    TO { 'filename' | stdout }
    [ [USING] DELIMITERS 'delimiter' ]
    [ WITH NULL AS 'null string' ]
```

Inputs

BINARY

Changes the behavior of field formatting, forcing all data to be stored or read in binary format rather than as text.

table

The name of an existing table.

WITH OIDS

Copies the internal unique object id (OID) for each row.

filename

The absolute Unix pathname of the input or output file.

stdin

Specifies that input comes from a pipe or terminal.

stdout

Specifies that output goes to a pipe or terminal.

delimiter

A character that delimits the input or output fields.

null print

A string to represent NULL values. The default is "\N" (backslash-N). You might prefer an empty string, for example.

Note

On a copy in, any data item that matches this string will be stored as a NULL value, so you should make sure that you use the same string as you used on copy out.

Outputs

COPY

The copy completed successfully.

ERROR: *reason*

The copy failed for the reason stated in the error message.

Description

COPY moves data between Postgres tables and standard file-system files. COPY instructs the Postgres backend to directly read from or write to a file. The file must be directly visible to the backend and the name must be specified from the viewpoint of the backend. If stdin or stdout are specified, data flows through the client frontend to the backend.

Notes

The BINARY keyword will force all data to be stored/read as binary format rather than as text. It is somewhat faster than the normal copy command, but is not generally portable, and the files generated are somewhat larger, although this factor is highly dependent on the data itself.

By default, a text copy uses a tab ("\t") character as a delimiter. The delimiter may also be changed to any other single character with the keyword phrase USING DELIMITERS. Characters in data fields which happen to match the delimiter character will be backslash quoted.

You must have *select access* on any table whose values are read by COPY, and either *insert or update access* to a table into which values are being inserted by COPY. The backend also needs appropriate Unix permissions for any file read or written by COPY.

The keyword phrase USING DELIMITERS specifies a single character to be used for all delimiters between columns. If multiple characters are specified in the delimiter string, only the first character is used.

Tip

Do not confuse COPY with the psql instruction \copy.

COPY neither invokes rules nor acts on column defaults. It does invoke triggers, however.

COPY stops operation at the first error. This should not lead to problems in the event of a COPY FROM, but the target relation will, of course, be partially modified in a COPY TO. VACUUM should be used to clean up after a failed copy.

Because the Postgres backend's current working directory is not usually the same as the user's working directory, the result of copying to a file "foo" (without additional path information) may yield unexpected results for the naive user. In this case, foo will wind up in $PGDATA/foo. In general, the full pathname as it would appear to the backend server machine should be used when specifying files to be copied.

Files used as arguments to COPY must reside on or be accessible to the database server machine by being either on local disks or on a networked file system.

When a TCP/IP connection from one machine to another is used, and a target file is specified, the target file will be written on the machine where the backend is running rather than the user's machine.

File Formats

Text Format

When COPY TO is used without the BINARY option, the file generated will have each row (instance) on a single line, with each column (attribute) separated by the delimiter character. Embedded delimiter characters will be preceded by a backslash character ("\"). The attribute values themselves are strings

generated by the output function associated with each attribute type. The output function for a type should not try to generate the backslash character; this will be handled by COPY itself.

The actual format for each instance is

```
<attr1><separator><attr2><separator>...<separator><attrn><newline>
```

The oid is placed on the beginning of the line if WITH OIDS is specified.

If COPY is sending its output to standard output instead of a file, it will send a backslash("\") and a period (".") followed immediately by a newline, on a separate line, when it is done. Similarly, if COPY is reading from standard input, it will expect a backslash ("\") and a period (".") followed by a newline, as the first three characters on a line to denote end-of-file. However, COPY will terminate (followed by the backend itself) if a true EOF is encountered before this special end-of-file pattern is found.

The backslash character has other special meanings. A literal backslash character is represented as two consecutive backslashes ("\\"). A literal tab character is represented as a backslash and a tab. A literal newline character is represented as a backslash and a newline. When loading text data not generated by Postgres, you will need to convert backslash characters ("\") to double-backslashes ("\\") to ensure that they are loaded properly.

Binary Format

In the case of COPY BINARY, the first four bytes in the file will be the number of instances in the file. If this number is zero, the COPY BINARY command will read until end of file is encountered. Otherwise, it will stop reading when this number of instances has been read. Remaining data in the file will be ignored.

The format for each instance in the file is as follows. Note that this format must be followed *exactly*. Unsigned four-byte integer quantities are called uint32 in the table below.

Table 19.1. Contents of a binary copy file

At the start of the file	
uint32	number of tuples
For each tuple	
uint32	total length of tuple data
uint32	oid (if specified)
uint32	number of null attributes
[uint32,...,uint32]	attribute numbers of attributes, counting from 0
-	<tuple data>

Alignment of Binary Data

On Sun-3s, 2-byte attributes are aligned on two-byte boundaries, and all larger attributes are aligned on four-byte boundaries. Character attributes are aligned on single-byte boundaries. On most other machines, all attributes larger than 1 byte are aligned on four-byte boundaries. Note that variable length attributes are preceded by the attribute's length; arrays are simply contiguous streams of the array element type.

Usage

The following example copies a table to standard output, using a vertical bar ("|") as the field delimiter:

```
COPY country TO stdout USING DELIMITERS '|';
```

To copy data from a Unix file into a table "country":

```
COPY country FROM '/usr1/proj/bray/sql/country_data';
```

Here is a sample of data suitable for copying into a table from `stdin` (so it has the termination sequence on the last line):

```
AF      AFGHANISTAN
AL      ALBANIA
DZ      ALGERIA
...
ZM      ZAMBIA
ZW      ZIMBABWE
\.
```

The same data, output in binary format on a Linux/i586 machine. The data is shown after filtering through the Unix utility `od -c`. The table has three fields; the first is `char(2)` and the second is `text`. All the rows have a null value in the third field. Notice how the `char(2)` field is padded with nulls to four bytes and the text field is preceded by its length:

```
355  \0  \0  \0 027  \0  \0  \0 001  \0  \0  \0 002  \0  \0  \0
006  \0  \0  \0   A   F  \0  \0 017  \0  \0  \0   A   F   G   H
  A   N   I   S   T   A   N 023  \0  \0  \0 001  \0  \0  \0 002
 \0  \0  \0 006  \0  \0  \0   A   L  \0  \0  \v  \0  \0  \0   A
  L   B   A   N   I   A 023  \0  \0  \0 001  \0  \0  \0 002  \0
 \0  \0 006  \0  \0  \0   D   Z  \0  \0  \v  \0  \0  \0   A   L
  G   E   R   I   A
...             \n  \0  \0  \0   Z   A   M   B   I   A 024  \0
 \0  \0 001  \0  \0  \0 002  \0  \0  \0 006  \0  \0  \0   Z   W
 \0  \0  \f  \0  \0  \0   Z   I   M   B   A   B   W   E
```

Compatibility

SQL92

There is no `COPY` statement in SQL92.

Name

CREATE AGGREGATE — Defines a new aggregate function

Synopsis

<refsynopsisdivinfo>1999-07-20</refsynopsisdivinfo>

```
CREATE AGGREGATE name ( BASETYPE = input_data_type
    [ , SFUNC1 = sfunc1, STYPE1 = state1_type ]
    [ , SFUNC2 = sfunc2, STYPE2 = state2_type ]
    [ , FINALFUNC = ffunc ]
    [ , INITCOND1 = initial_condition1 ]
    [ , INITCOND2 = initial_condition2 ] )
```

Inputs

name

The name of an aggregate function to create.

input_data_type

The input data type on which this aggregate function operates.

sfunc1

A state transition function to be called for every non-NULL input data value. This must be a function of two arguments, the first being of type *state1_type* and the second of type *input_data_type*. The function must return a value of type *state1_type*. This function takes the current state value 1 and the current input data item, and returns the next state value 1.

state1_type

The data type for the first state value of the aggregate.

sfunc2

A state transition function to be called for every non-NULL input data value. This must be a function of one argument of type *state2_type*, returning a value of the same type. This function takes the current state value 2 and returns the next state value 2.

state2_type

The data type for the second state value of the aggregate.

ffunc

The final function called to compute the aggregate's result after all input data has been traversed. If both state values are used, the final function must take two arguments of types *state1_type* and *state2_type*. If only one state value is used, the final function must take a single argument of that state value's type. The output datatype of the aggregate is defined as the return type of this function.

initial_condition1

The initial value for state value 1.

initial_condition2

The initial value for state value 2.

Outputs

`CREATE`

Message returned if the command completes successfully.

Description

`CREATE AGGREGATE` allows a user or programmer to extend Postgres functionality by defining new aggregate functions. Some aggregate functions for base types such as `min(int4)` and `avg(float8)` are already provided in the base distribution. If one defines new types or needs an aggregate function not already provided then `CREATE AGGREGATE` can be used to provide the desired features.

An aggregate function is identified by its name and input data type. Two aggregates can have the same name if they operate on different input types. To avoid confusion, do not make an ordinary function of the same name and input data type as an aggregate.

An aggregate function is made from between one and three ordinary functions: two state transition functions, *`sfunc1`* and *`sfunc2`*, and a final calculation function, *`ffunc`*. These are used as follows:

```
sfunc1( internal-state1, next-data-item ) ---> next-internal-state1
sfunc2( internal-state2 ) ---> next-internal-state2
ffunc(internal-state1, internal-state2) ---> aggregate-value
```

Postgres creates one or two temporary variables (of data types *`stype1`* and/or *`stype2`*) to hold the current internal states of the aggregate. At each input data item, the state transition function(s) are invoked to calculate new values for the internal state values. After all the data has been processed, the final function is invoked once to calculate the aggregate's output value.

`ffunc` must be specified if both transition functions are specified. If only one transition function is used, then *`ffunc`* is optional. The default behavior when *`ffunc`* is not provided is to return the ending value of the internal state value being used (and, therefore, the aggregate's output type is the same as that state value's type).

An aggregate function may also provide one or two initial conditions, that is, initial values for the internal state values being used. These are specified and stored in the database as fields of type `text`, but they must be valid external representations of constants of the state value datatypes. If *`sfunc1`* is specified without an *`initcond1`* value, then the system does not call *`sfunc1`* at the first input item; instead, the internal state value 1 is initialized with the first input value, and *`sfunc1`* is called beginning at the second input item. This is useful for aggregates like MIN and MAX. Note that an aggregate using this feature will return NULL when called with no input values. There is no comparable provision for state value 2; if *`sfunc2`* is specified then an *`initcond2`* is required.

Notes

Use `DROP AGGREGATE` to drop aggregate functions.

The parameters of `CREATE AGGREGATE` can be written in any order, not just the order illustrated above.

It is possible to specify aggregate functions that have varying combinations of state and final functions. For example, the `count` aggregate requires *`sfunc2`* (an incrementing function) but not *`sfunc1`* or *`ffunc`*, whereas the `sum` aggregate requires *`sfunc1`* (an addition function) but not *`sfunc2`* or *`ffunc`*, and the `avg` aggregate requires both state functions as well as a *`ffunc`* (a division function) to produce its answer. In any case, at least one state function must be defined, and any *`sfunc2`* must have a corresponding *`initcond2`*.

Usage

Refer to the chapter on aggregate functions in the *PostgreSQL Programmer's Guide* for complete examples of usage.

Compatibility

SQL92

`CREATE AGGREGATE` is a Postgres language extension. There is no `CREATE AGGREGATE` in SQL92.

Name

CREATE CONSTRAINT TRIGGER — Create a trigger to support a constraint

Synopsis

<refsynopsisdivinfo>2000-04-13</refsynopsisdivinfo>

```
CREATE CONSTRAINT TRIGGER name
    AFTER events ON
    relation constraint attributes
    FOR EACH ROW EXECUTE PROCEDURE func '(' args ')'
```

Inputs

name

The name of the constraint trigger.

events

The event categories for which this trigger should be fired.

relation

Table name of the triggering relation.

constraint

Actual onstraint specification.

attributes

Contraint attributes.

func(*args*)

Function to call as part of the trigger processing.

Outputs

CREATE CONSTRAINT

Message returned if successful.

Description

CREATE CONSTRAINT TRIGGER is used from inside of CREATE/ALTER TABLE and by pg_dump to create the special triggers for referential integrity.

It is not intended for general use.

Name

CREATE DATABASE — Creates a new database

Synopsis

<refsynopsisdivinfo>1999-12-11</refsynopsisdivinfo>

```
CREATE DATABASE name [ WITH LOCATION = 'dbpath' ]
```

Inputs

name

The name of a database to create.

dbpath

An alternate location where to store the new database in the filesystem. See below for caveats.

Outputs

`CREATE DATABASE`

Message returned if the command completes successfully.

`ERROR: user 'username' is not allowed to create/drop databases`

You must have the special CREATEDB privilege to create databases. See CREATE USER .

`ERROR: createdb: database "name" already exists`

This occurs if a database with the *name* specified already exists.

```
ERROR: Single quotes are not allowed in database names.
ERROR: Single quotes are not allowed in database paths.
```

The database *name* and *dbpath* cannot contain single quotes. This is required so that the shell commands that create the database directory can execute safely.

`ERROR: The path 'xxx' is invalid.`

The expansion of the specified *dbpath* (see below how) failed. Check the path you entered or make sure that the environment variable you are referencing does exist.

`ERROR: createdb: May not be called in a transaction block.`

If you have an explicit transaction block in progress you cannot call `CREATE DATABASE`. You must finish the transaction first.

```
ERROR: Unable to create database directory 'path'.
ERROR: Could not initialize database directory.
```

These are most likely related to insufficient permissions on the data directory, a full disk, or other file system problems. The user under which the database server is running, must have access to the location.

Description

`CREATE DATABASE` creates a new Postgres database. The creator becomes the owner of the new database.

An alternate location can be specified in order to, for example, store the database on a different disk. The path must have been prepared with the initlocation command.

If the path contains a slash, the leading part is interpreted as an environment variable, which must be known to the server process. This way the database administrator can exercise control over at which locations databases can be created. (A customary choice is, e.g., 'PGDATA2'.) If the server is compiled with ALLOW_ABSOLUTE_DBPATHS (not so by default), absolute path names, as identified by a leading slash (e.g. '/usr/local/pgsql/data'), are allowed as well.

Notes

CREATE DATABASE is a Postgres language extension.

Use DROP DATABASE to remove a database.

The program createdb is a shell script wrapper around this command, provided for convenience.

There are security and data integrity issues involved with using alternate database locations specified with absolute path names, and by default only an environment variable known to the backend may be specified for an alternate location. See the Administrator's Guide for more information.

Usage

To create a new database:

```
olly=> create database lusiadas;
```

To create a new database in an alternate area ~/private_db:

```
$ mkdir private_db
$ initlocation ~/private_db
Creating Postgres database system directory /home/olly/private_db/
base

$ psql olly
Welcome to psql, the PostgreSQL interactive terminal.

Type:  \copyright for distribution terms
       \h for help with SQL commands
       \? for help on internal slash commands
       \g or terminate with semicolon to execute query
       \q to quit

olly=> CREATE DATABASE elsewhere WITH LOCATION = '/home/olly/
private_db';
CREATE DATABASE
```

Compatibility

SQL92

There is no CREATE DATABASE statement in SQL92. Databases are equivalent to catalogs whose creation is implementation-defined.

Name

CREATE FUNCTION — Defines a new function

Synopsis

<refsynopsisdivinfo>2000-03-25</refsynopsisdivinfo>

```
CREATE FUNCTION name ( [ ftype [, ...] ] )
    RETURNS rtype
    AS definition
    LANGUAGE 'langname'
    [ WITH ( attribute [, ...] ) ]
CREATE FUNCTION name ( [ ftype [, ...] ] )
    RETURNS rtype
    AS obj_file , link_symbol
    LANGUAGE 'C'
    [ WITH ( attribute [, ...] ) ]
```

Inputs

name

The name of a function to create.

ftype

The data type of function arguments. The input types may be base or complex types, or *opaque*. `opaque` indicates that the function accepts arguments of an invalid type such as `char *`.

rtype

The return data type. The output type may be specified as a base type, complex type, `setof` *type*, or `opaque`. The `setof` modifier indicates that the function will return a set of items, rather than a single item.

attribute

An optional piece of information about the function, used for optimization. The only attribute currently supported is `iscachable`. `iscachable` indicates that the function always returns the same result when given the same input values (i.e., it does not do database lookups or otherwise use information not directly present in its parameter list). The optimizer uses `iscachable` to know whether it is safe to pre-evaluate a call of the function.

definition

A string defining the function; the meaning depends on the language. It may be an internal function name, the path to an object file, an SQL query, or text in a procedural language.

obj_file, *link_symbol*

This form of the `AS` clause is used for dynamically-linked, C language functions when the function name in the C language source code is not the same as the name of the SQL function. The string *obj_file* is the name of the file containing the dynamically loadable object, and *link_symbol*, is the object's link symbol which is the same as the name of the function in the C language source code.

langname

may be 'C', 'sql', 'internal' or '*plname*', where '*plname*' is the name of a created procedural language. See CREATE LANGUAGE for details.

Outputs

`CREATE`

This is returned if the command completes successfully.

Description

`CREATE FUNCTION` allows a Postgres user to register a function with a database. Subsequently, this user is considered the owner of the function.

Notes

Refer to the chapter in the *PostgreSQL Programmer's Guide* on the topic of extending Postgres via functions for further information on writing external functions.

Use `DROP FUNCTION` to remove user-defined functions.

Postgres allows function "overloading"; that is, the same name can be used for several different functions so long as they have distinct argument types. This facility must be used with caution for `internal` and C-language functions, however.

The full SQL92 type syntax is allowed for input arguments and return value. However, some details of the type specification (e.g. the precision field for `numeric` types) are the responsibility of the underlying function implementation and are silently swallowed (e.g. not recognized or enforced) by the `CREATE FUNCTION` command.

Two `internal` functions cannot have the same C name without causing errors at link time. To get around that, give them different C names (for example, use the argument types as part of the C names), then specify those names in the AS clause of `CREATE FUNCTION`. If the AS clause is left empty then `CREATE FUNCTION` assumes the C name of the function is the same as the SQL name.

When overloading SQL functions with C-language functions, give each C-language instance of the function a distinct name, and use the alternative form of the `AS` clause in the `CREATE FUNCTION` syntax to ensure that overloaded SQL functions names are resolved to the correct dynamically linked objects.

A C function cannot return a set of values.

Usage

To create a simple SQL function:

```
CREATE FUNCTION one() RETURNS int4
    AS 'SELECT 1 AS RESULT'
    LANGUAGE 'sql';
SELECT one() AS answer;

     answer
--------
      1
```

This example creates a C function by calling a routine from a user-created shared library. This particular routine calculates a check digit and returns TRUE if the check digit in the function parameters is correct. It is intended for use in a CHECK contraint.

```
CREATE FUNCTION ean_checkdigit(bpchar, bpchar) RETURNS bool
```

```
    AS '/usr1/proj/bray/sql/funcs.so' LANGUAGE 'c';

CREATE TABLE product (
    id        char(8) PRIMARY KEY,
    eanprefix char(8) CHECK (eanprefix ~ '[0-9]{2}-[0-9]{5}')
                      REFERENCES brandname(ean_prefix),
    eancode   char(6) CHECK (eancode ~ '[0-9]{6}'),
    CONSTRAINT ean    CHECK (ean_checkdigit(eanprefix, eancode))
);
```

This example creates a function that does type conversion between the user defined type complex, and the internal type point. The function is implemented by a dynamically loaded object that was compiled from C source. For Postgres to find a type conversion function automatically, the sql function has to have the same name as the return type, and overloading is unavoidable. The function name is overloaded by using the second form of the `AS` clause in the SQL definition

```
CREATE FUNCTION point(complex) RETURNS point
    AS '/home/bernie/pgsql/lib/complex.so', 'complex_to_point'
    LANGUAGE 'c';
```

The C decalaration of the function is:

```
Point * complex_to_point (Complex *z)
{
 Point *p;

 p = (Point *) palloc(sizeof(Point));
 p->x = z->x;
 p->y = z->y;

 return p;
}
```

Compatibility

SQL92

`CREATE FUNCTION` is a Postgres language extension.

SQL/PSM

Note

PSM stands for Persistent Stored Modules. It is a procedural language and it was originally hoped that PSM would be ratified as an official standard by late 1996. As of mid-1998, this has not yet happened, but it is hoped that PSM will eventually become a standard.

SQL/PSM `CREATE FUNCTION` has the following syntax:

```
CREATE FUNCTION name
    ( [ [ IN | OUT | INOUT ] type [, ...] ] )
     RETURNS rtype
     LANGUAGE 'langname'
     ESPECIFIC routine
     SQL-statement
```

Name

CREATE GROUP — Creates a new group

Synopsis

<refsynopsisdivinfo>2000-01-14</refsynopsisdivinfo>

```
CREATE GROUP name
    [ WITH
     [ SYSID gid ]
     [ USER  username [, ...] ] ]
```

Inputs

name

The name of the group.

gid

The SYSID clause can be used to choose the Postgres group id of the new group. It is not necessary to do so, however.

If this is not specified, the highest assigned group id plus one, starting at 1, will be used as default.

username

A list of users to include in the group. The users must already exist.

Outputs

CREATE GROUP

Message returned if the command completes successfully.

Description

CREATE GROUP will create a new group in the database installation. Refer to the adminstrator's guide for information about using groups for authentication. You must be a database superuser to use this command.

Use ALTER GROUP to change a group's membership, and DROP GROUP to remove a group.

Usage

Create an empty group:

```
CREATE GROUP staff
```

Create a group with members:

```
CREATE GROUP marketing WITH USER jonathan, david
```

Compatibility

SQL92

There is no CREATE GROUP statement in SQL92. Roles are similar in concept to groups.

Name

CREATE INDEX — Constructs a secondary index

Synopsis

<refsynopsisdivinfo>1999-07-20</refsynopsisdivinfo>

```
CREATE [ UNIQUE ] INDEX index_name ON table
    [ USING acc_name ] ( column [ ops_name ] [, ...] )
CREATE [ UNIQUE ] INDEX index_name ON table
    [ USING acc_name ] ( func_name( column [, ... ]) [ ops_name ] )
```

Inputs

UNIQUE

Causes the system to check for duplicate values in the table when the index is created (if data already exist) and each time data is added. Attempts to insert or update data which would result in duplicate entries will generate an error.

index_name

The name of the index to be created.

table

The name of the table to be indexed.

acc_name

The name of the access method to be used for the index. The default access method is BTREE. Postgres provides three access methods for indexes:

BTREE

an implementation of Lehman-Yao high-concurrency btrees.

RTREE

implements standard rtrees using Guttman's quadratic split algorithm.

HASH

an implementation of Litwin's linear hashing.

column

The name of a column of the table.

ops_name

An associated operator class. See below for details.

func_name

A function, which returns a value that can be indexed.

Outputs

CREATE

The message returned if the index is successfully created.

```
ERROR: Cannot create index: 'index_name' already exists.
```

This error occurs if it is impossible to create the index.

Description

`CREATE INDEX` constructs an index *`index_name`* on the specified *`table`*.

Tip

Indexes are primarily used to enhance database performance. But inappropriate use will result in slower performance.

In the first syntax shown above, the key field(s) for the index are specified as column names. Multiple fields can be specified if the index access method supports multi-column indexes.

In the second syntax shown above, an index is defined on the result of a user-specified function *`func_name`* applied to one or more attributes of a single class. These *functional indices* can be used to obtain fast access to data based on operators that would normally require some transformation to apply them to the base data.

Postgres provides btree, rtree and hash access methods for indices. The btree access method is an implementation of Lehman-Yao high-concurrency btrees. The rtree access method implements standard rtrees using Guttman's quadratic split algorithm. The hash access method is an implementation of Litwin's linear hashing. We mention the algorithms used solely to indicate that all of these access methods are fully dynamic and do not have to be optimized periodically (as is the case with, for example, static hash access methods).

Use DROP INDEX to remove an index.

Notes

The Postgres query optimizer will consider using a btree index whenever an indexed attribute is involved in a comparison using one of: <, <=, =, >=, >

The Postgres query optimizer will consider using an rtree index whenever an indexed attribute is involved in a comparison using one of: <<, &<, &>, >>, @, ~=, &&

The Postgres query optimizer will consider using a hash index whenever an indexed attribute is involved in a comparison using the = operator.

Currently, only the btree access method supports multi-column indexes. Up to 16 keys may be specified by default (this limit can be altered when building Postgres).

An *operator class* can be specified for each column of an index. The operator class identifies the operators to be used by the index for that column. For example, a btree index on four-byte integers would use the `int4_ops` class; this operator class includes comparison functions for four-byte integers. In practice the default operator class for the field's datatype is usually sufficient. The main point of having operator classes is that for some datatypes, there could be more than one meaningful ordering. For example, we might want to sort a complex-number datatype either by absolute value or by real part. We could do this by defining two operator classes for the datatype and then selecting the proper class when making an index. There are also some operator classes with special purposes:

- The operator classes `box_ops` and `bigbox_ops` both support rtree indices on the `box` datatype. The difference between them is that `bigbox_ops` scales box coordinates down, to avoid floating point exceptions from doing multiplication, addition, and subtraction on very large floating-point coordinates. If the field on which your rectangles lie is about 20,000 units square or larger, you should use `bigbox_ops`.

- The `int24_ops` operator class is useful for constructing indices on int2 data, and doing comparisons against int4 data in query qualifications. Similarly, `int42_ops` support indices on int4 data that is to be compared against int2 data in queries.

The following query shows all defined operator classes:

```
SELECT am.amname AS acc_name,
       opc.opcname AS ops_name,
       opr.oprname AS ops_comp
    FROM pg_am am, pg_amop amop,
         pg_opclass opc, pg_operator opr
    WHERE amop.amopid = am.oid AND
          amop.amopclaid = opc.oid AND
          amop.amopopr = opr.oid
    ORDER BY acc_name, ops_name, ops_comp
```

Usage

To create a btree index on the field `title` in the table `films`:

```
CREATE UNIQUE INDEX title_idx
    ON films (title);
```

Compatibility

SQL92

CREATE INDEX is a Postgres language extension.

There is no `CREATE INDEX` command in SQL92.

Name

CREATE LANGUAGE — Defines a new language for functions

Synopsis

<refsynopsisdivinfo>1999-07-20</refsynopsisdivinfo>

```
CREATE [ TRUSTED ] PROCEDURAL LANGUAGE 'langname'
    HANDLER call_handler
    LANCOMPILER 'comment'
```

Inputs

TRUSTED

`TRUSTED` specifies that the call handler for the language is safe; that is, it offers an unprivileged user no functionality to bypass access restrictions. If this keyword is omitted when registering the language, only users with the Postgres superuser privilege can use this language to create new functions (like the 'C' language).

langname

The name of the new procedural language. The language name is case insensitive. A procedural language cannot override one of the built-in languages of Postgres.

HANDLER *call_handler*

call_handler is the name of a previously registered function that will be called to execute the PL procedures.

comment

The `LANCOMPILER` argument is the string that will be inserted in the `LANCOMPILER` attribute of the new `pg_language` entry. At present, Postgres does not use this attribute in any way.

Outputs

`CREATE`

This message is returned if the language is successfully created.

`ERROR: PL handler function` *funcname*`() doesn't exist`

This error is returned if the function *funcname*() is not found.

Description

Using `CREATE LANGUAGE`, a Postgres user can register a new language with Postgres. Subsequently, functions and trigger procedures can be defined in this new language. The user must have the Postgres superuser privilege to register a new language.

Writing PL handlers

The call handler for a procedural language must be written in a compiler language such as 'C' and registered with Postgres as a function taking no arguments and returning the `opaque` type, a placeholder for unspecified or undefined types.. This prevents the call handler from being called directly as a function from queries.

However, arguments must be supplied on the actual call when a PL function or trigger procedure in the language offered by the handler is to be executed.

- When called from the trigger manager, the only argument is the object ID from the procedure's `pg_proc` entry. All other information from the trigger manager is found in the global `CurrentTriggerData` pointer.
- When called from the function manager, the arguments are the object ID of the procedure's `pg_proc` entry, the number of arguments given to the PL function, the arguments in a `FmgrValues` structure and a pointer to a boolean where the function tells the caller if the return value is the SQL NULL value.

It's up to the call handler to fetch the `pg_proc` entry and to analyze the argument and return types of the called procedure. The AS clause from the `CREATE FUNCTION` of the procedure will be found in the `prosrc` attribute of the `pg_proc` table entry. This may be the source text in the procedural language itself (like for PL/Tcl), a pathname to a file or anything else that tells the call handler what to do in detail.

Notes

Use `CREATE FUNCTION` to create a function.

Use `DROP LANGUAGE` to drop procedural languages.

Refer to the table `pg_language` for further information:

```
        Table "pg_language"
   Attribute   |  Type   | Modifier
---------------+---------+----------
 lanname       | name    |
 lanispl       | boolean |
 lanpltrusted  | boolean |
 lanplcallfoid | oid     |
 lancompiler   | text    |

 lanname  | lanispl | lanpltrusted | lanplcallfoid | lancompiler
----------+---------+--------------+---------------+-------------
 internal | f       | f            |             0 | n/a
 C        | f       | f            |             0 | /bin/cc
 sql      | f       | f            |             0 | postgres
```

Since the call handler for a procedural language must be registered with Postgres in the 'C' language, it inherits all the capabilities and restrictions of 'C' functions.

At present, the definitions for a procedural language cannot be changed once they have been created.

Usage

This is a template for a PL handler written in 'C':

```
#include "executor/spi.h"
#include "commands/trigger.h"
#include "utils/elog.h"
#include "fmgr.h"        /* for FmgrValues struct */
#include "access/heapam.h"
#include "utils/syscache.h"
#include "catalog/pg_proc.h"
#include "catalog/pg_type.h"
```

```
Datum
plsample_call_handler(
     Oid       prooid,
     int       pronargs,
     FmgrValues     *proargs,
     bool      *isNull)
{
     Datum          retval;
     TriggerData    *trigdata;

     if (CurrentTriggerData == NULL) {
          /*
           * Called as a function
           */

          retval = ...
     } else {
          /*
           * Called as a trigger procedure
           */
          trigdata = CurrentTriggerData;
          CurrentTriggerData = NULL;

          retval = ...
     }

     *isNull = false;
     return retval;
}
```

Only a few thousand lines of code have to be added instead of the dots to complete the PL call handler. See `CREATE FUNCTION` for information on how to compile it into a loadable module.

The following commands then register the sample procedural language:

```
CREATE FUNCTION plsample_call_handler () RETURNS opaque
    AS '/usr/local/pgsql/lib/plsample.so'
    LANGUAGE 'C';
CREATE PROCEDURAL LANGUAGE 'plsample'
    HANDLER plsample_call_handler
    LANCOMPILER 'PL/Sample';
```

Compatibility

SQL92

`CREATE LANGUAGE` is a Postgres extension. There is no `CREATE LANGUAGE` statement in SQL92.

Name

CREATE OPERATOR — Defines a new user operator

Synopsis

<refsynopsisdivinfo>2000-03-25</refsynopsisdivinfo>

```
CREATE OPERATOR name ( PROCEDURE = func_name
     [, LEFTARG = type1 ] [, RIGHTARG = type2 ]
     [, COMMUTATOR = com_op ] [, NEGATOR = neg_op ]
     [, RESTRICT = res_proc ] [, JOIN = join_proc ]
     [, HASHES ] [, SORT1 = left_sort_op ] [, SORT2 = right_sort_op
 ] )
```

Inputs

name

The operator to be defined. See below for allowable characters.

func_name

The function used to implement this operator.

type1

The type of the left-hand argument of the operator, if any. This option would be omitted for a left-unary operator.

type2

The type of the right-hand argument of the operator, if any. This option would be omitted for a right-unary operator.

com_op

The commutator of this operator.

neg_op

The negator of this operator.

res_proc

The restriction selectivity estimator function for this operator.

join_proc

The join selectivity estimator function for this operator.

HASHES

Indicates this operator can support a hash join.

left_sort_op

If this operator can support a merge join, the operator that sorts the left-hand data type of this operator.

right_sort_op

If this operator can support a merge join, the operator that sorts the right-hand data type of this operator.

Outputs

CREATE

Message returned if the operator is successfully created.

Description

CREATE OPERATOR defines a new operator, *name*. The user who defines an operator becomes its owner.

The operator *name* is a sequence of up to NAMEDATALEN-1 (31 by default) characters from the following list:

+ - * / < > = ~ ! @ # % ^ & | ` ? $:

There are a few restrictions on your choice of name:

- "$" and ":" cannot be defined as single-character operators, although they can be part of a multi-character operator name.
- "--" and "/*" cannot appear anywhere in an operator name, since they will be taken as the start of a comment.
- A multi-character operator name cannot end in "+" or "-", unless the name also contains at least one of these characters:

  ~ ! @ # % ^ & | ` ? $:

  For example, `@-` is an allowed operator name, but `*-` is not. This restriction allows Postgres to parse SQL-compliant queries without requiring spaces between tokens.

Note

When working with non-SQL-standard operator names, you will usually need to separate adjacent operators with spaces to avoid ambiguity. For example, if you have defined a left-unary operator named "@", you cannot write `X*@Y`; you must write `X* @Y` to ensure that Postgres reads it as two operator names not one.

The operator "!=" is mapped to "<>" on input, so these two names are always equivalent.

At least one of LEFTARG and RIGHTARG must be defined. For binary operators, both should be defined. For right unary operators, only LEFTARG should be defined, while for left unary operators only RIGHTARG should be defined.

The *func_name* procedure must have been previously defined using CREATE FUNCTION and must be defined to accept the correct number of arguments (either one or two) of the indicated types.

The commutator operator should be identified if one exists, so that Postgres can reverse the order of the operands if it wishes. For example, the operator area-less-than, <<<, would probably have a commutator operator, area-greater-than, >>>. Hence, the query optimizer could freely convert:

```
box '((0,0),(1,1))'  >>> MYBOXES.description
```

to

```
MYBOXES.description <<< box '((0,0),(1,1))'
```

This allows the execution code to always use the latter representation and simplifies the query optimizer somewhat.

Similarly, if there is a negator operator then it should be identified. Suppose that an operator, area-equal, ===, exists, as well as an area not equal, !==. The negator link allows the query optimizer to simplify

```
NOT MYBOXES.description === box '((0,0),(1,1))'
```

to

```
MYBOXES.description !== box '((0,0),(1,1))'
```

If a commutator operator name is supplied, Postgres searches for it in the catalog. If it is found and it does not yet have a commutator itself, then the commutator's entry is updated to have the newly created operator as its commutator. This applies to the negator, as well. This is to allow the definition of two operators that are the commutators or the negators of each other. The first operator should be defined without a commutator or negator (as appropriate). When the second operator is defined, name the first as the commutator or negator. The first will be updated as a side effect. (As of Postgres 6.5, it also works to just have both operators refer to each other.)

The HASHES, SORT1, and SORT2 options are present to support the query optimizer in performing joins. Postgres can always evaluate a join (i.e., processing a clause with two tuple variables separated by an operator that returns a boolean) by iterative substitution [WONG76]. In addition, Postgres can use a hash-join algorithm along the lines of [SHAP86]; however, it must know whether this strategy is applicable. The current hash-join algorithm is only correct for operators that represent equality tests; furthermore, equality of the datatype must mean bitwise equality of the representation of the type. (For example, a datatype that contains unused bits that don't matter for equality tests could not be hashjoined.) The HASHES flag indicates to the query optimizer that a hash join may safely be used with this operator.

Similarly, the two sort operators indicate to the query optimizer whether merge-sort is a usable join strategy and which operators should be used to sort the two operand classes. Sort operators should only be provided for an equality operator, and they should refer to less-than operators for the left and right side data types respectively.

If other join strategies are found to be practical, Postgres will change the optimizer and run-time system to use them and will require additional specification when an operator is defined. Fortunately, the research community invents new join strategies infrequently, and the added generality of user-defined join strategies was not felt to be worth the complexity involved.

The RESTRICT and JOIN options assist the query optimizer in estimating result sizes. If a clause of the form:

```
MYBOXES.description <<< box '((0,0),(1,1))'
```

is present in the qualification, then Postgres may have to estimate the fraction of the instances in MYBOXES that satisfy the clause. The function *res_proc* must be a registered function (meaning it is already defined using `CREATE FUNCTION`) which accepts arguments of the correct data types and returns a floating point number. The query optimizer simply calls this function, passing the

parameter `((0,0),(1,1))` and multiplies the result by the relation size to get the expected number of instances.

Similarly, when the operands of the operator both contain instance variables, the query optimizer must estimate the size of the resulting join. The function join_proc will return another floating point number which will be multiplied by the cardinalities of the two classes involved to compute the expected result size.

The difference between the function

```
my_procedure_1 (MYBOXES.description, box '((0,0),(1,1))')
```

and the operator

```
MYBOXES.description === box '((0,0),(1,1))'
```

is that Postgres attempts to optimize operators and can decide to use an index to restrict the search space when operators are involved. However, there is no attempt to optimize functions, and they are performed by brute force. Moreover, functions can have any number of arguments while operators are restricted to one or two.

Notes

Refer to the chapter on operators in the *PostgreSQL User's Guide* for further information. Refer to `DROP OPERATOR` to delete user-defined operators from a database.

Usage

The following command defines a new operator, area-equality, for the BOX data type.

```
CREATE OPERATOR === (
   LEFTARG = box,
   RIGHTARG = box,
   PROCEDURE = area_equal_procedure,
   COMMUTATOR = ===,
   NEGATOR = !==,
   RESTRICT = area_restriction_procedure,
   JOIN = area_join_procedure,
   HASHES,
   SORT1 = <<<,
   SORT2 = <<<
);
```

Compatibility

SQL92

`CREATE OPERATOR` is a Postgres extension. There is no `CREATE OPERATOR` statement in SQL92.

Name

CREATE RULE — Defines a new rule

Synopsis

<refsynopsisdivinfo>1999-07-20</refsynopsisdivinfo>

```
CREATE RULE name AS ON event
    TO object [ WHERE condition ]
    DO [ INSTEAD ] [ action | NOTHING ]
```

Inputs

name

The name of a rule to create.

event

Event is one of `select`, `update`, `delete` or `insert`.

object

Object is either *table* or *table.column*.

condition

Any SQL WHERE clause, `new` or `old` can appear instead of an instance variable whenever an instance variable is permissible in SQL.

action

Any SQL statement, `new` or `old` can appear instead of an instance variable whenever an instance variable is permissible in SQL.

Outputs

`CREATE`

Message returned if the rule is successfully created.

Description

The Postgres *rule system* allows one to define an alternate action to be performed on inserts, updates, or deletions from database tables or classes. Currently, rules are used to implement table views.

The semantics of a rule is that at the time an individual instance is accessed, inserted, updated, or deleted, there is a old instance (for selects, updates and deletes) and a new instance (for inserts and updates). If the *event* specified in the ON clause and the *condition* specified in the WHERE clause are true for the old instance, the *action* part of the rule is executed. First, however, values from fields in the old instance and/or the new instance are substituted for `old.`*attribute-name* and `new.`*attribute-name*.

The *action* part of the rule executes with the same command and transaction identifier as the user command that caused activation.

Notes

A caution about SQL rules is in order. If the same class name or instance variable appears in the *event*, *condition* and *action* parts of a rule, they are all considered different tuple variables.

More accurately, new and old are the only tuple variables that are shared between these clauses. For example, the following two rules have the same semantics:

```
ON UPDATE TO emp.salary WHERE emp.name = "Joe"
    DO
 UPDATE emp SET ... WHERE ...


ON UPDATE TO emp-1.salary WHERE emp-2.name = "Joe"
    DO
 UPDATE emp-3 SET ...  WHERE ...
```

Each rule can have the optional tag INSTEAD. Without this tag, *action* will be performed in addition to the user command when the *event* in the *condition* part of the rule occurs. Alternately, the *action* part will be done instead of the user command. In this later case, the *action* can be the keyword NOTHING.

It is very important to note to avoid circular rules. For example, though each of the following two rule definitions are accepted by Postgres, the select command will cause Postgres to report an error because the query cycled too many times:

Example 19.1. Example of a circular rewrite rule combination.

```
CREATE RULE bad_rule_combination_1 AS
    ON SELECT TO emp
    DO INSTEAD
 SELECT TO toyemp;


CREATE RULE bad_rule_combination_2 AS
    ON SELECT TO toyemp
    DO INSTEAD
 SELECT TO emp;
```

This attempt to select from EMP will cause Postgres to issue an error because the queries cycled too many times.

```
SELECT * FROM emp;
```

You must have rule definition access to a class in order to define a rule on it. Use GRANT and REVOKE to change permissions.

The object in a SQL rule cannot be an array reference and cannot have parameters.

Aside from the "oid" field, system attributes cannot be referenced anywhere in a rule. Among other things, this means that functions of instances (e.g., foo(emp) where emp is a class) cannot be called anywhere in a rule.

The rule system stores the rule text and query plans as text attributes. This implies that creation of rules may fail if the rule plus its various internal representations exceed some value that is on the order of one page (8KB).

Usage

Make Sam get the same salary adjustment as Joe:

```
CREATE RULE example_1 AS
    ON UPDATE emp.salary WHERE old.name = "Joe"
```

```
    DO
 UPDATE emp
 SET salary = new.salary
 WHERE emp.name = "Sam";
```

At the time Joe receives a salary adjustment, the event will become true and Joe's old instance and proposed new instance are available to the execution routines. Hence, his new salary is substituted into the action part of the rule which is subsequently executed. This propagates Joe's salary on to Sam.

Make Bill get Joe's salary when it is accessed:

```
CREATE RULE example_2 AS
    ON SELECT TO EMP.salary
    WHERE old.name = "Bill"
    DO INSTEAD
 SELECT emp.salary
 FROM emp
 WHERE emp.name = "Joe";
```

Deny Joe access to the salary of employees in the shoe department (`current_user` returns the name of the current user):

```
CREATE RULE example_3 AS
    ON
 SELECT TO emp.salary
    WHERE old.dept = "shoe" AND current_user = "Joe"
    DO INSTEAD NOTHING;
```

Create a view of the employees working in the toy department.

```
CREATE toyemp(name = char16, salary = int4);

CREATE RULE example_4 AS
    ON SELECT TO toyemp
    DO INSTEAD
     SELECT emp.name, emp.salary
 FROM emp
        WHERE emp.dept = "toy";
```

All new employees must make 5,000 or less

```
CREATE RULE example_5 AS
    ON INERT TO emp WHERE new.salary > 5000
    DO
 UPDATE emp SET salary = 5000
 WHERE emp.oid = new.oid;
```

Compatibility

SQL92

`CREATE RULE` statement is a Postgres language extension. There is no `CREATE RULE` statement in SQL92.

Name

CREATE SEQUENCE — Creates a new sequence number generator

Synopsis

<refsynopsisdivinfo>1999-07-20</refsynopsisdivinfo>

```
CREATE SEQUENCE seqname [ INCREMENT increment ]
    [ MINVALUE minvalue ] [ MAXVALUE maxvalue ]
    [ START start ] [ CACHE cache ] [ CYCLE ]
```

Inputs

seqname

The name of a sequence to be created.

increment

The `INCREMENT` *increment* clause is optional. A positive value will make an ascending sequence, a negative one a descending sequence. The default value is one (1).

minvalue

The optional clause `MINVALUE` *minvalue* determines the minimum value a sequence can generate. The defaults are 1 and -2147483647 for ascending and descending sequences, respectively.

maxvalue

Use the optional clause `MAXVALUE` *maxvalue* to determine the maximum value for the sequence. The defaults are 2147483647 and -1 for ascending and descending sequences, respectively.

start

The optional `START` *start* `clause` enables the sequence to begin anywhere. The default starting value is *minvalue* for ascending sequences and *maxvalue* for descending ones.

cache

The `CACHE` *cache* option enables sequence numbers to be preallocated and stored in memory for faster access. The minimum value is 1 (only one value can be generated at a time, i.e. no cache) and this is also the default.

CYCLE

The optional CYCLE keyword may be used to enable the sequence to continue when the *maxvalue* or *minvalue* has been reached by an ascending or descending sequence respectively. If the limit is reached, the next number generated will be whatever the *minvalue* or *maxvalue* is, as appropriate.

Outputs

`CREATE`

Message returned if the command is successful.

`ERROR: Relation '`*seqname*`' already exists`

If the sequence specified already exists.

`ERROR: DefineSequence: MINVALUE (start) can't be >= MAXVALUE (max)`

If the specified starting value is out of range.

`ERROR: DefineSequence: START value (start) can't be < MINVALUE (min)`

If the specified starting value is out of range.

`ERROR: DefineSequence: MINVALUE (min) can't be >= MAXVALUE (max)`

If the minimum and maximum values are inconsistant.

Description

`CREATE SEQUENCE` will enter a new sequence number generator into the current data base. This involves creating and initializing a new single-row table with the name *seqname*. The generator will be "owned" by the user issuing the command.

After a sequence is created, you may use the function `nextval(seqname)` to get a new number from the sequence. The function `currval('seqname')` may be used to determine the number returned by the last call to `nextval(seqname)` for the specified sequence in the current session. The function `setval('seqname', newvalue)` may be used to set the current value of the specified sequence. The next call to `nextval(seqname)` will return the given value plus the sequence increment.

Use a query like

```
SELECT * FROM seqname;
```

to get the parameters of a sequence. As an alternative to fetching the parameters from the original definition as above, you can use

```
SELECT last_value FROM seqname;
```

to obtain the last value allocated by any backend.

Low-level locking is used to enable multiple simultaneous calls to a generator.

Caution

Unexpected results may be obtained if a cache setting greater than one is used for a sequence object that will be used concurrently by multiple backends. Each backend will allocate and cache successive sequence values during one access to the sequence object and increase the sequence object's last_value accordingly. Then, the next cache-1 uses of nextval within that backend simply return the preallocated values without touching the shared object. So, numbers allocated but not used in the current session will be lost. Furthermore, although multiple backends are guaranteed to allocate distinct sequence values, the values may be generated out of sequence when all the backends are considered. (For example, with a cache setting of 10, backend A might reserve values 1..10 and return nextval=1, then backend B might reserve values 11..20 and return nextval=11 before backend A has generated nextval=2.) Thus, with a cache setting of one it is safe to assume that nextval values are generated sequentially; with a cache setting greater than one you should only assume that the nextval values are all distinct, not that they are generated purely sequentially. Also, last_value will reflect the latest value reserved by any backend, whether or not it has yet been returned by nextval.

Notes

Use `DROP SEQUENCE` to remove a sequence.

Each backend uses its own cache to store allocated numbers. Numbers that are cached but not used in the current session will be lost, resulting in "holes" in the sequence.

Usage

Create an ascending sequence called `serial`, starting at 101:

```
CREATE SEQUENCE serial START 101;
```

Select the next number from this sequence

```
SELECT NEXTVAL ('serial');

nextval
-------
    114
```

Use this sequence in an INSERT:

```
INSERT INTO distributors VALUES (NEXTVAL('serial'),'nothing');
```

Set the sequence value after a COPY FROM:

```
CREATE FUNCTION distributors_id_max() RETURNS INT4
    AS 'SELECT max(id) FROM distributors'
    LANGUAGE 'sql';
BEGIN;
    COPY distributors FROM 'input_file';
    SELECT setval('serial', distributors_id_max());
END;
```

Compatibility

SQL92

`CREATE SEQUENCE` is a Postgres language extension. There is no `CREATE SEQUENCE` statement in SQL92.

Name

CREATE TABLE — Creates a new table

Synopsis

<refsynopsisdivinfo>2000-03-25</refsynopsisdivinfo>

```
CREATE [ TEMPORARY | TEMP ] TABLE table (
    column type
    [ NULL | NOT NULL ] [ UNIQUE ] [ DEFAULT value ]
    [column_constraint_clause | PRIMARY KEY } [ ... ] ]
    [, ... ]
    [, PRIMARY KEY ( column [, ...] ) ]
    [, CHECK ( condition ) ]
    [, table_constraint_clause ]
    ) [ INHERITS ( inherited_table [, ...] ) ]
```

Inputs

TEMPORARY

The table is created only for this session, and is automatically dropped on session exit. Existing permanent tables with the same name are not visible while the temporary table exists.

table

The name of a new class or table to be created.

column

The name of a column.

type

The type of the column. This may include array specifiers. Refer to the *PostgreSQL User's Guide* for further information about data types and arrays.

DEFAULT *value*

A default value for a column. See the DEFAULT clause for more information.

column_constraint_clause

The optional column constraint clauses specify a list of integrity constraints or tests which new or updated entries must satisfy for an insert or update operation to succeed. Each constraint must evaluate to a boolean expression. Although SQL92 requires the *column_constraint_clause* to refer to that column only, Postgres allows multiple columns to be referenced within a single column constraint. See the column constraint clause for more information.

table_constraint_clause

The optional table CONSTRAINT clause specifies a list of integrity constraints which new or updated entries must satisfy for an insert or update operation to succeed. Each constraint must evaluate to a boolean expression. Multiple columns may be referenced within a single constraint. Only one PRIMARY KEY clause may be specified for a table; PRIMARY KEY *column* (a table constraint) and PRIMARY KEY (a column constraint) are mutually exclusive.. See the table constraint clause for more information.

INHERITS *inherited_table*

The optional INHERITS clause specifies a collection of table names from which this table automatically inherits all fields. If any inherited field name appears more than once, Postgres reports an error. Postgres automatically allows the created table to inherit functions on tables above it in the inheritance hierarchy.

Outputs

`CREATE`

Message returned if table is successfully created.

`ERROR`

Message returned if table creation failed. This is usually accompanied by some descriptive text, such as: `ERROR: Relation '`*table*`' already exists` which occurs at runtime, if the table specified already exists in the database.

`ERROR: DEFAULT: type mismatched`

If data type of default value doesn't match the column definition's data type.

Description

`CREATE TABLE` will enter a new class or table into the current data base. The table will be "owned" by the user issuing the command.

Each *type* may be a simple type, a complex type (set) or an array type. Each attribute may be specified to be non-null and each may have a default value, specified by the DEFAULT Clause .

Note

As of Postgres version 6.0, consistent array dimensions within an attribute are not enforced. This will likely change in a future release.

The optional INHERITS clause specifies a collection of class names from which this class automatically inherits all fields. If any inherited field name appears more than once, Postgres reports an error. Postgres automatically allows the created class to inherit functions on classes above it in the inheritance hierarchy. Inheritance of functions is done according to the conventions of the Common Lisp Object System (CLOS).

Each new table or class *table* is automatically created as a type. Therefore, one or more instances from the class are automatically a type and can be used in ALTER TABLE or other `CREATE TABLE` statements.

The new table is created as a heap with no initial data. A table can have no more than 1600 columns (realistically, this is limited by the fact that tuple sizes must be less than 8192 bytes), but this limit may be configured lower at some sites. A table cannot have the same name as a system catalog table.

DEFAULT Clause

`DEFAULT` *value*

Inputs

value

The possible values for the default value expression are:

- a literal value
- a user function
- a niladic function

Outputs

None.

Description

The DEFAULT clause assigns a default data value to a column (via a column definition in the CREATE TABLE statement). The data type of a default value must match the column definition's data type.

An INSERT operation that includes a column without a specified default value will assign the NULL value to the column if no explicit data value is provided for it. Default *`literal`* means that the default is the specified constant value. Default *`niladic-function`* or *`user-function`* means that the default is the value of the specified function at the time of the INSERT.

There are two types of niladic functions:

niladic USER

CURRENT_USER / USER

See CURRENT_USER function

SESSION_USER

See CURRENT_USER function

SYSTEM_USER

Not implemented

niladic datetime

CURRENT_DATE

See CURRENT_DATE function

CURRENT_TIME

See CURRENT_TIME function

CURRENT_TIMESTAMP

See CURRENT_TIMESTAMP function

Usage

To assign a constant value as the default for the columns `did` and `number`, and a string literal to the column `did`:

```
CREATE TABLE video_sales (
    did      VARCHAR(40) DEFAULT 'luso films',
    number   INTEGER DEFAULT 0,
    total    CASH DEFAULT '$0.0'
);
```

To assign an existing sequence as the default for the column `did`, and a literal to the column `name`:

```
CREATE TABLE distributors (
    did      DECIMAL(3)  DEFAULT NEXTVAL('serial'),
    name     VARCHAR(40) DEFAULT 'luso films'
);
```

Column CONSTRAINT Clause

```
[ CONSTRAINT name ] { [
    NULL | NOT NULL ] | UNIQUE | PRIMARY KEY | CHECK constraint |
 REFERENCES
     reftable
     (refcolumn)
     [ MATCH matchtype ]
     [ ON DELETE action ]
     [ ON UPDATE action ]
     [ [ NOT ] DEFERRABLE ]
     [ INITIALLY checktime ] }
    [, ...]
```

Inputs

name

An arbitrary name given to the integrity constraint. If *name* is not specified, it is generated from the table and column names, which should ensure uniqueness for *name*.

NULL

The column is allowed to contain NULL values. This is the default.

NOT NULL

The column is not allowed to contain NULL values. This is equivalent to the column constraint CHECK (*column* NOT NULL).

UNIQUE

The column must have unique values. In Postgres this is enforced by an implicit creation of a unique index on the table.

PRIMARY KEY

This column is a primary key, which implies that uniqueness is enforced by the system and that other tables may rely on this column as a unique identifier for rows. See PRIMARY KEY for more information.

constraint

The definition of the constraint.

Description

The optional constraint clauses specify constraints or tests which new or updated entries must satisfy for an insert or update operation to succeed. Each constraint must evaluate to a boolean expression.

Multiple attributes may be referenced within a single constraint. The use of PRIMARY KEY as a table constraint is mutually incompatible with PRIMARY KEY as a column constraint.

A constraint is a named rule: an SQL object which helps define valid sets of values by putting limits on the results of INSERT, UPDATE or DELETE operations performed on a Base Table.

There are two ways to define integrity constraints: table constraints, covered later, and column constraints, covered here.

A column constraint is an integrity constraint defined as part of a column definition, and logically becomes a table constraint as soon as it is created. The column constraints available are:

PRIMARY KEY
REFERENCES
UNIQUE
CHECK
NOT NULL

NOT NULL Constraint

```
[ CONSTRAINT name ] NOT NULL
```

The NOT NULL constraint specifies a rule that a column may contain only non-null values. This is a column constraint only, and not allowed as a table constraint.

Outputs

status

`ERROR: ExecAppend: Fail to add null value in not null attribute "column".`

This error occurs at runtime if one tries to insert a null value into a column which has a NOT NULL constraint.

Description

Usage

Define two NOT NULL column constraints on the table `distributors`, one of which being a named constraint:

```
CREATE TABLE distributors (
    did      DECIMAL(3) CONSTRAINT no_null NOT NULL,
    name     VARCHAR(40) NOT NULL
);
```

UNIQUE Constraint

```
[ CONSTRAINT name ] UNIQUE
```

Inputs

CONSTRAINT *name*

An arbitrary label given to a constraint.

Outputs

status

ERROR: Cannot insert a duplicate key into a unique index.

This error occurs at runtime if one tries to insert a duplicate value into a column.

Description

The UNIQUE constraint specifies a rule that a group of one or more distinct columns of a table may contain only unique values.

The column definitions of the specified columns do not have to include a NOT NULL constraint to be included in a UNIQUE constraint. Having more than one null value in a column without a NOT NULL constraint, does not violate a UNIQUE constraint. (This deviates from the SQL92 definition, but is a more sensible convention. See the section on compatibility for more details.).

Each UNIQUE column constraint must name a column that is different from the set of columns named by any other UNIQUE or PRIMARY KEY constraint defined for the table.

Note

Postgres automatically creates a unique index for each UNIQUE constraint, to assure data integrity. See CREATE INDEX for more information.

Usage

Defines a UNIQUE column constraint for the table distributors. UNIQUE column constraints can only be defined on one column of the table:

```
CREATE TABLE distributors (
    did      DECIMAL(3),
    name     VARCHAR(40) UNIQUE
);
```

which is equivalent to the following specified as a table constraint:

```
CREATE TABLE distributors (
    did      DECIMAL(3),
    name     VARCHAR(40),
    UNIQUE(name)
);
```

The CHECK Constraint

```
[ CONSTRAINT name ] CHECK
    ( condition [, ...] )
```

Inputs

name

An arbitrary name given to a constraint.

condition

Any valid conditional expression evaluating to a boolean result.

Outputs

status

ERROR: ExecAppend: rejected due to CHECK constraint "*table_column*".

This error occurs at runtime if one tries to insert an illegal value into a column subject to a CHECK constraint.

Description

The CHECK constraint specifies a restriction on allowed values within a column. The CHECK constraint is also allowed as a table constraint.

The SQL92 CHECK column constraints can only be defined on, and refer to, one column of the table. Postgres does not have this restriction.

PRIMARY KEY Constraint

```
[ CONSTRAINT name ] PRIMARY KEY
```

Inputs

CONSTRAINT *name*

An arbitrary name for the constraint.

Outputs

```
ERROR: Cannot insert a duplicate key into a unique index.
```

This occurs at run-time if one tries to insert a duplicate value into a column subject to a PRIMARY KEY constraint.

Description

The PRIMARY KEY column constraint specifies that a column of a table may contain only unique (non-duplicate), non-NULL values. The definition of the specified column does not have to include an explicit NOT NULL constraint to be included in a PRIMARY KEY constraint.

Only one PRIMARY KEY can be specified for a table.

Notes

Postgres automatically creates a unique index to assure data integrity. (See CREATE INDEX statement)

The PRIMARY KEY constraint should name a set of columns that is different from other sets of columns named by any UNIQUE constraint defined for the same table, since it will result in duplication of equivalent indexes and unproductive additional runtime overhead. However, Postgres does not specifically disallow this.

REFERENCES Constraint

```
[ CONSTRAINT name ] REFERENCES reftable [ ( refcolumn ) ]
    [ MATCH matchtype ]
    [ ON DELETE action ]
```

```
[ ON UPDATE action ]
[ [ NOT ] DEFERRABLE ]
[ INITIALLY checktime ]
```

The REFERENCES constraint specifies a rule that a column value is checked against the values of another column. REFERENCES can also be specified as part of a FOREIGN KEY table constraint.

Inputs

CONSTRAINT *name*

An arbitrary name for the constraint.

reftable

The table that contains the data to check against.

refcolumn

The column in *reftable* to check the data against. If this is not specified, the PRIMARY KEY of the *reftable* is used.

MATCH *matchtype*

There are three match types: MATCH FULL, MATCH PARTIAL, and a default match type if none is specified. MATCH FULL will not allow one column of a multi-column foreign key to be NULL unless all foreign key columns are NULL. The default MATCH type allows a some foreign key columns to be NULL while other parts of the foreign key are not NULL. MATCH PARTIAL is currently not supported.

ON DELETE *action*

The action to do when a referenced row in the referenced table is being deleted. There are the following actions.

NO ACTION

Produce error if foreign key violated. This is the default.

RESTRICT

Same as NO ACTION.

CASCADE

Delete any rows referencing the deleted row.

SET NULL

Set the referencing column values to NULL.

SET DEFAULT

Set the referencing column values to their default value.

ON UPDATE *action*

The action to do when a referenced column in the referenced table is being updated to a new value. If the row is updated, but the referenced column is not changed, no action is done. There are the following actions.

NO ACTION

Produce error if foreign key violated. This is the default.

RESTRICT

Same as NO ACTION.

CASCADE

Update the value of the referencing column to the new value of the referenced column.

SET NULL

Set the referencing column values to NULL.

SET DEFAULT

Set the referencing column values to their default value.

[NOT] DEFERRABLE

This controls whether the constraint can be deferred to the end of the transaction. If DEFERRABLE, SET CONSTRAINTS ALL DEFERRED will cause the foreign key to be checked only at the end of the transaction. NOT DEFERRABLE is the default.

INITIALLY *checktime*

checktime has two possible values which specify the default time to check the constraint.

DEFERRED

Check constraint only at the end of the transaction.

IMMEDIATE

Check constraint after each statement. This is the default.

Outputs

status

ERROR: *name* referential integrity violation - key referenced from *table* not found in *reftable*

This error occurs at runtime if one tries to insert a value into a column which does not have a matching column in the referenced table.

Description

The REFERENCES column constraint specifies that a column of a table must only contain values which match against values in a referenced column of a referenced table.

A value added to this column are matched against the values of the referenced table and referenced column using the given match type. In addition, when the referenced column data is changed, actions are run upon this column's matching data.

Notes

Currently Postgres only supports MATCH FULL and a default match type. In addition, the referenced columns are supposed to be the columns of a UNIQUE constraint in the referenced table, however Postgres does not enforce this.

Table CONSTRAINT Clause

```
[ CONSTRAINT name ] { PRIMARY KEY |  UNIQUE } ( column [, ...] )
[ CONSTRAINT name ] CHECK ( constraint )
[ CONSTRAINT name ] FOREIGN KEY ( column [, ...] )
                     REFERENCES reftable
                      (refcolumn [, ...] )
                      [ MATCH matchtype ]
                      [ ON DELETE action ]
                      [ ON UPDATE action ]
                      [ [ NOT ] DEFERRABLE ]
                      [ INITIALLY checktime ]
```

Inputs

CONSTRAINT *name*

An arbitrary name given to an integrity constraint.

column [, ...]

The column name(s) for which to define a unique index and, for PRIMARY KEY, a NOT NULL constraint.

CHECK (*constraint*)

A boolean expression to be evaluated as the constraint.

Outputs

The possible outputs for the table constraint clause are the same as for the corresponding portions of the column constraint clause.

Description

A table constraint is an integrity constraint defined on one or more columns of a base table. The four variations of "Table Constraint" are:

UNIQUE
CHECK
PRIMARY KEY
FOREIGN KEY

UNIQUE Constraint

```
[ CONSTRAINT name ] UNIQUE ( column [, ...] )
```

Inputs

CONSTRAINT *name*

An arbitrary name given to a constraint.

column

A name of a column in a table.

Outputs

status

ERROR: Cannot insert a duplicate key into a unique index

This error occurs at runtime if one tries to insert a duplicate value into a column.

Description

The UNIQUE constraint specifies a rule that a group of one or more distinct columns of a table may contain only unique values. The behavior of the UNIQUE table constraint is the same as that for column constraints, with the additional capability to span multiple columns.

See the section on the UNIQUE column constraint for more details.

Usage

Define a UNIQUE table constraint for the table distributors:

```
CREATE TABLE distributors (
    did      DECIMAL(03),
    name     VARCHAR(40),
    UNIQUE(name)
);
```

PRIMARY KEY Constraint

```
[ CONSTRAINT name ] PRIMARY KEY ( column [, ...] )
```

Inputs

CONSTRAINT *name*

An arbitrary name for the constraint.

column [, ...]

The names of one or more columns in the table.

Outputs

status

ERROR: Cannot insert a duplicate key into a unique index.

This occurs at run-time if one tries to insert a duplicate value into a column subject to a PRIMARY KEY constraint.

Description

The PRIMARY KEY constraint specifies a rule that a group of one or more distinct columns of a table may contain only unique, (non duplicate), non-null values. The column definitions of the specified columns do not have to include a NOT NULL constraint to be included in a PRIMARY KEY constraint.

The PRIMARY KEY table constraint is similar to that for column constraints, with the additional capability of encompassing multiple columns.

Refer to the section on the PRIMARY KEY column constraint for more information.

REFERENCES Constraint

```
[ CONSTRAINT name ] FOREIGN KEY ( column [, ...] )
    REFERENCES reftable [ ( refcolumn [, ...] ) ]
    [ MATCH matchtype ]
    [ ON DELETE action ]
    [ ON UPDATE action ]
    [ [ NOT ] DEFERRABLE ]
    [ INITIALLY checktime ]
```

The REFERENCES constraint specifies a rule that a column value is checked against the values of another column. REFERENCES can also be specified as part of a FOREIGN KEY table constraint.

Inputs

CONSTRAINT *name*

An arbitrary name for the constraint.

column [, ...]

The names of one or more columns in the table.

reftable

The table that contains the data to check against.

referenced column [, ...]

One or more column in the *reftable* to check the data against. If this is not specified, the PRIMARY KEY of the *reftable* is used.

MATCH *matchtype*

There are three match types: MATCH FULL, MATCH PARTIAL, and a default match type if none is specified. MATCH FULL will not allow one column of a multi-column foreign key to be NULL unless all foreign key columns are NULL. The default MATCH type allows a some foreign key columns to be NULL while other parts of the foreign key are not NULL. MATCH PARTIAL is currently not supported.

ON DELETE *action*

The action to do when a referenced row in the referenced table is being deleted. There are the following actions.

NO ACTION

Produce error if foreign key violated. This is the default.

RESTRICT

Same as NO ACTION.

CASCADE

Delete any rows referencing the deleted row.

SET NULL

Set the referencing column values to NULL.

SET DEFAULT

Set the referencing column values to their default value.

ON UPDATE *action*

The action to do when a referenced column in the referenced table is being updated to a new value. If the row is updated, but the referenced column is not changed, no action is done. There are the following actions.

NO ACTION

Produce error if foreign key violated. This is the default.

RESTRICT

Disallow update of row being referenced.

CASCADE

Update the value of the referencing column to the new value of the referenced column.

SET NULL

Set the referencing column values to NULL.

SET DEFAULT

Set the referencing column values to their default value.

[NOT] DEFERRABLE

This controls whether the constraint can be deferred to the end of the transaction. If DEFERRABLE, SET CONSTRAINTS ALL DEFERRED will cause the foreign key to be checked only at the end of the transaction. NOT DEFERRABLE is the default.

INITIALLY *checktime*

checktime has two possible values which specify the default time to check the constraint.

IMMEDIATE

Check constraint after each statement. This is the default.

DEFERRED

Check constraint only at the end of the transaction.

Outputs

status

ERROR: *name* referential integrity violation - key referenced from *table* not found in *reftable*

This error occurs at runtime if one tries to insert a value into a column which does not have a matching column in the referenced table.

Description

The FOREIGN KEY constraint specifies a rule that a group of one or more distinct columns of a table are related to a group of distinct columns in the referenced table.

The FOREIGN KEY table constraint is similar to that for column constraints, with the additional capability of encompassing multiple columns.

Refer to the section on the FOREIGN KEY column constraint for more information.

Usage

Create table films and table distributors:

```
CREATE TABLE films (
     code      CHARACTER(5) CONSTRAINT firstkey PRIMARY KEY,
     title     CHARACTER VARYING(40) NOT NULL,
     did       DECIMAL(3) NOT NULL,
     date_prod DATE,
     kind      CHAR(10),
     len       INTERVAL HOUR TO MINUTE
);


CREATE TABLE distributors (
     did      DECIMAL(03) PRIMARY KEY DEFAULT NEXTVAL('serial'),
     name     VARCHAR(40) NOT NULL CHECK (name <> '')
);
```

Create a table with a 2-dimensional array:

```
   CREATE TABLE array (
          vector INT[][]
          );
```

Define a UNIQUE table constraint for the table films. UNIQUE table constraints can be defined on one or more columns of the table:

```
CREATE TABLE films (
    code      CHAR(5),
    title     VARCHAR(40),
    did       DECIMAL(03),
    date_prod DATE,
    kind      CHAR(10),
    len       INTERVAL HOUR TO MINUTE,
    CONSTRAINT production UNIQUE(date_prod)
);
```

Define a CHECK column constraint:

```
CREATE TABLE distributors (
    did      DECIMAL(3) CHECK (did > 100),
    name     VARCHAR(40)
);
```

Define a CHECK table constraint:

```
CREATE TABLE distributors (
    did      DECIMAL(3),
    name     VARCHAR(40)
```

```
    CONSTRAINT con1 CHECK (did > 100 AND name > '')
);
```

Define a PRIMARY KEY table constraint for the table films. PRIMARY KEY table constraints can be defined on one or more columns of the table:

```
CREATE TABLE films (
    code      CHAR(05),
    title     VARCHAR(40),
    did       DECIMAL(03),
    date_prod DATE,
    kind      CHAR(10),
    len       INTERVAL HOUR TO MINUTE,
    CONSTRAINT code_title PRIMARY KEY(code,title)
);
```

Defines a PRIMARY KEY column constraint for table distributors. PRIMARY KEY column constraints can only be defined on one column of the table (the following two examples are equivalent):

```
CREATE TABLE distributors (
    did      DECIMAL(03),
    name     CHAR VARYING(40),
    PRIMARY KEY(did)
);


CREATE TABLE distributors (
    did      DECIMAL(03) PRIMARY KEY,
    name     VARCHAR(40)
);
```

Notes

CREATE TABLE/INHERITS is a Postgres language extension.

Compatibility

SQL92

In addition to the locally-visible temporary table, SQL92 also defines a CREATE GLOBAL TEMPORARY TABLE statement, and optionally an ON COMMIT clause:

```
CREATE GLOBAL TEMPORARY TABLE table ( column type [
    DEFAULT value ] [ CONSTRAINT column_constraint ] [, ...] )
    [ CONSTRAINT table_constraint ] [ ON COMMIT { DELETE |
 PRESERVE } ROWS ]
```

For temporary tables, the CREATE GLOBAL TEMPORARY TABLE statement names a new table visible to other clients and defines the table's columns and constraints.

The optional ON COMMIT clause of CREATE TEMPORARY TABLE specifies whether or not the temporary table should be emptied of rows whenever COMMIT is executed. If the ON COMMIT clause is omitted, the default option, ON COMMIT DELETE ROWS, is assumed.

To create a temporary table:

```
CREATE TEMPORARY TABLE actors (
    id          DECIMAL(03),
    name        VARCHAR(40),
    CONSTRAINT actor_id CHECK (id < 150)
) ON COMMIT DELETE ROWS;
```

UNIQUE clause

SQL92 specifies some additional capabilities for UNIQUE:

Table Constraint definition:

```
[ CONSTRAINT name ] UNIQUE ( column [, ...] )
    [ { INITIALLY DEFERRED | INITIALLY IMMEDIATE } ]
    [ [ NOT ] DEFERRABLE ]
```

Column Constraint definition:

```
[ CONSTRAINT name ] UNIQUE
      [ {INITIALLY DEFERRED | INITIALLY IMMEDIATE} ]
      [ [ NOT ] DEFERRABLE ]
```

NULL clause

The NULL "constraint" (actually a non-constraint) is a Postgres extension to SQL92 is included for symmetry with the NOT NULL clause. Since it is the default for any column, its presence is simply noise.

```
[ CONSTRAINT name ] NULL
```

NOT NULL clause

SQL92 specifies some additional capabilities for NOT NULL:

```
[ CONSTRAINT name ] NOT NULL
    [ {INITIALLY DEFERRED | INITIALLY IMMEDIATE} ]
    [ [ NOT ] DEFERRABLE ]
```

CONSTRAINT clause

SQL92 specifies some additional capabilities for constraints, and also defines assertions and domain constraints.

Note

Postgres does not yet support either domains or assertions.

An assertion is a special type of integrity constraint and share the same namespace as other constraints. However, an assertion is not necessarily dependent on one particular base table as constraints are, so SQL-92 provides the CREATE ASSERTION statement as an alternate method for defining a constraint:

```
CREATE ASSERTION name CHECK ( condition )
```

Domain constraints are defined by CREATE DOMAIN or ALTER DOMAIN statements:

Domain constraint:

```
[ CONSTRAINT name ] CHECK constraint
    [ {INITIALLY DEFERRED | INITIALLY IMMEDIATE} ]
    [ [ NOT ] DEFERRABLE ]
```

Table constraint definition:

```
[ CONSTRAINT name ] { PRIMARY KEY ( column, ... ) | FOREIGN
 KEY constraint | UNIQUE constraint | CHECK constraint }
    [ {INITIALLY DEFERRED | INITIALLY IMMEDIATE} ]
    [ [ NOT ] DEFERRABLE ]
```

Column constraint definition:

```
[ CONSTRAINT name ] { NOT NULL | PRIMARY KEY | FOREIGN
 KEY constraint | UNIQUE | CHECK constraint }
    [ {INITIALLY DEFERRED | INITIALLY IMMEDIATE} ]
    [ [ NOT ] DEFERRABLE ]
```

A CONSTRAINT definition may contain one deferment attribute clause and/or one initial constraint mode clause, in any order.

NOT DEFERRABLE

The constraint must be checked at the end of each statement. SET CONSTRAINTS ALL DEFERRED will have no effect on this type of constraint.

DEFERRABLE

This controls whether the constraint can be deferred to the end of the transaction. If SET CONSTRAINTS ALL DEFERRED is used or the constraint is set to INITIALLY DEFERRED, this will cause the foreign key to be checked only at the end of the transaction.

SET CONSTRAINT changes the foreign key constraint mode only for the current transaction.

INITIALLY IMMEDIATE

Check constraint only at the end of the transaction. This is the default

INITIALLY DEFERRED

Check constraint after each statement.

CHECK clause

SQL92 specifies some additional capabilities for CHECK in either table or column constraints.

table constraint definition:

```
[ CONSTRAINT name ] CHECK ( VALUE condition )
      [ {INITIALLY DEFERRED | INITIALLY IMMEDIATE} ]
      [ [ NOT ] DEFERRABLE ]
```

column constraint definition:

```
[ CONSTRAINT name ] CHECK ( VALUE condition )
```

```
    [ {INITIALLY DEFERRED | INITIALLY IMMEDIATE} ]
    [ [ NOT ] DEFERRABLE ]
```

PRIMARY KEY clause

SQL92 specifies some additional capabilities for PRIMARY KEY:

Table Constraint definition:

```
[ CONSTRAINT name ] PRIMARY KEY ( column [, ...] )
    [ {INITIALLY DEFERRED | INITIALLY IMMEDIATE} ]
    [ [ NOT ] DEFERRABLE ]
```

Column Constraint definition:

```
[ CONSTRAINT name ] PRIMARY KEY
    [ {INITIALLY DEFERRED | INITIALLY IMMEDIATE} ]
    [ [ NOT ] DEFERRABLE ]
```

Name

CREATE TABLE AS — Creates a new table

Synopsis

<refsynopsisdivinfo>1999-07-20</refsynopsisdivinfo>

```
CREATE TABLE table [ (column [, ...] ) ]
     AS select_clause
```

Inputs

table

The name of a new table to be created.

column

The name of a column. Multiple column names can be specified using a comma-delimited list of column names.

select_clause

A valid query statement. Refer to SELECT for a description of the allowed syntax.

Outputs

Refer to `CREATE TABLE` and `SELECT` for a summary of possible output messages.

Description

`CREATE TABLE AS` enables a table to be created from the contents of an existing table. It is functionality equivalent to SELECT INTO , but with perhaps a more direct syntax.

Name

CREATE TRIGGER — Creates a new trigger

Synopsis

<refsynopsisdivinfo>2000-03-25</refsynopsisdivinfo>

```
CREATE TRIGGER name { BEFORE | AFTER } { event [OR ...] }
    ON table FOR EACH { ROW | STATEMENT }
    EXECUTE PROCEDURE func ( arguments )
```

Inputs

name

The name of an existing trigger.

table

The name of a table.

event

One of INSERT, DELETE or UPDATE.

funcname

A user-supplied function.

Outputs

CREATE

This message is returned if the trigger is successfully created.

Description

CREATE TRIGGER will enter a new trigger into the current data base. The trigger will be associated with the relation *relname* and will execute the specified function *funcname*.

The trigger can be specified to fire either before BEFORE the operation is attempted on a tuple (before constraints are checked and the INSERT, UPDATE or DELETE is attempted) or AFTER the operation has been attempted (e.g. after constraints are checked and the INSERT, UPDATE or DELETE has completed). If the trigger fires before the event, the trigger may skip the operation for the current tuple, or change the tuple being inserted (for INSERT and UPDATE operations only). If the trigger fires after the event, all changes, including the last insertion, update, or deletion, are "visible" to the trigger.

Refer to the chapters on SPI and Triggers in the *PostgreSQL Programmer's Guide* for more information.

Notes

CREATE TRIGGER is a Postgres language extension.

Only the relation owner may create a trigger on this relation.

As of the current release (v7.0), STATEMENT triggers are not implemented.

Refer to DROP TRIGGER for information on how to remove triggers.

Usage

Check if the specified distributor code exists in the distributors table before appending or updating a row in the table films:

```
CREATE TRIGGER if_dist_exists
    BEFORE INSERT OR UPDATE ON films FOR EACH ROW
    EXECUTE PROCEDURE check_primary_key ('did', 'distributors',
 'did');
```

Before cancelling a distributor or updating its code, remove every reference to the table films:

```
CREATE TRIGGER if_film_exists
    BEFORE DELETE OR UPDATE ON distributors FOR EACH ROW
    EXECUTE PROCEDURE check_foreign_key (1, 'CASCADE', 'did',
 'films', 'did');
```

Compatibility

SQL92

There is no `CREATE TRIGGER` in SQL92.

The second example above may also be done by using a FOREIGN KEY constraint as in:

```
CREATE TABLE distributors (
    did      DECIMAL(3),
    name     VARCHAR(40),
    CONSTRAINT if_film_exists
    FOREIGN KEY(did) REFERENCES films
    ON UPDATE CASCADE ON DELETE CASCADE
);
```

Name

CREATE TYPE — Defines a new base data type

Synopsis

<refsynopsisdivinfo>1999-07-20</refsynopsisdivinfo>

```
CREATE TYPE typename ( INPUT = input_function, OUTPUT
 = output_function
      , INTERNALLENGTH = { internallength | VARIABLE } [ ,
 EXTERNALLENGTH = { externallength | VARIABLE } ]
    [ , DEFAULT = "default" ]
    [ , ELEMENT = element ] [ , DELIMITER = delimiter ]
    [ , SEND = send_function ] [ , RECEIVE = receive_function ]
    [ , PASSEDBYVALUE ] )
```

Inputs

typename

The name of a type to be created.

internallength

A literal value, which specifies the internal length of the new type.

externallength

A literal value, which specifies the external length of the new type.

input_function

The name of a function, created by `CREATE FUNCTION`, which converts data from its external form to the type's internal form.

output_function

The name of a function, created by CREATE FUNCTION, which converts data from its internal form to a form suitable for display.

element

The type being created is an array; this specifies the type of the array elements.

delimiter

The delimiter character for the array.

default

The default text to be displayed to indicate "data not present"

send_function

The name of a function, created by `CREATE FUNCTION`, which converts data of this type into a form suitable for transmission to another machine.

receive_function

The name of a function, created by `CREATE FUNCTION`, which converts data of this type from a form suitable for transmission from another machine to internal form.

Outputs

`CREATE`

Message returned if the type is successfully created.

Description

`CREATE TYPE` allows the user to register a new user data type with Postgres for use in the current data base. The user who defines a type becomes its owner. *`typename`* is the name of the new type and must be unique within the types defined for this database.

`CREATE TYPE` requires the registration of two functions (using create function) before defining the type. The representation of a new base type is determined by *`input_function`*, which converts the type's external representation to an internal representation usable by the operators and functions defined for the type. Naturally, *`output_function`* performs the reverse transformation. Both the input and output functions must be declared to take one or two arguments of type "`opaque`".

New base data types can be fixed length, in which case *`internallength`* is a positive integer, or variable length, in which case Postgres assumes that the new type has the same format as the Postgres-supplied data type, "`text`". To indicate that a type is variable-length, set *`internallength`* to `VARIABLE`. The external representation is similarly specified using the *`externallength`* keyword.

To indicate that a type is an array and to indicate that a type has array elements, indicate the type of the array element using the element keyword. For example, to define an array of 4 byte integers ("int4"), specify

```
ELEMENT = int4
```

To indicate the delimiter to be used on arrays of this type, *`delimiter`* can be set to a specific character. The default delimiter is the comma ("`,`").

A default value is optionally available in case a user wants some specific bit pattern to mean "data not present." Specify the default with the `DEFAULT` keyword.

The optional arguments *`send_function`* and *`receive_function`* are used when the application program requesting Postgres services resides on a different machine. In this case, the machine on which Postgres runs may use a format for the data type different from that used on the remote machine. In this case it is appropriate to convert data items to a standard form when sending from the server to the client and converting from the standard format to the machine specific format when the server receives the data from the client. If these functions are not specified, then it is assumed that the internal format of the type is acceptable on all relevant machine architectures. For example, single characters do not have to be converted if passed from a Sun-4 to a DECstation, but many other types do.

The optional flag, `PASSEDBYVALUE`, indicates that operators and functions which use this data type should be passed an argument by value rather than by reference. Note that you may not pass by value types whose internal representation is more than four bytes.

For new base types, a user can define operators, functions and aggregates using the appropriate facilities described in this section.

Array Types

Two generalized built-in functions, array_in and array_out, exist for quick creation of variable-length array types. These functions operate on arrays of any existing Postgres type.

Large Object Types

A "regular" Postgres type can only be 8192 bytes in length. If you need a larger type you must create a Large Object type. The interface for these types is discussed at length in the *PostgreSQL Programmer's Guide*. The length of all large object types is always VARIABLE.

Examples

This command creates the box data type and then uses the type in a class definition:

```
CREATE TYPE box (INTERNALLENGTH = 8,
    INPUT = my_procedure_1, OUTPUT = my_procedure_2);
CREATE TABLE myboxes (id INT4, description box);
```

This command creates a variable length array type with integer elements:

```
CREATE TYPE int4array (INPUT = array_in, OUTPUT = array_out,
    INTERNALLENGTH = VARIABLE, ELEMENT = int4);
CREATE TABLE myarrays (id int4, numbers int4array);
```

This command creates a large object type and uses it in a class definition:

```
CREATE TYPE bigobj (INPUT = lo_filein, OUTPUT = lo_fileout,
    INTERNALLENGTH = VARIABLE);
CREATE TABLE big_objs (id int4, obj bigobj);
```

Notes

Type names cannot begin with the underscore character ("_") and can only be 31 characters long. This is because Postgres silently creates an array type for each base type with a name consisting of the base type's name prepended with an underscore.

Refer to `DROP TYPE` to remove an existing type.

See also `CREATE FUNCTION`, `CREATE OPERATOR` and the chapter on Large Objects in the *PostgreSQL Programmer's Guide*.

Compatibility

SQL3

`CREATE TYPE` is an SQL3 statement.

Name

CREATE USER — Creates a new database user

Synopsis

<refsynopsisdivinfo>1999-07-20</refsynopsisdivinfo>

```
CREATE USER username
    [ WITH
     [ SYSID uid ]
     [ PASSWORD 'password' ] ]
    [ CREATEDB    | NOCREATEDB ] [ CREATEUSER | NOCREATEUSER ]
    [ IN GROUP     groupname [, ...] ]
    [ VALID UNTIL  'abstime' ]
```

Inputs

username

The name of the user.

uid

The SYSID clause can be used to choose the Postgres user id of the user that is being created. It is not at all necessary that those match the UNIX user ids, but some people choose to keep the numbers the same.

If this is not specified, the highest assigned user id plus one will be used as default.

password

Sets the user's password. If you do not plan to use password authentication you can omit this option, otherwise the user won't be able to connect to a password-authenticated server. See pg_hba.conf(5) or the administrator's guide for details on how to set up authentication mechanisms.

CREATEDB
NOCREATEDB

These clauses define a user's ability to create databases. If CREATEDB is specified, the user being defined will be allowed to create his own databases. Using NOCREATEDB will deny a user the ability to create databases. If this clause is omitted, NOCREATEDB is used by default.

CREATEUSER
NOCREATEUSER

These clauses determine whether a user will be permitted to create new users himself. This option will also make the user a superuser who can override all access restrictions. Omitting this clause will set the user's value of this attribute to be NOCREATEUSER.

groupname

A name of a group into which to insert the user as a new member.

abstime

The VALID UNTIL clause sets an absolute time after which the user's password is no longer valid. If this clause is omitted the login will be valid for all time.

Outputs

`CREATE USER`

Message returned if the command completes successfully.

Description

CREATE USER will add a new user to an instance of Postgres. Refer to the adminstrator's guide for information about managing users and authentication. You must be a database superuser to use this command.

Use ALTER USER to change a user's password and privileges, and DROP USER to remove a user. Use `ALTER GROUP` to add or remove the user from other groups. Postgres comes with a script createuser which has the same functionality as this command (in fact, it calls this command) but can be run from the command shell.

Usage

Create a user with no password:

```
CREATE USER jonathan
```

Create a user with a password:

```
CREATE USER davide WITH PASSWORD 'jw8s0F4'
```

Create a user with a password, whose account is valid until the end of 2001. Note that after one second has ticked in 2002, the account is not valid:

```
CREATE USER miriam WITH PASSWORD 'jw8s0F4' VALID UNTIL 'Jan 1 2002'
```

Create an account where the user can create databases:

```
CREATE USER manuel WITH PASSWORD 'jw8s0F4' CREATEDB
```

Compatibility

SQL92

There is no `CREATE USER` statement in SQL92.

Name

CREATE VIEW — Constructs a virtual table

Synopsis

<refsynopsisdivinfo>2000-03-25</refsynopsisdivinfo>

```
CREATE VIEW view AS SELECT query
```

Inputs

view

The name of a view to be created.

query

An SQL query which will provide the columns and rows of the view.

Refer to the SELECT statement for more information about valid arguments.

Outputs

`CREATE`

The message returned if the view is successfully created.

`ERROR: Relation 'view' already exists`

This error occurs if the view specified already exists in the database.

`NOTICE create: attribute named "column" has an unknown type`

The view will be created having a column with an unknown type if you do not specify it. For example, the following command gives a warning:

```
CREATE VIEW vista AS SELECT 'Hello World'
```

whereas this command does not:

```
CREATE VIEW vista AS SELECT text 'Hello World'
```

Description

`CREATE VIEW` will define a view of a table or class. This view is not physically materialized. Specifically, a query rewrite retrieve rule is automatically generated to support retrieve operations on views.

Notes

Currently, views are read only.

Use the `DROP VIEW` statement to drop views.

Usage

Create a view consisting of all Comedy films:

```
CREATE VIEW kinds AS
    SELECT *
    FROM films
    WHERE kind = 'Comedy';

SELECT * FROM kinds;

 code  |           title           | did | date_prod  |  kind  |
 len
-------+---------------------------+-----+------------+--------
+-------
 UA502 | Bananas                   | 105 | 1971-07-13 | Comedy |
 01:22
 C_701 | There's a Girl in my Soup | 107 | 1970-06-11 | Comedy |
 01:36
(2 rows)
```

Compatibility

SQL92

SQL92 specifies some additional capabilities for the `CREATE VIEW` statement:

```
CREATE VIEW view [ column [, ...] ]
    AS SELECT expression [ AS colname ] [, ...]
    FROM table [ WHERE condition ]
    [ WITH [ CASCADE | LOCAL ] CHECK OPTION ]
```

The optional clauses for the full SQL92 command are:

CHECK OPTION

This option is to do with updatable views. All INSERTs and UPDATEs on the view will be checked to ensure data satisfy the view-defining condition. If they do not, the update will be rejected.

LOCAL

Check for integrity on this view.

CASCADE

Check for integrity on this view and on any dependent view. CASCADE is assumed if neither CASCADE nor LOCAL is specified.

Name

DECLARE — Defines a cursor for table access

Synopsis

<refsynopsisdivinfo>1999-07-20</refsynopsisdivinfo>

```
DECLARE cursorname [ BINARY ] [ INSENSITIVE ] [ SCROLL ]
    CURSOR FOR query
    [ FOR { READ ONLY | UPDATE [ OF column [, ...] ] ]
```

Inputs

cursorname

The name of the cursor to be used in subsequent FETCH operations..

BINARY

Causes the cursor to fetch data in binary rather than in text format.

INSENSITIVE

SQL92 keyword indicating that data retrieved from the cursor should be unaffected by updates from other processes or cursors. Since cursor operations occur within transactions in Postgres this is always the case. This keyword has no effect.

SCROLL

SQL92 keyword indicating that data may be retrieved in multiple rows per FETCH operation. Since this is allowed at all times by Postgres this keyword has no effect.

query

An SQL query which will provide the rows to be governed by the cursor. Refer to the SELECT statement for further information about valid arguments.

READ ONLY

SQL92 keyword indicating that the cursor will be used in a readonly mode. Since this is the only cursor access mode available in Postgres this keyword has no effect.

UPDATE

SQL92 keyword indicating that the cursor will be used to update tables. Since cursor updates are not currently supported in Postgres this keyword provokes an informational error message.

column

Column(s) to be updated. Since cursor updates are not currently supported in Postgres the UPDATE clause provokes an informational error message.

Outputs

SELECT

The message returned if the SELECT is run successfully.

NOTICE BlankPortalAssignName: portal "*cursorname*" already exists

This error occurs if *cursorname* is already declared.

```
ERROR: Named portals may only be used in begin/end transaction blocks
```

This error occurs if the cursor is not declared within a transaction block.

Description

`DECLARE` allows a user to create cursors, which can be used to retrieve a small number of rows at a time out of a larger query. Cursors can return data either in text or in binary format using FETCH .

Normal cursors return data in text format, either ASCII or another encoding scheme depending on how the Postgres backend was built. Since data is stored natively in binary format, the system must do a conversion to produce the text format. In addition, text formats are often larger in size than the corresponding binary format. Once the information comes back in text form, the client application may need to convert it to a binary format to manipulate it. BINARY cursors give you back the data in the native binary representation.

As an example, if a query returns a value of one from an integer column, you would get a string of '1' with a default cursor whereas with a binary cursor you would get a 4-byte value equal to control-A ('^A').

BINARY cursors should be used carefully. User applications such as psql are not aware of binary cursors and expect data to come back in a text format.

String representation is architecture-neutral whereas binary representation can differ between different machine architectures and *Postgres does not resolve byte ordering or representation issues for binary cursors*. Therefore, if your client machine and server machine use different representations (e.g. "big-endian" versus "little-endian"), you will probably not want your data returned in binary format. However, binary cursors may be a little more efficient since there is less conversion overhead in the server to client data transfer.

Tip

If you intend to display the data in ASCII, getting it back in ASCII will save you some effort on the client side.

Notes

Cursors are only available in transactions. Use to BEGIN , COMMIT and ROLLBACK to define a transaction block.

In SQL92 cursors are only available in embedded SQL (ESQL) applications. The Postgres backend does not implement an explicit `OPEN cursor` statement; a cursor is considered to be open when it is declared. However, ecpg, the embedded SQL preprocessor for Postgres, supports the SQL92 cursor conventions, including those involving DECLARE and OPEN statements.

Usage

To declare a cursor:

```
DECLARE liahona CURSOR
    FOR SELECT * FROM films;
```

Compatibility

SQL92

SQL92 allows cursors only in embedded SQL and in modules. Postgres permits cursors to be used interactively. SQL92 allows embedded or modular cursors to update database information. All Postgres cursors are readonly. The BINARY keyword is a Postgres extension.

Name

DELETE — Removes rows from a table

Synopsis

<refsynopsisdivinfo>1999-07-20</refsynopsisdivinfo>

```
DELETE FROM table [ WHERE condition ]
```

Inputs

table

The name of an existing table.

condition

This is an SQL selection query which returns the rows which are to be deleted.

Refer to the SELECT statement for further description of the WHERE clause.

Outputs

DELETE *count*

Message returned if items are successfully deleted. The *count* is the number of rows deleted.

If *count* is 0, no rows were deleted.

Description

DELETE removes rows which satisfy the WHERE clause from the specified table.

If the *condition* (WHERE clause) is absent, the effect is to delete all rows in the table. The result is a valid, but empty table.

Tip

TRUNCATE is a Postgres extension which provides a faster mechanism to remove all rows from a table.

You must have write access to the table in order to modify it, as well as read access to any table whose values are read in the *condition*.

Usage

Remove all films but musicals:

```
DELETE FROM films WHERE kind <> 'Musical';
SELECT * FROM films;

 code  |           title           | did | date_prod  |  kind   |
 len
-------+---------------------------+-----+------------+---------
+-------
 UA501 | West Side Story           | 105 | 1961-01-03 | Musical |
 02:32
```

```
 TC901 | The King and I            | 109 | 1956-08-11 | Musical |
 02:13
 WD101 | Bed Knobs and Broomsticks | 111 |            | Musical |
 01:57
(3 rows)
```

Clear the table `films`:

```
DELETE FROM films;
SELECT * FROM films;

 code | title | did | date_prod | kind | len
------+-------+-----+-----------+------+-----
(0 rows)
```

Compatibility

SQL92

SQL92 allows a positioned DELETE statement:

```
DELETE FROM table WHERE
    CURRENT OF cursor
```

where *`cursor`* identifies an open cursor. Interactive cursors in Postgres are read-only.

Name

DROP AGGREGATE — Removes the definition of an aggregate function

Synopsis

<refsynopsisdivinfo>1999-07-20</refsynopsisdivinfo>

```
DROP AGGREGATE name type
```

Inputs

name

The name of an existing aggregate function.

type

The type of an existing aggregate function. (Refer to the *PostgreSQL User's Guide* for further information about data types).

Outputs

DROP

Message returned if the command is successful.

NOTICE RemoveAggregate: aggregate '*agg*' for '*type*' does not exist

This message occurs if the aggregate function specified does not exist in the database.

Description

DROP AGGREGATE will remove all references to an existing aggregate definition. To execute this command the current user must be the owner of the aggregate.

Notes

Use CREATE AGGREGATE to create aggregate functions.

Usage

To remove the myavg aggregate for type int4:

```
DROP AGGREGATE myavg int4;
```

Compatibility

SQL92

There is no DROP AGGREGATE statement in SQL92; the statement is a Postgres language extension.

Name

DROP DATABASE — Removes an existing database

Synopsis

<refsynopsisdivinfo>1999-12-11</refsynopsisdivinfo>

```
DROP DATABASE name
```

Inputs

name

The name of an existing database to remove.

Outputs

`DROP DATABASE`

This message is returned if the command is successful.

`ERROR: user 'username' is not allowed to create/drop databases`

You must have the special CREATEDB privilege to drop databases. See CREATE USER .

`ERROR: dropdb: cannot be executed on the template database`

The `template1` database cannot be removed. It's not in your interest.

`ERROR: dropdb: cannot be executed on an open database`

You cannot be connected to the the database your are about to remove. Instead, you could connect to `template1` or any other database and run this command again.

`ERROR: dropdb: database 'name' does not exist`

This message occurs if the specified database does not exist.

`ERROR: dropdb: database 'name' is not owned by you`

You must be the owner of the database. Being the owner usually means that you created it as well.

`ERROR: dropdb: May not be called in a transaction block.`

You must finish the transaction in progress before you can call this command.

`NOTICE: The database directory 'xxx' could not be removed.`

The database was dropped (unless other error messages came up), but the directory where the data is stored could not be removed. You must delete it manually.

Description

`DROP DATABASE` removes the catalog entries for an existing database and deletes the directory containing the data. It can only be executed by the database owner (usually the user that created it).

Notes

This command cannot be executed while connected to the target database. Thus, it might be more convenient to use the shell script dropdb , which is a wrapper around this command, instead.

Refer to CREATE DATABASE for information on how to create a database.

Compatibility

SQL92

`DROP DATABASE` statement is a Postgres language extension; there is no such command in SQL92.

Name

DROP FUNCTION — Removes a user-defined C function

Synopsis

<refsynopsisdivinfo>1999-07-20</refsynopsisdivinfo>

```
DROP FUNCTION name ( [ type [, ...] ] )
```

Inputs

name

The name of an existing function.

type

The type of function parameters.

Outputs

DROP

Message returned if the command completes successfully.

NOTICE RemoveFunction: Function "*name*" ("*types*") does not exist

This message is given if the function specified does not exist in the current database.

Description

DROP FUNCTION will remove references to an existing C function. To execute this command the user must be the owner of the function. The input argument types to the function must be specified, as only the function with the given name and argument types will be removed.

Notes

Refer to CREATE FUNCTION for information on creating aggregate functions.

No checks are made to ensure that types, operators or access methods that rely on the function have been removed first.

Usage

This command removes the square root function:

```
DROP FUNCTION sqrt(int4);
```

Compatibility

SQL92

DROP FUNCTION is a Postgres language extension.

SQL/PSM

SQL/PSM is a proposed standard to enable function extensibility. The SQL/PSM DROP FUNCTION statement has the following syntax:

```
DROP [ SPECIFIC ] FUNCTION name { RESTRICT | CASCADE }
```

Name

DROP GROUP — Removes a group

Synopsis

<refsynopsisdivinfo>2000-01-14</refsynopsisdivinfo>

```
DROP GROUP name
```

Inputs

name

The name of an existing group.

Outputs

`DROP GROUP`

The message returned if the group is successfully deleted.

Description

`DROP GROUP` removes the specified group from the database. The users in the group are not deleted.

Use CREATE GROUP to add new groups, and ALTER GROUP to change a group's membership.

Usage

To drop a group:

```
DROP GROUP staff;
```

Compatibility

SQL92

There is no `DROP GROUP` in SQL92.

Name

DROP INDEX — Removes an index from a database

Synopsis

<refsynopsisdivinfo>1999-07-20</refsynopsisdivinfo>

```
DROP INDEX index_name
```

Inputs

index_name

The name of the index to remove.

Outputs

`DROP`

The message returned if the index is successfully dropped.

`ERROR: index "index_name" nonexistent`

This message occurs if *index_name* is not an index in the database.

Description

`DROP INDEX` drops an existing index from the database system. To execute this command you must be the owner of the index.

Notes

`DROP INDEX` is a Postgres language extension.

Refer to CREATE INDEX for information on how to create indexes.

Usage

This command will remove the `title_idx` index:

```
DROP INDEX title_idx;
```

Compatibility

SQL92

SQL92 defines commands by which to access a generic relational database. Indexes are an implementation-dependent feature and hence there are no index-specific commands or definitions in the SQL92 language.

Name

DROP LANGUAGE — Removes a user-defined procedural language

Synopsis

<refsynopsisdivinfo>1999-07-20</refsynopsisdivinfo>

```
DROP PROCEDURAL LANGUAGE 'name'
```

Inputs

name

The name of an existing procedural language.

Outputs

DROP

This message is returned if the language is successfully dropped.

ERROR: Language "*name*" doesn't exist

This message occurs if a language called *name* is not found in the database.

Description

DROP PROCEDURAL LANGUAGE will remove the definition of the previously registered procedural language called *name*.

Notes

The DROP PROCEDURAL LANGUAGE statement is a Postgres language extension.

Refer to CREATE LANGUAGE for information on how to create procedural languages.

No checks are made if functions or trigger procedures registered in this language still exist. To re-enable them without having to drop and recreate all the functions, the pg_proc's prolang attribute of the functions must be adjusted to the new object ID of the recreated pg_language entry for the PL.

Usage

This command removes the PL/Sample language:

```
DROP PROCEDURAL LANGUAGE 'plsample';
```

Compatibility

SQL92

There is no DROP PROCEDURAL LANGUAGE in SQL92.

Name

DROP OPERATOR — Removes an operator from the database

Synopsis

<refsynopsisdivinfo>1999-07-20</refsynopsisdivinfo>

```
DROP OPERATOR id ( type | NONE [,...] )
```

Inputs

id

The identifier of an existing operator.

type

The type of function parameters.

Outputs

`DROP`

The message returned if the command is successful.

`ERROR: RemoveOperator: binary operator 'oper' taking 'type' and 'type2' does not exist`

This message occurs if the specified binary operator does not exist.

`ERROR: RemoveOperator: left unary operator 'oper' taking 'type' does not exist`

This message occurs if the specified left unary operator specified does not exist.

`ERROR: RemoveOperator: right unary operator 'oper' taking 'type' does not exist`

This message occurs if the specified right unary operator specified does not exist.

Description

`DROP OPERATOR` drops an existing operator from the database. To execute this command you must be the owner of the operator.

The left or right type of a left or right unary operator, respectively, may be specified as `NONE`.

Notes

The `DROP OPERATOR` statement is a Postgres language extension.

Refer to CREATE OPERATOR for information on how to create operators.

It is the user's responsibility to remove any access methods and operator classes that rely on the deleted operator.

Usage

Remove power operator `a^n` for `int4`:

```
DROP OPERATOR ^ (int4, int4);
```

Remove left unary negation operator (`b !`) for booleans:

```
DROP OPERATOR ! (none, bool);
```

Remove right unary factorial operator (`! i`) for `int4`:

```
DROP OPERATOR ! (int4, none);
```

Compatibility

SQL92

There is no `DROP OPERATOR` in SQL92.

Name

DROP RULE — Removes an existing rule from the database

Synopsis

<refsynopsisdivinfo>1998-09-22</refsynopsisdivinfo>

```
DROP RULE name
```

Inputs

name

The name of an existing rule to drop.

Outputs

`DROP`

Message returned if successfully.

`ERROR: RewriteGetRuleEventRel: rule "name" not found`

This message occurs if the specified rule does not exist.

Description

`DROP RULE` drops a rule from the specified Postgres rule system. Postgres will immediately cease enforcing it and will purge its definition from the system catalogs.

Notes

The `DROP RULE` statement is a Postgres language extension.

Refer to `CREATE RULE` for information on how to create rules.

Once a rule is dropped, access to historical information the rule has written may disappear.

Usage

To drop the rewrite rule `newrule`:

```
DROP RULE newrule;
```

Compatibility

SQL92

There is no `DROP RULE` in SQL92.

Name

DROP SEQUENCE — Removes an existing sequence

Synopsis

<refsynopsisdivinfo>1999-07-20</refsynopsisdivinfo>

```
DROP SEQUENCE name [, ...]
```

Inputs

name

The name of a sequence.

Outputs

`DROP`

The message returned if the sequence is successfully dropped.

`NOTICE: Relation "name" does not exist.`

This message occurs if the specified sequence does not exist.

Description

`DROP SEQUENCE` removes sequence number generators from the data base. With the current implementation of sequences as special tables it works just like the `DROP TABLE` statement.

Notes

The `DROP SEQUENCE` statement is a Postgres language extension.

Refer to the `CREATE SEQUENCE` statement for information on how to create a sequence.

Usage

To remove sequence `serial` from database:

```
DROP SEQUENCE serial;
```

Compatibility

SQL92

There is no `DROP SEQUENCE` in SQL92.

Name

DROP TABLE — Removes existing tables from a database

Synopsis

<refsynopsisdivinfo>1999-07-20</refsynopsisdivinfo>

```
DROP TABLE name [, ...]
```

Inputs

name

The name of an existing table or view to drop.

Outputs

`DROP`

The message returned if the command completes successfully.

`ERROR Relation "name" Does Not Exist!`

If the specified table or view does not exist in the database.

Description

`DROP TABLE` removes tables and views from the database. Only its owner may destroy a table or view. A table may be emptied of rows, but not destroyed, by using `DELETE`.

If a table being destroyed has secondary indexes on it, they will be removed first. The removal of just a secondary index will not affect the contents of the underlying table.

Notes

Refer to `CREATE TABLE` and `ALTER TABLE` for information on how to create or modify tables.

Usage

To destroy two tables, `films` and `distributors`:

```
DROP TABLE films, distributors;
```

Compatibility

SQL92

SQL92 specifies some additional capabilities for DROP TABLE:

```
DROP TABLE table { RESTRICT | CASCADE }
```

RESTRICT

Ensures that only a table with no dependent views or integrity constraints can be destroyed.

CASCADE

Any referencing views or integrity constraints will also be dropped.

Tip

At present, to remove a referenced view you must drop it explicitly.

Name

DROP TRIGGER — Removes the definition of a trigger

Synopsis

<refsynopsisdivinfo>1998-09-22</refsynopsisdivinfo>

```
DROP TRIGGER name ON table
```

Inputs

name

The name of an existing trigger.

table

The name of a table.

Outputs

DROP

The message returned if the trigger is successfully dropped.

ERROR: DropTrigger: there is no trigger *name* on relation "*table*"

This message occurs if the trigger specified does not exist.

Description

DROP TRIGGER will remove all references to an existing trigger definition. To execute this command the current user must be the owner of the trigger.

Notes

DROP TRIGGER is a Postgres language extension.

Refer to CREATE TRIGGER for information on how to create triggers.

Usage

Destroy the if_dist_exists trigger on table films:

```
DROP TRIGGER if_dist_exists ON films;
```

Compatibility

SQL92

There is no DROP TRIGGER statement in SQL92.

Name

DROP TYPE — Removes a user-defined type from the system catalogs

Synopsis

<refsynopsisdivinfo>1999-07-20</refsynopsisdivinfo>

```
DROP TYPE typename
```

Inputs

typename

The name of an existing type.

Outputs

DROP

The message returned if the command is successful.

ERROR: RemoveType: type '*typename*' does not exist

This message occurs if the specified type is not found.

Description

DROP TYPE will remove a user type from the system catalogs.

Only the owner of a type can remove it.

Notes

DROP TYPE statement is a Postgres language extension.

Refer to CREATE TYPE for inforamation on how to create types.

It is the user's responsibility to remove any operators, functions, aggregates, access methods, subtypes, and classes that use a deleted type.

If a built-in type is removed, the behavior of the backend is unpredictable.

Usage

To remove the box type:

```
DROP TYPE box;
```

Compatibility

SQL3

DROP TYPE is a SQL3 statement.

Name

DROP USER — Removes a user

Synopsis

<refsynopsisdivinfo>1999-07-20</refsynopsisdivinfo>

```
DROP USER name
```

Inputs

name

The name of an existing user.

Outputs

`DROP USER`

The message returned if the user is successfully deleted.

`ERROR: DROP USER: user "name" does not exist`

This message occurs if the username is not found.

`DROP USER: user "name" owns database "name", cannot be removed`

You must drop the database first or change its ownership.

Description

`DROP USER` removes the specified user from the database. It does not remove tables, views, or other objects owned by the user. If the user owns any database you get an error.

Use CREATE USER to add new users, and ALTER USER to change a user's properties. Postgres comes with a script dropuser which has the same functionality as this command (in fact, it calls this command) but can be run from the command shell.

Usage

To drop a user account:

```
DROP USER jonathan;
```

Compatibility

SQL92

There is no `DROP USER` in SQL92.

Name

DROP VIEW — Removes an existing view from a database

Synopsis

<refsynopsisdivinfo>1999-07-20</refsynopsisdivinfo>

```
DROP VIEW name
```

Inputs

name

The name of an existing view.

Outputs

`DROP`

The message returned if the command is successful.

`ERROR: RewriteGetRuleEventRel: rule "_RETname" not found`

This message occurs if the specified view does not exist in the database.

Description

`DROP VIEW` drops an existing view from the database. To execute this command you must be the owner of the view.

Notes

The Postgres `DROP TABLE` statement also drops views.

Refer to `CREATE VIEW` for information on how to create views.

Usage

This command will remove the view called `kinds`:

```
DROP VIEW kinds;
```

Compatibility

SQL92

SQL92 specifies some additional capabilities for `DROP VIEW`:

```
DROP VIEW view { RESTRICT | CASCADE }
```

Inputs

RESTRICT

Ensures that only a view with no dependent views or integrity constraints can be destroyed.

CASCADE

Any referencing views and integrity constraints will be dropped as well.

Notes

At present, to remove a referenced view from a Postgres database, you must drop it explicitly.

Name

END — Commits the current transaction

Synopsis

<refsynopsisdivinfo>1999-07-20</refsynopsisdivinfo>

```
END [ WORK | TRANSACTION ]
```

Inputs

WORK
TRANSACTION

Optional keywords. They have no effect.

Outputs

`COMMIT`

Message returned if the transaction is successfully committed.

```
NOTICE: COMMIT: no transaction in progress
```

If there is no transaction in progress.

Description

`END` is a Postgres extension, and is a synonym for the SQL92-compatible COMMIT .

Notes

The keywords WORK and TRANSACTION are noise and can be omitted.

Use ROLLBACK to abort a transaction.

Usage

To make all changes permanent:

```
END WORK;
```

Compatibility

SQL92

`END` is a PostgreSQL extension which provides functionality equivalent to COMMIT .

Name

EXPLAIN — Shows statement execution plan

Synopsis

<refsynopsisdivinfo>1999-07-20</refsynopsisdivinfo>

```
EXPLAIN [ VERBOSE ] query
```

Inputs

VERBOSE

Flag to show detailed query plan.

query

Any *query*.

Outputs

```
NOTICE: QUERY PLAN: plan
```

Explicit query plan from the Postgres backend.

```
EXPLAIN
```

Flag sent after query plan is shown.

Description

This command displays the execution plan that the Postgres planner generates for the supplied query. The execution plan shows how the table(s) referenced by the query will be scanned --- by plain sequential scan, index scan etc --- and if multiple tables are referenced, what join algorithms will be used to bring together the required tuples from each input table.

The most critical part of the display is the estimated query execution cost, which is the planner's guess at how long it will take to run the query (measured in units of disk page fetches). Actually two numbers are shown: the startup time before the first tuple can be returned, and the total time to return all the tuples. For most queries the total time is what matters, but in contexts such as an EXISTS sub-query the planner will choose the smallest startup time instead of the smallest total time (since the executor will stop after getting one tuple, anyway). Also, if you limit the number of tuples to return with a LIMIT clause, the planner makes an appropriate interpolation between the endpoint costs to estimate which plan is really the cheapest.

The VERBOSE option emits the full internal representation of the plan tree, rather than just a summary (and sends it to the postmaster log file, too). Usually this option is only useful for debugging Postgres.

Notes

There is only sparse documentation on the optimizer's use of cost information in Postgres. General information on cost estimation for query optimization can be found in database textbooks. Refer to the *Programmer's Guide* in the chapters on indexes and the genetic query optimizer for more information.

Usage

To show a query plan for a simple query on a table with a single `int4` column and 128 rows:

```
EXPLAIN SELECT * FROM foo;
    NOTICE:  QUERY PLAN:

Seq Scan on foo  (cost=0.00..2.28 rows=128 width=4)

EXPLAIN
```

For the same table with an index to support an *equijoin* condition on the query, `EXPLAIN` will show a different plan:

```
EXPLAIN SELECT * FROM foo WHERE i = 4;
    NOTICE:  QUERY PLAN:

Index Scan using fi on foo  (cost=0.00..0.42 rows=1 width=4)

EXPLAIN
```

And finally, for the same table with an index to support an *equijoin* condition on the query, `EXPLAIN` will show the following for a query using an aggregate function:

```
EXPLAIN SELECT sum(i) FROM foo WHERE i = 4;
    NOTICE:  QUERY PLAN:

Aggregate  (cost=0.42..0.42 rows=1 width=4)
  ->  Index Scan using fi on foo  (cost=0.00..0.42 rows=1 width=4)
```

Note that the specific numbers shown, and even the selected query strategy, may vary between Postgres releases due to planner improvements.

Compatibility

SQL92

There is no `EXPLAIN` statement defined in SQL92.

Name

FETCH — Gets rows using a cursor

Synopsis

<refsynopsisdivinfo>1999-07-20</refsynopsisdivinfo>

```
FETCH [ direction ] [ count ] { IN | FROM } cursor
FETCH [ FORWARD | BACKWARD | RELATIVE ] [ # | ALL | NEXT | PRIOR ]
 { IN | FROM } cursor
```

Inputs

direction

selector defines the fetch direction. It can be one the following:

FORWARD

fetch next row(s). This is the default if *selector* is omitted.

BACKWARD

fetch previous row(s).

RELATIVE

Noise word for SQL92 compatibility.

count

count determines how many rows to fetch. It can be one of the following:

#

A signed integer that specify how many rows to fetch. Note that a negative integer is equivalent to changing the sense of FORWARD and BACKWARD.

ALL

Retrieve all remaining rows.

NEXT

Equivalent to specifying a count of `1`.

PRIOR

Equivalent to specifying a count of `-1`.

cursor

An open cursor's name.

Outputs

`FETCH` returns the results of the query defined by the specified cursor. The following messages will be returned if the query fails:

`NOTICE: PerformPortalFetch: portal "`*cursor*`" not found`

If *cursor* is not previously declared. The cursor must be declared within a transaction block.

```
NOTICE: FETCH/ABSOLUTE not supported, using RELATIVE
```

Postgres does not support absolute positioning of cursors.

```
ERROR: FETCH/RELATIVE at current position is not supported
```

SQL92 allows one to repetatively retrieve the cursor at its "current position" using the syntax

```
FETCH RELATIVE 0 FROM cursor
```

Postgres does not currently support this notion; in fact the value zero is reserved to indicate that all rows should be retrieved and is equivalent to specifying the ALL keyword. If the RELATIVE keyword has been used, the Postgres assumes that the user intended SQL92 behavior and returns this error message.

Description

FETCH allows a user to retrieve rows using a cursor. The number of rows retrieved is specified by #. If the number of rows remaining in the cursor is less than #, then only those available are fetched. Substituting the keyword ALL in place of a number will cause all remaining rows in the cursor to be retrieved. Instances may be fetched in both FORWARD and BACKWARD directions. The default direction is FORWARD.

Tip

Negative numbers are allowed to be specified for the row count. A negative number is equivalent to reversing the sense of the FORWARD and BACKWARD keywords. For example, FORWARD -1 is the same as BACKWARD 1.

Notes

Note that the FORWARD and BACKWARD keywords are Postgres extensions. The SQL92 syntax is also supported, specified in the second form of the command. See below for details on compatibility issues.

Updating data in a cursor is not supported by Postgres, because mapping cursor updates back to base tables is not generally possible, as is also the case with VIEW updates. Consequently, users must issue explicit UPDATE commands to replace data.

Cursors may only be used inside of transactions because the data that they store spans multiple user queries.

Use MOVE to change cursor position. DECLARE will define a cursor. Refer to BEGIN , COMMIT , and ROLLBACK for further information about transactions.

Usage

The following examples traverses a table using a cursor.

```
-- set up and use a cursor:

BEGIN WORK;
DECLARE liahona CURSOR FOR SELECT * FROM films;

-- Fetch first 5 rows in the cursor liahona:
FETCH FORWARD 5 IN liahona;
```

```
 code  |          title          | did | date_prod  |  kind    |
 len
-------+-------------------------+-----+------------+----------
+-------
 BL101 | The Third Man           | 101 | 1949-12-23 | Drama    |
 01:44
 BL102 | The African Queen       | 101 | 1951-08-11 | Romantic |
 01:43
 JL201 | Une Femme est une Femme | 102 | 1961-03-12 | Romantic |
 01:25
 P_301 | Vertigo                 | 103 | 1958-11-14 | Action   |
 02:08
 P_302 | Becket                  | 103 | 1964-02-03 | Drama    |
 02:28

-- Fetch previous row:
FETCH BACKWARD 1 IN liahona;

 code  | title   | did | date_prod  | kind   | len
-------+---------+-----+------------+--------+-------
 P_301 | Vertigo | 103 | 1958-11-14 | Action | 02:08

-- close the cursor and commit work:

CLOSE liahona;
COMMIT WORK;
```

Compatibility

SQL92

Note

The non-embedded use of cursors is a Postgres extension. The syntax and usage of cursors is being compared against the embedded form of cursors defined in SQL92.

SQL92 allows absolute positioning of the cursor for FETCH, and allows placing the results into explicit variables.

```
FETCH ABSOLUTE #
    FROM cursor
    INTO :variable [, ...]
```

ABSOLUTE

The cursor should be positioned to the specified absolute row number. All row numbers in Postgres are relative numbers so this capability is not supported.

:*variable*

Target host variable(s).

Name

GRANT — Grants access privilege to a user, a group or all users

Synopsis

<refsynopsisdivinfo>1999-07-20</refsynopsisdivinfo>

```
GRANT privilege [, ...] ON object [, ...]
    TO { PUBLIC | GROUP group | username }
```

Inputs

privilege

The possible privileges are:

SELECT

Access all of the columns of a specific table/view.

INSERT

Insert data into all columns of a specific table.

UPDATE

Update all columns of a specific table.

DELETE

Delete rows from a specific table.

RULE

Define rules on the table/view (See CREATE RULE statement).

ALL

Grant all privileges.

object

The name of an object to which to grant access. The possible objects are:

- table
- view
- sequence

PUBLIC

A short form representing all users.

GROUP *group*

A *group* to whom to grant privileges.

username

The name of a user to whom grant privileges. PUBLIC is a short form representing all users.

Outputs

`CHANGE`

Message returned if successful.

`ERROR: ChangeAcl: class "object" not found`

Message returned if the specified object is not available or if it is impossible to give privileges to the specified group or users.

Description

`GRANT` allows the creator of an object to give specific permissions to all users (PUBLIC) or to a certain user or group. Users other than the creator don't have any access permission unless the creator GRANTs permissions, after the object is created.

Once a user has a privilege on an object, he is enabled to exercise that privilege. There is no need to GRANT privileges to the creator of an object, the creator automatically holds ALL privileges, and can also drop the object.

Notes

Currently, to grant privileges in Postgres to only few columns, you must create a view having desired columns and then grant privileges to that view.

Use `psql \z` for further information about permissions on existing objects:

```
   Database    = lusitania
   +------------------
+-----------------------------------------+
   |  Relation        |        Grant/Revoke Permissions
 |
   +------------------
+-----------------------------------------+
   | mytable          | {"=rw","miriam=arwR","group todos=rw"}
 |
   +------------------
+-----------------------------------------+
   Legend:
         uname=arwR -- privileges granted to a user
   group gname=arwR -- privileges granted to a GROUP
              =arwR -- privileges granted to PUBLIC

                  r -- SELECT
                  w -- UPDATE/DELETE
                  a -- INSERT
                  R -- RULE
               arwR -- ALL
```

Refer to REVOKE statements to revoke access privileges.

Usage

Grant insert privilege to all users on table films:

```
GRANT INSERT ON films TO PUBLIC;
```

Grant all privileges to user manuel on view kinds:

```
GRANT ALL ON kinds TO manuel;
```

Compatibility

SQL92

The SQL92 syntax for GRANT allows setting privileges for individual columns within a table, and allows setting a privilege to grant the same privileges to others:

```
GRANT privilege [, ...]
    ON object [ ( column [, ...] ) ] [, ...]
    TO { PUBLIC | username [, ...] } [ WITH GRANT OPTION ]
```

Fields are compatible with the those in the Postgres implementation, with the following additions:

privilege

SQL92 permits additional privileges to be specified:

SELECT

REFERENCES

Allowed to reference some or all of the columns of a specific table/view in integrity constraints.

USAGE

Allowed to use a domain, character set, collation or translation. If an object specifies anything other than a table/view, *privilege* must specify only USAGE.

object

[TABLE] *table*

SQL92 allows the additional non-functional keyword `TABLE`.

CHARACTER SET

Allowed to use the specified character set.

COLLATION

Allowed to use the specified collation sequence.

TRANSLATION

Allowed to use the specified character set translation.

DOMAIN

Allowed to use the specified domain.

WITH GRANT OPTION

Allowed to grant the same privilege to others.

Name

INSERT — Inserts new rows into a table

Synopsis

<refsynopsisdivinfo>1999-07-20</refsynopsisdivinfo>

```
INSERT INTO table [ ( column [, ...] ) ]
    { VALUES ( expression [, ...] ) | SELECT query }
```

Inputs

table

The name of an existing table.

column

The name of a column in *table*.

expression

A valid expression or value to assign to *column*.

query

A valid query. Refer to the SELECT statement for a further description of valid arguments.

Outputs

INSERT *oid* 1

Message returned if only one row was inserted. *oid* is the numeric OID of the inserted row.

INSERT 0 #

Message returned if more than one rows were inserted. # is the number of rows inserted.

Description

INSERT allows one to insert new rows into a class or table. One can insert a single row at time or several rows as a result of a query. The columns in the target list may be listed in any order.

Each column not present in the target list will be inserted using a default value, either a declared DEFAULT value or NULL. Postgres will reject the new column if a NULL is inserted into a column declared NOT NULL.

If the expression for each column is not of the correct data type, automatic type coercion will be attempted.

You must have insert privilege to a table in order to append to it, as well as select privilege on any table specified in a WHERE clause.

Usage

Insert a single row into table films:

```
INSERT INTO films VALUES
```

```
    ('UA502','Bananas',105,'1971-07-13','Comedy',INTERVAL '82
 minute');
```

In this second example the column `date_prod` is omitted and therefore it will have the default value of NULL:

```
INSERT INTO films (code, title, did, date_prod, kind)
    VALUES ('T_601', 'Yojimbo', 106, DATE '1961-06-16', 'Drama');
```

Insert a single row into table distributors; note that only column `name` is specified, so the omitted column `did` will be assigned its default value:

```
INSERT INTO distributors (name) VALUES ('British Lion');
```

Insert several rows into table films from table `tmp`:

```
INSERT INTO films SELECT * FROM tmp;
```

Insert into arrays (refer to the *PostgreSQL User's Guide* for further information about arrays):

```
-- Create an empty 3x3 gameboard for noughts-and-crosses
-- (all of these queries create the same board attribute)
INSERT INTO tictactoe (game, board[1:3][1:3])
    VALUES (1,'{{"","",""},{},{"",""}}');
INSERT INTO tictactoe (game, board[3][3])
    VALUES (2,'{}');
INSERT INTO tictactoe (game, board)
    VALUES (3,'{{,,},{,,},{,,}}');
```

Compatibility

SQL92

`INSERT` is fully compatible with SQL92. Possible limitations in features of the *`query`* clause are documented for SELECT .

Name

LISTEN — Listen for a response on a notify condition

Synopsis

<refsynopsisdivinfo>1999-07-20</refsynopsisdivinfo>

```
LISTEN name
```

Inputs

name

Name of notify condition.

Outputs

`LISTEN`

Message returned upon successful completion of registration.

`NOTICE Async_Listen: We are already listening on name`

If this backend is already registered for that notify condition.

Description

`LISTEN` registers the current Postgres backend as a listener on the notify condition *name*.

Whenever the command `NOTIFY` *name* is invoked, either by this backend or another one connected to the same database, all the backends currently listening on that notify condition are notified, and each will in turn notify its connected frontend application. See the discussion of `NOTIFY` for more information.

A backend can be unregistered for a given notify condition with the `UNLISTEN` command. Also, a backend's listen registrations are automatically cleared when the backend process exits.

The method a frontend application must use to detect notify events depends on which Postgres application programming interface it uses. With the basic libpq library, the application issues `LISTEN` as an ordinary SQL command, and then must periodically call the routine `PQnotifies` to find out whether any notify events have been received. Other interfaces such as libpgtcl provide higher-level methods for handling notify events; indeed, with libpgtcl the application programmer should not even issue `LISTEN` or `UNLISTEN` directly. See the documentation for the library you are using for more details.

NOTIFY contains a more extensive discussion of the use of `LISTEN` and `NOTIFY`.

Notes

name can be any string valid as a name; it need not correspond to the name of any actual table. If *notifyname* is enclosed in double-quotes, it need not even be a syntactically valid name, but can be any string up to 31 characters long.

In some previous releases of Postgres, *name* had to be enclosed in double-quotes when it did not correspond to any existing table name, even if syntactically valid as a name. That is no longer required.

Usage

Configure and execute a listen/notify sequence from psql:

```
LISTEN virtual;
NOTIFY virtual;

Asynchronous NOTIFY 'virtual' from backend with pid '8448'
 received.
```

Compatibility

SQL92

There is no `LISTEN` in SQL92.

Name

LOAD — Dynamically loads an object file

Synopsis

<refsynopsisdivinfo>1999-07-20</refsynopsisdivinfo>

```
LOAD 'filename'
```

Inputs

filename

Object file for dynamic loading.

Outputs

`LOAD`

Message returned on successful completion.

`ERROR: LOAD: could not open file 'filename'`

Message returned if the specified file is not found. The file must be visible *to the Postgres backend*, with the appropriate full path name specified, to avoid this message.

Description

Loads an object (or ".o") file into the Postgres backend address space. Once a file is loaded, all functions in that file can be accessed. This function is used in support of user-defined types and functions.

If a file is not loaded using `LOAD`, the file will be loaded automatically the first time the function is called by Postgres. `LOAD` can also be used to reload an object file if it has been edited and recompiled. Only objects created from C language files are supported at this time.

Notes

Functions in loaded object files should not call functions in other object files loaded through the `LOAD` command. For example, all functions in file `A` should call each other, functions in the standard or math libraries, or in Postgres itself. They should not call functions defined in a different loaded file `B`. This is because if `B` is reloaded, the Postgres loader is not able to relocate the calls from the functions in `A` into the new address space of `B`. If `B` is not reloaded, however, there will not be a problem.

Object files must be compiled to contain position independent code. For example, on DECstations you must use /bin/cc with the `-G 0` option when compiling object files to be loaded.

Note that if you are porting Postgres to a new platform, `LOAD` will have to work in order to support ADTs.

Usage

Load the file `/usr/postgres/demo/circle.o`:

```
LOAD '/usr/postgres/demo/circle.o'
```

Compatibility

SQL92

There is no `LOAD` in SQL92.

Name

LOCK — Explicitly lock a table inside a transaction

Synopsis

<refsynopsisdivinfo>1999-07-20</refsynopsisdivinfo>

```
LOCK [ TABLE ] name
LOCK [ TABLE ] name IN [ ROW | ACCESS ] { SHARE | EXCLUSIVE } MODE
LOCK [ TABLE ] name IN SHARE ROW EXCLUSIVE MODE
```

Inputs

name

The name of an existing table to lock.

ACCESS SHARE MODE

Note

This lock mode is acquired automatically over tables being queried.

This is the least restrictive lock mode. It conflicts only with ACCESS EXCLUSIVE mode. It is used to protect a table from being modified by concurrent `ALTER TABLE`, `DROP TABLE` and `VACUUM` commands.

ROW SHARE MODE

Note

Automatically acquired by `SELECT...FOR UPDATE`. While it is a shared lock, may be upgrade later to a ROW EXCLUSIVE lock.

Conflicts with EXCLUSIVE and ACCESS EXCLUSIVE lock modes.

ROW EXCLUSIVE MODE

Note

Automatically acquired by `UPDATE`, `DELETE`, and `INSERT` statements.

Conflicts with SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE and ACCESS EXCLUSIVE modes.

SHARE MODE

Note

Automatically acquired by `CREATE INDEX`. Share-locks the entire table.

Conflicts with ROW EXCLUSIVE, SHARE ROW EXCLUSIVE, EXCLUSIVE and ACCESS EXCLUSIVE modes. This mode protects a table against concurrent updates.

SHARE ROW EXCLUSIVE MODE

Note

This is like EXCLUSIVE MODE, but allows SHARE ROW locks by others.

Conflicts with ROW EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE and ACCESS EXCLUSIVE modes.

EXCLUSIVE MODE

Note

This mode is yet more restrictive than SHARE ROW EXCLUSIVE. It blocks all concurrent ROW SHARE/SELECT...FOR UPDATE queries.

Conflicts with ROW SHARE, ROW EXCLUSIVE, SHARE, SHARE ROW EXCLUSIVE, EXCLUSIVE and ACCESS EXCLUSIVE modes.

ACCESS EXCLUSIVE MODE

Note

Automatically acquired by `ALTER TABLE`, `DROP TABLE`, `VACUUM` statements. This is the most restrictive lock mode which conflicts with all other lock modes and protects a locked table from any concurrent operations.

Note

This lock mode is also acquired by an unqualified `LOCK TABLE` (i.e. the command without an explicit lock mode option).

Outputs

`LOCK TABLE`

The lock was successfully applied.

`ERROR` *`name`*`: Table does not exist.`

Message returned if *`name`* does not exist.

Description

`LOCK TABLE` controls concurrent access to a table for the duration of a transaction. Postgres always uses the least restrictive lock mode whenever possible. `LOCK TABLE` provided for cases when you might need more restrictive locking.

RDBMS locking uses the following terminology:

EXCLUSIVE

Exclusive lock that prevents other locks from being granted.

SHARE

Allows others to share lock. Prevents EXCLUSIVE locks.

ACCESS

Locks table schema.

ROW

Locks individual rows.

Note

If EXCLUSIVE or SHARE are not specified, EXCLUSIVE is assumed. Locks exist for the duration of the transaction.

For example, an application runs a transaction at READ COMMITTED isolation level and needs to ensure the existance of data in a table for the duration of the transaction. To achieve this you could use SHARE lock mode over the table before querying. This will protect data from concurrent changes and provide any further read operations over the table with data in their actual current state, because SHARE lock mode conflicts with any ROW EXCLUSIVE one acquired by writers, and your `LOCK TABLE name IN SHARE MODE` statement will wait until any concurrent write operations commit or rollback.

Note

To read data in their real current state when running a transaction at the SERIALIZABLE isolation level you have to execute a LOCK TABLE statement before execution any DML statement, when the transaction defines what concurrent changes will be visible to itself.

In addition to the requirements above, if a transaction is going to change data in a table then SHARE ROW EXCLUSIVE lock mode should be acquired to prevent deadlock conditions when two concurrent transactions attempt to lock the table in SHARE mode and then try to change data in this table, both (implicitly) acquiring ROW EXCLUSIVE lock mode that conflicts with concurrent SHARE lock.

To continue with the deadlock (when two transaction wait one another) issue raised above, you should follow two general rules to prevent deadlock conditions:

- Transactions have to acquire locks on the same objects in the same order.

  For example, if one application updates row R1 and than updates row R2 (in the same transaction) then the second application shouldn't update row R2 if it's going to update row R1 later (in a single transaction). Instead, it should update rows R1 and R2 in the same order as the first application.

- Transactions should acquire two conflicting lock modes only if one of them is self-conflicting (i.e. may be held by one transaction at time only). If multiple lock modes are involved, then transactions should always acquire the most restrictive mode first.

  An example for this rule was given previously when discussing the use of SHARE ROW EXCLUSIVE mode rather than SHARE mode.

Note

Postgres does detect deadlocks and will rollback at least one waiting transaction to resolve the deadlock.

Notes

`LOCK` is a Postgres language extension.

Except for ACCESS SHARE/EXCLUSIVE lock modes, all other Postgres lock modes and the `LOCK TABLE` syntax are compatible with those present in Oracle.

`LOCK` works only inside transactions.

Usage

Illustrate a SHARE lock on a primary key table when going to perform inserts into a foreign key table:

```
BEGIN WORK;
LOCK TABLE films IN SHARE MODE;
SELECT id FROM films
    WHERE name = 'Star Wars: Episode I - The Phantom Menace';
-- Do ROLLBACK if record was not returned
INSERT INTO films_user_comments VALUES
    (_id_, 'GREAT! I was waiting for it for so long!');
COMMIT WORK;
```

Take a SHARE ROW EXCLUSIVE lock on a primary key table when going to perform a delete operation:

```
BEGIN WORK;
LOCK TABLE films IN SHARE ROW EXCLUSIVE MODE;
DELETE FROM films_user_comments WHERE id IN
    (SELECT id FROM films WHERE rating < 5);
DELETE FROM films WHERE rating < 5;
COMMIT WORK;
```

Compatibility

SQL92

There is no `LOCK TABLE` in SQL92, which instead uses `SET TRANSACTION` to specify concurrency levels on transactions. We support that too; see SET for details.

Name

MOVE — Moves cursor position

Synopsis

<refsynopsisdivinfo>1999-07-20</refsynopsisdivinfo>

```
MOVE [ direction ] [ count ]
    { IN | FROM } cursor
```

Description

`MOVE` allows a user to move cursor position a specified number of rows. `MOVE` works like the `FETCH` command, but only positions the cursor and does not return rows.

Refer to FETCH for details on syntax and usage.

Notes

`MOVE` is a Postgres language extension.

Refer to FETCH for a description of valid arguments. Refer to DECLARE to define a cursor. Refer to BEGIN , COMMIT , and ROLLBACK for further information about transactions.

Usage

Set up and use a cursor:

```
BEGIN WORK;
DECLARE liahona CURSOR  FOR SELECT * FROM films;
-- Skip first 5 rows:
MOVE FORWARD 5 IN liahona;
MOVE
-- Fetch 6th row in the cursor liahona:
FETCH 1 IN liahona;
FETCH

 code  | title  | did | date_prod | kind   | len
-------+--------+-----+-----------+--------+-------
 P_303 | 48 Hrs | 103 | 1982-10-22| Action | 01:37
(1 row)
-- close the cursor liahona and commit work:
CLOSE liahona;
COMMIT WORK;
```

Compatibility

SQL92

There is no SQL92 `MOVE` statement. Instead, SQL92 allows one to `FETCH` rows from an absolute cursor position, implicitly moving the cursor to the correct position.

Name

NOTIFY — Signals all frontends and backends listening on a notify condition

Synopsis

<refsynopsisdivinfo>1999-07-20</refsynopsisdivinfo>

```
NOTIFY name
```

Inputs

notifyname

Notify condition to be signaled.

Outputs

NOTIFY

Acknowledgement that notify command has executed.

Notify events

Events are delivered to listening frontends; whether and how each frontend application reacts depends on its programming.

Description

The NOTIFY command sends a notify event to each frontend application that has previously executed LISTEN *notifyname* for the specified notify condition in the current database.

The information passed to the frontend for a notify event includes the notify condition name and the notifying backend process's PID. It is up to the database designer to define the condition names that will be used in a given database and what each one means.

Commonly, the notify condition name is the same as the name of some table in the database, and the notify event essentially means "I changed this table, take a look at it to see what's new". But no such association is enforced by the NOTIFY and LISTEN commands. For example, a database designer could use several different condition names to signal different sorts of changes to a single table.

NOTIFY provides a simple form of signal or IPC (interprocess communication) mechanism for a collection of processes accessing the same Postgres database. Higher-level mechanisms can be built by using tables in the database to pass additional data (beyond a mere condition name) from notifier to listener(s).

When NOTIFY is used to signal the occurrence of changes to a particular table, a useful programming technique is to put the NOTIFY in a rule that is triggered by table updates. In this way, notification happens automatically when the table is changed, and the application programmer can't accidentally forget to do it.

NOTIFY interacts with SQL transactions in some important ways. Firstly, if a NOTIFY is executed inside a transaction, the notify events are not delivered until and unless the transaction is committed. This is appropriate, since if the transaction is aborted we would like all the commands within it to have had no effect, including NOTIFY. But it can be disconcerting if one is expecting the notify events to be delivered immediately. Secondly, if a listening backend receives a notify signal while it is within a transaction, the notify event will not be delivered to its connected frontend until just after the transaction is completed (either committed or aborted). Again, the reasoning is that if a notify were delivered within a transaction that was later aborted, one would want the notification to be undone

somehow --- but the backend cannot "take back" a notify once it has sent it to the frontend. So notify events are only delivered between transactions. The upshot of this is that applications using NOTIFY for real-time signaling should try to keep their transactions short.

NOTIFY behaves like Unix signals in one important respect: if the same condition name is signaled multiple times in quick succession, recipients may get only one notify event for several executions of NOTIFY. So it is a bad idea to depend on the number of notifies received. Instead, use NOTIFY to wake up applications that need to pay attention to something, and use a database object (such as a sequence) to keep track of what happened or how many times it happened.

It is common for a frontend that sends NOTIFY to be listening on the same notify name itself. In that case it will get back a notify event, just like all the other listening frontends. Depending on the application logic, this could result in useless work --- for example, re-reading a database table to find the same updates that that frontend just wrote out. In Postgres 6.4 and later, it is possible to avoid such extra work by noticing whether the notifying backend process's PID (supplied in the notify event message) is the same as one's own backend's PID (available from libpq). When they are the same, the notify event is one's own work bouncing back, and can be ignored. (Despite what was said in the preceding paragraph, this is a safe technique. Postgres keeps self-notifies separate from notifies arriving from other backends, so you cannot miss an outside notify by ignoring your own notifies.)

Notes

name can be any string valid as a name; it need not correspond to the name of any actual table. If *name* is enclosed in double-quotes, it need not even be a syntactically valid name, but can be any string up to 31 characters long.

In some previous releases of Postgres, *name* had to be enclosed in double-quotes when it did not correspond to any existing table name, even if syntactically valid as a name. That is no longer required.

In Postgres releases prior to 6.4, the backend PID delivered in a notify message was always the PID of the frontend's own backend. So it was not possible to distinguish one's own notifies from other clients' notifies in those earlier releases.

Usage

Configure and execute a listen/notify sequence from psql:

```
=> LISTEN virtual;
=> NOTIFY virtual;
Asynchronous NOTIFY 'virtual' from backend with pid '8448'
 received.
```

Compatibility

SQL92

There is no NOTIFY statement in SQL92.

Name

REINDEX — Recover corrupted system indexes under standalone Postgres

Synopsis

<refsynopsisdivinfo>2000-03-30</refsynopsisdivinfo>

```
REINDEX { TABLE | DATABASE | INDEX } name [ FORCE ]
```

Inputs

TABLE

Recreate all indexes of a specfied table.

DATABASE

Recreate all system indexes of a specfied database.

INDEX

Recreate a specfied index.

name

The name of the specific table/database/index to be be reindexed.

FORCE

Recreate indexes forcedly. Without this keyword REINDEX does nothing unless target indexes are invalidated.

Outputs

`REINDEX`

Message returned if the table is successfully reindexed.

Description

`REINDEX` is used to recover corrupted system indexes. In order to run REINDEX command,Postmaster must be shutdown and standalone Postgres should be started instead with options -O and -P(an option to ignore system indexes). Note that we couldn't rely on system indexes for the recovery of system indexes.

Usage

Recreate the table `mytable`:

```
REINDEX TABLE mytable;
```

Some more examples:

```
REINDEX DATABASE my_database FORCE;
REINDEX INDEX my_index;
```

Compatibility

SQL92

There is no `REINDEX` in SQL92.

Name

RESET — Restores run-time parameters for session to default values

Synopsis

<refsynopsisdivinfo>1999-07-20</refsynopsisdivinfo>

```
RESET variable
```

Inputs

variable

Refer to SET for more information on available variables.

Outputs

```
RESET VARIABLE
```

Message returned if *variable* is successfully reset to its default value.

Description

RESET restores variables to their default values. Refer to SET for details on allowed values and defaults. RESET is an alternate form for

```
SET variable = DEFAULT
```

Notes

See also SET and SHOW to manipulate variable values.

Usage

Set DateStyle to its default value:

```
RESET DateStyle;
```

Set Geqo to its default value:

```
RESET GEQO;
```

Compatibility

SQL92

There is no RESET in SQL92.

Name

REVOKE — Revokes access privilege from a user, a group or all users.

Synopsis

<refsynopsisdivinfo>1999-07-20</refsynopsisdivinfo>

```
REVOKE privilege [, ...]
    ON object [, ...]
    FROM { PUBLIC | GROUP groupname | username }
```

Inputs

privilege

The possible privileges are:

SELECT

Privilege to access all of the columns of a specific table/view.

INSERT

Privilege to insert data into all columns of a specific table.

UPDATE

Privilege to update all columns of a specific table.

DELETE

Privilege to delete rows from a specific table.

RULE

Privilege to define rules on table/view. (See CREATE RULE).

ALL

Rescind all privileges.

object

The name of an object from which to revoke access. The possible objects are:

- table
- view
- sequence

group

The name of a group from whom to revoke privileges.

username

The name of a user from whom revoke privileges. Use the PUBLIC keyword to specify all users.

PUBLIC

Rescind the specified privilege(s) for all users.

Outputs

`CHANGE`

Message returned if successfully.

`ERROR`

Message returned if object is not available or impossible to revoke privileges from a group or users.

Description

`REVOKE` allows creator of an object to revoke permissions granted before, from all users (via PUBLIC) or a certain user or group.

Notes

Refer to psql \z command for further information about permissions on existing objects:

```
Database    = lusitania
+------------------+---------------------------------------------+
|  Relation        |        Grant/Revoke Permissions             |
+------------------+---------------------------------------------+
| mytable          | {"=rw","miriam=arwR","group todos=rw"}      |
+------------------+---------------------------------------------+
Legend:
     uname=arwR -- privileges granted to a user
     group gname=arwR -- privileges granted to a GROUP
     =arwR -- privileges granted to PUBLIC

     r -- SELECT
     w -- UPDATE/DELETE
     a -- INSERT
     R -- RULE
     arwR -- ALL
```

Tip

Currently, to create a GROUP you have to insert data manually into table pg_group as:

```
INSERT INTO pg_group VALUES ('todos');
CREATE USER miriam IN GROUP todos;
```

Usage

Revoke insert privilege from all users on table `films`:

```
REVOKE INSERT ON films FROM PUBLIC;
```

Revoke all privileges from user `manuel` on view `kinds`:

```
REVOKE ALL ON kinds FROM manuel;
```

Compatibility

SQL92

The SQL92 syntax for REVOKE has additional capabilities for rescinding privileges, including those on individual columns in tables:

```
REVOKE { SELECT | DELETE | USAGE | ALL PRIVILEGES } [, ...]
    ON object
    FROM { PUBLIC | username [, ...] } { RESTRICT |
 CASCADE }
REVOKE { INSERT | UPDATE | REFERENCES } [, ...] [ ( column
 [, ...] ) ]
    ON object
    FROM { PUBLIC | username [, ...] } { RESTRICT |
 CASCADE }
```

Refer to GRANT for details on individual fields.

```
REVOKE GRANT OPTION FOR privilege [, ...]
    ON object
    FROM { PUBLIC | username [, ...] } { RESTRICT |
 CASCADE }
```

Rescinds authority for a user to grant the specified privilege to others. Refer to GRANT for details on individual fields.

The possible objects are:

[TABLE] table/view
CHARACTER SET character-set
COLLATION collation
TRANSLATION translation
DOMAIN domain

If user1 gives a privilege WITH GRANT OPTION to user2, and user2 gives it to user3 then user1 can revoke this privilege in cascade using the CASCADE keyword.

If user1 gives a privilege WITH GRANT OPTION to user2, and user2 gives it to user3 then if user1 try revoke this privilege it fails if he/she specify the RESTRICT keyword.

Name

ROLLBACK — Aborts the current transaction

Synopsis

<refsynopsisdivinfo>1999-07-20</refsynopsisdivinfo>

```
ROLLBACK [ WORK | TRANSACTION ]
```

Inputs

None.

Outputs

`ABORT`

Message returned if successful.

`NOTICE: ROLLBACK: no transaction in progress`

If there is not any transaction currently in progress.

Description

`ROLLBACK` rolls back the current transaction and causes all the updates made by the transaction to be discarded.

Notes

Use COMMIT to successfully terminate a transaction. ABORT is a synonym for `ROLLBACK`.

Usage

To abort all changes:

```
ROLLBACK WORK;
```

Compatibility

SQL92

SQL92 only specifies the two forms `ROLLBACK` and `ROLLBACK WORK`. Otherwise full compatibility.

Name

SELECT — Retrieve rows from a table or view.

Synopsis

<refsynopsisdivinfo>1999-07-20</refsynopsisdivinfo>

```
SELECT [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]
    expression [ AS name ] [, ...]
    [ INTO [ TEMPORARY | TEMP ] [ TABLE ] new_table ]
    [ FROM table [ alias ] [, ...] ]
    [ WHERE condition ]
    [ GROUP BY column [, ...] ]
    [ HAVING condition [, ...] ]
    [ { UNION [ ALL ] | INTERSECT | EXCEPT } select ]
    [ ORDER BY column [ ASC | DESC | USING operator ] [, ...] ]
    [ FOR UPDATE [ OF class_name [, ...] ] ]
    LIMIT { count | ALL } [ { OFFSET | , } start ]
```

Inputs

expression

The name of a table's column or an expression.

name

Specifies another name for a column or an expression using the AS clause. This name is primarily used to label the column for display. It can also be used to refer to the column's value in ORDER BY and GROUP BY clauses. But the *name* cannot be used in the WHERE or HAVING clauses; write out the expression instead.

TEMPORARY
TEMP

If TEMPORARY or TEMP is specified, the table is created unique to this session, and is automatically dropped on session exit.

new_table

If the INTO TABLE clause is specified, the result of the query will be stored in a new table with the indicated name. The target table (*new_table*) will be created automatically and must not exist before this command. Refer to `SELECT INTO` for more information.

Note

The `CREATE TABLE AS` statement will also create a new table from a select query.

table

The name of an existing table referenced by the FROM clause.

alias

An alternate name for the preceding *table*. It is used for brevity or to eliminate ambiguity for joins within a single table.

condition

A boolean expression giving a result of true or false. See the WHERE clause.

column

The name of a table's column.

select

A select statement with all features except the ORDER BY and LIMIT clauses.

Outputs

Rows

The complete set of rows resulting from the query specification.

count

The count of rows returned by the query.

Description

SELECT will return rows from one or more tables. Candidates for selection are rows which satisfy the WHERE condition; if WHERE is omitted, all rows are candidates. (See WHERE Clause .)

DISTINCT will eliminate duplicate rows from the result. ALL (the default) will return all candidate rows, including duplicates.

DISTINCT ON eliminates rows that match on all the specified expressions, keeping only the first row of each set of duplicates. The DISTINCT ON expressions are interpreted using the same rules as for ORDER BY items; see below. Note that "the first row" of each set is unpredictable unless ORDER BY is used to ensure that the desired row appears first. For example,

```
SELECT DISTINCT ON (location) location, time, report
FROM weatherReports
ORDER BY location, time DESC;
```

retrieves the most recent weather report for each location. But if we had not used ORDER BY to force descending order of time values for each location, we'd have gotten a report of unpredictable age for each location.

The GROUP BY clause allows a user to divide a table into groups of rows that match on one or more values. (See GROUP BY Clause .)

The HAVING clause allows selection of only those groups of rows meeting the specified condition. (See HAVING Clause .)

The ORDER BY clause causes the returned rows to be sorted in a specified order. If ORDER BY is not given, the rows are returned in whatever order the system finds cheapest to produce. (See ORDER BY Clause .)

The UNION operator allows the result to be the collection of rows returned by the queries involved. (See UNION Clause .)

The INTERSECT operator gives you the rows that are common to both queries. (See INTERSECT Clause .)

The EXCEPT operator gives you the rows returned by the first query but not the second query. (See EXCEPT Clause .)

The FOR UPDATE clause allows the SELECT statement to perform exclusive locking of selected rows.

The LIMIT clause allows a subset of the rows produced by the query to be returned to the user. (See LIMIT Clause .)

You must have SELECT privilege to a table to read its values (See the GRANT/REVOKE statements).

WHERE Clause

The optional WHERE condition has the general form:

```
WHERE boolean_expr
```

boolean_expr can consist of any expression which evaluates to a boolean value. In many cases, this expression will be

```
expr cond_op expr
```

or

```
log_op expr
```

where *cond_op* can be one of: =, <, <=, >, >= or <>, a conditional operator like ALL, ANY, IN, LIKE, or a locally-defined operator, and *log_op* can be one of: AND, OR, NOT. SELECT will ignore all rows for which the WHERE condition does not return TRUE.

GROUP BY Clause

GROUP BY specifies a grouped table derived by the application of this clause:

```
GROUP BY column [, ...]
```

GROUP BY will condense into a single row all selected rows that share the same values for the grouped columns. Aggregate functions, if any, are computed across all rows making up each group, producing a separate value for each group (whereas without GROUP BY, an aggregate produces a single value computed across all the selected rows). When GROUP BY is present, it is not valid for the SELECT output expression(s) to refer to ungrouped columns except within aggregate functions, since there would be more than one possible value to return for an ungrouped column.

An item in GROUP BY can also be the name or ordinal number of an output column (SELECT expression), or it can be an arbitrary expression formed from input-column values. In case of ambiguity, a GROUP BY name will be interpreted as an input-column name rather than an output column name.

HAVING Clause

The optional HAVING condition has the general form:

```
HAVING cond_expr
```

where *cond_expr* is the same as specified for the WHERE clause.

HAVING specifies a grouped table derived by the elimination of group rows that do not satisfy the *cond_expr*. HAVING is different from WHERE: WHERE filters individual rows before application of GROUP BY, while HAVING filters group rows created by GROUP BY.

Each column referenced in *cond_expr* shall unambiguously reference a grouping column, unless the reference appears within an aggregate function.

ORDER BY Clause

```
ORDER BY column [ ASC | DESC ] [, ...]
```

column can be either a result column name or an ordinal number.

The ordinal numbers refers to the ordinal (left-to-right) position of the result column. This feature makes it possible to define an ordering on the basis of a column that does not have a proper name. This is never absolutely necessary because it is always possible to assign a name to a result column using the AS clause, e.g.:

```
SELECT title, date_prod + 1 AS newlen FROM films ORDER BY newlen;
```

It is also possible to ORDER BY arbitrary expressions (an extension to SQL92), including fields that do not appear in the SELECT result list. Thus the following statement is legal:

```
SELECT name FROM distributors ORDER BY code;
```

Note that if an ORDER BY item is a simple name that matches both a result column name and an input column name, ORDER BY will interpret it as the result column name. This is the opposite of the choice that GROUP BY will make in the same situation. This inconsistency is mandated by the SQL92 standard.

Optionally one may add the keyword DESC (descending) or ASC (ascending) after each column name in the ORDER BY clause. If not specified, ASC is assumed by default. Alternatively, a specific ordering operator name may be specified. ASC is equivalent to USING '<' and DESC is equivalent to USING '>'.

UNION Clause

```
table_query UNION [ ALL ] table_query
    [ ORDER BY column [ ASC | DESC ] [, ...] ]
```

where *table_query* specifies any select expression without an ORDER BY or LIMIT clause.

The UNION operator allows the result to be the collection of rows returned by the queries involved. The two SELECTs that represent the direct operands of the UNION must produce the same number of columns, and corresponding columns must be of compatible data types.

By default, the result of UNION does not contain any duplicate rows unless the ALL clause is specified.

Multiple UNION operators in the same SELECT statement are evaluated left to right. Note that the ALL keyword is not global in nature, being applied only for the current pair of table results.

INTERSECT Clause

```
table_query INTERSECT table_query
    [ ORDER BY column [ ASC | DESC ] [, ...] ]
```

where *table_query* specifies any select expression without an ORDER BY or LIMIT clause.

The INTERSECT operator gives you the rows that are common to both queries. The two SELECTs that represent the direct operands of the INTERSECT must produce the same number of columns, and corresponding columns must be of compatible data types.

Multiple INTERSECT operators in the same SELECT statement are evaluated left to right, unless parentheses dictate otherwise.

EXCEPT Clause

```
table_query EXCEPT table_query
     [ ORDER BY column [ ASC | DESC ] [, ...] ]
```

where *table_query* specifies any select expression without an ORDER BY or LIMIT clause.

The EXCEPT operator gives you the rows returned by the first query but not the second query. The two SELECTs that represent the direct operands of the EXCEPT must produce the same number of columns, and corresponding columns must be of compatible data types.

Multiple EXCEPT operators in the same SELECT statement are evaluated left to right, unless parentheses dictate otherwise.

LIMIT Clause

```
LIMIT { count | ALL } [ { OFFSET | , } start ]
OFFSET start
```

where *count* specifies the maximum number of rows to return, and *start* specifies the number of rows to skip before starting to return rows.

LIMIT allows you to retrieve just a portion of the rows that are generated by the rest of the query. If a limit count is given, no more than that many rows will be returned. If an offset is given, that many rows will be skipped before starting to return rows.

When using LIMIT, it is a good idea to use an ORDER BY clause that constrains the result rows into a unique order. Otherwise you will get an unpredictable subset of the query's rows --- you may be asking for the tenth through twentieth rows, but tenth through twentieth in what ordering? You don't know what ordering, unless you specified ORDER BY.

As of Postgres 7.0, the query optimizer takes LIMIT into account when generating a query plan, so you are very likely to get different plans (yielding different row orders) depending on what you give for LIMIT and OFFSET. Thus, using different LIMIT/OFFSET values to select different subsets of a query result *will give inconsistent results* unless you enforce a predictable result ordering with ORDER BY. This is not a bug; it is an inherent consequence of the fact that SQL does not promise to deliver the results of a query in any particular order unless ORDER BY is used to constrain the order.

Usage

To join the table `films` with the table `distributors`:

```
SELECT f.title, f.did, d.name, f.date_prod, f.kind
    FROM distributors d, films f
    WHERE f.did = d.did

           title           | did |   name           | date_prod  |
 kind
---------------------------+-----+------------------+------------
+----------
 The Third Man             | 101 | British Lion     | 1949-12-23 |
 Drama
 The African Queen         | 101 | British Lion     | 1951-08-11 |
 Romantic
 Une Femme est une Femme   | 102 | Jean Luc Godard  | 1961-03-12 |
 Romantic
```

```
 Vertigo                   | 103 | Paramount        | 1958-11-14 |
 Action
 Becket                    | 103 | Paramount        | 1964-02-03 |
 Drama
 48 Hrs                    | 103 | Paramount        | 1982-10-22 |
 Action
 War and Peace             | 104 | Mosfilm          | 1967-02-12 |
 Drama
 West Side Story           | 105 | United Artists   | 1961-01-03 |
 Musical
 Bananas                   | 105 | United Artists   | 1971-07-13 |
 Comedy
 Yojimbo                   | 106 | Toho             | 1961-06-16 |
 Drama
 There's a Girl in my Soup | 107 | Columbia         | 1970-06-11 |
 Comedy
 Taxi Driver               | 107 | Columbia         | 1975-05-15 |
 Action
 Absence of Malice         | 107 | Columbia         | 1981-11-15 |
 Action
 Storia di una donna       | 108 | Westward         | 1970-08-15 |
 Romantic
 The King and I            | 109 | 20th Century Fox | 1956-08-11 |
 Musical
 Das Boot                  | 110 | Bavaria Atelier  | 1981-11-11 |
 Drama
 Bed Knobs and Broomsticks | 111 | Walt Disney      |            |
 Musical
(17 rows)
```

To sum the column `len` of all films and group the results by `kind`:

```
SELECT kind, SUM(len) AS total FROM films GROUP BY kind;
```

```
   kind   | total
----------+-------
 Action   | 07:34
 Comedy   | 02:58
 Drama    | 14:28
 Musical  | 06:42
 Romantic | 04:38
(5 rows)
```

To sum the column `len` of all films, group the results by `kind` and show those group totals that are less than 5 hours:

```
SELECT kind, SUM(len) AS total
    FROM films
    GROUP BY kind
    HAVING SUM(len) < INTERVAL '5 hour';
```

```
 kind     | total
----------+-------
 Comedy   | 02:58
 Romantic | 04:38
(2 rows)
```

The following two examples are identical ways of sorting the individual results according to the contents of the second column (`name`):

```
SELECT * FROM distributors ORDER BY name;
SELECT * FROM distributors ORDER BY 2;

 did |       name
-----+-----------------
 109 | 20th Century Fox
 110 | Bavaria Atelier
 101 | British Lion
 107 | Columbia
 102 | Jean Luc Godard
 113 | Luso films
 104 | Mosfilm
 103 | Paramount
 106 | Toho
 105 | United Artists
 111 | Walt Disney
 112 | Warner Bros.
 108 | Westward
(13 rows)
```

This example shows how to obtain the union of the tables `distributors` and `actors`, restricting the results to those that begin with letter W in each table. Only distinct rows are wanted, so the ALL keyword is omitted:

```
distributors:               actors:
 did |     name              id |     name
-----+--------------       ----+----------------
 108 | Westward              1 | Woody Allen
 111 | Walt Disney           2 | Warren Beatty
 112 | Warner Bros.          3 | Walter Matthau
 ...                         ...

SELECT distributors.name
    FROM   distributors
    WHERE  distributors.name LIKE 'W%'
UNION
SELECT actors.name
    FROM   actors
    WHERE  actors.name LIKE 'W%'

      name
----------------
 Walt Disney
 Walter Matthau
 Warner Bros.
 Warren Beatty
 Westward
 Woody Allen
```

Compatibility

Extensions

Postgres allows one to omit the `FROM` clause from a query. This feature was retained from the original PostQuel query language:

```
SELECT distributors.* WHERE name = 'Westwood';
```

```
 did | name
-----+----------
 108 | Westward
```

SQL92

SELECT Clause

In the SQL92 standard, the optional keyword "AS" is just noise and can be omitted without affecting the meaning. The Postgres parser requires this keyword when renaming columns because the type extensibility features lead to parsing ambiguities in this context.

The DISTINCT ON phrase is not part of SQL92. Nor are LIMIT and OFFSET.

In SQL92, an ORDER BY clause may only use result column names or numbers, while a GROUP BY clause may only use input column names. Postgres extends each of these clauses to allow the other choice as well (but it uses the standard's interpretation if there is ambiguity). Postgres also allows both clauses to specify arbitrary expressions. Note that names appearing in an expression will always be taken as input-column names, not as result-column names.

UNION Clause

The SQL92 syntax for UNION allows an additional CORRESPONDING BY clause:

```
table_query UNION [ALL]
    [CORRESPONDING [BY (column [,...])]]
    table_query
```

The CORRESPONDING BY clause is not supported by Postgres.

Name

SELECT INTO — Create a new table from an existing table or view

Synopsis

<refsynopsisdivinfo>1999-07-20</refsynopsisdivinfo>

```
SELECT [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]
    expression [ AS name ] [, ...]
    [ INTO [ TEMPORARY | TEMP ] [ TABLE ] new_table ]
    [ FROM table [ alias ] [, ...] ]
    [ WHERE condition ]
    [ GROUP BY column [, ...] ]
    [ HAVING condition [, ...] ]
    [ { UNION [ ALL ] | INTERSECT | EXCEPT } select ]
    [ ORDER BY column [ ASC | DESC | USING operator ] [, ...] ]
    [ FOR UPDATE [ OF class_name [, ...] ] ]
    LIMIT { count | ALL } [ { OFFSET | , } start ]
```

Inputs

All input fields are described in detail for SELECT .

Outputs

All output fields are described in detail for SELECT .

Description

`SELECT INTO` creates a new table from the results of a query. Typically, this query draws data from an existing table, but any SQL query is allowed.

Note

CREATE TABLE AS is functionally equivalent to the `SELECT INTO` command.

Name

SET — Set run-time parameters for session

Synopsis

<refsynopsisdivinfo>1999-07-20</refsynopsisdivinfo>

```
SET variable { TO | = } { value | 'value' | DEFAULT }
SET CONSTRAINTS {ALL | constraintlist} mode
SET TIME ZONE { 'timezone' | LOCAL | DEFAULT }
SET TRANSACTION ISOLATION LEVEL { READ COMMITTED | SERIALIZABLE }
```

Inputs

variable

Settable global parameter.

value

New value of parameter. DEFAULT can be used to specify resetting the parameter to its default value. Lists of strings are allowed, but more complex constructs may need to be single or double quoted.

The possible variables and allowed values are:

CLIENT_ENCODING | NAMES

Sets the multi-byte client encoding. Parameters are:

value

Sets the multi-byte client encoding to *value*. The specified encoding must be supported by the backend.

This option is only available if MULTIBYTE support was enabled during the configure step of building Postgres.

DATESTYLE

Set the date/time representation style. Affects the output format, and in some cases it can affect the interpretation of input.

ISO

use ISO 8601-style dates and times

SQL

use Oracle/Ingres-style dates and times

Postgres

use traditional Postgres format

European

use dd/mm/yyyy for numeric date representations.

NonEuropean

use mm/dd/yyyy for numeric date representations.

German

use `dd.mm.yyyy` for numeric date representations.

US

same as `NonEuropean`

DEFAULT

restores the default values (`ISO`)

Date format initialization may be done by:

Setting the `PGDATESTYLE` environment variable. If PGDATESTYLE is set in the frontend environment of a client based on libpq, libpq will automatically set DATESTYLE to the value of PGDATESTYLE during connection startup.
Running postmaster using the option `-o -e` to set dates to the `European` convention. Note that this affects only some combinations of date styles; for example the ISO style is not affected by this parameter.
Changing variables in `src/backend/utils/init/globals.c`.

The variables in `globals.c` which can be changed are:

bool EuroDates = false | true
int DateStyle = USE_ISO_DATES | USE_POSTGRES_DATES | USE_SQL_DATES | USE_GERMAN_DATES

SEED

Sets the internal seed for the random number generator.

value

The value for the seed to be used by the `random` catalog function. Significant values are floating point numbers between 0 and 1, which are then multiplied by RAND_MAX. This product will silently overflow if a number outside the range is used.

The seed can also be set by invoking the `setseed` SQL function:

```
SELECT setseed(value);
```

This option is only available if MULTIBYTE support was enabled during the configure step of building Postgres.

SERVER_ENCODING

Sets the multi-byte server encoding to:

value

The identifying value for the server encoding.

This option is only available if MULTIBYTE support was enabled during the configure step of building Postgres.

CONSTRAINTS

SET CONSTRAINTS affects the behavior of constraint evaluation in the current transaction. SET CONSTRAINTS, specified in SQL3, has these allowed parameters:

constraintlist

Comma separated list of deferrable constraint names.

mode

The constraint mode. Allowed values are DEFERRED and IMMEDIATE.

In IMMEDIATE mode, foreign key constraints are checked at the end of each query.

In DEFERRED mode, foreign key constraints marked as DEFERRABLE are checked only at transaction commit or until its mode is explicitly set to IMMEDIATE. This is actually only done for foreign key constraints, so it does not apply to UNIQUE or other constraints.

TIME ZONE
TIMEZONE

The possible values for timezone depends on your operating system. For example on Linux /usr/lib/zoneinfo contains the database of timezones.

Here are some valid values for timezone:

PST8PDT

set the timezone for California

Portugal

set time zone for Portugal.

'Europe/Rome'

set time zone for Italy.

DEFAULT

set time zone to your local timezone (value of the TZ environment variable).

If an invalid time zone is specified, the time zone becomes GMT (on most systems anyway).

The second syntax shown above, allows one to set the timezone with a syntax similar to SQL92 SET TIME ZONE. The LOCAL keyword is just an alternate form of DEFAULT for SQL92 compatibility.

If the PGTZ environment variable is set in the frontend environment of a client based on libpq, libpq will automatically set TIMEZONE to the value of PGTZ during connection startup.

TRANSACTION ISOLATION LEVEL

Sets the isolation level for the current transaction.

READ COMMITTED

The current transaction queries read only rows committed before a query began. READ COMMITTED is the default.

Note

SQL92 standard requires SERIALIZABLE to be the default isolation level.

SERIALIZABLE

The current transaction queries read only rows committed before first DML statement (SELECT/INSERT/DELETE/UPDATE/FETCH/COPY_TO) was executed in this transaction.

There are also several internal or optimization parameters which can be specified by the `SET` command:

PG_OPTIONS

Sets various backend parameters.

RANDOM_PAGE_COST

Sets the optimizer's estimate of the cost of a nonsequentially fetched disk page. This is measured as a multiple of the cost of a sequential page fetch.

`float8`

Set the cost of a random page access to the specified floating-point value.

CPU_TUPLE_COST

Sets the optimizer's estimate of the cost of processing each tuple during a query. This is measured as a fraction of the cost of a sequential page fetch.

`float8`

Set the cost of per-tuple CPU processing to the specified floating-point value.

CPU_INDEX_TUPLE_COST

Sets the optimizer's estimate of the cost of processing each index tuple during an index scan. This is measured as a fraction of the cost of a sequential page fetch.

`float8`

Set the cost of per-index-tuple CPU processing to the specified floating-point value.

CPU_OPERATOR_COST

Sets the optimizer's estimate of the cost of processing each operator in a WHERE clause. This is measured as a fraction of the cost of a sequential page fetch.

`float8`

Set the cost of per-operator CPU processing to the specified floating-point value.

EFFECTIVE_CACHE_SIZE

Sets the optimizer's assumption about the effective size of the disk cache (that is, the portion of the kernel's disk cache that will be used for Postgres data files). This is measured in disk pages, which are normally 8Kb apiece.

`float8`

Set the assumed cache size to the specified floating-point value.

ENABLE_SEQSCAN

Enables or disables the planner's use of sequential scan plan types. (It's not possible to suppress sequential scans entirely, but turning this variable OFF discourages the planner from using one if there is any other method available.)

ON

enables use of sequential scans (default setting).

OFF

disables use of sequential scans.

ENABLE_INDEXSCAN

Enables or disables the planner's use of index scan plan types.

ON

enables use of index scans (default setting).

OFF

disables use of index scans.

ENABLE_TIDSCAN

Enables or disables the planner's use of TID scan plan types.

ON

enables use of TID scans (default setting).

OFF

disables use of TID scans.

ENABLE_SORT

Enables or disables the planner's use of explicit sort steps. (It's not possible to suppress explicit sorts entirely, but turning this variable OFF discourages the planner from using one if there is any other method available.)

ON

enables use of sorts (default setting).

OFF

disables use of sorts.

ENABLE_NESTLOOP

Enables or disables the planner's use of nested-loop join plans. (It's not possible to suppress nested-loop joins entirely, but turning this variable OFF discourages the planner from using one if there is any other method available.)

ON

enables use of nested-loop joins (default setting).

OFF

disables use of nested-loop joins.

ENABLE_MERGEJOIN

Enables or disables the planner's use of mergejoin plans.

ON

enables use of merge joins (default setting).

OFF

disables use of merge joins.

ENABLE_HASHJOIN

Enables or disables the planner's use of hashjoin plans.

ON

enables use of hash joins (default setting).

OFF

disables use of hash joins.

GEQO

Sets the threshold for using the genetic optimizer algorithm.

ON

enables the genetic optimizer algorithm for statements with 11 or more tables. (This is also the DEFAULT setting.)

ON=*#*

Takes an integer argument to enable the genetic optimizer algorithm for statements with *#* or more tables in the query.

OFF

disables the genetic optimizer algorithm.

See the chapter on GEQO in the Programmer's Guide for more information about query optimization.

If the PGGEQO environment variable is set in the frontend environment of a client based on libpq, libpq will automatically set GEQO to the value of PGGEQO during connection startup.

KSQO

Key Set Query Optimizer causes the query planner to convert queries whose WHERE clause contains many OR'ed AND clauses (such as "WHERE (a=1 AND b=2) OR (a=2 AND b=3) ...") into a UNION query. This method can be faster than the default implementation, but it doesn't necessarily give exactly the same results, since UNION implicitly adds a SELECT DISTINCT clause to eliminate identical output rows. KSQO is commonly used when working with products like MicroSoft Access, which tend to generate queries of this form.

ON

enables this optimization.

OFF

disables this optimization (default setting).

DEFAULT

Equivalent to specifying `SET KSQO=OFF`.

The KSQO algorithm used to be absolutely essential for queries with many OR'ed AND clauses, but in Postgres 7.0 and later the standard planner handles these queries fairly successfully.

MAX_EXPR_DEPTH

Sets the maximum expression nesting depth that the parser will accept. The default value is high enough for any normal query, but you can raise it if you need to. (But if you raise it too high, you run the risk of backend crashes due to stack overflow.)

integer

Maximum depth.

Outputs

`SET VARIABLE`

Message returned if successful.

`NOTICE: Bad value for` *`variable`* `(`*`value`*`)`

If the command fails to set the specified variable.

Description

`SET` will modify configuration parameters for variable during a session.

Current values can be obtained using `SHOW`, and values can be restored to the defaults using `RESET`. Parameters and values are case-insensitive. Note that the value field is always specified as a string, so is enclosed in single-quotes.

`SET TIME ZONE` changes the session's default time zone offset. An SQL-session always begins with an initial default time zone offset. The `SET TIME ZONE` statement is used to change the default time zone offset for the current SQL session.

Notes

The `SET` *`variable`* statement is a Postgres language extension.

Refer to `SHOW` and `RESET` to display or reset the current values.

Usage

Set the style of date to ISO (no quotes on the argument is required):

```
SET DATESTYLE TO ISO;
```

Enable GEQO for queries with 4 or more tables (note the use of single quotes to handle the equal sign inside the value argument):

```
SET GEQO = 'ON=4';
```

Set GEQO to default:

```
SET GEQO = DEFAULT;
```

Set the timezone for Berkeley, California, using double quotes to preserve the uppercase attributes of the time zone specifier:

```
SET TIME ZONE "PST8PDT";
SELECT CURRENT_TIMESTAMP AS today;

         today
------------------------
 1998-03-31 07:41:21-08
```

Set the timezone for Italy (note the required single or double quotes to handle the special characters):

```
SET TIME ZONE 'Europe/Rome';
SELECT CURRENT_TIMESTAMP AS today;

         today
------------------------
 1998-03-31 17:41:31+02
```

Compatibility

SQL92

There is no general `SET` *`variable`* in SQL92 (with the exception of `SET TRANSACTION ISOLATION LEVEL`). The SQL92 syntax for `SET TIME ZONE` is slightly different, allowing only a single integer value for time zone specification:

```
SET TIME ZONE { interval_value_expression | LOCAL }
```

Name

SHOW — Shows run-time parameters for session

Synopsis

<refsynopsisdivinfo>1999-07-20</refsynopsisdivinfo>

```
SHOW keyword
```

Inputs

keyword

Refer to SET for more information on available variables.

Outputs

`NOTICE: variable is value`

Message returned if successful.

`NOTICE: Unrecognized variable value`

Message returned if variable does not exist.

`NOTICE: Time zone is unknown`

If the `TZ` or `PGTZ` environment variable is not set.

Description

`SHOW` will display the current setting of a run-time parameter during a session.

These variables can be set using the `SET` statement, and can be restored to the default values using the `RESET` statement. Parameters and values are case-insensitive.

Notes

See also SET and RESET to manipulate variable values.

Usage

Show the current `DateStyle` setting:

```
SHOW DateStyle;
NOTICE:  DateStyle is ISO with US (NonEuropean) conventions
```

Show the current genetic optimizer (`geqo`) setting:

```
SHOW GEQO;
NOTICE:  GEQO is ON beginning with 11 relations
```

Compatibility

SQL92

There is no `SHOW` defined in SQL92.

Name

TRUNCATE — Empty a table

Synopsis

<refsynopsisdivinfo>1999-07-20</refsynopsisdivinfo>

```
TRUNCATE [ TABLE ] name
```

Inputs

name

The name of the table to be truncated.

Outputs

`TRUNCATE`

Message returned if the table is successfully truncated.

Description

`TRUNCATE` quickly removes all rows from a table. It has the same effect as an unqualified `DELETE` but since it does not actually scan the table it is faster. This is most effective on large tables.

Usage

Truncate the table `bigtable`:

```
TRUNCATE TABLE bigtable;
```

Compatibility

SQL92

There is no `TRUNCATE` in SQL92.

Name

UNLISTEN — Stop listening for notification

Synopsis

<refsynopsisdivinfo>1998-10-19</refsynopsisdivinfo>

```
UNLISTEN { notifyname | * }
```

Inputs

notifyname

Name of previously registered notify condition.

*

All current listen registrations for this backend are cleared.

Outputs

UNLISTEN

Acknowledgement that statement has executed.

Description

UNLISTEN is used to remove an existing NOTIFY registration. UNLISTEN cancels any existing registration of the current Postgres session as a listener on the notify condition *notifyname*. The special condition wildcard "*" cancels all listener registrations for the current session.

NOTIFY contains a more extensive discussion of the use of LISTEN and NOTIFY.

Notes

classname needs not to be a valid class name but can be any string valid as a name up to 32 characters long.

The backend does not complain if you UNLISTEN something you were not listening for. Each backend will automatically execute UNLISTEN * when exiting.

A restriction in some previous releases of Postgres that a *classname* which does not correspond to an actual table must be enclosed in double-quotes is no longer present.

Usage

To subscribe to an existing registration:

```
postgres=> LISTEN virtual;
LISTEN
postgres=> NOTIFY virtual;
NOTIFY
Asynchronous NOTIFY 'virtual' from backend with pid '8448' received
```

Once UNLISTEN has been executed, further NOTIFY commands will be ignored:

```
postgres=> UNLISTEN virtual;
UNLISTEN
```

```
postgres=> NOTIFY virtual;
NOTIFY
-- notice no NOTIFY event is received
```

Compatibility

SQL92

There is no `UNLISTEN` in SQL92.

Name

UPDATE — Replaces values of columns in a table

Synopsis

<refsynopsisdivinfo>1999-07-20</refsynopsisdivinfo>

```
UPDATE table SET col = expression [, ...]
    [ FROM fromlist ]
    [ WHERE condition ]
```

Inputs

table

The name of an existing table.

column

The name of a column in *table*.

expression

A valid expression or value to assign to column.

fromlist

A Postgres non-standard extension to allow columns from other tables to appear in the WHERE condition.

condition

Refer to the SELECT statement for a further description of the WHERE clause.

Outputs

`UPDATE #`

Message returned if successful. The # means the number of rows updated. If # is equal 0 no rows are updated.

Description

`UPDATE` changes the values of the columns specified for all rows which satisfy condition. Only the columns to be modified need appear as columns in the statement.

Array references use the same syntax found in SELECT . That is, either single array elements, a range of array elements or the entire array may be replaced with a single query.

You must have write access to the table in order to modify it, as well as read access to any table whose values are mentioned in the WHERE condition.

Usage

Change word "Drama" with "Dramatic" on column kind:

```
UPDATE films SET kind = 'Dramatic' WHERE kind = 'Drama';
SELECT * FROM films WHERE kind = 'Dramatic' OR kind = 'Drama';
```

```
 code  |     title     | did | date_prod  |   kind   | len
-------+---------------+-----+------------+----------+-------
 BL101 | The Third Man | 101 | 1949-12-23 | Dramatic | 01:44
 P_302 | Becket        | 103 | 1964-02-03 | Dramatic | 02:28
 M_401 | War and Peace | 104 | 1967-02-12 | Dramatic | 05:57
 T_601 | Yojimbo       | 106 | 1961-06-16 | Dramatic | 01:50
 DA101 | Das Boot      | 110 | 1981-11-11 | Dramatic | 02:29
```

Compatibility

SQL92

SQL92 defines a different syntax for the positioned UPDATE statement:

```
UPDATE table SET column = expression [, ...]
    WHERE CURRENT OF cursor
```

where *cursor* identifies an open cursor.

Name

VACUUM — Clean and analyze a Postgres database

Synopsis

<refsynopsisdivinfo>1999-07-20</refsynopsisdivinfo>

```
VACUUM [ VERBOSE ] [ ANALYZE ] [ table ]
VACUUM [ VERBOSE ] ANALYZE [ table [ (column [, ...] ) ] ]
```

Inputs

VERBOSE

Prints a detailed vacuum activity report for each table.

ANALYZE

Updates column statistics used by the optimizer to determine the most efficient way to execute a query.

table

The name of a specific table to vacuum. Defaults to all tables.

column

The name of a specific column to analyze. Defaults to all columns.

Outputs

VACUUM

The command has been accepted and the database is being cleaned.

NOTICE: --Relation *table*--

The report header for *table*.

```
NOTICE: Pages 98: Changed 25, Reapped 74, Empty 0, New 0; Tup 1000:
Vac 3000, Crash 0, UnUsed 0, MinLen 188, MaxLen 188; Re-using: Free/
Avail. Space 586952/586952; EndEmpty/Avail. Pages 0/74. Elapsed 0/0
sec.
```

The analysis for *table* itself.

NOTICE: Index *index*: Pages 28; Tuples 1000: Deleted 3000. Elapsed 0/0 sec.

The analysis for an index on the target table.

Description

VACUUM serves two purposes in Postgres as both a means to reclaim storage and also a means to collect information for the optimizer.

VACUUM opens every class in the database, cleans out records from rolled back transactions, and updates statistics in the system catalogs. The statistics maintained include the number of tuples and number of pages stored in all classes.

VACUUM ANALYZE collects statistics representing the disbursion of the data in each column. This information is valuable when several query execution paths are possible.

Running VACUUM periodically will increase the speed of the database in processing user queries.

Notes

The open database is the target for VACUUM.

We recommend that active production databases be VACUUMM-ed nightly, in order to keep remove expired rows. After copying a large class into Postgres or after deleting a large number of records, it may be a good idea to issue a VACUUM ANALYZE query. This will update the system catalogs with the results of all recent changes, and allow the Postgres query optimizer to make better choices in planning user queries.

Usage

The following is an example from running VACUUM on a table in the regression database:

```
regression=> vacuum verbose analyze onek;
NOTICE:  --Relation onek--
NOTICE:  Pages 98: Changed 25, Reapped 74, Empty 0, New 0;
         Tup 1000: Vac 3000, Crash 0, UnUsed 0, MinLen 188, MaxLen
 188;
         Re-using: Free/Avail. Space 586952/586952; EndEmpty/Avail.
 Pages 0/74.
         Elapsed 0/0 sec.
NOTICE:  Index onek_stringu1: Pages 28; Tuples 1000: Deleted 3000.
 Elapsed 0/0 sec.
NOTICE:  Index onek_hundred: Pages 12; Tuples 1000: Deleted 3000.
 Elapsed 0/0 sec.
NOTICE:  Index onek_unique2: Pages 19; Tuples 1000: Deleted 3000.
 Elapsed 0/0 sec.
NOTICE:  Index onek_unique1: Pages 17; Tuples 1000: Deleted 3000.
 Elapsed 0/0 sec.
NOTICE:  Rel onek: Pages: 98 --> 25; Tuple(s) moved: 1000. Elapsed
 0/1 sec.
NOTICE:  Index onek_stringu1: Pages 28; Tuples 1000: Deleted 1000.
 Elapsed 0/0 sec.
NOTICE:  Index onek_hundred: Pages 12; Tuples 1000: Deleted 1000.
 Elapsed 0/0 sec.
NOTICE:  Index onek_unique2: Pages 19; Tuples 1000: Deleted 1000.
 Elapsed 0/0 sec.
NOTICE:  Index onek_unique1: Pages 17; Tuples 1000: Deleted 1000.
 Elapsed 0/0 sec.
VACUUM
```

Compatibility

SQL92

There is no VACUUM statement in SQL92.

Chapter 20. Applications

This is reference information for Postgres applications and support utilities.

Name

createdb — Create a new Postgres database

Synopsis

<refsynopsisdivinfo>1999-11-07</refsynopsisdivinfo>

```
createdb [ options ] dbname [ description ]
```

Inputs

-h, --host *host*

Specifies the hostname of the machine on which the postmaster is running.

-p, --port *port*

Specifies the Internet TCP/IP port or local Unix domain socket file extension on which the postmaster is listening for connections.

-U, --username *username*

Username to connect as.

-W, --password

Force password prompt.

-e, --echo

Echo the queries that createdb generates and sends to the backend.

-q, --quiet

Do not display a response.

-D, --location *datadir*

Specifies the alternate database location for this database installation. This is the location of the installation system tables, not the location of this specific database, which may be different.

-E, --encoding *encoding*

Specifies the character encoding scheme to be used with this database.

dbname

Specifies the name of the database to be created. The name must be unique among all Postgres databases in this installation. The default is to create a database with the same name as the current system user.

description

This optionally specifies a comment to be associated with the newly created database.

The options `-h`, `-p`, `-U`, `-W`, and `-e` are passed on literally to psql .

Outputs

`CREATE DATABASE`

The database was successfully created.

`createdb: Database creation failed.`

(Says it all.)

`createdb: Comment creation failed. (Database was created.)`

The comment/description for the database could not be created. the database itself will have been created already. You can use the SQL command `COMMENT ON DATABASE` to create the comment later on.

If there is an error condition, the backend error message will be displayed. See CREATE DATABASE and psql for possibilities.

Description

createdb creates a new Postgres database. The user who executes this command becomes the database owner.

createdb is a shell script wrapper around the SQL command CREATE DATABASE via the Postgres interactive terminal psql . Thus, there is nothing special about creating databases via this or other methods. This means that the psql must be found by the script and that a database server is running at the targeted host. Also, any default settings and environment variables available to psql and the libpq front-end library do apply.

Usage

To create the database `demo` using the default database server:

```
$ createdb demo
CREATE DATABASE
```

The response is the same as you would have gotten from running the `CREATE DATABASE` SQL command.

To create the database `demo` using the postmaster on host eden, port 5000, using the `LATIN1` encoding scheme with a look at the underlying query:

```
$ createdb -p 5000 -h eden -E LATIN1 -e demo
CREATE DATABASE "demo" WITH ENCODING = 'LATIN1'
CREATE DATABASE
```

Name

createlang — Add a new programming language to a Postgres database

Synopsis

<refsynopsisdivinfo>1999-12-05</refsynopsisdivinfo>

```
createlang [ connection options ] [ langname [ dbname ] ]
createlang [ connection options ] --list|-l [ dbname ]
```

Inputs

createlang accepts the following command line arguments:

langname

Specifies the name of the backend programming language to be defined. createlang will prompt for *langname* if it is not specified on the command line.

[-d, --dbname] *dbname*

Specifies to which database the language should be added.

-l, --list

Shows a list of already installed languages in the target database (which must be specified).

createlang also accepts the following command line arguments for connection parameters:

-h, --host *host*

Specifies the hostname of the machine on which the postmaster is running.

-p, --port *port*

Specifies the Internet TCP/IP port or local Unix domain socket file extension on which the postmaster is listening for connections.

-U, --username *username*

Username to connect as.

-W, --password

Force password prompt.

Outputs

Most error messages are self-explanatory. If not, run createlang with the `--echo` option and see under the respective SQL command for details. Check also under psql for more possibilities.

Description

createlang is a utility for adding a new programming language to a Postgres database. createlang currently accepts two languages, `plsql` and `pltcl`.

Although backend programming languages can be added directly using several SQL commands, it is recommended to use createlang because it performs a number of checks and is much easier to use. See CREATE LANGUAGE for more.

Notes

Use droplang to remove a language.

Usage

To install `pltcl`:

```
$ createlang pltcl
```

Name

createuser — Create a new Postgres user

Synopsis

<refsynopsisdivinfo>1999-11-07</refsynopsisdivinfo>

```
createuser [ options ] [ username ]
```

Inputs

-h, --host *host*

Specifies the hostname of the machine on which the postmaster is running.

-p, --port *port*

Specifies the Internet TCP/IP port or local Unix domain socket file extension on which the postmaster is listening for connections.

-e, --echo

Echo the queries that createdb generates and sends to the backend.

-q, --quiet

Do not display a response.

-d, --createdb

Allows the new user to create databases.

-D, --no-createdb

Forbids the new user to create databases.

-a, --adduser

Allows the new user to create other users.

-A, --no-adduser

Forbids the new user to create other users.

-P, --pwprompt

If given, createuser will issue a prompt for the password of the new user. This is not necessary if you do not plan on using password authentication.

-i, --sysid *uid*

Allows you to pick a non-default user id for the new user. This is not necessary, but some people like it.

username

Specifies the name of the Postgres user to be created. This name must be unique among all Postgres users.

You will be prompted for a name and other missing information if it is not specified on the command line.

The options -h, -p, and -e, are passed on literally to psql . The psql options -U and -W are available as well, but their use can be confusing in this context.

Outputs

CREATE USER

All is well.

createuser: creation of user "*username*" failed

Something went wrong. The user was not created.

If there is an error condition, the backend error message will be displayed. See CREATE USER and psql for possibilities.

Description

createuser creates a new Postgres user. Only users with usesuper set in the pg_shadow class can create new Postgres users.

createuser is a shell script wrapper around the SQL command CREATE USER via the Postgres interactive terminal psql . Thus, there is nothing special about creating users via this or other methods. This means that the psql must be found by the script and that a database server is running at the targeted host. Also, any default settings and environment variables available to psql and the libpq front-end library do apply.

Usage

To create a user joe on the default database server:

```
$ createuser joe
Is the new user allowed to create databases? (y/n) n
Shall the new user be allowed to create more new users? (y/n) n
CREATE USER
```

To create the same user joe using the postmaster on host eden, port 5000, avoiding the prompts and taking a look at the underlying query:

```
$ createuser -p 5000 -h eden -D -A -e joe
CREATE USER "joe" NOCREATEDB NOCREATEUSER
CREATE USER
```

Name

dropdb — Remove an existing Postgres database

Synopsis

<refsynopsisdivinfo>1999-11-07</refsynopsisdivinfo>

```
dropdb [ options ] dbname
```

Inputs

-h, --host *host*

Specifies the hostname of the machine on which the postmaster is running.

-p, --port *port*

Specifies the Internet TCP/IP port or local Unix domain socket file extension on which the postmaster is listening for connections.

-U, --username *username*

Username to connect as.

-W, --password

Force password prompt.

-e, --echo

Echo the queries that dropdb generates and sends to the backend.

-q, --quiet

Do not display a response.

-i, --interactive

Issues a verification prompt before doing anything destructive.

dbname

Specifies the name of the database to be removed. The database must be one of the existing Postgres databases in this installation.

The options `-h`, `-p`, `-U`, `-W`, and `-e` are passed on literally to psql .

Outputs

`DROP DATABASE`

The database was successfully removed.

`dropdb: Database removal failed.`

Something didn't work out.

If there is an error condition, the backend error message will be displayed. See DROP DATABASE and psql for possibilities.

Description

dropdb destroys an existing Postgres database. The user who executes this command must be a database superuser or the owner of the database.

dropdb is a shell script wrapper around the SQL command DROP DATABASE via the Postgres interactive terminal psql . Thus, there is nothing special about dropping databases via this or other methods. This means that the psql must be found by the script and that a database server is running at the targeted host. Also, any default settings and environment variables available to psql and the libpq front-end library do apply.

Usage

To destroy the database `demo` on the default database server:

```
$ dropdb demo
DROP DATABASE
```

To destroy the database `demo` using the postmaster on host eden, port 5000, with verification and a peek at the underlying query:

```
$ dropdb -p 5000 -h eden -i -e demo
Database "demo" will be permanently deleted.
Are you sure? (y/n) y
DROP DATABASE "demo"
DROP DATABASE
```

Name

droplang — Remove a programming language from a Postgres database

Synopsis

<refsynopsisdivinfo>1999-12-05</refsynopsisdivinfo>

```
droplang [ connection options ] [ langname [ dbname ] ]
droplang [ connection options ] --list|-l
```

Inputs

droplang accepts the following command line arguments:

langname

Specifies the name of the backend programming language to be removed. droplang will prompt for *langname* if it is not specified on the command line.

[-d, --dbname] *dbname*

Specifies from which database the language should be removed.

-l, --list

Shows a list of already installed languages in the target database (which must be specified).

droplang also accepts the following command line arguments for connection parameters:

-h, --host *host*

Specifies the hostname of the machine on which the postmaster is running.

-p, --port *port*

Specifies the Internet TCP/IP port or local Unix domain socket file extension on which the postmaster is listening for connections.

-U, --username *username*

Username to connect as.

-W, --password

Force password prompt.

Outputs

Most error messages are self-explanatory. If not, run droplang with the `--echo` option and see under the respective SQL command for details. Check also under psql for more possibilities.

Description

droplang is a utility for removing an existing programming language from a Postgres database. droplang currently accepts two languages, `plsql` and `pltcl`.

Although backend programming languages can be removed directly using several SQL commands, it is recommended to use droplang because it performs a number of checks and is much easier to use. See DROP LANGUAGE for more.

Notes

Use createlang to add a language.

Usage

To remove `pltcl`:

```
$ droplang pltcl
```

Name

dropuser — Drops (removes) a Postgres user

Synopsis

<refsynopsisdivinfo>1999-11-07</refsynopsisdivinfo>

```
dropuser [ options ] [ username ]
```

Inputs

-h, --host *host*

Specifies the hostname of the machine on which the postmaster is running.

-p, --port *port*

Specifies the Internet TCP/IP port or local Unix domain socket file extension on which the postmaster is listening for connections.

-e, --echo

Echo the queries that createdb generates and sends to the backend.

-q, --quiet

Do not display a response.

-i, --interactive

Prompt for confirmation before actually removing the user.

username

Specifies the name of the Postgres user to be removed. This name must exist in the Postgres installation. You will be prompted for a name if none is specified on the command line.

The options `-h`, `-p`, and `-e`, are passed on literally to psql . The psql options `-U` and `-W` are available as well, but they can be confusing in this context.

Outputs

`DROP USER`

All is well.

`dropuser: deletion of user "`*username*`" failed`

Something went wrong. The user was not removed.

If there is an error condition, the backend error message will be displayed. See DROP USER and psql for possibilities.

Description

dropuser removes an existing Postgres user *and* the databases which that user owned. Only users with `usesuper` set in the `pg_shadow` class can destroy Postgres users.

dropuser is a shell script wrapper around the SQL command DROP USER via the Postgres interactive terminal psql . Thus, there is nothing special about removing users via this or other methods. This

means that the psql must be found by the script and that a database server is running at the targeted host. Also, any default settings and environment variables available to psql and the libpq front-end library do apply.

Usage

To remove user `joe` from the default database server:

```
$ dropuser joe
DROP USER
```

To remove user `joe` using the postmaster on host eden, port 5000, with verification and a peek at the underlying query:

```
$ dropuser -p 5000 -h eden -i -e joe
User "joe" and any owned databases will be permanently deleted.
Are you sure? (y/n) y
DROP USER "joe"
DROP USER
```

Name

ecpg — Embedded SQL C preprocessor

Synopsis

<refsynopsisdivinfo>1999-07-20</refsynopsisdivinfo>

```
ecpg [ -v ] [ -t ] [ -I include-path ] [ -o outfile ]  file1
 [ file2 ] [ ... ]
```

Inputs

ecpg accepts the following command line arguments:

-v

Print version information.

-t

Turn off auto-transactin mode.

-I *path*

Specify an additional include path. Defaults are `.`, `/usr/local/include`, the Postgres include path which is defined at compile time (default: `/usr/local/pgsql/lib`), and `/usr/include`.

-o

Specifies that ecpg should write all its output to outfile. If no such option is given the output is written to *name*`.c`, assuming the input file was named *name*`.pgc`. If the input file does have the expected `.pgc` suffix, then the output file will have `.pgc` appended to the input file name.

file

The files to be processed.

Outputs

ecpg will create a file or write to `stdout`.

return value

ecpg returns 0 to the shell on successful completion, -1 for errors.

Description

ecpg is an embedded SQL preprocessor for the C language and the Postgres. It enables development of C programs with embedded SQL code.

Linus Tolke[1] was the original author of ecpg (up to version 0.2). Michael Meskes[2] is the current author and maintainer of ecpg. Thomas Good[3] is the author of the last revision of the ecpg man page, on which this document is based.

[1] linus@epact.se
[2] meskes@debian.org
[3] tomg@q8.nrnet.org

Usage

Preprocessing for Compilation

An embedded SQL source file must be preprocessed before compilation:

```
ecpg [ -d ] [ -o file ] file.pgc
```

where the optional `-d` flag turns on debugging. The `.pgc` extension is an arbitrary means of denoting ecpg source.

You may want to redirect the preprocessor output to a log file.

Compiling and Linking

Assuming the Postgres binaries are in `/usr/local/pgsql`, you will need to compile and link your preprocessed source file:

```
gcc -g -I /usr/local/pgsql/include [ -o file ] file.c -L /usr/
local/pgsql/lib -lecpg -lpq
```

Grammar

Libraries

The preprocessor will prepend two directives to the source:

```
#include <ecpgtype.h>
#include <ecpglib.h>
```

Variable Declaration

Variables declared within ecpg source code must be prepended with:

```
EXEC SQL BEGIN DECLARE SECTION;
```

Similarly, variable declaration sections must terminate with:

```
EXEC SQL END DECLARE SECTION;
```

Note

Prior to version 2.1.0, each variable had to be declared on a separate line. As of version 2.1.0 multiple variables may be declared on a single line:

```
char  foo(16), bar(16);
```

Error Handling

The SQL communication area is defined with:

```
EXEC SQL INCLUDE sqlca;
```

Note

The `sqlca` is in lowercase. While SQL convention may be followed, i.e., using uppercase to separate embedded SQL from C statements, sqlca (which includes the sqlca.h header file) MUST be lowercase. This is because the EXEC SQL prefix indicates that this INCLUDE will be parsed by ecpg. ecpg observes case sensitivity (SQLCA.h will not be found.) `EXEC SQL INCLUDE` can be used to include other header files as long as case sensitivity is observed.

The sqlprint command is used with the EXEC SQL WHENEVER statement to turn on error handling throughout the program:

```
EXEC SQL WHENEVER sqlerror sqlprint;
```

and

```
EXEC SQL WHENEVER not found sqlprint;
```

Note

This is *not* an exhaustive example of usage for the `EXEC SQL WHENEVER` statement. Further examples of usage may be found in SQL manuals (e.g., `The LAN TIMES Guide to SQL' by Groff and Weinberg).

Connecting to the Database Server

One connects to a database using the following:

```
EXEC SQL CONNECT dbname;
```

where the database name is not quoted. Prior to version 2.1.0, the database name was required to be inside single quotes.

Specifying a server and port name in the connect statement is also possible. The syntax is:

```
dbname[@server][:port]
```

or

```
<tcp|unix>:postgresql://server[:port][/dbname][?options]
```

Queries

In general, SQL queries acceptable to other applications such as psql can be embedded into your C code. Here are some examples of how to do that.

Create Table:

```
EXEC SQL CREATE TABLE foo (number int4, ascii char(16));
EXEC SQL CREATE UNIQUE index num1 on foo(number);
EXEC SQL COMMIT;
```

Insert:

```
EXEC SQL INSERT INTO foo (number, ascii) VALUES (9999, 'doodad');
EXEC SQL COMMIT;
```

Delete:

```
EXEC SQL DELETE FROM foo WHERE number = 9999;
EXEC SQL COMMIT;
```

Singleton Select:

```
EXEC SQL SELECT foo INTO :FooBar FROM table1 WHERE ascii =
 'doodad';
```

Select using Cursors:

```
EXEC SQL DECLARE foo_bar CURSOR FOR
    SELECT number, ascii FROM foo
    ORDER BY ascii;
EXEC SQL FETCH foo_bar INTO :FooBar, DooDad;
...
EXEC SQL CLOSE foo_bar;
EXEC SQL COMMIT;
```

Updates:

```
EXEC SQL UPDATE foo
    SET ascii = 'foobar'
    WHERE number = 9999;
EXEC SQL COMMIT;
```

Notes

There is no `EXEC SQL PREPARE` statement.

The complete structure definition MUST be listed inside the declare section.

See the `TODO` file in the source for some more missing features.

Name

pgaccess — Postgres graphical interactive client

Synopsis

<refsynopsisdivinfo>1999-08-03</refsynopsisdivinfo>

```
pgaccess [ dbname ]
```

Inputs

dbname

The name of an existing database to access.

Outputs

Description

pgaccess provides a graphical interface for Postgres where you can manage your tables, edit them, define queries, sequences and functions.

Another way of accessing Postgres through tcl is to use pgtclsh or pgtksh .

pgaccess can:

- Opens any database on a specified host at the specified port, username and password.
- Execute VACUUM .
- Saves preferences in `~/.pgaccessrc` file.

For tables, pgaccess can:

- Open multiple tables for viewing, max n records (configurable).
- Resize columns by dragging the vertical grid lines.
- Wrap text in cells.
- Dynamically adjust row height when editing.
- Save table layout for every table.
- Import/export to external files (SDF,CSV).
- Use filter capabilities; enter filter like price>3.14.
- Specify sort order; enter manually the sort field(s).
- Edit in place; double click the text you want to change.
- Delete records; point to the record, press Del key.
- Add new records; save new row with right-button-click.
- Create tables with an assistant.
- Rename and delete (drop) tables.
- Retrieve information on tables, including owner, field information, indices.

For queries, pgaccess can:

- Define, edit and store *user defined queries*.
- Save view layouts.
- Store queries as views.
- Execute with optional user input parameters; e.g.

```
select * from invoices where year=[parameter "Year of selection"]
```

- View any select query result.

- Run action queries (insert, update, delete).
- Construct queries using a visual query builder with drag & drop support, table aliasing.

For sequences, pgaccess can:

- Define new instances.
- Inspect existing instances.
- Delete.

For views, pgaccess can:

- Define them by saving queries as views.
- View them, with filtering and sorting capabilities.
- Design new views.
- Delete (drop) existing views.

For functions, pgaccess can:

- Define.
- Inspect.
- Delete.

For reports, pgaccess can:

- Generate simple reports from a table (beta stage).
- Change font, size and style of fields and labels.
- Load and save reports from the database.
- Preview tables, sample postscript print.

For forms, pgaccess can:

- Open user defined forms.
- Use a form design module.
- Access record sets using a query widget.

For scripts, pgaccess can:

- Define.
- Modify.
- Call user defined scripts.

Name

pgadmin — Postgres database management and design tool for Windows 95/98/NT

Synopsis

<refsynopsisdivinfo>1999-12-24</refsynopsisdivinfo>

```
pgadmin [ datasourcename [ username [ password ] ] ]
```

Inputs

datasourcename

The name of an existing Postgres ODBC System or User Data Source.

username

A valid username for the specified *datasourcename*.

password

A valid password for the specified *datasourcename* and *username*.

Outputs

Description

pgadmin is a general purpose tool for designing, maintaining, and administering Postgres databases. It runs under Windows 95/98 and NT.

Features include:

- Arbitrary SQL entry.
- Info Browsers and 'Creators' for databases, tables, indexes, sequences, views, triggers, functions and languages.
- User, Group and Privilege configuration dialogues.
- Revision tracking with upgrade script generation.
- Configuration of Microsoft MSysConf table.
- Data Import and Export Wizards.
- Database Migration Wizard.
- Predefined reports on databases, tables, indexes, sequences, languages and views.

pgadmin is distributed separately from Postgres and may be downloaded from http://www.pgadmin.freeserve.co.uk

Name

pg_ctl — Starts, stops, and restarts postmaster

Synopsis

<refsynopsisdivinfo>2000-04-05</refsynopsisdivinfo>

```
pg_ctl [-w] [-D datadir][-p path] [-o "options"] start
pg_ctl [-w] [-D datadir] [-m [s[mart]|f[ast]|i[mmediate]]] stop
pg_ctl [-w] [-D datadir] [-m [s[mart]|f[ast]|i[mmediate]]
    [-o "options"] restart
pg_ctl [-D datadir] status
```

Inputs

-w

Wait for the database server comes up, by watching for creation of the pid file (PGDATA/postmaster.pid). Times out after 60 seconds.

-D *datadir*

Specifies the database location for this database installation.

-p *path*

Specifies the path to the postmaster image.

-o "*options*"

Specifies options to be passed directly to postmaster.

The parameters are usually surrounded by single- or double quotes to ensure that they are passed through as a group.

-m *mode*

Specifies the shutdown mode.

smart
s

smart mode waits for all the clients to logout. This is the default.

f[ast]
f

Fast mode sends SIGTERM to the backends, that means active transactions get rolled back.

immediate
i

Immediate mode sends SIGUSR1 to the backends and lets them abort. In this case, database recovery will be neccessary on the next startup.

start

Start up postmaster.

stop

Shut down postmaster.

restart

Restart the postmaster, performing a stop/start sequence.

status

Show the current state of postmaster.

Outputs

`pg_ctl: postmaster is *state* (pid: #)`

Postmaster status.

If there is an error condition, the backend error message will be displayed.

Description

pg_ctl is a utility for starting, stopping or restarting postmaster.

Usage

Starting postmaster

To start up postmaster:

```
> pg_ctl start
```

If -w is supplied, pg_ctl waits for the database server comes up, by watching for creation of the pid file (PGDATA/postmaster.pid), for up to 60 seconds.

Parameters to invoke postmaster are taken from the following sources:

- Path to postmaster: found in the command search path.
- Database directory: `PGDATA` environment variable.
- Other parameters: `PGDATA/postmaster.opts.default`.

`postmaster.opts.default` contains parameters for postmaster.

Note that `postmaster.opts.default` is installed by initdb from `lib/postmaster.opts.default.sample` under the Postgres installation directory (`lib/postmaster.opts.default.sample` is copied from `src/bin/pg_ctl/postmaster.opts.default.sample` while installing Postgres).

To override the default parameters you can use `-D`, `-p` and `-o` options.

An example of starting the postmaster, blocking until postmaster comes up is:

```
> pg_ctl -w start
```

To specify the postmaster binary path, try:

```
> pg_ctl -p /usr/local/pgsq/bin/postmaster start
```

For a postmaster using port 5433, and running without `fsync`, use:

```
> pg_ctl -o "-o -F -p 5433" start
```

Stopping postmaster

```
> pg_ctl stop
```

stops postmaster. Using the `-m` switch allows one to control *how* the backend shuts down. `-w` waits for postmaster to shut down. `-m` specifies the shut down mode.

Restarting postmaster

This is almost equivalent to stopping the postmaster then starting it again except that the parameters used before stopping it would be used too. This is done by saving them in $`PGDATA`/postmaster.opts file. `-w`, `-D`, `-m`, `-fast`, `-immediate` and `-o` can also be used in the restarting mode and they have same meanings as described above.

To restart postmaster in the simplest form:

```
> pg_ctl restart
```

To restart postmaster, waiting for it to shut down and to come up:

```
> pg_ctl -w restart
```

To restart using port 5433 and disabling fsync after restarting:

```
> pg_ctl -o "-o -F -p 5433" restart
```

postmaster status

To get status information from postmaster:

```
> pg_ctl status
```

Here is a sample output from pg_ctl:

```
pg_ctl: postmaster is running (pid: 13718)
options are:
/usr/local/src/pgsql/current/bin/postmaster
-p 5433
-D /usr/local/src/pgsql/current/data
-B 64
-b /usr/local/src/pgsql/current/bin/postgres
-N 32
-o '-F'
```

Name

pg_dump — Extract a Postgres database into a script file

Synopsis

<refsynopsisdivinfo>1999-07-20</refsynopsisdivinfo>

```
pg_dump [ dbname ]
pg_dump [ -h host ] [ -p port ]
     [ -t table ]
     [ -a ] [ -c ] [ -d ] [ -D ] [ -i ] [ -n ] [ -N ]
     [ -o ] [ -s ] [ -u ] [ -v ] [ -x ]
     [ dbname ]
```

Inputs

pg_dump accepts the following command line arguments:

dbname

Specifies the name of the database to be extracted. *dbname* defaults to the value of the USER environment variable.

-a

Dump out only the data, no schema (definitions).

-c

Clean(drop) schema prior to create.

-d

Dump data as proper insert strings.

-D

Dump data as inserts with attribute names

-i

Ignore version mismatch between pg_dump and the database server. Since pg_dump knows a great deal about system catalogs, any given version of pg_dump is only intended to work with the corresponding release of the database server. Use this option if you need to override the version check (and if pg_dump then fails, don't say you weren't warned).

-n

Suppress double quotes around identifiers unless absolutely necessary. This may cause trouble loading this dumped data if there are reserved words used for identifiers. This was the default behavior for pg_dump prior to v6.4.

-N

Include double quotes around identifiers. This is the default.

-o

Dump object identifiers (OIDs) for every table.

-s

Dump out only the schema (definitions), no data.

-t *table*

Dump data for *table* only.

-u

Use password authentication. Prompts for username and password.

-v

Specifies verbose mode

-x

Prevent dumping of ACLs (grant/revoke commands) and table ownership information.

pg_dump also accepts the following command line arguments for connection parameters:

-h *host*

Specifies the hostname of the machine on which the postmaster is running. Defaults to using a local Unix domain socket rather than an IP connection..

-p *port*

Specifies the Internet TCP/IP port or local Unix domain socket file extension on which the postmaster is listening for connections. The port number defaults to 5432, or the value of the PGPORT environment variable (if set).

-u

Use password authentication. Prompts for *username* and *password*.

Outputs

pg_dump will create a file or write to stdout.

Connection to database 'template1' failed. connectDB() failed: Is the postmaster running and accepting connections at 'UNIX Socket' on port '*port*'?

pg_dump could not attach to the postmaster process on the specified host and port. If you see this message, ensure that the postmaster is running on the proper host and that you have specified the proper port. If your site uses an authentication system, ensure that you have obtained the required authentication credentials.

Connection to database '*dbname*' failed. FATAL 1: SetUserId: user '*username*' is not in 'pg_shadow'

You do not have a valid entry in the relation pg_shadow and and will not be allowed to access Postgres. Contact your Postgres administrator.

dumpSequence(*table*): SELECT failed

You do not have permission to read the database. Contact your Postgres site administrator.

Note

pg_dump internally executes SELECT statements. If you have problems running pg_dump, make sure you are able to select information from the database using, for example, psql.

Description

pg_dump is a utility for dumping out a Postgres database into a script file containing query commands. The script files are in text format and can be used to reconstruct the database, even on other machines and other architectures. pg_dump will produce the queries necessary to re-generate all user-defined types, functions, tables, indices, aggregates, and operators. In addition, all the data is copied out in text format so that it can be readily copied in again, as well as imported into tools for editing.

pg_dump is useful for dumping out the contents of a database to move from one Postgres installation to another. After running pg_dump, one should examine the output script file for any warnings, especially in light of the limitations listed below.

Notes

pg_dump has a few limitations. The limitations mostly stem from difficulty in extracting certain meta-information from the system catalogs.

- pg_dump does not understand partial indices. The reason is the same as above; partial index predicates are stored as plans.
- pg_dump does not handle large objects. Large objects are ignored and must be dealt with manually.
- When doing a data only dump, pg_dump emits queries to disable triggers on user tables before inserting the data and queries to reenable them after the data has been inserted. If the restore is stopped in the middle, the system catalogs may be left in the wrong state.

Usage

To dump a database of the same name as the user:

```
% pg_dump > db.out
```

To reload this database:

```
% psql -e database < db.out
```

Name

pg_dumpall — Extract all Postgres databases into a script file

Synopsis

<refsynopsisdivinfo>1999-07-20</refsynopsisdivinfo>

```
pg_dumpall
pg_dumpall [ -h host ] [ -p port ] [ -a ] [ -d ] [ -D ] [ -O ] [ -
s ] [ -u ] [ -v ] [ -x ]
```

Inputs

pg_dumpall accepts the following command line arguments:

-a

Dump out only the data, no schema (definitions).

-d

Dump data as proper insert strings.

-D

Dump data as inserts with attribute names

-n

Suppress double quotes around identifiers unless absolutely necessary. This may cause trouble loading this dumped data if there are reserved words used for identifiers.

-o

Dump object identifiers (OIDs) for every table.

-s

Dump out only the schema (definitions), no data.

-u

Use password authentication. Prompts for username and password.

-v

Specifies verbose mode

-x

Prevent dumping ACLs (grant/revoke commands) and table ownership information.

pg_dumpall also accepts the following command line arguments for connection parameters:

-h *host*

Specifies the hostname of the machine on which the postmaster is running. Defaults to using a local Unix domain socket rather than an IP connection..

-p *port*

Specifies the Internet TCP/IP port or local Unix domain socket file extension on which the postmaster is listening for connections. The port number defaults to 5432, or the value of the PGPORT environment variable (if set).

-u

Use password authentication. Prompts for *username* and *password*.

Outputs

pg_dumpall will create a file or write to stdout.

```
Connection to database 'template1' failed. connectDB() failed: Is the postmaster running and accepting connections at 'UNIX Socket' on port 'port'?
```

pg_dumpall could not attach to the postmaster process on the specified host and port. If you see this message, ensure that the postmaster is running on the proper host and that you have specified the proper port. If your site uses an authentication system, ensure that you have obtained the required authentication credentials.

```
Connection to database 'dbname' failed. FATAL 1: SetUserId: user 'username' is not in 'pg_shadow'
```

You do not have a valid entry in the relation pg_shadow and and will not be allowed to access Postgres. Contact your Postgres administrator.

```
dumpSequence(table): SELECT failed
```

You do not have permission to read the database. Contact your Postgres site administrator.

Note

pg_dumpall internally executes SELECT statements. If you have problems running pg_dumpall, make sure you are able to select information from the database using, for example, psql.

Description

pg_dumpall is a utility for dumping out all Postgres databases into one file. It also dumps the pg_shadow table, which is global to all databases. pg_dumpall includes in this file the proper commands to automatically create each dumped database before loading.

pg_dumpall takes all pg_dump options, but -f, -t and *dbname* should be omitted.

Refer to pg_dump for more information on this capability.

Usage

To dump all databases:

```
% pg_dumpall > db.out
```

Tip

You can use most pg_dump options for pg_dumpall.

To reload this database:

```
% psql -e template1 < db.out
```

Tip

You can use most psql options when reloading.

Name

psql — Postgres interactive terminal

Synopsis

<refsynopsisdivinfo>1999-10-26</refsynopsisdivinfo>

```
psql [ options ] [ dbname [ user ] ]
```

Summary

psql is a terminal-based front-end to Postgres. It enables you to type in queries interactively, issue them to Postgres, and see the query results. Alternatively, input can be from a file. In addition, it provides a number of meta-commands and various shell-like features to facilitate writing scripts and automating a wide variety of tasks.

Description

Connecting To A Database

psql is a regular Postgres client application. In order to connect to a database you need to know the name of your target database, the hostname and port number of the server and what user name you want to connect as. psql can be told about those parameters via command line options, namely `-d`, `-h`, `-p`, and `-U` respectively. If an argument is found that does not belong to any option it will be interpreted as the database name (or the user name, if the database name is also given). Not all these options are required, defaults do apply. If you omit the host name psql will connect via a UNIX domain socket to a server on the local host. The default port number is compile-time determined. Since the database server uses the same default, you will not have to specify the port in most cases. The default user name is your Unix username, as is the default database name. Note that you can't just connect to any database under any username. Your database administrator should have informed you about your access rights. To save you some typing you can also set the environment variables `PGDATABASE`, `PGHOST`, `PGPORT` and `PGUSER` to appropriate values.

If the connection could not be made for any reason (e.g., insufficient privileges, postmaster is not running on the server, etc.), psql will return an error and terminate.

Entering Queries

In normal operation, psql provides a prompt with the name of the database to which psql is currently connected, followed by the string "=>". For example,

```
$ psql testdb
Welcome to psql, the PostgreSQL interactive terminal.

Type:  \copyright for distribution terms
       \h for help with SQL commands
       \? for help on internal slash commands
       \g or terminate with semicolon to execute query
       \q to quit

testdb=>
```

At the prompt, the user may type in SQL queries. Ordinarily, input lines are sent to the backend when a query-terminating semicolon is reached. An end of line does not terminate a query! Thus queries can be spread over several lines for clarity. If the query was sent and without error, the query results are displayed on the screen.

Whenever a query is executed, psql also polls for asynchronous notification events generated by LISTEN and NOTIFY .

psql Meta-Commands

Anything you enter in psql that begins with an unquoted backslash is a psql meta-command that is processed by psql itself. These commands are what makes psql interesting for administration or scripting. Meta-commands are more commonly called slash or backslash commands.

The format of a psql command is the backslash, followed immediately by a command verb, then any arguments. The arguments are separated from the command verb and each other by any number of white space characters.

To include whitespace into an argument you must quote it with a single quote. To include a single quote into such an argument, precede it by a backslash. Anything contained in single quotes is furthermore subject to C-like substitutions for `\n` (new line), `\t` (tab), `\`*`digits`*, `\0`*`digits`*, and `\0x`*`digits`* (the character with the given decimal, octal, or hexadecimal code).

If an unquoted argument begins with a colon (`:`), it is taken as a variable and the value of the variable is taken as the argument instead.

Arguments that are quoted in "backticks" (`` ` ``) are taken as a command line that is passed to the shell. The output of the command (with a trailing newline removed) is taken as the argument value. The above escape sequences also apply in backticks.

Some commands take the name of an SQL identifier (such as a table name) as argument. These arguments follow the syntax rules of SQL regarding double quotes: an identifier without double quotes is coerced to lower-case. For all other commands double quotes are not special and will become part of the argument.

Parsing for arguments stops when another unquoted backslash occurs. This is taken as the beginning of a new meta-command. The special sequence `\\` (two backslashes) marks the end of arguments and continues parsing SQL queries, if any. That way SQL and psql commands can be freely mixed on a line. But in any case, the arguments of a meta-command cannot continue beyond the end of the line.

The following meta-commands are defined:

`\a`

If the current table output format is unaligned, switch to aligned. If it is not unaligned, set it to unaligned. This command is kept for backwards compatibility. See `\pset` for a general solution.

`\C` [*`title`*]

Set the title of any tables being printed as the result of a query or unset any such title. This command is equivalent to `\pset title` *`title`*. (The name of this command derives from "caption", as it was previously only used to set the caption in an HTML table.)

`\connect` (or `\c`) [*`dbname`* [*`username`*]]

Establishes a connection to a new database and/or under a user name. The previous connection is closed. If *`dbname`* is `-` the current database name is assumed.

If *`username`* is omitted the current user name is assumed.

As a special rule, `\connect` without any arguments will connect to the default database as the default user (as you would have gotten by starting psql without any arguments).

If the connection attempt failed (wrong username, access denied, etc.) the previous connection will be kept if and only if psql is in interactive mode. When executing a non-interactive script, processing will immediately stop with an error. This distinction was chosen as a user convenience against typos on the one hand, and a safety mechanism that scripts are not accidentally acting on the wrong database on the other hand.

\copy *table* [with oids] { from | to } *filename* | stdin | stdout [with delimiters '*characters*'] [with null as '*string*']

Performs a frontend (client) copy. This is an operation that runs an SQL COPY command, but instead of the backend's reading or writing the specified file, and consequently requiring backend access and special user privilege, as well as being bound to the file system accessible by the backend, psql reads or writes the file and routes the data between the backend and the local file system.

The syntax of the command is similar to that of the SQL COPY command (see its description for the details). Note that, because of this, special parsing rules apply to the \copy command. In particular, the variable substitution rules and backslash escapes do not apply.

Tip

This operation is not as efficient as the SQL COPY command because all data must pass through the client/server IP or socket connection. For large amounts of data the other technique may be preferable.

Note

Note the difference in interpretation of stdin and stdout between frontend and backend copies: in a frontend copy these always refer to psql's input and output stream. On a backend copy stdin comes from wherever the COPY itself came from (for example, a script run with the -f option), and stdout refers to the query output stream (see \o meta-command below).

\copyright

Shows the copyright and distribution terms of Postgres.

\d *relation*

Shows all columns of *relation* (which could be a table, view, index, or sequence), their types, and any special attributes such as NOT NULL or defaults, if any. If the relation is, in fact, a table, any defined indices are also listed. If the relation is a view, the view definition is also shown.

The command form \d+ is identical, but any comments associated with the table columns are shown as well.

Note

If \d is called without any arguments, it is equivalent to \dtvs which will show a list of all tables, views, and sequences. This is purely a convenience measure.

\da [*pattern*]

Lists all available aggregate functions, together with the data type they operate on. If *pattern* (a regular expression) is specified, only matching aggregates are shown.

\dd [*object*]

Shows the descriptions of *object* (which can be a regular expression), or of all objects if no argument is given. ("Object" covers aggregates, functions, operators, types, relations (tables, views, indices, sequences, large objects), rules, and triggers.) For example:

```
=> \dd version
                Object descriptions
  Name   |   What   |        Description
---------+----------+---------------------------
 version | function | PostgreSQL version string
```

```
(1 row)
```

Descriptions for objects can be generated with the `COMMENT ON` SQL command.

Note

Postgres stores the object descriptions in the pg_description system table.

`\df [ pattern ]`

Lists available functions, together with their argument and return types. If *`pattern`* (a regular expression) is specified, only matching functions are shown. If the form `\df+` is used, additional information about each function, including language and description is shown.

`\distvS [ pattern ]`

This is not the actual command name: The letters i, s, t, v, S stand for index, sequence, table, view, and system table, respectively. You can specify any or all of them in any order to obtain a listing of them, together with who the owner is.

If *`pattern`* is specified, it is a regular expression restricts the listing to those objects whose name matches. If one appends a "+" to the command name, each object is listed with its associated description, if any.

`\dl`

This is an alias for `\lo_list`, which shows a list of large objects.

`\do [ name ]`

Lists available operators with their operand and return types. If *`name`* is specified, only operators with that name will be shown.

`\dp [ pattern ]`

This is an alias for `\z` which was included for its greater mnemonic value ("display permissions").

`\dT [ pattern ]`

Lists all data types or only those that match *`pattern`*. The command form `\dT+` shows extra information.

`\edit` (or `\e`) [*`filename`*]

If *`filename`* is specified, the file is edited; after the editor exits, its content is copied back to the query buffer. If no argument is given, the current query buffer is copied to a temporary file which is then edited in the same fashion.

The new query buffer is then re-parsed according to the normal rules of psql, where the whole buffer is treated as a single line. (Thus you cannot make "scripts" this way, use `\i` for that.) This means also that if the query ends with (or rather contains) a semicolon, it is immediately executed. In other cases it will merely wait in the query buffer.

Tip

psql searches the environment variables `PSQL_EDITOR`, `EDITOR`, and `VISUAL` (in that order) for an editor to use. If all of them are unset, `/bin/vi` is run.

`\echo text [ ... ]`

Prints the arguments to the standard output, separated by one space and followed by a newline. This can be useful to intersperse information in the output of scripts. For example:

```
=> \echo `date`
Tue Oct 26 21:40:57 CEST 1999
```

If the first argument is an unquoted `-n` the the trailing newline is not written.

> **Tip**
>
> If you use the `\o` command to redirect your query output you may wish to use `\qecho` instead of this command.

`\encoding` [*encoding*]

Sets the client encoding, if you are using multibyte encodings. Without an argument, this command shows the current encoding.

`\f` [*string*]

Sets the field separator for unaligned query output. The default is "`|`" (a "pipe" symbol). See also `\pset` for a generic way of setting output options.

`\g` [{ *filename* | `|` *command* }]

Sends the current query input buffer to the backend and optionally saves the output in *filename* or pipes the output into a separate Unix shell to execute *command*. A bare `\g` is virtually equivalent to a semicolon. A `\g` with argument is a "one-shot" alternative to the `\o` command.

`\help` (or `\h`) [*command*]

Give syntax help on the specified SQL command. If *command* is not specified, then psql will list all the commands for which syntax help is available. If *command* is an asterisk ("*"), then syntax help on all SQL commands is shown.

> **Note**
>
> To simplify typing, commands that consists of several words do not have to be quoted. Thus it is fine to type **`\help alter table`**.

`\H`

Turns on HTML query output format. If the HTML format is already on, it is switched back to the default aligned text format. This command is for compatibility and convenience, but see `\pset` about setting other output options.

`\i` *filename*

Reads input from the file *filename* and executes it as though it had been typed on the keyboard.

> **Note**
>
> If you want to see the lines on the screen as they are read you must set the variable `ECHO` to `all`.

`\l` (or `\list`)

List all the databases in the server as well as their owners. Append a "+" to the command name to see any descriptions for the databases as well. If your Postgres installation was compiled with multibyte encoding support, the encoding scheme of each database is shown as well.

`\lo_export` *loid filename*

Reads the large object with OID *loid* from the database and writes it to *filename*. Note that this is subtly different from the server function `lo_export`, which acts with the permissions of the user that the database server runs as and on the server's file system.

Tip

Use `\lo_list` to find out the large object's OID.

Note

See the description of the `LO_TRANSACTION` variable for important information concerning all large object operations.

`\lo_import` *`filename`* [*`comment`*]

Stores the file into a Postgres "large object". Optionally, it associates the given comment with the object. Example:

```
foo=> \lo_import '/home/peter/pictures/photo.xcf' 'a picture of
 me'
lo_import 152801
```

The response indicates that the large object received object id 152801 which one ought to remember if one wants to access the object ever again. For that reason it is recommended to always associate a human-readable comment with every object. Those can then be seen with the `\lo_list` command.

Note that this command is subtly different from the server-side `lo_import` because it acts as the local user on the local file system, rather than the server's user and file system.

Note

See the description of the `LO_TRANSACTION` variable for important information concerning all large object operations.

`\lo_list`

Shows a list of all Postgres "large objects" currently stored in the database along with their owners.

`\lo_unlink` *`loid`*

Deletes the large object with OID *`loid`* from the database.

Tip

Use `\lo_list` to find out the large object's OID.

Note

See the description of the `LO_TRANSACTION` variable for important information concerning all large object operations.

`\o` [{*`filename`* | | *`command`*}]

Saves future query results to the file *`filename`* or pipe future results into a separate Unix shell to execute *`command`*. If no arguments are specified, the query output will be reset to `stdout`.

"Query results" includes all tables, command responses, and notices obtained from the database server, as well as output of various backslash commands that query the database (such as `\d`), but not error messages.

Tip

To intersperse text output in between query results, use `\qecho`.

`\p`

Print the current query buffer to the standard output.

`\pset` *`parameter`* [*`value`*]

This command sets options affecting the output of query result tables. *`parameter`* describes which option is to be set. The semantics of *`value`* depend thereon.

Adjustable printing options are:

`format`

Sets the output format to one of `unaligned`, `aligned`, `html`, or `latex`. Unique abbreviations are allowed. (That would mean one letter is enough.)

"Unaligned" writes all fields of a tuple on a line, separated by the currently active field separator. This is intended to create output that might be intended to be read in by other programs (tab-separated, comma-separated). "Aligned" mode is the standard, human-readable, nicely formatted text output that is default. The "HTML" and "LaTeX" modes put out tables that are intended to be included in documents using the respective mark-up language. They are not complete documents! (This might not be so dramatic in HTML, but in LaTeX you must have a complete document wrapper.)

`border`

The second argument must be a number. In general, the higher the number the more borders and lines the tables will have, but this depends on the particular format. In HTML mode, this will translate directly into the `border=...` attribute, in the others only values 0 (no border), 1 (internal dividing lines), and 2 (table frame) make sense.

`expanded` (or `x`)

Toggles between regular and expanded format. When expanded format is enabled, all output has two columns with the field name on the left and the data on the right. This mode is useful if the data wouldn't fit on the screen in the normal "horizontal" mode.

Expanded mode is supported by all four output modes.

`null`

The second argument is a string that should be printed whenever a field is null. The default is not to print anything, which can easily be mistaken for, say, an empty string. Thus, one might choose to write `\pset null '(null)'`.

`fieldsep`

Specifies the field separator to be used in unaligned output mode. That way one can create, for example, tab- or comma-separated output, which other programs might prefer. To set a tab as field separator, type `\pset fieldsep '\t'`. The default field separator is `'|'` (a "pipe" symbol).

`recordsep`

Specifies the record (line) separator to use in unaligned output mode. The default is a newline character.

`tuples_only` (or `t`)

Toggles between tuples only and full display. Full display may show extra information such as column headers, titles, and various footers. In tuples only mode, only actual table data is shown.

`title` [*text*]

Sets the table title for any subsequently printed tables. This can be used to give your output descriptive tags. If no argument is given, the title is unset.

Note

This formerly only affected HTML mode. You can now set titles in any output format.

`tableattr` (or `T`) [*text*]

Allows you to specify any attributes to be placed inside the HTML `table` tag. This could for example be `cellpadding` or `bgcolor`. Note that you probably don't want to specify `border` here, as that is already taken care of by `\pset border`.

`pager`

Toggles the list of a pager to do table output. If the environment variable `PAGER` is set, the output is piped to the specified program. Otherwise `more` is used.

In any case, psql only uses the pager if it seems appropriate. That means among other things that the output is to a terminal and that the table would normally not fit on the screen. Because of the modular nature of the printing routines it is not always possible to predict the number of lines that will actually be printed. For that reason psql might not appear very discriminating about when to use the pager and when not to.

Illustrations on how these different formats look can be seen in the Examples section.

Tip

There are various shortcut commands for `\pset`. See `\a`, `\C`, `\H`, `\t`, `\T`, and `\x`.

Note

It is an error to call `\pset` without arguments. In the future this call might show the current status of all printing options.

`\q`

Quit the psql program.

`\qecho` *text* [...]

This command is identical to `\echo` except that all output will be written to the query output channel, as set by `\o`.

`\r`

Resets (clears) the query buffer.

`\s` [*filename*]

Print or save the command line history to *filename*. If *filename* is omitted, the history is written to the standard output. This option is only available if psql is configured to use the GNU history library.

Note

As of psql version 7.0 it is no longer necessary to save the command history, since that will be done automatically on program termination. The history is also loaded automatically every time psql starts up.

`\set [ name [ value [ ... ]]]`

Sets the internal variable *name* to *value* or, if more than one value is given, to the concatenation of all of them. If no second argument is given, the variable is just set with no value. To unset a variable, use the `\unset` command.

Valid variable names can contain characters, digits, and underscores. See the section about psql variables for details.

Although you are welcome to set any variable to anything you want, psql treats several variables as special. They are documented in the section about variables.

Note

This command is totally separate from the SQL command SET .

`\t`

Toggles the display of output column name headings and row count footer. This command is equivalent to `\pset tuples_only` and is provided for convenience.

`\T table_options`

Allows you to specify options to be placed within the `table` tag in HTML tabular output mode. This command is equivalent to `\pset tableattr table_options`.

`\w {filename||command}`

Outputs the current query buffer to the file *filename* or pipes it to the Unix command *command*.

`\x`

Toggles extended row format mode. As such it is equivalent to `\pset expanded`.

`\z [ pattern ]`

Produces a list of all tables in the database with their appropriate access permissions listed. If an argument is given it is taken as a regular expression which limits the listing to those tables which match it.

```
test=> \z
Access permissions for database "test"
 Relation |           Access permissions
----------+-------------------------------------
 my_table | {"=r","joe=arwR", "group staff=ar"}
(1 row )
```

Read this as follows:

- `"=r"`: `PUBLIC` has read (`SELECT`) permission on the table.
- `"joe=arwR"`: User `joe` has read, write (`UPDATE`, `DELETE`), “append” (`INSERT`) permissions, and permission to create rules on the table.
- `"group staff=ar"`: Group `staff` has `SELECT` and `INSERT` permission.

The commands GRANT and REVOKE are used to set access permissions.

`\! [ command ]`

Escapes to a separate Unix shell or executes the Unix command *command*. The arguments are not further interpreted, the shell will see them as is.

`\?`

Get help information about the slash ("\") commands.

Command-line Options

If so configured, psql understands both standard Unix short options, and GNU-style long options. The latter are not available on all systems.

-a, --echo-all

Print all the lines to the screen as they are read. This is more useful for script processing rather than interactive mode. This is equivalent to setting the variable `ECHO` to `all`.

-A, --no-align

Switches to unaligned output mode. (The default output mode is otherwise aligned.)

-c, --command *query*

Specifies that psql is to execute one query string, *query*, and then exit. This is useful in shell scripts.

query must be either a query string that is completely parseable by the backend (i.e., it contains no psql specific features), or it is a single backslash command. Thus you cannot mix SQL and psql meta-commands. To achieve that, you could pipe the string into psql, like this: `echo "\x \\ select * from foo;" | psql`.

-d, --dbname *dbname*

Specifies the name of the database to connect to. This is equivalent to specifying *dbname* as the first non-option argument on the command line.

-e, --echo-queries

Show all queries that are sent to the backend. This is equivalent to setting the variable `ECHO` to `queries`.

-E, --echo-hidden

Echoes the actual queries generated by \d and other backslash commands. You can use this if you wish to include similar functionality into your own programs. This is equivalent to setting the variable `ECHO_HIDDEN` from within psql.

-f, --file *filename*

Use the file *filename* as the source of queries instead of reading queries interactively. After the file is processed, psql terminates. This in many ways equivalent to the internal command `\i`.

Using this option is subtly different from writing `psql < filename`. In general, both will do what you expect, but using `-f` enables some nice features such as error messages with line numbers. There is also a slight chance that using this option will reduce the startup overhead. On the other hand, the variant using the shell's input redirection is (in theory) guaranteed to yield exactly the same output that you would have gotten had you entered everything by hand.

-F, --field-separator *separator*

Use *separator* as the field separator. This is equivalent to `\pset fieldsep` or `\f`.

-h, --host *hostname*

Specifies the host name of the machine on which the postmaster is running. Without this option, communication is performed using local Unix domain sockets.

-H, --html

Turns on HTML tabular output. This is equivalent to `\pset format html` or the `\H` command.

-l, --list

Lists all available databases, then exits. Other non-connection options are ignored. This is similar to the internal command `\list`.

-o, --output *`filename`*

Put all query output into file *`filename`*. This is equivalent to the command `\o`.

-p, --port *`port`*

Specifies the TCP/IP port or, by omission, the local Unix domain socket file extension on which the postmaster is listening for connections. Defaults to the value of the `PGPORT` environment variable or, if not set, to the port specified at compile time, usually 5432.

-P, --pset *`assignment`*

Allows you to specify printing options in the style of `\pset` on the command line. Note that here you have to separate name and value with an equal sign instead of a space. Thus to set the output format to LaTeX, you could write `-P format=latex`.

-q

Specifies that psql should do its work quietly. By default, it prints welcome messages and various informational output. If this option is used, none of this happens. This is useful with the `-c` option. Within psql you can also set the `QUIET` variable to achieve the same effect.

-R, --record-separator *`separator`*

Use *`separator`* as the record separator. This is equivalent to the `\pset recordsep` command.

-s, --single-step

Run in single-step mode. That means the user is prompted before each query is sent to the backend, with the option to cancel execution as well. Use this to debug scripts.

-S, --single-line

Runs in single-line mode where a newline terminates a query, as a semicolon does.

Note

This mode is provided for those who insist on it, but you are not necessarily encouraged to use it. In particular, if you mix SQL and meta-commands on a line the order of execution might not always be clear to the inexperienced user.

-t, --tuples-only

Turn off printing of column names and result row count footers, etc. It is completely equivalent to the `\t` meta-command.

-T, --table-attr *`table_options`*

Allows you to specify options to be placed within the HTML `table` tag. See `\pset` for details.

-u

Makes psql prompt for the user name and password before connecting to the database.

This option is deprecated, as it is conceptually flawed. (Prompting for a non-default user name and prompting for a password because the backend requires it are really two different things.) You are encouraged to look at the `-U` and `-W` options instead.

-U, --username *`username`*

Connects to the database as the user *`username`* instead of the default. (You must have permission to do so, of course.)

-v, --variable, --set *`assignment`*

Performs a variable assignment, like the `\set` internal command. Note that you must separate name and value, if any, by an equal sign on the command line. To unset a variable, leave off the equal sign. These assignments are done during a very early state of startup, so variables reserved for internal purposes might get overwritten later.

-V, --version

Shows the psql version.

-W, --password

Requests that psql should prompt for a password before connecting to a database. This will remain set for the entire session, even if you change the database connection with the meta-command `\connect`.

As of version 7.0, psql automatically issues a password prompt whenever the backend requests password authentication. Because this is currently based on a "hack", the automatic recognition might mysteriously fail, hence this option to force a prompt. If no password prompt is issued and the backend requires password authentication the connection attempt will fail.

-x, --expanded

Turns on extended row format mode. This is equivalent to the command `\x`.

-X, --no-psqlrc

Do not read the startup file `~/.psqlrc`.

-?, --help

Shows help about psql command line arguments.

Advanced features

Variables

psql provides variable substitution features similar to common Unix command shells. This feature is new and not very sophisticated, yet, but there are plans to expand it in the future. Variables are simply name/value pairs, where the value can be any string of any length. To set variables, use the psql meta-command `\set`:

```
testdb=> \set foo bar
```

sets the variable "foo" to the value "bar". To retrieve the content of the variable, precede the name with a colon and use it as the argument of any slash command:

```
testdb=> \echo :foo
bar
```

Note

The arguments of `\set` are subject to the same substitution rules as with other commands. Thus you can construct interesting references such as `\set :foo 'something'` and get "soft links" or "variable variables" of Perl or PHP fame, respectively. Unfortunately (or fortunately?), there is no way to do anything useful with these constructs. On the other hand, `\set bar :foo` is a perfectly valid way to copy a variable.

If you call `\set` without a second argument, the variable is simply set, but has no value. To unset (or delete) a variable, use the command `\unset`.

psql's internal variable names can consist of letters, numbers, and underscores in any order and any number of them. A number of regular variables are treated specially by psql. They indicate certain option settings that can be changed at runtime by altering the value of the variable or represent some state of the application. Although you can use these variables for any other purpose, this is not recommended, as the program behavior might grow really strange really quickly. By convention, all specially treated variables consist of all upper-case letters (and possibly numbers and underscores). To ensure maximum compatibility in the future, avoid such variables. A list of all specially treated variables follows.

`DBNAME`

The name of the database you are currently connected to. This is set everytime you connect to a database (including program startup), but can be unset.

`ECHO`

If set to "`all`", all lines entered or from a script are written to the standard output before they are parsed or executed. To specify this on program startup, use the switch `-a`. If set to "`queries`", psql merely prints all queries as they are sent to the backend. The option for this is `-e`.

`ECHO_HIDDEN`

When this variable is set and a backslash command queries the database, the query is first shown. This way you can study the Postgres internals and provide similar functionality in your own programs. If you set the variable to the value "noexec", the queries are just shown but are not actually sent to the backend and executed.

`ENCODING`

The current client multibyte encoding. If you are not set up to use multibyte characters, this variable will always contain "SQL_ASCII".

`HISTCONTROL`

If this variable is set to `ignorespace`, lines which begin with a space are not entered into the history list. If set to a value of `ignoredups`, lines matching the previous history line are not entered. A value of `ignoreboth` combines the two options. If unset, or if set to any other value than those above, all lines read in interactive mode are saved on the history list.

Note

This feature was shamelessly plagiarized from bash.

`HISTSIZE`

The number of commands to store in the command history. The default value is 500.

Note

This feature was shamelessly plagiarized from bash.

HOST

The database server host you are currently connected to. This is set everytime you connect to a database (including program startup), but can be unset.

IGNOREEOF

If unset, sending an EOF character (usually Control-D) to an interactive session of psql will terminate the application. If set to a numeric value, that many EOF characters are ignored before the application terminates. If the variable is set but has no numeric value, the default is 10.

Note

This feature was shamelessly plagiarized from bash.

LASTOID

The value of the last affected oid, as returned from an INSERT or lo_insert commmand. This variable is only guaranteed to be valid until after the result of the next SQL command has been displayed.

LO_TRANSACTION

If you use the Postgres large object interface to specially store data that does not fit into one tuple, all the operations must be contained in a transaction block. (See the documentation of the large object interface for more information.) Since psql has no way to tell if you already have a transaction in progress when you call one of its internal commands \lo_export, \lo_import, \lo_unlink it must take some arbitrary action. This action could either be to roll back any transaction that might already be in progress, or to commit any such transaction, or to do nothing at all. In the last case you must provide your own BEGIN TRANSACTION/COMMIT block or the results will be unpredictable (usually resulting in the desired action's not being performed in any case).

To choose what you want to do you set this variable to one of "rollback", "commit", or "nothing". The default is to roll back the transaction. If you just want to load one or a few objects this is fine. However, if you intend to transfer many large objects, it might be advisable to provide one explicit transaction block around all commands.

ON_ERROR_STOP

By default, if non-interactive scripts encounter an error, such as a malformed SQL query or internal meta-command, processing continues. This has been the traditional behaviour of psql but it is sometimes not desirable. If this variable is set, script processing will immediately terminate. If the script was called from another script it will terminate in the same fashion. If the outermost script was not called from an interactive psql session but rather using the -f option, psql will return error code 3, to distinguish this case from fatal error conditions (error code 1).

PORT

The database server port to which you are currently connected. This is set every time you connect to a database (including program startup), but can be unset.

PROMPT1, PROMPT2, PROMPT3

These specify what the prompt psql issues is supposed to look like. See "Prompting" below.

QUIET

This variable is equivalent to the command line option -q. It is probably not too useful in interactive mode.

SINGLELINE

This variable is set by the command line option -S. You can unset or reset it at run time.

SINGLESTEP

This variable is equivalent to the command line option -s.

USER

The database user you are currently connected as. This is set every time you connect to a database (including program startup), but can be unset.

SQL Interpolation

An additional useful feature of psql variables is that you can substitute ("interpolate") them into regular SQL statements. The syntax for this is again to prepend the variable name with a colon (:).

```
testdb=> \set foo 'my_table'
testdb=> SELECT * FROM :foo;
```

would then query the table my_table. The value of the variable is copied literally, so it can even contain unbalanced quotes or backslash commands. You must make sure that it makes sense where you put it. Variable interpolation will not be performed into quoted SQL entities.

A popular application of this facility is to refer to the last inserted OID in subsequent statement to build a foreign key scenario. Another possible use of this mechanism is to copy the contents of a file into a field. First load the file into a variable and then proceed as above.

```
testdb=> \set content '\'' `cat my_file.txt` '\''
testdb=> INSERT INTO my_table VALUES (:content);
```

One possible problem with this approach is that my_file.txt might contain single quotes. These need to be escaped so that they don't cause a syntax error when the third line is processed. This could be done with the program sed:

```
testdb=> \set content `sed -e "s/'/\\\\\\'/g" < my_file.txt`
```

Observe the correct number of backslashes (6)! You can resolve it this way: After psql has parsed this line, it passes sed -e "s/'/\\\'/g" < my_file.txt to the shell. The shell will do it's own thing inside the double quotes and execute sed with the arguments -e and s/'/\\'/g. When sed parses this it will replace the two backslashes with a single one and then do the substitution. Perhaps at one point you thought it was great that all Unix commands use the same escape character. And this is ignoring the fact that you might have to escape all backslashes as well because SQL text constants are also subject to certain interpretations. In that case you might be better off preparing the file externally.

Since colons may legally appear in queries, the following rule applies: If the variable is not set, the character sequence "colon+name" is not changed. In any case you can escape a colon with a backslash to protect it from interpretation. (The colon syntax for variables is standard SQL for embedded query languages, such as ecpg. The colon syntax for array slices and type casts are Postgres extensions, hence the conflict.)

Prompting

The prompts psql issues can be customized to your preference. The three variables PROMPT1, PROMPT2, and PROMPT3 contain strings and special escape sequences that describe the appearance of the prompt. Prompt 1 is the normal prompt that is issued when psql requests a new query. Prompt 2 is issued when more input is expected during query input because the query was not terminated with a semicolon or a quote was not closed. Prompt 3 is issued when you run an SQL COPY command and you are expected to type in the tuples on the terminal.

The value of the respective prompt variable is printed literally, except where a percent sign ("%") is encountered. Depending on the next character, certain other text is substituted instead. Defined substitutions are:

`%M`

The full hostname (with domainname) of the database server (or "localhost" if hostname information is not available).

`%m`

The hostname of the database server, truncated after the first dot.

`%>`

The port number at which the database server is listening.

`%n`

The username you are connected as (not your local system user name).

`%/`

The name of the current database.

`%~`

Like `%/`, but the output is "~" (tilde) if the database is your default database.

`%#`

If the current user is a database superuser, then a "#", otherwise a ">".

`%R`

In prompt 1 normally "=", but "^" if in single-line mode, and "!" if the session is disconnected from the database (which can happen if `\connect` fails). In prompt 2 the sequence is replaced by "-", "*", a single quote, or a double quote, depending on whether psql expects more input because the query wasn't terminated yet, because you are inside a `/* ... */` comment, or because you are inside a quote. In prompt 3 the sequence doesn't resolve to anything.

`%`*`digits`*

If *`digits`* starts with `0x` the rest of the characters are interpreted at a hexadecimal digit and the character with the corresponding code is subsituted. If the first digit is `0` the characters are interpreted as on octal number and the corresponding character is substituted. Otherwise a decimal number is assumed.

`%:`*`name`*`:`

The value of the psql, variable *`name`*. See the section "Variables" for details.

`%` ` *`command`* `

The output of *`command`*, similar to ordinary "back-tick" substitution.

To insert a percent sign into your prompt, write `%%`. The default prompts are equivalent to `'%/%R%# '` for prompts 1 and 2, and `'>> '` for prompt 3.

Note

This feature was shamelessly plagiarized from tcsh.

Miscellaneous

psql returns 0 to the shell if it finished normally, 1 if a fatal error of its own (out of memory, file not found) occurs, 2 if the connection to the backend went bad and the session is not interactive, and 3 if an error occurred in a script and the variable `ON_ERROR_STOP` was set.

Before starting up, psql attempts to read and execute commands from the file `$HOME/.psqlrc`. It could be used to set up the client or the server to taste (using the `\set` and `SET` commands).

GNU readline

psql supports the readline and history libraries for convenient line editing and retrieval. The command history is stored in a file named `.psql_history` in your home directory and is reloaded when psql starts up. Tab-completion is also supported, although the completion logic makes no claim to be an SQL parser. When available, psql is automatically built to use these features. If for some reason you do not like the tab completion, you can turn if off by putting this in a file named `.inputrc` in your home directory:

```
$if psql
set disable-completion on
$endif
```

(This is not a psql but a readline feature. Read its documentation for further details.)

If you have the readline library installed but psql does not seem to use it, you must make sure that Postgres's top-level `configure` script finds it. `configure` needs to find both the library `libreadline.a` (or a shared library equivalent) *and* the header files `readline.h` and `history.h` (or `readline/readline.h` and `readline/history.h`) in appropriate directories. If you have the library and header files installed in an obscure place you must tell `configure` about them, for example:

```
$ ./configure --with-includes=/opt/gnu/include --with-libs=/opt/
gnu/lib  ...
```

Then you have to recompile psql (not necessarily the entire code tree).

The GNU readline library can be obtained from the GNU project's FTP server at ftp://ftp.gnu.org.

Examples

Note

This section only shows a few examples specific to psql. If you want to learn SQL or get familiar with Postgres, you might wish to read the Tutorial that is included in the distribution.

The first example shows how to spread a query over several lines of input. Notice the changing prompt.

```
testdb=> CREATE TABLE my_table (
testdb(>  first integer not null default 0,
testdb(>  second text
testdb-> );
CREATE
```

Now look at the table definition again:

```
testdb=> \d my_table
             Table "my_table"
 Attribute |  Type   |      Modifier
-----------+---------+--------------------
```

```
 first     | integer | not null default 0
 second    | text    |
```

At this point you decide to change the prompt to something more interesting:

```
testdb=> \set PROMPT1 '%n@%m %~%R%# '
peter@localhost testdb=>
```

Let's assume you have filled the table with data and want to take a look at it:

```
peter@localhost testdb=> SELECT * FROM my_table;
 first | second
-------+--------
     1 | one
     2 | two
     3 | three
     4 | four
(4 rows)
```

Notice how the int4 colums in right aligned while the text column in left aligned. You can make this table look differently by using the `\pset` command.

```
peter@localhost testdb=> \pset border 2
Border style is 2.
peter@localhost testdb=> SELECT * FROM my_table;
+-------+--------+
| first | second |
+-------+--------+
|     1 | one    |
|     2 | two    |
|     3 | three  |
|     4 | four   |
+-------+--------+
(4 rows)

peter@localhost testdb=> \pset border 0
Border style is 0.
peter@localhost testdb=> SELECT * FROM my_table;
first second
----- ------
    1 one
    2 two
    3 three
    4 four
(4 rows)

peter@localhost testdb=> \pset border 1
Border style is 1.
peter@localhost testdb=> \pset format unaligned
Output format is unaligned.
peter@localhost testdb=> \pset fieldsep ","
Field separator is ",".
peter@localhost testdb=> \pset tuples_only
Showing only tuples.
peter@localhost testdb=> SELECT second, first FROM my_table;
one,1
two,2
three,3
four,4
```

Alternatively, use the short commands:

```
peter@localhost testdb=> \a \t \x
Output format is aligned.
Tuples only is off.
Expanded display is on.
peter@localhost testdb=> SELECT * FROM my_table;
-[ RECORD 1 ]-
first  | 1
second | one
-[ RECORD 2 ]-
first  | 2
second | two
-[ RECORD 3 ]-
first  | 3
second | three
-[ RECORD 4 ]-
first  | 4
second | four
```

Appendix

Bugs and Issues

- In some earlier life psql allowed the first argument to start directly after the (single-letter) command. For compatibility this is still supported to some extent but I am not going to explain the details here as this use is discouraged. But if you get strange messages, keep this in mind. For example

  ```
  testdb=> \foo
  Field separator is "oo".
  ```

  is perhaps not what one would expect.

- psql only works smootly with servers of the same version. That does not mean other combinations will fail outright, but subtle and not-so-subtle problems might come up.

- Pressing Control-C during a “copy in” (data sent to the server) doesn't show the most ideal of behaviours. If you get a message such as “PQexec: you gotta get out of a COPY state yourself”, simply reset the connection by entering `\c - -`.

Name

pgtclsh — Postgres TCL shell client

Synopsis

<refsynopsisdivinfo>1999-08-03</refsynopsisdivinfo>

```
pgtclsh [ dbname ]
```

Inputs

dbname

The name of an existing database to access.

Outputs

Description

pgtclsh provides a TCL shell interface for Postgres.

Another way of accessing Postgres through tcl is to use pgtksh or pgaccess .

Name

pgtksh — Postgres graphical TCL/TK shell

Synopsis

<refsynopsisdivinfo>1999-08-03</refsynopsisdivinfo>

```
pgtksh [ dbname ]
```

Inputs

dbname

The name of an existing database to access.

Outputs

Description

pgtksh provides a graphical TCL/TK shell interface for Postgres.

Another way of accessing Postgres through TCL is to use pgtclsh or pgaccess .

Name

vacuumdb — Clean and analyze a Postgres database

Synopsis

<refsynopsisdivinfo>1999-12-04</refsynopsisdivinfo>

```
vacuumdb [ options ] [ --analyze | -z ]
    [ --alldb | -a ] [ --verbose | -v ]
    [ --table 'table [ ( column [,...] ) ]' ] [ [-d] dbname ]
```

Inputs

vacuumdb accepts the following command line arguments:

-d *dbname*
--dbname *dbname*

Specifies the name of the database to be cleaned or analyzed.

-z
--analyze

Calculate statistics on the database for use by the optimizer.

-a
--alldb

Vacuum all databases.

-v
--verbose

Print detailed information during processing.

-t *table* [(*column* [,...])]
--table *table* [(*column* [,...])]

Clean or analyze *table* only. Column names may be specified only in conjunction with the `--analyze` option.

Tip

If you specify columns to vacuum, you probably have to escape the parentheses from the shell.

vacuumdb also accepts the following command line arguments for connection parameters:

-h *host*
--host *host*

Specifies the hostname of the machine on which the postmaster is running.

-p *port*
--port *port*

Specifies the Internet TCP/IP port or local Unix domain socket file extension on which the postmaster is listening for connections.

-U *username*
--username *username*

Username to connect as.

-W
--password

Force password prompt.

-e
--echo

Echo the commands that vacuumdb generates and sends to the backend.

-q
--quiet

Do not display a response.

Outputs

`VACUUM`

Everything went well.

`vacuumdb: Vacuum failed.`

Something went wrong. vacuumdb is only a wrapper script. See VACUUM and psql for a detailed discussion of error messages and potential problems.

Description

vacuumdb is a utility for cleaning a Postgres database. vacuumdb will also generate internal statistics used by the Postgres query optimizer.

vacuumdb is a shell script wrapper around the backend command VACUUM via the Postgres interactive terminal psql . There is no effective difference between vacuuming databases via this or other methods. psql must be found by the script and a database server must be running at the targeted host. Also, any default settings and environment variables available to psql and the libpq front-end library do apply.

Usage

To clean the database `test`:

```
$ vacuumdb test
```

To analyze a database named `bigdb` for the optimizer:

```
$ vacuumdb --analyze bigdb
```

To analyze a single column `bar` in table `foo` in a database named `xyzzy` for the optimizer:

```
$ vacuumdb --analyze --verbose --table 'foo(bar)' xyzzy
```

Chapter 21. System Applications

This is reference information for Postgres servers and support utilities.

Name

initdb — Create a new Postgres database installation

Synopsis

<refsynopsisdivinfo>1999-12-17</refsynopsisdivinfo>

```
initdb [ --pgdata|-D dbdir ]
     [ --sysid|-i sysid ]
     [ --pwprompt|-W ]
     [ --encoding|-E encoding ]
     [ --pglib|-L libdir ]
     [ --noclean | -n ] [ --debug | -d ] [ --template | -t ]
```

Inputs

--pgdata=*dbdir*
-D *dbdir*
PGDATA

This option specifies where in the file system the database should be stored. This is the only information required by initdb, but you can avoid it by setting the PGDATA environment variable, which can be convenient since the database server (postmaster) can find the database directory later by the same variable.

--sysid=*sysid*
-i *sysid*

Selects the system id of the database superuser. This defaults to the effective user id of the user running initdb. It is really not important what the superuser's sysid is, but one might choose to start the numbering at some number like 0 or 1.

--pwprompt
-W

Makes initdb prompt for a password of the database superuser. If you don't plan on using password authentication, this is not important. Otherwise you won't be able to use password authentication until you have a password set up.

--encoding=*encoding*
-E *encoding*

Selects the multibyte encoding of the template database. This will also be the default encoding of any database you create later, unless you override it there. To use the multibyte encoding feature, you must specify so at build time, at which time you also select the default for this option.

Other, less commonly used, parameters are also available:

--pglib=*libdir*
-l *libdir*

initdb needs a few input files to initialize the database. This option tells where to find them. You normally don't have to worry about this since initdb knows about the most common installation layouts and will find the files itself. You will be told if you need to specify their location explicitly. If that happens, one of the files is called global1.bki.source and is traditionally installed along with the others in the library directory (e.g., /usr/local/pgsql/lib).

--template
-t

Replace the `template1` database in an existing database system, and don't touch anything else. This is useful when you need to upgrade your `template1` database using initdb from a newer release of Postgres, or when your `template1` database has become corrupted by some system problem. Normally the contents of `template1` remain constant throughout the life of the database system. You can't destroy anything by running initdb with the `--template` option.

--noclean
-n

By default, when initdb determines that error prevent it from completely creating the database system, it removes any files it may have created before determining that it can't finish the job. This option inhibits any tidying-up and is thus useful for debugging.

--debug
-d

Print debugging output from the bootstrap backend and a few other messages of lesser interest for the general public. The bootstrap backend is the program initdb uses to create the catalog tables. This option generates a tremendous amount of output.

Outputs

initdb will create files in the specified data area which are the system tables and framework for a complete installation.

Description

initdb creates a new Postgres database system. A database system is a collection of databases that are all administered by the same Unix user and managed by a single postmaster.

Creating a database system consists of creating the directories in which the database data will live, generating the shared catalog tables (tables that don't belong to any particular database), and creating the `template1` database. When you create a new database, everything in the `template1` database is copied. It contains catalog tables filled in for things like the builtin types.

You must not execute initdb as root. This is because you cannot run the database server as root either, but the server needs to have access to the files initdb creates. Furthermore, during the initialization phase, when there are no users and no access controls installed, postgres will only connect with the name of the current Unix user, so you must log in under the account that will own the server process.

Although initdb will attempt to create the respective data directory, chances are that it won't have the permission to do so. Thus it is a good idea to create the data directory before running initdb *and* to hand over the ownership of it to the database superuser.

Name

initlocation — Create a secondary Postgres database storage area

Synopsis

<refsynopsisdivinfo>2000-01-17</refsynopsisdivinfo>

```
initlocation directory
```

Inputs

directory

Where in your Unix filesystem do you want alternate databases to go?

Outputs

initlocation will create directories in the specified place.

Description

initlocation creates a new Postgres secondary database storage area. See the discussion under CREATE DATABASE about how to manage and use secondary storage areas. If the argument does not contain a slash and is not valid as a path, it is assumed to be an environment variable, which is referenced. See the examples at the end.

In order to use this command you must be logged in (using 'su', for example) the database superuser.

Usage

To create a database in an alternate location, using an environment variable:

```
$ export PGDATA2=/opt/postgres/data
```

Start and stop postmaster so it sees the $PGDATA2 environment variable. The system must be configured so the postmaster sees $PGDATA2 every time it starts.

```
$ initlocation PGDATA2
$ createdb -D 'PGDATA2' 'testdb'
```

Alternatively, if you allow absolute paths you could write:

```
$ initlocation /opt/postgres/data
$ createdb -D '/opt/postgres/data/testdb' testdb
```

Name

ipcclean — Clean up shared memory and semaphores from aborted backends

Synopsis

<refsynopsisdivinfo>1999-07-20</refsynopsisdivinfo>

```
ipcclean
```

Inputs

None.

Outputs

None.

Description

ipcclean cleans up shared memory and semaphore space from aborted backends by deleting all instances owned by user `postgres`. Only the DBA should execute this program as it can cause bizarre behavior (i.e., crashes) if run during multi-user execution. This program should be executed if messages such as `semget: No space left on device` are encountered when starting up the postmaster or the backend server.

If this command is executed while postmaster is running, the shared memory and semaphores allocated by the postmaster will be deleted. This will result in a general failure of the backends servers started by that postmaster.

This script is a hack, but in the many years since it was written, no one has come up with an equally effective and portable solution. Suggestions are welcome.

The script makes assumption about the format of output of the ipcs utility which may not be true across different operating systems. Therefore, it may not work on your particular OS.

Name

pg_passwd — Manipulate the flat password file

Synopsis

<refsynopsisdivinfo>1999-07-20</refsynopsisdivinfo>

```
pg_passwd filename
```

Description

pg_passwd is a tool to manipulate the flat password file functionality of Postgres. This style of password authentication is not *required* in an installation, but is one of several supported security mechanisms.

Specify the password file in the same style of `Ident` authentication in `$PGDATA/pg_hba.conf`:

```
host  unv     133.65.96.250   255.255.255.255 password passwd
```

where the above line allows access from 133.65.96.250 using the passwords listed in `$PGDATA/passwd`. The format of the password file follows those of `/etc/passwd` and `/etc/shadow`. The first field is the user name, and the second field is the encrypted password. The rest is completely ignored. Thus the following three sample lines specify the same user and password pair:

```
pg_guest:/nB7.w5Auq.BY:10031::::::
pg_guest:/nB7.w5Auq.BY:93001:930::/home/guest:/bin/tcsh
pg_guest:/nB7.w5Auq.BY:93001
```

Supply the password file to the pg_passwd command. In the case described above, after changing the working directory to `PGDATA`, the following command execution specify the new password for `pg_guest`:

```
 % pg_passwd passwd
 Username: pg_guest
 Password:
 Re-enter password:
```

where the `Password:` and `Re-enter password:` prompts require the same password input which are not displayed on the terminal. The original password file is renamed to `passwd.bk`.

psql uses the `-u` option to invoke this style of authentication.

The following lines show the sample usage of the option:

```
% psql -h hyalos -u unv
Username: pg_guest
Password:
Welcome to the POSTGRESQL interactive sql monitor:
  Please read the file COPYRIGHT for copyright terms of POSTGRESQL
   type \? for help on slash commands
   type \q to quit
   type \g or terminate with semicolon to execute query
 You are currently connected to the database: unv
unv=>
```

Perl5 authentication uses the new style of the `Pg.pm` like this:

```
$conn = Pg::connectdb("host=hyalos dbname=unv
                       user=pg_guest password=xxxxxxx");
```

For more details, refer to `src/interfaces/perl5/Pg.pm`.

Pg{tcl,tk}sh authentication uses the `pg_connect` command with the `-conninfo` option thusly:

```
% set conn [pg_connect -conninfo \\
            "host=hyalos dbname=unv \\
             user=pg_guest password=xxxxxxx "]
```

You can list all of the keys for the option by executing the following command:

```
% puts [ pg_conndefaults]
```

Name

pg_upgrade — Allows upgrade from a previous release without reloading data

Synopsis

<refsynopsisdivinfo>1999-07-31</refsynopsisdivinfo>

```
pg_upgrade [ -f filename ] old_data_dir
```

Description

pg_upgrade is a utility for upgrading from a previous Postgres release without reloading all the data. Not all Postgres release transitions can be handled this way. Check the release notes for details on your installation.

Upgrading Postgres with pg_upgrade

1. Back up your existing data directory, preferably by making a complete dump with pg_dumpall.
2. Then do:

   ```
   % pg_dumpall -s >db.out
   ```

   to dump out your old database's table definitions without any data.
3. Stop the old postmaster and all backends.
4. Rename (using mv) your old pgsql `data/` directory to `data.old/`.
5. Do

   ```
   % make install
   ```

   to install the new binaries.
6. Run initdb to create a new template1 database containing the system tables for the new release.
7. Start the new postmaster. (Note: it is critical that no users connect to the database until the upgrade is complete. You may wish to start the postmaster without -i and/or alter `pg_hba.conf` temporarily.)
8. Change your working directory to the pgsql main directory, and type:

   ```
   % pg_upgrade -f db.out data.old
   ```

   The program will do some checking to make sure everything is properly configured, and will run your db.out script to recreate all the databases and tables you had, but with no data. It will then physically move the data files containing non-system tables and indexes from `data.old/` into the proper `data/` subdirectories, replacing the empty data files created during the db.out script.
9. Restore your old `pg_hba.conf` if needed to allow user logins.
10. Stop and restart the postmaster.
11. *Carefully* examine the contents of the upgraded database. If you detect problems, you'll need to recover by restoring from your full pg_dump backup. You can delete the `data.old/` directory when you are satisfied.

12. The upgraded database will be in an un-vacuumed state. You will probably want to run a `VACUUM ANALYZE` before beginning production work.

Name

postgres — Run a Postgres single-user backend

Synopsis

<refsynopsisdivinfo>1999-07-20</refsynopsisdivinfo>

```
postgres [ dbname ]
postgres [ -B nBuffers ] [ -C ] [ -D DataDir ] [ -E ] [ -F ]
    [ -O ] [ -P ] [ -Q ] [ -S SortSize ] [ -d [ DebugLevel ] ] [ -
e ]
    [ -o ] [ OutputFile ] [ -s ] [ -v protocol ] [ dbname ]
```

Inputs

postgres accepts the following command line arguments:

dbname

The optional argument *dbname* specifies the name of the database to be accessed. *dbname* defaults to the value of the USER environment variable.

-B *nBuffers*

If the backend is running under the postmaster, *nBuffers* is the number of shared-memory buffers that the postmaster has allocated for the backend server processes that it starts. If the backend is running standalone, this specifies the number of buffers to allocate. This value defaults to 64 buffers, where each buffer is 8k bytes (or whatever BLCKSZ is set to in config.h).

-C

Do not show the server version number.

-D *DataDir*

Specifies the directory to use as the root of the tree of database directories. If -D is not given, the default data directory name is the value of the environment variable PGDATA. If PGDATA is not set, then the directory used is $POSTGRESHOME/data. If neither environment variable is set and this command-line option is not specified, the default directory that was set at compile-time is used.

-E

Echo all queries.

-F

Disable an automatic fsync() call after each transaction. This option improves performance, but an operating system crash while a transaction is in progress may cause the loss of the most recently entered data. Without the fsync() call the data is buffered by the operating system, and written to disk sometime later.

-O

Override restrictions, so system table structures can be modified. These tables are typically those with a leading "pg_" in the table name.

-P

Ignore system indexes to scan/update system tuples. REINDEX for system tables/indexes requires this option. System tables are typically those with a leading "pg_" in the table name.

-Q

Specifies "quiet" mode.

-S *SortSize*

Specifies the amount of memory to be used by internal sorts and hashes before resorting to temporary disk files. The value is specified in kilobytes, and defaults to 512 kilobytes. Note that for a complex query, several sorts and/or hashes might be running in parallel, and each one will be allowed to use as much as *SortSize* kilobytes before it starts to put data into temporary files.

-d [*DebugLevel*]

The optional argument *DebugLevel* determines the amount of debugging output the backend servers will produce. If *DebugLevel* is one, the postmaster will trace all connection traffic, and nothing else. For levels two and higher, debugging is turned on in the backend process and the postmaster displays more information, including the backend environment and process traffic. Note that if no file is specified for backend servers to send their debugging output then this output will appear on the controlling tty of their parent postmaster.

-e

This option controls how dates are interpreted upon input to and output from the database. If the `-e` option is supplied, then dates passed to and from the frontend processes will be assumed to be in "European" format (`DD-MM-YYYY`), otherwise dates are assumed to be in "American" format (`MM-DD-YYYY`). Dates are accepted by the backend in a wide variety of formats, and for input dates this switch mostly affects the interpretation for ambiguous cases. See the *PostgreSQL User's Guide* for more information.

-o *OutputFile*

Sends all debugging and error output to *OutputFile*. If the backend is running under the postmaster, error messages are still sent to the frontend process as well as to *OutputFile*, but debugging output is sent to the controlling tty of the postmaster (since only one file descriptor can be sent to an actual file).

-s

Print time information and other statistics at the end of each query. This is useful for benchmarking or for use in tuning the number of buffers.

-v *protocol*

Specifies the number of the frontend/backend protocol to be used for this particular session.

There are several other options that may be specified, used mainly for debugging purposes. These are listed here only for the use by Postgres system developers. *Use of any of these options is highly discouraged.* Furthermore, any of these options may disappear or change at any time.

These special-case options are:

-A n|r|b|Q\fIn\fP|X\fIn\fP

This option generates a tremendous amount of output.

-L

Turns off the locking system.

-N

Disables use of newline as a query delimiter.

-f [s | i | m | n | h]

Forbids the use of particular scan and join methods: `s` and `i` disable sequential and index scans respectively, while `n`, `m`, and `h` disable nested-loop, merge and hash joins respectively.

Note

Neither sequential scans nor nested-loop joins can be disabled completely; the `-fs` and `-fn` options simply discourage the optimizer from using those plan types if it has any other alternative.

-i

Prevents query execution, but shows the plan tree.

-p *dbname*

Indicates to the backend server that it has been started by a postmaster and make different assumptions about buffer pool management, file descriptors, etc. Switches following -p are restricted to those considered "secure".

-t pa[rser] | pl[anner] | e[xecutor]

Print timing statistics for each query relating to each of the major system modules. This option cannot be used with `-s`.

Outputs

Of the nigh-infinite number of error messages you may see when you execute the backend server directly, the most common will probably be:

`semget: No space left on device`

If you see this message, you should run the ipcclean command. After doing this, try starting postmaster again. If this still doesn't work, you probably need to configure your kernel for shared memory and semaphores as described in the installation notes. If you have a kernel with particularly small shared memory and/or semaphore limits, you may have to reconfigure your kernel to increase its shared memory or semaphore parameters.

Tip

You may be able to postpone reconfiguring your kernel by decreasing -B to reduce Postgres' shared memory consumption.

Description

The Postgres backend server can be executed directly from the user shell. This should be done only while debugging by the DBA, and should not be done while other Postgres backends are being managed by a postmaster on this set of databases.

Some of the switches explained here can be passed to the backend through the "database options" field of a connection request, and thus can be set for a particular backend without going to the trouble of restarting the postmaster. This is particularly handy for debugging-related switches.

The optional argument *dbname* specifies the name of the database to be accessed. *dbname* defaults to the value of the `USER` environment variable.

Notes

Useful utilities for dealing with shared memory problems include ipcs(1), ipcrm(1), and ipcclean(1). See also postmaster .

Name

postmaster — Run the Postgres multi-user backend

Synopsis

<refsynopsisdivinfo>1999-07-20</refsynopsisdivinfo>

```
postmaster [ -B nBuffers ] [ -D DataDir ] [ -N maxBackends ] [ -S ]
     [ -d DebugLevel ] [ -i ] [ -l ]
     [ -o BackendOptions ] [ -p port ] [ -n | -s ]
```

Inputs

postmaster accepts the following command line arguments:

-B *nBuffers*

Sets the number of shared-memory disk buffers for the postmaster to allocate for use by the backend server processes that it starts. This value defaults to 64 buffers, where each buffer is 8k bytes (or whatever BLCKSZ is set to in src/include/config.h).

-D *DataDir*

Specifies the directory to use as the root of the tree of database directories. If -D is not given, the default data directory name is the value of the environment variable `PGDATA`. If `PGDATA` is not set, then the directory used is `$POSTGRESHOME/data`. If neither environment variable is set and this command-line option is not specified, the default directory that was set at compile-time is used.

-N *maxBackends*

Sets the maximum number of backend server processes that this postmaster is allowed to start. By default, this value is 32, but it can be set as high as 1024 if your system will support that many processes. (Note that -B is required to be at least twice -N, so you'll need to increase -B if you increase -N.) Both the default and upper limit values for -N can be altered when building Postgres (see src/include/config.h).

-S

Specifies that the postmaster process should start up in silent mode. That is, it will disassociate from the user's (controlling) tty, start its own process group, and redirect its standard output and standard error to `/dev/null`.

Note that using this switch makes it very difficult to troubleshoot problems, since all tracing and logging output that would normally be generated by this postmaster and its child backends will be discarded.

-d *DebugLevel*

Determines the amount of debugging output the backend servers will produce. If *DebugLevel* is one, the postmaster will trace all connection traffic. Levels two and higher turn on increasing amounts of debug output from the backend processes, and the postmaster displays more information including the backend environment and process traffic. Note that unless the postmaster's standard output and standard error are redirected into a log file, all this output will appear on the controlling tty of the postmaster.

-i

Allows clients to connect via TCP/IP (Internet domain) connections. Without this option, only local Unix domain socket connections are accepted.

-l

Enables secure connections using SSL. The `-i` option is also required. You must have compiled with SSL enabled to use this option.

-o *`BackendOptions`*

The `postgres` option(s) specified in *`BackendOptions`* are passed to all backend server processes started by this postmaster. If the option string contains any spaces, the entire string must be quoted.

-p *`port`*

Specifies the TCP/IP port or local Unix domain socket file extension on which the postmaster is to listen for connections from frontend applications. Defaults to the value of the `PGPORT` environment variable, or if `PGPORT` is not set, then defaults to the value established when Postgres was compiled (normally 5432). If you specify a port other than the default port then all frontend applications (including psql) must specify the same port using either command-line options or `PGPORT`.

Two additional command line options are available for debugging problems that cause a backend to die abnormally. These options control the behavior of the postmaster in this situation, and *neither option is intended for use in ordinary operation*.

The ordinary strategy for this situation is to notify all other backends that they must terminate and then reinitialize the shared memory and semaphores. This is because an errant backend could have corrupted some shared state before terminating.

These special-case options are:

-n

postmaster will not reinitialize shared data structures. A knowledgeable system programmer can then use a debugger to examine shared memory and semaphore state.

-s

postmaster will stop all other backend processes by sending the signal `SIGSTOP`, but will not cause them to terminate. This permits system programmers to collect core dumps from all backend processes by hand.

Outputs

`semget: No space left on device`

If you see this message, you should run the ipcclean command. After doing so, try starting postmaster again. If this still doesn't work, you probably need to configure your kernel for shared memory and semaphores as described in the installation notes. If you run multiple instances of postmaster on a single host, or have a kernel with particularly small shared memory and/or semaphore limits, you may have to reconfigure your kernel to increase its shared memory or semaphore parameters.

Tip

You may be able to postpone reconfiguring your kernel by decreasing -B to reduce Postgres' shared memory consumption, and/or by reducing -N to reduce Postgres' semaphore consumption.

`StreamServerPort: cannot bind to port`

If you see this message, you should make certain that there is no other postmaster process already running on the same port number. The easiest way to determine this is by using the command

```
% ps -ax | grep postmaster
```

on BSD-based systems, or

```
% ps -e | grep postmast
```

for System V-like or POSIX-compliant systems such as HP-UX.

If you are sure that no other postmaster processes are running and you still get this error, try specifying a different port using the `-p` option. You may also get this error if you terminate the postmaster and immediately restart it using the same port; in this case, you must simply wait a few seconds until the operating system closes the port before trying again. Finally, you may get this error if you specify a port number that your operating system considers to be reserved. For example, many versions of Unix consider port numbers under 1024 to be *trusted* and only permit the Unix superuser to access them.

`IpcMemoryAttach: shmat() failed: Permission denied`

A likely explanation is that another user attempted to start a postmaster process on the same port which acquired shared resources and then died. Since Postgres shared memory keys are based on the port number assigned to the postmaster, such conflicts are likely if there is more than one installation on a single host. If there are no other postmaster processes currently running (see above), run ipcclean and try again. If other postmaster images are running, you will have to find the owners of those processes to coordinate the assignment of port numbers and/or removal of unused shared memory segments.

Description

postmaster manages the communication between frontend and backend processes, as well as allocating the shared buffer pool and SysV semaphores (on machines without a test-and-set instruction). postmaster does not itself interact with the user and should be started as a background process.

Only one postmaster should be running at a time in a given Postgres installation. Here, an installation means a database directory and postmaster port number. You can run more than one postmaster on a machine only if each one has a separate directory and port number.

Notes

If at all possible, *do not* use `SIGKILL` when killing the postmaster. `SIGHUP`, `SIGINT`, or `SIGTERM` (the default signal for kill(1))" should be used instead. Using

```
% kill -KILL
```

or its alternative form

```
% kill -9
```

will prevent postmaster from freeing the system resources (e.g., shared memory and semaphores) that it holds before dying. Use `SIGTERM` instead to avoid having to clean up manually (as described earlier).

Useful utilities for dealing with shared memory problems include ipcs(1), ipcrm(1), and ipcclean(1).

Usage

To start postmaster using default values, type:

```
% nohup postmaster >logfile 2>&1 &
```

This command will start up postmaster on the default port (5432). This is the simplest and most common way to start the postmaster.

To start postmaster with a specific port:

```
% nohup postmaster -p 1234 &
```

This command will start up postmaster communicating through the port 1234. In order to connect to this postmaster using psql, you would need to run it as

```
% psql -p 1234
```

or set the environment variable `PGPORT`:

```
% setenv PGPORT 1234
% psql
```

Part II. Administrator's Guide

Installation and maintenance information.

Table of Contents

22. Ports 315
    22.1. Currently Supported Platforms 315
    22.2. Unsupported Platforms 316
23. Configuration Options 318
    23.1. Parameters for Configuration (configure) 318
    23.2. Parameters for Building (make) 319
    23.3. Locale Support 320
        23.3.1. What are the Benefits? 321
        23.3.2. What are the Drawbacks? 321
    23.4. Kerberos Authentication 321
        23.4.1. Availability 321
        23.4.2. Installation 322
        23.4.3. Operation 322
24. System Layout 323
25. Installation 325
    25.1. Before you start 325
    25.2. Installation Procedure 325
26. Installation on Win32 332
    26.1. Building the libraries 332
    26.2. Installing the libraries 332
    26.3. Using the libraries 332
27. Runtime Environment 333
    27.1. Using Postgres from Unix 333
    27.2. Starting postmaster 333
    27.3. Using pg_options 333
        27.3.1. Recognized Options 334
28. Security 338
    28.1. User Authentication 338
        28.1.1. Host-Based Access Control 338
    28.2. User Names and Groups 340
        28.2.1. Creating Users 340
        28.2.2. Creating Groups 340
        28.2.3. Assigning Users to Groups 341
    28.3. Access Control 341
    28.4. Functions and Rules 341
        28.4.1. Functions 341
        28.4.2. Rules 341
        28.4.3. Caveats 341
    28.5. Secure TCP/IP Connection 342
29. Adding and Deleting Users 343
30. Disk Management 344
    30.1. Alternate Locations 344
31. Managing a Database 346
    31.1. Creating a Database 346
    31.2. Accessing a Database 346
    31.3. Destroying a Database 347
    31.4. Backup and Restore 347
        31.4.1. Large Databases 348
32. Troubleshooting 349
    32.1. Postmaster Startup Failures 349
    32.2. Client Connection Problems 349
    32.3. Debugging Messages 350
        32.3.1. pg_options 351
33. Database Recovery 353
34. Regression Test 354

34.1. Regression Environment 354
34.2. Regression Test Procedure 355
34.3. Regression Analysis 356
34.3.1. Error message differences 356
34.3.2. Date and time differences 357
34.3.3. Floating point differences 357
34.3.4. Polygon differences 357
34.3.5. Random differences 357
34.3.6. The "expected" files 358
34.4. Platform-specific comparison files 358
35. Release Notes 359
35.1. Release 7.0.3 359
35.1.1. Migration to v7.0.3 359
35.1.2. Changes 359

Chapter 22. Ports

This manual describes version 7.0 of Postgres. The Postgres developer community has compiled and tested Postgres on a number of platforms. Check the web site[1] for the latest information.

22.1. Currently Supported Platforms

At the time of publication, the following platforms have been tested:

Table 22.1. Supported Platforms

OS	Processor	Version	Reported	Remarks
AIX 4.3.2	RS6000	v7.0	2000-04-05	Andreas Zeugswetter[2]
BSDI 4.01	x86	v7.0	2000-04-04	Bruce Momjian[3]
Compaq Tru64 5.0	Alpha	v7.0	2000-04-11	Andrew McMurry[4]
FreeBSD 4.0	x86	v7.0	2000-04-04	Marc Fournier[5]
HPUX	PA-RISC	v7.0	2000-04-12	Both 9.0x and 10.20. Tom Lane[6]
IRIX 6.5.6f	MIPS	v6.5.3	2000-02-18	MIPSPro 7.3.1.1m N32 build. Kevin Wheatley[7]
Linux 2.0.x	Alpha	v7.0	2000-04-05	With published patches. Ryan Kirkpatrick[8]
Linux 2.2.x	armv4l	v7.0	2000-04-17	Regression test needs work. Mark Knox[9]
Linux 2.2.x	x86	v7.0	2000-03-26	Lamar Owens[10]
Linux 2.0.x	MIPS	v7.0	2000-04-13	Cobalt Qube. Tatsuo Ishii[11]
Linux 2.2.5	Sparc	v7.0	2000-04-02	Tom Szybist[12]
LinuxPPC R4	PPC603e	v7.0	2000-04-13	Tatsuo Ishii[13]
mklinux	PPC750	v7.0	2000-04-13	Tatsuo Ishii[14]
NetBSD 1.4	arm32	v7.0	2000-04-08	Patrick Welche[15]
NetBSD 1.4U	x86	v7.0	2000-03-26	Patrick Welche[16]

[1] http://www.postgresql.org/docs/admin/ports.htm
[2] mailto:Andreas.Zeugswetter@telecom.at
[3] mailto:maillist@candle.pha.pa.us
[4] mailto:andrew.mcmurry@astro.uio.no
[5] mailto:scrappy@hub.org
[6] mailto:tgl@sss.pgh.pa.us
[7] mailto:hxpro@cinesite.co.uk
[8] mailto:pgsql@rkirkpat.net
[9] mailto:segfault@hardline.org
[10] mailto:lamar.owen@wgcr.org
[11] mailto:t-ishii@sra.co.jp
[12] mailto:szybist@boxhill.com
[13] mailto:t-ishii@sra.co.jp
[14] mailto:t-ishii@sra.co.jp
[15] mailto:prlw1@newn.cam.ac.uk
[16] mailto:prlw1@newn.cam.ac.uk

OS	Processor	Version	Reported	Remarks
NetBSD	m68k	v7.0	2000-04-10	Mac 8xx. Henry B. Hotz[17]
NetBSD/sparc	Sparc	v7.0	2000-04-13	Tom I Helbekkmo[18]
QNX 4.25	x86	v7.0	2000-04-01	Dr. Andreas Kardos[19]
SCO OpenServer 5	x86	v6.5	1999-05-25	Andrew Merrill[20]
SCO UnixWare 7	x86	v7.0	2000-04-18	See FAQ. Billy G. Allie[21]
Solaris	x86	v7.0	2000-04-12	Marc Fournier[22]
Solaris 2.5.1-2.7	Sparc	v7.0	2000-04-12	Peter Eisentraut[23], Marc Fournier[24]
SunOS 4.1.4	Sparc	v7.0	2000-04-13	Tatsuo Ishii[25]
Windows/Win32	x86	v7.0	2000-04-02	Client-side libraries or ODBC/JDBC. No server-side. Magnus Hagander[26]
WinNT/Cygwin	x86	v7.0	2000-03-30	Uses Cygwin library. Daniel Horak[27]

Note

For Windows NT, the server-side port of Postgres uses the RedHat/Cygnus Cygwin library and toolset. For Windows 9x, no server-side port is available due to OS limitations.

22.2. Unsupported Platforms

Platforms listed for v6.3.x-v6.5.x should also work with v7.0, but we did not receive explicit confirmation of such at the time this list was compiled. We include these here to let you know that these platforms *could* be supported if given some attention.

At the time of publication, the following platforms have not been tested for v7.0 or v6.5.x:

Table 22.2. Unsupported Platforms

OS	Processor	Version	Reported	Remarks
BeOS	x86	v7.0	2000-05-01	Client-side coming soon? Adam Haberlach[28]

[17] mailto:hotz@jpl.nasa.gov
[18] mailto:tih@kpnQwest.no
[19] mailto:kardos@repas-aeg.de
[20] mailto:andrew@compclass.com
[21] mailto:Bill.Allie@mug.org
[22] mailto:scrappy@hub.org
[23] mailto:peter_e@gmx.net
[24] mailto:scrappy@hub.org
[25] mailto:t-ishii@sra.co.jp
[26] mha@sollentuna.net
[27] mailto:horak@sit.plzen-city.cz
[28] mailto:adam@newsnipple.com

OS	Processor	Version	Reported	Remarks
DGUX 5.4R4.11	m88k	v6.3	1998-03-01	v6.4 probably OK. Needs new maintainer. Brian E Gallew[29]
NetBSD-current	NS32532	v6.4	1998-10-27	Date math annoyances. Jon Buller[30]
NetBSD 1.3	VAX	v6.3	1998-03-01	v7.0 should work. Tom I Helbekkmo[31]
SVR4 4.4	m88k	v6.2.1	1998-03-01	v6.4.x will need TAS spinlock code. Doug Winterburn[32]
SVR4	MIPS	v6.4	1998-10-28	No 64-bit int. Frank Ridderbusch[33]
Ultrix	MIPS, VAX	v6.x	1998-03-01	No recent reports; obsolete?

There are a few platforms which have been attempted and which have been reported to not work with the standard distribution. Others listed here do not provide sufficient library support for an attempt.

Table 22.3. Incompatible Platforms

OS	Processor	Version	Reported	Remarks
MacOS	all	v6.x	1998-03-01	Not library compatible; use ODBC/JDBC
NextStep	x86	v6.x	1998-03-01	Client-only support; v1.0.9 worked with patches David Wetzel[34]

[29] mailto:geek+@cmu.edu
[30] mailto:jonb@metronet.com
[31] mailto:tih@kpnQwest.no
[32] mailto:dlw@seavme.xroads.com
[33] mailto:ridderbusch.pad@sni.de
[34] mailto:dave@turbocat.de

Chapter 23. Configuration Options

23.1. Parameters for Configuration (configure)

The full set of parameters available in configure can be obtained by typing

```
$ ./configure --help
```

The following parameters may be of interest to installers:

```
Directories to install PostgreSQL in:
  --prefix=PREFIX         install architecture-independent files in
 PREFIX
                          [/usr/local/pgsql]
  --bindir=DIR            user executables in DIR [EPREFIX/bin]
  --libdir=DIR            object code libraries in DIR [EPREFIX/
lib]
  --includedir=DIR        C header files in DIR [PREFIX/include]
  --mandir=DIR            man documentation in DIR [PREFIX/man]
Features and packages:
  --disable-FEATURE       do not include FEATURE (same as --enable-
FEATURE=no)
  --enable-FEATURE[=ARG]  include FEATURE [ARG=yes]
  --with-PACKAGE[=ARG]    use PACKAGE [ARG=yes]
  --without-PACKAGE       do not use PACKAGE (same as --with-
PACKAGE=no)
--enable and --with options recognized:
  --with-template=template
                          use operating system template file
                              see template directory
  --with-includes=dirs    look for header files for tcl/tk, etc in
 DIRS
  --with-libraries=dirs   look for additional libraries in DIRS
  --with-libs=dirs        alternate spelling of --with-libraries
  --enable-locale         enable locale support
  --enable-recode         enable cyrillic recode support
  --enable-multibyte      enable multibyte character support
  --with-pgport=portnum   change default postmaster port
  --with-maxbackends=n    set default maximum number of server
 processes
  --with-tcl              build Tcl interfaces and pgtclsh
  --with-tclconfig=tcldir
                          tclConfig.sh and tkConfig.sh are in DIR
  --with-perl             build Perl interface and plperl
  --with-odbc             build ODBC driver package
  --with-odbcinst=odbcdir
                          change default directory for odbcinst.ini
  --enable-cassert        enable assertion checks (for debugging)
  --enable-debug          build with debugging symbols (-g)
  --with-CC=compiler
                          use specific C compiler
  --with-CXX=compiler
                          use specific C++ compiler
```

```
  --without-CXX           prevent building C++ code
```

Some systems may have trouble building a specific feature of Postgres. For example, systems with a damaged C++ compiler may need to specify `--without-CXX` to instruct the build procedure to skip construction of `libpq++`.

Use the `--with-includes` and `--with-libraries` options if you want to build Postgres using include files or libraries that are not installed in your system's standard search path. For example, you might use these to build with an experimental version of Tcl. If you need to specify more than one nonstandard directory for include files or libraries, do it like this:

```
--with-includes="/opt/tcl/include /opt/perl5/include"
```

23.2. Parameters for Building (make)

Many installation-related parameters can be set in the building stage of Postgres installation.

In most cases, these parameters should be placed in a file, `Makefile.custom`, intended just for that purpose. The default distribution does not contain this optional file, so you will create it using a text editor of your choice. When upgrading installations, you can simply copy your old Makefile.custom to the new installation before doing the build.

Alternatively, you can set variables on the make command line:

```
make [ variable=value [...] ]
```

A few of the many variables that can be specified are:

`POSTGRESDIR`

Top of the installation tree.

`BINDIR`

Location of applications and utilities.

`LIBDIR`

Location of object libraries, including shared libraries.

`HEADERDIR`

Location of include files.

`ODBCINST`

Location of installation-wide psqlODBC (ODBC) configuration file.

There are other optional parameters which are not as commonly used. Many of those listed below are appropriate when doing Postgres server code development.

`CFLAGS`

Set flags for the C compiler. Should be assigned with "+=" to retain relevant default parameters.

`YFLAGS`

Set flags for the yacc/bison parser. `-v` might be used to help diagnose problems building a new parser. Should be assigned with "+=" to retain relevant default parameters.

`USE_TCL`

Enable Tcl interface building.

`HSTYLE`

DocBook HTML style sheets for building the documentation from scratch. Not used unless you are developing new documentation from the DocBook-compatible SGML source documents in `doc/src/sgml/`.

`PSTYLE`

DocBook style sheets for building printed documentation from scratch. Not used unless you are developing new documentation from the DocBook-compatible SGML source documents in `doc/src/sgml/`.

Here is an example `Makefile.custom` for a PentiumPro Linux system:

```
# Makefile.custom
# Thomas Lockhart 1999-06-01

POSTGRESDIR= /opt/postgres/current
CFLAGS+= -m486 -O2

# documentation

HSTYLE= /home/tgl/SGML/db118.d/docbook/html
PSTYLE= /home/tgl/SGML/db118.d/docbook/print
```

23.3. Locale Support

Note

Written by Oleg Bartunov. See Oleg's web page[1] for additional information on locale and Russian language support.

While doing a project for a company in Moscow, Russia, I encountered the problem that postgresql had no support of national alphabets. After looking for possible workarounds I decided to develop support of locale myself. I'm not a C-programer but already had some experience with locale programming when I work with perl (debugging) and glimpse. After several days of digging through the Postgres source tree I made very minor corections to src/backend/utils/adt/varlena.c and src/backend/main/main.c and got what I needed! I did support only for `LC_CTYPE` and `LC_COLLATE`, but later `LC_MONETARY` was added by others. I got many messages from people about this patch so I decided to send it to developers and (to my surprise) it was incorporated into the Postgres distribution.

People often complain that locale doesn't work for them. There are several common mistakes:

- Didn't properly configure postgresql before compilation. You must run configure with --enable-locale option to enable locale support. Didn't setup environment correctly when starting postmaster. You must define environment variables `LC_CTYPE` and `LC_COLLATE` before running postmaster because backend gets information about locale from environment. I use following shell script (runpostgres):

```
#!/bin/sh

export LC_CTYPE=koi8-r
export LC_COLLATE=koi8-r
postmaster -B 1024 -S -D/usr/local/pgsql/data/ -o '-Fe'
```

[1] http://www.sai.msu.su/~megera/postgres/

and run it from rc.local as

```
/bin/su - postgres -c "/home/postgres/runpostgres"
```

- Broken locale support in OS (for example, locale support in libc under Linux several times has changed and this caused a lot of problems). Latest perl has also support of locale and if locale is broken `perl -v` will complain something like:

```
8:17[mira]:~/WWW/postgres>setenv LC_CTYPE not_exist
8:18[mira]:~/WWW/postgres>perl -v
perl: warning: Setting locale failed.
perl: warning: Please check that your locale settings:
LC_ALL = (unset),
    LC_CTYPE = "not_exist",
    LANG = (unset)
are supported and installed on your system.
perl: warning: Falling back to the standard locale ("C").
```

- Wrong location of locale files! Possible locations include: `/usr/lib/locale` (Linux, Solaris), `/usr/share/locale` (Linux), `/usr/lib/nls/loc` (DUX 4.0). Check `man locale` to find the correct location. Under Linux I did a symbolic link between `/usr/lib/locale` and `/usr/share/locale` to be sure that the next libc will not break my locale.

23.3.1. What are the Benefits?

You can use ~* and order by operators for strings contain characters from national alphabets. Non-english users definitely need that. If you won't use locale stuff just undefine the USE_LOCALE variable.

23.3.2. What are the Drawbacks?

There is one evident drawback of using locale - its speed! So, use locale only if you really need it.

23.4. Kerberos Authentication

Kerberos is an industry-standard secure authentication system suitable for distributed computing over a public network.

23.4.1. Availability

The Kerberos authentication system is not distributed with Postgres. Versions of Kerberos are typically available as optional software from operating system vendors. In addition, a source code distribution may be obtained through MIT Project Athena[2].

Note

You may wish to obtain the MIT version even if your vendor provides a version, since some vendor ports have been deliberately crippled or rendered non-interoperable with the MIT version.

Users located outside the United States of America and Canada are warned that distribution of the actual encryption code in Kerberos is restricted by U. S. Government export regulations.

[2] ftp://athena-dist.mit.edu

Inquiries regarding your Kerberos should be directed to your vendor or MIT Project Athena[3]. Note that FAQLs (Frequently-Asked Questions Lists) are periodically posted to the Kerberos mailing list[4] (send mail to subscribe[5]), and USENET news group[6].

23.4.2. Installation

Installation of Kerberos itself is covered in detail in the *Kerberos Installation Notes* . Make sure that the server key file (the `srvtab` or `keytab`) is somehow readable by the Postgres account.

Postgres and its clients can be compiled to use either Version 4 or Version 5 of the MIT Kerberos protocols by setting the `KRBVERS` variable in the file `src/Makefile.global` to the appropriate value. You can also change the location where Postgres expects to find the associated libraries, header files and its own server key file.

After compilation is complete, Postgres must be registered as a Kerberos service. See the *Kerberos Operations Notes* and related manual pages for more details on registering services.

23.4.3. Operation

After initial installation, Postgres should operate in all ways as a normal Kerberos service. For details on the use of authentication, see the *PostgreSQL User's Guide* reference sections for postmaster and psql.

In the Kerberos Version 5 hooks, the following assumptions are made about user and service naming:

- User principal names (anames) are assumed to contain the actual Unix/Postgres user name in the first component.
- The Postgres service is assumed to be have two components, the service name and a hostname, canonicalized as in Version 4 (i.e., with all domain suffixes removed).

Table 23.1. Kerberos Parameter Examples

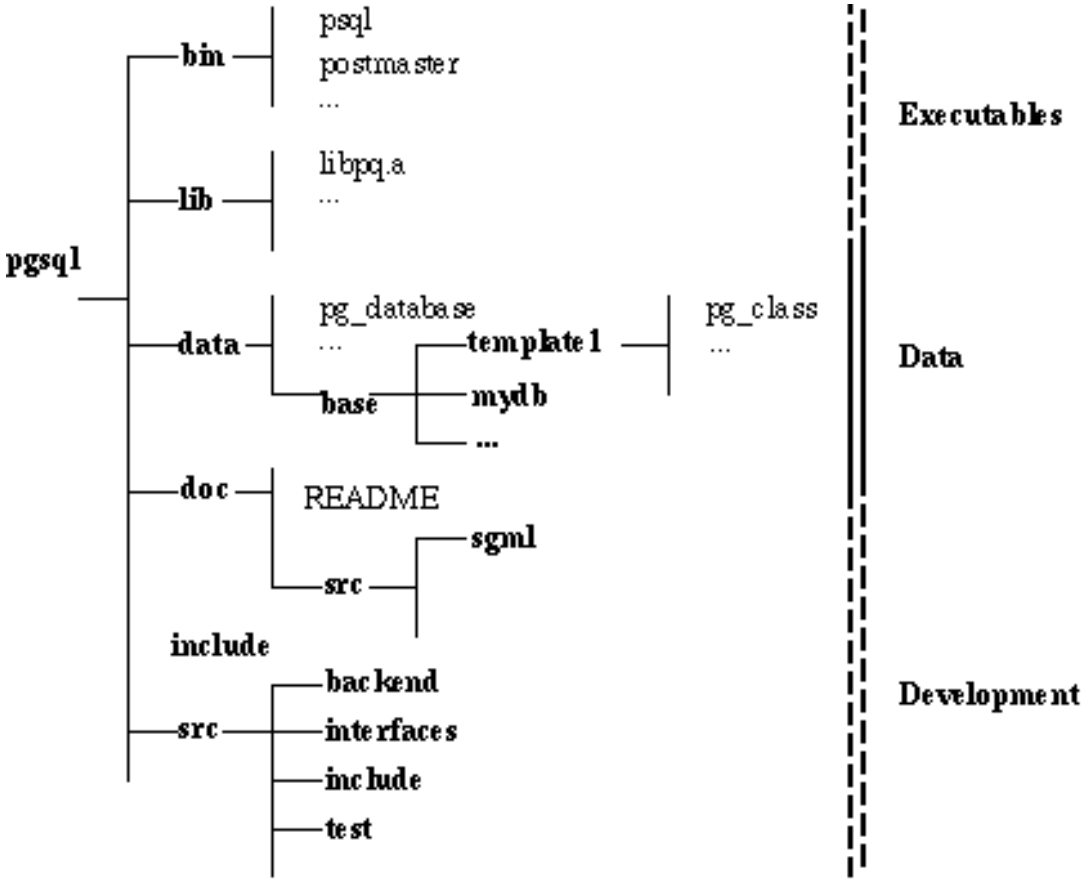
Parameter	Example
user	frew@S2K.ORG
user	aoki/ HOST=miyu.S2K.Berkeley.EDU@S2K.ORG
host	postgres_dbms/ucbvax@S2K.ORG

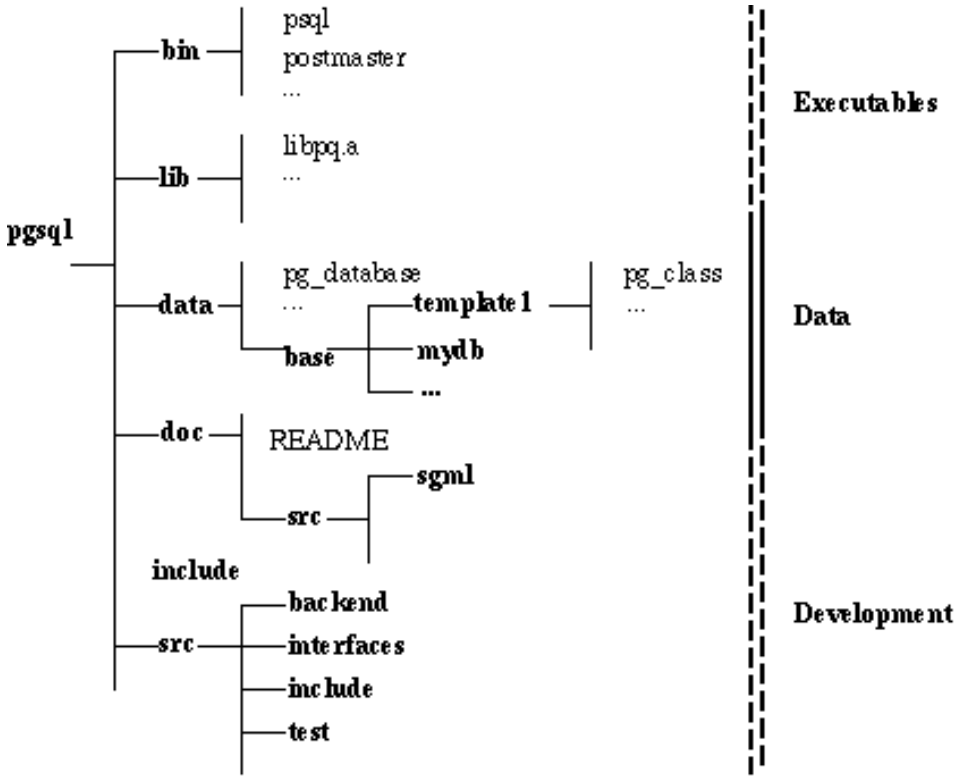
Support for Version 4 will disappear sometime after the production release of Version 5 by MIT.

[3] info-kerberos@athena.mit.edu
[4] mailto:kerberos@athena.mit.edu
[5] mailto:kerberos-request@athena.mit.edu
[6] news:comp.protocols.kerberos

Chapter 24. System Layout

Figure 24.1. Postgres file layout





shows how the Postgres distribution is laid out when installed in the default way. For simplicity, we will assume that Postgres has been installed in the directory `/usr/local/pgsql`. Therefore, wherever you see the directory `/usr/local/pgsql` you should substitute the name of the directory where Postgres is actually installed. All Postgres commands are installed in the directory `/usr/local/pgsql/bin`. Therefore, you should add this directory to your shell command path. If you use a variant of the Berkeley C shell, such as csh or tcsh, you would add

```
set path = ( /usr/local/pgsql/bin path )
```

in the .login file in your home directory. If you use a variant of the Bourne shell, such as sh, ksh, or bash, then you would add

```
PATH=/usr/local/pgsql/bin:$PATH
export PATH
```

to the .profile file in your home directory. From now on, we will assume that you have added the Postgres bin directory to your path. In addition, we will make frequent reference to "setting a shell variable" or "setting an environment variable" throughout this document. If you did not fully understand the last paragraph on modifying your search path, you should consult the Unix manual pages that describe your shell before going any further.

If you have not set things up in the default way, you may have some more work to do. For example, if the database server machine is a remote machine, you will need to set the `PGHOST` environment variable to the name of the database server machine. The environment variable `PGPORT` may also have to be set. The bottom line is this: if you try to start an application program and it complains that it cannot connect to the postmaster, you must go back and make sure that your environment is properly set up.

Chapter 25. Installation

Installation instructions for PostgreSQL 7.0.3.

If you haven't gotten the PostgreSQL distribution, get it from ftp.postgresql.org[1], then unpack it:

```
> gunzip postgresql-7.0.3.tar.gz
> tar -xf postgresql-7.0.3.tar
> mv postgresql-7.0.3 /usr/src
```

25.1. Before you start

Building PostgreSQL requires GNU make. It will *not* work with other make programs. On GNU/Linux systems GNU make is the default tool, on other systems you may find that GNU make is installed under the name `gmake`. We will use that name from now on to indicate GNU make, no matter what name it has on your system. To test for GNU make enter

```
> gmake --version
```

If you need to get GNU make, you can find it at ftp://ftp.gnu.org.

Up to date information on supported platforms is at http://www.postgresql.org/docs/admin/ports.htm[2]. In general, most Unix-compatible platforms with modern libraries should be able to run PostgreSQL. In the `doc` subdirectory of the distribution are several platform-specific FAQ and README documents you might wish to consult if you are having trouble.

Although the minimum required memory for running PostgreSQL can be as little as 8MB, there are noticeable speed improvements when expanding memory up to 96MB or beyond. The rule is you can never have too much memory.

Check that you have sufficient disk space. You will need about 30 Mbytes for the source tree during compilation and about 5 Mbytes for the installation directory. An empty database takes about 1 Mbyte, otherwise they take about five times the amount of space that a flat text file with the same data would take. If you run the regression tests you will temporarily need an extra 20MB.

To check for disk space, use

```
> df -k
```

Considering today's prices for hard disks, getting a large and fast hard disk should probably be in your plans before putting a database into production use.

25.2. Installation Procedure

PostgreSQL Installation

For a fresh install or upgrading from previous releases of PostgreSQL:

1. (Optional) Create the PostgreSQL superuser account. This is the user the server will run as. For production use you should create a separate, unprivileged account (`postgres` is commonly used). If you do not have root access or just want to play around, your own user account is enough.

   Running PostgreSQL as `root`, `bin`, or any other account with special access rights is a security risk; *don't do it*. The postmaster will in fact refuse to start as `root`.

[1] ftp://ftp.postgresql.org
[2] http://www.postgresql.org/docs/admin/ports.htm

You need not do the building and installation itself under this account (although you can). You will be told when you need to login as the database superuser.

2. Configure the source code for your system. It is this step at which you can specify your actual installation path for the build process and make choices about what gets installed. Change into the `src` subdirectory and type:

```
> ./configure
```

followed by any options you might want to give it. For a first installation you should be able to do fine without any. For a complete list of options, type:

```
> ./configure --help
```

Some of the more commonly used ones are:

--prefix=BASEDIR

Selects a different base directory for the installation of PostgreSQL. The default is `/usr/local/pgsql`.

--enable-locale

If you want to use locales.

--enable-multibyte

Allows the use of multibyte character encodings. This is primarily for languages like Japanese, Korean, or Chinese.

--with-perl

Builds the Perl interface and plperl extension language. Please note that the Perl interface needs to be installed into the usual place for Perl modules (typically under `/usr/lib/perl`), so you must have root access to perform the installation step. (It is often easiest to leave out `--with-perl` initially, and then build and install the Perl interface after completing the installation of PostgreSQL itself.)

--with-odbc

Builds the ODBC driver package.

--with-tcl

Builds interface libraries and programs requiring Tcl/Tk, including libpgtcl, pgtclsh, and pgtksh.

3. Compile the program. Type

```
> gmake
```

The compilation process can take anywhere from 10 minutes to an hour. Your mileage will most certainly vary. Remember to use GNU make.

The last line displayed will hopefully be

```
All of PostgreSQL is successfully made. Ready to install.
```

4. (Optional) If you want to test the newly built server before you install it, you can run the regression tests at this point. The regression tests are a test suite to verify that PostgreSQL runs on your machine in the way the developers expected it to. For detailed instructions see Regression Test. (Be sure to use the "parallel regress test" method, since the sequential method only works with an already-installed server.)

5. If you are not upgrading an existing system, skip to Step 7.

   If you are running 7.*, skip to step Step 6.

   You now need to back up your existing database. To dump your database installation, type:

   ```
   > pg_dumpall > db.out
   ```

   If you wish to preserve object id's (oids), then use the -o option when running pg_dumpall. However, unless you have a special reason for doing this (such as using OIDs as keys in tables), don't do it.

   Make sure to use the pg_dumpall command from the version you are currently running. 7.0.3's pg_dumpall should not be used on older databases.

   ## Caution

   You must make sure that your database is not updated in the middle of your backup. If necessary, bring down postmaster, edit the permissions in file `/usr/local/pgsql/data/pg_hba.conf` to allow only you on, then bring postmaster back up.

   Rather than using pg_dumpall, pg_upgrade can often be used.

6. If you are upgrading an existing system, kill the database server now. Type

   ```
   > ps ax | grep postmaster
   ```

   or

   ```
   > ps -e | grep postmaster
   ```

   (It depends on your system which one of these two works. No harm can be done by typing the wrong one.) This should list the process numbers for a number of processes, similar to this:

   ```
   263  ?  SW   0:00 (postmaster)
   777  p1 S    0:00 grep postmaster
   ```

   Type the following line, with *`pid`* replaced by the process id for process `postmaster` (263 in the above case). (Do not use the id for the process "grep postmaster".)

   ```
   > kill pid
   ```

   ## Tip

   On systems which have PostgreSQL started at boot time, there is probably a startup file that will accomplish the same thing. For example, on a Redhat Linux system one might find that

   ```
   > /etc/rc.d/init.d/postgres.init stop
   ```

works.

If you used pg_dumpall, move the old directory out of the way. Type the following:

```
> mv /usr/local/pgsql /usr/local/pgsql.old
```

(substitute your particular paths).

7. Install the PostgreSQL executable files and libraries. Type

   ```
   > gmake install
   ```

   You should do this step as the user that you want the installed executables to be owned by. This does not have to be the same as the database superuser; some people prefer to have the installed files be owned by root.

8. If necessary, tell your system how to find the new shared libraries. How to do this varies between platforms. The most widely usable method is to set the environment variable `LD_LIBRARY_PATH`:

   ```
   > LD_LIBRARY_PATH=/usr/local/pgsql/lib
   > export LD_LIBRARY_PATH
   ```

   on sh, ksh, bash, zsh or

   ```
   > setenv LD_LIBRARY_PATH /usr/local/pgsql/lib
   ```

   on csh or tcsh. You might want to put this into a shell startup file such as `/etc/profile`.

   On some systems the following is the preferred method, but you must have root access. Edit file `/etc/ld.so.conf` to add a line

   ```
   /usr/local/pgsql/lib
   ```

   Then run command `/sbin/ldconfig`.

   If in doubt, refer to the manual pages of your system. If you later on get a message like

   ```
   psql: error in loading shared libraries
   libpq.so.2.1: cannot open shared object file: No such file or
    directory
   ```

   then the above was necessary. Simply do this step then.

9. If you moved the old directory out of the way, create the database installation (the working data files). To do this you must log in to your PostgreSQL superuser account. It will not work as root.

   ```
   > mkdir /usr/local/pgsql/data
   > chown postgres /usr/local/pgsql/data
   > su - postgres
   > /usr/local/pgsql/bin/initdb -D /usr/local/pgsql/data
   ```

   The `-D` option specifies the location where the data will be stored. You can use any path you want, it does not have to be under the installation directory. Just make sure that the superuser

account can write to the directory (or create it, if it doesn't already exist) before starting `initdb`. (If you have already been doing the installation up to now as the PostgreSQL superuser, you may have to log in as root temporarily to create the data directory underneath a root-owned directory.)

10. The previous step should have told you how to start up the database server. Do so now. The command should look something like

    ```
    > /usr/local/pgsql/bin/postmaster -D /usr/local/pgsql/data
    ```

    This will start the server in the foreground. To make it detach to the background, you can use the `-S` option, but then you won't see any log messages the server produces. A better way to put the server in the background is

    ```
    > nohup /usr/local/pgsql/bin/postmaster -D /usr/local/pgsql/
    data \
        </dev/null >>server.log 2>>1 &
    ```

11. (Optional) If you did a pg_dumpall, reload your data back in:

    ```
    > /usr/local/pgsql/bin/psql -d template1 -f db.out
    ```

    You also might want to copy over the old `pg_hba.conf` file and any other files you might have had set up for authentication, such as password files.

This concludes the installation proper. To make your life more productive and enjoyable you should look at the following optional steps and suggestions:

- Life will be more convenient if you set up some environment variables. First of all you probably want to include `/usr/local/pgsql/bin` (or equivalent) into your `PATH`. To do this, add the following to your shell startup file, such as `~/.bash_profile` (or `/etc/profile`, if you want it to affect every user):

  ```
  > PATH=$PATH:/usr/local/pgsql/bin
  ```

  Furthermore, if you set `PGDATA` in the environment of the PostgreSQL superuser, you can omit the `-D` for `postmaster` and `initdb`.

- You probably want to install the man and HTML documentation. Type

  ```
  > cd /usr/src/pgsql/postgresql-7.0.3/doc
  > gmake install
  ```

  This will install files under `/usr/local/pgsql/doc` and `/usr/local/pgsql/man`. To enable your system to find the man documentation, you need to add a line like the following to a shell startup file:

  ```
  > MANPATH=$MANPATH:/usr/local/pgsql/man
  ```

  The documentation is also available in Postscript format. If you have a Postscript printer, or have your machine already set up to accept Postscript files using a print filter, then to print the User's Guide simply type

  ```
  > cd /usr/local/pgsql/doc
  > gunzip -c user.ps.tz | lpr
  ```

Here is how you might do it if you have Ghostscript on your system and are writing to a laserjet printer.

```
> gunzip -c user.ps.gz \
    | gs -sDEVICE=laserjet -r300 -q -dNOPAUSE -sOutputFile=- \
    | lpr
```

Printer setups can vary wildly from system to system. If in doubt, consult your manuals or your local expert.

The Adminstrator's Guide should probably be your first reading if you are completely new to PostgreSQL, as it contains information about how to set up database users and authentication.

- Usually, you will want to modify your computer so that it will automatically start the database server whenever it boots. This is not required; the PostgreSQL server can be run successfully from non-privileged accounts without root intervention.

  Different systems have different conventions for starting up daemons at boot time, so you are advised to familiarize yourself with them. Most systems have a file `/etc/rc.local` or `/etc/rc.d/rc.local` which is almost certainly no bad place to put such a command. Whatever you do, postmaster must be run by the PostgreSQL superuser (`postgres`) *and not by root* or any other user. Therefore you probably always want to form your command lines along the lines of `su -c '...' postgres`.

  It might be advisable to keep a log of the server output. To start the server that way try:

  ```
  > nohup su -c 'postmaster -D /usr/local/pgsql/data > server.log
   2>&1' postgres &
  ```

  Here are a few more operating system specific suggestions.

  - Edit file rc.local on NetBSD or file rc2.d on SPARC Solaris 2.5.1 to contain the following single line:

    ```
    > su postgres -c "/usr/local/pgsql/bin/postmaster -S -D /usr/
    local/pgsql/data"
    ```

  - In FreeBSD 2.2-RELEASE edit /usr/local/etc/rc.d/pgsql.sh to contain the following lines and make it chmod 755 and chown root:bin.

    ```
    #!/bin/sh
    [ -x /usr/local/pgsql/bin/postmaster ] && {
        su -l pgsql -c 'exec /usr/local/pgsql/bin/postmaster
            -D/usr/local/pgsql/data
            -S -o -F > /usr/local/pgsql/errlog' &
        echo -n ' pgsql'
    }
    ```

    You may put the line breaks as shown above. The shell is smart enough to keep parsing beyond end-of-line if there is an expression unfinished. The exec saves one layer of shell under the postmaster process so the parent is init.

  - In RedHat Linux add a file `/etc/rc.d/init.d/postgres.init` which is based on the example in `contrib/linux/`. Then make a softlink to this file from `/etc/rc.d/rc5.d/S98postgres.init`.

- Run the regression tests against the installed server (using the sequential test method). If you didn't run the tests before installation, you should definitely do it now. For detailed instructions see Regression Test.

To start experimenting with Postgres, set up the paths as explained above and start the server. To create a database, type

```
> createdb testdb
```

Then enter

```
> psql testdb
```

to connect to that database. At the prompt you can enter SQL commands and start experimenting.

Chapter 26. Installation on Win32

Build and installation instructions for Postgres v6.4 client libraries on Win32.

26.1. Building the libraries

The makefiles included in Postgres are written for Microsoft Visual C++, and will probably not work with other systems. It should be possible to compile the libaries manually in other cases.

To build the libraries, change directory into the `src` directory, and type the commands

```
copy include\config.h.win32 include\config.h
nmake /f win32.mak
```

This assumes that you have Visual C++ in your path.

The following files will be built:

- `interfaces\libpq\Release\libpq.dll` - The dynamically linkable frontend library
- `interfaces\libpq\Release\libpqdll.lib` - Import library to link your program to libpq.dll
- `interfaces\libpq\Release\libpq.lib` - Static library version of the frontend library
- `bin\psql\Release\psql.exe` - The Postgresql interactive SQL monitor

26.2. Installing the libraries

The only part of the library to really be installed is the `libpq.dll` library. This file should in most cases be placed in the `WINNT\SYSTEM32` directory (or in `WINDOWS\SYSTEM` on a Windows 95/98 system). If this file is installed using a setup program, it should be installed with version checking using the VERSIONINFO resource included in the file, to ensure that a newer version of the library is not overwritten.

If you plan to do development using libpq on this machine, you will have to add the `src\include` and `src\interfaces\libpq` directories to the include path in your compilers settings.

26.3. Using the libraries

To use the libraries, you must add the `libpqdll.lib` file to your project (in Visual C++, just right-click on the project and chose to add it).

Once this is done, it should be possible to use the library just as you would on a Unix platform.

Chapter 27. Runtime Environment

This chapter outlines the interaction between Postgres and the operating system.

27.1. Using Postgres from Unix

All Postgres commands that are executed directly from a Unix shell are found in the directory `.../bin`. Including this directory in your search path will make executing the commands easier.

A collection of system catalogs exist at each site. These include a class (`pg_user`) that contains an instance for each valid Postgres user. The instance specifies a set of Postgres privileges, such as the ability to act as Postgres super-user, the ability to create/destroy databases, and the ability to update the system catalogs. A Unix user cannot do anything with Postgres until an appropriate instance is installed in this class. Further information on the system catalogs is available by running queries on the appropriate classes.

27.2. Starting postmaster

Nothing can happen to a database unless the postmaster process is running. As the site administrator, there are a number of things you should remember before starting the postmaster. These are discussed in the installation and configuration sections of this manual. However, if Postgres has been installed by following the installation instructions exactly as written, the following simple command is all you should need to start the postmaster:

```
% postmaster
```

The postmaster occasionally prints out messages which are often helpful during troubleshooting. If you wish to view debugging messages from the postmaster, you can start it with the -d option and redirect the output to the log file:

```
% postmaster -d > pm.log 2>&1 &
```

If you do not wish to see these messages, you can type

```
% postmaster -S
```

and the postmaster will be "S"ilent. No ampersand ("&") is required in this case, since the postmaster automatically detaches from the terminal when -S is specified.

27.3. Using pg_options

Note

Contributed by Massimo Dal Zotto[1]

The optional file `data/pg_options` contains runtime options used by the backend to control trace messages and other backend tunable parameters. The file is re-read by a backend when it receives a SIGHUP signal, making thus possible to change run-time options on the fly without needing to restart Postgres. The options specified in this file may be debugging flags used by the trace package (`backend/utils/misc/trace.c`) or numeric parameters which can be used by the backend to control its behaviour.

[1] mailto:dz@cs.unitn.it

All pg_options are initialized to zero at backend startup. New or modified options will be read by all new backends when they are started. To make effective any changes for all running backends we need to send a SIGHUP to the postmaster. The signal will be automatically sent to all the backends. We can also activate the changes only for a specific backend by sending the SIGHUP directly to it.

pg_options can also be specified with the `-T` switch of Postgres:

```
postgres options -T "verbose=2,query,hostlookup-"
```

The functions used for printing errors and debug messages can now make use of the *syslog(2)* facility. Message printed to stdout or stderr are prefixed by a timestamp containing also the backend pid:

```
#timestamp          #pid    #message
980127.17:52:14.173 [29271] StartTransactionCommand
980127.17:52:14.174 [29271] ProcessUtility: drop table t;
980127.17:52:14.186 [29271] SIIncNumEntries: table is 70% full
980127.17:52:14.186 [29286] Async_NotifyHandler
980127.17:52:14.186 [29286] Waking up sleeping backend process
980127.19:52:14.292 [29286] Async_NotifyFrontEnd
980127.19:52:14.413 [29286] Async_NotifyFrontEnd done
980127.19:52:14.466 [29286] Async_NotifyHandler done
```

This format improves readability of the logs and allows people to understand exactly which backend is doing what and at which time. It also makes easier to write simple awk or perl scripts which monitor the log to detect database errors or problem, or to compute transaction time statistics.

Messages printed to syslog use the log facility LOG_LOCAL0. The use of syslog can be controlled with the syslog pg_option. Unfortunately many functions call directly `printf()` to print their messages to stdout or stderr and this output can't be redirected to syslog or have timestamps in it. It would be advisable that all calls to printf would be replaced with the PRINTF macro and output to stderr be changed to use EPRINTF instead so that we can control all output in a uniform way.

The format of the `pg_options` file is as follows:

```
# comment
option=integer_value  # set value for option
option                # set option = 1
option+               # set option = 1
option-               # set option = 0
```

Note that *`keyword`* can also be an abbreviation of the option name defined in `backend/utils/misc/trace.c`.

Example 27.1. pg_options File

For example my pg_options file contains the following values:

```
verbose=2
query
hostlookup
showportnumber
```

27.3.1. Recognized Options

The options currently defined are:

all

Global trace flag. Allowed values are:

0

Trace messages enabled individually

1

Enable all trace messages

-1

Disable all trace messages

verbose

Verbosity flag. Allowed values are:

0

No messages. This is the default.

1

Print information messages.

2

Print more information messages.

query

Query trace flag. Allowed values are:

0

Don't print query.

1

Print a condensed query in one line.

4

Print the full query.

plan

Print query plan.

parse

Print parser output.

rewritten

Print rewritten query.

pretty_plan

Pretty-print query plan.

pretty_parse

Pretty-print parser output.

pretty_rewritten

Pretty-print rewritten query.

parserstats

Print parser statistics.

plannerstats

Print planner statistics.

executorstats

Print executor statistics.

shortlocks

Currently unused but needed to enable features in the future.

locks

Trace locks.

userlocks

Trace user locks.

spinlocks

Trace spin locks.

notify

Trace notify functions.

malloc

Currently unused.

palloc

Currently unused.

lock_debug_oidmin

Minimum relation oid traced by locks.

lock_debug_relid

oid, if not zero, of relation traced by locks.

lock_read_priority

Currently unused.

deadlock_timeout

Deadlock check timer.

syslog

syslog flag. Allowed values are:

0

Messages to stdout/stderr.

1

Messages to stdout/stderr and syslog.

2

Messages only to syslog.

hostlookup

Enable hostname lookup in ps_status.

showportnumber

Show port number in ps_status.

nofsync

Disable fsync on a per-backend basis.

Chapter 28. Security

Database security is addressed at several levels:

- Data base file protection. All files stored within the database are protected from reading by any account other than the Postgres superuser account.
- Connections from a client to the database server are, by default, allowed only via a local Unix socket, not via TCP/IP sockets. The backend must be started with the `-i` option to allow non-local clients to connect.
- Client connections can be restricted by IP address and/or user name via the `pg_hba.conf` file in `PG_DATA`.
- Client connections may be authenticated via other external packages.
- Each user in Postgres is assigned a username and (optionally) a password. By default, users do not have write access to databases they did not create.
- Users may be assigned to *groups*, and table access may be restricted based on group privileges.

28.1. User Authentication

Authentication is the process by which the backend server and postmaster ensure that the user requesting access to data is in fact who he/she claims to be. All users who invoke Postgres are checked against the contents of the `pg_user` class to ensure that they are authorized to do so. However, verification of the user's actual identity is performed in a variety of ways:

From the user shell

> A backend server started from a user shell notes the user's (effective) user-id before performing a `setuid` to the user-id of user *`postgres`*. The effective user-id is used as the basis for access control checks. No other authentication is conducted.

From the network

> If the Postgres system is built as distributed, access to the Internet TCP port of the postmaster process is available to anyone. The DBA configures the pg_hba.conf file in the PGDATA directory to specify what authentication system is to be used according to the host making the connection and which database it is connecting to. See *pg_hba.conf(5)* for a description of the authentication systems available. Of course, host-based authentication is not fool-proof in Unix, either. It is possible for determined intruders to also masquerade the origination host. Those security issues are beyond the scope of Postgres.

28.1.1. Host-Based Access Control

Host-based access control is the name for the basic controls PostgreSQL exercises on what clients are allowed to access a database and how the users on those clients must authenticate themselves.

Each database system contains a file named `pg_hba.conf`, in its `PGDATA` directory, which controls who can connect to each database.

Every client accessing a database *must* be covered by one of the entries in `pg_hba.conf`. Otherwise all attempted connections from that client will be rejected with a "User authentication failed" error message.

The general format of the `pg_hba.conf` file is of a set of records, one per line. Blank lines and lines beginning with a hash character ("#") are ignored. A record is made up of a number of fields which are separated by spaces and/or tabs.

Connections from clients can be made using Unix domain sockets or Internet domain sockets (ie. TCP/IP). Connections made using Unix domain sockets are controlled using records of the following format:

```
local database authentication method
```

where

database specifies the database that this record applies to. The value `all` specifies that it applies to all databases.
authentication method specifies the method a user must use to authenticate themselves when connecting to that database using Unix domain sockets. The different methods are described below.

Connections made using Internet domain sockets are controlled using records of the following format.

```
host database TCP/IP address TCP/IP mask authentication method
```

The *TCP/IP address* is logically anded to both the specified *TCP/IP mask* and the TCP/IP address of the connecting client. If the two resulting values are equal then the record is used for this connection. If a connection matches more than one record then the earliest one in the file is used. Both the *TCP/IP address* and the *TCP/IP mask* are specified in dotted decimal notation.

If a connection fails to match any record then the *reject* authentication method is applied (see below).

28.1.1.1. Authentication Methods

The following authentication methods are supported for both Unix and TCP/IP domain sockets:

trust

The connection is allowed unconditionally.

reject

The connection is rejected unconditionally.

crypt

The client is asked for a password for the user. This is sent encrypted (using *crypt(3)*) and compared against the password held in the `pg_shadow` table. If the passwords match, the connection is allowed.

password

The client is asked for a password for the user. This is sent in clear and compared against the password held in the `pg_shadow` table. If the passwords match, the connection is allowed. An optional password file may be specified after the `password` keyword which is used to match the supplied password rather than the pg_shadow table. See pg_passwd.

The following authentication methods are supported for TCP/IP domain sockets only:

krb4

Kerberos V4 is used to authenticate the user.

krb5

Kerberos V5 is used to authenticate the user.

ident

The ident server on the client is used to authenticate the user (RFC 1413). An optional map name may be specified after the `ident` keyword which allows ident user names to be mapped onto Postgres user names. Maps are held in the file `$PGDATA/pg_ident.conf`.

28.1.1.2. Examples

```
# Trust any connection via Unix domain sockets.
local trust
# Trust any connection via TCP/IP from this machine.
host all 127.0.0.1 255.255.255.255  trust
# We don't like this machine.
host all 192.168.0.10 255.255.255.0  reject
# This machine can't encrypt so we ask for passwords in clear.
host all 192.168.0.3 255.255.255.0  password
# The rest of this group of machines should provide encrypted
 passwords.
host all 192.168.0.0 255.255.255.0  crypt
```

28.2. User Names and Groups

To define a new user, run the createuser utility program.

To assign a user or set of users to a new group, one must define the group itself, and assign users to that group. In Postgres these steps are not currently supported with a `create group` command. Instead, the groups are defined by inserting appropriate values into the `pg_group` system table, and then using the `grant` command to assign privileges to the group.

28.2.1. Creating Users

28.2.2. Creating Groups

Currently, there is no easy interface to set up user groups. You have to explicitly insert/update the `pg_group table`. For example:

```
jolly=> insert into pg_group (groname, grosysid, grolist)
jolly=>     values ('posthackers', '1234', '{5443, 8261}');
INSERT 548224
jolly=> grant insert on foo to group posthackers;
CHANGE
jolly=>
```

The fields in `pg_group` are:

groname

The group name. This a name and should be purely alphanumeric. Do not include underscores or other punctuation.

grosysid

The group id. This is an int4. This should be unique for each group.

grolist

The list of pg_user id's that belong in the group. This is an int4[].

28.2.3. Assigning Users to Groups

28.3. Access Control

Postgres provides mechanisms to allow users to limit the access to their data that is provided to other users.

Database superusers

Database super-users (i.e., users who have `pg_user.usesuper` set) silently bypass all of the access controls described below with two exceptions: manual system catalog updates are not permitted if the user does not have `pg_user.usecatupd` set, and destruction of system catalogs (or modification of their schemas) is never allowed.

Access Privilege

The use of access privilege to limit reading, writing and setting of rules on classes is covered in *grant/revoke(l)*.

Class removal and schema modification

Commands that destroy or modify the structure of an existing class, such as `alter`, `drop table`, and `drop index`, only operate for the owner of the class. As mentioned above, these operations are *never* permitted on system catalogs.

28.4. Functions and Rules

Functions and rules allow users to insert code into the backend server that other users may execute without knowing it. Hence, both mechanisms permit users to *trojan horse* others with relative impunity. The only real protection is tight control over who can define functions (e.g., write to relations with SQL fields) and rules. Audit trails and alerters on `pg_class`, `pg_user` and `pg_group` are also recommended.

28.4.1. Functions

Functions written in any language except SQL run inside the backend server process with the permissions of the user *`postgres`* (the backend server runs with its real and effective user-id set to *`postgres`*. It is possible for users to change the server's internal data structures from inside of trusted functions. Hence, among many other things, such functions can circumvent any system access controls. This is an inherent problem with user-defined C functions.

28.4.2. Rules

Like SQL functions, rules always run with the identity and permissions of the user who invoked the backend server.

28.4.3. Caveats

There are no plans to explicitly support encrypted data inside of Postgres (though there is nothing to prevent users from encrypting data within user-defined functions). There are no plans to explicitly support encrypted network connections, either, pending a total rewrite of the frontend/backend protocol.

User names, group names and associated system identifiers (e.g., the contents of `pg_user.usesysid`) are assumed to be unique throughout a database. Unpredictable results may occur if they are not.

28.5. Secure TCP/IP Connection

Author

From e-mail by Gene Selkov, Jr.[1] written on 1999-09-08 in response to a question from Eric Marsden.

One can use ssh to encrypt the network connection between clients and a Postgres server. Done properly, this should lead to an adequately secure network connection.

The documentation for ssh provides most of the information to get started. Please refer to http://www.heimhardt.de/htdocs/ssh.html for better insight.

A step-by-step explanation can be done in just two steps.

Running a secure tunnel via ssh

A step-by-step explanation can be done in just two steps.

1. Establish a tunnel to the backend machine, like this:

   ```
   ssh -L 3333:wit.mcs.anl.gov:5432 postgres@wit.mcs.anl.gov
   ```

   The first number in the -L argument, 3333, is the port number of your end of the tunnel. The second number, 5432, is the remote end of the tunnel -- the port number your backend is using. The name or the address in between the port numbers belongs to the server machine, as does the last argument to ssh that also includes the optional user name. Without the user name, ssh will try the name you are currently logged on as on the client machine. You can use any user name the server machine will accept, not necessarily those related to postgres.

2. Now that you have a running ssh session, you can connect a postgres client to your local host at the port number you specified in the previous step. If it's psql, you will need another shell because the shell session you used in Step 1 is now occupied with ssh.

   ```
   psql -h localhost -p 3333 -d mpw
   ```

   Note that you have to specify the `-h` argument to cause your client to use the TCP socket instead of the Unix socket. You can omit the port argument if you chose 5432 as your end of the tunnel.

[1] selkovjr@mcs.anl.gov

Chapter 29. Adding and Deleting Users

createuser enables specific users to access Postgres. dropuser removes users and prevents them from accessing Postgres.

These commands only affect users with respect to Postgres; they have no effect on a user's other privileges or status with regards to the underlying operating system.

Chapter 30. Disk Management

30.1. Alternate Locations

It is possible to create a database in a location other than the default location for the installation. Remember that all database access actually occurs through the database backend, so that any location specified must be accessible by the backend.

Alternate database locations are created and referenced by an environment variable which gives the absolute path to the intended storage location. This environment variable must have been defined before the backend was started and must be writable by the postgres administrator account. Any valid environment variable name may be used to reference an alternate location, although using variable name with a prefix of PGDATA is recommended to avoid confusion and conflict with other variables.

Note

In previous versions of Postgres, it was also permissable to use an absolute path name to specify an alternate storage location. The environment variable style of specification is to be preferred since it allows the site administrator more flexibility in managing disk storage. If you prefer using absolute paths, you may do so by defining "ALLOW_ABSOLUTE_DBPATHS" and recompiling Postgres To do this, either add this line

```
#define ALLOW_ABSOLUTE_DBPATHS 1
```

to the file `src/include/config.h`, or by specifying

```
 CFLAGS+= -DALLOW_ABSOLUTE_DBPATHS
```

in your `Makefile.custom`.

Remember that database creation is actually performed by the database backend. Therefore, any environment variable specifying an alternate location must have been defined before the backend was started. To define an alternate location PGDATA2 pointing to `/home/postgres/data`, first type

```
% setenv PGDATA2 /home/postgres/data
```

to define the environment variable to be used with subsequent commands. Usually, you will want to define this variable in the Postgres superuser's `.profile` or `.cshrc` initialization file to ensure that it is defined upon system startup. Any environment variable can be used to reference alternate location, although it is preferred that the variables be prefixed with "PGDATA" to eliminate confusion and the possibility of conflicting with or overwriting other variables.

To create a data storage area in PGDATA2, ensure that `/home/postgres` already exists and is writable by the postgres administrator. Then from the command line, type

```
% setenv PGDATA2 /home/postgres/data
% initlocation $PGDATA2
Creating Postgres database system directory /home/postgres/data

Creating Postgres database system directory /home/postgres/data/
base
```

To test the new location, create a database test by typing

```
% createdb -D PGDATA2 test
% dropdb test
```

Chapter 31. Managing a Database

If the Postgres postmaster is up and running we can create some databases to experiment with. Here, we describe the basic commands for managing a database.

31.1. Creating a Database

Let's say you want to create a database named mydb. You can do this with the following command:

```
% createdb dbname
```

Postgres allows you to create any number of databases at a given site and you automatically become the database administrator of the database you just created. Database names must have an alphabetic first character and are limited to 31 characters in length. Not every user has authorization to become a database administrator. If Postgres refuses to create databases for you, then the site administrator needs to grant you permission to create databases. Consult your site administrator if this occurs.

31.2. Accessing a Database

Once you have constructed a database, you can access it by:

- running the Postgres terminal monitor program (psql) which allows you to interactively enter, edit, and execute SQL commands.
- writing a C program using the `libpq` subroutine library. This allows you to submit SQL commands from C and get answers and status messages back to your program. This interface is discussed further in the *PostgreSQL Programmer's Guide*.

You might want to start up psql, to try out the examples in this manual. It can be activated for the *dbname* database by typing the command:

```
psql dbname
```

You will be greeted with the following message:

```
Welcome to psql, the PostgreSQL interactive terminal.

Type:  \copyright for distribution terms
       \h for help with SQL commands
       \? for help on internal slash commands
       \g or terminate with semicolon to execute query
       \q to quit

dbname=>
```

This prompt indicates that the terminal monitor is listening to you and that you can type SQL queries into a workspace maintained by the terminal monitor. The psql program responds to escape codes that begin with the backslash character, "\". For example, you can get help on the syntax of various Postgres SQL commands by typing:

```
dbname=> \h
```

Once you have finished entering your queries into the workspace, you can pass the contents of the workspace to the Postgres server by typing:

```
dbname=> \g
```

This tells the server to process the query. If you terminate your query with a semicolon, the backslash-g is not necessary. psql will automatically process semicolon terminated queries. To read queries from a file, instead of entering them interactively, type:

```
dbname=> \i filename
```

To get out of psql and return to Unix, type

```
dbname=> \q
```

and psql will quit and return you to your command shell. (For more escape codes, type backslash-h at the monitor prompt.) White space (i.e., spaces, tabs and newlines) may be used freely in SQL queries. Single-line comments are denoted by two dashes ("`--`"). Everything after the dashes up to the end of the line is ignored. Multiple-line comments, and comments within a line, are denoted by "`/* ... */`", a convention borrowed from Ingres.

31.3. Destroying a Database

If you are the database administrator for the database mydb, you can destroy it using the following Unix command:

```
% dropdb dbname
```

This action physically removes all of the Unix files associated with the database and cannot be undone, so this should only be done with a great deal of forethought.

It is also possible to destroy a database from within an SQL session by using

```
> drop database dbname
```

31.4. Backup and Restore

Caution

Every database should be backed up on a regular basis. Since Postgres manages it's own files in the file system, it is *not advisable* to rely on system backups of your file system for your database backups; there is no guarantee that the files will be in a usable, consistant state after restoration.

Postgres provides two utilities to backup your system: pg_dump to backup individual databases and pg_dumpall to backup your installation in one step.

An individual database can be backed up using the following command:

```
% pg_dump dbname > dbname.pgdump
```

and can be restored using

```
cat dbname.pgdump | psql dbname
```

This technique can be used to move databases to new locations, and to rename existing databases.

31.4.1. Large Databases

Author

Written by Hannu Krosing[1] on 1999-06-19.

Since Postgres allows tables larger than the maximum file size on your system, it can be problematic to dump the table to a file, since the resulting file will likely be larger than the maximum size allowed by your system.

As pg_dump writes to stdout, you can just use standard *nix tools to work around this possible problem:

- Use compressed dumps:

  ```
  % pg_dump dbname | gzip > filename.dump.gz
  ```

  reload with

  ```
  % createdb dbname
  % gunzip -c filename.dump.gz | psql dbname
  ```

  or

  ```
  % cat filename.dump.gz | gunzip | psql dbname
  ```

- Use split:

  ```
  % pg_dump dbname | split -b 1m - filename.dump.
  ```

  reload with

  ```
  % createdb dbname
  % cat filename.dump.* | pgsql dbname
  ```

Of course, the name of the file (*filename*) and the content of the pg_dump output need not match the name of the database. Also, the restored database can have an arbitrary new name, so this mechanism is also suitable for renaming databases.

[1] hannu@trust.ee

Chapter 32. Troubleshooting

32.1. Postmaster Startup Failures

There are several common reasons for the postmaster to fail to start up. Check the postmaster's log file, or start it by hand (without redirecting standard output or standard error) to see what complaint messages appear. Some of the possible error messages are reasonably self-explanatory, but here are some that are not:

```
FATAL: StreamServerPort: bind() failed: Address already in use
        Is another postmaster already running on that port?
```

This usually means just what it suggests: you accidentally started a second postmaster on the same port where one is already running. However, if the kernel error message is not "Address already in use" or some variant of that wording, there may be a different problem. For example, trying to start a postmaster on a reserved port number may draw something like

```
$ postmaster -i -p 666
FATAL: StreamServerPort: bind() failed: Permission denied
        Is another postmaster already running on that port?
```

```
IpcMemoryCreate: shmget failed (Invalid argument) key=5440001,
 size=83918612, permission=600
FATAL 1:  ShmemCreate: cannot create region
```

A message like this probably means that your kernel's limit on the size of shared memory areas is smaller than the buffer area that Postgres is trying to create. (Or it could mean that you don't have SysV-style shared memory support configured into your kernel at all.) As a temporary workaround, you can try starting the postmaster with a smaller-than-normal number of buffers (-B switch). You will eventually want to reconfigure your kernel to increase the allowed shared memory size, however. You may see this message when trying to start multiple postmasters on the same machine, if their total space requests exceed the kernel limit.

```
IpcSemaphoreCreate: semget failed (No space left on device)
 key=5440026, num=16, permission=600
```

A message like this does *not* mean that you've run out of disk space; it means that your kernel's limit on the number of SysV semaphores is smaller than the number Postgres wants to create. As above, you may be able to work around the problem by starting the postmaster with a reduced number of backend processes (-N switch), but you'll eventually want to increase the kernel limit.

32.2. Client Connection Problems

Once you have a running postmaster, trying to connect to it with client applications can fail for a variety of reasons. The sample error messages shown here are for clients based on recent versions of libpq --- clients based on other interface libraries may produce other messages with more or less information.

```
connectDB() -- connect() failed: Connection refused
Is the postmaster running (with -i) at 'server.joe.com' and
 accepting connections on TCP/IP port '5432'?
```

This is the generic "I couldn't find a postmaster to talk to" failure. It looks like the above when TCP/IP communication is attempted, or like this when attempting Unix-socket communication to a local postmaster:

```
connectDB() -- connect() failed: No such file or directory
Is the postmaster running at 'localhost' and accepting connections
 on Unix socket '5432'?
```

The last line is useful in verifying that the client is trying to connect where it is supposed to. If there is in fact no postmaster running there, the kernel error message will typically be either "Connection refused" or "No such file or directory", as illustrated. (It is particularly important to realize that "Connection refused" in this context does *not* mean that the postmaster got your connection request and rejected it --- that case will produce a different message, as shown below.) Other error messages such as "Connection timed out" may indicate more fundamental problems, like lack of network connectivity.

```
No pg_hba.conf entry for host 123.123.123.123, user joeblow,
 database testdb
```

This is what you are most likely to get if you succeed in contacting a postmaster, but it doesn't want to talk to you. As the message suggests, the postmaster refused the connection request because it found no authorizing entry in its pg_hba.conf configuration file.

```
Password authentication failed for user 'joeblow'
```

Messages like this indicate that you contacted the postmaster, and it's willing to talk to you, but not until you pass the authorization method specified in the pg_hba.conf file. Check the password you're providing, or check your Kerberos or IDENT software if the complaint mentions one of those authentication types.

```
FATAL 1:  SetUserId: user 'joeblow' is not in 'pg_shadow'
```

This is another variant of authentication failure: no Postgres create_user command has been executed for the given username.

```
FATAL 1:  Database testdb does not exist in pg_database
```

There's no database by that name under the control of this postmaster. Note that if you don't specify a database name, it defaults to your Postgres username, which may or may not be the right thing.

```
NOTICE:  Unrecognized variable client_encoding
```

This isn't an error; in fact, it's quite harmless. You'll see this message at startup if you use a client compiled with MULTIBYTE support to connect to a server compiled without it. (The client is trying to tell the server what character set encoding it wants, but the server has no idea what it's talking about.) If the message bothers you, use a client compiled with the same options as the server.

32.3. Debugging Messages

The postmaster occasionally prints out messages which are often helpful during troubleshooting. If you wish to view debugging messages from the postmaster, you can start it with the -d option and redirect the output to the log file:

```
% postmaster -d > pm.log 2>&1 &
```

If you do not wish to see these messages, you can type

```
% postmaster -S
```

and the postmaster will be "S"ilent. No ampersand ("&") is required in this case, since the postmaster automatically detaches from the terminal when -S is specified.

32.3.1. pg_options

Note

Contributed by Massimo Dal Zotto[1]

The optional file `data/pg_options` contains runtime options used by the backend to control trace messages and other backend tunable parameters. What makes this file interesting is the fact that it is re-read by a backend when it receives a SIGHUP signal, making thus possible to change run-time options on the fly without needing to restart Postgres. The options specified in this file may be debugging flags used by the trace package (`backend/utils/misc/trace.c`) or numeric parameters which can be used by the backend to control its behaviour. New options and parameters must be defined in `backend/utils/misc/trace.c` and `backend/include/utils/trace.h`.

pg_options can also be specified with the `-T` switch of Postgres:

```
postgres options -T "verbose=2,query,hostlookup-"
```

The functions used for printing errors and debug messages can now make use of the *syslog(2)* facility. Message printed to stdout or stderr are prefixed by a timestamp containing also the backend pid:

```
#timestamp          #pid    #message
980127.17:52:14.173 [29271] StartTransactionCommand
980127.17:52:14.174 [29271] ProcessUtility: drop table t;
980127.17:52:14.186 [29271] SIIncNumEntries: table is 70% full
980127.17:52:14.186 [29286] Async_NotifyHandler
980127.17:52:14.186 [29286] Waking up sleeping backend process
980127.19:52:14.292 [29286] Async_NotifyFrontEnd
980127.19:52:14.413 [29286] Async_NotifyFrontEnd done
980127.19:52:14.466 [29286] Async_NotifyHandler done
```

This format improves readability of the logs and allows people to understand exactly which backend is doing what and at which time. It also makes easier to write simple awk or perl scripts which monitor the log to detect database errors or problem, or to compute transaction time statistics.

Messages printed to syslog use the log facility LOG_LOCAL0. The use of syslog can be controlled with the syslog pg_option. Unfortunately many functions call directly `printf()` to print their messages to stdout or stderr and this output can't be redirected to syslog or have timestamps in it. It would be advisable that all calls to printf would be replaced with the PRINTF macro and output to stderr be changed to use EPRINTF instead so that we can control all output in a uniform way.

The format of the `pg_options` file is as follows:

```
# comment
option=integer_value  # set value for option
option                # set option = 1
option+               # set option = 1
option-               # set option = 0
```

[1] mailto:dz@cs.unitn.it

Note that *keyword* can also be an abbreviation of the option name defined in `backend/utils/misc/trace.c`.

Refer to Using pg_options for a complete list of option keywords and possible values.

Chapter 33. Database Recovery

This section needs to be written. Volunteers?

Chapter 34. Regression Test

Regression test instructions and analysis.

The PostgreSQL regression tests are a comprehensive set of tests for the SQL implementation embedded in PostgreSQL. They test standard SQL operations as well as the extended capabilities of PostgreSQL.

There are two different ways in which the regression tests can be run: the "sequential" method and the "parallel" method. The sequential method runs each test script in turn, whereas the parallel method starts up multiple server processes to run groups of tests in parallel. Parallel testing gives confidence that interprocess communication and locking are working correctly. Another key difference is that the sequential test procedure uses an already-installed postmaster, whereas the parallel test procedure tests a system that has been built but not yet installed. (The parallel test script actually does an installation into a temporary directory and fires up a private postmaster therein.)

Some properly installed and fully functional PostgreSQL installations can "fail" some of these regression tests due to artifacts of floating point representation and time zone support. The tests are currently evaluated using a simple diff comparison against the outputs generated on a reference system, so the results are sensitive to small system differences. When a test is reported as "failed", always examine the differences between expected and actual results; you may well find that the differences are not significant.

The regression tests were originally developed by Jolly Chen and Andrew Yu, and were extensively revised/repackaged by Marc Fournier and Thomas Lockhart. From PostgreSQL v6.1 onward the regression tests are current for every official release.

34.1. Regression Environment

The regression testing notes below assume the following (except where noted):

- Commands are Unix-compatible. See note below.
- Defaults are used except where noted.
- User postgres is the Postgres superuser.
- The source path is /usr/src/pgsql (other paths are possible).
- The runtime path is /usr/local/pgsql (other paths are possible).

Normally, the regression tests should be run as the postgres user since the 'src/test/regress' directory and sub-directories are owned by the postgres user. If you run the regression test as another user the 'src/test/regress' directory tree must be writeable by that user.

It was formerly necessary to run the postmaster with system time zone set to PST, but this is no longer required. You can run the regression tests under your normal postmaster configuration. The test script will set the PGTZ environment variable to ensure that timezone-dependent tests produce the expected results. However, your system must provide library support for the PST8PDT time zone, or the timezone-dependent tests will fail. To verify that your machine does have this support, type the following:

```
setenv TZ PST8PDT
date
```

The "date" command above should have returned the current system time in the PST8PDT time zone. If the PST8PDT database is not available, then your system may have returned the time in GMT. If the PST8PDT time zone is not available, you can set the time zone rules explicitly:

```
setenv PGTZ PST8PDT7,M04.01.0,M10.05.03
```

The directory layout for the regression test area is:

Table 34.1. Directory Layout

Directory	Description
Directory	Description
input	Source files that are converted using `make all` into some of the `.sql` files in the `sql` subdirectory.
output	Source files that are converted using `make all` into `.out` files in the `expected` subdirectory.
sql	`.sql` files used to perform the regression tests.
expected	`.out` files that represent what we *expect* the results to look like.
results	`.out` files that contain what the results *actually* look like. Also used as temporary storage for table copy testing.
tmp_check	Temporary installation created by parallel testing script.

34.2. Regression Test Procedure

Commands were tested on RedHat Linux version 4.2 using the bash shell. Except where noted, they will probably work on most systems. Commands like `ps` and `tar` vary wildly on what options you should use on each platform. *Use common sense* before typing in these commands.

Postgres Regression Test

1. Prepare the files needed for the regression test with:

```
cd /usr/src/pgsql/src/test/regress
gmake clean
gmake all
```

   You can skip "gmake clean" if this is the first time you are running the tests.

   This step compiles a C program with PostgreSQL extension functions into a shared library. Localized SQL scripts and output-comparison files are also created for the tests that need them. The localization replaces macros in the source files with absolute pathnames and user names.

2. (Optional) If you intend to use the "sequential" test procedure, which tests an already-installed postmaster, be sure that the postmaster is running. If it isn't already running, start the postmaster in an available window by typing

```
postmaster
```

   or start the postmaster daemon running in the background by typing

```
cd
nohup postmaster > regress.log 2>&1 &
```

   The latter is probably preferable, since the regression test log will be quite lengthy (60K or so, in Postgres 7.0) and you might want to review it for clues if things go wrong.

Note

Do not run `postmaster` from the root account.

3. Run the regression tests. For a sequential test, type

```
cd /usr/src/pgsql/src/test/regress
gmake runtest
```

For a parallel test, type

```
cd /usr/src/pgsql/src/test/regress
gmake runcheck
```

The sequential test just runs the test scripts using your already-running postmaster. The parallel test will perform a complete installation of Postgres into a temporary directory, start a private postmaster therein, and then run the test scripts. Finally it will kill the private postmaster (but the temporary directory isn't removed automatically).

4. You should get on the screen (and also written to file ./regress.out) a series of statements stating which tests passed and which tests failed. Please note that it can be normal for some of the tests to "fail" due to platform-specific variations. See the next section for details on determining whether a "failure" is significant.

   Some of the tests, notably "numeric", can take a while, especially on slower platforms. Have patience.

5. After running the tests and examining the results, type

```
cd /usr/src/pgsql/src/test/regress
gmake clean
```

to recover the temporary disk space used by the tests. If you ran a sequential test, also type

```
dropdb regression
```

34.3. Regression Analysis

The actual outputs of the regression tests are in files in the `./results` directory. The test script uses diff to compare each output file against the reference outputs stored in the `./expected` directory. Any differences are saved for your inspection in `./regression.diffs`. (Or you can run diff yourself, if you prefer.)

The files might not compare exactly. The test script will report any difference as a "failure", but the difference might be due to small cross-system differences in error message wording, math library behavior, etc. "Failures" of this type do not indicate a problem with Postgres.

Thus, it is necessary to examine the actual differences for each "failed" test to determine whether there is really a problem. The following paragraphs attempt to provide some guidance in determining whether a difference is significant or not.

34.3.1. Error message differences

Some of the regression tests involve intentional invalid input values. Error messages can come from either the Postgres code or from the host platform system routines. In the latter case, the messages

may vary between platforms, but should reflect similar information. These differences in messages will result in a "failed" regression test which can be validated by inspection.

34.3.2. Date and time differences

Most of the date and time results are dependent on timezone environment. The reference files are generated for timezone PST8PDT (Berkeley, California) and there will be apparent failures if the tests are not run with that timezone setting. The regression test driver sets environment variable PGTZ to PST8PDT to ensure proper results.

Some of the queries in the "timestamp" test will fail if you run the test on the day of a daylight-savings time changeover, or the day before or after one. These queries assume that the intervals between midnight yesterday, midnight today and midnight tomorrow are exactly twenty-four hours ... which is wrong if daylight-savings time went into or out of effect meanwhile.

There appear to be some systems which do not accept the recommended syntax for explicitly setting the local time zone rules; you may need to use a different PGTZ setting on such machines.

Some systems using older timezone libraries fail to apply daylight-savings corrections to pre-1970 dates, causing pre-1970 PDT times to be displayed in PST instead. This will result in localized differences in the test results.

34.3.3. Floating point differences

Some of the tests involve computing 64-bit (`float8`) numbers from table columns. Differences in results involving mathematical functions of `float8` columns have been observed. The float8 and geometry tests are particularly prone to small differences across platforms. Human eyeball comparison is needed to determine the real significance of these differences which are usually 10 places to the right of the decimal point.

Some systems signal errors from pow() and exp() differently from the mechanism expected by the current Postgres code.

34.3.4. Polygon differences

Several of the tests involve operations on geographic date about the Oakland/Berkley CA street map. The map data is expressed as polygons whose vertices are represented as pairs of `float8` numbers (decimal latitude and longitude). Initially, some tables are created and loaded with geographic data, then some views are created which join two tables using the polygon intersection operator (##), then a select is done on the view. When comparing the results from different platforms, differences occur in the 2nd or 3rd place to the right of the decimal point. The SQL statements where these problems occur are the following:

```
QUERY: SELECT * from street;
QUERY: SELECT * from iexit;
```

34.3.5. Random differences

There is at least one case in the "random" test script that is intended to produce random results. This causes random to fail the regression test once in a while (perhaps once in every five to ten trials). Typing

```
diff results/random.out expected/random.out
```

should produce only one or a few lines of differences. You need not worry unless the random test always fails in repeated attempts. (On the other hand, if the random test is *never* reported to fail even in many trials of the regress tests, you probably *should* worry.)

34.3.6. The "expected" files

The `./expected/*.out` files were adapted from the original monolithic `expected.input` file provided by Jolly Chen et al. Newer versions of these files generated on various development machines have been substituted after careful (?) inspection. Many of the development machines are running a Unix OS variant (FreeBSD, Linux, etc) on Ix86 hardware. The original `expected.input` file was created on a SPARC Solaris 2.4 system using the `postgres5-1.02a5.tar.gz` source tree. It was compared with a file created on an I386 Solaris 2.4 system and the differences were only in the floating point polygons in the 3rd digit to the right of the decimal point. The original `sample.regress.out` file was from the postgres-1.01 release constructed by Jolly Chen. It may have been created on a DEC ALPHA machine as the `Makefile.global` in the postgres-1.01 release has PORTNAME=alpha.

34.4. Platform-specific comparison files

Since some of the tests inherently produce platform-specific results, we have provided a way to supply platform-specific result comparison files. Frequently, the same variation applies to multiple platforms; rather than supplying a separate comparison file for every platform, there is a mapping file that defines which comparison file to use. So, to eliminate bogus test "failures" for a particular platform, you must choose or make a variant result file, and then add a line to the mapping file, which is "resultmap".

Each line in the mapping file is of the form

```
testname/platformnamepattern=comparisonfilename
```

The test name is just the name of the particular regression test module. The platform name pattern is a pattern in the style of expr(1) (that is, a regular expression with an implicit ^ anchor at the start). It is matched against the platform name as printed by config.guess. The comparison file name is the name of the substitute result comparison file.

For example: the int2 regress test includes a deliberate entry of a value that is too large to fit in int2. The specific error message that is produced is platform-dependent; our reference platform emits

```
   ERROR:  pg_atoi: error reading "100000": Numerical result out
of range
```

but a fair number of other Unix platforms emit

```
   ERROR:  pg_atoi: error reading "100000": Result too large
```

Therefore, we provide a variant comparison file, int2-too-large.out, that includes this spelling of the error message. To silence the bogus "failure" message on HPPA platforms, resultmap includes

```
int2/hppa=int2-too-large
```

which will trigger on any machine for which config.guess's output begins with 'hppa'. Other lines in resultmap select the variant comparison file for other platforms where it's appropriate.

Chapter 35. Release Notes

35.1. Release 7.0.3

Release date 2000-11-04. This has a variety of fixes from 7.0.2.

35.1.1. Migration to v7.0.3

A dump/restore is *not* required for those running v7.0.*.

35.1.2. Changes

```
Jdbc fixes (Peter)
Large object fix (Tom)
Fix lean in COPY WITH OIDS leak (Tom)
Fix backwards-index-scan (Tom)
Fix SELECT ... FOR UPDATE so it checks for duplicate keys (Hiroshi)
Add --enable-syslog to configure (Marc)
Fix abort transaction at backend exit in rare cases (Tom)
Fix for psql \l+ when multi-byte enabled (Tatsuo)
Allow PL/pgSQL to accept non ascii identifiers (Tatsuo)
Make vacuum always flush buffers (Tom)
Fix to allow cancel while waiting for a lock (Hiroshi)
Fix for memory aloocation problem in user authentication code (Tom)
Remove bogus use of int4out() (Tom)
Fixes for multiple subqueries in COALESCE or BETWEEN (Tom)
Fix for failure of triggers on heap open in certain cases (Jeroen
 van
    Vianen)
Fix for erroneous selectivity of not-equals (Tom)
Fix for erroneous use of strcmp() (Tom)
Fix for bug where storage manager accesses items beyond end of file
    (Tom)
Fix to include kernel errno message in all smgr elog messages (Tom)
Fix for '.' not in PATH at build time (SL Baur)
Fix for out-of-file-descriptors error (Tom)
Fix to make pg_dump dump 'iscachable' flag for functions (Tom)
Fix for subselect in targetlist of Append node (Tom)
Fix for mergejoin plans (Tom)
Fix TRUNCATE failure on relations with indexes (Tom)
Avoid database-wide restart on write error (Hiroshi)
Fix nodeMaterial to honor chgParam by recomputing its output (Tom)
Fix VACUUM problem with moving chain of update tuples when source
 and
    destination of a tuple lie on the same page (Tom)
Fix user.c CommandCounterIncrement (Tom)
Fix for AM/PM boundary problem in to_char() (Karel Zak)
Fix TIME aggregate handling (Tom)
Fix to_char() to avoid coredump on NULL input (Tom)
Buffer fix (Tom)
Fix for inserting/copying longer multibyte strings into char() data
    types (Tatsuo)
```

Chapter 36. Release Notes

36.1. Release 7.0.2

Release date 2000-06-05. This is a repackaging of 7.0.1 with added documentation.

36.1.1. Migration to v7.0.2

A dump/restore is *not* required for those running v7.0.*.

36.1.2. Changes

```
Added documentation to tarball.
```

36.2. Release 7.0.1

Release date 2000-06-01. This is a cleanup release for 7.0.

36.2.1. Migration to v7.0.1

A dump/restore is *not* required for those running v7.0.

36.2.2. Changes

```
Fix many CLUSTER failures (Tom)
Allow ALTER TABLE RENAME works on indexes (Tom)
Fix plpgsql to handle datetime->timestamp and timespan->interval
 (Bruce)
New configure --with-setproctitle switch to use setproctitle()
 (Marc, Bruce)
Fix the off by one errors in ResultSet from 6.5.3, and more.
jdbc ResultSet fixes (Joseph Shraibman)
optimizer tunings (Tom)
Fix create user for pgaccess
Fix for UNLISTEN failure
IRIX fixes (David Kaelbling)
QNX fixes (Andreas Kardos)
Reduce COPY IN lock level (Tom)
Change libpqeasy to use PQconnectdb() style parameters (Bruce)
Fix pg_dump to handle OID indexes (Tom)
Fix small memory leak (Tom)
Solaris fix for createdb/dropdb (Tatsuo)
Fix for non-blocking connections (Alfred Perlstein)
Fix improper recovery after RENAME TABLE failures (Tom)
Copy pg_ident.conf.sample into /lib directory in install (Bruce)
Add SJIS UDC (NEC selection IBM kanji) support (Eiji Tokuya)
Fix too long syslog message (Tatsuo)
Fix problem with quoted indexes that are too long (Tom)
JDBC ResultSet.getTimestamp() fix (Gregory Krasnow & Floyd
 Marinescu)
ecpg changes (Michael)
```

36.3. Release 7.0

Released 2000-05-08. This release contains improvements in many areas, demonstrating the continued growth of PostgreSQL. There are more improvements and fixes in 7.0 than in any previous release. The developers have confidence that this is the best release yet; we do our best to put out only solid releases, and this one is no exception.

Major changes in this release:

Foreign Keys

Foreign keys are now implemented, with the exception of PARTIAL MATCH foreign keys. Many users have been asking for this feature, and we are pleased to offer it.

Optimizer Overhaul

Continuing on work started a year ago, the optimizer has been improved, allowing better query plan selection and faster performance with less memory usage.

Updated psql

psql, our interactive terminal monitor, has been updated with a variety of new features. See the psql manual page for details.

Join Syntax

SQL92 join syntax is now supported, though only as INNER JOINs for this release. JOIN, NATURAL JOIN, JOIN/USING, JOIN/ON are available, as are column correlation names.

36.3.1. Migration to v7.0

A dump/restore using pg_dump is required for those wishing to migrate data from any previous release of Postgres. For those upgrading from 6.5.*, you may instead use pg_upgrade to upgrade to this release; however, a full dump/reload installation is always the most robust method for upgrades.

Interface and compatibility issues to consider for the new release include:

- The date/time types `datetime` and `timespan` have been superceded by the SQL92-defined types `timestamp` and `interval`. Although there has been some effort to ease the transition by allowing Postgres to recognize the deprecated type names and translate them to the new type names, this mechanism may not be completely transparent to your existing application.
- The optimizer has been substantially improved in the area of query cost estimation. In some cases, this will result in decreased query times as the optimizer makes a better choice for the preferred plan. However, in a small number of cases, usually involving pathological distributions of data, your query times may go up. If you are dealing with large amounts of data, you may want to check your queries to verify performance.
- The JDBC and ODBC interfaces have been upgraded and extended.
- The string function `CHAR_LENGTH` is now a native function. Previous versions translated this into a call to `LENGTH`, which could result in ambiguity with other types implementing `LENGTH` such as the geometric types.

36.3.2. Changes

```
Bug Fixes
---------
Prevent function calls exceeding maximum number of arguments (Tom)
Improve CASE construct (Tom)
Fix SELECT coalesce(f1,0) FROM int4_tbl GROUP BY f1 (Tom)
```

```
Fix SELECT sentence.words[0] FROM sentence GROUP BY
 sentence.words[0] (Tom)
Fix GROUP BY scan bug (Tom)
Improvements in SQL grammar processing (Tom)
Fix for views involved in INSERT ... SELECT ... (Tom)
Fix for SELECT a/2, a/2 FROM test_missing_target GROUP BY a/2 (Tom)
Fix for subselects in INSERT ... SELECT (Tom)
Prevent INSERT ... SELECT ... ORDER BY (Tom)
Fixes for relations greater than 2GB, including vacuum
Improve propagating system table changes to other backends (Tom)
Improve propagating user table changes to other backends (Tom)
Fix handling of temp tables in complex situations (Bruce, Tom)
Allow table locking at table open, improving concurrent reliability
 (Tom)
Properly quote sequence names in pg_dump (Ross J. Reedstrom)
Prevent DROP DATABASE while others accessing
Prevent any rows from being returned by GROUP BY if no rows
 processed (Tom)
Fix SELECT COUNT(1) FROM table WHERE ...' if no rows matching WHERE
 (Tom)
Fix pg_upgrade so it works for MVCC (Tom)
Fix for SELECT ... WHERE x IN (SELECT ... HAVING SUM(x) > 1) (Tom)
Fix for "f1 datetime DEFAULT 'now'"  (Tom)
Fix problems with CURRENT_DATE used in DEFAULT (Tom)
Allow comment-only lines, and ;;; lines too. (Tom)
Improve recovery after failed disk writes, disk full (Hiroshi)
Fix cases where table is mentioned in FROM but not joined (Tom)
Allow HAVING clause without aggregate functions (Tom)
Fix for "--" comment and no trailing newline, as seen in perl
 interface
Improve pg_dump failure error reports (Bruce)
Allow sorts and hashes to exceed 2GB file sizes (Tom)
Fix for pg_dump dumping of inherited rules (Tom)
Fix for NULL handling comparisons (Tom)
Fix inconsistent state caused by failed CREATE/DROP commands
 (Hiroshi)
Fix for dbname with dash
Prevent DROP INDEX from interfering with other backends (Tom)
Fix file descriptor leak in verify_password()
Fix for "Unable to identify an operator =$" problem
Fix ODBC so no segfault if CommLog and Debug enabled (Dirk
 Niggemann)
Fix for recursive exit call (Massimo)
Fix for extra-long timezones (Jeroen van Vianen)
Make pg_dump preserve primary key information (Peter E)
Prevent databases with single quotes (Peter E)
Prevent DROP DATABASE inside  transaction (Peter E)
ecpg memory leak fixes (Stephen Birch)
Fix for SELECT null::text, SELECT int4fac(null) and SELECT 2 +
 (null) (Tom)
Y2K timestamp fix (Massimo)
Fix for VACUUM 'HEAP_MOVED_IN was not expected' errors (Tom)
Fix for views with tables/columns containing spaces  (Tom)
Prevent permissions on indexes (Peter E)
Fix for spinlock stuck problem when error is generated (Hiroshi)
Fix ipcclean on Linux
Fix handling of NULL constraint conditions (Tom)
Fix memory leak in odbc driver (Nick Gorham)
```

```
Fix for permission check on UNION tables (Tom)
Fix to allow SELECT 'a' LIKE 'a' (Tom)
Fix for SELECT 1 + NULL (Tom)
Fixes to CHAR
Fix log() on numeric type (Tom)
Deprecate ':' and ';' operators
Allow vacuum of temporary tables
Disallow inherited columns with the same name as new columns
Recover or force failure when disk space is exhausted (Hiroshi)
Fix INSERT INTO ... SELECT with AS columns matching result columns
Fix INSERT ... SELECT ... GROUP BY groups by target columns not
 source columns (Tom)
Fix CREATE TABLE test (a char(5) DEFAULT text '', b int4) with
 INSERT (Tom)
Fix UNION with LIMIT
Fix CREATE TABLE x AS SELECT 1 UNION SELECT 2
Fix CREATE TABLE test(col char(2) DEFAULT user)
Fix mismatched types in CREATE TABLE ... DEFAULT
Fix SELECT * FROM pg_class where oid in (0,-1)
Fix SELECT COUNT('asdf') FROM pg_class WHERE oid=12
Prevent user who can create databases can modifying pg_database
 table (Peter E)
Fix btree to give a useful elog when key > 1/2 (page - overhead)
 (Tom)
Fix INSERT of 0.0 into DECIMAL(4,4) field (Tom)

Enhancements
------------
New CLI interface include file sqlcli.h, based on SQL3/SQL98
Remove all limits on query length, row length limit still exists
 (Tom)
Update jdbc protocol to 2.0 (Jens Glaser[1])
Add TRUNCATE command to quickly truncate relation (Mike Mascari)
Fix to give super user and createdb user proper update catalog
 rights (Peter E)
Allow ecpg bool variables to have NULL values (Christof)
Issue ecpg error if NULL value for variable with no NULL indicator
 (Christof)
Allow ^C to cancel COPY command (Massimo)
Add SET FSYNC and SHOW PG_OPTIONS commands(Massimo)
Function name overloading for dynamically-loaded C functions
 (Frankpitt)
Add CmdTuples() to libpq++(Vince)
New CREATE CONSTRAINT TRIGGER and SET CONSTRAINTS commands(Jan)
Allow CREATE FUNCTION/WITH clause to be used for all language types
configure --enable-debug adds -g (Peter E)
configure --disable-debug removes -g (Peter E)
Allow more complex default expressions (Tom)
First real FOREIGN KEY constraint trigger functionality (Jan)
Add FOREIGN KEY ... MATCH FULL ... ON DELETE CASCADE (Jan)
Add FOREIGN KEY ... MATCH <unspecified> referential actions (Don
 Baccus)
Allow WHERE restriction on ctid (physical heap location) (Hiroshi)
Move pginterface from contrib to interface directory, rename to
 pgeasy (Bruce)
Change pgeasy connectdb() parameter ordering (Bruce)
```

[1] mailto:jens@jens.de

```
Require SELECT DISTINCT target list to have all ORDER BY columns
 (Tom)
Add Oracle's COMMENT ON command (Mike Mascari[2])
libpq's PQsetNoticeProcessor function now returns previous
 hook(Peter E)
Prevent PQsetNoticeProcessor from being set to NULL (Peter E)
Make USING in COPY optional (Bruce)
Allow subselects in the target list (Tom)
Allow subselects on the left side of comparison operators (Tom)
New parallel regression test (Jan)
Change backend-side COPY to write files with permissions 644 not
 666 (Tom)
Force permissions on PGDATA directory to be secure, even if it
 exists (Tom)
Added psql LASTOID variable to return last inserted oid (Peter E)
Allow concurrent vacuum and remove pg_vlock vacuum lock file (Tom)
Add permissions check for vacuum (Peter E)
New libpq functions to allow asynchronous connections:
 PQconnectStart(),
   PQconnectPoll(), PQresetStart(), PQresetPoll(),
 PQsetenvStart(),
   PQsetenvPoll(), PQsetenvAbort (Ewan Mellor)
New libpq PQsetenv() function (Ewan Mellor)
create/alter user extension (Peter E)
New postmaster.pid and postmaster.opts under $PGDATA (Tatsuo)
New scripts for create/drop user/db (Peter E)
Major psql overhaul (Peter E)
Add const to libpq interface (Peter E)
New libpq function PQoidValue (Peter E)
Show specific non-aggregate causing problem with GROUP BY (Tom)
Make changes to pg_shadow recreate pg_pwd file (Peter E)
Add aggregate(DISTINCT ...) (Tom)
Allow flag to control COPY input/output of NULLs (Peter E)
Make postgres user have a password by default (Peter E)
Add CREATE/ALTER/DROP GROUP (Peter E)
All administration scripts now support --long options (Peter E,
 Karel)
Vacuumdb script now supports --all option (Peter E)
ecpg new portable FETCH syntax
Add ecpg EXEC SQL IFDEF, EXEC SQL IFNDEF, EXEC SQL ELSE, EXEC SQL
 ELIF
 and EXEC SQL ENDIF directives
Add pg_ctl script to control backend startup (Tatsuo)
Add postmaster.opts.default file to store startup flags (Tatsuo)
Allow --with-mb=SQL_ASCII
Increase maximum number of index keys to 16 (Bruce)
Increase maximum number of function arguments to 16 (Bruce)
Allow configuration of maximum number of index keys and arguments
 (Bruce)
Allow unprivileged users to change their passwords (Peter E)
Password authentication enabled; required for new users (Peter E)
Disallow dropping a user who owns a database (Peter E)
Change initdb option --with-mb to --enable-multibyte
Add option for initdb to prompts for superuser password (Peter E)
Allow complex type casts like col::numeric(9,2) and
 col::int2::float8 (Tom)
```

[2] mailto:mascarim@yahoo

```
Updated user interfaces on initdb, initlocation, pg_dump, ipcclean
 (Peter E)
New pg_char_to_encoding() and pg_encoding_to_char() functions
 (Tatsuo)
Libpq non-blocking mode (Alfred Perlstein)
Improve conversion of types in casts that don't specify a length
New plperl internal programming language (Mark Hollomon)
Allow COPY IN to read file that do not end with a newline (Tom)
Indicate when long identifiers are truncated (Tom)
Allow aggregates to use type equivalency (Peter E)
Add Oracle's to_char(), to_date(), to_datetime(), to_timestamp(),
 to_number()
 conversion functions (Karel Zak <zakkr@zf.jcu.cz>)
Add SELECT DISTINCT ON (expr [, expr ...]) targetlist ... (Tom)
Check to be sure ORDER BY is compatible with the DISTINCT operation
 (Tom)
Add NUMERIC and int8 types to ODBC
Improve EXPLAIN results for Append, Group, Agg, Unique (Tom)
Add ALTER TABLE ... ADD FOREIGN KEY (Stephan Szabo)
Allow SELECT .. FOR UPDATE in PL/pgSQL (Hiroshi)
Enable backward sequential scan even after reaching EOF (Hiroshi)
Add btree indexing of boolean values, >= and <= (Don Baccus)
Print current line number when COPY FROM fails (Massimo)
Recognize POSIX time zone e.g. "PST+8" and "GMT-8" (Thomas)
Add DEC as synonym for DECIMAL (Thomas)
Add SESSION_USER as SQL92 keyword, same as CURRENT_USER (Thomas)
Implement SQL92 column aliases (aka correlation names) (Thomas)
Implement SQL92 join syntax (Thomas)
Make INTERVAL reserved word allowed as a column identifier (Thomas)
Implement REINDEX command (Hiroshi)
Accept ALL in aggregate function SUM(ALL col) (Tom)
Prevent GROUP BY from using column aliases (Tom)
New psql \encoding option (Tatsuo)
Allow PQrequestCancel() to terminate when in waiting-for-lock state
 (Hiroshi)
Allow negation of a negative number in all cases
Add ecpg descriptors (Christof, Michael)
Allow CREATE VIEW v AS SELECT f1::char(8) FROM tbl
Allow casts with length, like foo::char(8)
New libpq functions PQsetClientEncoding(), PQclientEncoding()
 (Tatsuo)
Add support for SJIS user defined characters (Tatsuo)
Larger views/rules supported
Make libpq's PQconndefaults() thread-safe (Tom)
Disable // as comment to be ANSI conforming, should use -- (Tom)
Allow column aliases on views CREATE VIEW name (collist)
Fixes for views with subqueries (Tom)
Allow UPDATE table SET fld = (SELECT ...) (Tom)
SET command options no longer require quotes
Update pgaccess to 0.98.6
New SET SEED command
New pg_options.sample file
New SET FSYNC command (Massimo)
Allow pg_descriptions when creating tables
Allow pg_descriptions when creating types, columns, and functions
Allow psql \copy to allow delimiters (Peter E)
Allow psql to print nulls as distinct from "" [null] (Peter E)
```

```
Types
-----
Many array fixes (Tom)
Allow bare column names to be subscripted as arrays (Tom)
Improve type casting of int and float constants (Tom)
Cleanups for int8 inputs, range checking, and type conversion (Tom)
Fix for SELECT timespan('21:11:26'::time) (Tom)
netmask('x.x.x.x/0') is 255.255.255.255 instead of 0.0.0.0 (Oleg
 Sharoiko)
Add btree index on NUMERIC (Jan)
Perl fix for large objects containing NUL characters (Douglas
 Thomson)
ODBC fix for for large objects (free)
Fix indexing of cidr data type
Fix for Ethernet MAC addresses (macaddr type) comparisons
Fix for date/time types when overflows happened in computations
 (Tom)
Allow array on int8 (Peter E)
Fix for rounding/overflow of NUMERIC type, like NUMERIC(4,4) (Tom)
Allow NUMERIC arrays
Fix bugs in NUMERIC ceil() and floor() functions (Tom)
Make char_length()/octet_length including trailing blanks (Tom)
Made abstime/reltime use int4 instead of time_t (Peter E)
New lztext data type for compressed text fields
Revise code to handle coercion of int and float constants (Tom)
Start at new code to implement a BIT and BIT VARYING type (Adriaan
 Joubert)
NUMERIC now accepts scientific notation (Tom)
NUMERIC to int4 rounds (Tom)
Convert float4/8 to NUMERIC properly (Tom)
Allow type conversion with NUMERIC (Thomas)
Make ISO date style (2000-02-16 09:33) the default (Thomas)
Add NATIONAL CHAR [ VARYING ] (Thomas)
Allow NUMERIC round and trunc to accept negative scales (Tom)
New TIME WITH TIME ZONE type (Thomas)
Add MAX()/MIN() on time type (Thomas)
Add abs(), mod(), fac() for int8 (Thomas)
Rename functions to round(), sqrt(), cbrt(), pow() for float8
 (Thomas)
Add transcendental math functions (e.g. sin(), acos()) for float8
 (Thomas)
Add exp() and ln() for NUMERIC type
Rename NUMERIC power() to pow() (Thomas)
Improved TRANSLATE() function (Edwin Ramirez, Tom)
Allow X=-Y operators  (Tom)
Allow SELECT float8(COUNT(*))/(SELECT COUNT(*) FROM t) FROM t GROUP
 BY f1; (Tom)
Allow LOCALE to use indexes in regular expression searches (Tom)
Allow creation of functional indexes to use default types

Performance
-----------
Prevent exponential space consumption with many AND's and OR's
 (Tom)
Collect attribute selectivity values for system columns (Tom)
Reduce memory usage of aggregates (Tom)
Fix for LIKE optimization to use indexes with multi-byte encodings
 (Tom)
```

```
Fix r-tree index optimizer selectivity (Thomas)
Improve optimizer selectivity computations and functions (Tom)
Optimize btree searching for cases where many equal keys exist
 (Tom)
Enable fast LIKE index processing only if index present (Tom)
Re-use free space on index pages with duplicates (Tom)
Improve hash join processing (Tom)
Prevent descending sort if result is already sorted(Hiroshi)
Allow commuting of index scan query qualifications (Tom)
Prefer index scans in cases where ORDER BY/GROUP BY is required
 (Tom)
Allocate large memory requests in fix-sized chunks for performance
 (Tom)
Fix vacuum's performance by reducing memory allocation requests
 (Tom)
Implement constant-expression simplification (Bernard Frankpitt,
 Tom)
Use secondary columns to be used to determine start of index scan
 (Hiroshi)
Prevent quadruple use of disk space when doing internal sorting
 (Tom)
Faster sorting by calling fewer functions (Tom)
Create system indexes to match all system caches (Bruce, Hiroshi)
Make system caches use system indexes (Bruce)
Make all system indexes unique (Bruce)
Improve pg_statistics management for VACUUM speed improvement (Tom)
Flush backend cache less frequently (Tom, Hiroshi)
COPY now reuses previous memory allocation, improving performance
 (Tom)
Improve optimization cost estimation (Tom)
Improve optimizer estimate of range queries x > lowbound AND x <
 highbound (Tom)
Use DNF instead of CNF where appropriate (Tom, Taral)
Further cleanup for OR-of-AND WHERE-clauses (Tom)
Make use of index in OR clauses (x = 1 AND y = 2) OR (x = 2 AND y =
 4) (Tom)
Smarter optimizer computations for random index page access (Tom)
New SET variable to control optimizer costs (Tom)
Optimizer queries based on LIMIT, OFFSET, and EXISTS qualifications
 (Tom)
Reduce optimizer internal housekeeping of join paths for speedup
 (Tom)
Major subquery speedup (Tom)
Fewer fsync writes when fsync is not disabled (Tom)
Improved LIKE optimizer estimates (Tom)
Prevent fsync in SELECT-only queries (Vadim)
Make index creation use psort code, because it is now faster (Tom)
Allow creation of sort temp tables > 1 Gig

Source Tree Changes
-------------------
Fix for linux PPC compile
New generic expression-tree-walker subroutine (Tom)
Change form() to varargform() to prevent portability problems
Improved range checking for large integers on Alphas
Clean up #include in /include directory (Bruce)
Add scripts for checking includes (Bruce)
Remove un-needed #include's from *.c files (Bruce)
```

```
Change #include's to use <> and "" as appropriate (Bruce)
Enable WIN32 compilation of libpq
Alpha spinlock fix from Uncle George[3]
Overhaul of optimizer data structures (Tom)
Fix to cygipc library (Yutaka Tanida)
Allow pgsql to work on newer Cygwin snapshots (Dan)
New catalog version number (Tom)
Add Linux ARM
Rename heap_replace to heap_update
Update for QNX (Dr. Andreas Kardos)
New platform-specific regression handling (Tom)
Rename oid8 -> oidvector and int28 -> int2vector (Bruce)
Included all yacc and lex files into the distribution (Peter E.)
Remove lextest, no longer needed (Peter E)
Fix for libpq and psql on Win32 (Magnus)
Internally change datetime and timespan into timestamp and interval
 (Thomas)
Fix for plpgsql on BSDI
Add SQL_ASCII test case to the regression test (Tatsuo)
configure --with-mb now deprecated (Tatsuo)
NT fixes
NetBSD fixes Johnny C. Lam[4]
Fixes for Alpha compiles
New multibyte encodings
```

36.4. Release 6.5.3

Released 1999-10-13. This is basically a cleanup release for 6.5.2. We have added a new pgaccess that was missing in 6.5.2, and installed an NT-specific fix.

36.4.1. Migration to v6.5.3

A dump/restore is *not* required for those running 6.5.*.

36.4.2. Changes

```
Updated version of pgaccess 0.98
NT-specific patch
Fix dumping rules on inherited tables
```

36.5. Release 6.5.2

Released 1999-09-15. This is basically a cleanup release for 6.5.1. We have fixed a variety of problems reported by 6.5.1 users.

36.5.1. Migration to v6.5.2

A dump/restore is *not* required for those running 6.5.*.

36.5.2. Changes

```
subselect+CASE fixes(Tom)
```

[3] mailto:gatgul@voicenet.com
[4] mailto:lamj@stat.cmu.edu

```
Add SHLIB_LINK setting for solaris_i386 and solaris_sparc
 ports(Daren Sefcik)
Fixes for CASE in WHERE join clauses(Tom)
Fix BTScan abort(Tom)
Repair the check for redundant UNIQUE and PRIMARY KEY
 indices(Thomas)
Improve it so that it checks for multi-column constraints(Thomas)
Fix for Win32 making problem with MB enabled(Hiroki Kataoka)
Allow BSD yacc and bison to compile pl code(Bruce)
Fix SET NAMES working
int8 fixes(Thomas)
Fix vacuum's memory consumption(Hiroshi,Tatsuo)
Reduce the total memory consumption of vacuum(Tom)
Fix for timestamp(datetime)
Rule deparsing bugfixes(Tom)
Fix quoting problems in mkMakefile.tcldefs.sh.in and
 mkMakefile.tkdefs.sh.in(Tom)
This is to re-use space on index pages freed by vacuum(Vadim)
document -x for pg_dump(Bruce)
Fix for unary operators in rule deparser(Tom)
Comment out FileUnlink of excess segments during mdtruncate()(Tom)
Irix linking fix from Yu Cao >yucao@falcon.kla-tencor.com<
Repair logic error in LIKE: should not return LIKE_ABORT
   when reach end of pattern before end of text(Tom)
Repair incorrect cleanup of heap memory allocation during
 transaction abort(Tom)
Updated version of pgaccess 0.98
```

36.6. Release 6.5.1

Released 1999-07-15.

This is basically a cleanup release for 6.5. We have fixed a variety of problems reported by 6.5 users.

36.6.1. Migration to v6.5.1

A dump/restore is *not* required for those running 6.5.

36.6.2. Changes

```
Add NT README file
Portability fixes for linux_ppc, Irix, linux_alpha, OpenBSD, alpha
Remove QUERY_LIMIT, use SELECT...LIMIT
Fix for EXPLAIN on inheritance(Tom)
Patch to allow vacuum on multi-segment tables(Hiroshi)
R-Tree optimizer selectivity fix(Tom)
ACL file descriptor leak fix(Atsushi Ogawa)
New expresssion subtree code(Tom)
Avoid disk writes for read-only transactions(Vadim)
Fix for removal of temp tables if last transaction was
 aborted(Bruce)
Fix to prevent too large tuple from being created(Bruce)
plpgsql fixes
Allow port numbers 32k - 64k(Bruce)
Add ^ precidence(Bruce)
Rename sort files called pg_temp to pg_sorttemp(Bruce)
```

```
Fix for microseconds in time values(Tom)
Tutorial source cleanup
New linux_m68k port
Fix for sorting of NULL's in some cases(Tom)
Shared library dependencies fixed (Tom)
Fixed glitches affecting GROUP BY in subselects(Tom)
Fix some compiler warnings (Tomoaki Nishiyama)
Add Win1250 (Czech) support (Pavel Behal)
```

36.7. Release 6.5

Released 1999-06-09. This release marks a major step in the development team's mastery of the source code we inherited from Berkeley. You will see we are now easily adding major features, thanks to the increasing size and experience of our world-wide development team.

Here is a brief summary of the more notable changes:

Multi-version concurrency control(MVCC)

> This removes our old table-level locking, and replaces it with a locking system that is superior to most commercial database systems. In a traditional system, each row that is modified is locked until committed, preventing reads by other users. MVCC uses the natural multi-version nature of PostgreSQL to allow readers to continue reading consistent data during writer activity. Writers continue to use the compact pg_log transaction system. This is all performed without having to allocate a lock for every row like traditional database systems. So, basically, we no longer are restricted by simple table-level locking; we have something better than row-level locking.

Hot backups from pg_dump

> pg_dump takes advantage of the new MVCC features to give a consistant database dump/backup while the database stays online and available for queries.

Numeric data type

> We now have a true numeric data type, with user-specified precision.

Temporary tables

> Temporary tables are guaranteed to have unique names within a database session, and are destroyed on session exit.

New SQL features

> We now have CASE, INTERSECT, and EXCEPT statement support. We have new LIMIT/OFFSET, SET TRANSACTION ISOLATION LEVEL, SELECT ... FOR UPDATE, and an improved LOCK TABLE command.

Speedups

> We continue to speed up PostgreSQL, thanks to the variety of talents within our team. We have sped up memory allocation, optimization, table joins, and row transfer routines.

Ports

> We continue to expand our port list, this time including WinNT/ix86 and NetBSD/arm32.

Interfaces

> Most interfaces have new versions, and existing functionality has been improved.

Documentation

New and updated material is present throughout the documentation. New FAQs have been contributed for SGI and AIX platforms. The *Tutorial* has introductory information on SQL from Stefan Simkovics. For the *User's Guide*, there are reference pages covering the postmaster and more utility programs, and a new appendix contains details on date/time behavior. The *Administrator's Guide* has a new chapter on troubleshooting from Tom Lane. And the *Programmer's Guide* has a description of query processing, also from Stefan, and details on obtaining the Postgres source tree via anonymous CVS and CVSup.

36.7.1. Migration to v6.5

A dump/restore using pg_dump is required for those wishing to migrate data from any previous release of Postgres. pg_upgrade can *not* be used to upgrade to this release because the on-disk structure of the tables has changed compared to previous releases.

The new Multi-Version Concurrency Control (MVCC) features can give somewhat different behaviors in multi-user environments. *Read and understand the following section to ensure that your existing applications will give you the behavior you need.*

36.7.1.1. Multi-Version Concurrency Control

Because readers in 6.5 don't lock data, regardless of transaction isolation level, data read by one transaction can be overwritten by another. In other words, if a row is returned by `SELECT` it doesn't mean that this row really exists at the time it is returned (i.e. sometime after the statement or transaction began) nor that the row is protected from being deleted or updated by concurrent transactions before the current transaction does a commit or rollback.

To ensure the actual existence of a row and protect it against concurrent updates one must use `SELECT FOR UPDATE` or an appropriate `LOCK TABLE` statement. This should be taken into account when porting applications from previous releases of Postgres and other environments.

Keep the above in mind if you are using `contrib/refint.*` triggers for referential integrity. Additional technics are required now. One way is to use `LOCK parent_table IN SHARE ROW EXCLUSIVE MODE` command if a transaction is going to update/delete a primary key and use `LOCK parent_table IN SHARE MODE` command if a transaction is going to update/insert a foreign key.

Note

Note that if you run a transaction in SERIALIZABLE mode then you must execute the `LOCK` commands above before execution of any DML statement (`SELECT/INSERT/DELETE/UPDATE/FETCH/COPY_TO`) in the transaction.

These inconveniences will disappear in the future when the ability to read dirty (uncommitted) data (regardless of isolation level) and true referential integrity will be implemented.

36.7.2. Changes

```
Bug Fixes
---------
Fix text<->float8 and text<->float4 conversion functions(Thomas)
Fix for creating tables with mixed-case constraints(Billy)
Change exp()/pow() behavior to generate error on underflow/
overflow(Jan)
Fix bug in pg_dump -z
Memory overrun cleanups(Tatsuo)
Fix for lo_import crash(Tatsuo)
Adjust handling of data type names to suppress double
 quotes(Thomas)
Use type coercion for matching columns and DEFAULT(Thomas)
```

```
Fix deadlock so it only checks once after one second of
 sleep(Bruce)
Fixes for aggregates and PL/pgsql(Hiroshi)
Fix for subquery crash(Vadim)
Fix for libpq function PQfnumber and case-insensitive names(Bahman
 Rafatjoo)
Fix for large object write-in-middle, no extra block, memory
 consumption(Tatsuo)
Fix for pg_dump -d or -D and  quote special characters in INSERT
Repair serious problems with dynahash(Tom)
Fix INET/CIDR portability problems
Fix problem with selectivity error in ALTER TABLE ADD COLUMN(Bruce)
Fix executor so mergejoin of different column types works(Tom)
Fix for Alpha OR selectivity bug
Fix OR index selectivity problem(Bruce)
Fix so \d shows proper length for char()/varchar()(Ryan)
Fix tutorial code(Clark)
Improve destroyuser checking(Oliver)
Fix for Kerberos(Rodney McDuff)
Fix for dropping database while dirty buffers(Bruce)
Fix so sequence nextval() can be case-sensitive(Bruce)
Fix !!= operator
Drop buffers before destroying database files(Bruce)
Fix case where executor evaluates functions twice(Tatsuo)
Allow sequence nextval actions to be case-sensitive(Bruce)
Fix optimizer indexing not working for negative numbers(Bruce)
Fix for memory leak in executor with fjIsNull
Fix for aggregate memory leaks(Erik Riedel)
Allow username containing a dash GRANT permissions
Cleanup of NULL in inet types
Clean up system table bugs(Tom)
Fix problems of PAGER and \? command(Masaaki Sakaida)
Reduce default multi-segment file size limit to 1GB(Peter)
Fix for dumping of CREATE OPERATOR(Tom)
Fix for backward scanning of cursors(Hiroshi Inoue)
Fix for COPY FROM STDIN when using \i(Tom)
Fix for subselect is compared inside an expression(Jan)
Fix handling of error reporting while returning rows(Tom)
Fix problems with reference to array types(Tom,Jan)
Prevent UPDATE SET oid(Jan)
Fix pg_dump so -t option can handle case-sensitive tablenames
Fixes for GROUP BY in special cases(Tom, Jan)
Fix for memory leak in failed queries(Tom)
DEFAULT now supports mixed-case identifiers(Tom)
Fix for multi-segment uses of DROP/RENAME table, indexes(Ole
 Gjerde)
Disable use of pg_dump with both -o and -d options(Bruce)
Allow pg_dump to properly dump GROUP permissions(Bruce)
Fix GROUP BY in INSERT INTO table SELECT * FROM table2(Jan)
Fix for computations in views(Jan)
Fix for aggregates on array indexes(Tom)
Fix for DEFAULT handles single quotes in value requiring too many
 quotes
Fix security problem with non-super users importing/exporting large
 objects(Tom)
Rollback of transaction that creates table cleaned up properly(Tom)
Fix to allow long table and column names to generate proper serial
 names(Tom)
```

```
Enhancements
------------
Add "vacuumdb" utility
Speed up libpq by allocating memory better(Tom)
EXPLAIN all indices used(Tom)
Implement CASE, COALESCE, NULLIF  expression(Thomas)
New pg_dump table output format(Constantin)
Add string min()/max() functions(Thomas)
Extend new type coercion techniques to aggregates(Thomas)
New moddatetime contrib(Terry)
Update to pgaccess 0.96(Constantin)
Add routines for single-byte "char" type(Thomas)
Improved substr() function(Thomas)
Improved multi-byte handling(Tatsuo)
Multi-version concurrency control/MVCC(Vadim)
New Serialized mode(Vadim)
Fix for tables over 2gigs(Peter)
New SET TRANSACTION ISOLATION LEVEL(Vadim)
New LOCK TABLE IN ... MODE(Vadim)
Update ODBC driver(Byron)
New NUMERIC data type(Jan)
New SELECT FOR UPDATE(Vadim)
Handle "NaN" and "Infinity" for input values(Jan)
Improved date/year handling(Thomas)
Improved handling of backend connections(Magnus)
New options ELOG_TIMESTAMPS and USE_SYSLOG options for log
 files(Massimo)
New TCL_ARRAYS option(Massimo)
New INTERSECT and EXCEPT(Stefan)
New pg_index.indisprimary for primary key tracking(D'Arcy)
New pg_dump option to allow dropping of tables before
 creation(Brook)
Speedup of row output routines(Tom)
New READ COMMITTED isolation level(Vadim)
New TEMP tables/indexes(Bruce)
Prevent sorting if result is already sorted(Jan)
New memory allocation optimization(Jan)
Allow psql to do \p\g(Bruce)
Allow multiple rule actions(Jan)
Added LIMIT/OFFSET functionality(Jan)
Improve optimizer when joining a large number of tables(Bruce)
New intro to SQL from S. Simkovics' Master's Thesis (Stefan,
 Thomas)
New intro to backend processing from S. Simkovics' Master's Thesis
 (Stefan)
Improved int8 support(Ryan Bradetich, Thomas, Tom)
New routines to convert between int8 and text/varchar types(Thomas)
New bushy plans, where meta-tables are joined(Bruce)
Enable right-hand queries by default(Bruce)
Allow reliable maximum number of backends to be set at configure
 time
      (--with-maxbackends and postmaster switch (-N backends))(Tom)
GEQO default now 10 tables because of optimizer speedups(Tom)
Allow NULL=Var for MS-SQL portability(Michael, Bruce)
Modify contrib check_primary_key() so either "automatic" or
 "dependent"(Anand)
Allow psql \d on a view show query(Ryan)
```

```
Speedup for LIKE(Bruce)
Ecpg fixes/features, see src/interfaces/ecpg/ChangeLog
 file(Michael)
JDBC fixes/features, see src/interfaces/jdbc/CHANGELOG(Peter)
Make % operator have precedence like /(Bruce)
Add new postgres -O option to allow system table structure
 changes(Bruce)
Update contrib/pginterface/findoidjoins script(Tom)
Major speedup in vacuum of deleted rows with indexes(Vadim)
Allow non-SQL functions to run different versions based on
 arguments(Tom)
Add -E option that shows actual queries sent by \dt and
 friends(Masaaki Sakaida)
Add version number in startup banners for psql(Masaaki Sakaida)
New contrib/vacuumlo removes large objects not referenced(Peter)
New initialization for table sizes so non-vacuumed tables perform
 better(Tom)
Improve error messages when a connection is rejected(Tom)
Support for arrays of char() and varchar() fields(Massimo)
Overhaul of hash code to increase reliability and performance(Tom)
Update to PyGreSQL 2.4(D'Arcy)
Changed debug options so -d4 and -d5 produce different node
 displays(Jan)
New pg_options: pretty_plan, pretty_parse, pretty_rewritten(Jan)
Better optimization statistics for system table access(Tom)
Better handling of non-default block sizes(Massimo)
Improve GEQO optimizer memory consumption(Tom)
UNION now suppports ORDER BY of columns not in target list(Jan)
Major libpq++ improvements(Vince Vielhaber)
pg_dump now uses -z(ACL's) as default(Bruce)
backend cache, memory speedups(Tom)
have pg_dump do everything in one snapshot transaction(Vadim)
fix for large object memory leakage, fix for pg_dumping(Tom)
INET type now respects netmask for comparisons
Make VACUUM ANALYZE only use a readlock(Vadim)
Allow VIEWs on UNIONS(Jan)
pg_dump now can generate consistent snapshots on active
 databases(Vadim)

Source Tree Changes
-------------------
Improve port matching(Tom)
Portability fixes for SunOS
Add NT/Win32 backend port and enable dynamic loading(Magnus and
 Daniel Horak)
New port to Cobalt Qube(Mips) running Linux(Tatsuo)
Port to NetBSD/m68k(Mr. Mutsuki Nakajima)
Port to NetBSD/sun3(Mr. Mutsuki Nakajima)
Port to NetBSD/macppc(Toshimi Aoki)
Fix for tcl/tk configuration(Vince)
Removed CURRENT keyword for rule queries(Jan)
NT dynamic loading now works(Daniel Horak)
Add ARM32 support(Andrew McMurry)
Better support for HPUX 11 and Unixware
Improve file handling to be more uniform, prevent file descriptor
 leak(Tom)
New install commands for plpgsql(Jan)
```

36.8. Release 6.4.2

1999-12-20

The 6.4.1 release was improperly packaged. This also has one additional bug fix.

36.8.1. Migration to v6.4.2

A dump/restore is *not* required for those running 6.4.*.

36.8.2. Changes

```
Fix for datetime constant problem on some platforms(Thomas)
```

36.9. Release 6.4.1

1999-12-18

This is basically a cleanup release for 6.4. We have fixed a variety of problems reported by 6.4 users.

36.9.1. Migration to v6.4.1

A dump/restore is *not* required for those running 6.4.

36.9.2. Changes

```
Add pg_dump -N flag to force double quotes around identifiers.
 This is
 the default(Thomas)
Fix for NOT in where clause causing crash(Bruce)
EXPLAIN VERBOSE coredump fix(Vadim)
Fix shared-library problems on Linux
Fix test for table existance to allow mixed-case and whitespace in
 the table name(Thomas)
Fix a couple of pg_dump bugs
Configure matches template/.similar entries better(Tom)
Change builtin function names from SPI_* to spi_*
OR WHERE clause fix(Vadim)
Fixes for mixed-case table names(Billy)
contrib/linux/postgres.init.csh/sh fix(Thomas)
libpq memory overrun fix
SunOS fixes(Tom)
Change exp() behavior to generate error on underflow(Thomas)
pg_dump fixes for memory leak, inheritance constraints, layout
 change
update pgaccess to 0.93
Fix prototype for 64-bit platforms
Multi-byte fixes(Tatsuo)
New ecpg man page
Fix memory overruns(Tatsuo)
Fix for lo_import() crash(Bruce)
Better search for install program(Tom)
Timezone fixes(Tom)
HPUX fixes(Tom)
Use implicit type coercion for matching DEFAULT values(Thomas)
Add routines to help with single-byte (internal) character
 type(Thomas)
```

```
Compilation of libpq for Win32 fixes(Magnus)
Upgrade to PyGreSQL 2.2(D'Arcy)
```

36.10. Release 6.4

1998-10-30

There are *many* new features and improvements in this release. Thanks to our developers and maintainers, nearly every aspect of the system has received some attention since the previous release. Here is a brief, incomplete summary:

- Views and rules are now functional thanks to extensive new code in the rewrite rules system from Jan Wieck. He also wrote a chapter on it for the *Programmer's Guide*.
- Jan also contributed a second procedural language, PL/pgSQL, to go with the original PL/pgTCL procedural language he contributed last release.
- We have optional multiple-byte character set support from Tatsuo Iishi to complement our existing locale support.
- Client/server communications has been cleaned up, with better support for asynchronous messages and interrupts thanks to Tom Lane.
- The parser will now perform automatic type coercion to match arguments to available operators and functions, and to match columns and expressions with target columns. This uses a generic mechanism which supports the type extensibility features of Postgres. There is a new chapter in the *User's Guide* which covers this topic.
- Three new data types have been added. Two types, `inet` and `cidr`, support various forms of IP network, subnet, and machine addressing. There is now an 8-byte integer type available on some platforms. See the chapter on data types in the *User's Guide* for details. A fourth type, `serial`, is now supported by the parser as an amalgam of the `int4` type, a sequence, and a unique index.
- Several more SQL92-compatible syntax features have been added, including `INSERT DEFAULT VALUES`
- The automatic configuration and installation system has received some attention, and should be more robust for more platforms than it has ever been.

36.10.1. Migration to v6.4

A dump/restore using pg_dump or pg_dumpall is required for those wishing to migrate data from any previous release of Postgres.

36.10.2. Changes

```
Bug Fixes
---------
Fix for a tiny memory leak in PQsetdb/PQfinish(Bryan)
Remove char2-16 data types, use char/varchar(Darren)
Pqfn not handles a NOTICE message(Anders)
Reduced busywaiting overhead for spinlocks with many backends (dg)
Stuck spinlock detection (dg)
Fix up "ISO-style" timespan decoding and encoding(Thomas)
Fix problem with table drop after rollback of transaction(Vadim)
Change error message and remove non-functional update
 message(Vadim)
Fix for COPY array checking
Fix for SELECT 1 UNION SELECT NULL
Fix for buffer leaks in large object calls(Pascal)
```

```
Change owner from oid to int4 type(Bruce)
Fix a bug in the oracle compatibility functions btrim() ltrim() and
 rtrim()
Fix for shared invalidation cache overflow(Massimo)
Prevent file descriptor leaks in failed COPY's(Bruce)
Fix memory leak in libpgtcl's pg_select(Constantin)
Fix problems with username/passwords over 8 characters(Tom)
Fix problems with handling of asynchronous NOTIFY in backend(Tom)
Fix of many bad system table entries(Tom)

Enhancements
------------
Upgrade ecpg and ecpglib,see src/interfaces/ecpc/ChangeLog(Michael)
Show the index used in an EXPLAIN(Zeugswetter)
EXPLAIN  invokes  rule system and shows plan(s) for rewritten
 queries(Jan)
Multi-byte awareness of many data types and functions, via
 configure(Tatsuo)
New configure --with-mb option(Tatsuo)
New initdb --pgencoding option(Tatsuo)
New createdb -E multibyte option(Tatsuo)
Select version(); now returns PostgreSQL version(Jeroen)
Libpq now allows asynchronous clients(Tom)
Allow cancel from client of backend query(Tom)
Psql now cancels query with Control-C(Tom)
Libpq users need not issue dummy queries to get NOTIFY
 messages(Tom)
NOTIFY now sends sender's PID, so you can tell whether it was your
 own(Tom)
PGresult struct now includes associated error message, if any(Tom)
Define "tz_hour" and "tz_minute" arguments to date_part()(Thomas)
Add routines to convert between varchar and bpchar(Thomas)
Add routines to allow sizing of varchar and bpchar into target
 columns(Thomas)
Add bit flags to support timezonehour and minute in data
 retrieval(Thomas)
Allow more variations on valid floating point numbers (e.g. ".1",
 "1e6")(Thomas)
Fixes for unary minus parsing with leading spaces(Thomas)
Implement TIMEZONE_HOUR, TIMEZONE_MINUTE per SQL92 specs(Thomas)
Check for and properly ignore FOREIGN KEY column
 constraints(Thomas)
Define USER as synonym for CURRENT_USER per SQL92 specs(Thomas)
Enable HAVING clause but no fixes elsewhere yet.
Make "char" type a synonym for "char(1)" (actually implemented as
 bpchar)(Thomas)
Save string type if specified for DEFAULT clause handling(Thomas)
Coerce operations involving different data types(Thomas)
Allow some index use for columns of different types(Thomas)
Add capabilities for automatic type conversion(Thomas)
Cleanups for large objects, so file is truncated on open(Peter)
Readline cleanups(Tom)
Allow psql  \f \ to make spaces as delimiter(Bruce)
Pass pg_attribute.atttypmod to the frontend for column field
 lengths(Tom,Bruce)
Msql compatibility library in /contrib(Aldrin)
Remove the requirement that ORDER/GROUP BY clause identifiers be
included in the target list(David)
```

```
Convert columns to match columns in UNION clauses(Thomas)
Remove fork()/exec() and only do fork()(Bruce)
Jdbc cleanups(Peter)
Show backend status on ps command line(only works on some
 platforms)(Bruce)
Pg_hba.conf now has a sameuser option in the database field
Make lo_unlink take oid param, not int4
New DISABLE_COMPLEX_MACRO for compilers that can't handle our
 macros(Bruce)
Libpgtcl now handles NOTIFY as a Tcl event, need not send dummy
 queries(Tom)
libpgtcl cleanups(Tom)
Add -error option to libpgtcl's pg_result command(Tom)
New locale patch, see docs/README/locale(Oleg)
Fix for pg_dump so CONSTRAINT and CHECK syntax is correct(ccb)
New contrib/lo code for large object orphan removal(Peter)
New psql command "SET CLIENT_ENCODING TO 'encoding'" for multi-
bytes
feature, see /doc/README.mb(Tatsuo)
/contrib/noupdate code to revoke update permission on a column
Libpq can now be compiled on win32(Magnus)
Add PQsetdbLogin() in libpq
New 8-byte integer type, checked by configure for OS
 support(Thomas)
Better support for quoted table/column names(Thomas)
Surround table and column names with double-quotes in
 pg_dump(Thomas)
PQreset() now works with passwords(Tom)
Handle case of GROUP BY target list column number out of
 range(David)
Allow UNION in subselects
Add auto-size to screen to \d? commands(Bruce)
Use UNION to show all \d? results in one query(Bruce)
Add \d? field search feature(Bruce)
Pg_dump issues fewer \connect requests(Tom)
Make pg_dump -z flag work better, document it in manual page(Tom)
Add HAVING clause with full support for subselects and
 unions(Stephan)
Full text indexing routines in contrib/fulltextindex(Maarten)
Transaction ids now stored in shared memory(Vadim)
New PGCLIENTENCODING when issuing COPY command(Tatsuo)
Support for SQL92 syntax "SET NAMES"(Tatsuo)
Support for LATIN2-5(Tatsuo)
Add UNICODE regression test case(Tatsuo)
Lock manager cleanup, new locking modes for LLL(Vadim)
Allow index use with OR clauses(Bruce)
Allows "SELECT NULL ORDER BY 1;"
Explain VERBOSE prints the plan, and now pretty-prints the plan to
the postmaster log file(Bruce)
Add Indices display to \d command(Bruce)
Allow GROUP BY on functions(David)
New pg_class.relkind for large objects(Bruce)
New way to send libpq NOTICE messages to a different location(Tom)
New \w write command to psql(Bruce)
New /contrib/findoidjoins scans oid columns to find join
 relationships(Bruce)
Allow binary-compatible indices to be considered when checking for
 valid
```

```
indices for restriction clauses containing a constant(Thomas)
New ISBN/ISSN code in /contrib/isbn_issn
Allow NOT LIKE, IN, NOT IN, BETWEEN, and NOT BETWEEN
 constraint(Thomas)
New rewrite system fixes many problems with rules and views(Jan)
 * Rules on relations work
 * Event qualifications on insert/update/delete work
 * New OLD variable to reference CURRENT, CURRENT will be remove in
 future
 * Update rules can reference NEW and OLD in rule qualifications/
actions
 * Insert/update/delete rules on views work
 * Multiple rule actions are now supported, surrounded by
 parentheses
 * Regular users can create views/rules on tables they have RULE
 permits
 * Rules and views inherit the permissions on the creator
 * No rules at the column level
 * No UPDATE NEW/OLD rules
 * New pg_tables, pg_indexes, pg_rules and pg_views system views
 * Only a single action on SELECT rules
 * Total rewrite overhaul, perhaps for 6.5
 * handle subselects
 * handle aggregates on views
 * handle insert into select from view works
System indexes are now multi-key(Bruce)
Oidint2, oidint4, and oidname types are removed(Bruce)
Use system cache for more system table lookups(Bruce)
New backend programming language PL/pgSQL in backend/pl(Jan)
New SERIAL data type, auto-creates sequence/index(Thomas)
Enable assert checking without a recompile(Massimo)
User lock enhancements(Massimo)
New setval() command to set sequence value(Massimo)
Auto-remove unix socket file on startup if no postmaster
 running(Massimo)
Conditional trace package(Massimo)
New UNLISTEN command(Massimo)
Psql and libpq now compile under win32 using win32.mak(Magnus)
Lo_read no longer stores trailing NULL(Bruce)
Identifiers are now truncated to 31 characters internally(Bruce)
Createuser options now availble on the command line
Code for 64-bit integer supported added, configure tested, int8
 type(Thomas)
Prevent file descriptor leaf from failed COPY(Bruce)
New pg_upgrade command(Bruce)
Updated /contrib directories(Massimo)
New CREATE TABLE DEFAULT VALUES statement available(Thomas)
New INSERT INTO TABLE DEFAULT VALUES statement available(Thomas)
New DECLARE and FETCH feature(Thomas)
libpq's internal structures now not exported(Tom)
Allow up to 8 key indexes(Bruce)
Remove ARCHIVE keyword, that is no longer used(Thomas)
pg_dump -n flag to supress quotes around indentifiers
disable system columns for views(Jan)
new INET and CIDR types for network addresses(TomH, Paul)
no more double quotes in psql output
pg_dump now dumps views(Terry)
new SET QUERY_LIMIT(Tatsuo,Jan)
```

```
Source Tree Changes
-------------------
/contrib cleanup(Jun)
Inline some small functions called for every row(Bruce)
Alpha/linux fixes
Hp/UX cleanups(Tom)
Multi-byte regression tests(Soonmyung.)
Remove --disabled options from configure
Define PGDOC to use POSTGRESDIR by default
Make regression optional
Remove extra braces code to pgindent(Bruce)
Add bsdi shared library support(Bruce)
New --without-CXX support configure option(Brook)
New FAQ_CVS
Update backend flowchart in tools/backend(Bruce)
Change atttypmod from int16 to int32(Bruce, Tom)
Getrusage() fix for platforms that do not have it(Tom)
Add PQconnectdb, PGUSER, PGPASSWORD to libpq man page
NS32K platform fixes(Phil Nelson, John Buller)
Sco 7/UnixWare 2.x fixes(Billy,others)
Sparc/Solaris 2.5 fixes(Ryan)
Pgbuiltin.3 is obsolete, move to doc files(Thomas)
Even more documention(Thomas)
Nextstep support(Jacek)
Aix support(David)
pginterface manual page(Bruce)
shared libraries all have version numbers
merged all OS-specific shared library defines into one file
smarter TCL/TK configuration checking(Billy)
smarter perl configuration(Brook)
configure uses supplied install-sh if no install script found(Tom)
new Makefile.shlib for shared library configuration(Tom)
```

36.11. Release 6.3.2

1998-04-07

This is a bugfix release for 6.3.x. Refer to the release notes for v6.3 for a more complete summary of new features.

Summary:

- Repairs automatic configuration support for some platforms, including Linux, from breakage inadvertently introduced in v6.3.1.
- Correctly handles function calls on the left side of BETWEEN and LIKE clauses.

A dump/restore is NOT required for those running 6.3 or 6.3.1. A 'make distclean', 'make', and 'make install' is all that is required. This last step should be performed while the postmaster is not running. You should re-link any custom applications that use Postgres libraries.

For upgrades from pre-v6.3 installations, refer to the installation and migration instructions for v6.3.

36.11.1. Changes

```
Configure detection improvements for tcl/tk(Brook Milligan, Alvin)
Manual page improvements(Bruce)
BETWEEN and LIKE fix(Thomas)
```

```
fix for psql \connect used by pg_dump(Oliver Elphick)
New odbc driver
pgaccess, version 0.86
qsort removed, now uses libc version, cleanups(Jeroen)
fix for buffer over-runs detected(Maurice Gittens)
fix for buffer overrun in libpgtcl(Randy Kunkee)
fix for UNION with DISTINCT or ORDER BY(Bruce)
gettimeofday configure check(Doug Winterburn)
Fix "indexes not used" bug(Vadim)
docs additions(Thomas)
Fix for backend memory leak(Bruce)
libreadline cleanup(Erwan MAS)
Remove DISTDIR(Bruce)
Makefile dependency cleanup(Jeroen van Vianen)
ASSERT fixes(Bruce)
```

36.12. Release 6.3.1

Released 1998-03-23. Summary:

- Additional support for multi-byte character sets.
- Repair byte ordering for mixed-endian clients and servers.
- Minor updates to allowed SQL syntax.
- Improvements to the configuration autodetection for installation.

A dump/restore is NOT required for those running 6.3. A 'make distclean', 'make', and 'make install' is all that is required. This last step should be performed while the postmaster is not running. You should re-link any custom applications that use Postgres libraries.

For upgrades from pre-v6.3 installations, refer to the installation and migration instructions for v6.3.

36.12.1. Changes

```
ecpg cleanup/fixes, now version 1.1(Michael Meskes)
pg_user cleanup(Bruce)
large object fix for pg_dump and tclsh (alvin)
LIKE fix for multiple adjacent underscores
fix for redefining builtin functions(Thomas)
ultrix4 cleanup
upgrade to pg_access 0.83
updated CLUSTER manual page
multi-byte character set support, see doc/README.mb(Tatsuo)
configure --with-pgport fix
pg_ident fix
big-endian fix for backend communications(Kataoka)
SUBSTR() and substring() fix(Jan)
several jdbc fixes(Peter)
libpgtcl improvements, see libptcl/README(Randy Kunkee)
Fix for "Datasize = 0" error(Vadim)
Prevent \do from wrapping(Bruce)
Remove duplicate Russian character set entries
Sunos4 cleanup
Allow optional TABLE keyword in LOCK and SELECT INTO(Thomas)
CREATE SEQUENCE options to allow a negative integer(Thomas)
Add "PASSWORD" as an allowed column identifier(Thomas)
```

```
Add checks for UNION target fields(Bruce)
Fix Alpha port(Dwayne Bailey)
Fix for text arrays containing quotes(Doug Gibson)
Solaris compile fix(Albert Chin-A-Young)
Better identify tcl and tk libs and includes(Bruce)
```

36.13. Release 6.3

Released 1998-03-01. There are *many* new features and improvements in this release. Here is a brief, incomplete summary:

- Many new SQL features, including full SQL92 subselect capability (everything is here but target-list subselects).
- Support for client-side environment variables to specify time zone and date style.
- Socket interface for client/server connection. This is the default now so you may need to start postmaster with the `-i` flag.
- Better password authorization mechanisms. Default table permissions have changed.
- Old-style *time travel* has been removed. Performance has been improved.

Note

Bruce Momjian wrote the following notes to introduce the new release.

There are some general 6.3 issues that I want to mention. These are only the big items that can not be described in one sentence. A review of the detailed changes list is still needed.

First, we now have subselects. Now that we have them, I would like to mention that without subselects, SQL is a very limited language. Subselects are a major feature, and you should review your code for places where subselects provide a better solution for your queries. I think you will find that there are more uses for subselects than you may think. Vadim has put us on the big SQL map with subselects, and fully functional ones too. The only thing you can't do with subselects is to use them in the target list.

Second, 6.3 uses unix domain sockets rather than TCP/IP by default. To enable connections from other machines, you have to use the new postmaster -i option, and of course edit pg_hba.conf. Also, for this reason, the format of pg_hba.conf has changed.

Third, char() fields will now allow faster access than varchar() or text. Specifically, the text and varchar() have a penalty for access to any columns after the first column of this type. char() used to also have this access penalty, but it no longer does. This may suggest that you redesign some of your tables, especially if you have short character columns that you have defined as varchar() or text. This and other changes make 6.3 even faster than earlier releases.

We now have passwords definable independent of any Unix file. There are new SQL USER commands. See the pg_hba.conf manual page for more information. There is a new table, pg_shadow, which is used to store user information and user passwords, and it by default only SELECT-able by the postgres super-user. pg_user is now a view of pg_shadow, and is SELECT-able by PUBLIC. You should keep using pg_user in your application without changes.

User-created tables now no longer have SELECT permission to PUBLIC by default. This was done because the ANSI standard requires it. You can of course GRANT any permissions you want after the table is created. System tables continue to be SELECT-able by PUBLIC.

We also have real deadlock detection code. No more sixty-second timeouts. And the new locking code implements a FIFO better, so there should be less resource starvation during heavy use.

Many complaints have been made about inadequate documenation in previous releases. Thomas has put much effort into many new manuals for this release. Check out the doc/ directory.

For performance reasons, time travel is gone, but can be implemented using triggers (see pgsql/contrib/spi/README). Please check out the new \d command for types, operators, etc. Also, views have their own permissions now, not based on the underlying tables, so permissions on them have to be set separately. Check /pgsql/interfaces for some new ways to talk to Postgres.

This is the first release that really required an explanation for existing users. In many ways, this was necessary because the new release removes many limitations, and the work-arounds people were using are no longer needed.

36.13.1. Migration to v6.3

A dump/restore using pg_dump or pg_dumpall is required for those wishing to migrate data from any previous release of Postgres.

36.13.2. Changes

```
Bug Fixes
---------
Fix binary cursors broken by MOVE implementation(Vadim)
Fix for tcl library crash(Jan)
Fix for array handling, from Gerhard Hintermayer
Fix acl error, and remove duplicate pqtrace(Bruce)
Fix psql \e for empty file(Bruce)
Fix for textcat on varchar() fields(Bruce)
Fix for DBT Sendproc (Zeugswetter Andres)
Fix vacuum analyze syntax problem(Bruce)
Fix for international identifiers(Tatsuo)
Fix aggregates on inherited tables(Bruce)
Fix substr() for out-of-bounds data
Fix for select 1=1 or 2=2, select 1=1 and 2=2, and select sum(2+2)
(Bruce)
Fix notty output to show status result.  -q option still turns it
 off(Bruce)
Fix for count(*), aggs with views and multiple tables and sum(3)
(Bruce)
Fix cluster(Bruce)
Fix for PQtrace start/stop several times(Bruce)
Fix a variety of locking problems like newer lock waiters getting
 lock before older waiters, and having readlock people not share
 locks if a writer is waiting for a lock, and waiting writers not
 getting priority over waiting readers(Bruce)
Fix crashes in psql when executing queries from external
 files(James)
Fix problem with multiple order by columns, with the first one
 having
 NULL values(Jeroen)
Use correct hash table support functions for float8 and
 int4(Thomas)
Re-enable JOIN= option in CREATE OPERATOR statement (Thomas)
Change precedence for boolean operators to match expected
 behavior(Thomas)
Generate elog(ERROR) on over-large integer(Bruce)
Allow multiple-argument functions in constraint clauses(Thomas)
Check boolean input literals for 'true','false','yes','no','1','0'
 and throw elog(ERROR) if unrecognized(Thomas)
Major large objects fix
Fix for GROUP BY showing duplicates(Vadim)
Fix for index scans in MergeJion(Vadim)
```

```
Enhancements
------------
Subselects with EXISTS, IN, ALL, ANY keywords (Vadim, Bruce,
 Thomas)
New User Manual(Thomas, others)
Speedup by inlining some frequently-called functions
Real deadlock detection, no more timeouts(Bruce)
Add SQL92 "constants" CURRENT_DATE, CURRENT_TIME,
 CURRENT_TIMESTAMP,
 CURRENT_USER(Thomas)
Modify constraint syntax to be SQL92-compliant(Thomas)
Implement SQL92 PRIMARY KEY and UNIQUE clauses using
 indices(Thomas)
Recognize SQL92 syntax for FOREIGN KEY. Throw elog notice(Thomas)
Allow NOT NULL UNIQUE constraint clause (each allowed separately
 before)(Thomas)
Allow Postgres-style casting ("::") of non-constants(Thomas)
Add support for SQL3 TRUE and FALSE boolean constants(Thomas)
Support SQL92 syntax for IS TRUE/IS FALSE/IS NOT TRUE/IS NOT
 FALSE(Thomas)
Allow shorter strings for boolean literals (e.g. "t", "tr", "tru")
(Thomas)
Allow SQL92 delimited identifiers(Thomas)
Implement SQL92 binary and hexadecimal string decoding (b'10' and
 x'1F')(Thomas)
Support SQL92 syntax for type coercion of literal strings
 (e.g. "DATETIME 'now'")(Thomas)
Add conversions for int2, int4, and OID types to and from
 text(Thomas)
Use shared lock when building indices(Vadim)
Free memory allocated for an user query inside transaction block
 after
 this query is done, was turned off in <= 6.2.1(Vadim)
New SQL statement CREATE PROCEDURAL LANGUAGE(Jan)
New Postgres Procedural Language (PL) backend interface(Jan)
Rename pg_dump -H option to -h(Bruce)
Add Java support for passwords, European dates(Peter)
Use indices for LIKE and ~, !~ operations(Bruce)
Add hash functions for datetime and timespan(Thomas)
Time Travel removed(Vadim, Bruce)
Add paging for \d and \z, and fix \i(Bruce)
Add Unix domain socket support to backend and to frontend
 library(Goran)
Implement CREATE DATABASE/WITH LOCATION and initlocation
 utility(Thomas)
Allow more SQL92 and/or Postgres reserved words as column
 identifiers(Thomas)
Augment support for SQL92 SET TIME ZONE...(Thomas)
SET/SHOW/RESET TIME ZONE uses TZ backend environment
 variable(Thomas)
Implement SET keyword = DEFAULT and SET TIME ZONE DEFAULT(Thomas)
Enable SET TIME ZONE using TZ environment variable(Thomas)
Add PGDATESTYLE environment variable to frontend and backend
 initialization(Thomas)
Add PGTZ, PGCOSTHEAP, PGCOSTINDEX, PGRPLANS, PGGEQO
 frontend library initialization environment variables(Thomas)
```

```
Regression tests time zone automatically set with "setenv PGTZ
 PST8PDT"(Thomas)
Add pg_description table for info on tables, columns, operators,
 types, and
 aggregates(Bruce)
Increase 16 char limit on system table/index names to 32
 characters(Bruce)
Rename system indices(Bruce)
Add 'GERMAN' option to SET DATESTYLE(Thomas)
Define an "ISO-style" timespan output format with "hh:mm:ss"
 fields(Thomas)
Allow fractional values for delta times (e.g. '2.5 days')(Thomas)
Validate numeric input more carefully for delta times(Thomas)
Implement day of year as possible input to date_part()(Thomas)
Define timespan_finite() and text_timespan() functions(Thomas)
Remove archive stuff(Bruce)
Allow for a pg_password authentication database that is separate
 from
 the system password file(Todd)
Dump ACLs, GRANT, REVOKE permissions(Matt)
Define text, varchar, and bpchar string length functions(Thomas)
Fix Query handling for inheritance, and cost computations(Bruce)
Implement CREATE TABLE/AS SELECT (alternative to SELECT/INTO)
(Thomas)
Allow NOT, IS NULL, IS NOT NULL in constraints(Thomas)
Implement UNIONs for SELECT(Bruce)
Add UNION, GROUP, DISTINCT to INSERT(Bruce)
varchar() stores only necessary bytes on disk(Bruce)
Fix for BLOBs(Peter)
Mega-Patch for JDBC...see README_6.3 for list of changes(Peter)
Remove unused "option" from PQconnectdb()
New LOCK command and lock manual page describing deadlocks(Bruce)
Add new psql \da, \dd, \df, \do, \dS, and \dT commands(Bruce)
Enhance psql \z to show sequences(Bruce)
Show NOT NULL and DEFAULT in psql \d table(Bruce)
New psql .psqlrc file startup(Andrew)
Modify sample startup script in contrib/linux to show
 syslog(Thomas)
New types for IP and MAC addresses in contrib/ip_and_mac(TomH)
Unix system time conversions with date/time types in contrib/
unixdate(Thomas)
Update of contrib stuff(Massimo)
Add Unix socket support to DBD::Pg(Goran)
New python interface (PyGreSQL 2.0)(D'Arcy)
New frontend/backend protocol has a version number, network byte
 order(Phil)
Security features in pg_hba.conf enhanced and documented, many
 cleanups(Phil)
CHAR() now faster access than VARCHAR() or TEXT
ecpg embedded SQL preprocessor
Reduce system column overhead(Vadmin)
Remove pg_time table(Vadim)
Add pg_type attribute to identify types that need length (bpchar,
 varchar)
Add report of offending line when COPY command fails
Allow VIEW permissions to be set separately from the underlying
 tables.
 For security, use GRANT/REVOKE on views as appropriate(Jan)
```

```
Tables now have no default GRANT SELECT TO PUBLIC.  You must
 explicitly grant such permissions.
Clean up tutorial examples(Darren)

Source Tree Changes
-------------------
Add new html development tools, and flow chart in /tools/backend
Fix for SCO compiles
Stratus computer port Robert Gillies
Added support for shlib for BSD44_derived & i386_solaris
Make configure more automated(Brook)
Add script to check regression test results
Break parser functions into smaller files, group together(Bruce)
Rename heap_create to heap_create_and_catalog, rename heap_creatr
 to heap_create()(Bruce)
Sparc/Linux patch for locking(TomS)
Remove PORTNAME and reorganize port-specific stuff(Marc)
Add optimizer README file(Bruce)
Remove some recursion in optimizer and clean up some code
 there(Bruce)
Fix for NetBSD locking(Henry)
Fix for libptcl make(Tatsuo)
AIX patch(Darren)
Change IS TRUE, IS FALSE, ... to expressions using "=" rather than
 function calls to istrue() or isfalse() to allow
 optimization(Thomas)
Various fixes NetBSD/Sparc related(TomH)
Alpha linux locking(Travis,Ryan)
Change elog(WARN) to elog(ERROR)(Bruce)
FAQ for FreeBSD(Marc)
Bring in the PostODBC source tree as part of our standard
 distribution(Marc)
A minor patch for HP/UX 10 vs 9(Stan)
New pg_attribute.atttypmod for type-specific info like varchar
 length(Bruce)
Unixware patches(Billy)
New i386 'lock' for spin lock asm(Billy)
Support for multiplexed backends is removed
Start an OpenBSD port
Start an AUX port
Start a Cygnus port
Add string functions to regression suite(Thomas)
Expand a few function names formerly truncated to 16
 characters(Thomas)
Remove un-needed malloc() calls and replace with palloc()(Bruce)
```

36.14. Release 6.2.1

1997-10-17

v6.2.1 is a bug-fix and usability release on v6.2.

Summary:

- Allow strings to span lines, per SQL92.
- Include example trigger function for inserting user names on table updates.

This is a minor bug-fix release on v6.2. For upgrades from pre-v6.2 systems, a full dump/reload is required. Refer to the v6.2 release notes for instructions.

36.14.1. Migration from v6.2 to v6.2.1

This is a minor bug-fix release. A dump/reload is not required from v6.2, but is required from any release prior to v6.2.

In upgrading from v6.2, if you choose to dump/reload you will find that avg(money) is now calculated correctly. All other bug fixes take effect upon updating the executables.

Another way to avoid dump/reload is to use the following SQL command from psql to update the existing system table:

```
update pg_aggregate set aggfinalfn = 'cash_div_flt8'
 where aggname = 'avg' and aggbasetype = 790;
```

This will need to be done to every existing database, including template1.

36.14.2. Changes

```
Allow TIME and TYPE column names(Thomas)
Allow larger range of true/false as boolean values(Thomas)
Support output of "now" and "current"(Thomas)
Handle DEFAULT with INSERT of NULL properly(Vadim)
Fix for relation reference counts problem in buffer manager(Vadim)
Allow strings to span lines, like ANSI(Thomas)
Fix for backward cursor with ORDER BY(Vadim)
Fix avg(cash) computation(Thomas)
Fix for specifying a column twice in ORDER/GROUP BY(Vadim)
Documented new libpq function to return affected rows,
 PQcmdTuples(Bruce)
Trigger function for inserting user names for INSERT/UPDATE(Brook
 Milligan)
```

36.15. Release 6.2

1997-10-02

A dump/restore is required for those wishing to migrate data from previous releases of Postgres.

36.15.1. Migration from v6.1 to v6.2

This migration requires a complete dump of the 6.1 database and a restore of the database in 6.2.

Note that the pg_dump and pg_dumpall utility from 6.2 should be used to dump the 6.1 database.

36.15.2. Migration from v1.x to v6.2

Those migrating from earlier 1.* releases should first upgrade to 1.09 because the COPY output format was improved from the 1.02 release.

36.15.3. Changes

```
Bug Fixes
---------
Fix problems with pg_dump for inheritance, sequences, archive
 tables(Bruce)
```

```
Fix compile errors on overflow due to shifts, unsigned, and bad
 prototypes
  from Solaris(Diab Jerius)
Fix bugs in geometric line arithmetic (bad intersection
 calculations)(Thomas)
Check for geometric intersections at endpoints to avoid rounding
 ugliness(Thomas)
Catch non-functional delete attempts(Vadim)
Change time function names to be more consistent(Michael
 Reifenberg)
Check for zero divides(Michael Reifenberg)
Fix very old bug which made tuples changed/inserted by a commnd
 visible to the command itself (so we had multiple update of
 updated tuples, etc)(Vadim)
Fix for SELECT null, 'fail' FROM pg_am (Patrick)
SELECT NULL as EMPTY_FIELD now allowed(Patrick)
Remove un-needed signal stuff from contrib/pginterface
Fix OR (where x != 1 or x isnull didn't return tuples with x NULL)
 (Vadim)
Fix time_cmp function (Vadim)
Fix handling of functions with non-attribute first argument in
 WHERE clauses (Vadim)
Fix GROUP BY when order of entries is different from order
 in target list (Vadim)
Fix pg_dump for aggregates without sfunc1 (Vadim)

Enhancements
------------
Default genetic optimizer GEQO parameter is now 8(Bruce)
Allow use parameters in target list having aggregates in
 functions(Vadim)
Added JDBC driver as an interface(Adrian & Peter)
pg_password utility
Return number of tuples inserted/affected by INSERT/UPDATE/DELETE
 etc.(Vadim)
Triggers implemented with CREATE TRIGGER (SQL3)(Vadim)
SPI (Server Programming Interface) allows execution of queries
 inside
 C-functions (Vadim)
NOT NULL implemented (SQL92)(Robson Paniago de Miranda)
Include reserved words for string handling, outer joins, and
 unions(Thomas)
Implement extended comments ("/* ... */") using exclusive
 states(Thomas)
Add "//" single-line comments(Bruce)
Remove some restrictions on characters in operator names(Thomas)
DEFAULT and CONSTRAINT for tables implemented (SQL92)(Vadim &
 Thomas)
Add text concatenation operator and function (SQL92)(Thomas)
Support WITH TIME ZONE syntax (SQL92)(Thomas)
Support INTERVAL unit TO unit syntax (SQL92)(Thomas)
Define types DOUBLE PRECISION, INTERVAL, CHARACTER,
 and CHARACTER VARYING (SQL92)(Thomas)
Define type FLOAT(p) and rudimentary DECIMAL(p,s), NUMERIC(p,s)
 (SQL92)(Thomas)
Define EXTRACT(), POSITION(), SUBSTRING(), and TRIM() (SQL92)
(Thomas)
```

```
Define CURRENT_DATE, CURRENT_TIME, CURRENT_TIMESTAMP (SQL92)
(Thomas)
Add syntax and warnings for UNION, HAVING, INNER and OUTER JOIN
 (SQL92)(Thomas)
Add more reserved words, mostly for SQL92 compliance(Thomas)
Allow hh:mm:ss time entry for timespan/reltime types(Thomas)
Add center() routines for lseg, path, polygon(Thomas)
Add distance() routines for circle-polygon, polygon-polygon(Thomas)
Check explicitly for points and polygons contained within polygons
 using an axis-crossing algorithm(Thomas)
Add routine to convert circle-box(Thomas)
Merge conflicting operators for different geometric data
 types(Thomas)
Replace distance operator "<===>" with "<->"(Thomas)
Replace "above" operator "!^" with ">^" and "below" operator "!|"
 with "<^"(Thomas)
Add routines for text trimming on both ends, substring, and string
 position(Thomas)
Added conversion routines circle(box) and poly(circle)(Thomas)
Allow internal sorts to be stored in memory rather than in
 files(Bruce & Vadim)
Allow functions and operators on internally-identical types to
 succeed(Bruce)
Speed up backend startup after profiling analysis(Bruce)
Inline frequently called functions for performance(Bruce)
Reduce open() calls(Bruce)
psql:  Add PAGER for \h and \?,\C fix
Fix for psql pager when no tty(Bruce)
New entab utility(Bruce)
General trigger functions for referential integrity (Vadim)
General trigger functions for time travel (Vadim)
General trigger functions for AUTOINCREMENT/IDENTITY feature
 (Vadim)
MOVE implementation (Vadim)

Source Tree Changes
-------------------
HPUX 10 patches (Vladimir Turin)
Added SCO support, (Daniel Harris)
mkLinux patches (Tatsuo Ishii)
Change geometric box terminology from "length" to "width"(Thomas)
Deprecate temporary unstored slope fields in geometric code(Thomas)
Remove restart instructions from INSTALL(Bruce)
Look in /usr/ucb first for install(Bruce)
Fix c++ copy example code(Thomas)
Add -o to psql manual page(Bruce)
Prevent relname unallocated string length from being copied into
 database(Bruce)
Cleanup for NAMEDATALEN use(Bruce)
Fix pg_proc names over 15 chars in output(Bruce)
Add strNcpy() function(Bruce)
remove some (void) casts that are unnecessary(Bruce)
new interfaces directory(Marc)
Replace fopen() calls with calls to fd.c functions(Bruce)
Make functions static where possible(Bruce)
enclose unused functions in #ifdef NOT_USED(Bruce)
Remove call to difftime() in timestamp support to fix SunOS(Bruce &
 Thomas)
```

```
Changes for Digital Unix
Portability fix for pg_dumpall(Bruce)
Rename pg_attribute.attnvals to attdisbursion(Bruce)
"intro/unix" manual page now "pgintro"(Bruce)
"built-in" manual page now "pgbuiltin"(Bruce)
"drop" manual page now "drop_table"(Bruce)
Add "create_trigger", "drop_trigger" manual pages(Thomas)
Add constraints regression test(Vadim & Thomas)
Add comments syntax regression test(Thomas)
Add PGINDENT and support program(Bruce)
Massive commit to run PGINDENT on all *.c and *.h files(Bruce)
Files moved to /src/tools directory(Bruce)
SPI and Trigger programming guides (Vadim & D'Arcy)
```

36.16. Release 6.1.1

36.16.1. Migration from v6.1 to v6.1.1

1997-07-22

This is a minor bug-fix release. A dump/reload is not required from v6.1, but is required from any release prior to v6.1. Refer to the release notes for v6.1 for more details.

36.16.2. Changes

```
fix for SET with options (Thomas)
allow pg_dump/pg_dumpall to preserve ownership of all tables/
objects(Bruce)
new psql \connect option allows changing usernames without changing
 databases
fix for initdb --debug option(Yoshihiko Ichikawa))
lextest cleanup(Bruce)
hash fixes(Vadim)
fix date/time month boundary arithmetic(Thomas)
fix timezone daylight handling for some ports(Thomas, Bruce,
 Tatsuo)
timestamp overhauled to use standard functions(Thomas)
other code cleanup in date/time routines(Thomas)
psql's \d now case-insensitive(Bruce)
psql's backslash commands can now have trailing semicolon(Bruce)
fix memory leak in psql when using \g(Bruce)
major fix for endian handling of communication to server(Thomas,
 Tatsuo)
Fix for Solaris assembler and include files(Yoshihiko Ichikawa)
allow underscores in usernames(Bruce)
pg_dumpall now returns proper status, portability fix(Bruce)
```

36.17. Release 6.1

1997-06-08

The regression tests have been adapted and extensively modified for the v6.1 release of Postgres.

Three new data types (datetime, timespan, and circle) have been added to the native set of Postgres types. Points, boxes, paths, and polygons have had their output formats made consistant across the

data types. The polygon output in misc.out has only been spot-checked for correctness relative to the original regression output.

Postgres v6.1 introduces a new, alternate optimizer which uses *genetic* algorithms. These algorithms introduce a random behavior in the ordering of query results when the query contains multiple qualifiers or multiple tables (giving the optimizer a choice on order of evaluation). Several regression tests have been modified to explicitly order the results, and hence are insensitive to optimizer choices. A few regression tests are for data types which are inherently unordered (e.g. points and time intervals) and tests involving those types are explicitly bracketed with `set geqo to 'off'` and `reset geqo`.

The interpretation of array specifiers (the curly braces around atomic values) appears to have changed sometime after the original regression tests were generated. The current `./expected/*.out` files reflect this new interpretation, which may not be correct!

The float8 regression test fails on at least some platforms. This is due to differences in implementations of pow() and exp() and the signaling mechanisms used for overflow and underflow conditions.

The "random" results in the random test should cause the "random" test to be "failed", since the regression tests are evaluated using a simple diff. However, "random" does not seem to produce random results on my test machine (Linux/gcc/i686).

36.17.1. Migration to v6.1

This migration requires a complete dump of the 6.0 database and a restore of the database in 6.1.

Those migrating from earlier 1.* releases should first upgrade to 1.09 because the COPY output format was improved from the 1.02 release.

36.17.2. Changes

```
Bug Fixes
---------
packet length checking in library routines
lock manager priority patch
check for under/over flow of float8(Bruce)
multi-table join fix(Vadim)
SIGPIPE crash fix(Darren)
large object fixes(Sven)
allow btree indexes to handle NULLs(Vadim)
timezone fixes(D'Arcy)
select SUM(x) can return NULL on no rows(Thomas)
internal optimizer, executor bug fixes(Vadim)
fix problem where inner loop in < or <= has no rows(Vadim)
prevent re-commuting join index clauses(Vadim)
fix join clauses for multiple tables(Vadim)
fix hash, hashjoin for arrays(Vadim)
fix btree for abstime type(Vadim)
large object fixes(Raymond)
fix buffer leak in hash indices (Vadim)
fix rtree for use in inner scan (Vadim)
fix gist for use in inner scan, cleanups (Vadim, Andrea)
avoid unnecessary local buffers allocation (Vadim, Massimo)
fix local buffers leak in transaction aborts (Vadim)
fix file manager memmory leaks, cleanups (Vadim, Massimo)
fix storage manager memmory leaks (Vadim)
fix btree duplicates handling (Vadim)
fix deleted tuples re-incarnation caused by vacuum (Vadim)
```

```
fix SELECT varchar()/char() INTO TABLE made zero-length
 fields(Bruce)
many psql, pg_dump, and libpq memory leaks fixed using Purify
 (Igor)

Enhancements
------------
attribute optimization statistics(Bruce)
much faster new btree bulk load code(Paul)
BTREE UNIQUE added to bulk load code(Vadim)
new lock debug code(Massimo)
massive changes to libpg++(Leo)
new GEQO optimizer speeds table multi-table optimization(Martin)
new WARN message for non-unique insert into unique key(Marc)
update x=-3, no spaces, now valid(Bruce)
remove case-sensitive identifier handling(Bruce,Thomas,Dan)
debug backend now pretty-prints tree(Darren)
new Oracle character functions(Edmund)
new plaintext password functions(Dan)
no such class or insufficient privilege changed to distinct
 messages(Dan)
new ANSI timestamp function(Dan)
new ANSI Time and Date types (Thomas)
move large chunks of data in backend(Martin)
multi-column btree indexes(Vadim)
new SET var TO value command(Martin)
update transaction status on reads(Dan)
new locale settings for character types(Oleg)
new SEQUENCE serial number generator(Vadim)
GROUP BY function now possible(Vadim)
re-organize regression test(Thomas,Marc)
new optimizer operation weights(Vadim)
new psql \z grant/permit option(Marc)
new MONEY data type(D'Arcy,Thomas)
tcp socket communication speed improved(Vadim)
new VACUUM option for attribute statistics, and for certain columns
 (Vadim)
many geometric type improvements(Thomas,Keith)
additional regression tests(Thomas)
new datestyle variable(Thomas,Vadim,Martin)
more comparison operators for sorting types(Thomas)
new conversion functions(Thomas)
new more compact btree format(Vadim)
allow pg_dumpall to preserve database ownership(Bruce)
new SET GEQO=# and R_PLANS variable(Vadim)
old (!GEQO) optimizer can use right-sided plans (Vadim)
typechecking improvement in SQL parser(Bruce)
new SET, SHOW, RESET commands(Thomas,Vadim)
new \connect database USER option
new destroydb -i option (Igor)
new \dt and \di psql commands (Darren)
SELECT "\n" now escapes newline (A. Duursma)
new geometry conversion functions from old format (Thomas)

Source tree changes
-------------------
new configuration script(Marc)
readline configuration option added(Marc)
```

```
OS-specific configuration options removed(Marc)
new OS-specific template files(Marc)
no more need to edit Makefile.global(Marc)
re-arrange include files(Marc)
nextstep patches (Gregor Hoffleit)
removed WIN32-specific code(Bruce)
removed postmaster -e option, now only postgres -e option (Bruce)
merge duplicate library code in front/backends(Martin)
now works with eBones, international Kerberos(Jun)
more shared library support
c++ include file cleanup(Bruce)
warn about buggy flex(Bruce)
DG-UX, Ultrix, Irix, AIX portability fixes
```

36.18. Release v6.0

1997-01-29

A dump/restore is required for those wishing to migrate data from previous releases of Postgres.

36.18.1. Migration from v1.09 to v6.0

This migration requires a complete dump of the 1.09 database and a restore of the database in 6.0.

36.18.2. Migration from pre-v1.09 to v6.0

Those migrating from earlier 1.* releases should first upgrade to 1.09 because the COPY output format was improved from the 1.02 release.

36.18.3. Changes

```
Bug Fixes
---------
ALTER TABLE bug - running postgress process needs to re-read table
 definition
Allow vacuum to be run on one table or entire database(Bruce)
Array fixes
Fix array over-runs of memory writes(Kurt)
Fix elusive btree range/non-range bug(Dan)
Fix for hash indexes on some types like time and date
Fix for pg_log size explosion
Fix permissions on lo_export()(Bruce)
Fix unitialized reads of memory(Kurt)
Fixed ALTER TABLE ... char(3) bug(Bruce)
Fixed a few small memory leaks
Fixed EXPLAIN handling of options and changed full_path option name
Fixed output of group acl permissions
Memory leaks (hunt and destroy with tools like Purify(Kurt)
Minor improvements to rules system
NOTIFY fixes
New asserts for run-checking
Overhauled parser/analyze code to properly report errors and
 increase speed
Pg_dump -d now handles NULL's properly(Bruce)
Prevent SELECT NULL from crashing server (Bruce)
Properly report errors when INSERT ... SELECT columns did not match
Properly report errors when insert column names were not correct
```

```
Psql \g filename now works(Bruce)
Psql fixed problem with multiple statements on one line with
 multiple outputs
Removed duplicate system oid's
SELECT * INTO TABLE . GROUP/ORDER BY gives unlink error if table
 exists(Bruce)
Several fixes for queries that crashed the backend
Starting quote in insert string errors(Bruce)
Submitting an empty query now returns empty status, not just " "
 query(Bruce)

Enhancements
------------
Add EXPLAIN manual page(Bruce)
Add UNIQUE index capability(Dan)
Add hostname/user level access control rather than just hostname
 and user
Add synonym of != for <>(Bruce)
Allow "select oid,* from table"
Allow BY,ORDER BY to specify columns by number, or by non-alias
 table.column(Bruce)
Allow COPY from the frontend(Bryan)
Allow GROUP BY to use alias column name(Bruce)
Allow actual compression, not just reuse on the same page(Vadim)
Allow installation-configuration option to auto-add all local
 users(Bryan)
Allow libpq to distinguish between text value '' and null(Bruce)
Allow non-postgres users with createdb privs to destroydb's
Allow restriction on who can create C functions(Bryan)
Allow restriction on who can do backend COPY(Bryan)
Can shrink tables, pg_time and pg_log(Vadim & Erich)
Change debug level 2 to print queries only, changed debug heading
 layout(Bruce)
Change default decimal constant representation from float4 to
 float8(Bruce)
European date format now set when postmaster is started
Execute lowercase function names if not found with exact case
Fixes for aggregate/GROUP processing, allow 'select
 sum(func(x),sum(x+y) from z'
Gist now included in the distrubution(Marc)
Idend authentication of local users(Bryan)
Implement BETWEEN qualifier(Bruce)
Implement IN qualifier(Bruce)
Libpq has PQgetisnull()(Bruce)
Libpq++ improvements
New options to initdb(Bryan)
Pg_dump allow dump of oid's(Bruce)
Pg_dump create indexes after tables are loaded for speed(Bruce)
Pg_dumpall dumps all databases, and the user table
Pginterface additions for NULL values(Bruce)
Prevent postmaster from being run as root
Psql \h and \? is now readable(Bruce)
Psql allow backslashed, semicolons anywhere on the line(Bruce)
Psql changed command prompt for lines in query or in quotes(Bruce)
Psql char(3) now displays as (bp)char in \d output(Bruce)
Psql return code now more accurate(Bryan?)
Psql updated help syntax(Bruce)
Re-visit and fix vacuum(Vadim)
```

```
Reduce size of regression diffs, remove timezone name
 difference(Bruce)
Remove compile-time parameters to enable binary
 distributions(Bryan)
Reverse meaning of HBA masks(Bryan)
Secure Authentication of local users(Bryan)
Speed up vacuum(Vadim)
Vacuum now had VERBOSE option(Bruce)

Source tree changes
-------------------
All functions now have prototypes that are compared against the
 calls
Allow asserts to be disabled easly from Makefile.global(Bruce)
Change oid constants used in code to #define names
Decoupled sparc and solaris defines(Kurt)
Gcc -Wall compiles cleanly with warnings only from unfixable
 constructs
Major include file reorganization/reduction(Marc)
Make now stops on compile failure(Bryan)
Makefile restructuring(Bryan, Marc)
Merge bsdi_2_1 to bsdi(Bruce)
Monitor program removed
Name change from Postgres95 to PostgreSQL
New config.h file(Marc, Bryan)
PG_VERSION now set to 6.0 and used by postmaster
Portability additions, including Ultrix, DG/UX, AIX, and Solaris
Reduced the number of #define's, centeralized #define's
Remove duplicate OIDS in system tables(Dan)
Remove duplicate system catalog info or report mismatches(Dan)
Removed many os-specific #define's
Restructured object file generation/location(Bryan, Marc)
Restructured port-specific file locations(Bryan, Marc)
Unused/uninialized variables corrected
```

36.19. Release v1.09

Unknown Sorry, we stopped keeping track of changes from 1.02 to 1.09. Some of the changes listed in 6.0 were actually included in the 1.02.1 to 1.09 releases.

36.20. Release v1.02

36.20.1. Migration from v1.02 to v1.02.1

1996-08-01

Here is a new migration file for 1.02.1. It includes the 'copy' change and a script to convert old ascii files.

Note

The following notes are for the benefit of users who want to migrate databases from postgres95 1.01 and 1.02 to postgres95 1.02.1.

If you are starting afresh with postgres95 1.02.1 and do not need to migrate old databases, you do not need to read any further.

In order to upgrade older postgres95 version 1.01 or 1.02 databases to version 1.02.1, the following steps are required:

1. Start up a new 1.02.1 postmaster

2. Add the new built-in functions and operators of 1.02.1 to 1.01 or 1.02 databases. This is done by running the new 1.02.1 server against your own 1.01 or 1.02 database and applying the queries attached at the end of thie file. This can be done easily through psql. If your 1.01 or 1.02 database is named "testdb" and you have cut the commands from the end of this file and saved them in addfunc.sql:

   ```
   % psql testdb -f addfunc.sql
   ```

   Those upgrading 1.02 databases will get a warning when executing the last two statements in the file because they are already present in 1.02. This is not a cause for concern.

36.20.2. Dump/Reload Procedure

If you are trying to reload a pg_dump or text-mode 'copy tablename to stdout' generated with a previous version, you will need to run the attached sed script on the ASCII file before loading it into the database. The old format used '.' as end-of-data, while '\.' is now the end-of-data marker. Also, empty strings are now loaded in as '' rather than NULL. See the copy manual page for full details.

```
sed 's/^\.$/\\./g' <in_file >out_file
```

If you are loading an older binary copy or non-stdout copy, there is no end-of-data character, and hence no conversion necessary.

```
-- following lines added by agc to reflect the case-insensitive
-- regexp searching for varchar (in 1.02), and bpchar (in 1.02.1)
create operator ~* (leftarg = bpchar, rightarg = text, procedure =
 texticregexeq);
create operator !~* (leftarg = bpchar, rightarg = text, procedure =
 texticregexne);
create operator ~* (leftarg = varchar, rightarg = text, procedure =
 texticregexeq);
create operator !~* (leftarg = varchar, rightarg = text, procedure
 = texticregexne);
```

36.20.3. Changes

```
Source code maintenance and development
 * worldwide team of volunteers
 * the source tree now in CVS at ftp.ki.net

Enhancements
 * psql (and underlying libpq library) now has many more options
 for
   formatting output, including HTML
 * pg_dump now output the schema and/or the data, with many fixes
 to
   enhance completeness.
 * psql used in place of monitor in administration shell scripts.
   monitor to be depreciated in next release.
 * date/time functions enhanced
 * NULL insert/update/comparison fixed/enhanced
 * TCL/TK lib and shell fixed to work with both tck7.4/tk4.0 and
 tcl7.5/tk4.1

Bug Fixes (almost too numerous to mention)
```

```
 * indexes
 * storage management
 * check for NULL pointer before dereferencing
 * Makefile fixes

New Ports
 * added SolarisX86 port
 * added BSDI 2.1 port
 * added DGUX port
```

36.21. Release v1.01

36.21.1. Migration from v1.0 to v1.01

1996-02-23

The following notes are for the benefit of users who want to migrate databases from postgres95 1.0 to postgres95 1.01.

If you are starting afresh with postgres95 1.01 and do not need to migrate old databases, you do not need to read any further.

In order to postgres95 version 1.01 with databases created with postgres95 version 1.0, the following steps are required:

1. Set the definition of NAMEDATALEN in src/Makefile.global to 16 and OIDNAMELEN to 20.
2. Decide whether you want to use Host based authentication.
   a. If you do, you must create a file name "pg_hba" in your top-level data directory (typically the value of your $PGDATA). src/libpq/pg_hba shows an example syntax.
   b. If you do not want host-based authentication, you can comment out the line

      ```
      HBA = 1
      ```

      in src/Makefile.global

      Note that host-based authentication is turned on by default, and if you do not take steps A or B above, the out-of-the-box 1.01 will not allow you to connect to 1.0 databases.
3. Compile and install 1.01, but DO NOT do the initdb step.
4. Before doing anything else, terminate your 1.0 postmaster, and backup your existing $PGDATA directory.
5. Set your PGDATA environment variable to your 1.0 databases, but set up path up so that 1.01 binaries are being used.
6. Modify the file $PGDATA/PG_VERSION from 5.0 to 5.1
7. Start up a new 1.01 postmaster
8. Add the new built-in functions and operators of 1.01 to 1.0 databases. This is done by running the new 1.01 server against your own 1.0 database and applying the queries attached and saving in the file 1.0_to_1.01.sql. This can be done easily through psql. If your 1.0 database is name "testdb":

   ```
   % psql testdb -f 1.0_to_1.01.sql
   ```

   and then execute the following commands (cut and paste from here):

   ```
   -- add builtin functions that are new to 1.01
   ```

```
create function int4eqoid (int4, oid) returns bool as 'foo'
language 'internal';
create function oideqint4 (oid, int4) returns bool as 'foo'
language 'internal';
create function char2icregexeq (char2, text) returns bool as
 'foo'
language 'internal';
create function char2icregexne (char2, text) returns bool as
 'foo'
language 'internal';
create function char4icregexeq (char4, text) returns bool as
 'foo'
language 'internal';
create function char4icregexne (char4, text) returns bool as
 'foo'
language 'internal';
create function char8icregexeq (char8, text) returns bool as
 'foo'
language 'internal';
create function char8icregexne (char8, text) returns bool as
 'foo'
language 'internal';
create function char16icregexeq (char16, text) returns bool as
 'foo'
language 'internal';
create function char16icregexne (char16, text) returns bool as
 'foo'
language 'internal';
create function texticregexeq (text, text) returns bool as
 'foo'
language 'internal';
create function texticregexne (text, text) returns bool as
 'foo'
language 'internal';

-- add builtin functions that are new to 1.01

create operator = (leftarg = int4, rightarg = oid, procedure =
 int4eqoid);
create operator = (leftarg = oid, rightarg = int4, procedure =
 oideqint4);
create operator ~* (leftarg = char2, rightarg = text, procedure
 = char2icregexeq);
create operator !~* (leftarg = char2, rightarg = text,
 procedure = char2icregexne);
create operator ~* (leftarg = char4, rightarg = text, procedure
 = char4icregexeq);
create operator !~* (leftarg = char4, rightarg = text,
 procedure = char4icregexne);
create operator ~* (leftarg = char8, rightarg = text, procedure
 = char8icregexeq);
create operator !~* (leftarg = char8, rightarg = text,
 procedure = char8icregexne);
create operator ~* (leftarg = char16, rightarg = text,
 procedure = char16icregexeq);
create operator !~* (leftarg = char16, rightarg = text,
 procedure = char16icregexne);
```

```
    create operator ~* (leftarg = text, rightarg = text, procedure
     = texticregexeq);
    create operator !~* (leftarg = text, rightarg = text, procedure
     = texticregexne);
```

36.21.2. Changes

```
Incompatibilities:
 * 1.01 is backwards compatible with 1.0 database provided the user
   follow the steps outlined in the MIGRATION_from_1.0_to_1.01
 file.
   If those steps are not taken, 1.01 is not compatible with 1.0
 database.

Enhancements:
 * added PQdisplayTuples() to libpq and changed monitor and psql to
 use it
 * added NeXT port (requires SysVIPC implementation)
 * added CAST .. AS ... syntax
 * added ASC and DESC keywords
 * added 'internal' as a possible language for CREATE FUNCTION
   internal functions are C functions which have been statically
 linked
   into the postgres backend.
 * a new type "name" has been added for system identifiers (table
 names,
   attribute names, etc.)  This replaces the old char16 type.   The
   of name is set by the NAMEDATALEN #define in src/Makefile.global
 * a readable reference manual that describes the query language.
 * added host-based access control.  A configuration file ($PGDATA/
pg_hba)
   is used to hold the configuration data.  If host-based access
 control
   is not desired, comment out HBA=1 in src/Makefile.global.
 * changed regex handling to be uniform use of Henry Spencer's
 regex code
   regardless of platform.  The regex code is included in the
 distribution
 * added functions and operators for case-insensitive regular
 expressions.
   The operators are ~* and !~*.
 * pg_dump uses COPY instead of SELECT loop for better performance

Bug fixes:
 * fixed an optimizer bug that was causing core dumps when
   functions calls were used in comparisons in the WHERE clause
 * changed all uses of getuid to geteuid so that effective uids are
 used
 * psql now returns non-zero status on errors when using -c
 * applied public patches 1-14
```

36.22. Release v1.0

36.22.1. Changes

1995-09-05

```
Copyright change:
 * The copyright of Postgres 1.0 has been loosened to be freely
 modifiable
   and modifiable for any purpose.  Please read the COPYRIGHT file.
   Thanks to Professor Michael Stonebraker for making this
 possible.

Incompatibilities:
 *  date formats have to be MM-DD-YYYY (or DD-MM-YYYY if you're
 using
   EUROPEAN STYLE).  This follows SQL-92 specs.
 *  "delimiters" is now a keyword

Enhancements:
 *  sql LIKE syntax has been added
 *  copy command now takes an optional USING DELIMITER
 specification.
   delimiters can be any single-character string.
 *  IRIX 5.3 port has been added.
   Thanks to Paul Walmsley and others.
 *  updated pg_dump to work with new libpq
 *  \d has been added psql
   Thanks to Keith Parks
 *  regexp performance for architectures that use POSIX regex has
 been
   improved due to caching of precompiled patterns.
   Thanks to Alistair Crooks
 *  a new version of libpq++
   Thanks to William Wanders

Bug fixes:
 *  arbitrary userids can be specified in the createuser script
 *  \c to connect to other databases in psql now works.
 *  bad pg_proc entry for float4inc() is fixed
 *  users with usecreatedb field set can now create databases
 without
   having to be usesuper
 *  remove access control entries when the entry no longer has any
   permissions
 *  fixed non-portable datetimes implementation
 *  added kerberos flags to the src/backend/Makefile
 *  libpq now works with kerberos
 *  typographic errors in the user manual have been corrected.
 *  btrees with multiple index never worked, now we tell you they
 don't
   work when you try to use them
```

36.23. Postgres95 Beta 0.03

36.23.1. Changes

1995-07-21

```
Incompatible changes:
 * BETA-0.3 IS INCOMPATIBLE WITH DATABASES CREATED WITH PREVIOUS
 VERSIONS
   (due to system catalog changes and indexing structure changes).
```

```
 * double-quote (") is deprecated as a quoting character for string
 literals;
   you need to convert them to single quotes (').
 * name of aggregates (eg. int4sum) are renamed in accordance with
 the
   SQL standard (eg. sum).
 * CHANGE ACL syntax is replaced by GRANT/REVOKE syntax.
 * float literals (eg. 3.14) are now of type float4 (instead of
 float8 in
   previous releases); you might have to do typecasting if you
 depend on it
   being of type float8.  If you neglect to do the typecasting and
 you assign
   a float literal to a field of type float8, you may get incorrect
 values
   stored!
 * LIBPQ has been totally revamped so that frontend applications
   can connect to multiple backends
 * the usesysid field in pg_user has been changed from int2 to int4
 to
   allow wider range of Unix user ids.
 * the netbsd/freebsd/bsd o/s ports have been consolidated into a
   single BSD44_derived port.  (thanks to Alistair Crooks)

SQL standard-compliance (the following details changes that makes
 postgres95
more compliant to the SQL-92 standard):
 * the following SQL types are now built-in: smallint, int(eger),
 float, real,
   char(N), varchar(N), date and time.

   The following are aliases to existing postgres types:
  smallint -> int2
  integer, int -> int4
  float, real  -> float4
   char(N) and varchar(N) are implemented as truncated text types.
 In
   addition, char(N) does blank-padding.
 * single-quote (') is used for quoting string literals; '' (in
 addition to
   \') is supported as means of inserting a single quote in a
 string
 * SQL standard aggregate names (MAX, MIN, AVG, SUM, COUNT) are
 used
   (Also, aggregates can now be overloaded, i.e. you can define
 your
   own MAX aggregate to take in a user-defined type.)
 * CHANGE ACL removed. GRANT/REVOKE syntax added.
   - Privileges can be given to a group using the "GROUP" keyword.
 For example:
  GRANT SELECT ON foobar TO GROUP my_group;
 The keyword 'PUBLIC' is also supported to mean all users.

 Privileges can only be granted or revoked to one user or group
 at a time.

 "WITH GRANT OPTION" is not supported.  Only class owners can
 change
```

```
 access control
   - The default access control is to to grant users readonly
 access.
     You must explicitly grant insert/update access to users.  To
 change
     this, modify the line in
  src/backend/utils/acl.h
     that defines ACL_WORLD_DEFAULT

Bug fixes:
 * the bug where aggregates of empty tables were not run has been
 fixed. Now,
   aggregates run on empty tables will return the initial
 conditions of the
   aggregates. Thus, COUNT of an empty table will now properly
 return 0.
   MAX/MIN of an empty table will return a tuple of value NULL.
 * allow the use of \; inside the monitor
 * the LISTEN/NOTIFY asynchronous notification mechanism now work
 * NOTIFY in rule action bodies now work
 * hash indices work, and access methods in general should perform
 better.
   creation of large btree indices should be much faster.  (thanks
 to Paul
   Aoki)

Other changes and enhancements:
 * addition of an EXPLAIN statement used for explaining the query
 execution
   plan (eg. "EXPLAIN SELECT * FROM EMP" prints out the execution
 plan for
   the query).
 * WARN and NOTICE messages no longer have timestamps on them. To
 turn on
   timestamps of error messages, uncomment the line in
   src/backend/utils/elog.h:
 /* define ELOG_TIMESTAMPS */
 * On an access control violation, the message
 "Either no such class or insufficient privilege"
   will be given.  This is the same message that is returned when
   a class is not found.  This dissuades non-privileged users from
   guessing the existence of privileged classes.
 * some additional system catalog changes have been made that are
 not
   visible to the user.

libpgtcl changes:
 * The -oid option has been added to the "pg_result" tcl command.
   pg_result -oid returns oid of the last tuple inserted.   If the
   last command was not an INSERT, then pg_result -oid returns "".
 * the large object interface is available as pg_lo* tcl commands:
   pg_lo_open, pg_lo_close, pg_lo_creat, etc.

Portability enhancements and New Ports:
 * flex/lex problems have been cleared up.  Now, you should be able
 to use
   flex instead of lex on any platforms.  We no longer make
 assumptions of
```

```
    what lexer you use based on the platform you use.
  * The Linux-ELF port is now supported.  Various configuration have
  been
    tested:  The following configuration is known to work:
  kernel 1.2.10, gcc 2.6.3, libc 4.7.2, flex 2.5.2, bison 1.24
    with everything in ELF format,

New utilities:
  * ipcclean added to the distribution
    ipcclean usually does not need to be run, but if your backend
  crashes
    and leaves shared memory segments hanging around, ipcclean will
    clean them up for you.

New documentation:
  * the user manual has been revised and libpq documentation added.
```

36.24. Postgres95 Beta 0.02

36.24.1. Changes

1995-03-25

```
Incompatible changes:
  * The SQL statement for creating a database is 'CREATE DATABASE'
  instead
    of 'CREATEDB'. Similarly, dropping a database is 'DROP DATABASE'
  instead
    of 'DESTROYDB'. However, the names of the executables 'createdb'
  and
    'destroydb' remain the same.

New tools:
  * pgperl - a Perl (4.036) interface to Postgres95
  * pg_dump - a utility for dumping out a postgres database into a
  script file containing query commands. The script files are in a
  ASCII
  format and can be used to reconstruct the database, even on other
  machines and other architectures. (Also good for converting
  a Postgres 4.2 database to Postgres95 database.)

The following ports have been incorporated into postgres95-
beta-0.02:
  * the NetBSD port by Alistair Crooks
  * the AIX port by Mike Tung
  * the Windows NT port by Jon Forrest (more stuff but not done yet)
  * the Linux ELF port by Brian Gallew

The following bugs have been fixed in postgres95-beta-0.02:
  * new lines not escaped in COPY OUT and problem with COPY OUT when
  first
    attribute is a '.'
  * cannot type return to use the default user id in createuser
  * SELECT DISTINCT on big tables crashes
  * Linux installation problems
  * monitor doesn't allow use of 'localhost' as PGHOST
  * psql core dumps when doing \c or \l
```

```
 * the "pgtclsh" target missing from src/bin/pgtclsh/Makefile
 * libpgtcl has a hard-wired default port number
 * SELECT DISTINCT INTO TABLE hangs
 * CREATE TYPE doesn't accept 'variable' as the internallength
 * wrong result using more than 1 aggregate in a SELECT
```

36.25. Postgres95 Beta 0.01

1995-05-01

Initial release.

36.26. Timing Results

These timing results are from running the regression test with the commands

```
% cd src/test/regress
% make all
% time make runtest
```

Timing under Linux 2.0.27 seems to have a roughly 5% variation from run to run, presumably due to the scheduling vagaries of multitasking systems.

36.26.1. v6.5

As has been the case for previous releases, timing between releases is not directly comparable since new regression tests have been added. In general, v6.5 is faster than previous releases.

Timing with `fsync()` disabled:

```
  Time   System
  02:00  Dual Pentium Pro 180, 224MB, UW-SCSI, Linux 2.0.36, gcc
2.7.2.3 -O2 -m486
  04:38  Sparc Ultra 1 143MHz, 64MB, Solaris 2.6
```

Timing with `fsync()` enabled:

```
  Time   System
  04:21  Dual Pentium Pro 180, 224MB, UW-SCSI, Linux 2.0.36, gcc
2.7.2.3 -O2 -m486
```

For the linux system above, using UW-SCSI disks rather than (older) IDE disks leads to a 50% improvement in speed on the regression test.

36.26.2. v6.4beta

The times for this release are not directly comparable to those for previous releases since some additional regression tests have been included. In general, however, v6.4 should be slightly faster than the previous release (thanks, Bruce!).

```
  Time   System
  02:26  Dual Pentium Pro 180, 96MB, UW-SCSI, Linux 2.0.30, gcc
2.7.2.1 -O2 -m486
```

36.26.3. v6.3

The times for this release are not directly comparable to those for previous releases since some additional regression tests have been included and some obsolete tests involving time travel have been removed. In general, however, v6.3 is substantially faster than previous releases (thanks, Bruce!).

```
  Time   System
  02:30  Dual Pentium Pro 180, 96MB, UW-SCSI, Linux 2.0.30, gcc
 2.7.2.1 -O2 -m486
  04:12  Dual Pentium Pro 180, 96MB, EIDE, Linux 2.0.30, gcc
 2.7.2.1 -O2 -m486
```

36.26.4. v6.1

```
  Time   System
  06:12  Pentium Pro 180, 32MB, EIDE, Linux 2.0.30, gcc 2.7.2 -O2 -
m486
  12:06  P-100, 48MB, Linux 2.0.29, gcc
  39:58  Sparc IPC 32MB, Solaris 2.5, gcc 2.7.2.1 -O -g
```

Part III. Programmer's Guide

Information for extending Postgres.

Table of Contents

37. Architecture 409
    37.1. Postgres Architectural Concepts 409
38. Extending SQL: An Overview 415
    38.1. How Extensibility Works 415
    38.2. The Postgres Type System 415
    38.3. About the Postgres System Catalogs 415
39. Extending SQL: Functions 418
    39.1. Query Language (SQL) Functions 418
        39.1.1. Examples 418
        39.1.2. SQL Functions on Base Types 419
        39.1.3. SQL Functions on Composite Types 419
    39.2. Procedural Language Functions 421
    39.3. Internal Functions 421
    39.4. Compiled (C) Language Functions 421
        39.4.1. C Language Functions on Base Types 422
        39.4.2. C Language Functions on Composite Types 425
        39.4.3. Writing Code 426
    39.5. Function Overloading 427
        39.5.1. Name Space Conflicts 427
40. Extending SQL: Types 429
    40.1. User-Defined Types 429
        40.1.1. Functions Needed for a User-Defined Type 429
        40.1.2. Large Objects 430
41. Extending SQL: Operators 431
    41.1. Operator Optimization Information 431
        41.1.1. COMMUTATOR 432
        41.1.2. NEGATOR 432
        41.1.3. RESTRICT 433
        41.1.4. JOIN 433
        41.1.5. HASHES 434
        41.1.6. SORT1 and SORT2 434
42. Extending SQL: Aggregates 436
43. The Postgres Rule System 438
    43.1. What is a Querytree? 438
        43.1.1. The Parts of a Querytree 438
    43.2. Views and the Rule System 439
        43.2.1. Implementation of Views in Postgres 439
        43.2.2. How SELECT Rules Work 440
        43.2.3. View Rules in Non-SELECT Statements 445
        43.2.4. The Power of Views in Postgres 446
        43.2.5. Implementation Side Effects 447
    43.3. Rules on INSERT, UPDATE and DELETE 448
        43.3.1. Differences to View Rules 448
        43.3.2. How These Rules Work 448
        43.3.3. Cooperation with Views 452
    43.4. Rules and Permissions 458
    43.5. Rules versus Triggers 458
44. Interfacing Extensions To Indices 462
45. Index Cost Estimation Functions 468
46. GiST Indices 471
47. Linking Dynamically-Loaded Functions 473
    47.1. Linux 473
    47.2. DEC OSF/1 474
    47.3. SunOS 4.x, Solaris 2.x and HP-UX 474
48. Triggers 476

48.1. Trigger Creation 476
48.2. Interaction with the Trigger Manager 477
48.3. Visibility of Data Changes 479
48.4. Examples 479
49. Server Programming Interface 483
49.1. Interface Functions 483
49.2. Interface Support Functions 491
49.3. Memory Management 503
49.4. Visibility of Data Changes 504
49.5. Examples 504
50. Procedural Languages 507
50.1. Installing Procedural Languages 507

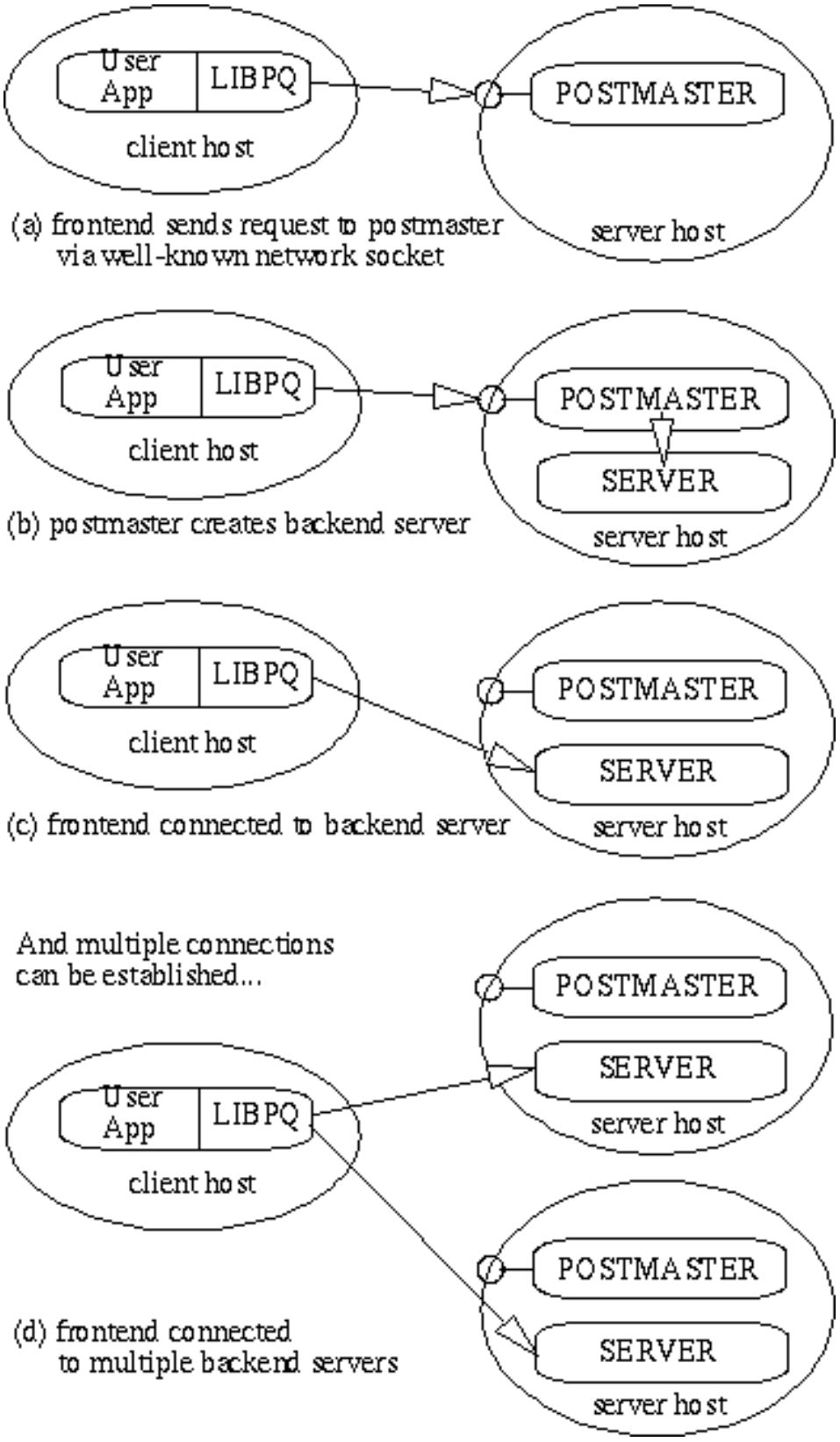
Chapter 37. Architecture

37.1. Postgres Architectural Concepts

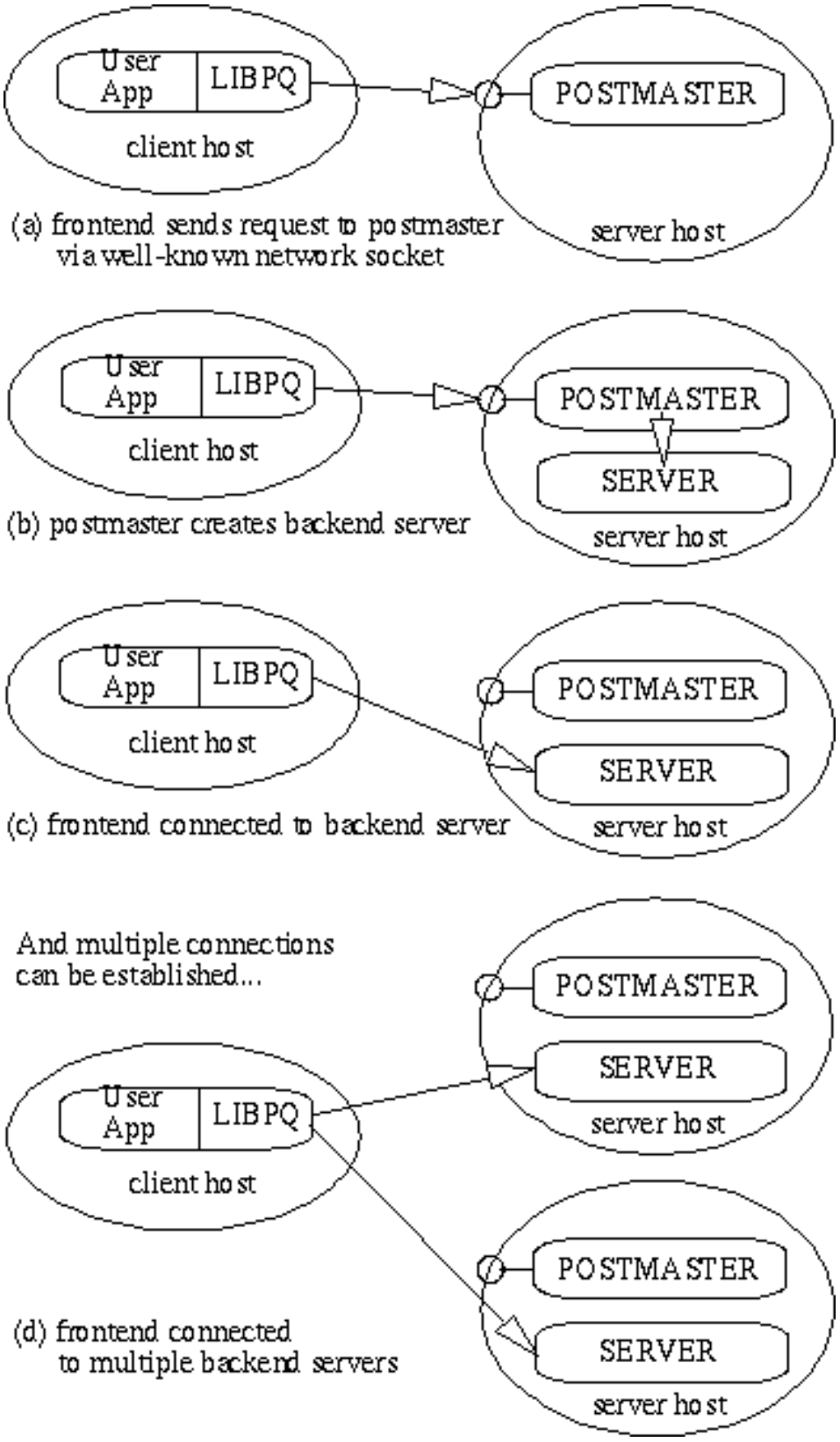
Before we continue, you should understand the basic Postgres system architecture. Understanding how the parts of Postgres interact will make the next chapter somewhat clearer. In database jargon, Postgres uses a simple "process per-user" client/server model. A Postgres session consists of the following cooperating Unix processes (programs):

- A supervisory daemon process (postmaster),
- the user's frontend application (e.g., the psql program), and
- the one or more backend database servers (the postgres process itself).

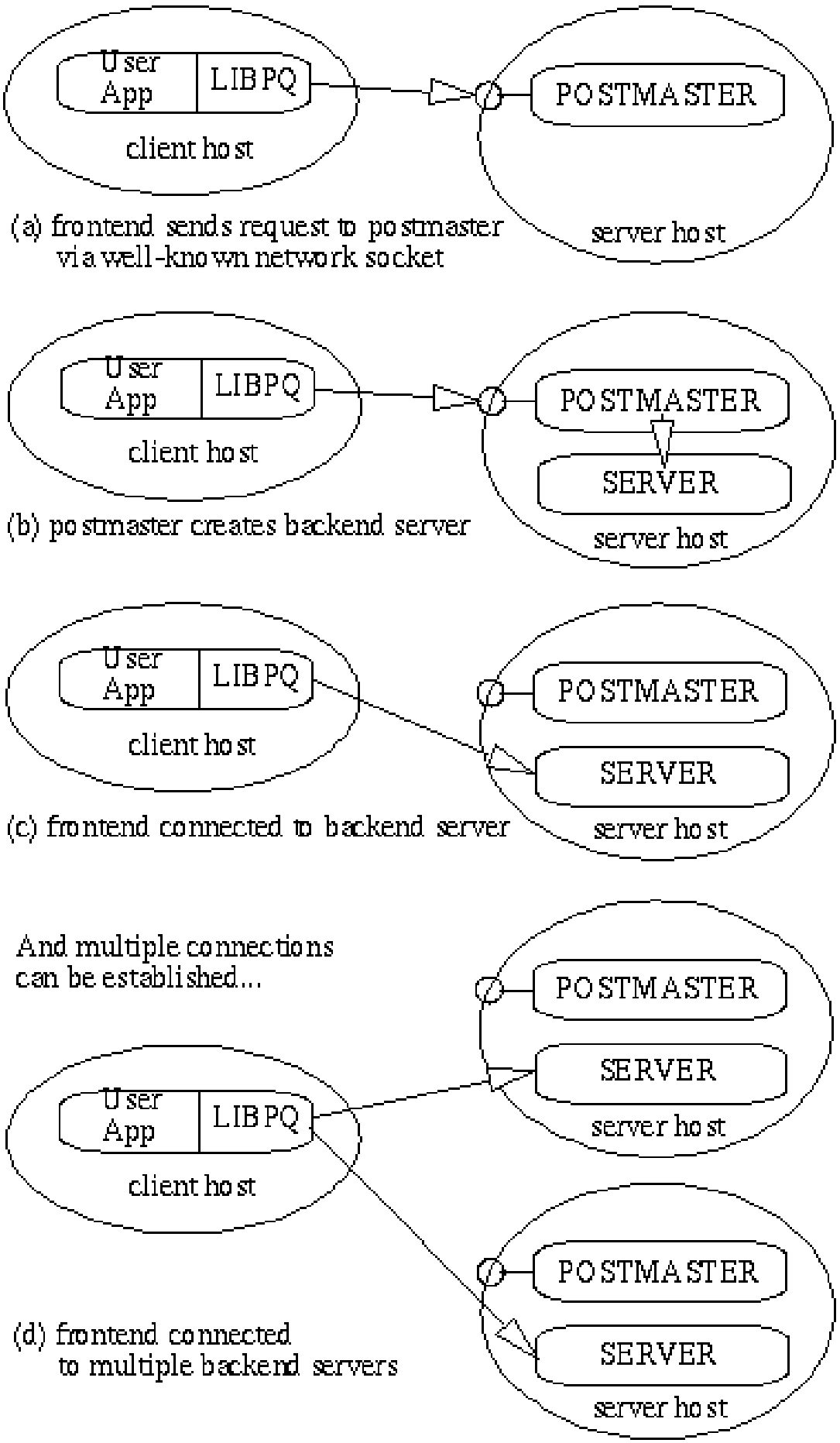
A single postmaster manages a given collection of databases on a single host. Such a collection of databases is called an installation or site. Frontend applications that wish to access a given database within an installation make calls to the library. The library sends user requests over the network to the postmaster (



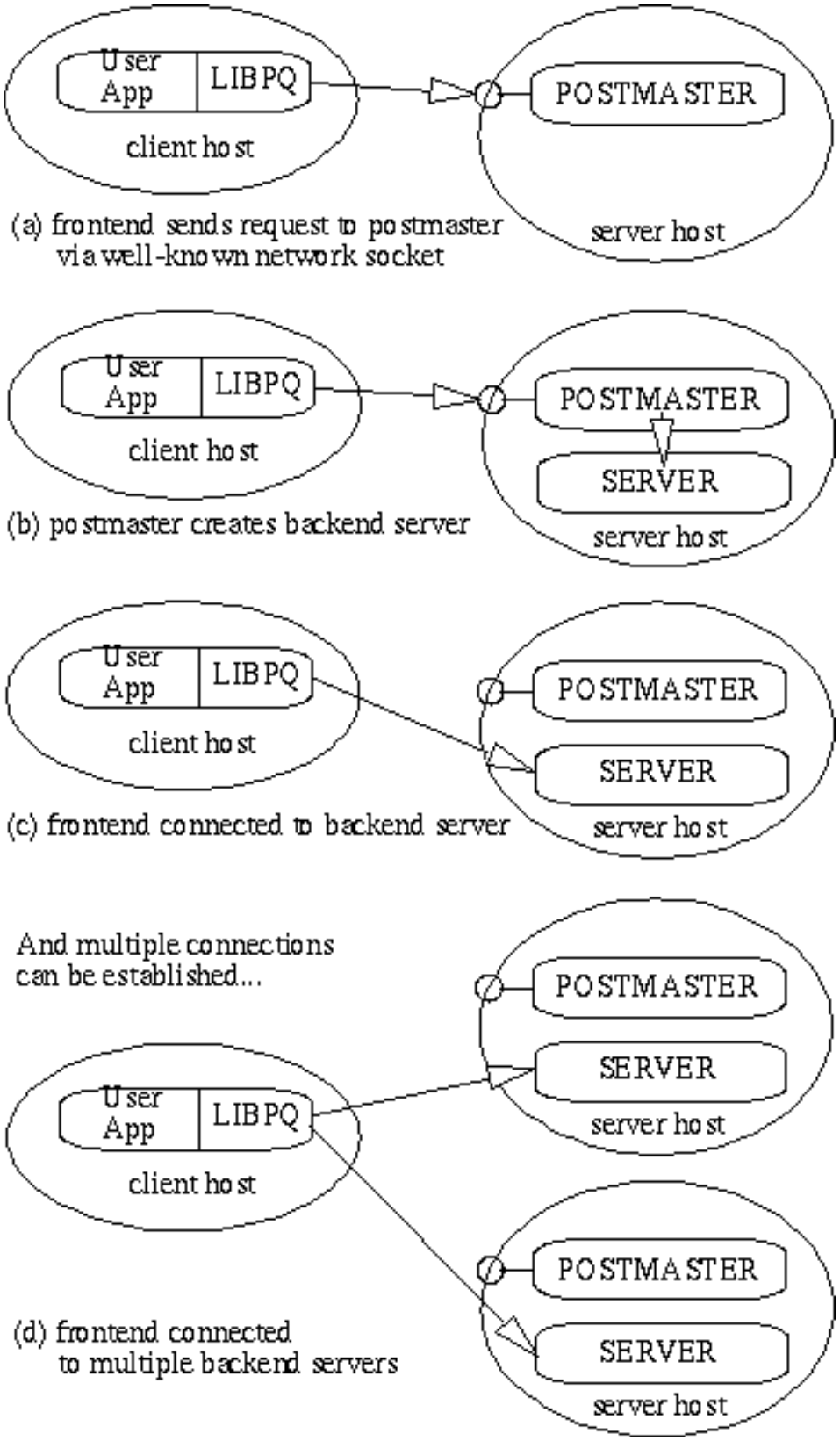
(a)), which in turn starts a new backend server process (


User App
LIBPQ
client host
POSTMASTER
server host
(a) frontend sends request to postmaster via well-known network socket
SERVER
(b) postmaster creates backend server
(c) frontend connected to backend server
And multiple connections can be established...
(d) frontend connected to multiple backend servers

(b))


User App
LIBPQ
client host
POSTMASTER
server host
(a) frontend sends request to postmaster via well-known network socket
User App
LIBPQ
client host
POSTMASTER
SERVER
server host
(b) postmaster creates backend server
User App
LIBPQ
client host
POSTMASTER
SERVER
server host
(c) frontend connected to backend server
And multiple connections can be established...
POSTMASTER
SERVER
server host
User App
LIBPQ
client host
POSTMASTER
SERVER
server host
(d) frontend connected to multiple backend servers

and connects the frontend process to the new server (



(c)). From that point on, the frontend process and the backend server communicate without intervention by the postmaster. Hence, the postmaster is always running, waiting for requests, whereas frontend and backend processes come and go. The `libpq` library allows a single frontend to make multiple connections to backend processes. However, the frontend application is still a single-threaded process. Multithreaded frontend/backend connections are not currently supported in `libpq`. One implication

of this architecture is that the postmaster and the backend always run on the same machine (the database server), while the frontend application may run anywhere. You should keep this in mind, because the files that can be accessed on a client machine may not be accessible (or may only be accessed using a different filename) on the database server machine. You should also be aware that the postmaster and postgres servers run with the user-id of the Postgres "superuser." Note that the Postgres superuser does not have to be a special user (e.g., a user named "postgres"), although many systems are installed that way. Furthermore, the Postgres superuser should definitely not be the Unix superuser, "root"! In any case, all files relating to a database should belong to this Postgres superuser.

Chapter 38. Extending SQL: An Overview

In the sections that follow, we will discuss how you can extend the Postgres SQL query language by adding:

- functions
- types
- operators
- aggregates

38.1. How Extensibility Works

Postgres is extensible because its operation is catalog-driven. If you are familiar with standard relational systems, you know that they store information about databases, tables, columns, etc., in what are commonly known as system catalogs. (Some systems call this the data dictionary). The catalogs appear to the user as classes, like any other, but the DBMS stores its internal bookkeeping in them. One key difference between Postgres and standard relational systems is that Postgres stores much more information in its catalogs -- not only information about tables and columns, but also information about its types, functions, access methods, and so on. These classes can be modified by the user, and since Postgres bases its internal operation on these classes, this means that Postgres can be extended by users. By comparison, conventional database systems can only be extended by changing hardcoded procedures within the DBMS or by loading modules specially-written by the DBMS vendor.

Postgres is also unlike most other data managers in that the server can incorporate user-written code into itself through dynamic loading. That is, the user can specify an object code file (e.g., a compiled .o file or shared library) that implements a new type or function and Postgres will load it as required. Code written in SQL are even more trivial to add to the server. This ability to modify its operation "on the fly" makes Postgres uniquely suited for rapid prototyping of new applications and storage structures.

38.2. The Postgres Type System

The Postgres type system can be broken down in several ways. Types are divided into base types and composite types. Base types are those, like *int4*, that are implemented in a language such as C. They generally correspond to what are often known as "abstract data types"; Postgres can only operate on such types through methods provided by the user and only understands the behavior of such types to the extent that the user describes them. Composite types are created whenever the user creates a class. EMP is an example of a composite type.

Postgres stores these types in only one way (within the file that stores all instances of the class) but the user can "look inside" at the attributes of these types from the query language and optimize their retrieval by (for example) defining indices on the attributes. Postgres base types are further divided into built-in types and user-defined types. Built-in types (like *int4*) are those that are compiled into the system. User-defined types are those created by the user in the manner to be described below.

38.3. About the Postgres System Catalogs

Having introduced the basic extensibility concepts, we can now take a look at how the catalogs are actually laid out. You can skip this section for now, but some later sections will be incomprehensible without the information given here, so mark this page for later reference. All system catalogs have names that begin with *pg_*. The following classes contain information that may be useful to the end user. (There are many other system catalogs, but there should rarely be a reason to query them directly.)

Table 38.1. Postgres System Catalogs

Catalog Name	Description
pg_database	databases
pg_class	classes
pg_attribute	class attributes
pg_index	secondary indices
pg_proc	procedures (both C and SQL)
pg_type	types (both base and complex)
pg_operator	operators
pg_aggregate	aggregates and aggregate functions
pg_am	access methods
pg_amop	access method operators
pg_amproc	access method support functions
pg_opclass	access method operator classes

Figure 38.1. The major Postgres system catalogs

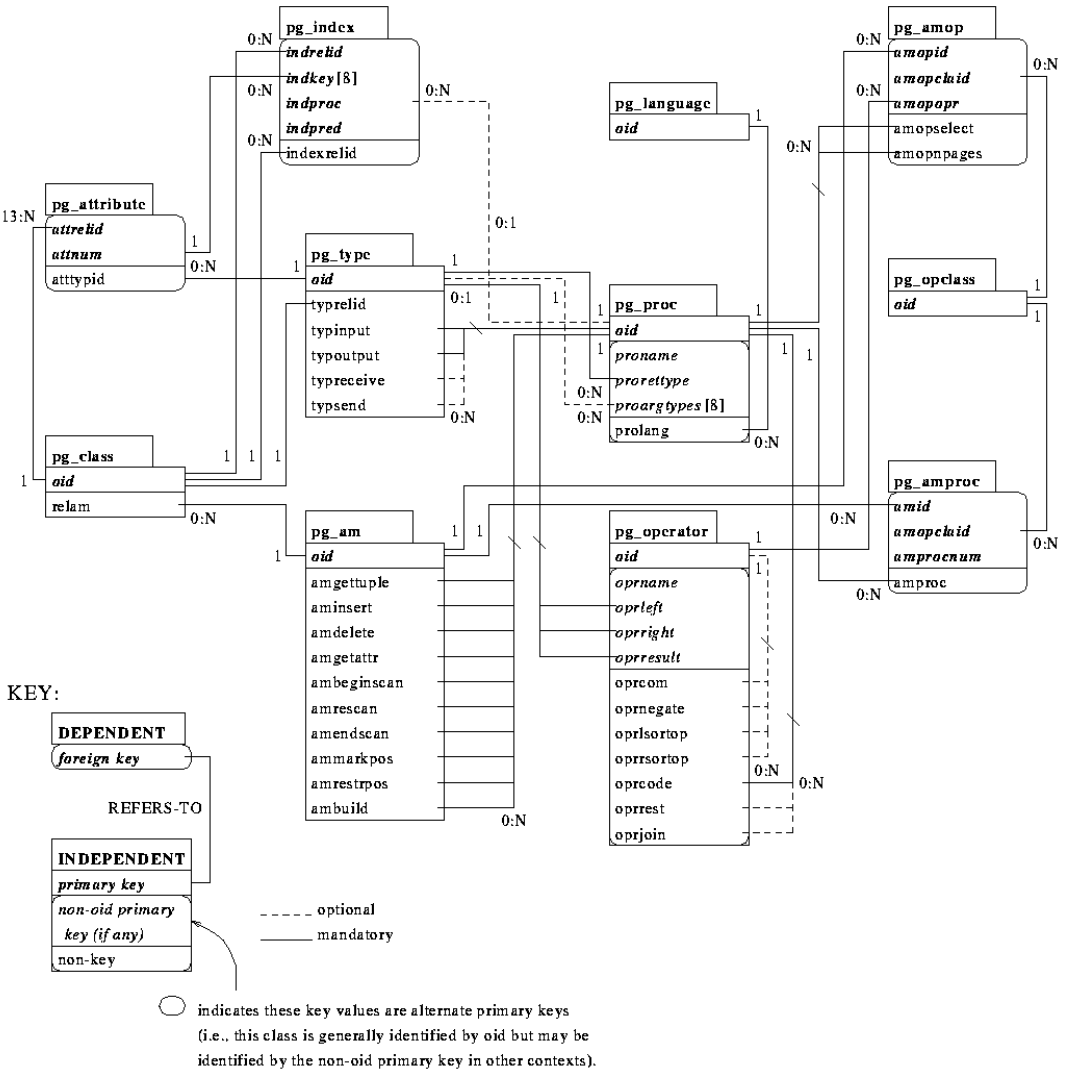


Figure 3. The major POSTGRES system catalogs.

The Reference Manual gives a more detailed explanation of these catalogs and their attributes. However,

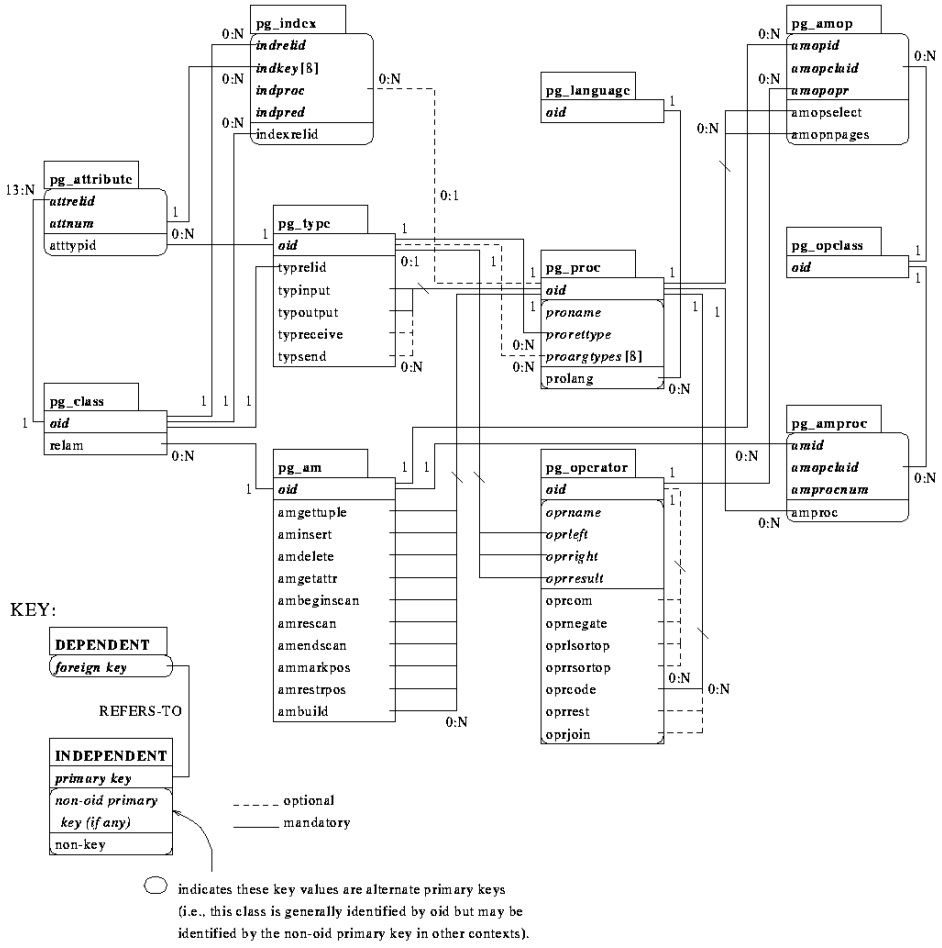


Figure 3. The major POSTGRES system catalogs.

shows the major entities and their relationships in the system catalogs. (Attributes that do not refer to other entities are not shown unless they are part of a primary key.) This diagram is more or less incomprehensible until you actually start looking at the contents of the catalogs and see how they relate to each other. For now, the main things to take away from this diagram are as follows:

- In several of the sections that follow, we will present various join queries on the system catalogs that display information we need to extend the system. Looking at this diagram should make some of these join queries (which are often three- or four-way joins) more understandable, because you will be able to see that the attributes used in the queries form foreign keys in other classes.
- Many different features (classes, attributes, functions, types, access methods, etc.) are tightly integrated in this schema. A simple create command may modify many of these catalogs.
- Types and procedures are central to the schema.

Note

We use the words *procedure* and *function* more or less interchangably.

Nearly every catalog contains some reference to instances in one or both of these classes. For example, Postgres frequently uses type signatures (e.g., of functions and operators) to identify unique instances of other catalogs.

- There are many attributes and relationships that have obvious meanings, but there are many (particularly those that have to do with access methods) that do not. The relationships between pg_am, pg_amop, pg_amproc, pg_operator and pg_opclass are particularly hard to understand and will be described in depth (in the section on interfacing types and operators to indices) after we have discussed basic extensions.

Chapter 39. Extending SQL: Functions

As it turns out, part of defining a new type is the definition of functions that describe its behavior. Consequently, while it is possible to define a new function without defining a new type, the reverse is not true. We therefore describe how to add new functions to Postgres before describing how to add new types.

Postgres SQL provides three types of functions:

- query language functions (functions written in SQL)
- procedural language functions (functions written in, for example, PLTCL or PLSQL)
- programming language functions (functions written in a compiled programming language such as C)

Every kind of function can take a base type, a composite type or some combination as arguments (parameters). In addition, every kind of function can return a base type or a composite type. It's easiest to define SQL functions, so we'll start with those. Examples in this section can also be found in `funcs.sql` and `funcs.c`.

39.1. Query Language (SQL) Functions

SQL functions execute an arbitrary list of SQL queries, returning the results of the last query in the list. SQL functions in general return sets. If their returntype is not specified as a `setof`, then an arbitrary element of the last query's result will be returned.

The body of a SQL function following AS should be a list of queries separated by whitespace characters and bracketed within quotation marks. Note that quotation marks used in the queries must be escaped, by preceding them with two backslashes.

Arguments to the SQL function may be referenced in the queries using a $n syntax: $1 refers to the first argument, $2 to the second, and so on. If an argument is complex, then a *dot* notation (e.g. "$1.emp") may be used to access attributes of the argument or to invoke functions.

39.1.1. Examples

To illustrate a simple SQL function, consider the following, which might be used to debit a bank account:

```
CREATE FUNCTION tp1 (int4, float8)
RETURNS int4
AS ' UPDATE bank
 SET balance = bank.balance - $2
        WHERE bank.acctountno = $1;
        SELECT 1;'
LANGUAGE 'sql';
```

A user could execute this function to debit account 17 by $100.00 as follows:

```
select (x = TP1( 17,100.0));
```

The following more interesting example takes a single argument of type EMP, and retrieves multiple results:

```
select function hobbies (EMP) returns set of HOBBIES
```

```
    as 'select HOBBIES.* from HOBBIES
        where $1.name = HOBBIES.person'
    language 'sql';
```

39.1.2. SQL Functions on Base Types

The simplest possible SQL function has no arguments and simply returns a base type, such as `int4`:

```
CREATE FUNCTION one()
RETURNS int4
AS 'SELECT 1 as RESULT;'
LANGUAGE 'sql';

SELECT one() AS answer;

     +-------+
     |answer |
     +-------+
     |1      |
     +-------+
```

Notice that we defined a target list for the function (with the name RESULT), but the target list of the query that invoked the function overrode the function's target list. Hence, the result is labelled answer instead of one.

It's almost as easy to define SQL functions that take base types as arguments. In the example below, notice how we refer to the arguments within the function as $1 and $2:

```
CREATE FUNCTION add_em(int4, int4)
RETURNS int4
AS 'SELECT $1 + $2;'
LANGUAGE 'sql';

SELECT add_em(1, 2) AS answer;

     +-------+
     |answer |
     +-------+
     |3      |
     +-------+
```

39.1.3. SQL Functions on Composite Types

When specifying functions with arguments of composite types (such as EMP), we must not only specify which argument we want (as we did above with $1 and $2) but also the attributes of that argument. For example, take the function double_salary that computes what your salary would be if it were doubled:

```
CREATE FUNCTION double_salary(EMP)
RETURNS int4
AS 'SELECT $1.salary * 2 AS salary;'
LANGUAGE 'sql';

SELECT name, double_salary(EMP) AS dream
FROM EMP
WHERE EMP.cubicle ~= '(2,1)'::point;
```

```
+-----+-------+
|name | dream |
+-----+-------+
|Sam  | 2400  |
+-----+-------+
```

Notice the use of the syntax $1.salary. Before launching into the subject of functions that return composite types, we must first introduce the function notation for projecting attributes. The simple way to explain this is that we can usually use the notation attribute(class) and class.attribute interchangably:

```
--
-- this is the same as:
--  SELECT EMP.name AS youngster FROM EMP WHERE EMP.age < 30
--
SELECT name(EMP) AS youngster
    FROM EMP
    WHERE age(EMP) < 30;

     +----------+
     |youngster |
     +----------+
     |Sam       |
     +----------+
```

As we shall see, however, this is not always the case. This function notation is important when we want to use a function that returns a single instance. We do this by assembling the entire instance within the function, attribute by attribute. This is an example of a function that returns a single EMP instance:

```
CREATE FUNCTION new_emp()
RETURNS EMP
AS ' SELECT \'None\'::text AS name,
         1000 AS salary,
         25 AS age,
         \'(2,2)\'::point AS cubicle'
LANGUAGE 'sql';
```

In this case we have specified each of the attributes with a constant value, but any computation or expression could have been substituted for these constants. Defining a function like this can be tricky. Some of the more important caveats are as follows:

- The target list order must be exactly the same as that in which the attributes appear in the CREATE TABLE statement (or when you execute a .* query).
- You must typecast the expressions (using ::) very carefully or you will see the following error:

  ```
   NOTICE::function declared to return type EMP does not retrieve
  (EMP.*)
  ```

- When calling a function that returns an instance, we cannot retrieve the entire instance. We must either project an attribute out of the instance or pass the entire instance into another function.

  ```
  SELECT name(new_emp()) AS nobody;
  ```

```
+-------+
|nobody |
+-------+
|None   |
+-------+
```

- The reason why, in general, we must use the function syntax for projecting attributes of function return values is that the parser just doesn't understand the other (dot) syntax for projection when combined with function calls.

  ```
  SELECT new_emp().name AS nobody;
  NOTICE:parser: syntax error at or near "."
  ```

Any collection of commands in the SQL query language can be packaged together and defined as a function. The commands can include updates (i.e., `INSERT`, `UPDATE`, and `DELETE`) as well as `SELECT` queries. However, the final command must be a `SELECT` that returns whatever is specified as the function's returntype.

```
CREATE FUNCTION clean_EMP ()
RETURNS int4
AS ' DELETE FROM EMP
 WHERE EMP.salary <= 0;
 SELECT 1 AS ignore_this;'
LANGUAGE 'sql';

SELECT clean_EMP();

     +--+
     |x |
     +--+
     |1 |
     +--+
```

39.2. Procedural Language Functions

Procedural languages aren't built into Postgres. They are offered by loadable modules. Please refer to the documentation for the PL in question for details about the syntax and how the AS clause is interpreted by the PL handler.

There are two procedural languages available with the standard Postgres distribution (PLTCL and PLSQL), and other languages can be defined. Refer to Procedural Languages for more information.

39.3. Internal Functions

Internal functions are functions written in C which have been statically linked into the Postgres backend process. The AS clause gives the C-language name of the function, which need not be the same as the name being declared for SQL use. (For reasons of backwards compatibility, an empty AS string is accepted as meaning that the C-language function name is the same as the SQL name.) Normally, all internal functions present in the backend are declared as SQL functions during database initialization, but a user could use `CREATE FUNCTION` to create additional alias names for an internal function.

39.4. Compiled (C) Language Functions

Functions written in C can be compiled into dynamically loadable objects, and used to implement user-defined SQL functions. The first time the user defined function is called inside the backend, the

dynamic loader loads the function's object code into memory, and links the function with the running Postgres executable. The SQL syntax for `CREATE FUNCTION` links the SQL function to the C source function in one of two ways. If the SQL function has the same name as the C source function the first form of the statement is used. The string argument in the AS clause is the full pathname of the file that contains the dynamically loadable compiled object. If the name of the C function is different from the desired name of the SQL function, then the second form is used. In this form the AS clause takes two string arguments, the first is the full pathname of the dynamically loadable object file, and the second is the link symbol that the dynamic loader should search for. This link symbol is just the function name in the C source code.

Note

After it is used for the first time, a dynamically loaded, user function is retained in memory, and future calls to the function only incur the small overhead of a symbol table lookup.

The string which specifies the object file (the string in the AS clause) should be the *full path* of the object code file for the function, bracketed by quotation marks. If a link symbol is used in the AS clause, the link symbol should also be bracketed by single quotation marks, and should be exactly the same as the name of the function in the C source code. On Unix systems the command `nm` will print all of the link symbols in a dynamically loadable object. (Postgres will not compile a function automatically; it must be compiled before it is used in a CREATE FUNCTION command. See below for additional information.)

39.4.1. C Language Functions on Base Types

The following table gives the C type required for parameters in the C functions that will be loaded into Postgres. The "Defined In" column gives the actual header file (in the `.../src/backend/` directory) that the equivalent C type is defined. However, if you include `utils/builtins.h`, these files will automatically be included.

Table 39.1. Equivalent C Types for Built-In Postgres Types

Built-In Type	C Type	Defined In
abstime	AbsoluteTime	utils/nabstime.h
bool	bool	include/c.h
box	(BOX *)	utils/geo-decls.h
bytea	(bytea *)	include/postgres.h
char	char	N/A
cid	CID	include/postgres.h
datetime	(DateTime *)	include/c.h or include/postgres.h
int2	int2	include/postgres.h
int2vector	(int2vector *)	include/postgres.h
int4	int4	include/postgres.h
float4	float32 or (float4 *)	include/c.h or include/postgres.h
float8	float64 or (float8 *)	include/c.h or include/postgres.h
lseg	(LSEG *)	include/geo-decls.h
name	(Name)	include/postgres.h
oid	oid	include/postgres.h
oidvector	(oidvector *)	include/postgres.h
path	(PATH *)	utils/geo-decls.h
point	(POINT *)	utils/geo-decls.h

Built-In Type	C Type	Defined In
regproc	regproc or REGPROC	include/postgres.h
reltime	RelativeTime	utils/nabstime.h
text	(text *)	include/postgres.h
tid	ItemPointer	storage/itemptr.h
timespan	(TimeSpan *)	include/c.h or include/postgres.h
tinterval	TimeInterval	utils/nabstime.h
uint2	uint16	include/c.h
uint4	uint32	include/c.h
xid	(XID *)	include/postgres.h

Internally, Postgres regards a base type as a "blob of memory." The user-defined functions that you define over a type in turn define the way that Postgres can operate on it. That is, Postgres will only store and retrieve the data from disk and use your user-defined functions to input, process, and output the data. Base types can have one of three internal formats:

- pass by value, fixed-length
- pass by reference, fixed-length
- pass by reference, variable-length

By-value types can only be 1, 2 or 4 bytes in length (even if your computer supports by-value types of other sizes). Postgres itself only passes integer types by value. You should be careful to define your types such that they will be the same size (in bytes) on all architectures. For example, the `long` type is dangerous because it is 4 bytes on some machines and 8 bytes on others, whereas `int` type is 4 bytes on most Unix machines (though not on most personal computers). A reasonable implementation of the `int4` type on Unix machines might be:

```
/* 4-byte integer, passed by value */
typedef int int4;
```

On the other hand, fixed-length types of any size may be passed by-reference. For example, here is a sample implementation of a Postgres type:

```
/* 16-byte structure, passed by reference */
typedef struct
{
    double  x, y;
} Point;
```

Only pointers to such types can be used when passing them in and out of Postgres functions. Finally, all variable-length types must also be passed by reference. All variable-length types must begin with a length field of exactly 4 bytes, and all data to be stored within that type must be located in the memory immediately following that length field. The length field is the total length of the structure (i.e., it includes the size of the length field itself). We can define the text type as follows:

```
typedef struct {
    int4 length;
    char data[1];
} text;
```

Obviously, the data field is not long enough to hold all possible strings; it's impossible to declare such a structure in C. When manipulating variable-length types, we must be careful to allocate the correct

amount of memory and initialize the length field. For example, if we wanted to store 40 bytes in a text structure, we might use a code fragment like this:

```
#include "postgres.h"
...
char buffer[40]; /* our source data */
...
text *destination = (text *) palloc(VARHDRSZ + 40);
destination->length = VARHDRSZ + 40;
memmove(destination->data, buffer, 40);
...
```

Now that we've gone over all of the possible structures for base types, we can show some examples of real functions. Suppose `funcs.c` look like:

```
#include <string.h>
#include "postgres.h"

/* By Value */

int
add_one(int arg)
{
    return(arg + 1);
}

/* By Reference, Fixed Length */

Point *
makepoint(Point *pointx, Point *pointy )
{
    Point     *new_point = (Point *) palloc(sizeof(Point));

    new_point->x = pointx->x;
    new_point->y = pointy->y;

    return new_point;
}

/* By Reference, Variable Length */

text *
copytext(text *t)
{
    /*
     * VARSIZE is the total size of the struct in bytes.
     */
    text *new_t = (text *) palloc(VARSIZE(t));
    memset(new_t, 0, VARSIZE(t));
    VARSIZE(new_t) = VARSIZE(t);
    /*
     * VARDATA is a pointer to the data region of the struct.
     */
    memcpy((void *) VARDATA(new_t), /* destination */
           (void *) VARDATA(t),     /* source */
           VARSIZE(t)-VARHDRSZ);        /* how many bytes */
    return(new_t);
}
```

```
text *
concat_text(text *arg1, text *arg2)
{
    int32 new_text_size = VARSIZE(arg1) + VARSIZE(arg2) - VARHDRSZ;
    text *new_text = (text *) palloc(new_text_size);

    memset((void *) new_text, 0, new_text_size);
    VARSIZE(new_text) = new_text_size;
    strncpy(VARDATA(new_text), VARDATA(arg1), VARSIZE(arg1)-
VARHDRSZ);
    strncat(VARDATA(new_text), VARDATA(arg2), VARSIZE(arg2)-
VARHDRSZ);
    return (new_text);
}
```

On OSF/1 we would type:

```
CREATE FUNCTION add_one(int4) RETURNS int4
     AS 'PGROOT/tutorial/funcs.so' LANGUAGE 'c';

CREATE FUNCTION makepoint(point, point) RETURNS point
     AS 'PGROOT/tutorial/funcs.so' LANGUAGE 'c';

CREATE FUNCTION concat_text(text, text) RETURNS text
     AS 'PGROOT/tutorial/funcs.so' LANGUAGE 'c';

CREATE FUNCTION copytext(text) RETURNS text
     AS 'PGROOT/tutorial/funcs.so' LANGUAGE 'c';
```

On other systems, we might have to make the filename end in .sl (to indicate that it's a shared library).

39.4.2. C Language Functions on Composite Types

Composite types do not have a fixed layout like C structures. Instances of a composite type may contain null fields. In addition, composite types that are part of an inheritance hierarchy may have different fields than other members of the same inheritance hierarchy. Therefore, Postgres provides a procedural interface for accessing fields of composite types from C. As Postgres processes a set of instances, each instance will be passed into your function as an opaque structure of type `TUPLE`. Suppose we want to write a function to answer the query

```
* SELECT name, c_overpaid(EMP, 1500) AS overpaid
  FROM EMP
  WHERE name = 'Bill' or name = 'Sam';
```

In the query above, we can define c_overpaid as:

```
#include "postgres.h"
#include "executor/executor.h"  /* for GetAttributeByName() */

bool
c_overpaid(TupleTableSlot *t, /* the current instance of EMP */
           int4 limit)
{
    bool isnull = false;
    int4 salary;
```

```
    salary = (int4) GetAttributeByName(t, "salary", &isnull);
    if (isnull)
        return (false);
    return(salary > limit);
}
```

`GetAttributeByName` is the Postgres system function that returns attributes out of the current instance. It has three arguments: the argument of type TUPLE passed into the function, the name of the desired attribute, and a return parameter that describes whether the attribute is null. `GetAttributeByName` will align data properly so you can cast its return value to the desired type. For example, if you have an attribute name which is of the type name, the `GetAttributeByName` call would look like:

```
char *str;
...
str = (char *) GetAttributeByName(t, "name", &isnull)
```

The following query lets Postgres know about the c_overpaid function:

```
CREATE FUNCTION c_overpaid(EMP, int4)
RETURNS bool
AS 'PGROOT/tutorial/obj/funcs.so'
LANGUAGE 'c';
```

While there are ways to construct new instances or modify existing instances from within a C function, these are far too complex to discuss in this manual.

39.4.3. Writing Code

We now turn to the more difficult task of writing programming language functions. Be warned: this section of the manual will not make you a programmer. You must have a good understanding of C (including the use of pointers and the malloc memory manager) before trying to write C functions for use with Postgres. While it may be possible to load functions written in languages other than C into Postgres, this is often difficult (when it is possible at all) because other languages, such as FORTRAN and Pascal often do not follow the same *calling convention* as C. That is, other languages do not pass argument and return values between functions in the same way. For this reason, we will assume that your programming language functions are written in C.

C functions with base type arguments can be written in a straightforward fashion. The C equivalents of built-in Postgres types are accessible in a C file if *PGROOT*`/src/backend/utils/builtins.h` is included as a header file. This can be achieved by having

```
#include <utils/builtins.h>
```

at the top of the C source file.

The basic rules for building C functions are as follows:

- Most of the header (include) files for Postgres should already be installed in *PGROOT*`/include` (see Figure 2). You should always include

  ```
  -I$PGROOT/include
  ```

  on your cc command lines. Sometimes, you may find that you require header files that are in the server source itself (i.e., you need a file we neglected to install in include). In those cases you may need to add one or more of

```
-I$PGROOT/src/backend
-I$PGROOT/src/backend/include
-I$PGROOT/src/backend/port/<PORTNAME>
-I$PGROOT/src/backend/obj
```

(where <PORTNAME> is the name of the port, e.g., alpha or sparc).

- When allocating memory, use the Postgres routines palloc and pfree instead of the corresponding C library routines malloc and free. The memory allocated by palloc will be freed automatically at the end of each transaction, preventing memory leaks.
- Always zero the bytes of your structures using memset or bzero. Several routines (such as the hash access method, hash join and the sort algorithm) compute functions of the raw bits contained in your structure. Even if you initialize all fields of your structure, there may be several bytes of alignment padding (holes in the structure) that may contain garbage values.
- Most of the internal Postgres types are declared in `postgres.h`, so it's a good idea to always include that file as well. Including postgres.h will also include elog.h and palloc.h for you.
- Compiling and loading your object code so that it can be dynamically loaded into Postgres always requires special flags. See Linking Dynamically-Loaded Functions for a detailed explanation of how to do it for your particular operating system.

39.5. Function Overloading

More than one function may be defined with the same name, as long as the arguments they take are different. In other words, function names can be *overloaded*. A function may also have the same name as an attribute. In the case that there is an ambiguity between a function on a complex type and an attribute of the complex type, the attribute will always be used.

39.5.1. Name Space Conflicts

As of Postgres v7.0, the alternative form of the AS clause for the SQL `CREATE FUNCTION` command decouples the SQL function name from the function name in the C source code. This is now the preferred technique to accomplish function overloading.

39.5.1.1. Pre-v7.0

For functions written in C, the SQL name declared in `CREATE FUNCTION` must be exactly the same as the actual name of the function in the C code (hence it must be a legal C function name).

There is a subtle implication of this restriction: while the dynamic loading routines in most operating systems are more than happy to allow you to load any number of shared libraries that contain conflicting (identically-named) function names, they may in fact botch the load in interesting ways. For example, if you define a dynamically-loaded function that happens to have the same name as a function built into Postgres, the DEC OSF/1 dynamic loader causes Postgres to call the function within itself rather than allowing Postgres to call your function. Hence, if you want your function to be used on different architectures, we recommend that you do not overload C function names.

There is a clever trick to get around the problem just described. Since there is no problem overloading SQL functions, you can define a set of C functions with different names and then define a set of identically-named SQL function wrappers that take the appropriate argument types and call the matching C function.

Another solution is not to use dynamic loading, but to link your functions into the backend statically and declare them as INTERNAL functions. Then, the functions must all have distinct C names but

they can be declared with the same SQL names (as long as their argument types differ, of course). This way avoids the overhead of an SQL wrapper function, at the cost of more effort to prepare a custom backend executable. (This option is only available in version 6.5 and later, since prior versions required internal functions to have the same name in SQL as in the C code.)

Chapter 40. Extending SQL: Types

As previously mentioned, there are two kinds of types in Postgres: base types (defined in a programming language) and composite types (instances). Examples in this section up to interfacing indices can be found in `complex.sql` and `complex.c`. Composite examples are in `funcs.sql`.

40.1. User-Defined Types

40.1.1. Functions Needed for a User-Defined Type

A user-defined type must always have input and output functions. These functions determine how the type appears in strings (for input by the user and output to the user) and how the type is organized in memory. The input function takes a null-delimited character string as its input and returns the internal (in memory) representation of the type. The output function takes the internal representation of the type and returns a null delimited character string. Suppose we want to define a complex type which represents complex numbers. Naturally, we choose to represent a complex in memory as the following C structure:

```
typedef struct Complex {
    double      x;
    double      y;
} Complex;
```

and a string of the form (x,y) as the external string representation. These functions are usually not hard to write, especially the output function. However, there are a number of points to remember:

- When defining your external (string) representation, remember that you must eventually write a complete and robust parser for that representation as your input function!

  ```
  Complex *
  complex_in(char *str)
  {
      double x, y;
      Complex *result;
      if (sscanf(str, " ( %lf , %lf )", &x, &y) != 2) {
          elog(NOTICE, "complex_in: error in parsing
          return NULL;
      }
      result = (Complex *)palloc(sizeof(Complex));
      result->x = x;
      result->y = y;
      return (result);
  }
  ```

  The output function can simply be:

  ```
  char *
  complex_out(Complex *complex)
  {
      char *result;
      if (complex == NULL)
          return(NULL);
      result = (char *) palloc(60);
      sprintf(result, "(%g,%g)", complex->x, complex->y);
      return(result);
  ```

```
}
```

- You should try to make the input and output functions inverses of each other. If you do not, you will have severe problems when you need to dump your data into a file and then read it back in (say, into someone else's database on another computer). This is a particularly common problem when floating-point numbers are involved.

To define the complex type, we need to create the two user-defined functions complex_in and complex_out before creating the type:

```
CREATE FUNCTION complex_in(opaque)
    RETURNS complex
    AS 'PGROOT/tutorial/obj/complex.so'
    LANGUAGE 'c';

CREATE FUNCTION complex_out(opaque)
    RETURNS opaque
    AS 'PGROOT/tutorial/obj/complex.so'
    LANGUAGE 'c';

CREATE TYPE complex (
    internallength = 16,
    input = complex_in,
    output = complex_out
);
```

As discussed earlier, Postgres fully supports arrays of base types. Additionally, Postgres supports arrays of user-defined types as well. When you define a type, Postgres automatically provides support for arrays of that type. For historical reasons, the array type has the same name as the user-defined type with the underscore character _ prepended. Composite types do not need any function defined on them, since the system already understands what they look like inside.

40.1.2. Large Objects

The types discussed to this point are all "small" objects -- that is, they are smaller than 8KB in size.

Note

1024 longwords == 8192 bytes. In fact, the type must be considerably smaller than 8192 bytes, since the Postgres tuple and page overhead must also fit into this 8KB limitation. The actual value that fits depends on the machine architecture.

If you require a larger type for something like a document retrieval system or for storing bitmaps, you will need to use the Postgres large object interface, or will need to recompile the Postgres backend to use internal storage blocks greater than 8kbytes..

Chapter 41. Extending SQL: Operators

Postgres supports left unary, right unary and binary operators. Operators can be overloaded; that is, the same operator name can be used for different operators that have different numbers and types of arguments. If there is an ambiguous situation and the system cannot determine the correct operator to use, it will return an error. You may have to typecast the left and/or right operands to help it understand which operator you meant to use.

Every operator is "syntactic sugar" for a call to an underlying function that does the real work; so you must first create the underlying function before you can create the operator. However, an operator is *not* merely syntactic sugar, because it carries additional information that helps the query planner optimize queries that use the operator. Much of this chapter will be devoted to explaining that additional information.

Here is an example of creating an operator for adding two complex numbers. We assume we've already created the definition of type complex. First we need a function that does the work; then we can define the operator:

```
CREATE FUNCTION complex_add(complex, complex)
    RETURNS complex
    AS '$PWD/obj/complex.so'
    LANGUAGE 'c';

CREATE OPERATOR + (
    leftarg = complex,
    rightarg = complex,
    procedure = complex_add,
    commutator = +
);
```

Now we can do:

```
SELECT (a + b) AS c FROM test_complex;

+----------------+
|c               |
+----------------+
|(5.2,6.05)      |
+----------------+
|(133.42,144.95) |
+----------------+
```

We've shown how to create a binary operator here. To create unary operators, just omit one of leftarg (for left unary) or rightarg (for right unary). The procedure clause and the argument clauses are the only required items in CREATE OPERATOR. The COMMUTATOR clause shown in the example is an optional hint to the query optimizer. Further details about COMMUTATOR and other optimizer hints appear below.

41.1. Operator Optimization Information

Author

Written by Tom Lane.

A Postgres operator definition can include several optional clauses that tell the system useful things about how the operator behaves. These clauses should be provided whenever appropriate, because they can make for considerable speedups in execution of queries that use the operator. But if you provide them, you must be sure that they are right! Incorrect use of an optimization clause can result in backend crashes, subtly wrong output, or other Bad Things. You can always leave out an optimization clause if you are not sure about it; the only consequence is that queries might run slower than they need to.

Additional optimization clauses might be added in future versions of Postgres. The ones described here are all the ones that release 6.5 understands.

41.1.1. COMMUTATOR

The COMMUTATOR clause, if provided, names an operator that is the commutator of the operator being defined. We say that operator A is the commutator of operator B if (x A y) equals (y B x) for all possible input values x,y. Notice that B is also the commutator of A. For example, operators '<' and '>' for a particular datatype are usually each others' commutators, and operator '+' is usually commutative with itself. But operator '-' is usually not commutative with anything.

The left argument type of a commuted operator is the same as the right argument type of its commutator, and vice versa. So the name of the commutator operator is all that Postgres needs to be given to look up the commutator, and that's all that need be provided in the COMMUTATOR clause.

When you are defining a self-commutative operator, you just do it. When you are defining a pair of commutative operators, things are a little trickier: how can the first one to be defined refer to the other one, which you haven't defined yet? There are two solutions to this problem:

- One way is to omit the COMMUTATOR clause in the first operator that you define, and then provide one in the second operator's definition. Since Postgres knows that commutative operators come in pairs, when it sees the second definition it will automatically go back and fill in the missing COMMUTATOR clause in the first definition.

- The other, more straightforward way is just to include COMMUTATOR clauses in both definitions. When Postgres processes the first definition and realizes that COMMUTATOR refers to a non-existent operator, the system will make a dummy entry for that operator in the system's pg_operator table. This dummy entry will have valid data only for the operator name, left and right argument types, and result type, since that's all that Postgres can deduce at this point. The first operator's catalog entry will link to this dummy entry. Later, when you define the second operator, the system updates the dummy entry with the additional information from the second definition. If you try to use the dummy operator before it's been filled in, you'll just get an error message. (Note: this procedure did not work reliably in Postgres versions before 6.5, but it is now the recommended way to do things.)

41.1.2. NEGATOR

The NEGATOR clause, if provided, names an operator that is the negator of the operator being defined. We say that operator A is the negator of operator B if both return boolean results and (x A y) equals NOT (x B y) for all possible inputs x,y. Notice that B is also the negator of A. For example, '<' and '>=' are a negator pair for most datatypes. An operator can never be validly be its own negator.

Unlike COMMUTATOR, a pair of unary operators could validly be marked as each others' negators; that would mean (A x) equals NOT (B x) for all x, or the equivalent for right-unary operators.

An operator's negator must have the same left and/or right argument types as the operator itself, so just as with COMMUTATOR, only the operator name need be given in the NEGATOR clause.

Providing NEGATOR is very helpful to the query optimizer since it allows expressions like NOT (x = y) to be simplified into x <> y. This comes up more often than you might think, because NOTs can be inserted as a consequence of other rearrangements.

Pairs of negator operators can be defined using the same methods explained above for commutator pairs.

41.1.3. RESTRICT

The RESTRICT clause, if provided, names a restriction selectivity estimation function for the operator (note that this is a function name, not an operator name). RESTRICT clauses only make sense for binary operators that return boolean. The idea behind a restriction selectivity estimator is to guess what fraction of the rows in a table will satisfy a WHERE-clause condition of the form

```
field OP constant
```

for the current operator and a particular constant value. This assists the optimizer by giving it some idea of how many rows will be eliminated by WHERE clauses that have this form. (What happens if the constant is on the left, you may be wondering? Well, that's one of the things that COMMUTATOR is for...)

Writing new restriction selectivity estimation functions is far beyond the scope of this chapter, but fortunately you can usually just use one of the system's standard estimators for many of your own operators. These are the standard restriction estimators:

```
eqsel  for =
neqsel  for <>
scalarltsel for < or <=
scalargtsel for > or >=
```

It might seem a little odd that these are the categories, but they make sense if you think about it. '=' will typically accept only a small fraction of the rows in a table; '<>' will typically reject only a small fraction. '<' will accept a fraction that depends on where the given constant falls in the range of values for that table column (which, it just so happens, is information collected by VACUUM ANALYZE and made available to the selectivity estimator). '<=' will accept a slightly larger fraction than '<' for the same comparison constant, but they're close enough to not be worth distinguishing, especially since we're not likely to do better than a rough guess anyhow. Similar remarks apply to '>' and '>='.

You can frequently get away with using either eqsel or neqsel for operators that have very high or very low selectivity, even if they aren't really equality or inequality. For example, the approximate-equality geometric operators use eqsel on the assumption that they'll usually only match a small fraction of the entries in a table.

You can use scalarltsel and scalargtsel for comparisons on datatypes that have some sensible means of being converted into numeric scalars for range comparisons. If possible, add the datatype to those understood by the routine convert_to_scalar() in src/backend/utils/adt/selfuncs.c. (Eventually, this routine should be replaced by per-datatype functions identified through a column of the pg_type table; but that hasn't happened yet.) If you do not do this, things will still work, but the optimizer's estimates won't be as good as they could be.

There are additional selectivity functions designed for geometric operators in src/backend/utils/adt/geo_selfuncs.c: areasel, positionsel, and contsel. At this writing these are just stubs, but you may want to use them (or even better, improve them) anyway.

41.1.4. JOIN

The JOIN clause, if provided, names a join selectivity estimation function for the operator (note that this is a function name, not an operator name). JOIN clauses only make sense for binary operators that return boolean. The idea behind a join selectivity estimator is to guess what fraction of the rows in a pair of tables will satisfy a WHERE-clause condition of the form

```
table1.field1 OP table2.field2
```

for the current operator. As with the RESTRICT clause, this helps the optimizer very substantially by letting it figure out which of several possible join sequences is likely to take the least work.

As before, this chapter will make no attempt to explain how to write a join selectivity estimator function, but will just suggest that you use one of the standard estimators if one is applicable:

```
eqjoinsel for =
neqjoinsel for <>
scalarltjoinsel for < or <=
scalargtjoinsel for > or >=
areajoinsel for 2D area-based comparisons
positionjoinsel for 2D position-based comparisons
contjoinsel for 2D containment-based comparisons
```

41.1.5. HASHES

The HASHES clause, if present, tells the system that it is OK to use the hash join method for a join based on this operator. HASHES only makes sense for binary operators that return boolean, and in practice the operator had better be equality for some data type.

The assumption underlying hash join is that the join operator can only return TRUE for pairs of left and right values that hash to the same hash code. If two values get put in different hash buckets, the join will never compare them at all, implicitly assuming that the result of the join operator must be FALSE. So it never makes sense to specify HASHES for operators that do not represent equality.

In fact, logical equality is not good enough either; the operator had better represent pure bitwise equality, because the hash function will be computed on the memory representation of the values regardless of what the bits mean. For example, equality of time intervals is not bitwise equality; the interval equality operator considers two time intervals equal if they have the same duration, whether or not their endpoints are identical. What this means is that a join using "=" between interval fields would yield different results if implemented as a hash join than if implemented another way, because a large fraction of the pairs that should match will hash to different values and will never be compared by the hash join. But if the optimizer chose to use a different kind of join, all the pairs that the equality operator says are equal will be found. We don't want that kind of inconsistency, so we don't mark interval equality as hashable.

There are also machine-dependent ways in which a hash join might fail to do the right thing. For example, if your datatype is a structure in which there may be uninteresting pad bits, it's unsafe to mark the equality operator HASHES. (Unless, perhaps, you write your other operators to ensure that the unused bits are always zero.) Another example is that the FLOAT datatypes are unsafe for hash joins. On machines that meet the IEEE floating point standard, minus zero and plus zero are different values (different bit patterns) but they are defined to compare equal. So, if float equality were marked HASHES, a minus zero and a plus zero would probably not be matched up by a hash join, but they would be matched up by any other join process.

The bottom line is that you should probably only use HASHES for equality operators that are (or could be) implemented by memcmp().

41.1.6. SORT1 and SORT2

The SORT clauses, if present, tell the system that it is permissible to use the merge join method for a join based on the current operator. Both must be specified if either is. The current operator must be equality for some pair of data types, and the SORT1 and SORT2 clauses name the ordering operator ('<' operator) for the left and right-side data types respectively.

Merge join is based on the idea of sorting the left and righthand tables into order and then scanning them in parallel. So, both data types must be capable of being fully ordered, and the join operator must be one that can only succeed for pairs of values that fall at the "same place" in the sort order. In practice this means that the join operator must behave like equality. But unlike hashjoin, where the left and right data types had better be the same (or at least bitwise equivalent), it is possible to mergejoin two distinct data types so long as they are logically compatible. For example, the int2-versus-int4 equality operator is mergejoinable. We only need sorting operators that will bring both datatypes into a logically compatible sequence.

When specifying merge sort operators, the current operator and both referenced operators must return boolean; the SORT1 operator must have both input datatypes equal to the current operator's left argument type, and the SORT2 operator must have both input datatypes equal to the current operator's right argument type. (As with COMMUTATOR and NEGATOR, this means that the operator name is sufficient to specify the operator, and the system is able to make dummy operator entries if you happen to define the equality operator before the other ones.)

In practice you should only write SORT clauses for an '=' operator, and the two referenced operators should always be named '<'. Trying to use merge join with operators named anything else will result in hopeless confusion, for reasons we'll see in a moment.

There are additional restrictions on operators that you mark mergejoinable. These restrictions are not currently checked by CREATE OPERATOR, but a merge join may fail at runtime if any are not true:

- The mergejoinable equality operator must have a commutator (itself if the two data types are the same, or a related equality operator if they are different).

- There must be '<' and '>' ordering operators having the same left and right input datatypes as the mergejoinable operator itself. These operators *must* be named '<' and '>'; you do not have any choice in the matter, since there is no provision for specifying them explicitly. Note that if the left and right data types are different, neither of these operators is the same as either SORT operator. But they had better order the data values compatibly with the SORT operators, or mergejoin will fail to work.

Chapter 42. Extending SQL: Aggregates

Aggregate functions in Postgres are expressed as *state values* and *state transition functions*. That is, an aggregate can be defined in terms of state that is modified whenever an input item is processed. To define a new aggregate function, one selects a datatype for the state value, an initial value for the state, and a state transition function. The state transition function is just an ordinary function that could also be used outside the context of the aggregate.

Actually, in order to make it easier to construct useful aggregates from existing functions, an aggregate can have one or two separate state values, one or two transition functions to update those state values, and a *final function* that computes the actual aggregate result from the ending state values.

Thus there can be as many as four datatypes involved: the type of the input data items, the type of the aggregate's result, and the types of the two state values. Only the input and result datatypes are seen by a user of the aggregate.

Some state transition functions need to look at each successive input to compute the next state value, while others ignore the specific input value and simply update their internal state. (The most useful example of the second kind is a running count of the number of input items.) The Postgres aggregate machinery defines sfunc1 for an aggregate as a function that is passed both the old state value and the current input value, while sfunc2 is a function that is passed only the old state value.

If we define an aggregate that uses only sfunc1, we have an aggregate that computes a running function of the attribute values from each instance. "Sum" is an example of this kind of aggregate. "Sum" starts at zero and always adds the current instance's value to its running total. For example, if we want to make a Sum aggregate to work on a datatype for complex numbers, we only need the addition function for that datatype. The aggregate definition is:

```
CREATE AGGREGATE complex_sum (
    sfunc1 = complex_add,
    basetype = complex,
    stype1 = complex,
    initcond1 = '(0,0)'
);

SELECT complex_sum(a) FROM test_complex;

         +------------+
         |complex_sum |
         +------------+
         |(34,53.9)   |
         +------------+
```

(In practice, we'd just name the aggregate "sum", and rely on Postgres to figure out which kind of sum to apply to a complex column.)

If we define only sfunc2, we are specifying an aggregate that computes a running function that is independent of the attribute values from each instance. "Count" is the most common example of this kind of aggregate. "Count" starts at zero and adds one to its running total for each instance, ignoring the instance value. Here, we use the built-in int4inc routine to do the work for us. This routine increments (adds one to) its argument.

```
CREATE AGGREGATE my_count (
    sfunc2 = int4inc, -- add one
    basetype = int4,
```

```
    stype2 = int4,
    initcond2 = '0'
);

SELECT my_count(*) as emp_count from EMP;

         +----------+
         |emp_count |
         +----------+
         |5         |
         +----------+
```

"Average" is an example of an aggregate that requires both a function to compute the running sum and a function to compute the running count. When all of the instances have been processed, the final answer for the aggregate is the running sum divided by the running count. We use the int4pl and int4inc routines we used before as well as the Postgres integer division routine, int4div, to compute the division of the sum by the count.

```
CREATE AGGREGATE my_average (
    sfunc1 = int4pl,     --  sum
    basetype = int4,
    stype1 = int4,
    sfunc2 = int4inc,    -- count
    stype2 = int4,
    finalfunc = int4div, -- division
    initcond1 = '0',
    initcond2 = '0'
);

SELECT my_average(salary) as emp_average FROM EMP;

         +------------+
         |emp_average |
         +------------+
         |1640        |
         +------------+
```

For further details see `CREATE AGGREGATE` in *The PostgreSQL User's Guide*.

Chapter 43. The Postgres Rule System

Production rule systems are conceptually simple, but there are many subtle points involved in actually using them. Some of these points and the theoretical foundations of the Postgres rule system can be found in [Stonebraker et al, ACM, 1990].

Some other database systems define active database rules. These are usually stored procedures and triggers and are implemented in Postgres as functions and triggers.

The query rewrite rule system (the "rule system" from now on) is totally different from stored procedures and triggers. It modifies queries to take rules into consideration, and then passes the modified query to the query optimizer for execution. It is very powerful, and can be used for many things such as query language procedures, views, and versions. The power of this rule system is discussed in [Ong and Goh, 1990] as well as [Stonebraker et al, ACM, 1990].

43.1. What is a Querytree?

To understand how the rule system works it is necessary to know when it is invoked and what it's input and results are.

The rule system is located between the query parser and the optimizer. It takes the output of the parser, one querytree, and the rewrite rules from the `pg_rewrite` catalog, which are querytrees too with some extra information, and creates zero or many querytrees as result. So it's input and output are always things the parser itself could have produced and thus, anything it sees is basically representable as an SQL statement.

Now what is a querytree? It is an internal representation of an SQL statement where the single parts that built it are stored separately. These querytrees are visible when starting the Postgres backend with debuglevel 4 and typing queries into the interactive backend interface. The rule actions in the `pg_rewrite` system catalog are also stored as querytrees. They are not formatted like the debug output, but they contain exactly the same information.

Reading a querytree requires some experience and it was a hard time when I started to work on the rule system. I can remember that I was standing at the coffee machine and I saw the cup in a targetlist, water and coffee powder in a rangetable and all the buttons in a qualification expression. Since SQL representations of querytrees are sufficient to understand the rule system, this document will not teach how to read them. It might help to learn it and the naming conventions are required in the later following descriptions.

43.1.1. The Parts of a Querytree

When reading the SQL representations of the querytrees in this document it is necessary to be able to identify the parts the statement is broken into when it is in the querytree structure. The parts of a querytree are

the commandtype

This is a simple value telling which command (SELECT, INSERT, UPDATE, DELETE) produced the parsetree.

the rangetable

The rangtable is a list of relations that are used in the query. In a SELECT statement that are the relations given after the FROM keyword.

Every rangetable entry identifies a table or view and tells by which name it is called in the other parts of the query. In the querytree the rangetable entries are referenced by index rather than by

name, so here it doesn't matter if there are duplicate names as it would in an SQL statement. This can happen after the rangetables of rules have been merged in. The examples in this document will not have this situation.

the resultrelation

This is an index into the rangetable that identifies the relation where the results of the query go.

SELECT queries normally don't have a result relation. The special case of a SELECT INTO is mostly identical to a CREATE TABLE, INSERT ... SELECT sequence and is not discussed separately here.

On INSERT, UPDATE and DELETE queries the resultrelation is the table (or view!) where the changes take effect.

the targetlist

The targetlist is a list of expressions that define the result of the query. In the case of a SELECT, the expressions are what builds the final output of the query. They are the expressions between the SELECT and the FROM keywords (* is just an abbreviation for all the attribute names of a relation).

DELETE queries don't need a targetlist because they don't produce any result. In fact the optimizer will add a special entry to the empty targetlist. But this is after the rule system and will be discussed later. For the rule system the targetlist is empty.

In INSERT queries the targetlist describes the new rows that should go into the resultrelation. Missing columns of the resultrelation will be added by the optimizer with a constant NULL expression. It is the expressions in the VALUES clause or the ones from the SELECT clause on INSERT ... SELECT.

On UPDATE queries, it describes the new rows that should replace the old ones. Here now the optimizer will add missing columns by inserting expressions that put the values from the old rows into the new one. And it will add the special entry like for DELETE too. It is the expressions from the SET attribute = expression part of the query.

Every entry in the targetlist contains an expression that can be a constant value, a variable pointing to an attribute of one of the relations in the rangetable, a parameter or an expression tree made of function calls, constants, variables, operators etc.

the qualification

The queries qualification is an expression much like one of those contained in the targetlist entries. The result value of this expression is a boolean that tells if the operation (INSERT, UPDATE, DELETE or SELECT) for the final result row should be executed or not. It is the WHERE clause of an SQL statement.

the others

The other parts of the querytree like the ORDER BY clause arent of interest here. The rule system substitutes entries there while applying rules, but that doesn't have much to do with the fundamentals of the rule system. GROUP BY is a special thing when it appears in a view definition and still needs to be documented.

43.2. Views and the Rule System

43.2.1. Implementation of Views in Postgres

Views in Postgres are implemented using the rule system. In fact there is absolutely no difference between a

```
CREATE VIEW myview AS SELECT * FROM mytab;
```

compared against the two commands

```
CREATE TABLE myview (same attribute list as for mytab);
CREATE RULE "_RETmyview" AS ON SELECT TO myview DO INSTEAD
    SELECT * FROM mytab;
```

because this is exactly what the CREATE VIEW command does internally. This has some side effects. One of them is that the information about a view in the Postgres system catalogs is exactly the same as it is for a table. So for the query parsers, there is absolutely no difference between a table and a view. They are the same thing - relations. That is the important one for now.

43.2.2. How SELECT Rules Work

Rules ON SELECT are applied to all queries as the last step, even if the command given is an INSERT, UPDATE or DELETE. And they have different semantics from the others in that they modify the parsetree in place instead of creating a new one. So SELECT rules are described first.

Currently, there could be only one action and it must be a SELECT action that is INSTEAD. This restriction was required to make rules safe enough to open them for ordinary users and it restricts rules ON SELECT to real view rules.

The example for this document are two join views that do some calculations and some more views using them in turn. One of the two first views is customized later by adding rules for INSERT, UPDATE and DELETE operations so that the final result will be a view that behaves like a real table with some magic functionality. It is not such a simple example to start from and this makes things harder to get into. But it's better to have one example that covers all the points discussed step by step rather than having many different ones that might mix up in mind.

The database needed to play on the examples is named al_bundy. You'll see soon why this is the database name. And it needs the procedural language PL/pgSQL installed, because we need a little min() function returning the lower of 2 integer values. We create that as

```
CREATE FUNCTION min(integer, integer) RETURNS integer AS
    'BEGIN
        IF $1 < $2 THEN
            RETURN $1;
        END IF;
        RETURN $2;
    END;'
LANGUAGE 'plpgsql';
```

The real tables we need in the first two rule system descripitons are these:

```
CREATE TABLE shoe_data (
    shoename   char(10),      -- primary key
    sh_avail   integer,       -- available # of pairs
    slcolor    char(10),      -- preferred shoelace color
    slminlen   float,         -- miminum shoelace length
    slmaxlen   float,         -- maximum shoelace length
    slunit     char(8)        -- length unit
);

CREATE TABLE shoelace_data (
    sl_name    char(10),      -- primary key
    sl_avail   integer,       -- available # of pairs
    sl_color   char(10),      -- shoelace color
    sl_len     float,         -- shoelace length
    sl_unit    char(8)        -- length unit
);
```

```
CREATE TABLE unit (
    un_name    char(8),       -- the primary key
    un_fact    float          -- factor to transform to cm
);
```

I think most of us wear shoes and can realize that this is really useful data. Well there are shoes out in the world that don't require shoelaces, but this doesn't make Al's life easier and so we ignore it.

The views are created as

```
CREATE VIEW shoe AS
    SELECT sh.shoename,
           sh.sh_avail,
           sh.slcolor,
           sh.slminlen,
           sh.slminlen * un.un_fact AS slminlen_cm,
           sh.slmaxlen,
           sh.slmaxlen * un.un_fact AS slmaxlen_cm,
           sh.slunit
      FROM shoe_data sh, unit un
     WHERE sh.slunit = un.un_name;

CREATE VIEW shoelace AS
    SELECT s.sl_name,
           s.sl_avail,
           s.sl_color,
           s.sl_len,
           s.sl_unit,
           s.sl_len * u.un_fact AS sl_len_cm
      FROM shoelace_data s, unit u
     WHERE s.sl_unit = u.un_name;

CREATE VIEW shoe_ready AS
    SELECT rsh.shoename,
           rsh.sh_avail,
           rsl.sl_name,
           rsl.sl_avail,
           min(rsh.sh_avail, rsl.sl_avail) AS total_avail
      FROM shoe rsh, shoelace rsl
     WHERE rsl.sl_color = rsh.slcolor
       AND rsl.sl_len_cm >= rsh.slminlen_cm
       AND rsl.sl_len_cm <= rsh.slmaxlen_cm;
```

The CREATE VIEW command for the `shoelace` view (which is the simplest one we have) will create a relation shoelace and an entry in `pg_rewrite` that tells that there is a rewrite rule that must be applied whenever the relation shoelace is referenced in a queries rangetable. The rule has no rule qualification (discussed in the non SELECT rules since SELECT rules currently cannot have them) and it is INSTEAD. Note that rule qualifications are not the same as query qualifications! The rules action has a qualification.

The rules action is one querytree that is an exact copy of the SELECT statement in the view creation command.

Note

The two extra range table entries for NEW and OLD (named *NEW* and *CURRENT* for historical reasons in the printed querytree) you can see in the `pg_rewrite` entry aren't of interest for SELECT rules.

Now we populate `unit`, `shoe_data` and `shoelace_data` and Al types the first SELECT in his life:

```
al_bundy=> INSERT INTO unit VALUES ('cm', 1.0);
al_bundy=> INSERT INTO unit VALUES ('m', 100.0);
al_bundy=> INSERT INTO unit VALUES ('inch', 2.54);
al_bundy=>
al_bundy=> INSERT INTO shoe_data VALUES
al_bundy->     ('sh1', 2, 'black', 70.0, 90.0, 'cm');
al_bundy=> INSERT INTO shoe_data VALUES
al_bundy->     ('sh2', 0, 'black', 30.0, 40.0, 'inch');
al_bundy=> INSERT INTO shoe_data VALUES
al_bundy->     ('sh3', 4, 'brown', 50.0, 65.0, 'cm');
al_bundy=> INSERT INTO shoe_data VALUES
al_bundy->     ('sh4', 3, 'brown', 40.0, 50.0, 'inch');
al_bundy=>
al_bundy=> INSERT INTO shoelace_data VALUES
al_bundy->     ('sl1', 5, 'black', 80.0, 'cm');
al_bundy=> INSERT INTO shoelace_data VALUES
al_bundy->     ('sl2', 6, 'black', 100.0, 'cm');
al_bundy=> INSERT INTO shoelace_data VALUES
al_bundy->     ('sl3', 0, 'black', 35.0 , 'inch');
al_bundy=> INSERT INTO shoelace_data VALUES
al_bundy->     ('sl4', 8, 'black', 40.0 , 'inch');
al_bundy=> INSERT INTO shoelace_data VALUES
al_bundy->     ('sl5', 4, 'brown', 1.0 , 'm');
al_bundy=> INSERT INTO shoelace_data VALUES
al_bundy->     ('sl6', 0, 'brown', 0.9 , 'm');
al_bundy=> INSERT INTO shoelace_data VALUES
al_bundy->     ('sl7', 7, 'brown', 60 , 'cm');
al_bundy=> INSERT INTO shoelace_data VALUES
al_bundy->     ('sl8', 1, 'brown', 40 , 'inch');
al_bundy=>
al_bundy=> SELECT * FROM shoelace;
sl_name   |sl_avail|sl_color  |sl_len|sl_unit |sl_len_cm
----------+--------+----------+------+--------+---------
sl1       |       5|black     |    80|cm      |       80
sl2       |       6|black     |   100|cm      |      100
sl7       |       7|brown     |    60|cm      |       60
sl3       |       0|black     |    35|inch    |     88.9
sl4       |       8|black     |    40|inch    |    101.6
sl8       |       1|brown     |    40|inch    |    101.6
sl5       |       4|brown     |     1|m       |      100
sl6       |       0|brown     |   0.9|m       |       90
(8 rows)
```

It's the simplest SELECT Al can do on our views, so we take this to explain the basics of view rules. The 'SELECT * FROM shoelace' was interpreted by the parser and produced the parsetree

```
SELECT shoelace.sl_name, shoelace.sl_avail,
       shoelace.sl_color, shoelace.sl_len,
       shoelace.sl_unit, shoelace.sl_len_cm
  FROM shoelace shoelace;
```

and this is given to the rule system. The rule system walks through the rangetable and checks if there are rules in `pg_rewrite` for any relation. When processing the rangetable entry for `shoelace` (the only one up to now) it finds the rule '_RETshoelace' with the parsetree

```
SELECT s.sl_name, s.sl_avail,
       s.sl_color, s.sl_len, s.sl_unit,
       float8mul(s.sl_len, u.un_fact) AS sl_len_cm
  FROM shoelace *OLD*, shoelace *NEW*,
```

```
        shoelace_data s, unit u
 WHERE bpchareq(s.sl_unit, u.un_name);
```

Note that the parser changed the calculation and qualification into calls to the appropriate functions. But in fact this changes nothing. The first step in rewriting is merging the two rangetables. The resulting parsetree then reads

```
SELECT shoelace.sl_name, shoelace.sl_avail,
       shoelace.sl_color, shoelace.sl_len,
       shoelace.sl_unit, shoelace.sl_len_cm
  FROM shoelace shoelace, shoelace *OLD*,
       shoelace *NEW*, shoelace_data s,
       unit u;
```

In step 2 it adds the qualification from the rule action to the parsetree resulting in

```
SELECT shoelace.sl_name, shoelace.sl_avail,
       shoelace.sl_color, shoelace.sl_len,
       shoelace.sl_unit, shoelace.sl_len_cm
  FROM shoelace shoelace, shoelace *OLD*,
       shoelace *NEW*, shoelace_data s,
       unit u
 WHERE bpchareq(s.sl_unit, u.un_name);
```

And in step 3 it replaces all the variables in the parsetree, that reference the rangetable entry (the one for `shoelace` that is currently processed) by the corresponding targetlist expressions from the rule action. This results in the final query

```
SELECT s.sl_name, s.sl_avail,
       s.sl_color, s.sl_len,
       s.sl_unit, float8mul(s.sl_len, u.un_fact) AS sl_len_cm
  FROM shoelace shoelace, shoelace *OLD*,
       shoelace *NEW*, shoelace_data s,
       unit u
 WHERE bpchareq(s.sl_unit, u.un_name);
```

Turning this back into a real SQL statement a human user would type reads

```
SELECT s.sl_name, s.sl_avail,
       s.sl_color, s.sl_len,
       s.sl_unit, s.sl_len * u.un_fact AS sl_len_cm
  FROM shoelace_data s, unit u
 WHERE s.sl_unit = u.un_name;
```

That was the first rule applied. While this was done, the rangetable has grown. So the rule system continues checking the range table entries. The next one is number 2 (shoelace *OLD*). Relation `shoelace` has a rule, but this rangetable entry isn't referenced in any of the variables of the parsetree, so it is ignored. Since all the remaining rangetable entries either have no rules in `pg_rewrite` or aren't referenced, it reaches the end of the rangetable. Rewriting is complete and the above is the final result given into the optimizer. The optimizer ignores the extra rangetable entries that aren't referenced by variables in the parsetree and the plan produced by the planner/optimizer would be exactly the same as if Al had typed the above SELECT query instead of the view selection.

Now we face Al with the problem that the Blues Brothers appear in his shop and want to buy some new shoes, and as the Blues Brothers are, they want to wear the same shoes. And they want to wear them immediately, so they need shoelaces too.

Al needs to know for which shoes currently in the store he has the matching shoelaces (color and size) and where the total number of exactly matching pairs is greater or equal to two. We theach him how to do and he asks his database:

```
al_bundy=> SELECT * FROM shoe_ready WHERE total_avail >= 2;
shoename  |sh_avail|sl_name   |sl_avail|total_avail
----------+--------+----------+--------+-----------
sh1       |       2|sl1       |       5|          2
sh3       |       4|sl7       |       7|          4
(2 rows)
```

Al is a shoe guru and so he knows that only shoes of type sh1 would fit (shoelace sl7 is brown and shoes that need brown shoelaces aren't shoes the Blues Brothers would ever wear).

The output of the parser this time is the parsetree

```
SELECT shoe_ready.shoename, shoe_ready.sh_avail,
       shoe_ready.sl_name, shoe_ready.sl_avail,
       shoe_ready.total_avail
  FROM shoe_ready shoe_ready
 WHERE int4ge(shoe_ready.total_avail, 2);
```

The first rule applied will be that one for the `shoe_ready` relation and it results in the parsetree

```
SELECT rsh.shoename, rsh.sh_avail,
       rsl.sl_name, rsl.sl_avail,
       min(rsh.sh_avail, rsl.sl_avail) AS total_avail
  FROM shoe_ready shoe_ready, shoe_ready *OLD*,
       shoe_ready *NEW*, shoe rsh,
       shoelace rsl
 WHERE int4ge(min(rsh.sh_avail, rsl.sl_avail), 2)
   AND (bpchareq(rsl.sl_color, rsh.slcolor)
        AND float8ge(rsl.sl_len_cm, rsh.slminlen_cm)
        AND float8le(rsl.sl_len_cm, rsh.slmaxlen_cm)
       );
```

In reality the AND clauses in the qualification will be operator nodes of type AND with a left and right expression. But that makes it lesser readable as it already is, and there are more rules to apply. So I only put them into some parantheses to group them into logical units in the order they where added and we continue with the rule for relation `shoe` as it is the next rangetable entry that is referenced and has a rule. The result of applying it is

```
SELECT sh.shoename, sh.sh_avail,
       rsl.sl_name, rsl.sl_avail,
       min(sh.sh_avail, rsl.sl_avail) AS total_avail,
  FROM shoe_ready shoe_ready, shoe_ready *OLD*,
       shoe_ready *NEW*, shoe rsh,
       shoelace rsl, shoe *OLD*,
       shoe *NEW*, shoe_data sh,
       unit un
 WHERE (int4ge(min(sh.sh_avail, rsl.sl_avail), 2)
        AND (bpchareq(rsl.sl_color, sh.slcolor)
             AND float8ge(rsl.sl_len_cm,
                          float8mul(sh.slminlen, un.un_fact))
             AND float8le(rsl.sl_len_cm,
                          float8mul(sh.slmaxlen, un.un_fact))
            )
       )
   AND bpchareq(sh.slunit, un.un_name);
```

And finally we apply the already well known rule for `shoelace` (this time on a parsetree that is a little more complex) and get

```
SELECT sh.shoename, sh.sh_avail,
```

```
          s.sl_name, s.sl_avail,
          min(sh.sh_avail, s.sl_avail) AS total_avail
     FROM shoe_ready shoe_ready, shoe_ready *OLD*,
          shoe_ready *NEW*, shoe rsh,
          shoelace rsl, shoe *OLD*,
          shoe *NEW*, shoe_data sh,
          unit un, shoelace *OLD*,
          shoelace *NEW*, shoelace_data s,
          unit u
    WHERE (    (int4ge(min(sh.sh_avail, s.sl_avail), 2)
                AND (bpchareq(s.sl_color, sh.slcolor)
                     AND float8ge(float8mul(s.sl_len, u.un_fact),
                                  float8mul(sh.slminlen,
un.un_fact))
                     AND float8le(float8mul(s.sl_len, u.un_fact),
                                  float8mul(sh.slmaxlen,
un.un_fact))
                    )
               )
           AND bpchareq(sh.slunit, un.un_name)
          )
      AND bpchareq(s.sl_unit, u.un_name);
```

Again we reduce it to a real SQL statement that is equivalent to the final output of the rule system:

```
SELECT sh.shoename, sh.sh_avail,
       s.sl_name, s.sl_avail,
       min(sh.sh_avail, s.sl_avail) AS total_avail
  FROM shoe_data sh, shoelace_data s, unit u, unit un
 WHERE min(sh.sh_avail, s.sl_avail) >= 2
   AND s.sl_color = sh.slcolor
   AND s.sl_len * u.un_fact >= sh.slminlen * un.un_fact
   AND s.sl_len * u.un_fact <= sh.slmaxlen * un.un_fact
   AND sh.sl_unit = un.un_name
   AND s.sl_unit = u.un_name;
```

Recursive processing of rules rewrote one SELECT from a view into a parsetree, that is equivalent to exactly that what Al had to type if there would be no views at all.

Note

There is currently no recursion stopping mechanism for view rules in the rule system (only for the other rules). This doesn't hurt much, because the only way to push this into an endless loop (blowing up the backend until it reaches the memory limit) is to create tables and then setup the view rules by hand with CREATE RULE in such a way, that one selects from the other that selects from the one. This could never happen if CREATE VIEW is used because on the first CREATE VIEW, the second relation does not exist and thus the first view cannot select from the second.

43.2.3. View Rules in Non-SELECT Statements

Two details of the parsetree aren't touched in the description of view rules above. These are the commandtype and the resultrelation. In fact, view rules don't need these informations.

There are only a few differences between a parsetree for a SELECT and one for any other command. Obviously they have another commandtype and this time the resultrelation points to the rangetable entry where the result should go. Anything else is absolutely the same. So having two tables t1 and t2 with attributes a and b, the parsetrees for the two statements

```
SELECT t2.b FROM t1, t2 WHERE t1.a = t2.a;

UPDATE t1 SET b = t2.b WHERE t1.a = t2.a;
```

are nearly identical.

- The rangetables contain entries for the tables t1 and t2.
- The targetlists contain one variable that points to attribute b of the rangetable entry for table t2.
- The qualification expressions compare the attributes a of both ranges for equality.

The consequence is, that both parsetrees result in similar execution plans. They are both joins over the two tables. For the UPDATE the missing columns from t1 are added to the targetlist by the optimizer and the final parsetree will read as

```
UPDATE t1 SET a = t1.a, b = t2.b WHERE t1.a = t2.a;
```

and thus the executor run over the join will produce exactly the same result set as a

```
SELECT t1.a, t2.b FROM t1, t2 WHERE t1.a = t2.a;
```

will do. But there is a little problem in UPDATE. The executor does not care what the results from the join it is doing are meant for. It just produces a result set of rows. The difference that one is a SELECT command and the other is an UPDATE is handled in the caller of the executor. The caller still knows (looking at the parsetree) that this is an UPDATE, and he knows that this result should go into table t1. But which of the 666 rows that are there has to be replaced by the new row? The plan executed is a join with a qualification that potentially could produce any number of rows between 0 and 666 in unknown order.

To resolve this problem, another entry is added to the targetlist in UPDATE and DELETE statements. The current tuple ID (ctid). This is a system attribute with a special feature. It contains the block and position in the block for the row. Knowing the table, the ctid can be used to find one specific row in a 1.5GB sized table containing millions of rows by fetching one single data block. After adding the ctid to the targetlist, the final result set could be defined as

```
SELECT t1.a, t2.b, t1.ctid FROM t1, t2 WHERE t1.a = t2.a;
```

Now another detail of Postgres enters the stage. At this moment, table rows aren't overwritten and this is why ABORT TRANSACTION is fast. In an UPDATE, the new result row is inserted into the table (after stripping ctid) and in the tuple header of the row that ctid pointed to the cmax and xmax entries are set to the current command counter and current transaction ID. Thus the old row is hidden and after the transaction commited the vacuum cleaner can really move it out.

Knowing that all, we can simply apply view rules in absolutely the same way to any command. There is no difference.

43.2.4. The Power of Views in Postgres

The above demonstrates how the rule system incorporates view definitions into the original parsetree. In the second example a simple SELECT from one view created a final parsetree that is a join of 4 tables (unit is used twice with different names).

43.2.4.1. Benefits

The benefit of implementing views with the rule system is, that the optimizer has all the information about which tables have to be scanned plus the relationships between these tables plus the restrictive qualifications from the views plus the qualifications from the original query in one single parsetree. And this is still the situation when the original query is already a join over views. Now the optimizer

has to decide which is the best path to execute the query. The more information the optimizer has, the better this decision can be. And the rule system as implemented in Postgres ensures, that this is all information available about the query up to now.

43.2.4.2. Concerns

There was a long time where the Postgres rule system was considered broken. The use of rules was not recommended and the only part working was view rules. And also these view rules gave problems because the rule system wasn't able to apply them properly on statements other than a SELECT (for example an UPDATE that used data from a view didn't work).

During that time, development moved on and many features were added to the parser and optimizer. The rule system got more and more out of sync with their capabilities and it became harder and harder to start fixing it. Thus, no one did.

For 6.4, someone locked the door, took a deep breath and shuffled that damned thing up. What came out was a rule system with the capabilities described in this document. But there are still some constructs not handled and some where it fails due to things that are currently not supported by the Postgres query optimizer.

- Views with aggregate columns have bad problems. Aggregate expressions in qualifications must be used in subselects. Currently it is not possible to do a join of two views, each having an aggregate column, and compare the two aggregate values in the qualification. In the meantime it is possible to put these aggregate expressions into functions with the appropriate arguments and use them in the view definition.

- Views of unions are currently not supported. Well it's easy to rewrite a simple SELECT into a union. But it is a little difficult if the view is part of a join doing an update.

- ORDER BY clauses in view definitions aren't supported.

- DISTINCT isn't supported in view definitions.

There is no good reason why the optimizer should not handle parsetree constructs that the parser could never produce due to limitations in the SQL syntax. The author hopes that these items disappear in the future.

43.2.5. Implementation Side Effects

Using the described rule system to implement views has a funny side effect. The following does not seem to work:

```
al_bundy=> INSERT INTO shoe (shoename, sh_avail, slcolor)
al_bundy->     VALUES ('sh5', 0, 'black');
INSERT 20128 1
al_bundy=> SELECT shoename, sh_avail, slcolor FROM shoe_data;
shoename  |sh_avail|slcolor
----------+--------+----------
sh1       |       2|black
sh3       |       4|brown
sh2       |       0|black
sh4       |       3|brown
(4 rows)
```

The interesting thing is that the return code for INSERT gave us an object ID and told that 1 row has been inserted. But it doesn't appear in `shoe_data`. Looking into the database directory we can see, that the database file for the view relation `shoe` seems now to have a data block. And that is definitely the case.

We can also issue a DELETE and if it does not have a qualification, it tells us that rows have been deleted and the next vacuum run will reset the file to zero size.

The reason for that behaviour is, that the parsetree for the INSERT does not reference the `shoe` relation in any variable. The targetlist contains only constant values. So there is no rule to apply and it goes down unchanged into execution and the row is inserted. And so for the DELETE.

To change this we can define rules that modify the behaviour of non-SELECT queries. This is the topic of the next section.

43.3. Rules on INSERT, UPDATE and DELETE

43.3.1. Differences to View Rules

Rules that are defined ON INSERT, UPDATE and DELETE are totally different from the view rules described in the previous section. First, their CREATE RULE command allows more:

- They can have no action.
- They can have multiple actions.
- The keyword INSTEAD is optional.
- The pseudo relations NEW and OLD become useful.
- They can have rule qualifications.

Second, they don't modify the parsetree in place. Instead they create zero or many new parsetrees and can throw away the original one.

43.3.2. How These Rules Work

Keep the syntax

```
CREATE RULE rule_name AS ON event
    TO object [WHERE rule_qualification]
    DO [INSTEAD] [action | (actions) | NOTHING];
```

in mind. In the following, "update rules" means rules that are defined ON INSERT, UPDATE or DELETE.

Update rules get applied by the rule system when the result relation and the commandtype of a parsetree are equal to the object and event given in the CREATE RULE command. For update rules, the rule system creates a list of parsetrees. Initially the parsetree list is empty. There can be zero (NOTHING keyword), one or multiple actions. To simplify, we look at a rule with one action. This rule can have a qualification or not and it can be INSTEAD or not.

What is a rule qualification? It is a restriction that tells when the actions of the rule should be done and when not. This qualification can only reference the NEW and/or OLD pseudo relations which are basically the relation given as object (but with a special meaning).

So we have four cases that produce the following parsetrees for a one-action rule.

- No qualification and not INSTEAD:
  - The parsetree from the rule action where the original parsetrees qualification has been added.
- No qualification but INSTEAD:

    - The parsetree from the rule action where the original parsetrees qualification has been added.

- Qualification given and not INSTEAD:

    - The parsetree from the rule action where the rule qualification and the original parsetrees qualification have been added.

- Qualification given and INSTEAD:

    - The parsetree from the rule action where the rule qualification and the original parsetrees qualification have been added.

    - The original parsetree where the negated rule qualification has been added.

Finally, if the rule is not INSTEAD, the unchanged original parsetree is added to the list. Since only qualified INSTEAD rules already add the original parsetree, we end up with a total maximum of two parsetrees for a rule with one action.

The parsetrees generated from rule actions are thrown into the rewrite system again and maybe more rules get applied resulting in more or less parsetrees. So the parsetrees in the rule actions must have either another commandtype or another resultrelation. Otherwise this recursive process will end up in a loop. There is a compiled in recursion limit of currently 10 iterations. If after 10 iterations there are still update rules to apply the rule system assumes a loop over multiple rule definitions and aborts the transaction.

The parsetrees found in the actions of the `pg_rewrite` system catalog are only templates. Since they can reference the rangetable entries for NEW and OLD, some substitutions have to be made before they can be used. For any reference to NEW, the targetlist of the original query is searched for a corresponding entry. If found, that entries expression is placed into the reference. Otherwise NEW means the same as OLD. Any reference to OLD is replaced by a reference to the rangetable entry which is the resultrelation.

43.3.2.1. A First Rule Step by Step

We want to trace changes to the sl_avail column in the `shoelace_data` relation. So we setup a log table and a rule that writes us entries every time and UPDATE is performed on `shoelace_data`.

```
CREATE TABLE shoelace_log (
    sl_name    char(10),      -- shoelace changed
    sl_avail   integer,       -- new available value
    log_who    name,          -- who did it
    log_when   datetime       -- when
);

CREATE RULE log_shoelace AS ON UPDATE TO shoelace_data
    WHERE NEW.sl_avail != OLD.sl_avail
    DO INSERT INTO shoelace_log VALUES (
                                    NEW.sl_name,
                                    NEW.sl_avail,
                                    getpgusername(),
                                    'now'::text
                                );
```

One interesting detail is the casting of 'now' in the rules INSERT action to type text. Without that, the parser would see at CREATE RULE time, that the target type in `shoelace_log` is a datetime and tries to make a constant from it - with success. So a constant datetime value would be stored in the rule action and all log entries would have the time of the CREATE RULE statement. Not exactly what we want. The casting causes that the parser constructs a datetime('now'::text) from it and this will be evaluated when the rule is executed.

Now Al does

```
al_bundy=> UPDATE shoelace_data SET sl_avail = 6

al_bundy->     WHERE sl_name = 'sl7';
```

and we look at the logtable.

```
al_bundy=> SELECT * FROM shoelace_log;
sl_name   |sl_avail|log_who|log_when
----------+--------+-------+--------------------------------
sl7       |       6|Al     |Tue Oct 20 16:14:45 1998 MET DST
(1 row)
```

That's what we expected. What happened in the background is the following. The parser created the parsetree (this time the parts of the original parsetree are highlighted because the base of operations is the rule action for update rules).

```
UPDATE shoelace_data SET sl_avail = 6
  FROM shoelace_data shoelace_data
 WHERE bpchareq(shoelace_data.sl_name, 'sl7');
```

There is a rule 'log_shoelace' that is ON UPDATE with the rule qualification expression

```
int4ne(NEW.sl_avail, OLD.sl_avail)
```

and one action

```
INSERT INTO shoelace_log SELECT
       *NEW*.sl_name, *NEW*.sl_avail,
       getpgusername(), datetime('now'::text)
  FROM shoelace_data *NEW*, shoelace_data *OLD*,
       shoelace_log shoelace_log;
```

Don't trust the output of the pg_rules system view. It specially handles the situation that there are only references to NEW and OLD in the INSERT and outputs the VALUES format of INSERT. In fact there is no difference between an INSERT ... VALUES and an INSERT ... SELECT on parsetree level. They both have rangetables, targetlists and maybe qualifications etc. The optimizer later decides, if to create an execution plan of type result, seqscan, indexscan, join or whatever for that parsetree. If there are no references to rangetable entries leftin the parsetree , it becomes a result execution plan (the INSERT ... VALUES version). The rule action above can truely result in both variants.

The rule is a qualified non-INSTEAD rule, so the rule system has to return two parsetrees. The modified rule action and the original parsetree. In the first step the rangetable of the original query is incorporated into the rules action parsetree. This results in

```
INSERT INTO shoelace_log SELECT
       *NEW*.sl_name, *NEW*.sl_avai,
       getpgusername(), datetime('now'::text)
  FROM shoelace_data shoelace_data, shoelace_data *NEW*,
       shoelace_data *OLD*, shoelace_log shoelace_log;
```

In step 2 the rule qualification is added to it, so the result set is restricted to rows where sl_avail changes.

```
INSERT INTO shoelace_log SELECT
       *NEW*.sl_name, *NEW*.sl_avai,
       getpgusername(), datetime('now'::text)
  FROM shoelace_data shoelace_data, shoelace_data *NEW*,
       shoelace_data *OLD*, shoelace_log shoelace_log
 WHERE int4ne(*NEW*.sl_avail, *OLD*.sl_avail);
```

In step 3 the original parsetrees qualification is added, restricting the resultset further to only the rows touched by the original parsetree.

```
INSERT INTO shoelace_log SELECT
       *NEW*.sl_name, *NEW*.sl_avai,
       getpgusername(), datetime('now'::text)
  FROM shoelace_data shoelace_data, shoelace_data *NEW*,
       shoelace_data *OLD*, shoelace_log shoelace_log
 WHERE int4ne(*NEW*.sl_avail, *OLD*.sl_avail)
   AND bpchareq(shoelace_data.sl_name, 'sl7');
```

Step 4 substitutes NEW references by the targetlist entries from the original parsetree or with the matching variable references from the result relation.

```
INSERT INTO shoelace_log SELECT
       shoelace_data.sl_name, 6,
       getpgusername(), datetime('now'::text)
  FROM shoelace_data shoelace_data, shoelace_data *NEW*,
       shoelace_data *OLD*, shoelace_log shoelace_log
 WHERE int4ne(6, *OLD*.sl_avail)
   AND bpchareq(shoelace_data.sl_name, 'sl7');
```

Step 5 replaces OLD references into resultrelation references.

```
INSERT INTO shoelace_log SELECT
       shoelace_data.sl_name, 6,
       getpgusername(), datetime('now'::text)
  FROM shoelace_data shoelace_data, shoelace_data *NEW*,
       shoelace_data *OLD*, shoelace_log shoelace_log
 WHERE int4ne(6, shoelace_data.sl_avail)
   AND bpchareq(shoelace_data.sl_name, 'sl7');
```

That's it. So reduced to the max the return from the rule system is a list of two parsetrees that are the same as the statements:

```
INSERT INTO shoelace_log SELECT
       shoelace_data.sl_name, 6,
       getpgusername(), 'now'
  FROM shoelace_data
 WHERE 6 != shoelace_data.sl_avail
   AND shoelace_data.sl_name = 'sl7';

UPDATE shoelace_data SET sl_avail = 6
 WHERE sl_name = 'sl7';
```

These are executed in this order and that is exactly what the rule defines. The subtitutions and the qualifications added ensure, that if the original query would be an

```
UPDATE shoelace_data SET sl_color = 'green'
 WHERE sl_name = 'sl7';
```

No log entry would get written because due to the fact that this time the original parsetree does not contain a targetlist entry for sl_avail, NEW.sl_avail will get replaced by shoelace_data.sl_avail resulting in the extra query

```
INSERT INTO shoelace_log SELECT
       shoelace_data.sl_name, shoelace_data.sl_avail,
       getpgusername(), 'now'
  FROM shoelace_data
 WHERE shoelace_data.sl_avail != shoelace_data.sl_avail
```

```
    AND shoelace_data.sl_name = 'sl7';
```

and that qualification will never be true. Since the is no difference on parsetree level between an INSERT ... SELECT, and an INSERT ... VALUES, it will also work if the original query modifies multiple rows. So if Al would issue the command

```
UPDATE shoelace_data SET sl_avail = 0
 WHERE sl_color = 'black';
```

four rows in fact get updated (sl1, sl2, sl3 and sl4). But sl3 already has sl_avail = 0. This time, the original parsetrees qualification is different and that results in the extra parsetree

```
INSERT INTO shoelace_log SELECT
       shoelace_data.sl_name, 0,
       getpgusername(), 'now'
  FROM shoelace_data
 WHERE 0 != shoelace_data.sl_avail
   AND shoelace_data.sl_color = 'black';
```

This parsetree will surely insert three new log entries. And that's absolutely correct.

It is important, that the original parsetree is executed last. The Postgres "traffic cop" does a command counter increment between the execution of the two parsetrees so the second one can see changes made by the first. If the UPDATE would have been executed first, all the rows are already set to zero, so the logging INSERT would not find any row where 0 != shoelace_data.sl_avail.

43.3.3. Cooperation with Views

A simple way to protect view relations from the mentioned possibility that someone can INSERT, UPDATE and DELETE invisible data on them is to let those parsetrees get thrown away. We create the rules

```
CREATE RULE shoe_ins_protect AS ON INSERT TO shoe
    DO INSTEAD NOTHING;
CREATE RULE shoe_upd_protect AS ON UPDATE TO shoe
    DO INSTEAD NOTHING;
CREATE RULE shoe_del_protect AS ON DELETE TO shoe
    DO INSTEAD NOTHING;
```

If Al now tries to do any of these operations on the view relation `shoe`, the rule system will apply the rules. Since the rules have no actions and are INSTEAD, the resulting list of parsetrees will be empty and the whole query will become nothing because there is nothing left to be optimized or executed after the rule system is done with it.

Note

This fact might irritate frontend applications because absolutely nothing happened on the database and thus, the backend will not return anything for the query. Not even a PGRES_EMPTY_QUERY or so will be available in libpq. In psql, nothing happens. This might change in the future.

A more sophisticated way to use the rule system is to create rules that rewrite the parsetree into one that does the right operation on the real tables. To do that on the `shoelace` view, we create the following rules:

```
CREATE RULE shoelace_ins AS ON INSERT TO shoelace
    DO INSTEAD
    INSERT INTO shoelace_data VALUES (
           NEW.sl_name,
```

```
            NEW.sl_avail,
            NEW.sl_color,
            NEW.sl_len,
            NEW.sl_unit);

CREATE RULE shoelace_upd AS ON UPDATE TO shoelace
    DO INSTEAD
    UPDATE shoelace_data SET
           sl_name = NEW.sl_name,
           sl_avail = NEW.sl_avail,
           sl_color = NEW.sl_color,
           sl_len = NEW.sl_len,
           sl_unit = NEW.sl_unit
     WHERE sl_name = OLD.sl_name;

CREATE RULE shoelace_del AS ON DELETE TO shoelace
    DO INSTEAD
    DELETE FROM shoelace_data
     WHERE sl_name = OLD.sl_name;
```

Now there is a pack of shoelaces arriving in Al's shop and it has a big partlist. Al is not that good in calculating and so we don't want him to manually update the shoelace view. Instead we setup two little tables, one where he can insert the items from the partlist and one with a special trick. The create commands for anything are:

```
CREATE TABLE shoelace_arrive (
    arr_name    char(10),
    arr_quant   integer
);

CREATE TABLE shoelace_ok (
    ok_name     char(10),
    ok_quant    integer
);

CREATE RULE shoelace_ok_ins AS ON INSERT TO shoelace_ok
    DO INSTEAD
    UPDATE shoelace SET
           sl_avail = sl_avail + NEW.ok_quant
     WHERE sl_name = NEW.ok_name;
```

Now Al can sit down and do whatever until

```
al_bundy=> SELECT * FROM shoelace_arrive;
arr_name  |arr_quant
----------+---------
sl3       |       10
sl6       |       20
sl8       |       20
(3 rows)
```

is exactly that what's on the part list. We take a quick look at the current data,

```
al_bundy=> SELECT * FROM shoelace ORDER BY sl_name;
sl_name   |sl_avail|sl_color  |sl_len|sl_unit |sl_len_cm
----------+--------+----------+------+--------+---------
sl1       |       5|black     |    80|cm      |       80
sl2       |       6|black     |   100|cm      |      100
sl7       |       6|brown     |    60|cm      |       60
```

```
    sl3       |       0|black     |    35|inch    |     88.9
    sl4       |       8|black     |    40|inch    |    101.6
    sl8       |       1|brown     |    40|inch    |    101.6
    sl5       |       4|brown     |     1|m       |      100
    sl6       |       0|brown     |   0.9|m       |       90
    (8 rows)
```

move the arrived shoelaces in

```
    al_bundy=> INSERT INTO shoelace_ok SELECT * FROM
 shoelace_arrive;
```

and check the results

```
    al_bundy=> SELECT * FROM shoelace ORDER BY sl_name;
    sl_name   |sl_avail|sl_color  |sl_len|sl_unit |sl_len_cm
    ----------+--------+----------+------+--------+---------
    sl1       |       5|black     |    80|cm      |       80
    sl2       |       6|black     |   100|cm      |      100
    sl7       |       6|brown     |    60|cm      |       60
    sl4       |       8|black     |    40|inch    |    101.6
    sl3       |      10|black     |    35|inch    |     88.9
    sl8       |      21|brown     |    40|inch    |    101.6
    sl5       |       4|brown     |     1|m       |      100
    sl6       |      20|brown     |   0.9|m       |       90
    (8 rows)

    al_bundy=> SELECT * FROM shoelace_log;
    sl_name   |sl_avail|log_who|log_when
    ----------+--------+-------+--------------------------------
    sl7       |       6|Al     |Tue Oct 20 19:14:45 1998 MET DST
    sl3       |      10|Al     |Tue Oct 20 19:25:16 1998 MET DST
    sl6       |      20|Al     |Tue Oct 20 19:25:16 1998 MET DST
    sl8       |      21|Al     |Tue Oct 20 19:25:16 1998 MET DST
    (4 rows)
```

It's a long way from the one INSERT ... SELECT to these results. And it's description will be the last in this document (but not the last example :-). First there was the parsers output

```
    INSERT INTO shoelace_ok SELECT
           shoelace_arrive.arr_name, shoelace_arrive.arr_quant
      FROM shoelace_arrive shoelace_arrive, shoelace_ok
 shoelace_ok;
```

Now the first rule 'shoelace_ok_ins' is applied and turns it into

```
    UPDATE shoelace SET
           sl_avail = int4pl(shoelace.sl_avail,
 shoelace_arrive.arr_quant)
      FROM shoelace_arrive shoelace_arrive, shoelace_ok
 shoelace_ok,
           shoelace_ok *OLD*, shoelace_ok *NEW*,
           shoelace shoelace
     WHERE bpchareq(shoelace.sl_name, showlace_arrive.arr_name);
```

and throws away the original INSERT on `shoelace_ok`. This rewritten query is passed to the rule system again and the second applied rule 'shoelace_upd' produced

```
    UPDATE shoelace_data SET
```

```
          sl_name = shoelace.sl_name,
          sl_avail = int4pl(shoelace.sl_avail,
shoelace_arrive.arr_quant),
          sl_color = shoelace.sl_color,
          sl_len = shoelace.sl_len,
          sl_unit = shoelace.sl_unit
     FROM shoelace_arrive shoelace_arrive, shoelace_ok
shoelace_ok,
          shoelace_ok *OLD*, shoelace_ok *NEW*,
          shoelace shoelace, shoelace *OLD*,
          shoelace *NEW*, shoelace_data showlace_data
    WHERE bpchareq(shoelace.sl_name, showlace_arrive.arr_name)
      AND bpchareq(shoelace_data.sl_name, shoelace.sl_name);
```

Again it's an INSTEAD rule and the previous parsetree is trashed. Note that this query sill uses the view `shoelace` But the rule system isn't finished with this loop so it continues and applies the rule '_RETshoelace' on it and we get

```
   UPDATE shoelace_data SET
          sl_name = s.sl_name,
          sl_avail = int4pl(s.sl_avail,
shoelace_arrive.arr_quant),
          sl_color = s.sl_color,
          sl_len = s.sl_len,
          sl_unit = s.sl_unit
     FROM shoelace_arrive shoelace_arrive, shoelace_ok
shoelace_ok,
          shoelace_ok *OLD*, shoelace_ok *NEW*,
          shoelace shoelace, shoelace *OLD*,
          shoelace *NEW*, shoelace_data showlace_data,
          shoelace *OLD*, shoelace *NEW*,
          shoelace_data s, unit u
    WHERE bpchareq(s.sl_name, showlace_arrive.arr_name)
      AND bpchareq(shoelace_data.sl_name, s.sl_name);
```

Again an update rule has been applied and so the wheel turns on and we are in rewrite round 3. This time rule 'log_shoelace' gets applied what produces the extra parsetree

```
   INSERT INTO shoelace_log SELECT
          s.sl_name,
          int4pl(s.sl_avail, shoelace_arrive.arr_quant),
          getpgusername(),
          datetime('now'::text)
     FROM shoelace_arrive shoelace_arrive, shoelace_ok
shoelace_ok,
          shoelace_ok *OLD*, shoelace_ok *NEW*,
          shoelace shoelace, shoelace *OLD*,
          shoelace *NEW*, shoelace_data showlace_data,
          shoelace *OLD*, shoelace *NEW*,
          shoelace_data s, unit u,
          shoelace_data *OLD*, shoelace_data *NEW*
          shoelace_log shoelace_log
    WHERE bpchareq(s.sl_name,  showlace_arrive.arr_name)
      AND bpchareq(shoelace_data.sl_name, s.sl_name);
      AND int4ne(int4pl(s.sl_avail, shoelace_arrive.arr_quant),
                                                 s.sl_avail);
```

After that the rule system runs out of rules and returns the generated parsetrees. So we end up with two final parsetrees that are equal to the SQL statements

```
    INSERT INTO shoelace_log SELECT
           s.sl_name,
           s.sl_avail + shoelace_arrive.arr_quant,
           getpgusername(),
           'now'
      FROM shoelace_arrive shoelace_arrive, shoelace_data
 shoelace_data,
           shoelace_data s
     WHERE s.sl_name = shoelace_arrive.arr_name
       AND shoelace_data.sl_name = s.sl_name
       AND s.sl_avail + shoelace_arrive.arr_quant != s.sl_avail;

    UPDATE shoelace_data SET
           sl_avail = shoelace_data.sl_avail +
 shoelace_arrive.arr_quant
     FROM shoelace_arrive shoelace_arrive,
          shoelace_data shoelace_data,
          shoelace_data s
    WHERE s.sl_name = shoelace_arrive.sl_name
      AND shoelace_data.sl_name = s.sl_name;
```

The result is that data coming from one relation inserted into another, changed into updates on a third, changed into updating a fourth plus logging that final update in a fifth gets reduced into two queries.

There is a little detail that's a bit ugly. Looking at the two queries turns out, that the `shoelace_data` relation appears twice in the rangetable where it could definitely be reduced to one. The optimizer does not handle it and so the execution plan for the rule systems output of the INSERT will be

```
Nested Loop
  ->  Merge Join
        ->  Seq Scan
              ->  Sort
                    ->  Seq Scan on s
        ->  Seq Scan
              ->  Sort
                    ->  Seq Scan on shoelace_arrive
  ->  Seq Scan on shoelace_data
```

while omitting the extra rangetable entry would result in a

```
Merge Join
  ->  Seq Scan
        ->  Sort
              ->  Seq Scan on s
  ->  Seq Scan
        ->  Sort
              ->  Seq Scan on shoelace_arrive
```

that totally produces the same entries in the log relation. Thus, the rule system caused one extra scan on the `shoelace_data` relation that is absolutely not necessary. And the same obsolete scan is done once more in the UPDATE. But it was a really hard job to make that all possible at all.

A final demonstration of the Postgres rule system and it's power. There is a cute blonde that sells shoelaces. And what Al could never realize, she's not only cute, she's smart too - a little too smart. Thus, it happens from time to time that Al orders shoelaces that are absolutely not sellable. This time he ordered 1000 pairs of magenta shoelaces and since another kind is currently not available but he committed to buy some, he also prepared his database for pink ones.

```
    al_bundy=> INSERT INTO shoelace VALUES
```

```
al_bundy->     ('sl9', 0, 'pink', 35.0, 'inch', 0.0);
al_bundy=> INSERT INTO shoelace VALUES
al_bundy->     ('sl10', 1000, 'magenta', 40.0, 'inch', 0.0);
```

Since this happens often, we must lookup for shoelace entries, that fit for absolutely no shoe sometimes. We could do that in a complicated statement every time, or we can setup a view for it. The view for this is

```
CREATE VIEW shoelace_obsolete AS
    SELECT * FROM shoelace WHERE NOT EXISTS
        (SELECT shoename FROM shoe WHERE slcolor = sl_color);
```

It's output is

```
al_bundy=> SELECT * FROM shoelace_obsolete;
sl_name   |sl_avail|sl_color  |sl_len|sl_unit |sl_len_cm
----------+--------+----------+------+--------+---------
sl9       |       0|pink      |    35|inch    |     88.9
sl10      |    1000|magenta   |    40|inch    |    101.6
```

For the 1000 magenta shoelaces we must debt Al before we can throw 'em away, but that's another problem. The pink entry we delete. To make it a little harder for Postgres, we don't delete it directly. Instead we create one more view

```
CREATE VIEW shoelace_candelete AS
    SELECT * FROM shoelace_obsolete WHERE sl_avail = 0;
```

and do it this way:

```
DELETE FROM shoelace WHERE EXISTS
    (SELECT * FROM shoelace_candelete
             WHERE sl_name = shoelace.sl_name);
```

Voila:

```
al_bundy=> SELECT * FROM shoelace;
sl_name   |sl_avail|sl_color  |sl_len|sl_unit |sl_len_cm
----------+--------+----------+------+--------+---------
sl1       |       5|black     |    80|cm      |       80
sl2       |       6|black     |   100|cm      |      100
sl7       |       6|brown     |    60|cm      |       60
sl4       |       8|black     |    40|inch    |    101.6
sl3       |      10|black     |    35|inch    |     88.9
sl8       |      21|brown     |    40|inch    |    101.6
sl10      |    1000|magenta   |    40|inch    |    101.6
sl5       |       4|brown     |     1|m       |      100
sl6       |      20|brown     |   0.9|m       |       90
(9 rows)
```

A DELETE on a view, with a subselect qualification that in total uses 4 nesting/joined views, where one of them itself has a subselect qualification containing a view and where calculated view columns are used, gets rewritten into one single parsetree that deletes the requested data from a real table.

I think there are only a few situations out in the real world, where such a construct is necessary. But it makes me feel comfortable that it works.

The truth is

Doing this I found one more bug while writing this document. But after fixing that I was a little amazed that it works at all.

43.4. Rules and Permissions

Due to rewriting of queries by the Postgres rule system, other tables/views than those used in the original query get accessed. Using update rules, this can include write access to tables.

Rewrite rules don't have a separate owner. The owner of a relation (table or view) is automatically the owner of the rewrite rules that are defined for it. The Postgres rule system changes the behaviour of the default access control system. Relations that are used due to rules get checked during the rewrite against the permissions of the relation owner, the rule is defined on. This means, that a user does only need the required permissions for the tables/views he names in his queries.

For example: A user has a list of phone numbers where some of them are private, the others are of interest for the secretary of the office. He can construct the following:

```
    CREATE TABLE phone_data (person text, phone text, private
 bool);
    CREATE VIEW phone_number AS
        SELECT person, phone FROM phone_data WHERE NOT private;
    GRANT SELECT ON phone_number TO secretary;
```

Nobody except him (and the database superusers) can access the phone_data table. But due to the GRANT, the secretary can SELECT from the phone_number view. The rule system will rewrite the SELECT from phone_number into a SELECT from phone_data and add the qualification that only entries where private is false are wanted. Since the user is the owner of phone_number, the read access to phone_data is now checked against his permissions and the query is considered granted. The check for accessing phone_number is still performed, so nobody than the secretary can use it.

The permissions are checked rule by rule. So the secretary is for now the only one who can see the public phone numbers. But the secretary can setup another view and grant access to that to public. Then, anyone can see the phone_number data through the secretaries view. What the secretary cannot do is to create a view that directly accesses phone_data (actually he can, but it will not work since every access aborts the transaction during the permission checks). And as soon as the user will notice, that the secretary opened his phone_number view, he can REVOKE his access. Immediately any access to the secretaries view will fail.

Someone might think that this rule by rule checking is a security hole, but in fact it isn't. If this would not work, the secretary could setup a table with the same columns as phone_number and copy the data to there once per day. Then it's his own data and he can grant access to everyone he wants. A GRANT means "I trust you". If someone you trust does the thing above, it's time to think it over and then REVOKE.

This mechanism does also work for update rules. In the examples of the previous section, the owner of the tables in Al's database could GRANT SELECT, INSERT, UPDATE and DELETE on the shoelace view to al. But only SELECT on shoelace_log. The rule action to write log entries will still be executed successfull. And Al could see the log entries. But he cannot create fake entries, nor could he manipulate or remove existing ones.

Warning

GRANT ALL currently includes RULE permission. This means the granted user could drop the rule, do the changes and reinstall it. I think this should get changed quickly.

43.5. Rules versus Triggers

Many things that can be done using triggers can also be implemented using the Postgres rule system. What currently cannot be implemented by rules are some kinds of constraints. It is possible, to place a qualified rule that rewrites a query to NOTHING if the value of a column does not appear in another

table. But then the data is silently thrown away and that's not a good idea. If checks for valid values are required, and in the case of an invalid value an error message should be generated, it must be done by a trigger for now.

On the other hand a trigger that is fired on INSERT on a view can do the same as a rule, put the data somewhere else and suppress the insert in the view. But it cannot do the same thing on UPDATE or DELETE, because there is no real data in the view relation that could be scanned and thus the trigger would never get called. Only a rule will help.

For the things that can be implemented by both, it depends on the usage of the database, which is the best. A trigger is fired for any row affected once. A rule manipulates the parsetree or generates an additional one. So if many rows are affected in one statement, a rule issuing one extra query would usually do a better job than a trigger that is called for any single row and must execute his operations this many times.

For example: There are two tables

```
    CREATE TABLE computer (
        hostname        text     -- indexed
 manufacturer    text     -- indexed
    );

    CREATE TABLE software (
        software        text,    -- indexed
        hostname        text     -- indexed
    );
```

Both tables have many thousands of rows and the index on hostname is unique. The hostname column contains the full qualified domain name of the computer. The rule/trigger should constraint delete rows from software that reference the deleted host. Since the trigger is called for each individual row deleted from computer, it can use the statement

```
    DELETE FROM software WHERE hostname = $1;
```

in a prepared and saved plan and pass the hostname in the parameter. The rule would be written as

```
    CREATE RULE computer_del AS ON DELETE TO computer
        DO DELETE FROM software WHERE hostname = OLD.hostname;
```

Now we look at different types of deletes. In the case of a

```
    DELETE FROM computer WHERE hostname = 'mypc.local.net';
```

the table computer is scanned by index (fast) and the query issued by the trigger would also be an index scan (fast too). The extra query from the rule would be a

```
    DELETE FROM software WHERE computer.hostname = 'mypc.local.net'
                           AND software.hostname =
 computer.hostname;
```

Since there are appropriate indices setup, the optimizer will create a plan of

```
    Nestloop
      ->  Index Scan using comp_hostidx on computer
      ->  Index Scan using soft_hostidx on software
```

So there would be not that much difference in speed between the trigger and the rule implementation. With the next delete we want to get rid of all the 2000 computers where the hostname starts with 'old'. There are two possible queries to do that. One is

```
    DELETE FROM computer WHERE hostname >= 'old'
```

```
                          AND hostname <  'ole'
```

Where the plan for the rule query will be a

```
Hash Join
  ->  Seq Scan on software
  ->  Hash
 ->  Index Scan using comp_hostidx on computer
```

The other possible query is a

```
DELETE FROM computer WHERE hostname ~ '^old';
```

with the execution plan

```
Nestloop
  ->  Index Scan using comp_hostidx on computer
  ->  Index Scan using soft_hostidx on software
```

This shows, that the optimizer does not realize that the qualification for the hostname on computer could also be used for an index scan on software when there are multiple qualification expressions combined with AND, what he does in the regexp version of the query. The trigger will get invoked once for any of the 2000 old computers that have to be deleted and that will result in one index scan over computer and 2000 index scans for the software. The rule implementation will do it with two queries over indices. And it depends on the overall size of the software table if the rule will still be faster in the seqscan situation. 2000 query executions over the SPI manager take some time, even if all the index blocks to look them up will soon appear in the cache.

The last query we look at is a

```
DELETE FROM computer WHERE manufacurer = 'bim';
```

Again this could result in many rows to be deleted from computer. So the trigger will again fire many queries into the executor. But the rule plan will again be the Nestloop over two IndexScan's. Only using another index on computer:

```
Nestloop
  ->  Index Scan using comp_manufidx on computer
  ->  Index Scan using soft_hostidx on software
```

resulting from the rules query

```
DELETE FROM software WHERE computer.manufacurer = 'bim'
                       AND software.hostname =
computer.hostname;
```

In any of these cases, the extra queries from the rule system will be more or less independent from the number of affected rows in a query.

Another situation is cases on UPDATE where it depends on the change of an attribute if an action should be performed or not. In Postgres version 6.4, the attribute specification for rule events is disabled (it will have it's comeback latest in 6.5, maybe earlier - stay tuned). So for now the only way to create a rule as in the shoelace_log example is to do it with a rule qualification. That results in an extra query that is performed always, even if the attribute of interest cannot change at all because it does not appear in the targetlist of the initial query. When this is enabled again, it will be one more advantage of rules over triggers. Optimization of a trigger must fail by definition in this case, because the fact that it's actions will only be done when a specific attribute is updated is hidden in it's functionality. The definition of a trigger only allows to specify it on row level, so whenever a row is touched, the trigger must be called to make it's decision. The rule system will know it by looking up the targetlist and will suppress the additional query completely if the attribute isn't touched. So the rule, qualified or not, will only do its scans if there ever could be something to do.

Rules will only be significant slower than triggers if their actions result in large and bad qualified joins, a situation where the optimizer fails. They are a big hammer. Using a big hammer without caution can cause big damage. But used with the right touch, they can hit any nail on the head.

Chapter 44. Interfacing Extensions To Indices

The procedures described thus far let you define a new type, new functions and new operators. However, we cannot yet define a secondary index (such as a B-tree, R-tree or hash access method) over a new type or its operators.

Look back at

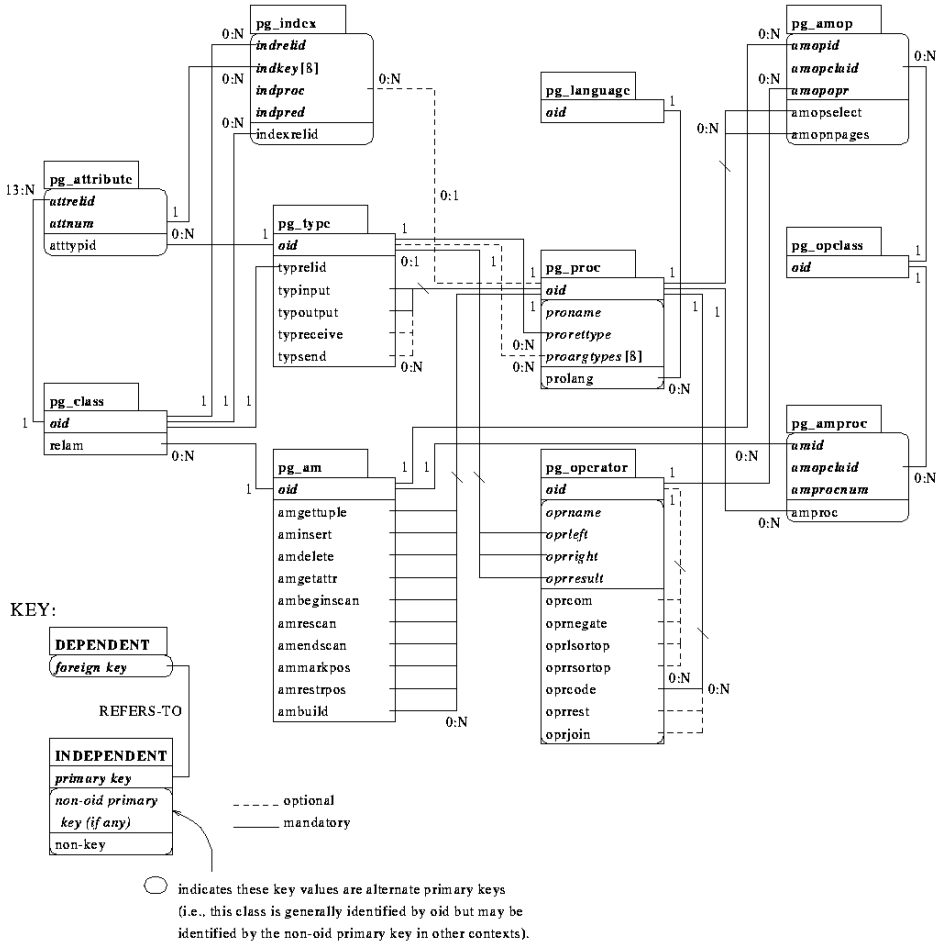


Figure 3. The major POSTGRES system catalogs.

. The right half shows the catalogs that we must modify in order to tell Postgres how to use a user-defined type and/or user-defined operators with an index (i.e., `pg_am`, `pg_amop`, `pg_amproc`, `pg_operator` and `pg_opclass`). Unfortunately, there is no simple command to do this. We will demonstrate how to modify these catalogs through a running example: a new operator class for the B-tree access method that stores and sorts complex numbers in ascending absolute value order.

The `pg_am` class contains one instance for every user defined access method. Support for the heap access method is built into Postgres, but every other access method is described here. The schema is

Table 44.1. Index Schema

Attribute	Description
amname	name of the access method
amowner	object id of the owner's instance in pg_user

Attribute	Description
amstrategies	number of strategies for this access method (see below)
amsupport	number of support routines for this access method (see below)
amorderstrategy	zero if the index offers no sort order, otherwise the strategy number of the strategy operator that describes the sort order
amgettuple	
aminsert	
...	procedure identifiers for interface routines to the access method. For example, regproc ids for opening, closing, and getting instances from the access method appear here.

The object ID of the instance in `pg_am` is used as a foreign key in lots of other classes. You don't need to add a new instance to this class; all you're interested in is the object ID of the access method instance you want to extend:

```
SELECT oid FROM pg_am WHERE amname = 'btree';

 oid
-----
 403
(1 row)
```

We will use that `SELECT` in a `WHERE` clause later.

The `amstrategies` attribute exists to standardize comparisons across data types. For example, B-trees impose a strict ordering on keys, lesser to greater. Since Postgres allows the user to define operators, Postgres cannot look at the name of an operator (eg, ">" or "<") and tell what kind of comparison it is. In fact, some access methods don't impose any ordering at all. For example, R-trees express a rectangle-containment relationship, whereas a hashed data structure expresses only bitwise similarity based on the value of a hash function. Postgres needs some consistent way of taking a qualification in your query, looking at the operator and then deciding if a usable index exists. This implies that Postgres needs to know, for example, that the "<=" and ">" operators partition a B-tree. Postgres uses strategies to express these relationships between operators and the way they can be used to scan indices.

Defining a new set of strategies is beyond the scope of this discussion, but we'll explain how B-tree strategies work because you'll need to know that to add a new operator class. In the `pg_am` class, the amstrategies attribute is the number of strategies defined for this access method. For B-trees, this number is 5. These strategies correspond to

Table 44.2. B-tree Strategies

Operation	Index
less than	1
less than or equal	2
equal	3
greater than or equal	4
greater than	5

The idea is that you'll need to add procedures corresponding to the comparisons above to the pg_amop relation (see below). The access method code can use these strategy numbers, regardless of data type, to figure out how to partition the B-tree, compute selectivity, and so on. Don't worry about the details of adding procedures yet; just understand that there must be a set of these procedures for int2, int4, oid, and every other data type on which a B-tree can operate.

Sometimes, strategies aren't enough information for the system to figure out how to use an index. Some access methods require other support routines in order to work. For example, the B-tree access method must be able to compare two keys and determine whether one is greater than, equal to, or less than the other. Similarly, the R-tree access method must be able to compute intersections, unions, and sizes of rectangles. These operations do not correspond to user qualifications in SQL queries; they are administrative routines used by the access methods, internally.

In order to manage diverse support routines consistently across all Postgres access methods, pg_am includes an attribute called amsupport. This attribute records the number of support routines used by an access method. For B-trees, this number is one -- the routine to take two keys and return -1, 0, or +1, depending on whether the first key is less than, equal to, or greater than the second.

Note

Strictly speaking, this routine can return a negative number (< 0), 0, or a non-zero positive number (> 0).

The amstrategies entry in pg_am is just the number of strategies defined for the access method in question. The procedures for less than, less equal, and so on don't appear in pg_am. Similarly, amsupport is just the number of support routines required by the access method. The actual routines are listed elsewhere.

By the way, the amorderstrategy entry tells whether the access method supports ordered scan. Zero means it doesn't; if it does, amorderstrategy is the number of the strategy routine that corresponds to the ordering operator. For example, btree has amorderstrategy = 1 which is its "less than" strategy number.

The next class of interest is pg_opclass. This class exists only to associate an operator class name and perhaps a default type with an operator class oid. Some existing opclasses are int2_ops, int4_ops, and oid_ops. You need to add an instance with your opclass name (for example, complex_abs_ops) to pg_opclass. The oid of this instance will be a foreign key in other classes, notably pg_amop.

```
INSERT INTO pg_opclass (opcname, opcdeftype)
    SELECT 'complex_abs_ops', oid FROM pg_type WHERE typname =
 'complex';

SELECT oid, opcname, opcdeftype
    FROM pg_opclass
    WHERE opcname = 'complex_abs_ops';

  oid   |     opcname     | opcdeftype
--------+-----------------+------------
 277975 | complex_abs_ops |     277946
(1 row)
```

Note that the oid for your pg_opclass instance will be different! Don't worry about this though. We'll get this number from the system later just like we got the oid of the type here.

The above example assumes that you want to make this new opclass the default index opclass for the complex datatype. If you don't, just insert zero into opcdeftype, rather than inserting the datatype's oid:

```
INSERT INTO pg_opclass (opcname, opcdeftype) VALUES
 ('complex_abs_ops', 0);
```

So now we have an access method and an operator class. We still need a set of operators. The procedure for defining operators was discussed earlier in this manual. For the `complex_abs_ops` operator class on Btrees, the operators we require are:

```
absolute value less-than
absolute value less-than-or-equal
absolute value equal
absolute value greater-than-or-equal
absolute value greater-than
```

Suppose the code that implements the functions defined is stored in the file `PGROOT/src/tutorial/complex.c`

Part of the C code looks like this: (note that we will only show the equality operator for the rest of the examples. The other four operators are very similar. Refer to `complex.c` or `complex.source` for the details.)

```
#define Mag(c) ((c)->x*(c)->x + (c)->y*(c)->y)

bool
complex_abs_eq(Complex *a, Complex *b)
{
    double amag = Mag(a), bmag = Mag(b);
    return (amag==bmag);
}
```

We make the function known to Postgres like this:

```
CREATE FUNCTION complex_abs_eq(complex, complex)
        RETURNS bool
        AS 'PGROOT/tutorial/obj/complex.so'
        LANGUAGE 'c';
```

There are some important things that are happening here.

First, note that operators for less-than, less-than-or-equal, equal, greater-than-or-equal, and greater-than for `complex` are being defined. We can only have one operator named, say, = and taking type `complex` for both operands. In this case we don't have any other operator = for `complex`, but if we were building a practical datatype we'd probably want = to be the ordinary equality operation for complex numbers. In that case, we'd need to use some other operator name for complex_abs_eq.

Second, although Postgres can cope with operators having the same name as long as they have different input datatypes, C can only cope with one global routine having a given name, period. So we shouldn't name the C function something simple like `abs_eq`. Usually it's a good practice to include the datatype name in the C function name, so as not to conflict with functions for other datatypes.

Third, we could have made the Postgres name of the function `abs_eq`, relying on Postgres to distinguish it by input datatypes from any other Postgres function of the same name. To keep the example simple, we make the function have the same names at the C level and Postgres level.

Finally, note that these operator functions return Boolean values. The access methods rely on this fact. (On the other hand, the support function returns whatever the particular access method expects -- in this case, a signed integer.) The final routine in the file is the "support routine" mentioned when we discussed the amsupport attribute of the `pg_am` class. We will use this later on. For now, ignore it.

Now we are ready to define the operators:

```
CREATE OPERATOR = (
     leftarg = complex, rightarg = complex,
     procedure = complex_abs_eq,
     restrict = eqsel, join = eqjoinsel
         )
```

The important things here are the procedure names (which are the C functions defined above) and the restriction and join selectivity functions. You should just use the selectivity functions used in the example (see `complex.source`). Note that there are different such functions for the less-than, equal, and greater-than cases. These must be supplied, or the optimizer will be unable to make effective use of the index.

The next step is to add entries for these operators to the `pg_amop` relation. To do this, we'll need the `oid`s of the operators we just defined. We'll look up the names of all the operators that take two `complex`es, and pick ours out:

```
    SELECT o.oid AS opoid, o.oprname
     INTO TABLE complex_ops_tmp
     FROM pg_operator o, pg_type t
     WHERE o.oprleft = t.oid and o.oprright = t.oid
      and t.typname = 'complex';

 opoid  | oprname
--------+---------
 277963 | +
 277970 | <
 277971 | <=
 277972 | =
 277973 | >=
 277974 | >
(6 rows)
```

(Again, some of your `oid` numbers will almost certainly be different.) The operators we are interested in are those with `oid`s 277970 through 277974. The values you get will probably be different, and you should substitute them for the values below. We will do this with a select statement.

Now we're ready to update `pg_amop` with our new operator class. The most important thing in this entire discussion is that the operators are ordered, from less than through greater than, in `pg_amop`. We add the instances we need:

```
    INSERT INTO pg_amop (amopid, amopclaid, amopopr, amopstrategy)
        SELECT am.oid, opcl.oid, c.opoid, 1
        FROM pg_am am, pg_opclass opcl, complex_ops_tmp c
        WHERE amname = 'btree' AND
            opcname = 'complex_abs_ops' AND
            c.oprname = '<';
```

Now do this for the other operators substituting for the "1" in the third line above and the "<" in the last line. Note the order: "less than" is 1, "less than or equal" is 2, "equal" is 3, "greater than or equal" is 4, and "greater than" is 5.

The next step is registration of the "support routine" previously described in our discussion of `pg_am`. The `oid` of this support routine is stored in the `pg_amproc` class, keyed by the access method `oid` and the operator class `oid`. First, we need to register the function in Postgres (recall that we put the

C code that implements this routine in the bottom of the file in which we implemented the operator routines):

```
    CREATE FUNCTION complex_abs_cmp(complex, complex)
     RETURNS int4
     AS 'PGROOT/tutorial/obj/complex.so'
     LANGUAGE 'c';

    SELECT oid, proname FROM pg_proc
     WHERE proname = 'complex_abs_cmp';

  oid   |     proname
--------+-----------------
 277997 | complex_abs_cmp
(1 row)
```

(Again, your `oid` number will probably be different.) We can add the new instance as follows:

```
    INSERT INTO pg_amproc (amid, amopclaid, amproc, amprocnum)
        SELECT a.oid, b.oid, c.oid, 1
            FROM pg_am a, pg_opclass b, pg_proc c
            WHERE a.amname = 'btree' AND
                b.opcname = 'complex_abs_ops' AND
                c.proname = 'complex_abs_cmp';
```

And we're done! (Whew.) It should now be possible to create and use btree indexes on `complex` columns.

Chapter 45. Index Cost Estimation Functions

Author

Written by Tom Lane[1] on 2000-01-24.

Note

This must eventually become part of a much larger chapter about writing new index access methods.

Every index access method must provide a cost estimation function for use by the planner/optimizer. The procedure OID of this function is given in the `amcostestimate` field of the access method's `pg_am` entry.

Note

Prior to Postgres 7.0, a different scheme was used for registering index-specific cost estimation functions.

The amcostestimate function is given a list of WHERE clauses that have been determined to be usable with the index. It must return estimates of the cost of accessing the index and the selectivity of the WHERE clauses (that is, the fraction of main-table tuples that will be retrieved during the index scan). For simple cases, nearly all the work of the cost estimator can be done by calling standard routines in the optimizer; the point of having an amcostestimate function is to allow index access methods to provide index-type-specific knowledge, in case it is possible to improve on the standard estimates.

Each amcostestimate function must have the signature:

```
void
amcostestimate (Query *root,
                RelOptInfo *rel,
                IndexOptInfo *index,
                List *indexQuals,
                Cost *indexStartupCost,
                Cost *indexTotalCost,
                Selectivity *indexSelectivity);
```

The first four parameters are inputs:

root

The query being processed.

rel

The relation the index is on.

index

The index itself.

indexQuals

List of index qual clauses (implicitly ANDed); a NIL list indicates no qualifiers are available.

[1] mailto:tgl@sss.pgh.pa.us

The last three parameters are pass-by-reference outputs:

*indexStartupCost

Set to cost of index startup processing

*indexTotalCost

Set to total cost of index processing

*indexSelectivity

Set to index selectivity

Note that cost estimate functions must be written in C, not in SQL or any available procedural language, because they must access internal data structures of the planner/optimizer.

The index access costs should be computed in the units used by src/backend/optimizer/path/costsize.c: a sequential disk block fetch has cost 1.0, a nonsequential fetch has cost random_page_cost, and the cost of processing one index tuple should usually be taken as cpu_index_tuple_cost (which is a user-adjustable optimizer parameter). In addition, an appropriate multiple of cpu_operator_cost should be charged for any comparison operators invoked during index processing (especially evaluation of the indexQuals themselves).

The access costs should include all disk and CPU costs associated with scanning the index itself, but NOT the costs of retrieving or processing the main-table tuples that are identified by the index.

The "startup cost" is the part of the total scan cost that must be expended before we can begin to fetch the first tuple. For most indexes this can be taken as zero, but an index type with a high startup cost might want to set it nonzero.

The indexSelectivity should be set to the estimated fraction of the main table tuples that will be retrieved during the index scan. In the case of a lossy index, this will typically be higher than the fraction of tuples that actually pass the given qual conditions.

Cost Estimation

A typical cost estimator will proceed as follows:

1. Estimate and return the fraction of main-table tuples that will be visited based on the given qual conditions. In the absence of any index-type-specific knowledge, use the standard optimizer function clauselist_selectivity():

```
*indexSelectivity = clauselist_selectivity(root, indexQuals,
                                           lfirsti(rel-
>relids));
```

2. Estimate the number of index tuples that will be visited during the scan. For many index types this is the same as indexSelectivity times the number of tuples in the index, but it might be more. (Note that the index's size in pages and tuples is available from the IndexOptInfo struct.)

3. Estimate the number of index pages that will be retrieved during the scan. This might be just indexSelectivity times the index's size in pages.

4. Compute the index access cost. A generic estimator might do this:

```
    /*
     * Our generic assumption is that the index pages will be
 read
```

```
     * sequentially, so they have cost 1.0 each, not
random_page_cost.
     * Also, we charge for evaluation of the indexquals at each
index tuple.
     * All the costs are assumed to be paid incrementally
during the scan.
     */
    *indexStartupCost = 0;
    *indexTotalCost = numIndexPages +
        (cpu_index_tuple_cost + cost_qual_eval(indexQuals)) *
numIndexTuples;
```

Examples of cost estimator functions can be found in `src/backend/utils/adt/selfuncs.c`.

By convention, the `pg_proc` entry for an `amcostestimate` function should show

```
prorettype = 0
pronargs = 7
proargtypes = 0 0 0 0 0 0 0
```

We use zero ("opaque") for all the arguments since none of them have types that are known in pg_type.

Chapter 46. GiST Indices

Gene Selkov

The information about GIST is at http://GiST.CS.Berkeley.EDU:8000/gist/ with more on different indexing and sorting schemes at http://s2k-ftp.CS.Berkeley.EDU:8000/personal/jmh/ And there is more interesting reading at the Berkely database site at http://epoch.cs.berkeley.edu:8000/.

Author

This extraction from an e-mail sent by Eugene Selkov Jr.[1] contains good information on GiST. Hopefully we will learn more in the future and update this information. - thomas 1998-03-01

Well, I can't say I quite understand what's going on, but at least I (almost) succeeded in porting GiST examples to linux. The GiST access method is already in the postgres tree (`src/backend/access/gist`).

Examples at Berkeley[2] come with an overview of the methods and demonstrate spatial index mechanisms for 2D boxes, polygons, integer intervals and text (see also GiST at Berkeley[3]). In the box example, we are supposed to see a performance gain when using the GiST index; it did work for me but I do not have a reasonably large collection of boxes to check that. Other examples also worked, except polygons: I got an error doing

```
test=> create index pix on polytmp
test-> using gist (p:box gist_poly_ops) with (islossy);
ERROR:  cannot open pix

(PostgreSQL 6.3              Sun Feb  1 14:57:30 EST 1998)
```

I could not get sense of this error message; it appears to be something we'd rather ask the developers about (see also Note 4 below). What I would suggest here is that someone of you linux guys (linux==gcc?) fetch the original sources quoted above and apply my patch (see attachment) and tell us what you feel about it. Looks cool to me, but I would not like to hold it up while there are so many competent people around.

A few notes on the sources:

1. I failed to make use of the original (HPUX) Makefile and rearranged the Makefile from the ancient postgres95 tutorial to do the job. I tried to keep it generic, but I am a very poor makefile writer -- just did some monkey work. Sorry about that, but I guess it is now a little more portable that the original makefile.

2. I built the example sources right under pgsql/src (just extracted the tar file there). The aforementioned Makefile assumes it is one level below pgsql/src (in our case, in pgsql/src/pggist).

3. The changes I made to the *.c files were all about #include's, function prototypes and typecasting. Other than that, I just threw away a bunch of unused vars and added a couple parentheses to please gcc. I hope I did not screw up too much :)

4. There is a comment in polyproc.sql:

```
-- -- there's a memory leak in rtree poly_ops!!
-- -- create index pix2 on polytmp using rtree (p poly_ops);
```

Roger that!! I thought it could be related to a number of Postgres versions back and tried the query. My system went nuts and I had to shoot down the postmaster in about ten minutes.

[1] mailto:selkovjr@mcs.anl.gov
[2] ftp://s2k-ftp.cs.berkeley.edu/pub/gist/pggist/pggist.tgz
[3] http://gist.cs.berkeley.edu:8000/gist/

I will continue to look into GiST for a while, but I would also appreciate more examples of R-tree usage.

Chapter 47. Linking Dynamically-Loaded Functions

After you have created and registered a user-defined function, your work is essentially done. Postgres, however, must load the object code (e.g., a `.o` file, or a shared library) that implements your function. As previously mentioned, Postgres loads your code at runtime, as required. In order to allow your code to be dynamically loaded, you may have to compile and link-edit it in a special way. This section briefly describes how to perform the compilation and link-editing required before you can load your user-defined functions into a running Postgres server.

You should expect to read (and reread, and re-reread) the manual pages for the C compiler, cc(1), and the link editor, ld(1), if you have specific questions. In addition, the contrib area (`PGROOT/contrib`) and the regression test suites in the directory `PGROOT/src/test/regress` contain several working examples of this process. If you copy an example then you should not have any problems.

The following terminology will be used below:

- *Dynamic loading* is what Postgres does to an object file. The object file is copied into the running Postgres server and the functions and variables within the file are made available to the functions within the Postgres process. Postgres does this using the dynamic loading mechanism provided by the operating system.
- *Loading and link editing* is what you do to an object file in order to produce another kind of object file (e.g., an executable program or a shared library). You perform this using the link editing program, ld(1).

The following general restrictions and notes also apply to the discussion below:

- Paths given to the create function command must be absolute paths (i.e., start with "/") that refer to directories visible on the machine on which the Postgres server is running.

  Tip

  Relative paths do in fact work, but are relative to the directory where the database resides (which is generally invisible to the frontend application). Obviously, it makes no sense to make the path relative to the directory in which the user started the frontend application, since the server could be running on a completely different machine!

- The Postgres user must be able to traverse the path given to the create function command and be able to read the object file. This is because the Postgres server runs as the Postgres user, not as the user who starts up the frontend process. (Making the file or a higher-level directory unreadable and/or unexecutable by the "postgres" user is an extremely common mistake.)
- Symbol names defined within object files must not conflict with each other or with symbols defined in Postgres.
- The GNU C compiler usually does not provide the special options that are required to use the operating system's dynamic loader interface. In such cases, the C compiler that comes with the operating system must be used.

47.1. Linux

Under Linux ELF, object files can be generated by specifying the compiler flag -fpic.

For example,

```
# simple Linux example
% cc -fpic -c foo.c
```

produces an object file called *foo.o* that can then be dynamically loaded into Postgres. No additional loading or link-editing must be performed.

47.2. DEC OSF/1

Under DEC OSF/1, you can take any simple object file and produce a shared object file by running the ld command over it with the correct options. The commands to do this look like:

```
# simple DEC OSF/1 example
% cc -c foo.c
% ld -shared -expect_unresolved '*' -o foo.so foo.o
```

The resulting shared object file can then be loaded into Postgres. When specifying the object file name to the create function command, one must give it the name of the shared object file (ending in .so) rather than the simple object file.

Tip

Actually, Postgres does not care what you name the file as long as it is a shared object file. If you prefer to name your shared object files with the extension .o, this is fine with Postgres so long as you make sure that the correct file name is given to the create function command. In other words, you must simply be consistent. However, from a pragmatic point of view, we discourage this practice because you will undoubtedly confuse yourself with regards to which files have been made into shared object files and which have not. For example, it's very hard to write Makefiles to do the link-editing automatically if both the object file and the shared object file end in .o!

If the file you specify is not a shared object, the backend will hang!

47.3. SunOS 4.x, Solaris 2.x and HP-UX

Under SunOS 4.x, Solaris 2.x and HP-UX, the simple object file must be created by compiling the source file with special compiler flags and a shared library must be produced. The necessary steps with HP-UX are as follows. The +z flag to the HP-UX C compiler produces *Position Independent Code* (PIC) and the +u flag removes some alignment restrictions that the PA-RISC architecture normally enforces. The object file must be turned into a shared library using the HP-UX link editor with the -b option. This sounds complicated but is actually very simple, since the commands to do it are just:

```
# simple HP-UX example
% cc +z +u -c foo.c
% ld -b -o foo.sl foo.o
```

As with the .so files mentioned in the last subsection, the create function command must be told which file is the correct file to load (i.e., you must give it the location of the shared library, or .sl file). Under SunOS 4.x, the commands look like:

```
# simple SunOS 4.x example
% cc -PIC -c foo.c
% ld -dc -dp -Bdynamic -o foo.so foo.o
```

and the equivalent lines under Solaris 2.x are:

```
# simple Solaris 2.x example
% cc -K PIC -c foo.c
% ld -G -Bdynamic -o foo.so foo.o
```

or

```
# simple Solaris 2.x example
% gcc -fPIC -c foo.c
% ld -G -Bdynamic -o foo.so foo.o
```

When linking shared libraries, you may have to specify some additional shared libraries (typically system libraries, such as the C and math libraries) on your ld command line.

Chapter 48. Triggers

Postgres has various server-side function interfaces. Server-side functions can be written in SQL, PLPGSQL, TCL, or C. Trigger functions can be written in any of these languages except SQL. Note that STATEMENT-level trigger events are not supported in the current version. You can currently specify BEFORE or AFTER on INSERT, DELETE or UPDATE of a tuple as a trigger event.

48.1. Trigger Creation

If a trigger event occurs, the trigger manager (called by the Executor) initializes the global structure TriggerData *CurrentTriggerData (described below) and calls the trigger function to handle the event.

The trigger function must be created before the trigger is created as a function taking no arguments and returns opaque.

The syntax for creating triggers is as follows:

```
CREATE TRIGGER trigger [ BEFORE | AFTER ] [ INSERT | DELETE |
 UPDATE [ OR ... ] ]
    ON relation FOR EACH [ ROW | STATEMENT ]
    EXECUTE PROCEDURE procedure
     (args);
```

where the arguments are:

trigger

The name of the trigger is used if you ever have to delete the trigger. It is used as an argument to the `DROP TRIGGER` command.

BEFORE
AFTER

Determines whether the function is called before or after the event.

INSERT
DELETE
UPDATE

The next element of the command determines on what event(s) will trigger the function. Multiple events can be specified separated by OR.

relation

The relation name determines which table the event applies to.

ROW
STATEMENT

The FOR EACH clause determines whether the trigger is fired for each affected row or before (or after) the entire statement has completed.

procedure

The procedure name is the function called.

args

The arguments passed to the function in the CurrentTriggerData structure. The purpose of passing arguments to the function is to allow different triggers with similar requirements to call the same function.

Also, *procedure* may be used for triggering different relations (these functions are named as "general trigger functions").

As example of using both features above, there could be a general function that takes as its arguments two field names and puts the current user in one and the current timestamp in the other. This allows triggers to be written on INSERT events to automatically track creation of records in a transaction table for example. It could also be used as a "last updated" function if used in an UPDATE event.

Trigger functions return HeapTuple to the calling Executor. This is ignored for triggers fired after an INSERT, DELETE or UPDATE operation but it allows BEFORE triggers to:

- Return NULL to skip the operation for the current tuple (and so the tuple will not be inserted/updated/deleted).
- Return a pointer to another tuple (INSERT and UPDATE only) which will be inserted (as the new version of the updated tuple if UPDATE) instead of original tuple.

Note that there is no initialization performed by the CREATE TRIGGER handler. This will be changed in the future. Also, if more than one trigger is defined for the same event on the same relation, the order of trigger firing is unpredictable. This may be changed in the future.

If a trigger function executes SQL-queries (using SPI) then these queries may fire triggers again. This is known as cascading triggers. There is no explicit limitation on the number of cascade levels.

If a trigger is fired by INSERT and inserts a new tuple in the same relation then this trigger will be fired again. Currently, there is nothing provided for synchronization (etc) of these cases but this may change. At the moment, there is function funny_dup17() in the regress tests which uses some techniques to stop recursion (cascading) on itself...

48.2. Interaction with the Trigger Manager

As mentioned above, when function is called by the trigger manager, structure TriggerData *CurrentTriggerData is NOT NULL and initialized. So it is better to check CurrentTriggerData against being NULL at the start and set it to NULL just after fetching the information to prevent calls to a trigger function not from the trigger manager.

struct TriggerData is defined in src/include/commands/trigger.h:

```
typedef struct TriggerData
{
    TriggerEvent  tg_event;
    Relation      tg_relation;
    HeapTuple     tg_trigtuple;
    HeapTuple     tg_newtuple;
    Trigger      *tg_trigger;
} TriggerData;
```

where the members are defined as follows:

tg_event

describes the event for which the function is called. You may use the following macros to examine `tg_event`:

TRIGGER_FIRED_BEFORE(tg_event)

returns TRUE if trigger fired BEFORE.

TRIGGER_FIRED_AFTER(tg_event)

Returns TRUE if trigger fired AFTER.

TRIGGER_FIRED_FOR_ROW(event)

Returns TRUE if trigger fired for a ROW-level event.

TRIGGER_FIRED_FOR_STATEMENT(event)

Returns TRUE if trigger fired for STATEMENT-level event.

TRIGGER_FIRED_BY_INSERT(event)

Returns TRUE if trigger fired by INSERT.

TRIGGER_FIRED_BY_DELETE(event)

Returns TRUE if trigger fired by DELETE.

TRIGGER_FIRED_BY_UPDATE(event)

Returns TRUE if trigger fired by UPDATE.

tg_relation

is a pointer to structure describing the triggered relation. Look at src/include/utils/rel.h for details about this structure. The most interest things are tg_relation->rd_att (descriptor of the relation tuples) and tg_relation->rd_rel->relname (relation's name. This is not char*, but NameData. Use SPI_getrelname(tg_relation) to get char* if you need a copy of name).

tg_trigtuple

is a pointer to the tuple for which the trigger is fired. This is the tuple being inserted (if INSERT), deleted (if DELETE) or updated (if UPDATE). If INSERT/DELETE then this is what you are to return to Executor if you don't want to replace tuple with another one (INSERT) or skip the operation.

tg_newtuple

is a pointer to the new version of tuple if UPDATE and NULL if this is for an INSERT or a DELETE. This is what you are to return to Executor if UPDATE and you don't want to replace this tuple with another one or skip the operation.

tg_trigger

is pointer to structure Trigger defined in src/include/utils/rel.h:

```
typedef struct Trigger
{
    Oid         tgoid;
    char       *tgname;
    Oid         tgfoid;
    FmgrInfo    tgfunc;
    int16       tgtype;
    bool        tgenabled;
    bool        tgisconstraint;
    bool        tgdeferrable;
    bool        tginitdeferred;
    int16       tgnargs;
    int16       tgattr[FUNC_MAX_ARGS];
    char      **tgargs;
} Trigger;
```

where tgname is the trigger's name, tgnargs is number of arguments in tgargs, tgargs is an array of pointers to the arguments specified in the CREATE TRIGGER statement. Other members are for internal use only.

48.3. Visibility of Data Changes

Postgres data changes visibility rule: during a query execution, data changes made by the query itself (via SQL-function, SPI-function, triggers) are invisible to the query scan. For example, in query

```
INSERT INTO a SELECT * FROM a;
```

tuples inserted are invisible for SELECT scan. In effect, this duplicates the database table within itself (subject to unique index rules, of course) without recursing.

But keep in mind this notice about visibility in the SPI documentation:

> Changes made by query Q are visible by queries which are started after query Q, no matter whether they are started inside Q (during the execution of Q) or after Q is done.

This is true for triggers as well so, though a tuple being inserted (tg_trigtuple) is not visible to queries in a BEFORE trigger, this tuple (just inserted) is visible to queries in an AFTER trigger, and to queries in BEFORE/AFTER triggers fired after this!

48.4. Examples

There are more complex examples in `src/test/regress/regress.c` and in `contrib/spi`.

Here is a very simple example of trigger usage. Function trigf reports the number of tuples in the triggered relation ttest and skips the operation if the query attempts to insert NULL into x (i.e - it acts as a NOT NULL constraint but doesn't abort the transaction).

```
#include "executor/spi.h" /* this is what you need to work with SPI
 */
#include "commands/trigger.h" /* -"- and triggers */

HeapTuple  trigf(void);

HeapTuple
trigf()
{
 TupleDesc tupdesc;
 HeapTuple rettuple;
 char  *when;
 bool  checknull = false;
 bool  isnull;
 int  ret, i;

 if (!CurrentTriggerData)
  elog(NOTICE, "trigf: triggers are not initialized");

 /* tuple to return to Executor */
 if (TRIGGER_FIRED_BY_UPDATE(CurrentTriggerData->tg_event))
  rettuple = CurrentTriggerData->tg_newtuple;
 else
  rettuple = CurrentTriggerData->tg_trigtuple;

 /* check for NULLs ? */
 if (!TRIGGER_FIRED_BY_DELETE(CurrentTriggerData->tg_event) &&
  TRIGGER_FIRED_BEFORE(CurrentTriggerData->tg_event))
  checknull = true;
```

```
 if (TRIGGER_FIRED_BEFORE(CurrentTriggerData->tg_event))
  when = "before";
 else
  when = "after ";

 tupdesc = CurrentTriggerData->tg_relation->rd_att;
 CurrentTriggerData = NULL;

 /* Connect to SPI manager */
 if ((ret = SPI_connect()) < 0)
  elog(NOTICE, "trigf (fired %s): SPI_connect returned %d", when,
 ret);

 /* Get number of tuples in relation */
 ret = SPI_exec("select count(*) from ttest", 0);

 if (ret < 0)
  elog(NOTICE, "trigf (fired %s): SPI_exec returned %d", when,
 ret);

 i = SPI_getbinval(SPI_tuptable->vals[0], SPI_tuptable->tupdesc, 1,
 &isnull);

 elog (NOTICE, "trigf (fired %s): there are %d tuples in ttest",
 when, i);

 SPI_finish();

 if (checknull)
 {
  i = SPI_getbinval(rettuple, tupdesc, 1, &isnull);
  if (isnull)
   rettuple = NULL;
 }

 return (rettuple);
}
```

Now, compile and create table ttest (x int4):

```
create function trigf () returns opaque as
'...path_to_so' language 'c';


vac=> create trigger tbefore before insert or update or delete on
 ttest
for each row execute procedure trigf();
CREATE
vac=> create trigger tafter after insert or update or delete on
 ttest
for each row execute procedure trigf();
CREATE
vac=> insert into ttest values (null);
NOTICE:trigf (fired before): there are 0 tuples in ttest
INSERT 0 0

-- Insertion skipped and AFTER trigger is not fired
```

```
vac=> select * from ttest;
x
-
(0 rows)

vac=> insert into ttest values (1);
NOTICE:trigf (fired before): there are 0 tuples in ttest
NOTICE:trigf (fired after ): there are 1 tuples in ttest
                                       ^^^^^^^^
                             remember what we said about
 visibility.
INSERT 167793 1
vac=> select * from ttest;
x
-
1
(1 row)

vac=> insert into ttest select x * 2 from ttest;
NOTICE:trigf (fired before): there are 1 tuples in ttest
NOTICE:trigf (fired after ): there are 2 tuples in ttest
                                       ^^^^^^^^
                             remember what we said about
 visibility.
INSERT 167794 1
vac=> select * from ttest;
x
-
1
2
(2 rows)

vac=> update ttest set x = null where x = 2;
NOTICE:trigf (fired before): there are 2 tuples in ttest
UPDATE 0
vac=> update ttest set x = 4 where x = 2;
NOTICE:trigf (fired before): there are 2 tuples in ttest
NOTICE:trigf (fired after ): there are 2 tuples in ttest
UPDATE 1
vac=> select * from ttest;
x
-
1
4
(2 rows)

vac=> delete from ttest;
NOTICE:trigf (fired before): there are 2 tuples in ttest
NOTICE:trigf (fired after ): there are 1 tuples in ttest
NOTICE:trigf (fired before): there are 1 tuples in ttest
NOTICE:trigf (fired after ): there are 0 tuples in ttest
                                       ^^^^^^^^
                             remember what we said about
 visibility.
DELETE 2
vac=> select * from ttest;
x
-
```

```
(0 rows)
```

Chapter 49. Server Programming Interface

Vadim Mikheev

The *Server Programming Interface* (SPI) gives users the ability to run SQL queries inside user-defined C functions. The available Procedural Languages (PL) give an alternate means to access these capabilities.

In fact, SPI is just a set of native interface functions to simplify access to the Parser, Planner, Optimizer and Executor. SPI also does some memory management.

To avoid misunderstanding we'll use *function* to mean SPI interface functions and *procedure* for user-defined C-functions using SPI.

SPI procedures are always called by some (upper) Executor and the SPI manager uses the Executor to run your queries. Other procedures may be called by the Executor running queries from your procedure.

Note, that if during execution of a query from a procedure the transaction is aborted then control will not be returned to your procedure. Rather, all work will be rolled back and the server will wait for the next command from the client. This will be changed in future versions.

Other restrictions are the inability to execute BEGIN, END and ABORT (transaction control statements) and cursor operations. This will also be changed in the future.

If successful, SPI functions return a non-negative result (either via a returned integer value or in SPI_result global variable, as described below). On error, a negative or NULL result will be returned.

49.1. Interface Functions

Name

SPI_connect — Connects your procedure to the SPI manager.

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

```
int SPI_connect(void)
```

Inputs

None

Outputs

int

Return status

SPI_OK_CONNECT

if connected

SPI_ERROR_CONNECT

if not connected

Description

`SPI_connect` opens a connection to the Postgres backend. You should call this function if you will need to execute queries. Some utility SPI functions may be called from un-connected procedures.

You may get SPI_ERROR_CONNECT error if `SPI_connect` is called from an already connected procedure - e.g. if you directly call one procedure from another connected one. Actually, while the child procedure will be able to use SPI, your parent procedure will not be able to continue to use SPI after the child returns (if `SPI_finish` is called by the child). It's bad practice.

Usage

Algorithm

`SPI_connect` performs the following:

- Initializes the SPI internal structures for query execution and memory management.

Name

SPI_finish — Disconnects your procedure from the SPI manager.

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

```
SPI_finish(void)
```

Inputs

None

Outputs

int

SPI_OK_FINISH if properly disconnected
SPI_ERROR_UNCONNECTED if called from an un-connected procedure

Description

`SPI_finish` closes an existing connection to the Postgres backend. You should call this function after completing operations through the SPI manager.

You may get the error return SPI_ERROR_UNCONNECTED if `SPI_finish` is called without having a current valid connection. There is no fundamental problem with this; it means that nothing was done by the SPI manager.

Usage

`SPI_finish` *must* be called as a final step by a connected procedure or you may get unpredictable results! Note that you can safely skip the call to `SPI_finish` if you abort the transaction (via elog(ERROR)).

Algorithm

`SPI_finish` performs the following:

- Disconnects your procedure from the SPI manager and frees all memory allocations made by your procedure via `palloc` since the `SPI_connect`. These allocations can't be used any more! See Memory management.

Name

SPI_exec — Creates an execution plan (parser+planner+optimizer) and executes a query.

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

```
SPI_exec(query, tcount)
```

Inputs

char *query

String containing query plan

int tcount

Maximum number of tuples to return

Outputs

int

SPI_OK_EXEC if properly disconnected
SPI_ERROR_UNCONNECTED if called from an un-connected procedure
SPI_ERROR_ARGUMENT if query is NULL or tcount < 0.
SPI_ERROR_UNCONNECTED if procedure is unconnected.
SPI_ERROR_COPY if COPY TO/FROM stdin.
SPI_ERROR_CURSOR if DECLARE/CLOSE CURSOR, FETCH.
SPI_ERROR_TRANSACTION if BEGIN/ABORT/END.
SPI_ERROR_OPUNKNOWN if type of query is unknown (this shouldn't occur).

If execution of your query was successful then one of the following (non-negative) values will be returned:

SPI_OK_UTILITY if some utility (e.g. CREATE TABLE ...) was executed
SPI_OK_SELECT if SELECT (but not SELECT ... INTO!) was executed
SPI_OK_SELINTO if SELECT ... INTO was executed
SPI_OK_INSERT if INSERT (or INSERT ... SELECT) was executed
SPI_OK_DELETE if DELETE was executed
SPI_OK_UPDATE if UPDATE was executed

Description

SPI_exec creates an execution plan (parser+planner+optimizer) and executes the query for tcount tuples.

Usage

This should only be called from a connected procedure. If tcount is zero then it executes the query for all tuples returned by the query scan. Using tcount > 0 you may restrict the number of tuples for which the query will be executed. For example,

```
SPI_exec ("insert into table select * from table", 5);
```

will allow at most 5 tuples to be inserted into table. If execution of your query was successful then a non-negative value will be returned.

Note

You may pass many queries in one string or query string may be re-written by RULEs. SPI_exec returns the result for the last query executed.

The actual number of tuples for which the (last) query was executed is returned in the global variable SPI_processed (if not SPI_OK_UTILITY). If SPI_OK_SELECT returned and SPI_processed > 0 then you may use global pointer SPITupleTable *SPI_tuptable to access the selected tuples: Also NOTE, that `SPI_finish` frees and makes all SPITupleTables unusable! (See Memory management).

`SPI_exec` may return one of the following (negative) values:

SPI_ERROR_ARGUMENT if query is NULL or *`tcount`* < 0.
SPI_ERROR_UNCONNECTED if procedure is unconnected.
SPI_ERROR_COPY if COPY TO/FROM stdin.
SPI_ERROR_CURSOR if DECLARE/CLOSE CURSOR, FETCH.
SPI_ERROR_TRANSACTION if BEGIN/ABORT/END.
SPI_ERROR_OPUNKNOWN if type of query is unknown (this shouldn't occur).

Algorithm

`SPI_exec` performs the following:

- Disconnects your procedure from the SPI manager and frees all memory allocations made by your procedure via `palloc` since the `SPI_connect`. These allocations can't be used any more! See Memory management.

Name

SPI_prepare — Connects your procedure to the SPI manager.

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

```
SPI_prepare(query, nargs, argtypes)
```

Inputs

query

Query string

nargs

Number of input parameters ($1 ... $nargs - as in SQL-functions)

argtypes

Pointer list of type OIDs to input arguments

Outputs

void *

Pointer to an execution plan (parser+planner+optimizer)

Description

`SPI_prepare` creates and returns an execution plan (parser+planner+optimizer) but doesn't execute the query. Should only be called from a connected procedure.

Usage

nargs is number of parameters ($1 ... $nargs - as in SQL-functions), and nargs may be 0 only if there is not any $1 in query.

Execution of prepared execution plans is sometimes much faster so this feature may be useful if the same query will be executed many times.

The plan returned by `SPI_prepare` may be used only in current invocation of the procedure since `SPI_finish` frees memory allocated for a plan. See `SPI_saveplan`.

If successful, a non-null pointer will be returned. Otherwise, you'll get a NULL plan. In both cases SPI_result will be set like the value returned by SPI_exec, except that it is set to SPI_ERROR_ARGUMENT if query is NULL or nargs < 0 or nargs > 0 && argtypes is NULL.

Name

SPI_saveplan — Saves a passed plan

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

```
SPI_saveplan(plan)
```

Inputs

void *query

Passed plan

Outputs

void *

Execution plan location. NULL if unsuccessful.

SPI_result

SPI_ERROR_ARGUMENT if plan is NULL
SPI_ERROR_UNCONNECTED if procedure is un-connected

Description

`SPI_saveplan` stores a plan prepared by `SPI_prepare` in safe memory protected from freeing by `SPI_finish` or the transaction manager.

In the current version of Postgres there is no ability to store prepared plans in the system catalog and fetch them from there for execution. This will be implemented in future versions. As an alternative, there is the ability to reuse prepared plans in the consequent invocations of your procedure in the current session. Use `SPI_execp` to execute this saved plan.

Usage

`SPI_saveplan` saves a passed plan (prepared by `SPI_prepare`) in memory protected from freeing by `SPI_finish` and by the transaction manager and returns a pointer to the saved plan. You may save the pointer returned in a local variable. Always check if this pointer is NULL or not either when preparing a plan or using an already prepared plan in SPI_execp (see below).

Note

If one of the objects (a relation, function, etc.) referenced by the prepared plan is dropped during your session (by your backend or another process) then the results of `SPI_execp` for this plan will be unpredictable.

Name

SPI_execp — Executes a plan from `SPI_saveplan`

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

```
SPI_execp(plan,
values,
nulls,
tcount)
```

Inputs

void **plan*

Execution plan

Datum **values*

Actual parameter values

char **nulls*

Array describing what parameters get NULLs

'n' indicates NULL allowed
' ' indicates NULL not allowed

int *tcount*

Number of tuples for which plan is to be executed

Outputs

int

Returns the same value as `SPI_exec` as well as

SPI_ERROR_ARGUMENT if *plan* is NULL or *tcount* < 0
SPI_ERROR_PARAM if *values* is NULL and *plan* was prepared with some parameters.

SPI_tuptable

initialized as in `SPI_exec` if successful

SPI_processed

initialized as in `SPI_exec` if successful

Description

`SPI_execp` stores a plan prepared by `SPI_prepare` in safe memory protected from freeing by `SPI_finish` or the transaction manager.

In the current version of Postgres there is no ability to store prepared plans in the system catalog and fetch them from there for execution. This will be implemented in future versions. As a work arround, there is the ability to reuse prepared plans in the consequent invocations of your procedure in the current session. Use `SPI_execp` to execute this saved plan.

Usage

If *nulls* is NULL then SPI_execp assumes that all values (if any) are NOT NULL.

Note

If one of the objects (a relation, function, etc.) referenced by the prepared plan is dropped during your session (by your backend or another process) then the results of SPI_execp for this plan will be unpredictable.

49.2. Interface Support Functions

All functions described below may be used by connected and unconnected procedures.

Name

SPI_copytuple — Makes copy of tuple in upper Executor context

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

```
SPI_copytuple(tuple)
```

Inputs

HeapTuple *tuple*

Input tuple to be copied

Outputs

HeapTuple

Copied tuple

non-NULL if *tuple* is not NULL and the copy was successful
NULL only if *tuple* is NULL

Description

`SPI_copytuple` makes a copy of tuple in upper Executor context. See the section on Memory Management.

Usage

TBD

Name

SPI_modifytuple — Modifies tuple of relation

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

```
SPI_modifytuple(rel, tuple , nattrs
, attnum , Values , Nulls)
```

Inputs

Relation *rel*

HeapTuple *tuple*

Input tuple to be modified

int *nattrs*

Number of attribute numbers in attnum

int * *attnum*

Array of numbers of the attributes which are to be changed

Datum * *Values*

New values for the attributes specified

char * *Nulls*

Which attributes are NULL, if any

Outputs

HeapTuple

New tuple with modifications

non-NULL if *tuple* is not NULL and the modify was successful
NULL only if *tuple* is NULL

SPI_result

SPI_ERROR_ARGUMENT if rel is NULL or tuple is NULL or natts ≤ 0 or attnum is NULL or Values is NULL.
SPI_ERROR_NOATTRIBUTE if there is an invalid attribute number in attnum (attnum ≤ 0 or > number of attributes in tuple)

Description

`SPI_modifytuple` Modifies a tuple in upper Executor context. See the section on Memory Management.

Usage

If successful, a pointer to the new tuple is returned. The new tuple is allocated in upper Executor context (see Memory management). Passed tuple is not changed.

Name

SPI_fnumber — Finds the attribute number for specified attribute

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

```
SPI_fnumber(tupdesc, fname)
```

Inputs

TupleDesc *tupdesc*

Input tuple description

char * *fname*

Field name

Outputs

int

Attribute number

Valid one-based index number of attribute
SPI_ERROR_NOATTRIBUTE if the named attribute is not found

Description

`SPI_fnumber` returns the attribute number for the attribute with name in fname.

Usage

Attribute numbers are 1 based.

Name

SPI_fname — Finds the attribute name for the specified attribute

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

```
SPI_fname(tupdesc, fname)
```

Inputs

TupleDesc *tupdesc*

Input tuple description

char * *fnumber*

Attribute number

Outputs

char *

Attribute name

NULL if fnumber is out of range
SPI_result set to SPI_ERROR_NOATTRIBUTE on error

Description

`SPI_fname` returns the attribute name for the specified attribute.

Usage

Attribute numbers are 1 based.

Algorithm

Returns a newly-allocated copy of the attribute name.

Name

SPI_getvalue — Returns the string value of the specified attribute

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

```
SPI_getvalue(tuple, tupdesc, fnumber)
```

Inputs

HeapTuple *tuple*

Input tuple to be examined

TupleDesc *tupdesc*

Input tuple description

int *fnumber*

Attribute number

Outputs

char *

Attribute value or NULL if

attribute is NULL
fnumber is out of range (SPI_result set to SPI_ERROR_NOATTRIBUTE)
no output function available (SPI_result set to SPI_ERROR_NOOUTFUNC)

Description

`SPI_getvalue` returns an external (string) representation of the value of the specified attribute.

Usage

Attribute numbers are 1 based.

Algorithm

Allocates memory as required by the value.

Name

SPI_getbinval — Returns the binary value of the specified attribute

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

```
SPI_getbinval(tuple, tupdesc, fnumber, isnull)
```

Inputs

HeapTuple *tuple*

Input tuple to be examined

TupleDesc *tupdesc*

Input tuple description

int *fnumber*

Attribute number

Outputs

Datum

Attribute binary value

bool * *isnull*

flag for null value in attribute

SPI_result

SPI_ERROR_NOATTRIBUTE

Description

`SPI_getbinval` returns the binary value of the specified attribute.

Usage

Attribute numbers are 1 based.

Algorithm

Does not allocate new space for the binary value.

Name

SPI_gettype — Returns the type name of the specified attribute

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

```
SPI_gettype(tupdesc, fnumber)
```

Inputs

TupleDesc *tupdesc*

Input tuple description

int *fnumber*

Attribute number

Outputs

char *

The type name for the specified attribute number

SPI_result

SPI_ERROR_NOATTRIBUTE

Description

`SPI_gettype` returns a copy of the type name for the specified attribute.

Usage

Attribute numbers are 1 based.

Algorithm

Does not allocate new space for the binary value.

Name

SPI_gettypeid — Returns the type OID of the specified attribute

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

```
SPI_gettypeid(tupdesc, fnumber)
```

Inputs

TupleDesc *tupdesc*

Input tuple description

int *fnumber*

Attribute number

Outputs

OID

The type OID for the specified attribute number

SPI_result

SPI_ERROR_NOATTRIBUTE

Description

`SPI_gettypeid` returns the type OID for the specified attribute.

Usage

Attribute numbers are 1 based.

Algorithm

TBD

Name

SPI_getrelname — Returns the name of the specified relation

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

```
SPI_getrelname(rel)
```

Inputs

Relation *rel*

Input relation

Outputs

char *

The name of the specified relation

Description

`SPI_getrelname` returns the name of the specified relation.

Usage

TBD

Algorithm

Copies the relation name into new storage.

Name

SPI_palloc — Allocates memory in upper Executor context

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

```
SPI_palloc(size)
```

Inputs

Size *size*

Octet size of storage to allocate

Outputs

void *

New storage space of specified size

Description

`SPI_palloc` allocates memory in upper Executor context. See section on memory management.

Usage

TBD

Name

SPI_repalloc — Re-allocates memory in upper Executor context

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

```
SPI_repalloc(pointer, size)
```

Inputs

void * *pointer*

Pointer to existing storage

Size *size*

Octet size of storage to allocate

Outputs

void *

New storage space of specified size with contents copied from existing area

Description

`SPI_repalloc` re-allocates memory in upper Executor context. See section on memory management.

Usage

TBD

Name

SPI_pfree — Frees memory from upper Executor context

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

`SPI_pfree(`*`pointer`*`)`

Inputs

void * *`pointer`*

Pointer to existing storage

Outputs

None

Description

`SPI_pfree` frees memory in upper Executor context. See section on memory management.

Usage

TBD

49.3. Memory Management

Server allocates memory in memory contexts in such way that allocations made in one context may be freed by context destruction without affecting allocations made in other contexts. All allocations (via `palloc`, etc) are made in the context which are chosen as current one. You'll get unpredictable results if you'll try to free (or reallocate) memory allocated not in current context.

Creation and switching between memory contexts are subject of SPI manager memory management.

SPI procedures deal with two memory contexts: upper Executor memory context and procedure memory context (if connected).

Before a procedure is connected to the SPI manager, current memory context is upper Executor context so all allocation made by the procedure itself via `palloc/repalloc` or by SPI utility functions before connecting to SPI are made in this context.

After `SPI_connect` is called current context is the procedure's one. All allocations made via `palloc/repalloc` or by SPI utility functions (except for `SPI_copytuple`, `SPI_modifytuple`, `SPI_palloc` and `SPI_repalloc`) are made in this context.

When a procedure disconnects from the SPI manager (via `SPI_finish`) the current context is restored to the upper Executor context and all allocations made in the procedure memory context are freed and can't be used any more!

If you want to return something to the upper Executor then you have to allocate memory for this in the upper context!

SPI has no ability to automatically free allocations in the upper Executor context!

SPI automatically frees memory allocated during execution of a query when this query is done!

49.4. Visibility of Data Changes

Postgres data changes visibility rule: during a query execution, data changes made by the query itself (via SQL-function, SPI-function, triggers) are invisible to the query scan. For example, in query INSERT INTO a SELECT * FROM a tuples inserted are invisible for SELECT' scan. In effect, this duplicates the database table within itself (subject to unique index rules, of course) without recursing.

Changes made by query Q are visible by queries which are started after query Q, no matter whether they are started inside Q (during the execution of Q) or after Q is done.

49.5. Examples

This example of SPI usage demonstrates the visibility rule. There are more complex examples in in src/test/regress/regress.c and in contrib/spi.

This is a very simple example of SPI usage. The procedure execq accepts an SQL-query in its first argument and tcount in its second, executes the query using SPI_exec and returns the number of tuples for which the query executed:

```
#include "executor/spi.h" /* this is what you need to work with SPI
 */

int execq(text *sql, int cnt);

int
execq(text *sql, int cnt)
{
 int ret;
 int proc = 0;

 SPI_connect();

 ret = SPI_exec(textout(sql), cnt);

 proc = SPI_processed;
 /*
  * If this is SELECT and some tuple(s) fetched -
  * returns tuples to the caller via elog (NOTICE).
  */
 if ( ret == SPI_OK_SELECT && SPI_processed > 0 )
 {
  TupleDesc tupdesc = SPI_tuptable->tupdesc;
  SPITupleTable *tuptable = SPI_tuptable;
  char buf[8192];
  int i;

  for (ret = 0; ret < proc; ret++)
  {
   HeapTuple tuple = tuptable->vals[ret];

   for (i = 1, buf[0] = 0; i <= tupdesc->natts; i++)
    sprintf(buf + strlen (buf), " %s%s",
     SPI_getvalue(tuple, tupdesc, i),
     (i == tupdesc->natts) ? " " : " |");
   elog (NOTICE, "EXECQ: %s", buf);
  }
 }
```

```
 SPI_finish();

 return (proc);
}
```

Now, compile and create the function:

```
create function execq (text, int4) returns int4 as '...path_to_so'
 language 'c';

vac=> select execq('create table a (x int4)', 0);
execq
-----
    0
(1 row)

vac=> insert into a values (execq('insert into a values (0)',0));
INSERT 167631 1
vac=> select execq('select * from a',0);
NOTICE:EXECQ:  0 <<< inserted by execq

NOTICE:EXECQ:  1 <<< value returned by execq and inserted by upper
 INSERT

execq
-----
    2
(1 row)

vac=> select execq('insert into a select x + 2 from a',1);
execq
-----
    1
(1 row)

vac=> select execq('select * from a', 10);
NOTICE:EXECQ:  0

NOTICE:EXECQ:  1

NOTICE:EXECQ:  2 <<< 0 + 2, only one tuple inserted - as specified

execq
-----
    3            <<< 10 is max value only, 3 is real # of tuples
(1 row)

vac=> delete from a;
DELETE 3
vac=> insert into a values (execq('select * from a', 0) + 1);
INSERT 167712 1
vac=> select * from a;
x
-
1                <<< no tuples in a (0) + 1
(1 row)

vac=> insert into a values (execq('select * from a', 0) + 1);
```

```
NOTICE:EXECQ:  0
INSERT 167713 1
vac=> select * from a;
x
-
1
2               <<< there was single tuple in a + 1
(2 rows)

--   This demonstrates data changes visibility rule:

vac=> insert into a select execq('select * from a', 0) * x from a;
NOTICE:EXECQ:  1
NOTICE:EXECQ:  2
NOTICE:EXECQ:  1
NOTICE:EXECQ:  2
NOTICE:EXECQ:  2
INSERT 0 2
vac=> select * from a;
x
-
1
2
2               <<< 2 tuples * 1 (x in first tuple)
6               <<< 3 tuples (2 + 1 just inserted) * 2 (x in
 second tuple)
(4 rows)            ^^^^^^^^
                    tuples visible to execq() in different
 invocations
```

Chapter 50. Procedural Languages

Postgres supports the definition of procedural languages. In the case of a function or trigger procedure defined in a procedural language, the database has no built-in knowledge about how to interpret the function's source text. Instead, the task is passed to a handler that knows the details of the language. The handler itself is a special programming language function compiled into a shared object and loaded on demand.

Writing a handler for a new procedural language (PL) is outside the scope of this manual.

50.1. Installing Procedural Languages

Procedural Language Installation

A procedural language is installed in the database in three steps. (For the languages supplied with the standard distribution, the shell script `createlang` can be used instead of carrying out the details manually.)

1. The shared object for the language handler must be compiled and installed. By default the handler for PL/pgSQL is built and installed into the database library directory. If Tcl/Tk support is configured in, the handler for PL/Tcl is also built and installed in the same location.

2. The handler must be declared with the command

   ```
   CREATE FUNCTION handler_function_name ()
       RETURNS OPAQUE AS
       'path-to-shared-object' LANGUAGE 'C';
   ```

   The special return type of OPAQUE tells the database that this function does not return one of the defined SQL datatypes and is not directly usable in SQL statements.

3. The PL must be declared with the command

   ```
   CREATE [ TRUSTED ] PROCEDURAL LANGUAGE 'language-name'
       HANDLER handler_function_name
       LANCOMPILER 'description';
   ```

   The optional keyword TRUSTED tells whether ordinary database users that have no superuser privileges should be allowed to use this language to create functions and trigger procedures. Since PL functions are executed inside the database backend, the TRUSTED flag should only be given for languages that don't allow access to database backends internals or the filesystem. The languages PL/pgSQL and PL/Tcl are known to be trusted.

Example

1. The following command tells the database where to find the shared object for the PL/pgSQL language's call handler function.

   ```
   CREATE FUNCTION plpgsql_call_handler () RETURNS OPAQUE AS
       '/usr/local/pgsql/lib/plpgsql.so' LANGUAGE 'C';
   ```

2. The command

   ```
   CREATE TRUSTED PROCEDURAL LANGUAGE 'plpgsql'
       HANDLER plpgsql_call_handler
   ```

```
    LANCOMPILER 'PL/pgSQL';
```

then defines that the previously declared call handler function should be invoked for functions and trigger procedures where the language attribute is 'plpgsql'.

PL handler functions have a special call interface that is different from regular C language functions. One of the arguments given to the handler is the object ID in the `pg_proc` tables entry for the function that should be executed. The handler examines various system catalogs to analyze the functions call arguments and it's return data type. The source text of the functions body is found in the prosrc attribute of `pg_proc`. Due to this, PL functions can be overloaded like SQL language functions. There can be multiple different PL functions having the same function name, as long as the call arguments differ.

Procedural languages defined in the `template1` database are automatically defined in all subsequently created databases. So the database administrator can decide which languages are available by default.

Part IV. Interfaces

User and programmer interfaces.

Table of Contents

51. Functions 512
52. Large Objects 513
    52.1. Historical Note 513
    52.2. Implementation Features 513
    52.3. Interfaces 513
        52.3.1. Creating a Large Object 513
        52.3.2. Importing a Large Object 514
        52.3.3. Exporting a Large Object 514
        52.3.4. Opening an Existing Large Object 514
        52.3.5. Writing Data to a Large Object 514
        52.3.6. Reading Data from a Large Object 514
        52.3.7. Seeking on a Large Object 514
        52.3.8. Closing a Large Object Descriptor 515
        52.3.9. Removing a Large Object 515
    52.4. Built in registered functions 515
    52.5. Accessing Large Objects from LIBPQ 515
    52.6. Sample Program 515
53. ecpg - Embedded SQL in C 520
    53.1. Why Embedded SQL? 520
    53.2. The Concept 520
    53.3. How To Use ecpg 520
        53.3.1. Preprocessor 520
        53.3.2. Library 521
        53.3.3. Error handling 521
    53.4. Limitations 523
    53.5. Porting From Other RDBMS Packages 523
    53.6. Installation 524
    53.7. For the Developer 524
        53.7.1. ToDo List 524
        53.7.2. The Preprocessor 524
        53.7.3. A Complete Example 527
        53.7.4. The Library 528
54. libpq - C Library 530
    54.1. Database Connection Functions 530
    54.2. Query Execution Functions 535
    54.3. Asynchronous Query Processing 539
    54.4. Fast Path 541
    54.5. Asynchronous Notification 542
    54.6. Functions Associated with the COPY Command 543
    54.7. libpq Tracing Functions 545
    54.8. libpq Control Functions 545
    54.9. Environment Variables 545
    54.10. Threading Behavior 546
    54.11. Sample Programs 546
        54.11.1. Sample Program 1 546
        54.11.2. Sample Program 2 549
        54.11.3. Sample Program 3 551
55. libpq - C++ Binding Library 555
    55.1. Control and Initialization 555
        55.1.1. Environment Variables 555
    55.2. libpq++ Classes 556
        55.2.1. Connection Class: `PgConnection` 556
        55.2.2. Database Class: `PgDatabase` 556
    55.3. Database Connection Functions 556
    55.4. Query Execution Functions 557

55.5. Asynchronous Notification 560
55.6. Functions Associated with the COPY Command 560
56. pgtcl - TCL Binding Library 562
56.1. Commands 562
56.2. Examples 562
56.3. pgtcl Command Reference Information 563
57. libpgeasy - Simplified C Binding Library 582
58. ODBC Interface 583
58.1. Background 583
58.2. Windows Applications 583
58.2.1. Writing Applications 583
58.3. Unix Installation 584
58.3.1. Building the Driver 584
58.4. Configuration Files 587
58.5. ApplixWare 588
58.5.1. Configuration 588
58.5.2. Common Problems 589
58.5.3. Debugging ApplixWare ODBC Connections 590
58.5.4. Running the ApplixWare Demo 590
58.5.5. Useful Macros 591
58.5.6. Supported Platforms 591
59. JDBC Interface 592
59.1. Building the JDBC Interface 592
59.1.1. Compiling the Driver 592
59.1.2. Installing the Driver 592
59.2. Preparing the Database for JDBC 592
59.3. Using the Driver 593
59.4. Importing JDBC 593
59.5. Loading the Driver 593
59.6. Connecting to the Database 594
59.7. Issuing a Query and Processing the Result 594
59.7.1. Using the Statement Interface 594
59.7.2. Using the ResultSet Interface 595
59.8. Performing Updates 595
59.9. Closing the Connection 595
59.10. Using Large Objects 595
59.11. Postgres Extensions to the JDBC API 596
59.12. Further Reading 635
60. Lisp Programming Interface 636

Chapter 51. Functions

Reference information for user-callable functions.

Note

This section needs to be written. Volunteers?

Chapter 52. Large Objects

In Postgres, data values are stored in tuples and individual tuples cannot span data pages. Since the size of a data page is 8192 bytes, the upper limit on the size of a data value is relatively low. To support the storage of larger atomic values, Postgres provides a large object interface. This interface provides file oriented access to user data that has been declared to be a large type. This section describes the implementation and the programming and query language interfaces to Postgres large object data.

52.1. Historical Note

Originally, Postgres 4.2 supported three standard implementations of large objects: as files external to Postgres, as external files managed by Postgres, and as data stored within the Postgres database. It causes considerable confusion among users. As a result, we only support large objects as data stored within the Postgres database in PostgreSQL. Even though it is slower to access, it provides stricter data integrity. For historical reasons, this storage scheme is referred to as Inversion large objects. (We will use Inversion and large objects interchangeably to mean the same thing in this section.)

52.2. Implementation Features

The Inversion large object implementation breaks large objects up into "chunks" and stores the chunks in tuples in the database. A B-tree index guarantees fast searches for the correct chunk number when doing random access reads and writes.

52.3. Interfaces

The facilities Postgres provides to access large objects, both in the backend as part of user-defined functions or the front end as part of an application using the interface, are described below. For users familiar with Postgres 4.2, PostgreSQL has a new set of functions providing a more coherent interface.

> **Note**
>
> All large object manipulation *must* take place within an SQL transaction. This requirement is strictly enforced as of Postgres v6.5, though it has been an implicit requirement in previous versions, resulting in misbehavior if ignored.

The Postgres large object interface is modeled after the Unix file system interface, with analogues of `open(2)`, `read(2)`, `write(2)`, `lseek(2)`, etc. User functions call these routines to retrieve only the data of interest from a large object. For example, if a large object type called mugshot existed that stored photographs of faces, then a function called beard could be declared on mugshot data. Beard could look at the lower third of a photograph, and determine the color of the beard that appeared there, if any. The entire large object value need not be buffered, or even examined, by the beard function. Large objects may be accessed from dynamically-loaded C functions or database client programs that link the library. Postgres provides a set of routines that support opening, reading, writing, closing, and seeking on large objects.

52.3.1. Creating a Large Object

The routine

```
Oid lo_creat(PGconn *conn, int mode)
```

creates a new large object. *`mode`* is a bitmask describing several different attributes of the new object. The symbolic constants listed here are defined in `$PGROOT/src/backend/libpq/libpq-fs.h` The access type (read, write, or both) is controlled by OR ing together the bits INV_READ and INV_WRITE. If the large object should be archived -- that is, if historical versions of it should be moved periodically to a special archive relation -- then the INV_ARCHIVE bit should be set. The low-

order sixteen bits of mask are the storage manager number on which the large object should reside. For sites other than Berkeley, these bits should always be zero. The commands below create an (Inversion) large object:

```
inv_oid = lo_creat(INV_READ|INV_WRITE|INV_ARCHIVE);
```

52.3.2. Importing a Large Object

To import a UNIX file as a large object, call

```
Oid lo_import(PGconn *conn, const char *filename)
```

filename specifies the Unix pathname of the file to be imported as a large object.

52.3.3. Exporting a Large Object

To export a large object into UNIX file, call

```
int lo_export(PGconn *conn, Oid lobjId, const char *filename)
```

The lobjId argument specifies the Oid of the large object to export and the filename argument specifies the UNIX pathname of the file.

52.3.4. Opening an Existing Large Object

To open an existing large object, call

```
int lo_open(PGconn *conn, Oid lobjId, int mode)
```

The lobjId argument specifies the Oid of the large object to open. The mode bits control whether the object is opened for reading INV_READ), writing or both. A large object cannot be opened before it is created. `lo_open` returns a large object descriptor for later use in `lo_read`, `lo_write`, `lo_lseek`, `lo_tell`, and `lo_close`.

52.3.5. Writing Data to a Large Object

The routine

```
int lo_write(PGconn *conn, int fd, const char *buf, size_t len)
```

writes len bytes from buf to large object fd. The fd argument must have been returned by a previous `lo_open`. The number of bytes actually written is returned. In the event of an error, the return value is negative.

52.3.6. Reading Data from a Large Object

The routine

```
int lo_read(PGconn *conn, int fd, char *buf, size_t len)
```

reads len bytes from large object fd into byf. The fd argument must have been returned by a previous `lo_open`. The number of bytes actually read is returned. In the event of an error, the return value is negative.

52.3.7. Seeking on a Large Object

To change the current read or write location on a large object, call

```
int lo_lseek(PGconn *conn, int fd, int offset, int whence)
```

This routine moves the current location pointer for the large object described by fd to the new location specified by offset. The valid values for whence are SEEK_SET, SEEK_CUR, and SEEK_END.

52.3.8. Closing a Large Object Descriptor

A large object may be closed by calling

```
int lo_close(PGconn *conn, int fd)
```

where fd is a large object descriptor returned by `lo_open`. On success, `lo_close` returns zero. On error, the return value is negative.

52.3.9. Removing a Large Object

To remove a large object from the database, call

```
Oid lo_unlink(PGconn *conn, Oid lobjId)
```

The lobjId argument specifies the Oid of the large object to remove.

52.4. Built in registered functions

There are two built-in registered functions, lo_import and lo_export which are convenient for use in SQL queries. Here is an example of their use

```
CREATE TABLE image (
    name            text,
    raster          oid
);

INSERT INTO image (name, raster)
    VALUES ('beautiful image', lo_import('/etc/motd'));

SELECT lo_export(image.raster, '/tmp/motd') from image
    WHERE name = 'beautiful image';
```

52.5. Accessing Large Objects from LIBPQ

Below is a sample program which shows how the large object interface in LIBPQ can be used. Parts of the program are commented out but are left in the source for the readers benefit. This program can be found in `../src/test/examples` Frontend applications which use the large object interface in LIBPQ should include the header file libpq/libpq-fs.h and link with the libpq library.

52.6. Sample Program

```
/*--------------------------------------------------------------
 *
 * testlo.c--
 *    test using large objects with libpq
 *
 * Copyright (c) 1994, Regents of the University of California
 *
 *
 * IDENTIFICATION
 *    /usr/local/devel/pglite/cvs/src/doc/manual.me,v 1.16
 1995/09/01 23:55:00 jolly Exp
 *
 *--------------------------------------------------------------
```

```
 */
#include <stdio.h>
#include "libpq-fe.h"
#include "libpq/libpq-fs.h"

#define BUFSIZE          1024

/*
 * importFile *    import file "in_filename" into database as large
 object "lobjOid"
 *
 */
Oid importFile(PGconn *conn, char *filename)
{
    Oid lobjId;
    int lobj_fd;
    char buf[BUFSIZE];
    int nbytes, tmp;
    int fd;

    /*
     * open the file to be read in
     */
    fd = open(filename, O_RDONLY, 0666);
    if (fd < 0)  {   /* error */
     fprintf(stderr, "can't open unix file %s\n", filename);
    }

    /*
     * create the large object
     */
    lobjId = lo_creat(conn, INV_READ|INV_WRITE);
    if (lobjId == 0) {
     fprintf(stderr, "can't create large object\n");
    }

    lobj_fd = lo_open(conn, lobjId, INV_WRITE);
    /*
     * read in from the Unix file and write to the inversion file
     */
    while ((nbytes = read(fd, buf, BUFSIZE)) > 0) {
     tmp = lo_write(conn, lobj_fd, buf, nbytes);
     if (tmp < nbytes) {
         fprintf(stderr, "error while reading large object\n");
     }
    }

    (void) close(fd);
    (void) lo_close(conn, lobj_fd);

    return lobjId;
}

void pickout(PGconn *conn, Oid lobjId, int start, int len)
{
    int lobj_fd;
    char* buf;
    int nbytes;
```

```
    int nread;

    lobj_fd = lo_open(conn, lobjId, INV_READ);
    if (lobj_fd < 0) {
     fprintf(stderr,"can't open large object %d\n",
          lobjId);
    }

    lo_lseek(conn, lobj_fd, start, SEEK_SET);
    buf = malloc(len+1);

    nread = 0;
    while (len - nread > 0) {
     nbytes = lo_read(conn, lobj_fd, buf, len - nread);
     buf[nbytes] = ' ';
     fprintf(stderr,">>> %s", buf);
     nread += nbytes;
    }
    fprintf(stderr,"\n");
    lo_close(conn, lobj_fd);
}

void overwrite(PGconn *conn, Oid lobjId, int start, int len)
{
    int lobj_fd;
    char* buf;
    int nbytes;
    int nwritten;
    int i;

    lobj_fd = lo_open(conn, lobjId, INV_READ);
    if (lobj_fd < 0) {
     fprintf(stderr,"can't open large object %d\n",
          lobjId);
    }

    lo_lseek(conn, lobj_fd, start, SEEK_SET);
    buf = malloc(len+1);

    for (i=0;i<len;i++)
     buf[i] = 'X';
    buf[i] = ' ';

    nwritten = 0;
    while (len - nwritten > 0) {
     nbytes = lo_write(conn, lobj_fd, buf + nwritten, len -
 nwritten);
     nwritten += nbytes;
    }
    fprintf(stderr,"\n");
    lo_close(conn, lobj_fd);
}

/*
 * exportFile *    export large object "lobjOid" to file
 "out_filename"
 *
 */
```

```
void exportFile(PGconn *conn, Oid lobjId, char *filename)
{
    int lobj_fd;
    char buf[BUFSIZE];
    int nbytes, tmp;
    int fd;

    /*
     * create an inversion "object"
     */
    lobj_fd = lo_open(conn, lobjId, INV_READ);
    if (lobj_fd < 0) {
     fprintf(stderr,"can't open large object %d\n",
          lobjId);
    }

    /*
     * open the file to be written to
     */
    fd = open(filename, O_CREAT|O_WRONLY, 0666);
    if (fd < 0)  {   /* error */
     fprintf(stderr, "can't open unix file %s\n",
          filename);
    }

    /*
     * read in from the Unix file and write to the inversion file
     */
    while ((nbytes = lo_read(conn, lobj_fd, buf, BUFSIZE)) > 0) {
     tmp = write(fd, buf, nbytes);
        if (tmp < nbytes) {
         fprintf(stderr,"error while writing %s\n",
              filename);
     }
    }

    (void) lo_close(conn, lobj_fd);
    (void) close(fd);

    return;
}

void
exit_nicely(PGconn* conn)
{
  PQfinish(conn);
  exit(1);
}

int
main(int argc, char **argv)
{
    char *in_filename, *out_filename;
    char *database;
    Oid lobjOid;
    PGconn *conn;
    PGresult *res;
```

```
    if (argc != 4) {
     fprintf(stderr, "Usage: %s database_name in_filename
 out_filename\n",
          argv[0]);
     exit(1);
    }

    database = argv[1];
    in_filename = argv[2];
    out_filename = argv[3];

    /*
     * set up the connection
     */
    conn = PQsetdb(NULL, NULL, NULL, NULL, database);

    /* check to see that the backend connection was successfully
 made */
    if (PQstatus(conn) == CONNECTION_BAD) {
     fprintf(stderr,"Connection to database '%s' failed.\n",
 database);
     fprintf(stderr,"%s",PQerrorMessage(conn));
     exit_nicely(conn);
    }

    res = PQexec(conn, "begin");
    PQclear(res);

    printf("importing file %s\n", in_filename);
/*  lobjOid = importFile(conn, in_filename); */
    lobjOid = lo_import(conn, in_filename);
/*
    printf("as large object %d.\n", lobjOid);

    printf("picking out bytes 1000-2000 of the large object\n");
    pickout(conn, lobjOid, 1000, 1000);

    printf("overwriting bytes 1000-2000 of the large object with
 X's\n");
    overwrite(conn, lobjOid, 1000, 1000);
*/

    printf("exporting large object to file %s\n", out_filename);
/*    exportFile(conn, lobjOid, out_filename); */
    lo_export(conn, lobjOid,out_filename);

    res = PQexec(conn, "end");
    PQclear(res);
    PQfinish(conn);
    exit(0);
}
```

Chapter 53. ecpg - Embedded SQL in C

Linus Tolke
Michael Meskes

Copyright © 1996-1997 Linus Tolke
Copyright © 1998 Michael Meskes

This describes an embedded SQL in C package for Postgres. It is written by Linus Tolke[1] and Michael Meskes[2].

> **Note**
>
> Permission is granted to copy and use in the same way as you are allowed to copy and use the rest of PostgreSQL.

53.1. Why Embedded SQL?

Embedded SQL has some small advantages over other ways to handle SQL queries. It takes care of all the tedious moving of information to and from variables in your C program. Many RDBMS packages support this embedded language.

There is an ANSI-standard describing how the embedded language should work. ecpg was designed to meet this standard as much as possible. So it is possible to port programs with embedded SQL written for other RDBMS packages to Postgres and thus promoting the spirit of free software.

53.2. The Concept

You write your program in C with some special SQL things. For declaring variables that can be used in SQL statements you need to put them in a special declare section. You use a special syntax for the SQL queries.

Before compiling you run the file through the embedded SQL C preprocessor and it converts the SQL statements you used to function calls with the variables used as arguments. Both variables that are used as input to the SQL statements and variables that will contain the result are passed.

Then you compile and at link time you link with a special library that contains the functions used. These functions (actually it is mostly one single function) fetches the information from the arguments, performs the SQL query using the ordinary interface (`libpq`) and puts back the result in the arguments dedicated for output.

Then you run your program and when the control arrives to the SQL statement the SQL statement is performed against the database and you can continue with the result.

53.3. How To Use ecpg

This section describes how to use the ecpg tool.

53.3.1. Preprocessor

The preprocessor is called ecpg. After installation it resides in the Postgres `bin/` directory.

[1] linus@epact.se
[2] meskes@debian.org

53.3.2. Library

The ecpg library is called `libecpg.a` or `libecpg.so`. Additionally, the library uses the `libpq` library for communication to the Postgres server so you will have to link your program with *`-lecpg -lpq`*.

The library has some methods that are "hidden" but that could prove very useful sometime.

- `ECPGdebug(int` *`on`*`, FILE *`*`stream`*`)` turns on debug logging if called with the first argument non-zero. Debug logging is done on *`stream`*. Most SQL statement logs its arguments and result.

  The most important one (`ECPGdo`) that is called on almost all SQL statements logs both its expanded string, i.e. the string with all the input variables inserted, and the result from the Postgres server. This can be very useful when searching for errors in your SQL statements.

- `ECPGstatus()` This method returns TRUE if we are connected to a database and FALSE if not.

53.3.3. Error handling

To be able to detect errors from the Postgres server you include a line like

```
exec sql include sqlca;
```

in the include section of your file. This will define a struct and a variable with the name *`sqlca`* as following:

```
struct sqlca
{
 char sqlcaid[8];
 long sqlabc;
 long sqlcode;
 struct
 {
  int sqlerrml;
  char sqlerrmc[70];
 } sqlerrm;
 char sqlerrp[8];
 long sqlerrd[6];
 /* 0: empty                                        */
 /* 1: OID of processed tuple if applicable         */
 /* 2: number of rows processed in an INSERT, UPDATE */
 /*    or DELETE statement                          */
 /* 3: empty                                        */
 /* 4: empty                                        */
 /* 5: empty                                        */
 char sqlwarn[8];
 /* 0: set to 'W' if at least one other is 'W'      */
 /* 1: if 'W' at least one character string      */
 /*    value was truncated when it was              */
 /*    stored into a host variable.                 */
 /* 2: empty                                        */
 /* 3: empty                                        */
 /* 4: empty                                        */
 /* 5: empty                                        */
 /* 6: empty                                        */
 /* 7: empty                                        */
```

```
 char sqlext[8];
} sqlca;
```

If an error occured in the last SQL statement then *`sqlca.sqlcode`* will be non-zero. If *`sqlca.sqlcode`* is less that 0 then this is some kind of serious error, like the database definition does not match the query given. If it is bigger than 0 then this is a normal error like the table did not contain the requested row.

sqlca.sqlerrm.sqlerrmc will contain a string that describes the error. The string ends with the line number in the source file.

List of errors that can occur:

-12, Out of memory in line %d.

Does not normally occur. This is a sign that your virtual memory is exhausted.

-200, Unsupported type %s on line %d.

Does not normally occur. This is a sign that the preprocessor has generated something that the library does not know about. Perhaps you are running incompatible versions of the preprocessor and the library.

-201, Too many arguments line %d.

This means that Postgres has returned more arguments than we have matching variables. Perhaps you have forgotten a couple of the host variables in the `INTO :var1,:var2`-list.

-202, Too few arguments line %d.

This means that Postgres has returned fewer arguments than we have host variables. Perhaps you have too many host variables in the `INTO :var1,:var2`-list.

-203, Too many matches line %d.

This means that the query has returned several lines but the variables specified are no arrays. The `SELECT` you made probably was not unique.

-204, Not correctly formatted int type: %s line %d.

This means that the host variable is of an `int` type and the field in the Postgres database is of another type and contains a value that cannot be interpreted as an `int`. The library uses `strtol` for this conversion.

-205, Not correctly formatted unsigned type: %s line %d.

This means that the host variable is of an `unsigned int` type and the field in the Postgres database is of another type and contains a value that cannot be interpreted as an `unsigned int`. The library uses `strtoul` for this conversion.

-206, Not correctly formatted floating point type: %s line %d.

This means that the host variable is of a `float` type and the field in the Postgres database is of another type and contains a value that cannot be interpreted as an `float`. The library uses `strtod` for this conversion.

-207, Unable to convert %s to bool on line %d.

This means that the host variable is of a `bool` type and the field in the Postgres database is neither 't' nor 'f'.

-208, Empty query line %d.

Postgres returned PGRES_EMPTY_QUERY, probably because the query indeed was empty.

-220, No such connection %s in line %d.

The program tries to access a connection that does not exist.

-221, Not connected in line %d.

The program tries to access a connection that does exist but is not open.

-230, Invalid statement name %s in line %d.

The statement you are trying to use has not been prepared.

-400, Postgres error: %s line %d.

Some Postgres error. The message contains the error message from the Postgres backend.

-401, Error in transaction processing line %d.

Postgres signalled to us that we cannot start, commit or rollback the transaction.

-402, connect: could not open database %s.

The connect to the database did not work.

100, Data not found line %d.

This is a "normal" error that tells you that what you are quering cannot be found or we have gone through the cursor.

53.4. Limitations

What will never be included and why or what cannot be done with this concept.

Oracle's single tasking possibility

Oracle version 7.0 on AIX 3 uses the OS-supported locks on the shared memory segments and allows the application designer to link an application in a so called single tasking way. Instead of starting one client process per application process both the database part and the application part is run in the same process. In later versions of Oracle this is no longer supported.

This would require a total redesign of the Postgres access model and that effort can not justify the performance gained.

53.5. Porting From Other RDBMS Packages

The design of ecpg follows SQL standard. So porting from a standard RDBMS should not be a problem. Unfortunately there is no such thing as a standard RDBMS. So ecpg also tries to understand syntax additions as long as they do not create conflicts with the standard.

The following list shows all the known incompatibilities. If you find one not listed please notify Michael Meskes[3]. Note, however, that we list only incompatibilities from a precompiler of another RDBMS to ecpg and not additional ecpg features that these RDBMS do not have.

Syntax of FETCH command

The standard syntax of the FETCH command is:

[3] meskes@debian.org

FETCH [direction] [amount] IN|FROM *cursor name*.

ORACLE, however, does not use the keywords IN resp. FROM. This feature cannot be added since it would create parsing conflicts.

53.6. Installation

Since version 0.5 ecpg is distributed together with Postgres. So you should get your precompiler, libraries and header files compiled and installed by default as a part of your installation.

53.7. For the Developer

This section is for those who want to develop the ecpg interface. It describes how the things work. The ambition is to make this section contain things for those that want to have a look inside and the section on How to use it should be enough for all normal questions. So, read this before looking at the internals of the ecpg. If you are not interested in how it really works, skip this section.

53.7.1. ToDo List

This version the preprocessor has some flaws:

Library functions

to_date et al. do not exists. But then Postgres has some good conversion routines itself. So you probably won't miss these.

Structures ans unions

Structures and unions have to be defined in the declare section.

Missing statements

The following statements are not implemented thus far:

exec sql allocate

exec sql deallocate

SQLSTATE

message 'no data found'

The error message for "no data" in an exec sql insert select from statement has to be 100.

sqlwarn[6]

sqlwarn[6] should be 'W' if the PRECISION or SCALE value specified in a SET DESCRIPTOR statement will be ignored.

53.7.2. The Preprocessor

The first four lines written to the output are constant additions by ecpg. These are two comments and two include lines necessary for the interface to the library.

Then the preprocessor works in one pass only, reading the input file and writing to the output as it goes along. Normally it just echoes everything to the output without looking at it further.

When it comes to an `EXEC SQL` statements it intervenes and changes them depending on what it is. The `EXEC SQL` statement can be one of these:

Declare sections

Declare sections begins with

```
exec sql begin declare section;
```

and ends with

```
exec sql end declare section;
```

In the section only variable declarations are allowed. Every variable declare within this section is also entered in a list of variables indexed on their name together with the corresponding type.

In particular the definition of a structure or union also has to be listed inside a declare section. Otherwise ecpg cannot handle these types since it simply does not know the definition.

The declaration is echoed to the file to make the variable a normal C-variable also.

The special types VARCHAR and VARCHAR2 are converted into a named struct for every variable. A declaration like:

```
VARCHAR var[180];
```

is converted into

```
struct varchar_var { int len; char arr[180]; } var;
```

Include statements

An include statement looks like:

```
exec sql include filename;
```

Note that this is NOT the same as

```
#include <filename.h>
```

Instead the file specified is parsed by ecpg itself. So the contents of the specified file is included in the resulting C code. This way you are able to specify EXEC SQL commands in an include file.

Connect statement

A connect statement looks like:

```
exec sql connect to connection target;
```

It creates a connection to the specified database.

The *`connection target`* can be specified in the following ways:

dbname[@server][:port][as *`connection name`*][user *`user name`*]

tcp:postgresql://server[:port][/dbname][as *connection name*][user *user name*]

unix:postgresql://server[:port][/dbname][as *connection name*][user *user name*]

character variable[as *connection name*][user *user name*]

character string[as *connection name*][*user*]

default

user

There are also different ways to specify the user name:

userid

userid/*password*

userid identified by *password*

userid using *password*

Finally the userid and the password. Each may be a constant text, a character variable or a chararcter string.

Disconnect statements

A disconnect statement looks loke:

```
exec sql disconnect [connection target];
```

It closes the connection to the specified database.

The *connection target* can be specified in the following ways:

connection name

default

current

all

Open cursor statement

An open cursor statement looks like:

```
exec sql open cursor;
```

and is ignore and not copied from the output.

Commit statement

A commit statement looks like

```
exec sql commit;
```

and is translated on the output to

```
ECPGcommit(__LINE__);
```

Rollback statement

A rollback statement looks like

```
exec sql rollback;
```

and is translated on the output to

```
ECPGrollback(__LINE__);
```

Other statements

Other SQL statements are other statements that start with `exec sql` and ends with `;`. Everything inbetween is treated as an SQL statement and parsed for variable substitution.

Variable substitution occur when a symbol starts with a colon (`:`). Then a variable with that name is looked for among the variables that were previously declared within a declare section and depending on the variable being for input or output the pointers to the variables are written to the output to allow for access by the function.

For every variable that is part of the SQL request the function gets another ten arguments:

The type as a special symbol.
A pointer to the value or a pointer to the pointer.
The size of the variable if it is a char or varchar.
Number of elements in the array (for array fetches).
The offset to the next element in the array (for array fetches)
The type of the indicator variable as a special symbol.
A pointer to the value of the indicator variable or a pointer to the pointer of the indicator variable.
0.
Number of elements in the indicator array (for array fetches).
The offset to the next element in the indicator array (for array fetches)

53.7.3. A Complete Example

Here is a complete example describing the output of the preprocessor of a file foo.pgc:

```
exec sql begin declare section;
int index;
int result;
exec sql end declare section;
...
```

```
exec sql select res into :result from mytable where index = :index;
```

is translated into:

```
/* Processed by ecpg (2.6.0) */
/* These two include files are added by the preprocessor */
#include <ecpgtype.h>;
#include <ecpglib.h>;

/* exec sql begin declare section */

#line 1 "foo.pgc"

 int index;
 int result;
/* exec sql end declare section */
...
ECPGdo(__LINE__, NULL, "select  res  from mytable where index = ?
   ",
 ECPGt_int,&(index),1L,1L,sizeof(int),
        ECPGt_NO_INDICATOR, NULL , 0L, 0L, 0L, ECPGt_EOIT,
        ECPGt_int,&(result),1L,1L,sizeof(int),
        ECPGt_NO_INDICATOR, NULL , 0L, 0L, 0L, ECPGt_EORT);
#line 147 "foo.pgc"
```

(the indentation in this manual is added for readability and not something that the preprocessor can do.)

53.7.4. The Library

The most important function in the library is the `ECPGdo` function. It takes a variable amount of arguments. Hopefully we will not run into machines with limits on the amount of variables that can be accepted by a vararg function. This could easily add up to 50 or so arguments.

The arguments are:

A line number

This is a line number for the original line used in error messages only.

A string

This is the SQL request that is to be issued. This request is modified by the input variables, i.e. the variables that where not known at compile time but are to be entered in the request. Where the variables should go the string contains ";".

Input variables

As described in the section about the preprocessor every input variable gets ten arguments.

ECPGt_EOIT

An enum telling that there are no more input variables.

Output variables

As described in the section about the preprocessor every input variable gets ten arguments. These variables are filled by the function.

ECPGt_EORT

An enum telling that there are no more variables.

All the SQL statements are performed in one transaction unless you issue a commit transaction. To get this auto-transaction going the first statement or the first after statement after a commit or rollback always begins a transaction. To disable this feature per default use the `-t` option on the commandline.

To be completed: entries describing the other entries.

Chapter 54. libpq - C Library

libpq is the C application programmer's interface to Postgres. libpq is a set of library routines that allow client programs to pass queries to the Postgres backend server and to receive the results of these queries. libpq is also the underlying engine for several other Postgres application interfaces, including libpq++ (C++), libpgtcl (Tcl), Perl, and ecpg. So some aspects of libpq's behavior will be important to you if you use one of those packages.

Three short programs are included at the end of this section to show how to write programs that use libpq. There are several complete examples of libpq applications in the following directories:

```
../src/test/regress
../src/test/examples
../src/bin/psql
```

Frontend programs which use libpq must include the header file libpq-fe.h and must link with the libpq library.

54.1. Database Connection Functions

The following routines deal with making a connection to a Postgres backend server. The application program can have several backend connections open at one time. (One reason to do that is to access more than one database.) Each connection is represented by a PGconn object which is obtained from PQconnectdb() or PQsetdbLogin(). Note that these functions will always return a non-null object pointer, unless perhaps there is too little memory even to allocate the PGconn object. The PQstatus function should be called to check whether a connection was successfully made before queries are sent via the connection object.

- PQconnectdb Makes a new connection to the database server.

  ```
  PGconn *PQconnectdb(const char *conninfo)
  ```

  This routine opens a new database connection using the parameters taken from the string conninfo. Unlike PQsetdbLogin() below, the parameter set can be extended without changing the function signature, so use either of this routine or the non-blocking analogues PQconnectStart / PQconnectPoll is prefered for application programming. The passed string can be empty to use all default parameters, or it can contain one or more parameter settings separated by whitespace.

  Each parameter setting is in the form keyword = value. (To write a null value or a value containing spaces, surround it with single quotes, e.g., keyword = 'a value'. Single quotes within the value must be written as \'. Spaces around the equal sign are optional.) The currently recognized parameter keywords are:

  host

  Name of host to connect to. If a non-zero-length string is specified, TCP/IP communication is used. Using this parameter causes a hostname look-up. See hostaddr.

  hostaddr

  IP address of host to connect to. This should be in standard numbers-and-dots form, as used by the BSD functions inet_aton et al. If a non-zero-length string is specified, TCP/IP communication is used.

  Using hostaddr instead of host allows the application to avoid a host name look-up, which may be important in applications with time constraints. However, Kerberos authentication requires the host name. The following therefore applies. If host is specified without hostaddr, a

hostname look-up is forced. If hostaddr is specified without host, the value for hostaddr gives the remote address; if Kerberos is used, this causes a reverse name query. If both host and hostaddr are specified, the value for hostaddr gives the remote address; the value for host is ignored, unless Kerberos is used, in which case that value is used for Kerberos authentication. Note that authentication is likely to fail if libpq is passed a host name which is not the name of the machine at hostaddr.

Without both a host name and host address, libpq will connect using a local Unix domain socket.

`port`

Port number to connect to at the server host, or socket filename extension for Unix-domain connections.

`dbname`

The database name.

`user`

User name to connect as.

`password`

Password to be used if the server demands password authentication.

`options`

Trace/debug options to be sent to the server.

`tty`

A file or tty for optional debug output from the backend.

If any parameter is unspecified, then the corresponding environment variable (see "Environment Variables" section) is checked. If the environment variable is not set either, then hardwired defaults are used. The return value is a pointer to an abstract struct representing the connection to the backend.

- `PQsetdbLogin` Makes a new connection to the database server.

```
PGconn *PQsetdbLogin(const char *pghost,
                     const char *pgport,
                     const char *pgoptions,
                     const char *pgtty,
                     const char *dbName,
                     const char *login,
                     const char *pwd)
```

This is the predecessor of `PQconnectdb` with a fixed number of parameters but the same functionality.

- `PQsetdb` Makes a new connection to the database server.

```
PGconn *PQsetdb(char *pghost,
                char *pgport,
                char *pgoptions,
                char *pgtty,
                char *dbName)
```

This is a macro that calls `PQsetdbLogin()` with null pointers for the login and pwd parameters. It is provided primarily for backward compatibility with old programs.

- `PQconnectStart` `PQconnectPoll` Make a connection to the database server in a non-blocking manner.

  ```
  PGconn *PQconnectStart(const char *conninfo)

  PostgresPollingStatusType *PQconnectPoll(PQconn *conn)
  ```

  These two routines are used to open a connection to a database server such that your application's thread of execution is not blocked on remote I/O whilst doing so.

  The database connection is made using the parameters taken from the string `conninfo`, passed to PQconnectStart. This string is in the same format as described above for PQconnectdb.

  Neither PQconnectStart nor PQconnectPoll will block, as long as a number of restrictions are met:

  - The hostaddr and host parameters are used appropriately to ensure that name and reverse name queries are not made. See the documentation of these parameters under PQconnectdb above for details.
  - If you call PQtrace, ensure that the stream object into which you trace will not block.
  - You ensure for yourself that the socket is in the appropriate state before calling PQconnectPoll, as described below.

  To begin, call `conn=PQconnectStart("<connection_info_string>")`. If conn is NULL, then libpq has been unable to allocate a new PGconn structure. Otherwise, a valid PGconn pointer is returned (though not yet representing a valid connection to the database). On return from PQconnectStart, call status=PQstatus(conn). If status equals CONNECTION_BAD, PQconnectStart has failed.

  If PQconnectStart succeeds, the next stage is to poll libpq so that it may proceed with the connection sequence. Loop thus: Consider a connection 'inactive' by default. If PQconnectPoll last returned PGRES_POLLING_ACTIVE, consider it 'active' instead. If PQconnectPoll(conn) last returned PGRES_POLLING_READING, perform a select for reading on PQsocket(conn). If it last returned PGRES_POLLING_WRITING, perform a select for writing on PQsocket(conn). If you have yet to call PQconnectPoll, i.e. after the call to PQconnectStart, behave as if it last returned PGRES_POLLING_WRITING. If the select shows that the socket is ready, consider it 'active'. If it has been decided that this connection is 'active', call PQconnectPoll(conn) again. If this call returns PGRES_POLLING_FAILED, the connection procedure has failed. If this call returns PGRES_POLLING_OK, the connection has been successfully made.

  Note that the use of select() to ensure that the socket is ready is merely a (likely) example; those with other facilities available, such as a poll() call, may of course use that instead.

  At any time during connection, the status of the connection may be checked, by calling PQstatus. If this is CONNECTION_BAD, then the connection procedure has failed; if this is CONNECTION_OK, then the connection is ready. Either of these states should be equally detectable from the return value of PQconnectPoll, as above. Other states may be shown during (and only during) an asynchronous connection procedure. These indicate the current stage of the connection procedure, and may be useful to provide feedback to the user for example. These statuses may include:

  - CONNECTION_STARTED: Waiting for connection to be made.
  - CONNECTION_MADE: Connection OK; waiting to send.
  - CONNECTION_AWAITING_RESPONSE: Waiting for a response from the postmaster.
  - CONNECTION_AUTH_OK: Received authentication; waiting for backend startup.

• CONNECTION_SETENV: Negotiating environment.

Note that, although these constants will remain (in order to maintain compatibility) an application should never rely upon these appearing in a particular order, or at all, or on the status always being one of these documented values. An application may do something like this:

```
    switch(PQstatus(conn))
    {
        case CONNECTION_STARTED:
            feedback = "Connecting...";
     break;

        case CONNECTION_MADE:
            feedback = "Connected to server...";
            break;
.
.
.
        default:
     feedback = "Connecting...";
    }
```

Note that if PQconnectStart returns a non-NULL pointer, you must call PQfinish when you are finished with it, in order to dispose of the structure and any associated memory blocks. This must be done even if a call to PQconnectStart or PQconnectPoll failed.

PQconnectPoll will currently block if libpq is compiled with USE_SSL defined. This restriction may be removed in the future.

PQconnectPoll will currently block under Windows, unless libpq is compiled with WIN32_NON_BLOCKING_CONNECTIONS defined. This code has not yet been tested under Windows, and so it is currently off by default. This may be changed in the future.

These functions leave the socket in a non-blocking state as if `PQsetnonblocking` had been called.

• `PQconndefaults` Returns the default connection options.

```
PQconninfoOption *PQconndefaults(void)

struct PQconninfoOption
{
    char   *keyword;   /* The keyword of the option */
    char   *envvar;    /* Fallback environment variable name */
    char   *compiled;  /* Fallback compiled in default value */
    char   *val;       /* Option's current value, or NULL */
    char   *label;     /* Label for field in connect dialog */
    char   *dispchar;  /* Character to display for this field
                          in a connect dialog. Values are:
                          ""        Display entered value as is
                          "*"       Password field - hide value
                          "D"       Debug option - don't show by
 default */
    int     dispsize;  /* Field size in characters for dialog */
}
```

Returns a connection options array. This may be used to determine all possible PQconnectdb options and their current default values. The return value points to an array of PQconninfoOption structs, which ends with an entry having a NULL keyword pointer. Note that the default values ("val" fields)

will depend on environment variables and other context. Callers must treat the connection options data as read-only.

After processing the options array, free it by passing it to PQconninfoFree(). If this is not done, a small amount of memory is leaked for each call to PQconndefaults().

In Postgres versions before 7.0, PQconndefaults() returned a pointer to a static array, rather than a dynamically allocated array. That wasn't thread-safe, so the behavior has been changed.

- `PQfinish` Close the connection to the backend. Also frees memory used by the PGconn object.

  ```
  void PQfinish(PGconn *conn)
  ```

  Note that even if the backend connection attempt fails (as indicated by PQstatus), the application should call PQfinish to free the memory used by the PGconn object. The PGconn pointer should not be used after PQfinish has been called.

- `PQreset` Reset the communication port with the backend.

  ```
  void PQreset(PGconn *conn)
  ```

  This function will close the connection to the backend and attempt to reestablish a new connection to the same postmaster, using all the same parameters previously used. This may be useful for error recovery if a working connection is lost.

- `PQresetStart PQresetPoll` Reset the communication port with the backend, in a non-blocking manner.

  ```
  int PQresetStart(PGconn *conn);

  PostgresPollingStatusType PQresetPoll(PGconn *conn);
  ```

  These functions will close the connection to the backend and attempt to reestablish a new connection to the same postmaster, using all the same parameters previously used. This may be useful for error recovery if a working connection is lost. They differ from PQreset (above) in that they act in a non-blocking manner. These functions suffer from the same restrictions as PQconnectStart and PQconnectPoll.

  Call PQresetStart. If it returns 0, the reset has failed. If it returns 1, poll the reset using PQresetPoll in exactly the same way as you would create the connection using PQconnectPoll.

libpq application programmers should be careful to maintain the PGconn abstraction. Use the accessor functions below to get at the contents of PGconn. Avoid directly referencing the fields of the PGconn structure because they are subject to change in the future. (Beginning in Postgres release 6.4, the definition of struct PGconn is not even provided in `libpq-fe.h`. If you have old code that accesses PGconn fields directly, you can keep using it by including `libpq-int.h` too, but you are encouraged to fix the code soon.)

- `PQdb` Returns the database name of the connection.

  ```
  char *PQdb(const PGconn *conn)
  ```

  PQdb and the next several functions return the values established at connection. These values are fixed for the life of the PGconn object.

- `PQuser` Returns the user name of the connection.

  ```
  char *PQuser(const PGconn *conn)
  ```

- `PQpass` Returns the password of the connection.

  ```
  char *PQpass(const PGconn *conn)
  ```

- `PQhost` Returns the server host name of the connection.

  ```
  char *PQhost(const PGconn *conn)
  ```

- `PQport` Returns the port of the connection.

  ```
  char *PQport(const PGconn *conn)
  ```

- `PQtty` Returns the debug tty of the connection.

  ```
  char *PQtty(const PGconn *conn)
  ```

- `PQoptions` Returns the backend options used in the connection.

  ```
  char *PQoptions(const PGconn *conn)
  ```

- `PQstatus` Returns the status of the connection.

  ```
  ConnStatusType PQstatus(const PGconn *conn)
  ```

  The status can be one of a number of values. However, only two of these are seen outside of an asynchronous connection procedure - `CONNECTION_OK` or `CONNECTION_BAD`. A good connection to the database has the status CONNECTION_OK. A failed connection attempt is signaled by status `CONNECTION_BAD`. Ordinarily, an OK status will remain so until `PQfinish`, but a communications failure might result in the status changing to `CONNECTION_BAD` prematurely. In that case the application could try to recover by calling `PQreset`.

  See the entry for PQconnectStart and PQconnectPoll with regards to other status codes that might be seen.

- `PQerrorMessage` Returns the error message most recently generated by an operation on the connection.

  ```
  char *PQerrorMessage(const PGconn* conn);
  ```

  Nearly all libpq functions will set `PQerrorMessage` if they fail. Note that by libpq convention, a non-empty `PQerrorMessage` will include a trailing newline.

- `PQbackendPID` Returns the process ID of the backend server handling this connection.

  ```
  int PQbackendPID(const PGconn *conn);
  ```

  The backend PID is useful for debugging purposes and for comparison to NOTIFY messages (which include the PID of the notifying backend). Note that the PID belongs to a process executing on the database server host, not the local host!

54.2. Query Execution Functions

Once a connection to a database server has been successfully established, the functions described here are used to perform SQL queries and commands.

- `PQexec` Submit a query to Postgres and wait for the result.

  ```
  PGresult *PQexec(PGconn *conn,
                   const char *query);
  ```

  Returns a PGresult pointer or possibly a NULL pointer. A non-NULL pointer will generally be returned except in out-of-memory conditions or serious errors such as inability to send the query to the backend. If a NULL is returned, it should be treated like a PGRES_FATAL_ERROR result. Use PQerrorMessage to get more information about the error.

The PGresult structure encapsulates the query result returned by the backend. libpq application programmers should be careful to maintain the PGresult abstraction. Use the accessor functions below to get at the contents of PGresult. Avoid directly referencing the fields of the PGresult structure because they are subject to change in the future. (Beginning in Postgres release 6.4, the definition of struct PGresult is not even provided in libpq-fe.h. If you have old code that accesses PGresult fields directly, you can keep using it by including libpq-int.h too, but you are encouraged to fix the code soon.)

- PQresultStatus Returns the result status of the query.

  ```
  ExecStatusType PQresultStatus(const PGresult *res)
  ```

  PQresultStatus can return one of the following values:

  - PGRES_EMPTY_QUERY -- The string sent to the backend was empty.
  - PGRES_COMMAND_OK -- Successful completion of a command returning no data
  - PGRES_TUPLES_OK -- The query successfully executed
  - PGRES_COPY_OUT -- Copy Out (from server) data transfer started
  - PGRES_COPY_IN -- Copy In (to server) data transfer started
  - PGRES_BAD_RESPONSE -- The server's response was not understood
  - PGRES_NONFATAL_ERROR
  - PGRES_FATAL_ERROR

  If the result status is PGRES_TUPLES_OK, then the routines described below can be used to retrieve the tuples returned by the query. Note that a SELECT that happens to retrieve zero tuples still shows PGRES_TUPLES_OK. PGRES_COMMAND_OK is for commands that can never return tuples (INSERT, UPDATE, etc.). A response of PGRES_EMPTY_QUERY often exposes a bug in the client software.

- PQresStatus Converts the enumerated type returned by PQresultStatus into a string constant describing the status code.

  ```
  char *PQresStatus(ExecStatusType status);
  ```

- PQresultErrorMessage returns the error message associated with the query, or an empty string if there was no error.

  ```
  char *PQresultErrorMessage(const PGresult *res);
  ```

  Immediately following a PQexec or PQgetResult call, PQerrorMessage (on the connection) will return the same string as PQresultErrorMessage (on the result). However, a PGresult will retain its error message until destroyed, whereas the connection's error message will change when subsequent operations are done. Use PQresultErrorMessage when you want to know the status associated with a particular PGresult; use PQerrorMessage when you want to know the status from the latest operation on the connection.

- PQntuples Returns the number of tuples (instances) in the query result.

  ```
  int PQntuples(const PGresult *res);
  ```

- PQnfields Returns the number of fields (attributes) in each tuple of the query result.

  ```
  int PQnfields(const PGresult *res);
  ```

- PQbinaryTuples Returns 1 if the PGresult contains binary tuple data, 0 if it contains ASCII data.

  ```
  int PQbinaryTuples(const PGresult *res);
  ```

Currently, binary tuple data can only be returned by a query that extracts data from a BINARY cursor.

- `PQfname` Returns the field (attribute) name associated with the given field index. Field indices start at 0.

```
char *PQfname(const PGresult *res,
                    int field_index);
```

- `PQfnumber` Returns the field (attribute) index associated with the given field name.

```
int PQfnumber(const PGresult *res,
              const char *field_name);
```

-1 is returned if the given name does not match any field.

- `PQftype` Returns the field type associated with the given field index. The integer returned is an internal coding of the type. Field indices start at 0.

```
Oid PQftype(const PGresult *res,
            int field_num);
```

You can query the system table `pg_type` to obtain the name and properties of the various datatypes. The OIDs of the built-in datatypes are defined in `src/include/catalog/pg_type.h` in the source tree.

- `PQfsize` Returns the size in bytes of the field associated with the given field index. Field indices start at 0.

```
int PQfsize(const PGresult *res,
            int field_index);
```

PQfsize returns the space allocated for this field in a database tuple, in other words the size of the server's binary representation of the data type. -1 is returned if the field is variable size.

- `PQfmod` Returns the type-specific modification data of the field associated with the given field index. Field indices start at 0.

```
int PQfmod(const PGresult *res,
           int field_index);
```

- `PQgetvalue` Returns a single field (attribute) value of one tuple of a PGresult. Tuple and field indices start at 0.

```
char* PQgetvalue(const PGresult *res,
                 int tup_num,
                 int field_num);
```

For most queries, the value returned by `PQgetvalue` is a null-terminated ASCII string representation of the attribute value. But if `PQbinaryTuples()` is 1, the value returned by `PQgetvalue` is the binary representation of the type in the internal format of the backend server (but not including the size word, if the field is variable-length). It is then the programmer's responsibility to cast and convert the data to the correct C type. The pointer returned by `PQgetvalue` points to storage that is part of the PGresult structure. One should not modify it, and one must explicitly copy the value into other storage if it is to be used past the lifetime of the PGresult structure itself.

- `PQgetlength` Returns the length of a field (attribute) in bytes. Tuple and field indices start at 0.

```
int PQgetlength(const PGresult *res,
```

```
                int tup_num,
                int field_num);
```

This is the actual data length for the particular data value, that is the size of the object pointed to by PQgetvalue. Note that for ASCII-represented values, this size has little to do with the binary size reported by PQfsize.

- `PQgetisnull` Tests a field for a NULL entry. Tuple and field indices start at 0.

  ```
  int PQgetisnull(const PGresult *res,
                  int tup_num,
                  int field_num);
  ```

  This function returns 1 if the field contains a NULL, 0 if it contains a non-null value. (Note that PQgetvalue will return an empty string, not a null pointer, for a NULL field.)

- `PQcmdStatus` Returns the command status string from the SQL command that generated the PGresult.

  ```
  char * PQcmdStatus(const PGresult *res);
  ```

- `PQcmdTuples` Returns the number of rows affected by the SQL command.

  ```
  char * PQcmdTuples(const PGresult *res);
  ```

  If the SQL command that generated the PGresult was INSERT, UPDATE or DELETE, this returns a string containing the number of rows affected. If the command was anything else, it returns the empty string.

- `PQoidValue` Returns the object id of the tuple inserted, if the SQL command was an INSERT. Otherwise, returns `InvalidOid`.

  ```
  Oid PQoidValue(const PGresult *res);
  ```

  The type `Oid` and the constant `InvalidOid` will be defined if you include the libpq header file. They will both be some integer type.

- `PQoidStatus` Returns a string with the object id of the tuple inserted, if the SQL command was an INSERT. Otherwise, returns an empty string.

  ```
  char * PQoidStatus(const PGresult *res);
  ```

  This function is deprecated in favor of `PQoidValue` and is not thread-safe.

- `PQprint` Prints out all the tuples and, optionally, the attribute names to the specified output stream.

  ```
  void PQprint(FILE* fout,      /* output stream */
               const PGresult *res,
               const PQprintOpt *po);

  struct {
      pqbool  header;      /* print output field headings and row
   count */
      pqbool  align;       /* fill align the fields */
      pqbool  standard;    /* old brain dead format */
      pqbool  html3;       /* output html tables */
      pqbool  expanded;    /* expand tables */
      pqbool  pager;       /* use pager for output if needed */
      char    *fieldSep;   /* field separator */
      char    *tableOpt;   /* insert to HTML table ... */
      char    *caption;    /* HTML caption */
  ```

```
    char    **fieldName; /* null terminated array of replacement
 field names */
} PQprintOpt;
```

This function was formerly used by psql to print query results, but this is no longer the case and this function is no longer actively supported.

- `PQclear` Frees the storage associated with the PGresult. Every query result should be freed via PQclear when it is no longer needed.

  ```
  void PQclear(PQresult *res);
  ```

  You can keep a PGresult object around for as long as you need it; it does not go away when you issue a new query, nor even if you close the connection. To get rid of it, you must call `PQclear`. Failure to do this will result in memory leaks in the frontend application.

- `PQmakeEmptyPGresult` Constructs an empty PGresult object with the given status.

  ```
  PGresult* PQmakeEmptyPGresult(PGconn *conn, ExecStatusType
   status);
  ```

  This is libpq's internal routine to allocate and initialize an empty PGresult object. It is exported because some applications find it useful to generate result objects (particularly objects with error status) themselves. If conn is not NULL and status indicates an error, the connection's current errorMessage is copied into the PGresult. Note that PQclear should eventually be called on the object, just as with a PGresult returned by libpq itself.

54.3. Asynchronous Query Processing

The `PQexec` function is adequate for submitting queries in simple synchronous applications. It has a couple of major deficiencies however:

- `PQexec` waits for the query to be completed. The application may have other work to do (such as maintaining a user interface), in which case it won't want to block waiting for the response.
- Since control is buried inside `PQexec`, it is hard for the frontend to decide it would like to try to cancel the ongoing query. (It can be done from a signal handler, but not otherwise.)
- `PQexec` can return only one PGresult structure. If the submitted query string contains multiple SQL commands, all but the last PGresult are discarded by `PQexec`.

Applications that do not like these limitations can instead use the underlying functions that `PQexec` is built from: `PQsendQuery` and `PQgetResult`.

Older programs that used this functionality as well as `PQputline` and `PQputnbytes` could block waiting to send data to the backend, to address that issue, the function `PQsetnonblocking` was added.

Old applications can neglect to use `PQsetnonblocking` and get the older potentially blocking behavior. Newer programs can use `PQsetnonblocking` to achieve a completely non-blocking connection to the backend.

- `PQsetnonblocking` Sets the state of the connection to non-blocking.

  ```
  int PQsetnonblocking(PGconn *conn)
  ```

  this function will ensure that calls to `PQputline`, `PQputnbytes`, `PQsendQuery` and `PQendcopy` will not block but instead return an error if they need to be called again.

  When a database connection has been set to non-blocking mode and `PQexec` is called, it will temporarily set the state of the connection to blocking until the `PQexec` completes.

More of libpq is expected to be made safe for PQsetnonblocking functionality in the near future.

- PQisnonblocking Returns the blocking status of the database connection.

  ```
  int PQisnonblocking(const PGconn *conn)
  ```

  Returns TRUE if the connection is set to non-blocking mode, FALSE if blocking.

- PQsendQuery Submit a query to Postgres without waiting for the result(s). TRUE is returned if the query was successfully dispatched, FALSE if not (in which case, use PQerrorMessage to get more information about the failure).

  ```
  int PQsendQuery(PGconn *conn,
                  const char *query);
  ```

  After successfully calling PQsendQuery, call PQgetResult one or more times to obtain the query results. PQsendQuery may not be called again (on the same connection) until PQgetResult has returned NULL, indicating that the query is done.

- PQgetResult Wait for the next result from a prior PQsendQuery, and return it. NULL is returned when the query is complete and there will be no more results.

  ```
  PGresult *PQgetResult(PGconn *conn);
  ```

  PQgetResult must be called repeatedly until it returns NULL, indicating that the query is done. (If called when no query is active, PQgetResult will just return NULL at once.) Each non-null result from PQgetResult should be processed using the same PGresult accessor functions previously described. Don't forget to free each result object with PQclear when done with it. Note that PQgetResult will block only if a query is active and the necessary response data has not yet been read by PQconsumeInput.

Using PQsendQuery and PQgetResult solves one of PQexec's problems: If a query string contains multiple SQL commands, the results of those commands can be obtained individually. (This allows a simple form of overlapped processing, by the way: the frontend can be handling the results of one query while the backend is still working on later queries in the same query string.) However, calling PQgetResult will still cause the frontend to block until the backend completes the next SQL command. This can be avoided by proper use of three more functions:

- PQconsumeInput If input is available from the backend, consume it.

  ```
  int PQconsumeInput(PGconn *conn);
  ```

  PQconsumeInput normally returns 1 indicating "no error", but returns 0 if there was some kind of trouble (in which case PQerrorMessage is set). Note that the result does not say whether any input data was actually collected. After calling PQconsumeInput, the application may check PQisBusy and/or PQnotifies to see if their state has changed.

  PQconsumeInput may be called even if the application is not prepared to deal with a result or notification just yet. The routine will read available data and save it in a buffer, thereby causing a select(2) read-ready indication to go away. The application can thus use PQconsumeInput to clear the select condition immediately, and then examine the results at leisure.

- PQisBusy Returns 1 if a query is busy, that is, PQgetResult would block waiting for input. A 0 return indicates that PQgetResult can be called with assurance of not blocking.

  ```
  int PQisBusy(PGconn *conn);
  ```

  PQisBusy will not itself attempt to read data from the backend; therefore PQconsumeInput must be invoked first, or the busy state will never end.

- `PQflush` Attempt to flush any data queued to the backend, returns 0 if successful (or if the send queue is empty) or EOF if it failed for some reason.

  ```
  int PQflush(PGconn *conn);
  ```

  `PQflush` needs to be called on a non-blocking connection before calling `select` to determine if a responce has arrived. If 0 is returned it ensures that there is no data queued to the backend that has not actually been sent. Only applications that have used `PQsetnonblocking` have a need for this.

- `PQsocket` Obtain the file descriptor number for the backend connection socket. A valid descriptor will be >= 0; a result of -1 indicates that no backend connection is currently open.

  ```
  int PQsocket(const PGconn *conn);
  ```

  `PQsocket` should be used to obtain the backend socket descriptor in preparation for executing `select`(2). This allows an application using a blocking connection to wait for either backend responses or other conditions. If the result of `select`(2) indicates that data can be read from the backend socket, then `PQconsumeInput` should be called to read the data; after which, `PQisBusy`, `PQgetResult`, and/or `PQnotifies` can be used to process the response.

  Non-blocking connections (that have used `PQsetnonblocking`) should not use `select` until `PQflush` has returned 0 indicating that there is no buffered data waiting to be sent to the backend.

A typical frontend using these functions will have a main loop that uses `select`(2) to wait for all the conditions that it must respond to. One of the conditions will be input available from the backend, which in `select`'s terms is readable data on the file descriptor identified by `PQsocket`. When the main loop detects input ready, it should call `PQconsumeInput` to read the input. It can then call `PQisBusy`, followed by `PQgetResult` if `PQisBusy` returns false (0). It can also call `PQnotifies` to detect NOTIFY messages (see "Asynchronous Notification", below).

A frontend that uses `PQsendQuery`/`PQgetResult` can also attempt to cancel a query that is still being processed by the backend.

- `PQrequestCancel` Request that Postgres abandon processing of the current query.

  ```
  int PQrequestCancel(PGconn *conn);
  ```

  The return value is 1 if the cancel request was successfully dispatched, 0 if not. (If not, `PQerrorMessage` tells why not.) Successful dispatch is no guarantee that the request will have any effect, however. Regardless of the return value of `PQrequestCancel`, the application must continue with the normal result-reading sequence using `PQgetResult`. If the cancellation is effective, the current query will terminate early and return an error result. If the cancellation fails (say, because the backend was already done processing the query), then there will be no visible result at all.

Note that if the current query is part of a transaction, cancellation will abort the whole transaction.

`PQrequestCancel` can safely be invoked from a signal handler. So, it is also possible to use it in conjunction with plain `PQexec`, if the decision to cancel can be made in a signal handler. For example, psql invokes `PQrequestCancel` from a SIGINT signal handler, thus allowing interactive cancellation of queries that it issues through `PQexec`. Note that `PQrequestCancel` will have no effect if the connection is not currently open or the backend is not currently processing a query.

54.4. Fast Path

Postgres provides a fast path interface to send function calls to the backend. This is a trapdoor into system internals and can be a potential security hole. Most users will not need this feature.

- `PQfn` Request execution of a backend function via the fast path interface.

  ```
  PGresult* PQfn(PGconn* conn,
  ```

```
                int fnid,
                int *result_buf,
                int *result_len,
                int result_is_int,
                const PQArgBlock *args,
                int nargs);
```

The fnid argument is the object identifier of the function to be executed. result_buf is the buffer in which to place the return value. The caller must have allocated sufficient space to store the return value (there is no check!). The actual result length will be returned in the integer pointed to by result_len. If a 4-byte integer result is expected, set result_is_int to 1; otherwise set it to 0. (Setting result_is_int to 1 tells libpq to byte-swap the value if necessary, so that it is delivered as a proper int value for the client machine. When result_is_int is 0, the byte string sent by the backend is returned unmodified.) args and nargs specify the arguments to be passed to the function.

```
typedef struct {
    int len;
    int isint;
    union {
        int *ptr;
        int integer;
    } u;
} PQArgBlock;
```

`PQfn` always returns a valid PGresult*. The resultStatus should be checked before the result is used. The caller is responsible for freeing the PGresult with `PQclear` when it is no longer needed.

54.5. Asynchronous Notification

Postgres supports asynchronous notification via the LISTEN and NOTIFY commands. A backend registers its interest in a particular notification condition with the LISTEN command (and can stop listening with the UNLISTEN command). All backends listening on a particular condition will be notified asynchronously when a NOTIFY of that condition name is executed by any backend. No additional information is passed from the notifier to the listener. Thus, typically, any actual data that needs to be communicated is transferred through a database relation. Commonly the condition name is the same as the associated relation, but it is not necessary for there to be any associated relation.

`libpq` applications submit LISTEN and UNLISTEN commands as ordinary SQL queries. Subsequently, arrival of NOTIFY messages can be detected by calling PQnotifies().

- `PQnotifies` Returns the next notification from a list of unhandled notification messages received from the backend. Returns NULL if there are no pending notifications. Once a notification is returned from PQnotifies, it is considered handled and will be removed from the list of notifications.

  ```
  PGnotify* PQnotifies(PGconn *conn);

  typedef struct pgNotify {
      char relname[NAMEDATALEN];       /* name of relation
                                        * containing data */
      int  be_pid;                     /* process id of backend */
  } PGnotify;
  ```

  After processing a PGnotify object returned by `PQnotifies`, be sure to free it with `free()` to avoid a memory leak.

  ### Note

  In Postgres 6.4 and later, the `be_pid` is the notifying backend's, whereas in earlier versions it was always your own backend's PID.

The second sample program gives an example of the use of asynchronous notification.

`PQnotifies()` does not actually read backend data; it just returns messages previously absorbed by another libpq function. In prior releases of libpq, the only way to ensure timely receipt of NOTIFY messages was to constantly submit queries, even empty ones, and then check `PQnotifies()` after each `PQexec()`. While this still works, it is deprecated as a waste of processing power.

A better way to check for NOTIFY messages when you have no useful queries to make is to call `PQconsumeInput()`, then check `PQnotifies()`. You can use `select`(2) to wait for backend data to arrive, thereby using no CPU power unless there is something to do. (See `PQsocket()` to obtain the file descriptor number to use with `select`.) Note that this will work OK whether you submit queries with `PQsendQuery`/`PQgetResult` or simply use `PQexec`. You should, however, remember to check `PQnotifies()` after each `PQgetResult` or `PQexec`, to see if any notifications came in during the processing of the query.

54.6. Functions Associated with the COPY Command

The COPY command in Postgres has options to read from or write to the network connection used by `libpq`. Therefore, functions are necessary to access this network connection directly so applications may take advantage of this capability.

These functions should be executed only after obtaining a `PGRES_COPY_OUT` or `PGRES_COPY_IN` result object from `PQexec` or `PQgetResult`.

- `PQgetline` Reads a newline-terminated line of characters (transmitted by the backend server) into a buffer string of size length.

  ```
  int PQgetline(PGconn *conn,
                char *string,
                int length)
  ```

  Like `fgets`(3), this routine copies up to length-1 characters into string. It is like `gets`(3), however, in that it converts the terminating newline into a null character. `PQgetline` returns `EOF` at EOF, 0 if the entire line has been read, and 1 if the buffer is full but the terminating newline has not yet been read.

  Notice that the application must check to see if a new line consists of the two characters "\.", which indicates that the backend server has finished sending the results of the copy command. If the application might receive lines that are more than length-1 characters long, care is needed to be sure one recognizes the "\." line correctly (and does not, for example, mistake the end of a long data line for a terminator line). The code in `src/bin/psql/copy.c` contains example routines that correctly handle the copy protocol.

- `PQgetlineAsync` Reads a newline-terminated line of characters (transmitted by the backend server) into a buffer without blocking.

  ```
  int PQgetlineAsync(PGconn *conn,
                     char *buffer,
                     int bufsize)
  ```

  This routine is similar to `PQgetline`, but it can be used by applications that must read COPY data asynchronously, that is without blocking. Having issued the COPY command and gotten a `PGRES_COPY_OUT` response, the application should call `PQconsumeInput` and `PQgetlineAsync` until the end-of-data signal is detected. Unlike `PQgetline`, this routine takes responsibility for detecting end-of-data. On each call, `PQgetlineAsync` will return data if a complete newline- terminated data line is available in libpq's input buffer, or if the incoming data

line is too long to fit in the buffer offered by the caller. Otherwise, no data is returned until the rest of the line arrives.

The routine returns -1 if the end-of-copy-data marker has been recognized, or 0 if no data is available, or a positive number giving the number of bytes of data returned. If -1 is returned, the caller must next call `PQendcopy`, and then return to normal processing. The data returned will not extend beyond a newline character. If possible a whole line will be returned at one time. But if the buffer offered by the caller is too small to hold a line sent by the backend, then a partial data line will be returned. This can be detected by testing whether the last returned byte is "`\n`" or not. The returned string is not null-terminated. (If you want to add a terminating null, be sure to pass a bufsize one smaller than the room actually available.)

- `PQputline` Sends a null-terminated string to the backend server. Returns 0 if OK, `EOF` if unable to send the string.

  ```
  int PQputline(PGconn *conn,
                const char *string);
  ```

  Note the application must explicitly send the two characters "`\.`" on a final line to indicate to the backend that it has finished sending its data.

- `PQputnbytes` Sends a non-null-terminated string to the backend server. Returns 0 if OK, EOF if unable to send the string.

  ```
  int PQputnbytes(PGconn *conn,
                  const char *buffer,
                  int nbytes);
  ```

  This is exactly like `PQputline`, except that the data buffer need not be null-terminated since the number of bytes to send is specified directly.

- `PQendcopy` Syncs with the backend. This function waits until the backend has finished the copy. It should either be issued when the last string has been sent to the backend using `PQputline` or when the last string has been received from the backend using `PGgetline`. It must be issued or the backend may get "out of sync" with the frontend. Upon return from this function, the backend is ready to receive the next query. The return value is 0 on successful completion, nonzero otherwise.

  ```
  int PQendcopy(PGconn *conn);
  ```

  As an example:

  ```
  PQexec(conn, "create table foo (a int4, b char(16), d float8)");
  PQexec(conn, "copy foo from stdin");
  PQputline(conn, "3\thello world\t4.5\n");
  PQputline(conn,"4\tgoodbye world\t7.11\n");
  ...
  PQputline(conn,"\\.\n");
  PQendcopy(conn);
  ```

When using `PQgetResult`, the application should respond to a `PGRES_COPY_OUT` result by executing `PQgetline` repeatedly, followed by `PQendcopy` after the terminator line is seen. It should then return to the `PQgetResult` loop until `PQgetResult` returns NULL. Similarly a `PGRES_COPY_IN` result is processed by a series of `PQputline` calls followed by `PQendcopy`, then return to the `PQgetResult` loop. This arrangement will ensure that a copy in or copy out command embedded in a series of SQL commands will be executed correctly.

Older applications are likely to submit a copy in or copy out via `PQexec` and assume that the transaction is done after `PQendcopy`. This will work correctly only if the copy in/out is the only SQL command in the query string.

54.7. libpq Tracing Functions

- `PQtrace` Enable tracing of the frontend/backend communication to a debugging file stream.

```
void PQtrace(PGconn *conn
             FILE *debug_port)
```

- `PQuntrace` Disable tracing started by PQtrace

```
void PQuntrace(PGconn *conn)
```

54.8. libpq Control Functions

- `PQsetNoticeProcessor` Control reporting of notice and warning messages generated by libpq.

```
typedef void (*PQnoticeProcessor) (void *arg, const char
 *message);

PQnoticeProcessor
PQsetNoticeProcessor(PGconn *conn,
                     PQnoticeProcessor proc,
                     void *arg);
```

By default, libpq prints "notice" messages from the backend on `stderr`, as well as a few error messages that it generates by itself. This behavior can be overridden by supplying a callback function that does something else with the messages. The callback function is passed the text of the error message (which includes a trailing newline), plus a void pointer that is the same one passed to `PQsetNoticeProcessor`. (This pointer can be used to access application-specific state if needed.) The default notice processor is simply

```
static void
defaultNoticeProcessor(void * arg, const char * message)
{
    fprintf(stderr, "%s", message);
}
```

To use a special notice processor, call `PQsetNoticeProcessor` just after creation of a new PGconn object.

The return value is the pointer to the previous notice processor. If you supply a callback function pointer of NULL, no action is taken, but the current pointer is returned.

Once you have set a notice processor, you should expect that that function could be called as long as either the PGconn object or PGresult objects made from it exist. At creation of a PGresult, the PGconn's current notice processor pointer is copied into the PGresult for possible use by routines like `PQgetvalue`.

54.9. Environment Variables

The following environment variables can be used to select default connection parameter values, which will be used by `PQconnectdb` or `PQsetdbLogin` if no value is directly specified by the calling code. These are useful to avoid hard-coding database names into simple application programs.

- `PGHOST` sets the default server name. If a non-zero-length string is specified, TCP/IP communication is used. Without a host name, libpq will connect using a local Unix domain socket.

- `PGPORT` sets the default port or local Unix domain socket file extension for communicating with the Postgres backend.

- PGDATABASE sets the default Postgres database name.
- PGUSER sets the username used to connect to the database and for authentication.
- PGPASSWORD sets the password used if the backend demands password authentication.
- PGREALM sets the Kerberos realm to use with Postgres, if it is different from the local realm. If PGREALM is set, Postgres applications will attempt authentication with servers for this realm and use separate ticket files to avoid conflicts with local ticket files. This environment variable is only used if Kerberos authentication is selected by the backend.
- PGOPTIONS sets additional runtime options for the Postgres backend.
- PGTTY sets the file or tty on which debugging messages from the backend server are displayed.

The following environment variables can be used to specify user-level default behavior for every Postgres session:

- PGDATESTYLE sets the default style of date/time representation.
- PGTZ sets the default time zone.
- PGCLIENTENCODING sets the default client encoding (if MULTIBYTE support was selected when configuring Postgres).

The following environment variables can be used to specify default internal behavior for every Postgres session:

- PGGEQO sets the default mode for the genetic optimizer.

Refer to the SET SQL command for information on correct values for these environment variables.

54.10. Threading Behavior

libpq is thread-safe as of Postgres 7.0, so long as no two threads attempt to manipulate the same PGconn object at the same time. In particular, you can't issue concurrent queries from different threads through the same connection object. (If you need to run concurrent queries, start up multiple connections.)

PGresult objects are read-only after creation, and so can be passed around freely between threads.

The deprecated functions PQoidStatus and fe_setauthsvc are not thread-safe and should not be used in multi-thread programs. PQoidStatus can be replaced by PQoidValue. There is no good reason to call fe_setauthsvc at all.

54.11. Sample Programs

54.11.1. Sample Program 1

```
/*
 * testlibpq.c Test the C version of Libpq, the Postgres frontend
 * library.
 *
 *
 */
#include <stdio.h>
#include "libpq-fe.h"

void
exit_nicely(PGconn *conn)
{
```

```
    PQfinish(conn);
    exit(1);
}

main()
{
    char       *pghost,
               *pgport,
               *pgoptions,
               *pgtty;
    char       *dbName;
    int         nFields;
    int         i,
                j;

    /* FILE *debug; */

    PGconn     *conn;
    PGresult   *res;

    /*
     * begin, by setting the parameters for a backend connection if
 the
     * parameters are null, then the system will try to use
 reasonable
     * defaults by looking up environment variables or, failing
 that,
     * using hardwired constants
     */
    pghost = NULL;              /* host name of the backend server
 */
    pgport = NULL;              /* port of the backend server */
    pgoptions = NULL;           /* special options to start up the
 backend
                                 * server */
    pgtty = NULL;               /* debugging tty for the backend
 server */
    dbName = "template1";

    /* make a connection to the database */
    conn = PQsetdb(pghost, pgport, pgoptions, pgtty, dbName);

    /*
     * check to see that the backend connection was successfully
 made
     */
    if (PQstatus(conn) == CONNECTION_BAD)
    {
        fprintf(stderr, "Connection to database '%s' failed.\n",
 dbName);
        fprintf(stderr, "%s", PQerrorMessage(conn));
        exit_nicely(conn);
    }

    /* debug = fopen("/tmp/trace.out","w"); */
    /* PQtrace(conn, debug);  */

    /* start a transaction block */
```

```
    res = PQexec(conn, "BEGIN");
    if (!res || PQresultStatus(res) != PGRES_COMMAND_OK)
    {
        fprintf(stderr, "BEGIN command failed\n");
        PQclear(res);
        exit_nicely(conn);
    }

    /*
     * should PQclear PGresult whenever it is no longer needed to
avoid
     * memory leaks
     */
    PQclear(res);

    /*
     * fetch instances from the pg_database, the system catalog of
     * databases
     */
    res = PQexec(conn, "DECLARE mycursor CURSOR FOR select * from
pg_database");
    if (!res || PQresultStatus(res) != PGRES_COMMAND_OK)
    {
        fprintf(stderr, "DECLARE CURSOR command failed\n");
        PQclear(res);
        exit_nicely(conn);
    }
    PQclear(res);
    res = PQexec(conn, "FETCH ALL in mycursor");
    if (!res || PQresultStatus(res) != PGRES_TUPLES_OK)
    {
        fprintf(stderr, "FETCH ALL command didn't return tuples
properly\n");
        PQclear(res);
        exit_nicely(conn);
    }

    /* first, print out the attribute names */
    nFields = PQnfields(res);
    for (i = 0; i < nFields; i++)
        printf("%-15s", PQfname(res, i));
    printf("\n\n");

    /* next, print out the instances */
    for (i = 0; i < PQntuples(res); i++)
    {
        for (j = 0; j < nFields; j++)
            printf("%-15s", PQgetvalue(res, i, j));
        printf("\n");
    }
    PQclear(res);

    /* close the cursor */
    res = PQexec(conn, "CLOSE mycursor");
    PQclear(res);

    /* commit the transaction */
    res = PQexec(conn, "COMMIT");
```

```
    PQclear(res);

    /* close the connection to the database and cleanup */
    PQfinish(conn);

    /* fclose(debug); */
    return 0;

}
```

54.11.2. Sample Program 2

```
/*
 * testlibpq2.c
 *  Test of the asynchronous notification interface
 *
 * Start this program, then from psql in another window do
 *   NOTIFY TBL2;
 *
 * Or, if you want to get fancy, try this:
 * Populate a database with the following:
 *
 *   CREATE TABLE TBL1 (i int4);
 *
 *   CREATE TABLE TBL2 (i int4);
 *
 *   CREATE RULE r1 AS ON INSERT TO TBL1 DO
 *     (INSERT INTO TBL2 values (new.i); NOTIFY TBL2);
 *
 * and do
 *
 *   INSERT INTO TBL1 values (10);
 *
 */
#include <stdio.h>
#include "libpq-fe.h"

void
exit_nicely(PGconn *conn)
{
    PQfinish(conn);
    exit(1);
}

main()
{
    char       *pghost,
               *pgport,
               *pgoptions,
               *pgtty;
    char       *dbName;
    int         nFields;
    int         i,
                j;

    PGconn     *conn;
    PGresult   *res;
    PGnotify   *notify;
```

```
   /*
    * begin, by setting the parameters for a backend connection if
the
    * parameters are null, then the system will try to use
reasonable
    * defaults by looking up environment variables or, failing
that,
    * using hardwired constants
    */
   pghost = NULL;              /* host name of the backend server
*/
   pgport = NULL;              /* port of the backend server */
   pgoptions = NULL;           /* special options to start up the
backend
                                * server */
   pgtty = NULL;               /* debugging tty for the backend
server */
   dbName = getenv("USER");    /* change this to the name of your
test
                                * database */

   /* make a connection to the database */
   conn = PQsetdb(pghost, pgport, pgoptions, pgtty, dbName);

   /*
    * check to see that the backend connection was successfully
made
    */
   if (PQstatus(conn) == CONNECTION_BAD)
   {
       fprintf(stderr, "Connection to database '%s' failed.\n",
dbName);
       fprintf(stderr, "%s", PQerrorMessage(conn));
       exit_nicely(conn);
   }

   res = PQexec(conn, "LISTEN TBL2");
   if (!res || PQresultStatus(res) != PGRES_COMMAND_OK)
   {
       fprintf(stderr, "LISTEN command failed\n");
       PQclear(res);
       exit_nicely(conn);
   }

   /*
    * should PQclear PGresult whenever it is no longer needed to
avoid
    * memory leaks
    */
   PQclear(res);

   while (1)
   {

       /*
        * wait a little bit between checks; waiting with select()
        * would be more efficient.
```

```
         */
        sleep(1);
        /* collect any asynchronous backend messages */
        PQconsumeInput(conn);
        /* check for asynchronous notify messages */
        while ((notify = PQnotifies(conn)) != NULL)
        {
            fprintf(stderr,
                 "ASYNC NOTIFY of '%s' from backend pid '%d'
 received\n",
                    notify->relname, notify->be_pid);
            free(notify);
        }
    }

    /* close the connection to the database and cleanup */
    PQfinish(conn);

    return 0;
}
```

54.11.3. Sample Program 3

```
/*
 * testlibpq3.c Test the C version of Libpq, the Postgres frontend
 * library. tests the binary cursor interface
 *
 *
 *
 * populate a database by doing the following:
 *
 * CREATE TABLE test1 (i int4, d float4, p polygon);
 *
 * INSERT INTO test1 values (1, 3.567, '(3.0, 4.0, 1.0,
 * 2.0)'::polygon);
 *
 * INSERT INTO test1 values (2, 89.05, '(4.0, 3.0, 2.0,
 * 1.0)'::polygon);
 *
 * the expected output is:
 *
 * tuple 0: got i = (4 bytes) 1, d = (4 bytes) 3.567000, p = (4
 * bytes) 2 points   boundbox = (hi=3.000000/4.000000, lo =
 * 1.000000,2.000000) tuple 1: got i = (4 bytes) 2, d = (4 bytes)
 * 89.050003, p = (4 bytes) 2 points   boundbox =
 * (hi=4.000000/3.000000, lo = 2.000000,1.000000)
 *
 *
 */
#include <stdio.h>
#include "libpq-fe.h"
#include "utils/geo-decls.h"    /* for the POLYGON type */

void
exit_nicely(PGconn *conn)
{
    PQfinish(conn);
    exit(1);
```

```
}

main()
{
    char       *pghost,
               *pgport,
               *pgoptions,
               *pgtty;
    char       *dbName;
    int         nFields;
    int         i,
                j;
    int         i_fnum,
                d_fnum,
                p_fnum;
    PGconn     *conn;
    PGresult   *res;

    /*
     * begin, by setting the parameters for a backend connection if
 the
     * parameters are null, then the system will try to use
 reasonable
     * defaults by looking up environment variables or, failing
 that,
     * using hardwired constants
     */
    pghost = NULL;              /* host name of the backend server
 */
    pgport = NULL;              /* port of the backend server */
    pgoptions = NULL;           /* special options to start up the
 backend
                                 * server */
    pgtty = NULL;               /* debugging tty for the backend
 server */

    dbName = getenv("USER");    /* change this to the name of your
 test
                                 * database */

    /* make a connection to the database */
    conn = PQsetdb(pghost, pgport, pgoptions, pgtty, dbName);

    /*
     * check to see that the backend connection was successfully
 made
     */
    if (PQstatus(conn) == CONNECTION_BAD)
    {
        fprintf(stderr, "Connection to database '%s' failed.\n",
 dbName);
        fprintf(stderr, "%s", PQerrorMessage(conn));
        exit_nicely(conn);
    }

    /* start a transaction block */
    res = PQexec(conn, "BEGIN");
    if (!res || PQresultStatus(res) != PGRES_COMMAND_OK)
```

```
    {
        fprintf(stderr, "BEGIN command failed\n");
        PQclear(res);
        exit_nicely(conn);
    }

    /*
     * should PQclear PGresult whenever it is no longer needed to
avoid
     * memory leaks
     */
    PQclear(res);

    /*
     * fetch instances from the pg_database, the system catalog of
     * databases
     */
    res = PQexec(conn, "DECLARE mycursor BINARY CURSOR FOR select *
from test1");
    if (!res || PQresultStatus(res) != PGRES_COMMAND_OK)
    {
        fprintf(stderr, "DECLARE CURSOR command failed\n");
        PQclear(res);
        exit_nicely(conn);
    }
    PQclear(res);

    res = PQexec(conn, "FETCH ALL in mycursor");
    if (!res || PQresultStatus(res) != PGRES_TUPLES_OK)
    {
        fprintf(stderr, "FETCH ALL command didn't return tuples
properly\n");
        PQclear(res);
        exit_nicely(conn);
    }

    i_fnum = PQfnumber(res, "i");
    d_fnum = PQfnumber(res, "d");
    p_fnum = PQfnumber(res, "p");

    for (i = 0; i < 3; i++)
    {
        printf("type[%d] = %d, size[%d] = %d\n",
               i, PQftype(res, i),
               i, PQfsize(res, i));
    }
    for (i = 0; i < PQntuples(res); i++)
    {
        int        *ival;
        float      *dval;
        int         plen;
        POLYGON    *pval;

        /* we hard-wire this to the 3 fields we know about */
        ival = (int *) PQgetvalue(res, i, i_fnum);
        dval = (float *) PQgetvalue(res, i, d_fnum);
        plen = PQgetlength(res, i, p_fnum);
```

```
        /*
         * plen doesn't include the length field so need to
         * increment by VARHDSZ
         */
        pval = (POLYGON *) malloc(plen + VARHDRSZ);
        pval->size = plen;
        memmove((char *) &pval->npts, PQgetvalue(res, i, p_fnum),
 plen);
        printf("tuple %d: got\n", i);
        printf(" i = (%d bytes) %d,\n",
               PQgetlength(res, i, i_fnum), *ival);
        printf(" d = (%d bytes) %f,\n",
               PQgetlength(res, i, d_fnum), *dval);
        printf(" p = (%d bytes) %d points \tboundbox = (hi=%f/%f,
 lo = %f,%f)\n",
               PQgetlength(res, i, d_fnum),
               pval->npts,
               pval->boundbox.xh,
               pval->boundbox.yh,
               pval->boundbox.xl,
               pval->boundbox.yl);
    }
    PQclear(res);

    /* close the cursor */
    res = PQexec(conn, "CLOSE mycursor");
    PQclear(res);

    /* commit the transaction */
    res = PQexec(conn, "COMMIT");
    PQclear(res);

    /* close the connection to the database and cleanup */
    PQfinish(conn);

    return 0;
}
```

Chapter 55. libpq - C++ Binding Library

libpq++ is the C++ API to Postgres. libpq++ is a set of classes which allow client programs to connect to the Postgres backend server. These connections come in two forms: a Database Class and a Large Object class.

The Database Class is intended for manipulating a database. You can send all sorts of SQL queries to the Postgres backend server and retrieve the responses of the server.

The Large Object Class is intended for manipulating a large object in a database. Although a Large Object instance can send normal queries to the Postgres backend server it is only intended for simple queries that do not return any data. A large object should be seen as a file stream. In the future it should behave much like the C++ file streams cin, cout and cerr.

This chapter is based on the documentation for the libpq C library. Three short programs are listed at the end of this section as examples of libpq++ programming (though not necessarily of good programming). There are several examples of libpq++ applications in src/libpq++/examples, including the source code for the three examples in this chapter.

55.1. Control and Initialization

55.1.1. Environment Variables

The following environment variables can be used to set up default values for an environment and to avoid hard-coding database names into an application program:

Note

Refer to the libpq - C Library for a complete list of available connection options.

The following environment variables can be used to select default connection parameter values, which will be used by PQconnectdb or PQsetdbLogin if no value is directly specified by the calling code. These are useful to avoid hard-coding database names into simple application programs.

Note

libpq++ uses only environment variables or PQconnectdb conninfo style strings.

- PGHOST sets the default server name. If a non-zero-length string is specified, TCP/IP communication is used. Without a host name, libpq will connect using a local Unix domain socket.
- PGPORT sets the default port or local Unix domain socket file extension for communicating with the Postgres backend.
- PGDATABASE sets the default Postgres database name.
- PGUSER sets the username used to connect to the database and for authentication.
- PGPASSWORD sets the password used if the backend demands password authentication.
- PGREALM sets the Kerberos realm to use with Postgres, if it is different from the local realm. If PGREALM is set, Postgres applications will attempt authentication with servers for this realm and use separate ticket files to avoid conflicts with local ticket files. This environment variable is only used if Kerberos authentication is selected by the backend.

- `PGOPTIONS` sets additional runtime options for the Postgres backend.
- `PGTTY` sets the file or tty on which debugging messages from the backend server are displayed.

The following environment variables can be used to specify user-level default behavior for every Postgres session:

- `PGDATESTYLE` sets the default style of date/time representation.
- `PGTZ` sets the default time zone.

The following environment variables can be used to specify default internal behavior for every Postgres session:

- `PGGEQO` sets the default mode for the genetic optimizer.

Refer to the `SET` SQL command for information on correct values for these environment variables.

55.2. libpq++ Classes

55.2.1. Connection Class: `PgConnection`

The connection class makes the actual connection to the database and is inherited by all of the access classes.

55.2.2. Database Class: `PgDatabase`

The database class provides C++ objects that have a connection to a backend server. To create such an object one first needs the apropriate environment for the backend to access. The following constructors deal with making a connection to a backend server from a C++ program.

55.3. Database Connection Functions

- `PgConnection` makes a new connection to a backend database server.

  ```
  PgConnection::PgConnection(const char *conninfo)
  ```

  Although typically called from one of the access classes, a connection to a backend server is possible by creating a PgConnection object.

- `ConnectionBad` returns whether or not the connection to the backend server succeeded or failed.

  ```
  int PgConnection::ConnectionBad()
  ```

  Returns TRUE if the connection failed.

- `Status` returns the status of the connection to the backend server.

  ```
  ConnStatusType PgConnection::Status()
  ```

  Returns either CONNECTION_OK or CONNECTION_BAD depending on the state of the connection.

- `PgDatabase` makes a new connection to a backend database server.

  ```
  PgDatabase(const char *conninfo)
  ```

After a PgDatabase has been created it should be checked to make sure the connection to the database succeded before sending queries to the object. This can easily be done by retrieving the current status of the PgDatabase object with the `Status` or `ConnectionBad` methods.

- `DBName` Returns the name of the current database.

```
const char *PgConnection::DBName()
```

- `Notifies` Returns the next notification from a list of unhandled notification messages received from the backend.

```
PGnotify* PgConnection::Notifies()
```

See PQnotifies() for details.

55.4. Query Execution Functions

- `Exec` Sends a query to the backend server. It's probably more desirable to use one of the next two functions.

```
ExecStatusType PgConnection::Exec(const char* query)
```

Returns the result of the query. The following status results can be expected:

PGRES_EMPTY_QUERY
PGRES_COMMAND_OK, if the query was a command
PGRES_TUPLES_OK, if the query successfully returned tuples
PGRES_COPY_OUT
PGRES_COPY_IN
PGRES_BAD_RESPONSE, if an unexpected response was received
PGRES_NONFATAL_ERROR
PGRES_FATAL_ERROR

- `ExecCommandOk` Sends a command query to the backend server.

```
int PgConnection::ExecCommandOk(const char *query)
```

Returns TRUE if the command query succeeds.

- `ExecTuplesOk` Sends a command query to the backend server.

```
int PgConnection::ExecTuplesOk(const char *query)
```

Returns TRUE if the command query succeeds.

- `ErrorMessage` Returns the last error message text.

```
const char *PgConnection::ErrorMessage()
```

- `Tuples` Returns the number of tuples (instances) in the query result.

```
int PgDatabase::Tuples()
```

- `CmdTuples` Returns the number of rows affected after an INSERT, UPDATE or DELETE. If the command was anything else, it returns -1.

```
int PgDatabase::CmdTuples()
```

- `Fields` Returns the number of fields (attributes) in each tuple of the query result.

```
int PgDatabase::Fields()
```

- `FieldName` Returns the field (attribute) name associated with the given field index. Field indices start at 0.

```
const char *PgDatabase::FieldName(int field_num)
```

- `FieldNum` PQfnumber Returns the field (attribute) index associated with the given field name.

```
int PgDatabase::FieldNum(const char* field_name)
```

  -1 is returned if the given name does not match any field.

- `FieldType` Returns the field type associated with the given field index. The integer returned is an internal coding of the type. Field indices start at 0.

```
Oid PgDatabase::FieldType(int field_num)
```

- `FieldType` Returns the field type associated with the given field name. The integer returned is an internal coding of the type. Field indices start at 0.

```
Oid PgDatabase::FieldType(const char* field_name)
```

- `FieldSize` Returns the size in bytes of the field associated with the given field index. Field indices start at 0.

```
short PgDatabase::FieldSize(int field_num)
```

  Returns the space allocated for this field in a database tuple given the field number. In other words the size of the server's binary representation of the data type. -1 is returned if the field is variable size.

- `FieldSize` Returns the size in bytes of the field associated with the given field index. Field indices start at 0.

```
short PgDatabase::FieldSize(const char *field_name)
```

  Returns the space allocated for this field in a database tuple given the field name. In other words the size of the server's binary representation of the data type. -1 is returned if the field is variable size.

- `GetValue` Returns a single field (attribute) value of one tuple of a PGresult. Tuple and field indices start at 0.

```
const char *PgDatabase::GetValue(int tup_num, int field_num)
```

  For most queries, the value returned by GetValue is a null-terminated ASCII string representation of the attribute value. But if BinaryTuples() is TRUE, the value returned by GetValue is the binary

representation of the type in the internal format of the backend server (but not including the size word, if the field is variable-length). It is then the programmer's responsibility to cast and convert the data to the correct C type. The pointer returned by GetValue points to storage that is part of the PGresult structure. One should not modify it, and one must explicitly copy the value into other storage if it is to be used past the lifetime of the PGresult structure itself. BinaryTuples() is not yet implemented.

- `GetValue` Returns a single field (attribute) value of one tuple of a PGresult. Tuple and field indices start at 0.

```
const char *PgDatabase::GetValue(int tup_num, const char
*field_name)
```

For most queries, the value returned by GetValue is a null-terminated ASCII string representation of the attribute value. But if BinaryTuples() is TRUE, the value returned by GetValue is the binary representation of the type in the internal format of the backend server (but not including the size word, if the field is variable-length). It is then the programmer's responsibility to cast and convert the data to the correct C type. The pointer returned by GetValue points to storage that is part of the PGresult structure. One should not modify it, and one must explicitly copy the value into other storage if it is to be used past the lifetime of the PGresult structure itself. BinaryTuples() is not yet implemented.

- `GetLength` Returns the length of a field (attribute) in bytes. Tuple and field indices start at 0.

```
int PgDatabase::GetLength(int tup_num, int field_num)
```

This is the actual data length for the particular data value, that is the size of the object pointed to by GetValue. Note that for ASCII-represented values, this size has little to do with the binary size reported by PQfsize.

- `GetLength` Returns the length of a field (attribute) in bytes. Tuple and field indices start at 0.

```
int PgDatabase::GetLength(int tup_num, const char* field_name)
```

This is the actual data length for the particular data value, that is the size of the object pointed to by GetValue. Note that for ASCII-represented values, this size has little to do with the binary size reported by PQfsize.

- `DisplayTuples` Prints out all the tuples and, optionally, the attribute names to the specified output stream.

```
void PgDatabase::DisplayTuples(FILE *out = 0, int fillAlign =
1,
const char* fieldSep = "|",int printHeader = 1, int quiet = 0)
```

- `PrintTuples` Prints out all the tuples and, optionally, the attribute names to the specified output stream.

```
void PgDatabase::PrintTuples(FILE *out = 0, int printAttName =
1,
int terseOutput = 0, int width = 0)
```

- `GetLine`

```
int PgDatabase::GetLine(char* string, int length)
```

- PutLine

```
void PgDatabase::PutLine(const char* string)
```

- OidStatus

```
const char *PgDatabase::OidStatus()
```

- EndCopy

```
int PgDatabase::EndCopy()
```

55.5. Asynchronous Notification

Postgres supports asynchronous notification via the LISTEN and NOTIFY commands. A backend registers its interest in a particular semaphore with the LISTEN command. All backends that are listening on a particular named semaphore will be notified asynchronously when a NOTIFY of that name is executed by another backend. No additional information is passed from the notifier to the listener. Thus, typically, any actual data that needs to be communicated is transferred through the relation.

> **Note**
>
> In the past, the documentation has associated the names used for asyncronous notification with relations or classes. However, there is in fact no direct linkage of the two concepts in the implementation, and the named semaphore in fact does not need to have a corresponding relation previously defined.

libpq++ applications are notified whenever a connected backend has received an asynchronous notification. However, the communication from the backend to the frontend is not asynchronous. The libpq++ application must poll the backend to see if there is any pending notification information. After the execution of a query, a frontend may call PgDatabase::Notifies to see if any notification data is currently available from the backend. PgDatabase::Notifies returns the notification from a list of unhandled notifications from the backend. The function eturns NULL if there is no pending notifications from the backend. PgDatabase::Notifies behaves like the popping of a stack. Once a notification is returned from PgDatabase::Notifies, it is considered handled and will be removed from the list of notifications.

- PgDatabase::Notifies retrieves pending notifications from the server.

```
PGnotify* PgDatabase::Notifies()
```

The second sample program gives an example of the use of asynchronous notification.

55.6. Functions Associated with the COPY Command

The copy command in Postgres has options to read from or write to the network connection used by libpq++. Therefore, functions are necessary to access this network connection directly so applications may take full advantage of this capability.

- PgDatabase::GetLine reads a newline-terminated line of characters (transmitted by the backend server) into a buffer *string* of size *length*.

```
int PgDatabase::GetLine(char* string, int length)
```

Like the Unix system routine `fgets (3)`, this routine copies up to *`length`*`-1` characters into *`string`*. It is like `gets (3)`, however, in that it converts the terminating newline into a null character.

`PgDatabase::GetLine` returns EOF at end of file, 0 if the entire line has been read, and 1 if the buffer is full but the terminating newline has not yet been read.

Notice that the application must check to see if a new line consists of a single period ("."), which indicates that the backend server has finished sending the results of the `copy`. Therefore, if the application ever expects to receive lines that are more than *`length`*`-1` characters long, the application must be sure to check the return value of `PgDatabase::GetLine` very carefully.

- `PgDatabase::PutLine` Sends a null-terminated *`string`* to the backend server.

```
void PgDatabase::PutLine(char* string)
```

The application must explicitly send a single period character (".") to indicate to the backend that it has finished sending its data.

- `PgDatabase::EndCopy` syncs with the backend.

```
int PgDatabase::EndCopy()
```

This function waits until the backend has finished processing the `copy`. It should either be issued when the last string has been sent to the backend using `PgDatabase::PutLine` or when the last string has been received from the backend using `PgDatabase::GetLine`. It must be issued or the backend may get "out of sync" with the frontend. Upon return from this function, the backend is ready to receive the next query.

The return value is 0 on successful completion, nonzero otherwise.

As an example:

```
PgDatabase data;
data.Exec("create table foo (a int4, b char(16), d float8)");
data.Exec("copy foo from stdin");
data.PutLine("3\tHello World\t4.5\n");
data.PutLine("4\tGoodbye World\t7.11\n");
&...
data.PutLine("\\.\n");
data.EndCopy();
```

Chapter 56. pgtcl - TCL Binding Library

`pgtcl` is a tcl package for front-end programs to interface with Postgres backends. It makes most of the functionality of `libpq` available to tcl scripts.

This package was originally written by Jolly Chen.

56.1. Commands

Table 56.1. `pgtcl` Commands

Command	Description
pg_connect	opens a connection to the backend server
pg_disconnect	closes a connection
pg_conndefaults	get connection options and their defaults
pg_exec	send a query to the backend
pg_result	manipulate the results of a query
pg_select	loop over the result of a SELECT statement
pg_listen	establish a callback for NOTIFY messages
pg_lo_creat	create a large object
pg_lo_open	open a large object
pg_lo_close	close a large object
pg_lo_read	read a large object
pg_lo_write	write a large object
pg_lo_lseek	seek to a position in a large object
pg_lo_tell	return the current seek position of a large object
pg_lo_unlink	delete a large object
pg_lo_import	import a Unix file into a large object
pg_lo_export	export a large object into a Unix file

These commands are described further on subsequent pages.

The pg_lo* routines are interfaces to the Large Object features of Postgres. The functions are designed to mimic the analogous file system functions in the standard Unix file system interface. The pg_lo* routines should be used within a BEGIN/END transaction block because the file descriptor returned by pg_lo_open is only valid for the current transaction. pg_lo_import and pg_lo_export MUST be used in a BEGIN/END transaction block.

56.2. Examples

Here's a small example of how to use the routines:

```
# getDBs :
#   get the names of all the databases at a given host and port
 number
#   with the defaults being the localhost and port 5432
#   return them in alphabetical order
```

```
proc getDBs { {host "localhost"} {port "5432"} } {
    # datnames is the list to be result
    set conn [pg_connect template1 -host $host -port $port]
    set res [pg_exec $conn "SELECT datname FROM pg_database ORDER
 BY datname"]
    set ntups [pg_result $res -numTuples]
    for {set i 0} {$i < $ntups} {incr i} {
 lappend datnames [pg_result $res -getTuple $i]
    }
    pg_result $res -clear
    pg_disconnect $conn
    return $datnames
}
```

56.3. pgtcl Command Reference Information

Name

pg_connect — opens a connection to the backend server

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

```
pg_connect -conninfo connectOptions
pg_connect dbName [-host hostName]
  [-port portNumber] [-tty pqtty]
  [-options optionalBackendArgs]
```

Inputs (new style)

connectOptions

A string of connection options, each written in the form keyword = value.

Inputs (old style)

dbName

Specifies a valid database name.

[-host *hostName*]

Specifies the domain name of the backend server for *dbName*.

[-port *portNumber*]

Specifies the IP port number of the backend server for *dbName*.

[-tty *pqtty*]

Specifies file or tty for optional debug output from backend.

[-options *optionalBackendArgs*]

Specifies options for the backend server for *dbName*.

Outputs

dbHandle

If successful, a handle for a database connection is returned. Handles start with the prefix "pgsql".

Description

`pg_connect` opens a connection to the Postgres backend.

Two syntaxes are available. In the older one, each possible option has a separate option switch in the pg_connect statement. In the newer form, a single option string is supplied that can contain multiple option values. See `pg_conndefaults` for info about the available options in the newer syntax.

Usage

XXX thomas 1997-12-24

Name

pg_disconnect — closes a connection to the backend server

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

```
pg_disconnect dbHandle
```

Inputs

dbHandle

Specifies a valid database handle.

Outputs

None

Description

`pg_disconnect` closes a connection to the Postgres backend.

Name

pg_conndefaults — obtain information about default connection parameters

Synopsis

<refsynopsisdivinfo>1998-10-07</refsynopsisdivinfo>

```
pg_conndefaults
```

Inputs

None.

Outputs

option list

The result is a list describing the possible connection options and their current default values. Each entry in the list is a sublist of the format:

{optname label dispchar dispsize value}

where the optname is usable as an option in `pg_connect -conninfo`.

Description

`pg_conndefaults` returns info about the connection options available in `pg_connect -conninfo` and the current default value for each option.

Usage

pg_conndefaults

Name

pg_exec — send a query string to the backend

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

`pg_exec` *`dbHandle queryString`*

Inputs

`dbHandle`

Specifies a valid database handle.

`queryString`

Specifies a valid SQL query.

Outputs

`resultHandle`

A Tcl error will be returned if Pgtcl was unable to obtain a backend response. Otherwise, a query result object is created and a handle for it is returned. This handle can be passed to `pg_result` to obtain the results of the query.

Description

`pg_exec` submits a query to the Postgres backend and returns a result. Query result handles start with the connection handle and add a period and a result number.

Note that lack of a Tcl error is not proof that the query succeeded! An error message returned by the backend will be processed as a query result with failure status, not by generating a Tcl error in pg_exec.

Name

pg_result — get information about a query result

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

`pg_result` *`resultHandle resultOption`*

Inputs

`resultHandle`

The handle for a query result.

`resultOption`

Specifies one of several possible options.

Options

-status

the status of the result.

-error

the error message, if the status indicates error; otherwise an empty string.

-conn

the connection that produced the result.

-oid

if the command was an INSERT, the OID of the inserted tuple; otherwise an empty string.

-numTuples

the number of tuples returned by the query.

-numAttrs

the number of attributes in each tuple.

-list VarName

assign the results to a list of lists.

-assign arrayName

assign the results to an array, using subscripts of the form (tupno,attributeName).

-assignbyidx arrayName ?appendstr?

assign the results to an array using the first attribute's value and the remaining attributes' names as keys. If appendstr is given then it is appended to each key. In short, all but the first field of each tuple are stored into the array, using subscripts of the form (firstFieldValue,fieldNameAppendStr).

-getTuple tupleNumber

returns the fields of the indicated tuple in a list. Tuple numbers start at zero.

-tupleArray tupleNumber arrayName

stores the fields of the tuple in array arrayName, indexed by field names. Tuple numbers start at zero.

-attributes

returns a list of the names of the tuple attributes.

-lAttributes

returns a list of sublists, {name ftype fsize} for each tuple attribute.

-clear

clear the result query object.

Outputs

The result depends on the selected option, as described above.

Description

`pg_result` returns information about a query result created by a prior `pg_exec`.

You can keep a query result around for as long as you need it, but when you are done with it, be sure to free it by executing `pg_result -clear`. Otherwise, you have a memory leak, and Pgtcl will eventually start complaining that you've created too many query result objects.

Name

pg_select — loop over the result of a SELECT statement

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

```
pg_select dbHandle queryString
  arrayVar queryProcedure
```

Inputs

dbHandle

Specifies a valid database handle.

queryString

Specifies a valid SQL select query.

arrayVar

Array variable for tuples returned.

queryProcedure

Procedure run on each tuple found.

Outputs

resultHandle

the return result is either an error message or a handle for a query result.

Description

`pg_select` submits a SELECT query to the Postgres backend, and executes a given chunk of code for each tuple in the result. The *queryString* must be a SELECT statement. Anything else returns an error. The *arrayVar* variable is an array name used in the loop. For each tuple, *arrayVar* is filled in with the tuple field values, using the field names as the array indexes. Then the *queryProcedure* is executed.

Usage

This would work if table "table" has fields "control" and "name" (and, perhaps, other fields):

```
pg_select $pgconn "SELECT * from table" array {
 puts [format "%5d %s" array(control) array(name)]
}
```

Name

pg_listen — sets or changes a callback for asynchronous NOTIFY messages

Synopsis

<refsynopsisdivinfo>1998-5-22</refsynopsisdivinfo>

`pg_listen` *`dbHandle notifyName callbackCommand`*

Inputs

`dbHandle`

Specifies a valid database handle.

`notifyName`

Specifies the notify condition name to start or stop listening to.

`callbackCommand`

If present and not empty, provides the command string to execute when a matching notification arrives.

Outputs

None

Description

`pg_listen` creates, changes, or cancels a request to listen for asynchronous NOTIFY messages from the Postgres backend. With a callbackCommand parameter, the request is established, or the command string of an already existing request is replaced. With no callbackCommand parameter, a prior request is canceled.

After a `pg_listen` request is established, the specified command string is executed whenever a NOTIFY message bearing the given name arrives from the backend. This occurs when any Postgres client application issues a NOTIFY command referencing that name. (Note that the name can be, but does not have to be, that of an existing relation in the database.) The command string is executed from the Tcl idle loop. That is the normal idle state of an application written with Tk. In non-Tk Tcl shells, you can execute `update` or `vwait` to cause the idle loop to be entered.

You should not invoke the SQL statements LISTEN or UNLISTEN directly when using `pg_listen`. Pgtcl takes care of issuing those statements for you. But if you want to send a NOTIFY message yourself, invoke the SQL NOTIFY statement using `pg_exec`.

Name

pg_lo_creat — create a large object

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

```
pg_lo_creat conn mode
```

Inputs

conn

Specifies a valid database connection.

mode

Specifies the access mode for the large object

Outputs

objOid

The oid of the large object created.

Description

`pg_lo_creat` creates an Inversion Large Object.

Usage

mode can be any OR'ing together of INV_READ, INV_WRITE, and INV_ARCHIVE. The OR delimiter character is "|".

```
[pg_lo_creat $conn "INV_READ|INV_WRITE"]
```

Name

pg_lo_open — open a large object

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

`pg_lo_open` *`conn objOid mode`*

Inputs

`conn`

Specifies a valid database connection.

`objOid`

Specifies a valid large object oid.

`mode`

Specifies the access mode for the large object

Outputs

`fd`

A file descriptor for use in later pg_lo* routines.

Description

`pg_lo_open` open an Inversion Large Object.

Usage

Mode can be either "r", "w", or "rw".

Name

pg_lo_close — close a large object

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

`pg_lo_close` *`conn fd`*

Inputs

`conn`

Specifies a valid database connection.

`fd`

A file descriptor for use in later pg_lo* routines.

Outputs

None

Description

`pg_lo_close` closes an Inversion Large Object.

Usage

Name

pg_lo_read — read a large object

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

`pg_lo_read` *`conn fd bufVar len`*

Inputs

`conn`

Specifies a valid database connection.

`fd`

File descriptor for the large object from pg_lo_open.

`bufVar`

Specifies a valid buffer variable to contain the large object segment.

`len`

Specifies the maximum allowable size of the large object segment.

Outputs

None

Description

`pg_lo_read` reads at most *`len`* bytes from a large object into a variable named *`bufVar`*.

Usage

`bufVar` must be a valid variable name.

Name

pg_lo_write — write a large object

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

`pg_lo_write` *conn fd buf len*

Inputs

conn

Specifies a valid database connection.

fd

File descriptor for the large object from pg_lo_open.

buf

Specifies a valid string variable to write to the large object.

len

Specifies the maximum size of the string to write.

Outputs

None

Description

`pg_lo_write` writes at most *len* bytes to a large object from a variable *buf*.

Usage

buf must be the actual string to write, not a variable name.

Name

pg_lo_lseek — seek to a position in a large object

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

`pg_lo_lseek` *`conn fd offset whence`*

Inputs

`conn`

Specifies a valid database connection.

`fd`

File descriptor for the large object from pg_lo_open.

`offset`

Specifies a zero-based offset in bytes.

`whence`

whence can be "SEEK_CUR", "SEEK_END", or "SEEK_SET"

Outputs

None

Description

`pg_lo_lseek` positions to *`offset`* bytes from the beginning of the large object.

Usage

`whence` can be "SEEK_CUR", "SEEK_END", or "SEEK_SET".

Name

pg_lo_tell — return the current seek position of a large object

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

```
pg_lo_tell conn fd
```

Inputs

conn

Specifies a valid database connection.

fd

File descriptor for the large object from pg_lo_open.

Outputs

offset

A zero-based offset in bytes suitable for input to `pg_lo_lseek`.

Description

`pg_lo_tell` returns the current to *offset* in bytes from the beginning of the large object.

Usage

Name

pg_lo_unlink — delete a large object

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

```
pg_lo_unlink conn lobjId
```

Inputs

conn

Specifies a valid database connection.

lobjId

Identifier for a large object. XXX Is this the same as objOid in other calls?? - thomas 1998-01-11

Outputs

None

Description

`pg_lo_unlink` deletes the specified large object.

Usage

Name

pg_lo_import — import a large object from a Unix file

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

```
pg_lo_import conn filename
```

Inputs

conn

Specifies a valid database connection.

filename

Unix file name.

Outputs

None XXX Does this return a lobjId? Is that the same as the objOid in other calls? thomas - 1998-01-11

Description

`pg_lo_import` reads the specified file and places the contents into a large object.

Usage

`pg_lo_import` must be called within a BEGIN/END transaction block.

Name

pg_lo_export — export a large object to a Unix file

Synopsis

<refsynopsisdivinfo>1997-12-24</refsynopsisdivinfo>

```
pg_lo_export conn lobjId filename
```

Inputs

conn

Specifies a valid database connection.

lobjId

Large object identifier. XXX Is this the same as the objOid in other calls?? thomas - 1998-01-11

filename

Unix file name.

Outputs

None XXX Does this return a lobjId? Is that the same as the objOid in other calls? thomas - 1998-01-11

Description

`pg_lo_export` writes the specified large object into a Unix file.

Usage

`pg_lo_export` must be called within a BEGIN/END transaction block.

Chapter 57. libpgeasy - Simplified C Binding Library

Author

Written by Bruce Momjian (pgman@candle.pha.pa.us[1]) and last updated 2000-03-30.

pgeasy allows you to cleanly interface to the libpq library, more like a 4GL SQL interface.

It consists of set of simplified C functions that encapsulate the functionality of libpq. The functions are:

- PGresult *doquery(char *query);
- PGconn *connectdb(char *options);
- void disconnectdb();
- int fetch(void *param,...);
- int fetchwithnulls(void *param,...);
- void reset_fetch();
- void on_error_continue();
- void on_error_stop();
- PGresult *get_result();
- void set_result(PGresult *newres);
- void unset_result(PGresult *oldres);

Many functions return a structure or value, so you can do more work with the result if required.

You basically connect to the database with `connectdb`, issue your query with `doquery`, fetch the results with `fetch`, and finish with `disconnectdb`.

For `SELECT` queries, `fetch` allows you to pass pointers as parameters, and on return the variables are filled with data from the binary cursor you opened. These binary cursors can not be used if you are running the pgeasy client on a system with a different architecture than the database server. If you pass a NULL pointer parameter, the column is skipped. `fetchwithnulls` allows you to retrieve the NULL status of the field by passing an `int*` after each result pointer, which returns true or false if the field is null. You can always use libpq functions on the PGresult pointer returned by `doquery`. `reset_fetch` starts the fetch back at the beginning.

`get_result`, `set_result`, and `unset_result` allow you to handle multiple result sets at the same time.

There are a variety of demonstration programs in the source directory.

[1] mailto:pgman@candle.pha.pa.us

Chapter 58. ODBC Interface

Tim Goeke
Thomas Lockhart

Note

Background information originally by Tim Goeke[1]

ODBC (Open Database Connectivity) is an abstract API which allows you to write applications which can interoperate with various RDBMS servers. ODBC provides a product-neutral interface between frontend applications and database servers, allowing a user or developer to write applications which are transportable between servers from different manufacturers..

58.1. Background

The ODBC API matches up on the backend to an ODBC-compatible data source. This could be anything from a text file to an Oracle or Postgres RDBMS.

The backend access come from ODBC drivers, or vendor specifc drivers that allow data access. psqlODBC is such a driver, along with others that are available, such as the OpenLink ODBC drivers.

Once you write an ODBC application, you *should* be able to connect to *any* back end database, regardless of the vendor, as long as the database schema is the same.

For example. you could have MS SQL Server and Postgres servers which have exactly the same data. Using ODBC, your Windows application would make exactly the same calls and the back end data source would look the same (to the Windows app).

58.2. Windows Applications

In the real world, differences in drivers and the level of ODBC support lessens the potential of ODBC:

- Access, Delphi, and Visual Basic all support ODBC directly.
- Under C++, such as Visual C++, you can use the C++ ODBC API.
- In Visual C++, you can use the CRecordSet class, which wraps the ODBC API set within an MFC 4.2 class. This is the easiest route if you are doing Windows C++ development under Windows NT.

58.2.1. Writing Applications

“ If I write an application for Postgres can I write it using ODBC calls to the Postgres server, or is that only when another database program like MS SQL Server or Access needs to access the data? ”

The ODBC API is the way to go. For Visual C++ coding you can find out more at Microsoft's web site or in your VC++ docs.

Visual Basic and the other RAD tools have Recordset objects that use ODBC directly to access data. Using the data-aware controls, you can quickly link to the ODBC back end database (*very* quickly).

Playing around with MS Access will help you sort this out. Try using `File->Get External Data`.

Tip

You'll have to set up a DSN first.

[1] mailto:tgoeke@xpressway.com

58.3. Unix Installation

ApplixWare has an ODBC database interface supported on at least some platforms. ApplixWare v4.4.2 has been demonstrated under Linux with Postgres v7.0 using the psqlODBC driver contained in the Postgres distribution.

58.3.1. Building the Driver

The first thing to note about the psqlODBC driver (or any ODBC driver) is that there must exist a driver manager on the system where the ODBC driver is to be used. There exists a freeware ODBC driver for Unix called iodbc which can be obtained from various locations on the Net, including at AS200[2]. Instructions for installing iodbc are beyond the scope of this document, but there is a `README` that can be found inside the iodbc compressed .shar file that should explain how to get it up and running.

Having said that, any driver manager that you can find for your platform should support the psqlODBC driver or any ODBC driver.

The Unix configuration files for psqlODBC have recently been extensively reworked to allow for easy building on supported platforms as well as to allow for support of other Unix platforms in the future. The new configuration and build files for the driver should make it a simple process to build the driver on the supported platforms. Currently these include Linux and FreeBSD but we are hoping other users will contribute the necessary information to quickly expand the number of platforms for which the driver can be built.

There are actually two separate methods to build the driver depending on how you received it and these differences come down to only where and how to run configure and make. The driver can be built in a standalone, client-only installation, or can be built as a part of the main Postgres distribution. The standalone installation is convenient if you have ODBC client applications on multiple, heterogeneous platforms. The integrated installation is convenient when the target client is the same as the server, or when the client and server have similar runtime configurations.

Specifically if you have received the psqlODBC driver as part of the Postgres distribution (from now on referred to as an "integrated" build) then you will configure and make the ODBC driver from the top level source directory of the Postgres distribution along with the rest of its libraries. If you received the driver as a standalone package than you will run configure and make from the directory in which you unpacked the driver source.

Integrated Installation

This installation procedure is appropriate for an integrated installation.

1. Specify the `--with-odbc` command-line argument for src/configure:

   ```
   % ./configure --with-odbc
   % make
   ```

2. Rebuild the Postgres distribution:

   ```
   % make install
   ```

3. (Optional) Install the ODBC catalog extensions available in `PGROOT/contrib/odbc/odbc.sql`:

   ```
   % psql -e template1 < $PGROOT/contrib/odbc/odbc.sql
   ```

[2] http://www.as220.org/FreeODBC/iodbc-2.12.shar.Z

where specifying `template1` as the target database will ensure that all subsequent new databases will have these same definitions.

Once configured, the ODBC driver will be built and installed into the areas defined for the other components of the Postgres system. The installation-wide ODBC configuration file will be placed into the top directory of the Postgres target tree (`POSTGRESDIR`). This can be overridden from the make command-line as

```
% make ODBCINST=filename install
```

Pre-v6.4 Integrated Installation

If you have a Postgres installation older than v6.4, you have the original source tree available, and you want to use the newest version of the ODBC driver, then you may want to try this form of installation.

1. Copy the output tar file to your target system and unpack it into a clean directory.

2. From the directory containing the sources, type:

   ```
   % ./configure
   % make
   % make POSTGRESDIR=PostgresTopDir install
   ```

3. (Optional) If you would like to install components into different trees, then you can specify various destinations explicitly:

   ```
   % make BINDIR=bindir  LIBDIR=libdir  HEADERDIR=headerdir
    ODBCINST=instfile install
   ```

Standalone Installation

A standalone installation is not integrated with or built on the normal Postgres distribution. It should be best suited for building the ODBC driver for multiple, heterogeneous clients who do not have a locally-installed Postgres source tree.

The default location for libraries and headers for the standalone installation is `/usr/local/lib` and `/usr/local/include/iodbc`, respectively. There is another system wide configuration file that gets installed as `/share/odbcinst.ini` (if `/share` exists) or as `/etc/odbcinst.ini` (if `/share` does not exist).

Note

Installation of files into `/share` or `/etc` requires system root privileges. Most installation steps for Postgres do not have this requirement, and you can choose another destination which is writable by your non-root Postgres superuser account instead.

1. The standalone installation distribution can be built from the Postgres distribution or may be obtained from Insight Distributors[3], the current maintainers of the non-Unix sources.

   Copy the zip or gzipped tarfile to an empty directory. If using the zip package unzip it with the command

   ```
   % unzip -a packagename
   ```

[3] http://www.insightdist.com/psqlodbc

The -a option is necessary to get rid of DOS CR/LF pairs in the source files.

If you have the gzipped tar package than simply run

```
% tar -xzf packagename
```

- (Optional) To create a tar file for a complete standalone installation from the main Postgres source tree:

2. Configure the main Postgres distribution.

3. Create the tar file:

```
% cd interfaces/odbc
% make standalone
```

4. Copy the output tar file to your target system. Be sure to transfer as a binary file if using ftp.

5. Unpack the tar file into a clean directory.

6. Configure the standalone installation:

```
% ./configure
```

The configuration can be done with options:

```
% ./configure --prefix=rootdir --with-odbc=inidir
```

where --prefix installs the libraries and headers in the directories *rootdir*/lib and *rootdir*/include/iodbc, and --with-odbc installs odbcinst.ini in the specified directory.

Note that both of these options can also be used from the integrated build but be aware that *when used in the integrated build* --prefix will also apply to the rest of your Postgres installation. --with-odbc applies only to the configuration file odbcinst.ini.

7. Compile and link the source code:

```
% make ODBCINST=instdir
```

You can also override the default location for installation on the 'make' command line. This only applies to the installation of the library and header files. Since the driver needs to know the location of the odbcinst.ini file attempting to override the enviroment variable that specifies its installation directory will probably cause you headaches. It is safest simply to allow the driver to install the odbcinst.ini file in the default directory or the directory you specified on the './configure' command line with --with-odbc.

8. Install the source code:

```
% make POSTGRESDIR=targettree install
```

To override the library and header installation directories separately you need to pass the correct installation variables on the make install command line. These variables are LIBDIR,

HEADERDIR and ODBCINST. Overriding POSTGRESDIR on the make command line will cause LIBDIR and HEADERDIR to be rooted at the new directory you specify. ODBCINST is independent of POSTGRESDIR.

Here is how you would specify the various destinations explicitly:

```
% make BINDIR=bindir LIBDIR=libdir HEADERDIR=headerdir install
```

For example, typing

```
% make POSTGRESDIR=/opt/psqlodbc install
```

(after you've used ./configure and make) will cause the libraries and headers to be installed in the directories /opt/psqlodbc/lib and /opt/psqlodbc/include/iodbc respectively.

The command

```
% make POSTGRESDIR=/opt/psqlodbc HEADERDIR=/usr/local install
```

should cause the libraries to be installed in /opt/psqlodbc/lib and the headers in /usr/local/include/iodbc. If this doesn't work as expected please contact one of the maintainers.

58.4. Configuration Files

~/.odbc.ini contains user-specified access information for the psqlODBC driver. The file uses conventions typical for Windows Registry files, but despite this restriction can be made to work.

The .odbc.ini file has three required sections. The first is [ODBC Data Sources] which is a list of arbitrary names and descriptions for each database you wish to access. The second required section is the Data Source Specification and there will be one of these sections for each database. Each section must be labeled with the name given in [ODBC Data Sources] and must contain the following entries:

```
Driver = POSTGRESDIR/lib/libpsqlodbc.so
Database=DatabaseName
Servername=localhost
Port=5432
```

Tip

Remember that the Postgres database name is usually a single word, without path names of any sort. The Postgres server manages the actual access to the database, and you need only specify the name from the client.

Other entries may be inserted to control the format of the display. The third required section is [ODBC] which must contain the InstallDir keyword and which may contain other options.

Here is an example .odbc.ini file, showing access information for three databases:

```
[ODBC Data Sources]
DataEntry = Read/Write Database
QueryOnly = Read-only Database
Test = Debugging Database
Default = Postgres Stripped

[DataEntry]
```

```
ReadOnly = 0
Servername = localhost
Database = Sales

[QueryOnly]
ReadOnly = 1
Servername = localhost
Database = Sales

[Test]
Debug = 1
CommLog = 1
ReadOnly = 0
Servername = localhost
Username = tgl
Password = "no$way"
Port = 5432
Database = test

[Default]
Servername = localhost
Database = tgl
Driver = /opt/postgres/current/lib/libpsqlodbc.so

[ODBC]
InstallDir = /opt/applix/axdata/axshlib
```

58.5. ApplixWare

58.5.1. Configuration

ApplixWare must be configured correctly in order for it to be able to access the Postgres ODBC software drivers.

Enabling ApplixWare Database Access

These instructions are for the `4.4.2` release of ApplixWare on Linux. Refer to the *Linux Sys Admin* on-line book for more detailed information.

1. You must modify `axnet.cnf` so that `elfodbc` can find `libodbc.so` (the ODBC driver manager) shared library. This library is included with the ApplixWare distribution, but `axnet.cnf` needs to be modified to point to the correct location.

   As root, edit the file *`applixroot`*`/applix/axdata/axnet.cnf`.

   a. At the bottom of `axnet.cnf`, find the line that starts with

      ```
      #libFor elfodbc /ax/...
      ```

   b. Change line to read

      ```
      libFor elfodbc applixroot/applix/axdata/axshlib/lib
      ```

      which will tell elfodbc to look in this directory for the ODBC support library. Typically Applix is installed in `/opt` so the full path would be `/opt/applix/axdata/`

`axshlib/lib`, but if you have installed Applix somewhere else then change the path accordingly.

2. Create `.odbc.ini` as described above. You may also want to add the flag

   ```
   TextAsLongVarchar=0
   ```

   to the database-specific portion of `.odbc.ini` so that text fields will not be shown as `**BLOB**`.

Testing ApplixWare ODBC Connections

1. Bring up Applix Data
2. Select the Postgres database of interest.
   a. Select `Query->Choose Server`.
   b. Select ODBC, and click `Browse`. The database you configured in `.odbc.ini` should be shown. Make sure that the `Host: field` is empty (if it is not, axnet will try to contact axnet on another machine to look for the database).
   c. Select the database in the box that was launched by `Browse`, then click `OK`.
   d. Enter username and password in the login identification dialog, and click `OK`.

   You should see "`Starting elfodbc server`" in the lower left corner of the data window. If you get an error dialog box, see the debugging section below.
3. The 'Ready' message will appear in the lower left corner of the data window. This indicates that you can now enter queries.
4. Select a table from Query->Choose tables, and then select Query->Query to access the database. The first 50 or so rows from the table should appear.

58.5.2. Common Problems

The following messages can appear while trying to make an ODBC connection through Applix Data:

Cannot launch gateway on server

`elfodbc` can't find `libodbc.so`. Check your `axnet.cnf`.

Error from ODBC Gateway: IM003::[iODBC][Driver Manager]Specified driver could not be loaded

`libodbc.so` cannot find the driver listed in `.odbc.ini`. Verify the settings.

Server: Broken Pipe

The driver process has terminated due to some other problem. You might not have an up-to-date version of the Postgres ODBC package.

setuid to 256: failed to launch gateway

The September release of ApplixWare v4.4.1 (the first release with official ODBC support under Linux) shows problems when usernames exceed eight (8) characters in length. Problem description ontributed by Steve Campbell[4].

[4] mailto:scampbell@lear.com

Author

Contributed by Steve Campbell[5] on 1998-10-20.

The axnet program's security system seems a little suspect. axnet does things on behalf of the user and on a true multiple user system it really should be run with root security (so it can read/write in each user's directory). I would hesitate to recommend this, however, since we have no idea what security holes this creates.

58.5.3. Debugging ApplixWare ODBC Connections

One good tool for debugging connection problems uses the Unix system utility strace.

Debugging with strace

1. Start applixware.
2. Start an strace on the axnet process. For example, if

   ```
   % ps -aucx | grep ax
   ```

   shows

   ```
   cary   10432  0.0  2.6  1740   392  ?  S  Oct  9  0:00 axnet
   cary   27883  0.9 31.0 12692  4596  ?  S   10:24  0:04 axmain
   ```

   Then run

   ```
   % strace -f -s 1024 -p 10432
   ```

3. Check the strace output.

   ### Note from Cary

   Many of the error messages from ApplixWare go to `stderr`, but I'm not sure where `stderr` is sent, so strace is the way to find out.

For example, after getting a "`Cannot launch gateway on server`", I ran strace on axnet and got

```
[pid 27947] open("/usr/lib/libodbc.so", O_RDONLY) = -1 ENOENT
(No such file or directory)
[pid 27947] open("/lib/libodbc.so", O_RDONLY) = -1 ENOENT
(No such file or directory)
[pid 27947] write(2, "/usr2/applix/axdata/elfodbc:
can't load library 'libodbc.so'\n", 61) = -1 EIO (I/O error)
```

So what is happening is that applix elfodbc is searching for libodbc.so, but it can't find it. That is why axnet.cnf needed to be changed.

58.5.4. Running the ApplixWare Demo

In order to go through the *ApplixWare Data Tutorial*, you need to create the sample tables that the Tutorial refers to. The ELF Macro used to create the tables tries to use a NULL condition on many of the database columns, and Postgres does not currently allow this option.

[5] mailto:scampbell@lear.com

To get around this problem, you can do the following:

Modifying the ApplixWare Demo

1. Copy `/opt/applix/axdata/eng/Demos/sqldemo.am` to a local directory.
2. Edit this local copy of `sqldemo.am`:
   a. Search for 'null_clause = "NULL"
   b. Change this to null_clause = ""
3. Start Applix Macro Editor.
4. Open the sqldemo.am file from the Macro Editor.
5. Select `File->Compile and Save`.
6. Exit Macro Editor.
7. Start Applix Data.
8. Select `*->Run Macro`
9. Enter the value "`sqldemo`", then click `OK`.

   You should see the progress in the status line of the data window (in the lower left corner).
10. You should now be able to access the demo tables.

58.5.5. Useful Macros

You can add information about your database login and password to the standard Applix startup macro file. This is an example `~/axhome/macros/login.am` file:

```
macro login
set_set_system_var@("sql_username@","tgl")
set_system_var@("sql_passwd@","no$way")
endmacro
```

Caution

You should be careful about the file protections on any file containing username and password information.

58.5.6. Supported Platforms

psqlODBC has been built and tested on Linux. There have been reports of success with FreeBSD and with Solaris. There are no known restrictions on the basic code for other platforms which already support Postgres.

Chapter 59. JDBC Interface

Author

Written by Peter T. Mount[1], the author of the JDBC driver.

JDBC is a core API of Java 1.1 and later. It provides a standard set of interfaces to SQL-compliant databases.

Postgres provides a *type 4* JDBC Driver. Type 4 indicates that the driver is written in Pure Java, and communicates in the database's own network protocol. Because of this, the driver is platform independent. Once compiled, the driver can be used on any platform.

59.1. Building the JDBC Interface

59.1.1. Compiling the Driver

The driver's source is located in the `src/interfaces/jdbc` directory of the source tree. To compile simply change directory to that directory, and type:

```
% make
```

Upon completion, you will find the archive `postgresql.jar` in the current directory. This is the JDBC driver.

Note

You must use make, not javac, as the driver uses some dynamic loading techniques for performance reasons, and javac cannot cope. The `Makefile` will generate the jar archive.

59.1.2. Installing the Driver

To use the driver, the jar archive postgresql.jar needs to be included in the `CLASSPATH`.

59.1.2.1. Example

I have an application that uses the JDBC driver to access a large database containing astronomical objects. I have the application and the jdbc driver installed in the /usr/local/lib directory, and the java jdk installed in /usr/local/jdk1.1.6.

To run the application, I would use:

```
export CLASSPATH = /usr/local/lib/finder.jar:/usr/local/lib/
postgresql.jar:.
java uk.org.retep.finder.Main
```

Loading the driver is covered later on in this chapter.

59.2. Preparing the Database for JDBC

Because Java can only use TCP/IP connections, the Postgres postmaster must be running with the -i flag.

[1] peter@retep.org.uk

Also, the `pg_hba.conf` file must be configured. It's located in the PGDATA directory. In a default installation, this file permits access only by Unix domain sockets. For the JDBC driver to connect to the same localhost, you need to add something like:

```
host          all          127.0.0.1     255.255.255.255   password
```

Here access to all databases are possible from the local machine with JDBC.

The JDBC Driver supports trust, ident, password and crypt authentication methods.

59.3. Using the Driver

This section is not intended as a complete guide to JDBC programming, but should help to get you started. For more information refer to the standard JDBC API documentation. Also, take a look at the examples included with the source. The basic example is used here.

59.4. Importing JDBC

Any source that uses JDBC needs to import the java.sql package, using:

```
import java.sql.*;
```

Important

Do not import the postgresql package. If you do, your source will not compile, as javac will get confused.

59.5. Loading the Driver

Before you can connect to a database, you need to load the driver. There are two methods available, and it depends on your code to the best one to use.

In the first method, your code implicitly loads the driver using the `Class.forName()` method. For Postgres, you would use:

```
Class.forName("postgresql.Driver");
```

This will load the driver, and while loading, the driver will automatically register itself with JDBC.

Note: The `forName()` method can throw a `ClassNotFoundException`, so you will need to catch it if the driver is not available.

This is the most common method to use, but restricts your code to use just Postgres. If your code may access another database in the future, and you don't use our extensions, then the second method is advisable.

The second method passes the driver as a parameter to the JVM as it starts, using the -D argument. Example:

```
% java -Djdbc.drivers=postgresql.Driver example.ImageViewer
```

In this example, the JVM will attempt to load the driver as part of it's initialisation. Once done, the ImageViewer is started.

Now, this method is the better one to use because it allows your code to be used with other databases, without recompiling the code. The only thing that would also change is the URL, which is covered next.

One last thing. When your code then tries to open a Connection, and you get a `No driver available` SQLException being thrown, this is probably caused by the driver not being in the classpath, or the value in the parameter not being correct.

59.6. Connecting to the Database

With JDBC, a database is represented by a URL (Uniform Resource Locator). With Postgres, this takes one of the following forms:

- jdbc:postgresql:*database*
- jdbc:postgresql://*host*/*database*
- jdbc:postgresql://*hostport*/*database*

where:

host

The hostname of the server. Defaults to "localhost".

port

The port number the server is listening on. Defaults to the Postgres standard port number (5432).

database

The database name.

To connect, you need to get a Connection instance from JDBC. To do this, you would use the DriverManager.getConnection() method:

```
Connection db = DriverManager.getConnection(url,user,pwd);
```

59.7. Issuing a Query and Processing the Result

Any time you want to issue SQL statements to the database, you require a Statement instance. Once you have a Statement, you can use the executeQuery() method to issue a query. This will return a ResultSet instance, which contains the entire result.

59.7.1. Using the Statement Interface

The following must be considered when using the Statement interface:

- You can use a Statement instance as many times as you want. You could create one as soon as you open the connection, and use it for the connections lifetime. You have to remember that only one ResultSet can exist per Statement.
- If you need to perform a query while processing a ResultSet, you can simply create and use another Statement.
- If you are using Threads, and several are using the database, you must use a separate Statement for each thread. Refer to the sections covering Threads and Servlets later in this document if you are thinking of using them, as it covers some important points.

59.7.2. Using the ResultSet Interface

The following must be considered when using the ResultSet interface:

- Before reading any values, you must call `next()`. This returns true if there is a result, but more importantly, it prepares the row for processing.
- Under the JDBC spec, you should access a field only once. It's safest to stick to this rule, although at the current time, the Postgres driver will allow you to access a field as many times as you want.
- You must close a ResultSet by calling `close()` once you have finished with it.
- Once you request another query with the Statement used to create a ResultSet, the currently open instance is closed.

An example is as follows:

```
Statement st = db.createStatement();
ResultSet rs = st.executeQuery("select * from mytable");
while(rs.next()) {
    System.out.print("Column 1 returned ");
    System.out.println(rs.getString(1));
}
rs.close();
st.close();
```

59.8. Performing Updates

To perform an update (or any other SQL statement that does not return a result), you simply use the `executeUpdate()` method:

```
st.executeUpdate("create table basic (a int2, b int2)");
```

59.9. Closing the Connection

To close the database connection, simply call the close() method to the Connection:

```
db.close();
```

59.10. Using Large Objects

In Postgres, large objects (also known as *blobs*) are used to hold data in the database that cannot be stored in a normal SQL table. They are stored as a Table/Index pair, and are referred to from your own tables by an OID value.

Important

For Postgres, you must access large objects within an SQL transaction. Although this has always been true in principle, it was not strictly enforced until the release of v6.5. You would open a transaction by using the `setAutoCommit()` method with an input parameter of `false`:

```
Connection mycon;
...
mycon.setAutoCommit(false);
... now use Large Objects
```

Now, there are two methods of using Large Objects. The first is the standard JDBC way, and is documented here. The other, uses our own extension to the api, which presents the libpq large object API to Java, providing even better access to large objects than the standard. Internally, the driver uses the extension to provide large object support.

In JDBC, the standard way to access them is using the getBinaryStream() method in ResultSet, and setBinaryStream() method in PreparedStatement. These methods make the large object appear as a Java stream, allowing you to use the java.io package, and others, to manipulate the object.

For example, suppose you have a table containing the file name of an image, and a large object containing that image:

```
create table images (imgname name,imgoid oid);
```

To insert an image, you would use:

```
File file = new File("myimage.gif");
FileInputStream fis = new FileInputStream(file);
PreparedStatement ps = conn.prepareStatement("insert into images
 values (?,?)");
ps.setString(1,file.getName());
ps.setBinaryStream(2,fis,file.length());
ps.executeUpdate();
ps.close();
fis.close();
```

Now in this example, setBinaryStream transfers a set number of bytes from a stream into a large object, and stores the OID into the field holding a reference to it.

Retrieving an image is even easier (I'm using PreparedStatement here, but Statement can equally be used):

```
PreparedStatement ps = con.prepareStatement("select oid from images
 where name=?");
ps.setString(1,"myimage.gif");
ResultSet rs = ps.executeQuery();
if(rs!=null) {
    while(rs.next()) {
        InputStream is = rs.getBinaryInputStream(1);
        // use the stream in some way here
        is.close();
    }
    rs.close();
}
ps.close();
```

Now here you can see where the Large Object is retrieved as an InputStream. You'll also notice that we close the stream before processing the next row in the result. This is part of the JDBC Specification, which states that any InputStream returned is closed when ResultSet.next() or ResultSet.close() is called.

59.11. Postgres Extensions to the JDBC API

Postgres is an extensible database system. You can add your own functions to the backend, which can then be called from queries, or even add your own data types.

Now, as these are facilities unique to us, we support them from Java, with a set of extension API's. Some features within the core of the standard driver actually use these extensions to implement Large Objects, etc.

```
Accessing the extensions

To access some of the extensions, you need to use some extra
 methods
in the postgresql.Connection class. In this case, you would need
 to
case the return value of Driver.getConnection().

For example:

    Connection db = Driver.getConnection(url,user,pass);

    // later on
    Fastpath fp = ((postgresql.Connection)db).getFastpathAPI();

Class postgresql.Connection

java.lang.Object
   |
   +----postgresql.Connection

   public class Connection extends Object implements Connection

These are the extra methods used to gain access to our extensions.
 I
have not listed the methods defined by java.sql.Connection.

 public Fastpath getFastpathAPI() throws SQLException

          This returns the Fastpath API for the current connection.

          NOTE: This is not part of JDBC, but allows access to
functions on the postgresql backend itself.

          It is primarily used by the LargeObject API

          The best way to use this is as follows:

 import postgresql.fastpath.*;
 ...
 Fastpath fp = ((postgresql.Connection)myconn).getFastpathAPI();

          where myconn is an open Connection to postgresql.

        Returns:
                Fastpath object allowing access to functions on
 the
postgresql backend.

        Throws: SQLException
                by Fastpath when initialising for first time

 public LargeObjectManager getLargeObjectAPI() throws SQLException
```

```
          This returns the LargeObject API for the current
 connection.

          NOTE: This is not part of JDBC, but allows access to
functions on the postgresql backend itself.

          The best way to use this is as follows:

 import postgresql.largeobject.*;
 ...
 LargeObjectManager lo =
((postgresql.Connection)myconn).getLargeObjectAPI();

          where myconn is an open Connection to postgresql.

        Returns:
                LargeObject object that implements the API

        Throws: SQLException
                by LargeObject when initialising for first time

 public void addDataType(String type,
                         String name)

          This allows client code to add a handler for one of
postgresql's more unique data types. Normally, a data type not
 known
by the driver is returned by ResultSet.getObject() as a PGobject
instance.

This method allows you to write a class that extends PGobject, and
tell the driver the type name, and class name to use.

The down side to this, is that you must call this method each time
 a
connection is made.

          NOTE: This is not part of JDBC, but an extension.

          The best way to use this is as follows:

 ...
 ((postgresql.Connection)myconn).addDataType("mytype","my.class.name"-
);
 ...

          where myconn is an open Connection to postgresql.

          The handling class must extend postgresql.util.PGobject

        See Also:
                PGobject

Fastpath

Fastpath is an API that exists within the libpq C interface, and
allows a client machine to execute a function on the database
 backend.
```

```
Most client code will not need to use this method, but it's
 provided
because the Large Object API uses it.

To use, you need to import the postgresql.fastpath package, using
 the
line:
     import postgresql.fastpath.*;

Then, in your code, you need to get a FastPath object:
     Fastpath fp = ((postgresql.Connection)conn).getFastpathAPI();

This will return an instance associated with the database
 connection
that you can use to issue commands. The casing of Connection to
postgresql.Connection is required, as the getFastpathAPI() is one
 of
our own methods, not JDBC's.

Once you have a Fastpath instance, you can use the fastpath()
 methods
to execute a backend function.

Class postgresql.fastpath.Fastpath

java.lang.Object
   |
   +----postgresql.fastpath.Fastpath

   public class Fastpath

   extends Object

   This class implements the Fastpath api.

   This is a means of executing functions imbeded in the
 postgresql
backend from within a java application.

   It is based around the file src/interfaces/libpq/fe-exec.c

   See Also:
          FastpathFastpathArg, LargeObject

Methods

 public Object fastpath(int fnid,
                        boolean resulttype,
                        FastpathArg args[]) throws SQLException

          Send a function call to the PostgreSQL backend

        Parameters:
                fnid - Function id
                resulttype - True if the result is an integer,
 false
for
                other results
```

```
                args - FastpathArguments to pass to fastpath

        Returns:
                null if no data, Integer if an integer result, or
byte[]
                otherwise

        Throws: SQLException
                if a database-access error occurs.

 public Object fastpath(String name,
                        boolean resulttype,
                        FastpathArg args[]) throws SQLException

          Send a function call to the PostgreSQL backend by name.

Note:
          the mapping for the procedure name to function id needs
 to
exist, usually to an earlier call to addfunction(). This is the
prefered method to call, as function id's can/may change between
versions of the backend. For an example of how this works, refer
 to
postgresql.LargeObject

        Parameters:
                name - Function name
                resulttype - True if the result is an integer,
 false
for
                other results
                args - FastpathArguments to pass to fastpath

        Returns:
                null if no data, Integer if an integer result, or
byte[]
                otherwise

        Throws: SQLException
                if name is unknown or if a database-access error
occurs.

        See Also:
                LargeObject

 public int getInteger(String name,
                       FastpathArg args[]) throws SQLException

          This convenience method assumes that the return value is
 an
Integer

        Parameters:
                name - Function name
                args - Function arguments

        Returns:
                integer result
```

```
        Throws: SQLException
                if a database-access error occurs or no result

 public byte[] getData(String name,
                       FastpathArg args[]) throws SQLException

          This convenience method assumes that the return value is
binary data

        Parameters:
                name - Function name
                args - Function arguments

        Returns:
                byte[] array containing result

        Throws: SQLException
                if a database-access error occurs or no result

 public void addFunction(String name,
                         int fnid)

          This adds a function to our lookup table.

          User code should use the addFunctions method, which is
 based
upon a query, rather than hard coding the oid. The oid for a
 function
is not guaranteed to remain static, even on different servers of
 the
same version.

        Parameters:
                name - Function name
                fnid - Function id

 public void addFunctions(ResultSet rs) throws SQLException

          This takes a ResultSet containing two columns. Column 1
contains the function name, Column 2 the oid.

          It reads the entire ResultSet, loading the values into
 the
function table.

          REMEMBER to close() the resultset after calling this!!

          Implementation note about function name lookups:

          PostgreSQL stores the function id's and their
 corresponding
names in the pg_proc table. To speed things up locally, instead of
querying each function from that table when required, a Hashtable
 is
used. Also, only the function's required are entered into this
 table,
keeping connection times as fast as possible.
```

```
          The postgresql.LargeObject class performs a query upon
 it's
startup, and passes the returned ResultSet to the addFunctions()
method here.

          Once this has been done, the LargeObject api refers to
 the
functions by name.

          Dont think that manually converting them to the oid's
 will
work. Ok, they will for now, but they can change during
 development
(there was some discussion about this for V7.0), so this is
implemented to prevent any unwarranted headaches in the future.

        Parameters:
                rs - ResultSet

        Throws: SQLException
                if a database-access error occurs.

        See Also:
                LargeObjectManager

 public int getID(String name) throws SQLException

          This returns the function id associated by its name

          If addFunction() or addFunctions() have not been called
 for
this name, then an SQLException is thrown.

        Parameters:
                name - Function name to lookup

        Returns:
                Function ID for fastpath call

        Throws: SQLException
                is function is unknown.

Class postgresql.fastpath.FastpathArg

java.lang.Object
   |
   +----postgresql.fastpath.FastpathArg

   public class FastpathArg extends Object

   Each fastpath call requires an array of arguments, the number
 and
type dependent on the function being called.

   This class implements methods needed to provide this capability.

   For an example on how to use this, refer to the
```

```
postgresql.largeobject package

   See Also:
          Fastpath, LargeObjectManager, LargeObject

Constructors

 public FastpathArg(int value)

          Constructs an argument that consists of an integer value

        Parameters:
                value - int value to set

 public FastpathArg(byte bytes[])

          Constructs an argument that consists of an array of bytes

        Parameters:
                bytes - array to store

 public FastpathArg(byte buf[],
                    int off,
                    int len)

           Constructs an argument that consists of part of a byte
array

        Parameters:
                buf - source array
                off - offset within array
                len - length of data to include

 public FastpathArg(String s)

          Constructs an argument that consists of a String.

        Parameters:
                s - String to store

Geometric Data Types

PostgreSQL has a set of datatypes that can store geometric
 features
into a table. These range from single points, lines, and polygons.

We support these types in Java with the postgresql.geometric
 package.

It contains classes that extend the postgresql.util.PGobject
 class.
Refer to that class for details on how to implement your own data
 type
handlers.

Class postgresql.geometric.PGbox

java.lang.Object
```

```
   |
   +----postgresql.util.PGobject
           |
           +----postgresql.geometric.PGbox

   public class PGbox extends PGobject implements Serializable,
Cloneable

   This represents the box datatype within postgresql.

Variables

 public PGpoint point[]

          These are the two corner points of the box.

Constructors

 public PGbox(double x1,
              double y1,
              double x2,
              double y2)

        Parameters:
                x1 - first x coordinate
                y1 - first y coordinate
                x2 - second x coordinate
                y2 - second y coordinate

 public PGbox(PGpoint p1,
              PGpoint p2)

        Parameters:
                p1 - first point
                p2 - second point

 public PGbox(String s) throws SQLException

        Parameters:
                s - Box definition in PostgreSQL syntax

        Throws: SQLException
                if definition is invalid

 public PGbox()

          Required constructor

Methods

 public void setValue(String value) throws SQLException

          This method sets the value of this object. It should be
overidden, but still called by subclasses.

        Parameters:
                value - a string representation of the value of
 the
```

```
object
        Throws: SQLException
                thrown if value is invalid for this type

        Overrides:
                setValue in class PGobject

 public boolean equals(Object obj)

        Parameters:
                obj - Object to compare with

        Returns:
                true if the two boxes are identical

        Overrides:
                equals in class PGobject

 public Object clone()

          This must be overidden to allow the object to be cloned

        Overrides:
                clone in class PGobject

 public String getValue()

        Returns:
                the PGbox in the syntax expected by postgresql

        Overrides:
                getValue in class PGobject

Class postgresql.geometric.PGcircle

java.lang.Object
   |
   +----postgresql.util.PGobject
           |
           +----postgresql.geometric.PGcircle

   public class PGcircle extends PGobject implements Serializable,
Cloneable

   This represents postgresql's circle datatype, consisting of a
 point
and a radius

Variables

 public PGpoint center

          This is the centre point

public double radius

          This is the radius
```

```
Constructors

 public PGcircle(double x,
                 double y,
                 double r)

        Parameters:
               x - coordinate of centre
                y - coordinate of centre
                r - radius of circle

 public PGcircle(PGpoint c,
                 double r)

        Parameters:
                c - PGpoint describing the circle's centre
                r - radius of circle

 public PGcircle(String s) throws SQLException

        Parameters:
                s - definition of the circle in PostgreSQL's
 syntax.

        Throws: SQLException
                on conversion failure

 public PGcircle()

          This constructor is used by the driver.

Methods

 public void setValue(String s) throws SQLException

        Parameters:
                s - definition of the circle in PostgreSQL's
 syntax.

        Throws: SQLException
                on conversion failure

        Overrides:
                setValue in class PGobject

 public boolean equals(Object obj)

        Parameters:
                obj - Object to compare with

        Returns:
                true if the two boxes are identical

        Overrides:
                equals in class PGobject

 public Object clone()
```

```
          This must be overidden to allow the object to be cloned

        Overrides:
                clone in class PGobject

 public String getValue()

        Returns:
                the PGcircle in the syntax expected by postgresql

        Overrides:
                getValue in class PGobject

Class postgresql.geometric.PGline

java.lang.Object
   |
   +----postgresql.util.PGobject
           |
           +----postgresql.geometric.PGline

   public class PGline extends PGobject implements Serializable,
Cloneable

   This implements a line consisting of two points. Currently line
 is
not yet implemented in the backend, but this class ensures that
 when
it's done were ready for it.

Variables

 public PGpoint point[]

          These are the two points.

Constructors

 public PGline(double x1,
               double y1,
               double x2,
               double y2)

        Parameters:
                x1 - coordinate for first point
                y1 - coordinate for first point
                x2 - coordinate for second point
                y2 - coordinate for second point

 public PGline(PGpoint p1,
               PGpoint p2)

        Parameters:
                p1 - first point
                p2 - second point

 public PGline(String s) throws SQLException
```

```
        Parameters:
                s - definition of the circle in PostgreSQL's
 syntax.

        Throws: SQLException
                on conversion failure

 public PGline()

          reuired by the driver

Methods

 public void setValue(String s) throws SQLException

        Parameters:
                s - Definition of the line segment in PostgreSQL's
syntax

        Throws: SQLException
                on conversion failure

        Overrides:
                setValue in class PGobject

 public boolean equals(Object obj)

        Parameters:
                obj - Object to compare with

        Returns:
                true if the two boxes are identical

        Overrides:
                equals in class PGobject

 public Object clone()

          This must be overidden to allow the object to be cloned

        Overrides:
                clone in class PGobject

 public String getValue()

        Returns:
                the PGline in the syntax expected by postgresql

        Overrides:
                getValue in class PGobject

Class postgresql.geometric.PGlseg

java.lang.Object
   |
   +----postgresql.util.PGobject
           |
           +----postgresql.geometric.PGlseg
```

```
   public class PGlseg extends PGobject implements Serializable,
Cloneable

   This implements a lseg (line segment) consisting of two points

Variables

 public PGpoint point[]

          These are the two points.

Constructors

 public PGlseg(double x1,
               double y1,
               double x2,
               double y2)

        Parameters:

                x1 - coordinate for first point
                y1 - coordinate for first point
                x2 - coordinate for second point
                y2 - coordinate for second point

 public PGlseg(PGpoint p1,
               PGpoint p2)

        Parameters:
                p1 - first point
                p2 - second point

 public PGlseg(String s) throws SQLException

        Parameters:
                s - definition of the circle in PostgreSQL's
 syntax.

        Throws: SQLException
                on conversion failure

 public PGlseg()

          reuired by the driver

Methods

 public void setValue(String s) throws SQLException

        Parameters:
                s - Definition of the line segment in PostgreSQL's
syntax

        Throws: SQLException
                on conversion failure

        Overrides:
```

```
                setValue in class PGobject

 public boolean equals(Object obj)

        Parameters:
                obj - Object to compare with

        Returns:
                true if the two boxes are identical

        Overrides:
                equals in class PGobject

 public Object clone()

          This must be overidden to allow the object to be cloned

        Overrides:
               clone in class PGobject

 public String getValue()

        Returns:
                the PGlseg in the syntax expected by postgresql

        Overrides:
                getValue in class PGobject

Class postgresql.geometric.PGpath

java.lang.Object
   |
   +----postgresql.util.PGobject
           |
           +----postgresql.geometric.PGpath

   public class PGpath extends PGobject implements Serializable,
Cloneable

   This implements a path (a multiple segmented line, which may be
closed)

Variables

 public boolean open

          True if the path is open, false if closed

 public PGpoint points[]

          The points defining this path

Constructors

 public PGpath(PGpoint points[],
               boolean open)

        Parameters:
```

```
                points - the PGpoints that define the path
                open - True if the path is open, false if closed

 public PGpath()

          Required by the driver

 public PGpath(String s) throws SQLException

        Parameters:
                s - definition of the circle in PostgreSQL's
 syntax.

        Throws: SQLException
                on conversion failure

Methods

 public void setValue(String s) throws SQLException

        Parameters:
                s - Definition of the path in PostgreSQL's syntax

        Throws: SQLException
                on conversion failure

        Overrides:
                setValue in class PGobject

 public boolean equals(Object obj)

        Parameters:
                obj - Object to compare with

        Returns:
                true if the two boxes are identical

        Overrides:
                equals in class PGobject

 public Object clone()

          This must be overidden to allow the object to be cloned

        Overrides:
                clone in class PGobject

 public String getValue()

          This returns the polygon in the syntax expected by
postgresql

        Overrides:
                getValue in class PGobject

 public boolean isOpen()

     This returns true if the path is open
```

```
 public boolean isClosed()

     This returns true if the path is closed

 public void closePath()

     Marks the path as closed

 public void openPath()

     Marks the path as open

Class postgresql.geometric.PGpoint

java.lang.Object
   |
   +----postgresql.util.PGobject
           |
           +----postgresql.geometric.PGpoint

   public class PGpoint extends PGobject implements Serializable,
Cloneable

   This implements a version of java.awt.Point, except it uses
 double
to represent the coordinates.

   It maps to the point datatype in postgresql.

Variables

 public double x

          The X coordinate of the point

 public double y

          The Y coordinate of the point

Constructors

 public PGpoint(double x,
                double y)

        Parameters:
                x - coordinate
                y - coordinate

 public PGpoint(String value) throws SQLException

          This is called mainly from the other geometric types,
 when a
point is imbeded within their definition.

        Parameters:
                value - Definition of this point in PostgreSQL's
syntax
```

```
public PGpoint()

          Required by the driver

Methods

public void setValue(String s) throws SQLException

        Parameters:
                s - Definition of this point in PostgreSQL's syntax

        Throws: SQLException
                on conversion failure

        Overrides:
                setValue in class PGobject

public boolean equals(Object obj)

        Parameters:
                obj - Object to compare with

        Returns:
                true if the two boxes are identical

        Overrides:
                equals in class PGobject

public Object clone()

          This must be overidden to allow the object to be cloned

        Overrides:
                clone in class PGobject

public String getValue()

        Returns:
                the PGpoint in the syntax expected by postgresql

        Overrides:
                getValue in class PGobject

public void translate(int x,
                      int y)

          Translate the point with the supplied amount.

        Parameters:
                x - integer amount to add on the x axis
                y - integer amount to add on the y axis

public void translate(double x,
                      double y)

          Translate the point with the supplied amount.
```

```
        Parameters:
                x - double amount to add on the x axis
                y - double amount to add on the y axis

 public void move(int x,
                  int y)

          Moves the point to the supplied coordinates.

        Parameters:
                x - integer coordinate
                y - integer coordinate

public void move(double x,
                  double y)

          Moves the point to the supplied coordinates.

        Parameters:
                x - double coordinate
                y - double coordinate

 public void setLocation(int x,
                         int y)

          Moves the point to the supplied coordinates. refer to
          java.awt.Point for description of this

        Parameters:
                x - integer coordinate
                y - integer coordinate

        See Also:
                Point

 public void setLocation(Point p)

          Moves the point to the supplied java.awt.Point refer to
          java.awt.Point for description of this

        Parameters:
                p - Point to move to

        See Also:
                Point

Class postgresql.geometric.PGpolygon

java.lang.Object
   |
   +----postgresql.util.PGobject
           |
           +----postgresql.geometric.PGpolygon

   public class PGpolygon extends PGobject implements
 Serializable,
Cloneable
```

```
   This implements the polygon datatype within PostgreSQL.

Variables

 public PGpoint points[]

          The points defining the polygon

Constructors

 public PGpolygon(PGpoint points[])

          Creates a polygon using an array of PGpoints

        Parameters:
                points - the points defining the polygon

 public PGpolygon(String s) throws SQLException

        Parameters:
                s - definition of the circle in PostgreSQL's
 syntax.

        Throws: SQLException
                on conversion failure

 public PGpolygon()

          Required by the driver

Methods

 public void setValue(String s) throws SQLException

        Parameters:
                s - Definition of the polygon in PostgreSQL's
 syntax

        Throws: SQLException
                on conversion failure

        Overrides:
                setValue in class PGobject

 public boolean equals(Object obj)

        Parameters:
                obj - Object to compare with

        Returns:
                true if the two boxes are identical

        Overrides:
                equals in class PGobject

 public Object clone()

          This must be overidden to allow the object to be cloned
```

```
        Overrides:
                clone in class PGobject

 public String getValue()

        Returns:
                the PGpolygon in the syntax expected by postgresql

        Overrides:
                getValue in class PGobject
```

Large Objects

Large objects are supported in the standard JDBC specification. However, that interface is limited, and the api provided by PostgreSQL allows for random access to the objects contents, as if it was a local file.

The postgresql.largeobject package profides to Java the libpq C interface's large object API. It consists of two classes, LargeObjectManager, which deals with creating, opening and deleting large obejects, and LargeObject which deals with an individual object.

Class postgresql.largeobject.LargeObject

```
java.lang.Object
   |
   +----postgresql.largeobject.LargeObject

public class LargeObject extends Object
```

This class implements the large object interface to postgresql.

It provides the basic methods required to run the interface, plus a pair of methods that provide InputStream and OutputStream classes for this object.

Normally, client code would use the getAsciiStream, getBinaryStream, or getUnicodeStream methods in ResultSet, or setAsciiStream, setBinaryStream, or setUnicodeStream methods in PreparedStatement to access Large Objects.

However, sometimes lower level access to Large Objects are required, that are not supported by the JDBC specification.

Refer to postgresql.largeobject.LargeObjectManager on how to gain access to a Large Object, or how to create one.

```
   See Also:
          LargeObjectManager
```

```
Variables

 public static final int SEEK_SET

          Indicates a seek from the begining of a file

 public static final int SEEK_CUR

          Indicates a seek from the current position

 public static final int SEEK_END

          Indicates a seek from the end of a file

Methods

 public int getOID()

        Returns:
                the OID of this LargeObject

 public void close() throws SQLException

          This method closes the object. You must not call methods
 in
this object after this is called.

    Throws: SQLException
                if a database-access error occurs.

 public byte[] read(int len) throws SQLException

          Reads some data from the object, and return as a byte[]
array

        Parameters:
                len - number of bytes to read

        Returns:
                byte[] array containing data read

        Throws: SQLException
                if a database-access error occurs.

 public void read(byte buf[],
                  int off,
                  int len) throws SQLException

          Reads some data from the object into an existing array

        Parameters:
                buf - destination array
                off - offset within array
                len - number of bytes to read

        Throws: SQLException
                if a database-access error occurs.
```

```
public void write(byte buf[]) throws SQLException

  Writes an array to the object


Parameters:
        buf - array to write

Throws: SQLException
        if a database-access error occurs.

public void write(byte buf[],
                  int off,
                  int len) throws SQLException

  Writes some data from an array to the object

Parameters:
        buf - destination array
        off - offset within array
        len - number of bytes to write

Throws: SQLException
        if a database-access error occurs.

public void seek(int pos,
                 int ref) throws SQLException

  Sets the current position within the object.

  This is similar to the fseek() call in the standard C
library.It allows you to have random access to the large object.

Parameters:
        pos - position within object
        ref - Either SEEK_SET, SEEK_CUR or SEEK_END
Throws: SQLException
        if a database-access error occurs.

public void seek(int pos) throws SQLException

  Sets the current position within the object.

  This is similar to the fseek() call in the standard C
library.It allows you to have random access to the large object.

Parameters:
        pos - position within object from begining

Throws: SQLException
        if a database-access error occurs.

public int tell() throws SQLException

Returns:
        the current position within the object
```

```
        Throws: SQLException
                if a database-access error occurs.

 public int size() throws SQLException

          This method is inefficient, as the only way to find out
 the
size of the object is to seek to the end, record the current
 position,
then return to the original position.

          A better method will be found in the future.

        Returns:
                the size of the large object

        Throws: SQLException
                if a database-access error occurs.

 public InputStream getInputStream() throws SQLException

          Returns an InputStream from this object.

          This InputStream can then be used in any method that
requires an InputStream.

        Throws: SQLException
                if a database-access error occurs.

 public OutputStream getOutputStream() throws SQLException

          Returns an OutputStream to this object

          This OutputStream can then be used in any method that
requires an OutputStream.

        Throws: SQLException
                if a database-access error occurs.

Class postgresql.largeobject.LargeObjectManager

java.lang.Object
   |
   +----postgresql.largeobject.LargeObjectManager

public class LargeObjectManager extends Object

This class implements the large object interface to postgresql.

   It provides methods that allow client code to create, open and
delete large objects from the database. When opening an object, an
instance of postgresql.largeobject.LargeObject is returned, and
 its
methods then allow access to the object.

This class can only be created by postgresql.Connection

To get access to this class, use the following segment of code:
```

```
 import postgresql.largeobject.*;
 Connection  conn;
 LargeObjectManager lobj;
 ... code that opens a connection ...
 lobj = ((postgresql.Connection)myconn).getLargeObjectAPI();

Normally, client code would use the getAsciiStream,
 getBinaryStream,
or getUnicodeStream methods in ResultSet, or setAsciiStream,
setBinaryStream, or setUnicodeStream methods in PreparedStatement
 to
access Large Objects.

   However, sometimes lower level access to Large Objects are
required, that are not supported by the JDBC specification.

   Refer to postgresql.largeobject.LargeObject on how to
 manipulate
the contents of a Large Object.

   See Also:
          LargeObject

Variables

 public static final int WRITE

          This mode indicates we want to write to an object

 public static final int READ

          This mode indicates we want to read an object

 public static final int READWRITE

          This mode is the default. It indicates we want read and
write access to a large object

Methods

 public LargeObject open(int oid) throws SQLException

          This opens an existing large object, based on its OID.
 This
          method assumes that READ and WRITE access is required
 (the
default).

        Parameters:
                oid - of large object

        Returns:
                LargeObject instance providing access to the object

        Throws: SQLException
                on error
```

```
public LargeObject open(int oid,
                        int mode) throws SQLException

          This opens an existing large object, based on its OID

        Parameters:
                oid - of large object
                mode - mode of open

        Returns:
                LargeObject instance providing access to the object

        Throws: SQLException
                on error

public int create() throws SQLException

          This creates a large object, returning its OID.

          It defaults to READWRITE for the new object's attributes.

        Returns:
                oid of new object

        Throws: SQLException
                on error

public int create(int mode) throws SQLException

          This creates a large object, returning its OID

        Parameters:
                mode - a bitmask describing different attributes
 of
the
                new object

        Returns:
                oid of new object

        Throws: SQLException
                on error

public void delete(int oid) throws SQLException

          This deletes a large object.

        Parameters:
                oid - describing object to delete

        Throws: SQLException
                on error

public void unlink(int oid) throws SQLException

          This deletes a large object.
```

```
          It is identical to the delete method, and is supplied as
 the
C API uses unlink.

        Parameters:
                oid - describing object to delete

        Throws: SQLException
                on error

Object Serialisation
PostgreSQL is not a normal SQL Database. It is far more extensible
than most other databases, and does support Object Oriented
 features
that are unique to it.

One of the consequences of this, is that you can have one table
 refer
to a row in another table. For example:

test=> create table users (username name,fullname text);
CREATE
test=> create table server (servername name,adminuser users);
CREATE
test=> insert into users values ('peter','Peter Mount');
INSERT 2610132 1
test=> insert into server values ('maidast',2610132::users);
INSERT 2610133 1
test=> select * from users;
username|fullname
--------+--------------
peter   |Peter Mount
(1 row)

test=> select * from server;
servername|adminuser
----------+---------
maidast   |  2610132
(1 row)

Ok, the above example shows that we can use a table name as a
 field,
and the row's oid value is stored in that field.

What does this have to do with Java?

In Java, you can store an object to a Stream as long as it's class
implements the java.io.Serializable interface. This process, known
 as
Object Serialization, can be used to store complex objects into
 the
database.

Now, under JDBC, you would have to use a LargeObject to store
 them.
However, you cannot perform queries on those objects.
```

```
What the postgresql.util.Serialize class does, is provide a means
 of
storing an object as a table, and to retrieve that object from a
table. In most cases, you would not need to access this class
 direct,
but you would use the PreparedStatement.setObject() and
ResultSet.getObject() methods. Those methods will check the
 objects
class name against the table's in the database. If a match is
 found,
it assumes that the object is a Serialized object, and retrieves
 it
from that table. As it does so, if the object contains other
serialized objects, then it recurses down the tree.

Sound's complicated? In fact, it's simpler than what I wrote -
 it's
just difficult to explain.

The only time you would access this class, is to use the create()
methods. These are not used by the driver, but issue one or more
"create table" statements to the database, based on a Java Object
 or
Class that you want to serialize.

Oh, one last thing. If your object contains a line like:

     public int oid;

then, when the object is retrieved from the table, it is set to
 the
oid within the table. Then, if the object is modified, and re-
serialized, the existing entry is updated.

If the oid variable is not present, then when the object is
serialized, it is always inserted into the table, and any existing
entry in the table is preserved.

Setting oid to 0 before serialization, will also cause the object
 to
be inserted. This enables an object to be duplicated in the
 database.

Class postgresql.util.Serialize

java.lang.Object
   |
   +----postgresql.util.Serialize

   public class Serialize extends Object

   This class uses PostgreSQL's object oriented features to store
 Java
Objects. It does this by mapping a Java Class name to a table in
 the
database. Each entry in this new table then represents a
 Serialized
```

```
instance of this class. As each entry has an OID (Object
 IDentifier),
this OID can be included in another table. This is too complex to
 show
here, and will be documented in the main documents in more detail.

Constructors

 public Serialize(Connection c,
                  String type) throws SQLException

          This creates an instance that can be used to serialize
ordeserialize a Java object from a PostgreSQL table.

Methods

 public Object fetch(int oid) throws SQLException

          This fetches an object from a table, given it's OID

        Parameters:
                oid - The oid of the object

        Returns:
                Object relating to oid

        Throws: SQLException
                on error

 public int store(Object o) throws SQLException

          This stores an object into a table, returning it's OID.

          If the object has an int called OID, and it is > 0, then
that value is used for the OID, and the table will be updated. If
 the
value of OID is 0, then a new row will be created, and the value
 of
OID will be set in the object. This enables an object's value in
 the
database to be updateable. If the object has no int called OID,
 then
the object is stored. However if the object is later retrieved,
amended and stored again, it's new state will be appended to the
table, and will not overwrite the old entries.

        Parameters:
                o - Object to store (must implement Serializable)

        Returns:
                oid of stored object

        Throws: SQLException
                on error

 public static void create(Connection con,
                           Object o) throws SQLException
```

```
          This method is not used by the driver, but it creates a
table, given a Serializable Java Object. It should be used before
serializing any objects.

        Parameters:
                c - Connection to database
                o - Object to base table on

        Throws: SQLException
                on error

                   Returns:
                Object relating to oid

        Throws: SQLException
                on error

 public int store(Object o) throws SQLException

          This stores an object into a table, returning it's OID.

          If the object has an int called OID, and it is > 0, then
that value is used for the OID, and the table will be updated. If
 the
value of OID is 0, then a new row will be created, and the value
 of
OID will be set in the object. This enables an object's value in
 the
database to be updateable. If the object has no int called OID,
 then
the object is stored. However if the object is later retrieved,
amended and stored again, it's new state will be appended to the
table, and will not overwrite the old entries.

        Parameters:
                o - Object to store (must implement Serializable)

        Returns:
                oid of stored object

        Throws: SQLException
                on error

 public static void create(Connection con,
                           Object o) throws SQLException

          This method is not used by the driver, but it creates a
table, given a Serializable Java Object. It should be used before
serializing any objects.

        Parameters:
                c - Connection to database
                o - Object to base table on

        Throws: SQLException
                on error

 public static void create(Connection con,
```

```
                            Class c) throws SQLException

          This method is not used by the driver, but it creates a
table, given a Serializable Java Object. It should be used before
serializing any objects.

        Parameters:
                c - Connection to database
                o - Class to base table on

        Throws: SQLException
                on error

 public static String toPostgreSQL(String name) throws SQLException

          This converts a Java Class name to a postgresql table, by
          replacing . with _

          Because of this, a Class name may not have _ in the name.

          Another limitation, is that the entire class name
 (including
          packages) cannot be longer than 31 characters (a limit
forced by PostgreSQL).

        Parameters:
                name - Class name

        Returns:
                PostgreSQL table name

        Throws: SQLException
                on error

 public static String toClassName(String name) throws SQLException

          This converts a postgresql table to a Java Class name, by
          replacing _ with .

        Parameters:
                name - PostgreSQL table name

        Returns:
                Class name

        Throws: SQLException
                on error

Utility Classes

The postgresql.util package contains classes used by the internals
 of
the main driver, and the other extensions.

Class postgresql.util.PGmoney

java.lang.Object
   |
```

```
   +----postgresql.util.PGobject
           |
           +----postgresql.util.PGmoney

   public class PGmoney extends PGobject implements Serializable,
Cloneable

   This implements a class that handles the PostgreSQL money type

Variables

 public double val

          The value of the field

Constructors

 public PGmoney(double value)

        Parameters:
                value - of field

 public PGmoney(String value) throws SQLException

          This is called mainly from the other geometric types,
 when a
point is imbeded within their definition.

        Parameters:
                value - Definition of this point in PostgreSQL's
syntax

 public PGmoney()

          Required by the driver

Methods

 public void setValue(String s) throws SQLException

        Parameters:
                s - Definition of this point in PostgreSQL's syntax

        Throws: SQLException
                on conversion failure

        Overrides:
                setValue in class PGobject

 public boolean equals(Object obj)

        Parameters:
                obj - Object to compare with

        Returns:
                true if the two boxes are identical

        Overrides:
```

```
                equals in class PGobject

 public Object clone()

          This must be overidden to allow the object to be cloned

        Overrides:
                clone in class PGobject

 public String getValue()

        Returns:
                the PGpoint in the syntax expected by postgresql

        Overrides:
                getValue in class PGobject

Class postgresql.util.PGobject

java.lang.Object
   |
   +----postgresql.util.PGobject

   public class PGobject extends Object implements Serializable,
Cloneable

   This class is used to describe data types that are unknown by
     JDBC
Standard.
    A call to postgresql.Connection permits a class that extends
 this
class to be associated with a named type. This is how the
postgresql.geometric package operates.
    ResultSet.getObject() will return this class for any type that
 is
not recognised on having it's own handler. Because of this, any
postgresql data type is supported.

Constructors

 public PGobject()

          This is called by postgresql.Connection.getObject() to
create the object.

Methods

 public final void setType(String type)

          This method sets the type of this object.

          It should not be extended by subclasses, hence its final

        Parameters:
                type - a string describing the type of the object

 public void setValue(String value) throws SQLException
```

```
          This method sets the value of this object. It must be
overidden.

        Parameters:
                value - a string representation of the value of the
                object

        Throws: SQLException
                thrown if value is invalid for this type

 public final String getType()

          As this cannot change during the life of the object,
 it's
final.

        Returns:
                the type name of this object

 public String getValue()

          This must be overidden, to return the value of the
 object,
in the form required by postgresql.

        Returns:
                the value of this object

 public boolean equals(Object obj)

          This must be overidden to allow comparisons of objects

        Parameters:
                obj - Object to compare with

        Returns:
                true if the two boxes are identical

        Overrides:
                equals in class Object

 public Object clone()

          This must be overidden to allow the object to be cloned

        Overrides:
                clone in class Object

 public String toString()

          This is defined here, so user code need not overide it.

        Returns:
                the value of this object, in the syntax expected
 by
postgresql

        Overrides:
```

```
                toString in class Object

Class postgresql.util.PGtokenizer

java.lang.Object
   |
   +----postgresql.util.PGtokenizer

   public class PGtokenizer extends Object

   This class is used to tokenize the text output of postgres.

   We could have used StringTokenizer to do this, however, we
 needed
to handle nesting of '(' ')' '[' ']' '<' and '>' as these are used
 by
the geometric data types.

   It's mainly used by the geometric classes, but is useful in
 parsing
any output from custom data types output from postgresql.

   See Also:
          PGbox, PGcircle, PGlseg, PGpath, PGpoint, PGpolygon

Constructors

 public PGtokenizer(String string,
                    char delim)

          Create a tokeniser.

        Parameters:
                string - containing tokens
                delim - single character to split the tokens

Methods

 public int tokenize(String string,
                     char delim)

          This resets this tokenizer with a new string and/or
delimiter.

        Parameters:
                string - containing tokens
                delim - single character to split the tokens

 public int getSize()

        Returns:
                the number of tokens available

 public String getToken(int n)

        Parameters:
                n - Token number ( 0 ... getSize()-1 )
```

```
        Returns:
                The token value

 public PGtokenizer tokenizeToken(int n,
                                  char delim)

          This returns a new tokenizer based on one of our tokens.
 The
geometric datatypes use this to process nested tokens (usually
PGpoint).

        Parameters:
                n - Token number ( 0 ... getSize()-1 )
                delim - The delimiter to use

        Returns:
                A new instance of PGtokenizer based on the token

 public static String remove(String s,
                             String l,
                             String t)

          This removes the lead/trailing strings from a string

        Parameters:
                s - Source string
                l - Leading string to remove
                t - Trailing string to remove

        Returns:
                String without the lead/trailing strings

 public void remove(String l,
                    String t)

          This removes the lead/trailing strings from all tokens

        Parameters:
                l - Leading string to remove
                t - Trailing string to remove

 public static String removePara(String s)

          Removes ( and ) from the beginning and end of a string

        Parameters:
                s - String to remove from

        Returns:
                String without the ( or )

 public void removePara()

          Removes ( and ) from the beginning and end of all tokens

        Returns:
                String without the ( or )
```

```
 public static String removeBox(String s)

          Removes [ and ] from the beginning and end of a string

        Parameters:
                s - String to remove from

        Returns:
                String without the [ or ]

 public void removeBox()

          Removes [ and ] from the beginning and end of all tokens

        Returns:
                String without the [ or ]

 public static String removeAngle(String s)

          Removes < and > from the beginning and end of a string

        Parameters:
                s - String to remove from

        Returns:
                String without the < or >

 public void removeAngle()

          Removes < and > from the beginning and end of all tokens

        Returns:
                String without the < or >
```

Class postgresql.util.Serialize

This was documented earlier under Object Serialisation.

Class postgresql.util.UnixCrypt

```
java.lang.Object
   |
   +----postgresql.util.UnixCrypt

   public class UnixCrypt extends Object

   This class provides us with the ability to encrypt passwords
 when
sent over the network stream

   Contains static methods to encrypt and compare passwords with
 Unix
encrypted passwords.

   See John Dumas's Java Crypt page for the original source.

   http://www.zeh.com/local/jfd/crypt.html
```

```
Methods

 public static final String crypt(String salt,
                                  String original)

          Encrypt a password given the cleartext password and a
"salt".

        Parameters:
                salt - A two-character string representing the
 salt
used
                to iterate the encryption engine in lots of
 different
                ways. If you are generating a new encryption then
 this
                value should be randomised.
                original - The password to be encrypted.

        Returns:
                A string consisting of the 2-character salt
 followed
by
                the encrypted password.

 public static final String crypt(String original)

          Encrypt a password given the cleartext password. This
 method
generates a random salt using the 'java.util.Random' class.

        Parameters:
                original - The password to be encrypted.

        Returns:
                A string consisting of the 2-character salt
 followed
by
                the encrypted password.

 public static final boolean matches(String encryptedPassword,
                                     String enteredPassword)

          Check that enteredPassword encrypts to encryptedPassword.

        Parameters:
                encryptedPassword - The encryptedPassword. The
 first
two characters are assumed to be the salt. This string would be
 the
same as one found in a Unix /etc/passwd file.
                enteredPassword - The password as entered by the
 user
(or otherwise aquired).

        Returns:
                true if the password should be considered correct.
```

```
Using the driver in a multi Threaded or Servlet environment

A problem with many JDBC drivers, is that only one thread can use
 a
Connection at any one time - otherwise a thread could send a query
while another one is receiving results, and this would be a bad
 thing
for the database engine.

PostgreSQL 6.4, brings thread safety to the entire driver.
 Standard
JDBC was thread safe in 6.3.x, but the Fastpath
     API wasn't.

So, if your application uses multiple threads (which most decent
 ones
would), then you don't have to worry about complex schemes to
 ensure
only one uses the database at any time.

If a thread attempts to use the connection while another is using
 it,
it will wait until the other thread has finished it's current
operation.

If it's a standard SQL statement, then the operation is sending
 the
statement, and retrieving any ResultSet (in full).

If it's a Fastpath call (ie: reading a block from a LargeObject),
 then
it's the time to send, and retrieve that block.

This is fine for applications & applets, but can cause a
 performance
problem with servlets.

With servlets, you can have a heavy load on the connection. If you
have several threads performing queries, then each one will pause,
which may not be what you are after.

To solve this, you would be advised to create a pool of
 Connections.

When ever a thread needs to use the database, it asks a manager
 class
for a Connection. It hands a free connection to the thread, and
 marks
it as busy. If a free connection is not available, it opens one.

Once the thread has finished with it, it returns it to the
 manager,
who can then either close it, or add it to the pool. The manager
 would
also check that the connection is still alive, and remove it from
 the
pool if it's dead.
```

```
So, with servlets, it's up to you to use either a single
 connection,
or a pool. The plus side for a pool is that threads will not be hit
 by
the bottle neck caused by a single network connection. The down
 side,
is that it increases the load on the server, as a backend is
 created
for each Connection.

It's up to you, and your applications requirements.
```

59.12. Further Reading

If you have not yet read it, I'd advise you read the JDBC API Documentation (supplied with Sun's JDK), and the JDBC Specification. Both are available on JavaSoft's web site[2].

My own web site[3] contains updated information not included in this document, and also includes precompiled drivers for v6.4, and earlier.

[2] http://www.javasoft.com
[3] http://www.retep.org.uk

Chapter 60. Lisp Programming Interface

`pg.el` is a socket-level interface to Postgres for emacs.

Author

Written by Eric Marsden[1] on 21 Jul 1999.

`pg.el` is a socket-level interface to Postgres for emacs (text editor extraordinaire). The module is capable of type coercions from a range of SQL types to the equivalent Emacs Lisp type. It currently supports neither crypt or Kerberos authentication, nor large objects.

The code (version 0.2) is available under GNU GPL from Eric Marsden[2]

Changes since last release:

- now works with XEmacs (tested with Emacs 19.34 & 20.2, and XEmacs 20.4)
- added functions to provide database metainformation (list of databases, of tables, of columns)
- arguments to `pg:result' are now :keywords
- MULE-resistant
- more self-testing code

Please note that this is a programmer's API, and doesn't provide any form of user interface. Example:

```
(defun demo ()
   (interactive)
   (let* ((conn (pg:connect "template1" "postgres" "postgres"))
          (res (pg:exec conn "SELECT * from scshdemo WHERE a =
42")))
     (message "status is %s"   (pg:result res :status))
     (message "metadata is %s" (pg:result res :attributes))
     (message "data is %s"     (pg:result res :tuples))
     (pg:disconnect conn)))
```

[1] mailto:emarsden@mail.dotcom.fr
[2] http://www.chez.com/emarsden/downloads/pg.el

Part V. Developer's Guide

The Developer's Guide includes discussion of design decisions and suggestions for future development.

Table of Contents

61. Postgres Source Code 639
    61.1. Formatting 639
62. Overview of PostgreSQL Internals 640
    62.1. The Path of a Query 640
    62.2. How Connections are Established 640
    62.3. The Parser Stage 641
        62.3.1. Parser 641
        62.3.2. Transformation Process 642
    62.4. The Postgres Rule System 643
        62.4.1. The Rewrite System 643
    62.5. Planner/Optimizer 644
        62.5.1. Generating Possible Plans 644
        62.5.2. Data Structure of the Plan 645
    62.6. Executor 645
63. pg_options 647
64. Genetic Query Optimization in Database Systems 650
    64.1. Query Handling as a Complex Optimization Problem 650
    64.2. Genetic Algorithms (GA) 650
    64.3. Genetic Query Optimization (GEQO) in Postgres 651
    64.4. Future Implementation Tasks for Postgres GEQO 652
        64.4.1. Basic Improvements 652
65. Frontend/Backend Protocol 653
    65.1. Overview 653
    65.2. Protocol 653
        65.2.1. Startup 654
        65.2.2. Query 655
        65.2.3. Function Call 656
        65.2.4. Notification Responses 657
        65.2.5. Cancelling Requests in Progress 657
        65.2.6. Termination 657
    65.3. Message Data Types 658
    65.4. Message Formats 658
66. Postgres Signals 665
67. gcc Default Optimizations 667
68. Backend Interface 668
    68.1. BKI File Format 668
    68.2. General Commands 668
    68.3. Macro Commands 669
    68.4. Debugging Commands 669
    68.5. Example 670
69. Page Files 671
    69.1. Page Structure 671
    69.2. Files 672
    69.3. Bugs 672

Chapter 61. Postgres Source Code

61.1. Formatting

Source code formatting uses a 4 column tab spacing, currently with tabs preserved (i.e. tabs are not expanded to spaces).

For emacs, add the following (or something similar) to your `~/.emacs` initialization file:

```
;; check for files with a path containing "postgres" or "pgsql"
(setq auto-mode-alist (cons '("\\(postgres\\|pgsql\\).*\\.[ch]\
\'" . pgsql-c-mode) auto-mode-alist))
(setq auto-mode-alist (cons '("\\(postgres\\|pgsql\\).*\\.cc\\'" .
 pgsql-c-mode) auto-mode-alist))

(defun pgsql-c-mode ()
  ;; sets up formatting for Postgres C code
  (interactive)
  (c-mode)
  (setq-default tab-width 4)
  (c-set-style "bsd")             ; set c-basic-offset to 4, plus
 other stuff
  (c-set-offset 'case-label '+)   ; tweak case indent to match PG
 custom
  (setq indent-tabs-mode t))      ; make sure we keep tabs when
 indenting
```

For vi, your `~/.vimrc` or equivalent file should contain the following:

```
set tabstop=4
```

or equivalently from within vi, try

```
:set ts=4
```

The text browsing tools more and less can be invoked as

```
more -x4
less -x4
```

Chapter 62. Overview of PostgreSQL Internals

Author

This chapter originally appeared as a part of Simkovics, 1998 , Stefan Simkovics' Master's Thesis prepared at Vienna University of Technology under the direction of O.Univ.Prof.Dr. Georg Gottlob and Univ.Ass. Mag. Katrin Seyr.

This chapter gives an overview of the internal structure of the backend of Postgres. After having read the following sections you should have an idea of how a query is processed. Don't expect a detailed description here (I think such a description dealing with all data structures and functions used within Postgres would exceed 1000 pages!). This chapter is intended to help understanding the general control and data flow within the backend from receiving a query to sending the results.

62.1. The Path of a Query

Here we give a short overview of the stages a query has to pass in order to obtain a result.

1. A connection from an application program to the Postgres server has to be established. The application program transmits a query to the server and receives the results sent back by the server.

2. The *parser stage* checks the query transmitted by the application program (client) for correct syntax and creates a *query tree*.

3. The *rewrite system* takes the query tree created by the parser stage and looks for any *rules* (stored in the *system catalogs*) to apply to the *querytree* and performs the transformations given in the *rule bodies*. One application of the rewrite system is given in the realization of *views*.

   Whenever a query against a view (i.e. a *virtual table*) is made, the rewrite system rewrites the user's query to a query that accesses the *base tables* given in the *view definition* instead.

4. The *planner/optimizer* takes the (rewritten) querytree and creates a *queryplan* that will be the input to the *executor*.

   It does so by first creating all possible *paths* leading to the same result. For example if there is an index on a relation to be scanned, there are two paths for the scan. One possibility is a simple sequential scan and the other possibility is to use the index. Next the cost for the execution of each plan is estimated and the cheapest plan is chosen and handed back.

5. The executor recursively steps through the *plan tree* and retrieves tuples in the way represented by the plan. The executor makes use of the *storage system* while scanning relations, performs *sorts* and *joins*, evaluates *qualifications* and finally hands back the tuples derived.

In the following sections we will cover every of the above listed items in more detail to give a better understanding on Postgres's internal control and data structures.

62.2. How Connections are Established

Postgres is implemented using a simple "process per-user" client/server model. In this model there is one *client process* connected to exactly one *server process*. As we don't know *per se* how many connections will be made, we have to use a *master process* that spawns a new server process every time a connection is requested. This master process is called `postmaster` and listens at a specified TCP/IP port for incoming connections. Whenever a request for a connection is detected the `postmaster` process spawns a new server process called `postgres`. The server tasks (`postgres`

processes) communicate with each other using *semaphores* and *shared memory* to ensure data integrity throughout concurrent data access. Figure \ref{connection} illustrates the interaction of the master process `postmaster` the server process `postgres` and a client application.

The client process can either be the psql frontend (for interactive SQL queries) or any user application implemented using the `libpg` library. Note that applications implemented using ecpg (the Postgres embedded SQL preprocessor for C) also use this library.

Once a connection is established the client process can send a query to the *backend* (server). The query is transmitted using plain text, i.e. there is no parsing done in the *frontend* (client). The server parses the query, creates an *execution plan*, executes the plan and returns the retrieved tuples to the client by transmitting them over the established connection.

62.3. The Parser Stage

The *parser stage* consists of two parts:

- The *parser* defined in `gram.y` and `scan.l` is built using the Unix tools yacc and lex.
- The *transformation process* does modifications and augmentations to the data structures returned by the parser.

62.3.1. Parser

The parser has to check the query string (which arrives as plain ASCII text) for valid syntax. If the syntax is correct a *parse tree* is built up and handed back otherwise an error is returned. For the implementation the well known Unix tools lex and yacc are used.

The *lexer* is defined in the file `scan.l` and is responsible for recognizing *identifiers*, the *SQL keywords* etc. For every keyword or identifier that is found, a *token* is generated and handed to the parser.

The parser is defined in the file `gram.y` and consists of a set of *grammar rules* and *actions* that are executed whenever a rule is fired. The code of the actions (which is actually C-code) is used to build up the parse tree.

The file `scan.l` is transformed to the C-source file `scan.c` using the program lex and `gram.y` is transformed to `gram.c` using yacc. After these transformations have taken place a normal C-compiler can be used to create the parser. Never make any changes to the generated C-files as they will be overwritten the next time lex or yacc is called.

Note

The mentioned transformations and compilations are normally done automatically using the *makefiles* shipped with the Postgres source distribution.

A detailed description of yacc or the grammar rules given in `gram.y` would be beyond the scope of this paper. There are many books and documents dealing with lex and yacc. You should be familiar with yacc before you start to study the grammar given in `gram.y` otherwise you won't understand what happens there.

For a better understanding of the data structures used in Postgres for the processing of a query we use an example to illustrate the changes made to these data structures in every stage.

Example 62.1. A Simple Select

This example contains the following simple query that will be used in various descriptions and figures throughout the following sections. The query assumes that the tables given in *The Supplier Database* have already been defined.

```
select s.sname, se.pno
    from supplier s, sells se
    where s.sno > 2 and s.sno = se.sno;
```

Figure \ref{parsetree} shows the *parse tree* built by the grammar rules and actions given in `gram.y` for the query given in

> This example contains the following simple query that will be used in various descriptions and figures throughout the following sections. The query assumes that the tables given in *The Supplier Database* have already been defined.
>
> ```
> select s.sname, se.pno
>
>     from supplier s, sells se
>     where s.sno > 2 and s.sno = se.sno;
> ```

(without the *operator tree* for the *where clause* which is shown in figure \ref{where_clause} because there was not enough space to show both data structures in one figure).

The top node of the tree is a `SelectStmt` node. For every entry appearing in the *from clause* of the SQL query a `RangeVar` node is created holding the name of the *alias* and a pointer to a `RelExpr` node holding the name of the *relation*. All `RangeVar` nodes are collected in a list which is attached to the field `fromClause` of the `SelectStmt` node.

For every entry appearing in the *select list* of the SQL query a `ResTarget` node is created holding a pointer to an `Attr` node. The `Attr` node holds the *relation name* of the entry and a pointer to a `Value` node holding the name of the *attribute*. All `ResTarget` nodes are collected to a list which is connected to the field `targetList` of the `SelectStmt` node.

Figure \ref{where_clause} shows the operator tree built for the where clause of the SQL query given in example

> This example contains the following simple query that will be used in various descriptions and figures throughout the following sections. The query assumes that the tables given in *The Supplier Database* have already been defined.
>
> ```
> select s.sname, se.pno
>
>     from supplier s, sells se
>     where s.sno > 2 and s.sno = se.sno;
> ```

which is attached to the field `qual` of the `SelectStmt` node. The top node of the operator tree is an `A_Expr` node representing an `AND` operation. This node has two successors called `lexpr` and `rexpr` pointing to two *subtrees*. The subtree attached to `lexpr` represents the qualification `s.sno > 2` and the one attached to `rexpr` represents `s.sno = se.sno`. For every attribute an `Attr` node is created holding the name of the relation and a pointer to a `Value` node holding the name of the attribute. For the constant term appearing in the query a `Const` node is created holding the value.

62.3.2. Transformation Process

The *transformation process* takes the tree handed back by the parser as input and steps recursively through it. If a `SelectStmt` node is found, it is transformed to a `Query` node which will be the top most node of the new data structure. Figure \ref{transformed} shows the transformed data structure (the part for the transformed *where clause* is given in figure \ref{transformed_where} because there was not enough space to show all parts in one figure).

Now a check is made, if the *relation names* in the *FROM clause* are known to the system. For every relation name that is present in the *system catalogs* a RTE node is created containing the relation name, the *alias name* and the *relation id*. From now on the relation ids are used to refer to the *relations* given

in the query. All RTE nodes are collected in the *range table entry list* which is connected to the field `rtable` of the `Query` node. If a name of a relation that is not known to the system is detected in the query an error will be returned and the query processing will be aborted.

Next it is checked if the *attribute names* used are contained in the relations given in the query. For every attribute} that is found a TLE node is created holding a pointer to a `Resdom` node (which holds the name of the column) and a pointer to a `VAR` node. There are two important numbers in the `VAR` node. The field `varno` gives the position of the relation containing the current attribute} in the range table entry list created above. The field `varattno` gives the position of the attribute within the relation. If the name of an attribute cannot be found an error will be returned and the query processing will be aborted.

62.4. The Postgres Rule System

Postgres supports a powerful *rule system* for the specification of *views* and ambiguous *view updates*. Originally the Postgres rule system consisted of two implementations:

- The first one worked using *tuple level* processing and was implemented deep in the *executor*. The rule system was called whenever an individual tuple had been accessed. This implementation was removed in 1995 when the last official release of the Postgres project was transformed into Postgres95.
- The second implementation of the rule system is a technique called *query rewriting*. The *rewrite system*} is a module that exists between the *parser stage* and the *planner/optimizer*. This technique is still implemented.

For information on the syntax and creation of rules in the Postgres system refer to *The PostgreSQL User's Guide*.

62.4.1. The Rewrite System

The *query rewrite system* is a module between the parser stage and the planner/optimizer. It processes the tree handed back by the parser stage (which represents a user query) and if there is a rule present that has to be applied to the query it rewrites the tree to an alternate form.

62.4.1.1. Techniques To Implement Views

Now we will sketch the algorithm of the query rewrite system. For better illustration we show how to implement views using rules as an example.

Let the following rule be given:

```
create rule view_rule
as on select
to test_view
do instead
   select s.sname, p.pname
   from supplier s, sells se, part p
   where s.sno = se.sno and
         p.pno = se.pno;
```

The given rule will be *fired* whenever a select against the relation `test_view` is detected. Instead of selecting the tuples from `test_view` the select statement given in the *action part* of the rule is executed.

Let the following user-query against `test_view` be given:

```
select sname
```

```
from test_view
where sname <> 'Smith';
```

Here is a list of the steps performed by the query rewrite system whenever a user-query against `test_view` appears. (The following listing is a very informal description of the algorithm just intended for basic understanding. For a detailed description refer to Stonebraker et al, 1989).

`test_view` Rewrite

1. Take the query given in the action part of the rule.
2. Adapt the targetlist to meet the number and order of attributes given in the user-query.
3. Add the qualification given in the where clause of the user-query to the qualification of the query given in the action part of the rule.

Given the rule definition above, the user-query will be rewritten to the following form (Note that the rewriting is done on the internal representation of the user-query handed back by the parser stage but the derived new data structure will represent the following query):

```
select s.sname
from supplier s, sells se, part p
where s.sno = se.sno and
      p.pno = se.pno and
      s.sname <> 'Smith';
```

62.5. Planner/Optimizer

The task of the *planner/optimizer* is to create an optimal execution plan. It first combines all possible ways of *scanning* and *joining* the relations that appear in a query. All the created paths lead to the same result and it's the task of the optimizer to estimate the cost of executing each path and find out which one is the cheapest.

62.5.1. Generating Possible Plans

The planner/optimizer decides which plans should be generated based upon the types of indices defined on the relations appearing in a query. There is always the possibility of performing a sequential scan on a relation, so a plan using only sequential scans is always created. Assume an index is defined on a relation (for example a B-tree index) and a query contains the restriction `relation.attribute OPR constant`. If `relation.attribute` happens to match the key of the B-tree index and `OPR` is anything but '<>' another plan is created using the B-tree index to scan the relation. If there are further indices present and the restrictions in the query happen to match a key of an index further plans will be considered.

After all feasible plans have been found for scanning single relations, plans for joining relations are created. The planner/optimizer considers only joins between every two relations for which there exists a corresponding join clause (i.e. for which a restriction like `where rel1.attr1=rel2.attr2` exists) in the where qualification. All possible plans are generated for every join pair considered by the planner/optimizer. The three possible join strategies are:

- *nested iteration join*: The right relation is scanned once for every tuple found in the left relation. This strategy is easy to implement but can be very time consuming.
- *merge sort join*: Each relation is sorted on the join attributes before the join starts. Then the two relations are merged together taking into account that both relations are ordered on the join attributes. This kind of join is more attractive because every relation has to be scanned only once.

- *hash join*: the right relation is first hashed on its join attributes. Next the left relation is scanned and the appropriate values of every tuple found are used as hash keys to locate the tuples in the right relation.

62.5.2. Data Structure of the Plan

Here we will give a little description of the nodes appearing in the plan. Figure \ref{plan} shows the plan produced for the query in example \ref{simple_select}.

The top node of the plan is a `MergeJoin` node which has two successors, one attached to the field `lefttree` and the second attached to the field `righttree`. Each of the subnodes represents one relation of the join. As mentioned above a merge sort join requires each relation to be sorted. That's why we find a `Sort` node in each subplan. The additional qualification given in the query (`s.sno > 2`) is pushed down as far as possible and is attached to the `qpqual` field of the leaf `SeqScan` node of the corresponding subplan.

The list attached to the field `mergeclauses` of the `MergeJoin` node contains information about the join attributes. The values `65000` and `65001` for the `varno` fields in the `VAR` nodes appearing in the `mergeclauses` list (and also in the `targetlist`) mean that not the tuples of the current node should be considered but the tuples of the next "deeper" nodes (i.e. the top nodes of the subplans) should be used instead.

Note that every `Sort` and `SeqScan` node appearing in figure \ref{plan} has got a `targetlist` but because there was not enough space only the one for the `MergeJoin` node could be drawn.

Another task performed by the planner/optimizer is fixing the *operator ids* in the `Expr` and `Oper` nodes. As mentioned earlier, Postgres supports a variety of different data types and even user defined types can be used. To be able to maintain the huge amount of functions and operators it is necessary to store them in a system table. Each function and operator gets a unique operator id. According to the types of the attributes used within the qualifications etc., the appropriate operator ids have to be used.

62.6. Executor

The *executor* takes the plan handed back by the planner/optimizer and starts processing the top node. In the case of our example (the query given in example \ref{simple_select}) the top node is a `MergeJoin` node.

Before any merge can be done two tuples have to be fetched (one from each subplan). So the executor recursively calls itself to process the subplans (it starts with the subplan attached to `lefttree`). The new top node (the top node of the left subplan) is a `SeqScan` node and again a tuple has to be fetched before the node itself can be processed. The executor calls itself recursively another time for the subplan attached to `lefttree` of the `SeqScan` node.

Now the new top node is a `Sort` node. As a sort has to be done on the whole relation, the executor starts fetching tuples from the `Sort` node's subplan and sorts them into a temporary relation (in memory or a file) when the `Sort` node is visited for the first time. (Further examinations of the `Sort` node will always return just one tuple from the sorted temporary relation.)

Every time the processing of the `Sort` node needs a new tuple the executor is recursively called for the `SeqScan` node attached as subplan. The relation (internally referenced by the value given in the `scanrelid` field) is scanned for the next tuple. If the tuple satisfies the qualification given by the tree attached to `qpqual` it is handed back, otherwise the next tuple is fetched until the qualification is satisfied. If the last tuple of the relation has been processed a `NULL` pointer is returned.

After a tuple has been handed back by the `lefttree` of the `MergeJoin` the `righttree` is processed in the same way. If both tuples are present the executor processes the `MergeJoin` node. Whenever a new tuple from one of the subplans is needed a recursive call to the executor is performed to obtain it. If a joined tuple could be created it is handed back and one complete processing of the plan tree has finished.

Now the described steps are performed once for every tuple, until a `NULL` pointer is returned for the processing of the `MergeJoin` node, indicating that we are finished.

Chapter 63. pg_options

Massimo Dal Zotto

Note

Contributed by Massimo Dal Zotto[1]

The optional file `data/pg_options` contains runtime options used by the backend to control trace messages and other backend tunable parameters. What makes this file interesting is the fact that it is re-read by a backend when it receives a SIGHUP signal, making thus possible to change run-time options on the fly without needing to restart Postgres. The options specified in this file may be debugging flags used by the trace package (`backend/utils/misc/trace.c`) or numeric parameters which can be used by the backend to control its behaviour. New options and parameters must be defined in `backend/utils/misc/trace.c` and `backend/include/utils/trace.h`.

For example suppose we want to add conditional trace messages and a tunable numeric parameter to the code in file `foo.c`. All we need to do is to add the constant TRACE_FOO and OPT_FOO_PARAM into `backend/include/utils/trace.h`:

```
/* file trace.h */
enum pg_option_enum {
    ...
    TRACE_FOO,    /* trace foo functions */
    OPT_FOO_PARAM,  /* foo tunable parameter */

    NUM_PG_OPTIONS              /* must be the last item of enum */
};
```

and a corresponding line in `backend/utils/misc/trace.c`:

```
/* file trace.c */
static char *opt_names[] = {
    ...
    "foo",              /* trace foo functions */
    "fooparam"           /* foo tunable parameter */
};
```

Options in the two files must be specified in exactly the same order. In the foo source files we can now reference the new flags with:

```
/* file foo.c */
#include "trace.h"
#define foo_param pg_options[OPT_FOO_PARAM]

int
foo_function(int x, int y)
{
    TPRINTF(TRACE_FOO, "entering foo_function, foo_param=%d",
 foo_param);
    if (foo_param > 10) {
        do_more_foo(x, y);
    }
}
```

[1] mailto:dz@cs.unitn.it

Existing files using private trace flags can be changed by simply adding the following code:

```
#include "trace.h"
/* int my_own_flag = 0; -- removed */
#define my_own_flag pg_options[OPT_MY_OWN_FLAG]
```

All pg_options are initialized to zero at backend startup. If we need a different default value we must add some initialization code at the beginning of `PostgresMain`. Now we can set the foo_param and enable foo trace by writing values into the `data/pg_options` file:

```
# file pg_options
...
foo=1
fooparam=17
```

The new options will be read by all new backends when they are started. To make effective the changes for all running backends we need to send a SIGHUP to the postmaster. The signal will be automatically sent to all the backends. We can also activate the changes only for a specific backend by sending the SIGHUP directly to it.

pg_options can also be specified with the `-T` switch of Postgres:

```
postgres options -T "verbose=2,query,hostlookup-"
```

The functions used for printing errors and debug messages can now make use of the *syslog(2)* facility. Message printed to stdout or stderr are prefixed by a timestamp containing also the backend pid:

```
#timestamp          #pid    #message
980127.17:52:14.173 [29271] StartTransactionCommand
980127.17:52:14.174 [29271] ProcessUtility: drop table t;
980127.17:52:14.186 [29271] SIIncNumEntries: table is 70% full
980127.17:52:14.186 [29286] Async_NotifyHandler
980127.17:52:14.186 [29286] Waking up sleeping backend process
980127.19:52:14.292 [29286] Async_NotifyFrontEnd
980127.19:52:14.413 [29286] Async_NotifyFrontEnd done
980127.19:52:14.466 [29286] Async_NotifyHandler done
```

This format improves readability of the logs and allows people to understand exactly which backend is doing what and at which time. It also makes easier to write simple awk or perl scripts which monitor the log to detect database errors or problem, or to compute transaction time statistics.

Messages printed to syslog use the log facility LOG_LOCAL0. The use of syslog can be controlled with the syslog pg_option. Unfortunately many functions call directly `printf()` to print their messages to stdout or stderr and this output can't be redirected to syslog or have timestamps in it. It would be advisable that all calls to printf would be replaced with the PRINTF macro and output to stderr be changed to use EPRINTF instead so that we can control all output in a uniform way.

The new pg_options mechanism is more convenient than defining new backend option switches because:

- we don't have to define a different switch for each thing we want to control. All options are defined as keywords in an external file stored in the data directory.
- we don't have to restart Postgres to change the setting of some option. Normally backend options are specified to the postmaster and passed to each backend when it is started. Now they are read from a file.

- we can change options on the fly while a backend is running. We can thus investigate some problem by activating debug messages only when the problem appears. We can also try different values for tunable parameters.

The format of the `pg_options` file is as follows:

```
# comment
option=integer_value  # set value for option
option                # set option = 1
option+               # set option = 1
option-               # set option = 0
```

Note that *`keyword`* can also be an abbreviation of the option name defined in `backend/utils/misc/trace.c`.

Refer to *The Administrator's Guide* chapter on runtime options for a complete list of currently supported options.

Some of the existing code using private variables and option switches has been changed to make use of the pg_options feature, mainly in `postgres.c`. It would be advisable to modify all existing code in this way, so that we can get rid of many of the switches on the Postgres command line and can have more tunable options with a unique place to put option values.

Chapter 64. Genetic Query Optimization in Database Systems

Martin Utesch, University of Mining and Technology

Author

Written by Martin Utesch[1] for the Institute of Automatic Control at the University of Mining and Technology in Freiberg, Germany.

64.1. Query Handling as a Complex Optimization Problem

Among all relational operators the most difficult one to process and optimize is the *join*. The number of alternative plans to answer a query grows exponentially with the number of `joins` included in it. Further optimization effort is caused by the support of a variety of *join methods* (e.g., nested loop, index scan, merge join in Postgres) to process individual `joins` and a diversity of *indices* (e.g., r-tree, b-tree, hash in Postgres) as access paths for relations.

The current Postgres optimizer implementation performs a *near- exhaustive search* over the space of alternative strategies. This query optimization technique is inadequate to support database application domains that involve the need for extensive queries, such as artificial intelligence.

The Institute of Automatic Control at the University of Mining and Technology, in Freiberg, Germany, encountered the described problems as its folks wanted to take the Postgres DBMS as the backend for a decision support knowledge based system for the maintenance of an electrical power grid. The DBMS needed to handle large `join` queries for the inference machine of the knowledge based system.

Performance difficulties within exploring the space of possible query plans arose the demand for a new optimization technique being developed.

In the following we propose the implementation of a *Genetic Algorithm* as an option for the database query optimization problem.

64.2. Genetic Algorithms (GA)

The GA is a heuristic optimization method which operates through determined, randomized search. The set of possible solutions for the optimization problem is considered as a *population* of *individuals*. The degree of adaption of an individual to its environment is specified by its *fitness*.

The coordinates of an individual in the search space are represented by *chromosomes*, in essence a set of character strings. A *gene* is a subsection of a chromosome which encodes the value of a single parameter being optimized. Typical encodings for a gene could be *binary* or *integer*.

Through simulation of the evolutionary operations *recombination*, *mutation*, and *selection* new generations of search points are found that show a higher average fitness than their ancestors.

According to the "comp.ai.genetic" FAQ it cannot be stressed too strongly that a GA is not a pure random search for a solution to a problem. A GA uses stochastic processes, but the result is distinctly non-random (better than random).

```
Structured Diagram of a GA:
```

[1] utesch@aut.tu-freiberg.de

```
---------------------------

P(t)    generation of ancestors at a time t
P''(t)  generation of descendants at a time t

+=========================================+
|>>>>>>>>>>>  Algorithm GA  <<<<<<<<<<<<<|
+=========================================+
| INITIALIZE t := 0                       |
+=========================================+
| INITIALIZE P(t)                         |
+=========================================+
| evalute FITNESS of P(t)                 |
+=========================================+
| while not STOPPING CRITERION do         |
|   +-------------------------------------+
|   | P'(t)  := RECOMBINATION{P(t)}       |
|   +-------------------------------------+
|   | P''(t) := MUTATION{P'(t)}           |
|   +-------------------------------------+
|   | P(t+1) := SELECTION{P''(t) + P(t)}  |
|   +-------------------------------------+
|   | evalute FITNESS of P''(t)           |
|   +-------------------------------------+
|   | t := t + 1                          |
+===+=====================================+
```

64.3. Genetic Query Optimization (GEQO) in Postgres

The GEQO module is intended for the solution of the query optimization problem similar to a traveling salesman problem (TSP). Possible query plans are encoded as integer strings. Each string represents the `join` order from one relation of the query to the next. E. g., the query tree

```
     /\
    /\ 2
   /\ 3
  4  1
```

is encoded by the integer string '4-1-3-2', which means, first join relation '4' and '1', then '3', and then '2', where 1, 2, 3, 4 are relids in Postgres.

Parts of the GEQO module are adapted from D. Whitley's Genitor algorithm.

Specific characteristics of the GEQO implementation in Postgres are:

- Usage of a *steady state* GA (replacement of the least fit individuals in a population, not whole-generational replacement) allows fast convergence towards improved query plans. This is essential for query handling with reasonable time;
- Usage of *edge recombination crossover* which is especially suited to keep edge losses low for the solution of the TSP by means of a GA;
- Mutation as genetic operator is deprecated so that no repair mechanisms are needed to generate legal TSP tours.

The GEQO module gives the following benefits to the Postgres DBMS compared to the Postgres query optimizer implementation:

- Handling of large `join` queries through non-exhaustive search;

- Improved cost size approximation of query plans since no longer plan merging is needed (the GEQO module evaluates the cost for a query plan as an individual).

64.4. Future Implementation Tasks for Postgres GEQO

64.4.1. Basic Improvements

64.4.1.1. Improve freeing of memory when query is already processed

With large `join` queries the computing time spent for the genetic query optimization seems to be a mere *fraction* of the time Postgres needs for freeing memory via routine `MemoryContextFree`, file `backend/utils/mmgr/mcxt.c`. Debugging showed that it get stucked in a loop of routine `OrderedElemPop`, file `backend/utils/mmgr/oset.c`. The same problems arise with long queries when using the normal Postgres query optimization algorithm.

64.4.1.2. Improve genetic algorithm parameter settings

In file `backend/optimizer/geqo/geqo_params.c`, routines `gimme_pool_size` and `gimme_number_generations`, we have to find a compromise for the parameter settings to satisfy two competing demands:

- Optimality of the query plan
- Computing time

64.4.1.3. Find better solution for integer overflow

In file `backend/optimizer/geqo/geqo_eval.c`, routine `geqo_joinrel_size`, the present hack for MAXINT overflow is to set the Postgres integer value of `rel->size` to its logarithm. Modifications of `Rel` in `backend/nodes/relation.h` will surely have severe impacts on the whole Postgres implementation.

64.4.1.4. Find solution for exhausted memory

Memory exhaustion may occur with more than 10 relations involved in a query. In file `backend/optimizer/geqo/geqo_eval.c`, routine `gimme_tree` is recursively called. Maybe I forgot something to be freed correctly, but I dunno what. Of course the `rel` data structure of the `join` keeps growing and growing the more relations are packed into it. Suggestions are welcome :-(

References

Reference information for GEQ algorithms.

The Hitch-Hiker's Guide to Evolutionary Computation. Jörg Heitkötter and David Beasley. InterNet resource. *The Design and Implementation of the Postgres Query Optimizer*. Z. Fong. University of California, Berkeley Computer Science Department. *Fundamentals of Database Systems*. R. Elmasri and S. Navathe. The Benjamin/Cummings Pub., Inc..

Chapter 65. Frontend/Backend Protocol

Phil Thompson

Note

Written by Phil Thompson[1]. Updates for protocol 2.0 by Tom Lane[2].

Postgres uses a message-based protocol for communication between frontends and backends. The protocol is implemented over TCP/IP and also on Unix sockets. Postgres v6.3 introduced version numbers into the protocol. This was done in such a way as to still allow connections from earlier versions of frontends, but this document does not cover the protocol used by those earlier versions.

This document describes version 2.0 of the protocol, implemented in Postgres v6.4 and later.

Higher level features built on this protocol (for example, how `libpq` passes certain environment variables after the connection is established) are covered elsewhere.

65.1. Overview

The three major components are the frontend (running on the client) and the postmaster and backend (running on the server). The postmaster and backend have different roles but may be implemented by the same executable.

A frontend sends a startup packet to the postmaster. This includes the names of the user and the database the user wants to connect to. The postmaster then uses this, and the information in the pg_hba.conf(5) file to determine what further authentication information it requires the frontend to send (if any) and responds to the frontend accordingly.

The frontend then sends any required authentication information. Once the postmaster validates this it responds to the frontend that it is authenticated and hands over the connection to a backend. The backend then sends a message indicating successful startup (normal case) or failure (for example, an invalid database name).

Subsequent communications are query and result packets exchanged between the frontend and the backend. The postmaster takes no further part in ordinary query/result communication. (However, the postmaster is involved when the frontend wishes to cancel a query currently being executed by its backend. Further details about that appear below.)

When the frontend wishes to disconnect it sends an appropriate packet and closes the connection without waiting for a response for the backend.

Packets are sent as a data stream. The first byte determines what should be expected in the rest of the packet. The exception is packets sent from a frontend to the postmaster, which comprise a packet length then the packet itself. The difference is historical.

65.2. Protocol

This section describes the message flow. There are four different types of flows depending on the state of the connection: startup, query, function call, and termination. There are also special provisions for notification responses and command cancellation, which can occur at any time after the startup phase.

[1] mailto:phil@river-bank.demon.co.uk
[2] mailto:tgl@sss.pgh.pa.us

65.2.1. Startup

Startup is divided into an authentication phase and a backend startup phase.

Initially, the frontend sends a StartupPacket. The postmaster uses this info and the contents of the pg_hba.conf(5) file to determine what authentication method the frontend must use. The postmaster then responds with one of the following messages:

ErrorResponse

The postmaster then immediately closes the connection.

AuthenticationOk

The postmaster then hands over to the backend. The postmaster takes no further part in the communication.

AuthenticationKerberosV4

The frontend must then take part in a Kerberos V4 authentication dialog (not described here) with the postmaster. If this is successful, the postmaster responds with an AuthenticationOk, otherwise it responds with an ErrorResponse.

AuthenticationKerberosV5

The frontend must then take part in a Kerberos V5 authentication dialog (not described here) with the postmaster. If this is successful, the postmaster responds with an AuthenticationOk, otherwise it responds with an ErrorResponse.

AuthenticationUnencryptedPassword

The frontend must then send an UnencryptedPasswordPacket. If this is the correct password, the postmaster responds with an AuthenticationOk, otherwise it responds with an ErrorResponse.

AuthenticationEncryptedPassword

The frontend must then send an EncryptedPasswordPacket. If this is the correct password, the postmaster responds with an AuthenticationOk, otherwise it responds with an ErrorResponse.

If the frontend does not support the authentication method requested by the postmaster, then it should immediately close the connection.

After sending AuthenticationOk, the postmaster attempts to launch a backend process. Since this might fail, or the backend might encounter a failure during startup, the frontend must wait for the backend to acknowledge successful startup. The frontend should send no messages at this point. The possible messages from the backend during this phase are:

BackendKeyData

This message is issued after successful backend startup. It provides secret-key data that the frontend must save if it wants to be able to issue cancel requests later. The frontend should not respond to this message, but should continue listening for a ReadyForQuery message.

ReadyForQuery

Backend startup is successful. The frontend may now issue query or function call messages.

ErrorResponse

Backend startup failed. The connection is closed after sending this message.

NoticeResponse

A warning message has been issued. The frontend should display the message but continue listening for ReadyForQuery or ErrorResponse.

The ReadyForQuery message is the same one that the backend will issue after each query cycle. Depending on the coding needs of the frontend, it is reasonable to consider ReadyForQuery as starting a query cycle (and then BackendKeyData indicates successful conclusion of the startup phase), or to consider ReadyForQuery as ending the startup phase and each subsequent query cycle.

65.2.2. Query

A Query cycle is initiated by the frontend sending a Query message to the backend. The backend then sends one or more response messages depending on the contents of the query command string, and finally a ReadyForQuery response message. ReadyForQuery informs the frontend that it may safely send a new query or function call.

The possible response messages from the backend are:

CompletedResponse

An SQL command completed normally.

CopyInResponse

The backend is ready to copy data from the frontend to a relation. The frontend should then send a CopyDataRows message. The backend will then respond with a CompletedResponse message with a tag of "COPY".

CopyOutResponse

The backend is ready to copy data from a relation to the frontend. It then sends a CopyDataRows message, and then a CompletedResponse message with a tag of "COPY".

CursorResponse

The query was either an insert(l), delete(l), update(l), fetch(l) or a select(l) command. If the transaction has been aborted then the backend sends a CompletedResponse message with a tag of "*ABORT STATE*". Otherwise the following responses are sent.

For an insert(l) command, the backend then sends a CompletedResponse message with a tag of "INSERT *oid rows*" where *rows* is the number of rows inserted, and *oid* is the object ID of the inserted row if *rows* is 1, otherwise *oid* is 0.

For a delete(l) command, the backend then sends a CompletedResponse message with a tag of "DELETE *rows*" where *rows* is the number of rows deleted.

For an update(l) command, the backend then sends a CompletedResponse message with a tag of "UPDATE *rows*" where *rows* is the number of rows deleted.

For a fetch(l) or select(l) command, the backend sends a RowDescription message. This is then followed by an AsciiRow or BinaryRow message (depending on whether a binary cursor was specified) for each row being returned to the frontend. Finally, the backend sends a CompletedResponse message with a tag of "SELECT".

EmptyQueryResponse

An empty query string was recognized. (The need to specially distinguish this case is historical.)

ErrorResponse

An error has occurred.

ReadyForQuery

Processing of the query string is complete. A separate message is sent to indicate this because the query string may contain multiple SQL commands. (CompletedResponse marks the end of processing one SQL command, not the whole string.) ReadyForQuery will always be sent, whether processing terminates successfully or with an error.

NoticeResponse

A warning message has been issued in relation to the query. Notices are in addition to other responses, ie. the backend will continue processing the command.

A frontend must be prepared to accept ErrorResponse and NoticeResponse messages whenever it is expecting any other type of message.

Actually, it is possible for NoticeResponse to arrive even when the frontend is not expecting any kind of message, that is, the backend is nominally idle. (In particular, the backend can be commanded to terminate by its postmaster. In that case it will send a NoticeResponse before closing the connection.) It is recommended that the frontend check for such asynchronous notices just before issuing any new command.

Also, if the frontend issues any listen(l) commands then it must be prepared to accept NotificationResponse messages at any time; see below.

65.2.3. Function Call

A Function Call cycle is initiated by the frontend sending a FunctionCall message to the backend. The backend then sends one or more response messages depending on the results of the function call, and finally a ReadyForQuery response message. ReadyForQuery informs the frontend that it may safely send a new query or function call.

The possible response messages from the backend are:

ErrorResponse

An error has occurred.

FunctionResultResponse

The function call was executed and returned a result.

FunctionVoidResponse

The function call was executed and returned no result.

ReadyForQuery

Processing of the function call is complete. ReadyForQuery will always be sent, whether processing terminates successfully or with an error.

NoticeResponse

A warning message has been issued in relation to the function call. Notices are in addition to other responses, ie. the backend will continue processing the command.

A frontend must be prepared to accept ErrorResponse and NoticeResponse messages whenever it is expecting any other type of message. Also, if it issues any listen(l) commands then it must be prepared to accept NotificationResponse messages at any time; see below.

65.2.4. Notification Responses

If a frontend issues a listen(l) command, then the backend will send a NotificationResponse message (not to be confused with NoticeResponse!) whenever a notify(l) command is executed for the same notification name.

Notification responses are permitted at any point in the protocol (after startup), except within another backend message. Thus, the frontend must be prepared to recognize a NotificationResponse message whenever it is expecting any message. Indeed, it should be able to handle NotificationResponse messages even when it is not engaged in a query.

NotificationResponse

A notify(l) command has been executed for a name for which a previous listen(l) command was executed. Notifications may be sent at any time.

It may be worth pointing out that the names used in listen and notify commands need not have anything to do with names of relations (tables) in the SQL database. Notification names are simply arbitrarily chosen condition names.

65.2.5. Cancelling Requests in Progress

During the processing of a query, the frontend may request cancellation of the query by sending an appropriate request to the postmaster. The cancel request is not sent directly to the backend for reasons of implementation efficiency: we don't want to have the backend constantly checking for new input from the frontend during query processing. Cancel requests should be relatively infrequent, so we make them slightly cumbersome in order to avoid a penalty in the normal case.

To issue a cancel request, the frontend opens a new connection to the postmaster and sends a CancelRequest message, rather than the StartupPacket message that would ordinarily be sent across a new connection. The postmaster will process this request and then close the connection. For security reasons, no direct reply is made to the cancel request message.

A CancelRequest message will be ignored unless it contains the same key data (PID and secret key) passed to the frontend during connection startup. If the request matches the PID and secret key for a currently executing backend, the postmaster signals the backend to abort processing of the current query.

The cancellation signal may or may not have any effect --- for example, if it arrives after the backend has finished processing the query, then it will have no effect. If the cancellation is effective, it results in the current command being terminated early with an error message.

The upshot of all this is that for reasons of both security and efficiency, the frontend has no direct way to tell whether a cancel request has succeeded. It must continue to wait for the backend to respond to the query. Issuing a cancel simply improves the odds that the current query will finish soon, and improves the odds that it will fail with an error message instead of succeeding.

Since the cancel request is sent to the postmaster and not across the regular frontend/backend communication link, it is possible for the cancel request to be issued by any process, not just the frontend whose query is to be canceled. This may have some benefits of flexibility in building multiple-process applications. It also introduces a security risk, in that unauthorized persons might try to cancel queries. The security risk is addressed by requiring a dynamically generated secret key to be supplied in cancel requests.

65.2.6. Termination

The normal, graceful termination procedure is that the frontend sends a Terminate message and immediately closes the connection. On receipt of the message, the backend immediately closes the connection and terminates.

An ungraceful termination may occur due to software failure (i.e., core dump) at either end. If either frontend or backend sees an unexpected closure of the connection, it should clean up and terminate. The frontend has the option of launching a new backend by recontacting the postmaster, if it doesn't want to terminate itself.

65.3. Message Data Types

This section describes the base data types used in messages.

Int*n*(*i*)

An *n* bit integer in network byte order. If *i* is specified it is the literal value. Eg. Int16, Int32(42).

LimString*n*(*s*)

A character array of exactly *n* bytes interpreted as a '\0' terminated string. The '\0' is omitted if there is insufficient room. If *s* is specified it is the literal value. Eg. LimString32, LimString64("user").

String(*s*)

A conventional C '\0' terminated string with no length limitation. If *s* is specified it is the literal value. Eg. String, String("user").

Note

There is no predefined limit on the length of a string that can be returned by the backend. Good coding strategy for a frontend is to use an expandable buffer so that anything that fits in memory can be accepted. If that's not feasible, read the full string and discard trailing characters that don't fit into your fixed-size buffer.

Byte*n*(*c*)

Exactly *n* bytes. If *c* is specified it is the literal value. Eg. Byte, Byte1('\n').

65.4. Message Formats

This section describes the detailed format of each message. Each can be sent by either a frontend (F), a postmaster/backend (B), or both (F & B).

AsciiRow (B)

Byte1('D')

Identifies the message as an ASCII data row. (A prior RowDescription message defines the number of fields in the row and their data types.)

Byte*n*

A bit map with one bit for each field in the row. The 1st field corresponds to bit 7 (MSB) of the 1st byte, the 2nd field corresponds to bit 6 of the 1st byte, the 8th field corresponds to bit 0 (LSB) of the 1st byte, the 9th field corresponds to bit 7 of the 2nd byte, and so on. Each bit is set if the value of the corresponding field is not NULL. If the number of fields is not a multiple of 8, the remainder of the last byte in the bit map is wasted.

Then, for each field with a non-NULL value, there is the following:

Int32

Specifies the size of the value of the field, including this size.

Byte*n*

Specifies the value of the field itself in ASCII characters. *n* is the above size minus 4. There is no trailing '\0' in the field data; the front end must add one if it wants one.

AuthenticationOk (B)

Byte1('R')

Identifies the message as an authentication request.

Int32(0)

Specifies that the authentication was successful.

AuthenticationKerberosV4 (B)

Byte1('R')

Identifies the message as an authentication request.

Int32(1)

Specifies that Kerberos V4 authentication is required.

AuthenticationKerberosV5 (B)

Byte1('R')

Identifies the message as an authentication request.

Int32(2)

Specifies that Kerberos V5 authentication is required.

AuthenticationUnencryptedPassword (B)

Byte1('R')

Identifies the message as an authentication request.

Int32(3)

Specifies that an unencrypted password is required.

AuthenticationEncryptedPassword (B)

Byte1('R')

Identifies the message as an authentication request.

Int32(4)

Specifies that an encrypted password is required.

Byte2

The salt to use when encrypting the password.

BackendKeyData (B)

Byte1('K')

Identifies the message as cancellation key data. The frontend must save these values if it wishes to be able to issue CancelRequest messages later.

Int32

The process ID of this backend.

Int32

The secret key of this backend.

BinaryRow (B)

Byte1('B')

Identifies the message as a binary data row. (A prior RowDescription message defines the number of fields in the row and their data types.)

Byte*n*

A bit map with one bit for each field in the row. The 1st field corresponds to bit 7 (MSB) of the 1st byte, the 2nd field corresponds to bit 6 of the 1st byte, the 8th field corresponds to bit 0 (LSB) of the 1st byte, the 9th field corresponds to bit 7 of the 2nd byte, and so on. Each bit is set if the value of the corresponding field is not NULL. If the number of fields is not a multiple of 8, the remainder of the last byte in the bit map is wasted.

Then, for each field with a non-NULL value, there is the following:

Int32

Specifies the size of the value of the field, excluding this size.

Byte*n*

Specifies the value of the field itself in binary format. *n* is the above size.

CancelRequest (F)

Int32(16)

The size of the packet in bytes.

Int32(80877102)

The cancel request code. The value is chosen to contain "1234" in the most significant 16 bits, and "5678" in the least 16 significant bits. (To avoid confusion, this code must not be the same as any protocol version number.)

Int32

The process ID of the target backend.

Int32

The secret key for the target backend.

CompletedResponse (B)

Byte1('C')

Identifies the message as a completed response.

String

The command tag. This is usually (but not always) a single word that identifies which SQL command was completed.

CopyDataRows (B & F)

This is a stream of rows where each row is terminated by a Byte1('\n'). This is then followed by the sequence Byte1('\\'), Byte1('.'), Byte1('\n').

CopyInResponse (B)

Byte1('G')

Identifies the message as a Start Copy In response. The frontend must now send a CopyDataRows message.

CopyOutResponse (B)

Byte1('H')

Identifies the message as a Start Copy Out response. This message will be followed by a CopyDataRows message.

CursorResponse (B)

Byte1('P')

Identifies the message as a cursor response.

String

The name of the cursor. This will be "blank" if the cursor is implicit.

EmptyQueryResponse (B)

Byte1('I')

Identifies the message as a response to an empty query string.

String("")

Unused.

EncryptedPasswordPacket (F)

Int32

The size of the packet in bytes.

String

The encrypted (using crypt()) password.

ErrorResponse (B)

Byte1('E')

Identifies the message as an error.

String

The error message itself.

FunctionCall (F)

Byte1('F')

Identifies the message as a function call.

String("")

Unused.

Int32

Specifies the object ID of the function to call.

Int32

Specifies the number of arguments being supplied to the function.

Then, for each argument, there is the following:

Int32

Specifies the size of the value of the argument, excluding this size.

Byte*n*

Specifies the value of the field itself in binary format. *n* is the above size.

FunctionResultResponse (B)

Byte1('V')

Identifies the message as a function call result.

Byte1('G')

Specifies that a nonempty result was returned.

Int32

Specifies the size of the value of the result, excluding this size.

Byte*n*

Specifies the value of the result itself in binary format. *n* is the above size.

Byte1('0')

Unused. (Strictly speaking, FunctionResultResponse and FunctionVoidResponse are the same thing but with some optional parts to the message.)

FunctionVoidResponse (B)

Byte1('V')

Identifies the message as a function call result.

Byte1('0')

Specifies that an empty result was returned.

NoticeResponse (B)

Byte1('N')

Identifies the message as a notice.

String

The notice message itself.

NotificationResponse (B)

Byte1('A')

Identifies the message as a notification response.

Int32

The process ID of the notifying backend process.

String

The name of the condition that the notify has been raised on.

Query (F)

Byte1('Q')

Identifies the message as a query.

String

The query string itself.

ReadyForQuery (B)

Byte1('Z')

Identifies the message type. ReadyForQuery is sent whenever the backend is ready for a new query cycle.

RowDescription (B)

Byte1('T')

Identifies the message as a row description.

Int16

Specifies the number of fields in a row (may be zero).

Then, for each field, there is the following:

String

Specifies the field name.

Int32

Specifies the object ID of the field type.

Int16

Specifies the type size.

Int32

Specifies the type modifier.

StartupPacket (F)

Int32(296)

The size of the packet in bytes.

Int32

The protocol version number. The most significant 16 bits are the major version number. The least 16 significant bits are the minor version number.

LimString64

The database name, defaults to the user name if empty.

LimString32

The user name.

LimString64

Any additional command line arguments to be passed to the backend by the postmaster.

LimString64

Unused.

LimString64

The optional tty the backend should use for debugging messages.

Terminate (F)

Byte1('X')

Identifies the message as a termination.

UnencryptedPasswordPacket (F)

Int32

The size of the packet in bytes.

String

The unencrypted password.

Chapter 66. Postgres Signals

Massimo Dal Zotto

Note

Contributed by Massimo Dal Zotto[1]

Postgres uses the following signals for communication between the postmaster and backends:

Table 66.1. Postgres Signals

Signal	postmaster Action	Server Action
SIGHUP	kill(*,sighup)	read_pg_options
SIGINT	die	cancel query
SIGQUIT	kill(*,sigterm)	handle_warn
SIGTERM	kill(*,sigterm), kill(*,9), die	die
SIGPIPE	ignored	die
SIGUSR1	kill(*,sigusr1), die	quickdie
SIGUSR2	kill(*,sigusr2)	async notify (SI flush)
SIGCHLD	reaper	ignored (alive test)
SIGTTIN	ignored	
SIGTTOU	ignored	
SIGCONT	dumpstatus	
SIGFPE		FloatExceptionHandler

Note

"`kill(*,signal)`" means sending a signal to all backends.

The main changes to the old signal handling are the use of SIGQUIT instead of SIGHUP to handle warns, SIGHUP to re-read the pg_options file and the redirection to all active backends of SIGHUP, SIGTERM, SIGUSR1 and SIGUSR2 sent to the postmaster. In this way these signals sent to the postmaster can be sent automatically to all the backends without need to know their pids. To shut down postgres one needs only to send a SIGTERM to postmaster and it will stop automatically all the backends.

The SIGUSR2 signal is also used to prevent SI cache table overflow which happens when some backend doesn't process SI cache for a long period. When a backend detects the SI table full at 70% it simply sends a signal to the postmaster which will wake up all idle backends and make them flush the cache.

The typical use of signals by programmers could be the following:

```
# stop postgres
kill -TERM $postmaster_pid

# kill all the backends
kill -QUIT $postmaster_pid

# kill only the postmaster
```

[1] mailto:dz@cs.unitn.it

```
kill -INT $postmaster_pid

# change pg_options
cat new_pg_options > $DATA_DIR/pg_options
kill -HUP $postmaster_pid

# change pg_options only for a backend
cat new_pg_options > $DATA_DIR/pg_options
kill -HUP $backend_pid
cat old_pg_options > $DATA_DIR/pg_options
```

Chapter 67. gcc Default Optimizations

Brian Gallew

Note

Contributed by Brian Gallew[1]

Configuring gcc to use certain flags by default is a simple matter of editing the `/usr/local/lib/gcc-lib/`*`platform`*`/`*`version`*`/specs` file. The format of this file pretty simple. The file is broken into sections, each of which is three lines long. The first line is "*`section_name`*:" (e.g. "*asm:"). The second line is a list of flags, and the third line is blank.

The easiest change to make is to append the desired default flags to the list in the appropriate section. As an example, let's suppose that I have linux running on a '486 with gcc 2.7.2 installed in the default location. In the file /usr/local/lib/gcc-lib/i486-linux/2.7.2/specs, 13 lines down I find the following section:

```
- ----------SECTION----------
*cc1:


- ----------SECTION----------
```

As you can see, there aren't any default flags. If I always wanted compiles of C code to use "-m486 -fomit-frame-pointer", I would change it to look like:

```
- ----------SECTION----------
*cc1:
- -m486 -fomit-frame-pointer

- ----------SECTION----------
```

If I wanted to be able to generate 386 code for another, older linux box lying around, I'd have to make it look like this:

```
- ----------SECTION----------
*cc1:
%{!m386:-m486} -fomit-frame-pointer

- ----------SECTION----------
```

This will always omit frame pointers, any will build 486-optimized code unless -m386 is specified on the command line.

You can actually do quite a lot of customization with the specs file. Always remember, however, that these changes are global, and affect all users of the system.

[1] mailto:geek+@cmu.edu

Chapter 68. Backend Interface

Backend Interface (BKI) files are scripts that are input to the Postgres backend running in the special "bootstrap" mode that allows it to perform database functions without a database system already existing. BKI files can therefore be used to create the database system in the first place. initdb uses BKI files to do just that: to create a database system. However, initdb's BKI files are generated internally. It generates them using the files `global1.bki.source` and `local1.template1.bki.source`, which it finds in the Postgres "library" directory. They get installed there as part of installing Postgres. These .source files get build as part of the Postgres build process, by a build program called genbki. genbki takes as input Postgres source files that double as genbki input that builds tables and C header files that describe those tables.

Related information may be found in documentation for initdb, createdb, and the SQL command `CREATE DATABASE`.

68.1. BKI File Format

The Postgres backend interprets BKI files as described below. This description will be easier to understand if the `global1.bki.source` file is at hand as an example. (As explained above, this .source file isn't quite a BKI file, but you'll be able to guess what the resulting BKI file would be anyway).

Commands are composed of a command name followed by space separated arguments. Arguments to a command which begin with a "$" are treated specially. If "$$" are the first two characters, then the first "$" is ignored and the argument is then processed normally. If the "$" is followed by space, then it is treated as a NULL value. Otherwise, the characters following the "$" are interpreted as the name of a macro causing the argument to be replaced with the macro's value. It is an error for this macro to be undefined.

Macros are defined using

```
define macro macro_name = macro_value
```

and are undefined using

```
undefine macro macro_name
```

and redefined using the same syntax as define.

Lists of general commands and macro commands follow.

68.2. General Commands

OPEN *classname*

Open the class called *classname* for further manipulation.

CLOSE [*classname*]

Close the open class called *classname*. It is an error if *classname* is not already opened. If no *classname* is given, then the currently open class is closed.

PRINT

Print the currently open class.

INSERT [OID=*oid_value*] (*value1 value2* ...)

Insert a new instance to the open class using *value1*, *value2*, etc., for its attribute values and *oid_value* for its OID. If *oid_value* is not zero (0), then this value will be used as the instance's object identifier. Otherwise, it is an error.

INSERT (*value1 value2* ...)

As above, but the system generates a unique object identifier.

CREATE *classname* (*name1* = *type1* [,*name2* = *type2*[,...]])

Create a class named *classname* with the attributes given in parentheses.

OPEN (*name1* = *type1* [,*name2* = *type2*[,...]]) AS *classname*

Open a class named *classname* for writing but do not record its existence in the system catalogs. (This is primarily to aid in bootstrapping.)

DESTROY *classname*

Destroy the class named *classname*.

DEFINE INDEX *indexname* ON *class_name* USING *amname* (*opclass attr* | (*function*(*attr*))

Create an index named *indexname* on the class named *classname* using the *amname* access method. The fields to index are called *name1*, *name2* etc., and the operator collections to use are *collection_1*, *collection_2* etc., respectively.

Note

This last sentence doesn't reference anything in the example. Should be changed to make sense. - Thomas 1998-08-04

68.3. Macro Commands

DEFINE FUNCTION *macro_name* AS *rettype function_name*(*args*)

Define a function prototype for a function named *macro_name* which has its value of type *rettype* computed from the execution *function_name* with the arguments *args* declared in a C-like manner.

DEFINE MACRO *macro_name* FROM FILE *filename*

Define a macro named *macro_name* which has its value read from the file called *filename*.

68.4. Debugging Commands

Note

This section on debugging commands was commented-out in the original documentation. Thomas 1998-08-05

r

Randomly print the open class.

m -l

Toggle display of time information.

m 0

Set retrievals to now.

m 1 Jan 1 01:00:00 1988

Set retrievals to snapshots of the specfied time.

m 2 Jan 1 01:00:00 1988, Feb 1 01:00:00 1988

Set retrievals to ranges of the specified times. Either time may be replaced with space if an unbounded time range is desired.

&A *classname natts name1 type1 name2 type2 ...*

Add *natts* attributes named *name1*, *name2*, etc. of types *type1*, *type2*, etc. to the class *classname*.

&RR *oldclassname newclassname*

Rename the *oldclassname* class to *newclassname*.

&RA classname oldattname newattname *classname oldattname newattname*

Rename the *oldattname* attribute in the class named *classname* to *newattname*.

68.5. Example

The following set of commands will create the pg_opclass class containing the *int_ops* collection as an object with an OID of *421*, print out the class, and then close it.

```
create pg_opclass (opcname=name)
open pg_opclass
insert oid=421 (int_ops)
print
close pg_opclass
```

Chapter 69. Page Files

A description of the database file default page format.

This section provides an overview of the page format used by Postgres classes. User-defined access methods need not use this page format.

In the following explanation, a *byte* is assumed to contain 8 bits. In addition, the term *item* refers to data which is stored in Postgres classes.

69.1. Page Structure

The following table shows how pages in both normal Postgres classes and Postgres index classes (e.g., a B-tree index) are structured.

Table 69.1. Sample Page Layout

Item	Description
itemPointerData	
filler	
itemData...	
Unallocated Space	
ItemContinuationData	
Special Space	
``ItemData 2''	
``ItemData 1''	
ItemIdData	
PageHeaderData	

The first 8 bytes of each page consists of a page header (PageHeaderData). Within the header, the first three 2-byte integer fields (*lower*, *upper*, and *special*) represent byte offsets to the start of unallocated space, to the end of unallocated space, and to the start of *special space*. Special space is a region at the end of the page which is allocated at page initialization time and which contains information specific to an access method. The last 2 bytes of the page header, *opaque*, encode the page size and information on the internal fragmentation of the page. Page size is stored in each page because frames in the buffer pool may be subdivided into equal sized pages on a frame by frame basis within a class. The internal fragmentation information is used to aid in determining when page reorganization should occur.

Following the page header are item identifiers (*ItemIdData*). New item identifiers are allocated from the first four bytes of unallocated space. Because an item identifier is never moved until it is freed, its index may be used to indicate the location of an item on a page. In fact, every pointer to an item (*ItemPointer*) created by Postgres consists of a frame number and an index of an item identifier. An item identifier contains a byte-offset to the start of an item, its length in bytes, and a set of attribute bits which affect its interpretation.

The items themselves are stored in space allocated backwards from the end of unallocated space. Usually, the items are not interpreted. However when the item is too long to be placed on a single page or when fragmentation of the item is desired, the item is divided and each piece is handled as distinct items in the following manner. The first through the next to last piece are placed in an item continuation structure (*ItemContinuationData*). This structure contains itemPointerData which points to the next piece and the piece itself. The last piece is handled normally.

69.2. Files

`data/`

Location of shared (global) database files.

`data/base/`

Location of local database files.

69.3. Bugs

The page format may change in the future to provide more efficient access to large objects.

This section contains insufficient detail to be of any assistance in writing a new access method.

Part VI. Tutorial

Introduction for new users.

Table of Contents

70. SQL 675
70.1. The Relational Data Model 675
70.2. Relational Data Model Formalities 676
70.2.1. Domains vs. Data Types 677
70.3. Operations in the Relational Data Model 677
70.3.1. Relational Algebra 677
70.3.2. Relational Calculus 679
70.3.3. Tuple Relational Calculus 679
70.3.4. Relational Algebra vs. Relational Calculus 680
70.4. The SQL Language 680
70.4.1. Select 680
70.4.2. Data Definition 687
70.4.3. Data Manipulation 689
70.4.4. System Catalogs 690
70.4.5. Embedded SQL 690
71. Architecture 692
71.1. Postgres Architectural Concepts 692
72. Getting Started 694
72.1. Setting Up Your Environment 694
72.2. Starting the Interactive Monitor (psql) 694
72.3. Managing a Database 695
72.3.1. Creating a Database 695
72.3.2. Accessing a Database 696
72.3.3. Destroying a Database 697
73. The Query Language 698
73.1. Interactive Monitor 698
73.2. Concepts 698
73.3. Creating a New Class 698
73.4. Populating a Class with Instances 699
73.5. Querying a Class 699
73.6. Redirecting SELECT Queries 700
73.7. Joins Between Classes 700
73.8. Updates 701
73.9. Deletions 701
73.10. Using Aggregate Functions 701
74. Advanced Postgres SQL Features 703
74.1. Inheritance 703
74.2. Non-Atomic Values 704
74.2.1. Arrays 704
74.3. More Advanced Features 705

Chapter 70. SQL

This chapter introduces the mathematical concepts behind relational databases. It is not required reading, so if you bog down or want to get straight to some simple examples feel free to jump ahead to the next chapter and come back when you have more time and patience. This stuff is supposed to be fun!

This material originally appeared as a part of Stefan Simkovics' Master's Thesis (Simkovics, 1998).

SQL has become the most popular relational query language. The name "SQL" is an abbreviation for *Structured Query Language*. In 1974 Donald Chamberlin and others defined the language SEQUEL (*Structured English Query Language*) at IBM Research. This language was first implemented in an IBM prototype called SEQUEL-XRM in 1974-75. In 1976-77 a revised version of SEQUEL called SEQUEL/2 was defined and the name was changed to SQL subsequently.

A new prototype called System R was developed by IBM in 1977. System R implemented a large subset of SEQUEL/2 (now SQL) and a number of changes were made to SQL during the project. System R was installed in a number of user sites, both internal IBM sites and also some selected customer sites. Thanks to the success and acceptance of System R at those user sites IBM started to develop commercial products that implemented the SQL language based on the System R technology.

Over the next years IBM and also a number of other vendors announced SQL products such as SQL/DS (IBM), DB2 (IBM), ORACLE (Oracle Corp.), DG/SQL (Data General Corp.), and SYBASE (Sybase Inc.).

SQL is also an official standard now. In 1982 the American National Standards Institute (ANSI) chartered its Database Committee X3H2 to develop a proposal for a standard relational language. This proposal was ratified in 1986 and consisted essentially of the IBM dialect of SQL. In 1987 this ANSI standard was also accepted as an international standard by the International Organization for Standardization (ISO). This original standard version of SQL is often referred to, informally, as "SQL/86". In 1989 the original standard was extended and this new standard is often, again informally, referred to as "SQL/89". Also in 1989, a related standard called *Database Language Embedded SQL* (ESQL) was developed.

The ISO and ANSI committees have been working for many years on the definition of a greatly expanded version of the original standard, referred to informally as *SQL2* or *SQL/92*. This version became a ratified standard - "International Standard ISO/IEC 9075:1992, Database Language SQL" - in late 1992. SQL/92 is the version normally meant when people refer to "the SQL standard". A detailed description of SQL/92 is given in Date and Darwen, 1997 . At the time of writing this document a new standard informally referred to as *SQL3* is under development. It is planned to make SQL a Turing-complete language, i.e. all computable queries (e.g. recursive queries) will be possible. This is a very complex task and therefore the completion of the new standard can not be expected before 1999.

70.1. The Relational Data Model

As mentioned before, SQL is a relational language. That means it is based on the *relational data model* first published by E.F. Codd in 1970. We will give a formal description of the relational model later (in Relational Data Model Formalities) but first we want to have a look at it from a more intuitive point of view.

A *relational database* is a database that is perceived by its users as a *collection of tables* (and nothing else but tables). A table consists of rows and columns where each row represents a record and each column represents an attribute of the records contained in the table. The Suppliers and Parts Database shows an example of a database consisting of three tables:

- SUPPLIER is a table storing the number (SNO), the name (SNAME) and the city (CITY) of a supplier.

- PART is a table storing the number (PNO) the name (PNAME) and the price (PRICE) of a part.
- SELLS stores information about which part (PNO) is sold by which supplier (SNO). It serves in a sense to connect the other two tables together.

Example 70.1. The Suppliers and Parts Database

```
SUPPLIER:                    SELLS:
 SNO |  SNAME  |  CITY        SNO | PNO
-----+---------+--------     -----+-----
 1   |  Smith  | London        1  |  1
 2   |  Jones  | Paris         1  |  2
 3   |  Adams  | Vienna        2  |  4
 4   |  Blake  | Rome          3  |  1
                               3  |  3
                               4  |  2
PART:                          4  |  3
 PNO |  PNAME  |  PRICE        4  |  4
-----+---------+---------
 1   |  Screw  |   10
 2   |  Nut    |    8
 3   |  Bolt   |   15
 4   |  Cam    |   25
```

The tables PART and SUPPLIER may be regarded as *entities* and SELLS may be regarded as a *relationship* between a particular part and a particular supplier.

As we will see later, SQL operates on tables like the ones just defined but before that we will study the theory of the relational model.

70.2. Relational Data Model Formalities

The mathematical concept underlying the relational model is the set-theoretic *relation* which is a subset of the Cartesian product of a list of domains. This set-theoretic relation gives the model its name (do not confuse it with the relationship from the *Entity-Relationship model*). Formally a domain is simply a set of values. For example the set of integers is a domain. Also the set of character strings of length 20 and the real numbers are examples of domains.

The *Cartesian product* of domains D_1, D_2, ... D_k, written $D_1 \times D_2 \times ... \times D_k$ is the set of all k-tuples v_1, v_2, ... v_k, such that $v_1 \in D_1$, $v_1 \in D_1$, ... $v_k \in D_k$.

For example, when we have k=2, D_1=`{0,1}` and D_2=`{a,b,c}` then $D_1 \times D_2$ is `{(0,a),(0,b),(0,c),(1,a),(1,b),(1,c)}`.

A Relation is any subset of the Cartesian product of one or more domains: $R \subseteq D_1 \times D_2 \times ... \times D_k$.

For example `{(0,a),(0,b),(1,a)}` is a relation; it is in fact a subset of $D_1 \times D_2$ mentioned above.

The members of a relation are called tuples. Each relation of some Cartesian product $D_1 \times D_2 \times ... \times D_k$ is said to have arity `k` and is therefore a set of `k`-tuples.

A relation can be viewed as a table (as we already did, remember The Suppliers and Parts Database where every tuple is represented by a row and every column corresponds to one component of a tuple. Giving names (called attributes) to the columns leads to the definition of a *relation scheme*.

A *relation scheme* `R` is a finite set of attributes A_1, A_2, ... A_k. There is a domain D_i, for each attribute A_i, 1 <= `i` <= `k`, where the values of the attributes are taken from. We often write a relation scheme as `R(`A_1`, `A_2`, ... `A_k`)`.

Note

A *relation scheme* is just a kind of template whereas a *relation* is an instance of a *relation scheme*. The relation consists of tuples (and can therefore be viewed as a table); not so the relation scheme.

70.2.1. Domains vs. Data Types

We often talked about *domains* in the last section. Recall that a domain is, formally, just a set of values (e.g., the set of integers or the real numbers). In terms of database systems we often talk of *data types* instead of domains. When we define a table we have to make a decision about which attributes to include. Additionally we have to decide which kind of data is going to be stored as attribute values. For example the values of `SNAME` from the table `SUPPLIER` will be character strings, whereas `SNO` will store integers. We define this by assigning a data type to each attribute. The type of `SNAME` will be `VARCHAR(20)` (this is the SQL type for character strings of length <= 20), the type of `SNO` will be `INTEGER`. With the assignment of a data type we also have selected a domain for an attribute. The domain of `SNAME` is the set of all character strings of length <= 20, the domain of `SNO` is the set of all integer numbers.

70.3. Operations in the Relational Data Model

In the previous section (Relational Data Model Formalities) we defined the mathematical notion of the relational model. Now we know how the data can be stored using a relational data model but we do not know what to do with all these tables to retrieve something from the database yet. For example somebody could ask for the names of all suppliers that sell the part 'Screw'. Therefore two rather different kinds of notations for expressing operations on relations have been defined:

- The *Relational Algebra* which is an algebraic notation, where queries are expressed by applying specialized operators to the relations.
- The *Relational Calculus* which is a logical notation, where queries are expressed by formulating some logical restrictions that the tuples in the answer must satisfy.

70.3.1. Relational Algebra

The *Relational Algebra* was introduced by E. F. Codd in 1972. It consists of a set of operations on relations:

- SELECT (σ): extracts *tuples* from a relation that satisfy a given restriction. Let *R* be a table that contains an attribute *A*. $\sigma_{A=a}(R) = \{t \in R \mid t(A) = a\}$ where `t` denotes a tuple of *R* and `t(A)` denotes the value of attribute *A* of tuple `t`.
- PROJECT (π): extracts specified *attributes* (columns) from a relation. Let `R` be a relation that contains an attribute `X`. $\pi_X(R) = \{t(X) \mid t \in R\}$, where `t(X)` denotes the value of attribute `X` of tuple `t`.
- PRODUCT ($\times$): builds the Cartesian product of two relations. Let `R` be a table with arity k_1 and let `S` be a table with arity k_2. $R \times S$ is the set of all $k_1 + k_2$-tuples whose first k_1 components form a tuple in `R` and whose last k_2 components form a tuple in `S`.
- UNION ($\cup$): builds the set-theoretic union of two tables. Given the tables `R` and `S` (both must have the same arity), the union $R \cup S$ is the set of tuples that are in `R` or `S` or both.
- INTERSECT ($\cap$): builds the set-theoretic intersection of two tables. Given the tables `R` and `S`, $R \cap S$ is the set of tuples that are in `R` and in `S`. We again require that `R` and `S` have the same arity.
- DIFFERENCE ($-$ or $\setminus$): builds the set difference of two tables. Let `R` and `S` again be two tables with the same arity. `R - S` is the set of tuples in `R` but not in `S`.

- JOIN (∏): connects two tables by their common attributes. Let `R` be a table with the attributes `A,B` and `C` and let `S` be a table with the attributes `C,D` and `E`. There is one attribute common to both relations, the attribute `C`. R ∏ S = $\pi_{R.A,R.B,R.C,S.D,S.E}(\sigma_{R.C=S.C}(R \times S))$. What are we doing here? We first calculate the Cartesian product `R × S`. Then we select those tuples whose values for the common attribute `C` are equal ($\sigma_{R.C = S.C}$). Now we have a table that contains the attribute `C` two times and we correct this by projecting out the duplicate column.

Example 70.2. An Inner Join

Let's have a look at the tables that are produced by evaluating the steps necessary for a join. Let the following two tables be given:

```
R:                  S:
 A | B | C           C | D | E
---+---+---         ---+---+---
 1 | 2 | 3           3 | a | b
 4 | 5 | 6           6 | c | d
 7 | 8 | 9
```

First we calculate the Cartesian product `R × S` and get:

```
R x S:
 A | B | R.C | S.C | D | E
---+---+-----+-----+---+---
 1 | 2 |  3  |  3  | a | b
 1 | 2 |  3  |  6  | c | d
 4 | 5 |  6  |  3  | a | b
 4 | 5 |  6  |  6  | c | d
 7 | 8 |  9  |  3  | a | b
 7 | 8 |  9  |  6  | c | d
```

After the selection $\sigma_{R.C=S.C}(R \times S)$ we get:

```
 A | B | R.C | S.C | D | E
---+---+-----+-----+---+---
 1 | 2 |  3  |  3  | a | b
 4 | 5 |  6  |  6  | c | d
```

To remove the duplicate column `S.C` we project it out by the following operation: $\pi_{R.A,R.B,R.C,S.D,S.E}(\sigma_{R.C=S.C}(R \times S))$ and get:

```
 A | B | C | D | E
---+---+---+---+---
 1 | 2 | 3 | a | b
 4 | 5 | 6 | c | d
```

- DIVIDE (÷): Let `R` be a table with the attributes A, B, C, and D and let `S` be a table with the attributes C and D. Then we define the division as:

```
R ÷ S = {t | [forall] t_s ∈ S ∃ t_r ∈ R
```

such that $t_r(A,B)=t \wedge t_r(C,D)=t_s\}$ where $t_r(x,y)$ denotes a tuple of table `R` that consists only of the components `x` and `y`. Note that the tuple `t` only consists of the components `A` and `B` of relation `R`.

Given the following tables

```
R:                      S:
 A | B | C | D           C | D
---+---+---+---         ---+---
 a | b | c | d           c | d
 a | b | e | f           e | f
 b | c | e | f
 e | d | c | d
 e | d | e | f
 a | b | d | e
```

R ÷ S is derived as

```
 A | B
---+---
 a | b
 e | d
```

For a more detailed description and definition of the relational algebra refer to [Ullman, 1988] or [Date, 1994].

Example 70.3. A Query Using Relational Algebra

Recall that we formulated all those relational operators to be able to retrieve data from the database. Let's return to our example from the previous section (Operations in the Relational Data Model) where someone wanted to know the names of all suppliers that sell the part `Screw`. This question can be answered using relational algebra by the following operation:

$\pi_{SUPPLIER.SNAME}(\sigma_{PART.PNAME='Screw'}(SUPPLIER \prod SELLS \prod PART))$

We call such an operation a query. If we evaluate the above query against the our example tables (The Suppliers and Parts Database) we will obtain the following result:

```
 SNAME
-------
 Smith
 Adams
```

70.3.2. Relational Calculus

The relational calculus is based on the *first order logic*. There are two variants of the relational calculus:

- The *Domain Relational Calculus* (DRC), where variables stand for components (attributes) of the tuples.
- The *Tuple Relational Calculus* (TRC), where variables stand for tuples.

We want to discuss the tuple relational calculus only because it is the one underlying the most relational languages. For a detailed discussion on DRC (and also TRC) see Date, 1994 or Ullman, 1988 .

70.3.3. Tuple Relational Calculus

The queries used in TRC are of the following form:

```
x(A) | F(x)
```

where `x` is a tuple variable `A` is a set of attributes and `F` is a formula. The resulting relation consists of all tuples `t(A)` that satisfy `F(t)`.

If we want to answer the question from example A Query Using Relational Algebra using TRC we formulate the following query:

```
{x(SNAME) | x ∈ SUPPLIER ∧
    ∃ y ∈ SELLS ∃ z ∈ PART (y(SNO)=x(SNO) ∧
    z(PNO)=y(PNO) ∧
    z(PNAME)='Screw')}
```

Evaluating the query against the tables from The Suppliers and Parts Database again leads to the same result as in A Query Using Relational Algebra.

70.3.4. Relational Algebra vs. Relational Calculus

The relational algebra and the relational calculus have the same *expressive power*; i.e. all queries that can be formulated using relational algebra can also be formulated using the relational calculus and vice versa. This was first proved by E. F. Codd in 1972. This proof is based on an algorithm ("Codd's reduction algorithm") by which an arbitrary expression of the relational calculus can be reduced to a semantically equivalent expression of relational algebra. For a more detailed discussion on that refer to Date, 1994 and Ullman, 1988 .

It is sometimes said that languages based on the relational calculus are "higher level" or "more declarative" than languages based on relational algebra because the algebra (partially) specifies the order of operations while the calculus leaves it to a compiler or interpreter to determine the most efficient order of evaluation.

70.4. The SQL Language

As is the case with most modern relational languages, SQL is based on the tuple relational calculus. As a result every query that can be formulated using the tuple relational calculus (or equivalently, relational algebra) can also be formulated using SQL. There are, however, capabilities beyond the scope of relational algebra or calculus. Here is a list of some additional features provided by SQL that are not part of relational algebra or calculus:

- Commands for insertion, deletion or modification of data.
- Arithmetic capability: In SQL it is possible to involve arithmetic operations as well as comparisons, e.g.

  ```
  A < B + 3.
  ```

  Note that + or other arithmetic operators appear neither in relational algebra nor in relational calculus.
- Assignment and Print Commands: It is possible to print a relation constructed by a query and to assign a computed relation to a relation name.
- Aggregate Functions: Operations such as *average*, *sum*, *max*, etc. can be applied to columns of a relation to obtain a single quantity.

70.4.1. Select

The most often used command in SQL is the SELECT statement, used to retrieve data. The syntax is:

```
SELECT [ALL|DISTINCT]
    { * | expr_1 [AS c_alias_1] [, ...
     [, expr_k [AS c_alias_k]]]}
    FROM table_name_1 [t_alias_1]
     [, ... [, table_name_n [t_alias_n]]]
    [WHERE condition]
    [GROUP BY name_of_attr_i
     [,... [, name_of_attr_j]] [HAVING condition]]
    [{UNION [ALL] | INTERSECT | EXCEPT} SELECT ...]
    [ORDER BY name_of_attr_i [ASC|DESC]
     [, ... [, name_of_attr_j [ASC|DESC]]]];
```

Now we will illustrate the complex syntax of the SELECT statement with various examples. The tables used for the examples are defined in The Suppliers and Parts Database.

70.4.1.1. Simple Selects

Here are some simple examples using a SELECT statement:

Example 70.4. Simple Query with Qualification

To retrieve all tuples from table PART where the attribute PRICE is greater than 10 we formulate the following query:

```
SELECT * FROM PART
    WHERE PRICE > 10;
```

and get the table:

```
 PNO |  PNAME  |  PRICE
-----+---------+--------
  3  |  Bolt   |   15
  4  |  Cam    |   25
```

Using "*" in the SELECT statement will deliver all attributes from the table. If we want to retrieve only the attributes PNAME and PRICE from table PART we use the statement:

```
SELECT PNAME, PRICE
    FROM PART
    WHERE PRICE > 10;
```

In this case the result is:

```
 PNAME  |  PRICE
--------+--------
 Bolt   |   15
 Cam    |   25
```

Note that the SQL SELECT corresponds to the "projection" in relational algebra not to the "selection" (see Relational Algebra for more details).

The qualifications in the WHERE clause can also be logically connected using the keywords OR, AND, and NOT:

```
SELECT PNAME, PRICE
```

```
    FROM PART
    WHERE PNAME = 'Bolt' AND
          (PRICE = 0 OR PRICE < 15);
```

will lead to the result:

```
 PNAME   |  PRICE
---------+--------
 Bolt    |   15
```

Arithmetic operations may be used in the target list and in the WHERE clause. For example if we want to know how much it would cost if we take two pieces of a part we could use the following query:

```
SELECT PNAME, PRICE * 2 AS DOUBLE
    FROM PART
    WHERE PRICE * 2 < 50;
```

and we get:

```
 PNAME   |  DOUBLE
---------+---------
 Screw   |    20
 Nut     |    16
 Bolt    |    30
```

Note that the word DOUBLE after the keyword AS is the new title of the second column. This technique can be used for every element of the target list to assign a new title to the resulting column. This new title is often referred to as alias. The alias cannot be used throughout the rest of the query.

70.4.1.2. Joins

The following example shows how *joins* are realized in SQL.

To join the three tables SUPPLIER, PART and SELLS over their common attributes we formulate the following statement:

```
SELECT S.SNAME, P.PNAME
    FROM SUPPLIER S, PART P, SELLS SE
    WHERE S.SNO = SE.SNO AND
          P.PNO = SE.PNO;
```

and get the following table as a result:

```
 SNAME | PNAME
-------+-------
 Smith | Screw
 Smith | Nut
 Jones | Cam
 Adams | Screw
 Adams | Bolt
 Blake | Nut
 Blake | Bolt
 Blake | Cam
```

In the FROM clause we introduced an alias name for every relation because there are common named attributes (SNO and PNO) among the relations. Now we can distinguish between the common named attributes by simply prefixing the attribute name with the alias name followed by a dot. The join is calculated in the same way as shown in An Inner Join. First the Cartesian product SUPPLIER × PART × SELLS is derived. Now only those tuples satisfying the conditions given in the WHERE clause are selected (i.e. the common named attributes have to be equal). Finally we project out all columns but S.SNAME and P.PNAME.

70.4.1.3. Aggregate Operators

SQL provides aggregate operators (e.g. AVG, COUNT, SUM, MIN, MAX) that take the name of an attribute as an argument. The value of the aggregate operator is calculated over all values of the specified attribute (column) of the whole table. If groups are specified in the query the calculation is done only over the values of a group (see next section).

Example 70.5. Aggregates

If we want to know the average cost of all parts in table PART we use the following query:

```
SELECT AVG(PRICE) AS AVG_PRICE
    FROM PART;
```

The result is:

```
 AVG_PRICE
-----------
   14.5
```

If we want to know how many parts are stored in table PART we use the statement:

```
SELECT COUNT(PNO)
    FROM PART;
```

and get:

```
 COUNT
-------
   4
```

70.4.1.4. Aggregation by Groups

SQL allows one to partition the tuples of a table into groups. Then the aggregate operators described above can be applied to the groups (i.e. the value of the aggregate operator is no longer calculated over all the values of the specified column but over all values of a group. Thus the aggregate operator is evaluated individually for every group.)

The partitioning of the tuples into groups is done by using the keywords `GROUP BY` followed by a list of attributes that define the groups. If we have `GROUP BY` $A_1, \cdots, A_k$ we partition the relation into groups, such that two tuples are in the same group if and only if they agree on all the attributes $A_1, \cdots, A_k$.

Example 70.6. Aggregates

If we want to know how many parts are sold by every supplier we formulate the query:

```
SELECT S.SNO, S.SNAME, COUNT(SE.PNO)
    FROM SUPPLIER S, SELLS SE
```

```
    WHERE S.SNO = SE.SNO
    GROUP BY S.SNO, S.SNAME;
```

and get:

```
 SNO | SNAME | COUNT
-----+-------+-------
  1  | Smith |   2
  2  | Jones |   1
  3  | Adams |   2
  4  | Blake |   3
```

Now let's have a look of what is happening here. First the join of the tables SUPPLIER and SELLS is derived:

```
 S.SNO | S.SNAME | SE.PNO
-------+---------+--------
   1   |  Smith  |   1
   1   |  Smith  |   2
   2   |  Jones  |   4
   3   |  Adams  |   1
   3   |  Adams  |   3
   4   |  Blake  |   2
   4   |  Blake  |   3
   4   |  Blake  |   4
```

Next we partition the tuples into groups by putting all tuples together that agree on both attributes S.SNO and S.SNAME:

```
 S.SNO | S.SNAME | SE.PNO
-------+---------+--------
   1   |  Smith  |   1
                 |   2
-------------------------
   2   |  Jones  |   4
-------------------------
   3   |  Adams  |   1
                 |   3
-------------------------
   4   |  Blake  |   2
                 |   3
                 |   4
```

In our example we got four groups and now we can apply the aggregate operator COUNT to every group leading to the total result of the query given above.

Note that for the result of a query using GROUP BY and aggregate operators to make sense the attributes grouped by must also appear in the target list. All further attributes not appearing in the GROUP BY clause can only be selected by using an aggregate function. On the other hand you can not use aggregate functions on attributes appearing in the GROUP BY clause.

70.4.1.5. Having

The HAVING clause works much like the WHERE clause and is used to consider only those groups satisfying the qualification given in the HAVING clause. The expressions allowed in the HAVING

clause must involve aggregate functions. Every expression using only plain attributes belongs to the WHERE clause. On the other hand every expression involving an aggregate function must be put to the HAVING clause.

Example 70.7. Having

If we want only those suppliers selling more than one part we use the query:

```
SELECT S.SNO, S.SNAME, COUNT(SE.PNO)
    FROM SUPPLIER S, SELLS SE
    WHERE S.SNO = SE.SNO
    GROUP BY S.SNO, S.SNAME
    HAVING COUNT(SE.PNO) > 1;
```

and get:

```
 SNO | SNAME | COUNT
-----+-------+-------
  1  | Smith |   2
  3  | Adams |   2
  4  | Blake |   3
```

70.4.1.6. Subqueries

In the WHERE and HAVING clauses the use of subqueries (subselects) is allowed in every place where a value is expected. In this case the value must be derived by evaluating the subquery first. The usage of subqueries extends the expressive power of SQL.

Example 70.8. Subselect

If we want to know all parts having a greater price than the part named 'Screw' we use the query:

```
SELECT *
    FROM PART
    WHERE PRICE > (SELECT PRICE FROM PART
                   WHERE PNAME='Screw');
```

The result is:

```
 PNO |  PNAME  |  PRICE
-----+---------+--------
  3  |  Bolt   |   15
  4  |  Cam    |   25
```

When we look at the above query we can see the keyword SELECT two times. The first one at the beginning of the query - we will refer to it as outer SELECT - and the one in the WHERE clause which begins a nested query - we will refer to it as inner SELECT. For every tuple of the outer SELECT the inner SELECT has to be evaluated. After every evaluation we know the price of the tuple named 'Screw' and we can check if the price of the actual tuple is greater.

If we want to know all suppliers that do not sell any part (e.g. to be able to remove these suppliers from the database) we use:

```
SELECT *
    FROM SUPPLIER S
```

```
    WHERE NOT EXISTS
        (SELECT * FROM SELLS SE
         WHERE SE.SNO = S.SNO);
```

In our example the result will be empty because every supplier sells at least one part. Note that we use S.SNO from the outer SELECT within the WHERE clause of the inner SELECT. As described above the subquery is evaluated for every tuple from the outer query i.e. the value for S.SNO is always taken from the actual tuple of the outer SELECT.

70.4.1.7. Union, Intersect, Except

These operations calculate the union, intersect and set theoretic difference of the tuples derived by two subqueries.

Example 70.9. Union, Intersect, Except

The following query is an example for UNION:

```
SELECT S.SNO, S.SNAME, S.CITY
    FROM SUPPLIER S
    WHERE S.SNAME = 'Jones'
    UNION
    SELECT S.SNO, S.SNAME, S.CITY
    FROM SUPPLIER S
    WHERE S.SNAME = 'Adams';
```

gives the result:

```
 SNO | SNAME |  CITY
-----+-------+--------
  2  | Jones | Paris
  3  | Adams | Vienna
```

Here an example for INTERSECT:

```
SELECT S.SNO, S.SNAME, S.CITY
    FROM SUPPLIER S
    WHERE S.SNO > 1
    INTERSECT
    SELECT S.SNO, S.SNAME, S.CITY
    FROM SUPPLIER S
    WHERE S.SNO > 2;
```

gives the result:

```
 SNO | SNAME |  CITY
-----+-------+--------
  2  | Jones | Paris
```

The only tuple returned by both parts of the query is the one having $SNO=2$.

Finally an example for EXCEPT:

```
SELECT S.SNO, S.SNAME, S.CITY
    FROM SUPPLIER S
```

```
    WHERE S.SNO > 1
    EXCEPT
    SELECT S.SNO, S.SNAME, S.CITY
    FROM SUPPLIER S
    WHERE S.SNO > 3;
```

gives the result:

```
 SNO | SNAME |  CITY
-----+-------+--------
  2  | Jones | Paris
  3  | Adams | Vienna
```

70.4.2. Data Definition

There is a set of commands used for data definition included in the SQL language.

70.4.2.1. Create Table

The most fundamental command for data definition is the one that creates a new relation (a new table). The syntax of the CREATE TABLE command is:

```
CREATE TABLE table_name
    (name_of_attr_1 type_of_attr_1
     [, name_of_attr_2 type_of_attr_2
     [, ...]]);
```

Example 70.10. Table Creation

To create the tables defined in The Suppliers and Parts Database the following SQL statements are used:

```
CREATE TABLE SUPPLIER
    (SNO   INTEGER,
     SNAME VARCHAR(20),
     CITY  VARCHAR(20));


CREATE TABLE PART
    (PNO   INTEGER,
     PNAME VARCHAR(20),
     PRICE DECIMAL(4 , 2));


CREATE TABLE SELLS
    (SNO INTEGER,
     PNO INTEGER);
```

70.4.2.2. Data Types in SQL

The following is a list of some data types that are supported by SQL:

- INTEGER: signed fullword binary integer (31 bits precision).
- SMALLINT: signed halfword binary integer (15 bits precision).

- DECIMAL (p[,q]): signed packed decimal number of p digits precision with assumed q of them right to the decimal point. ($15 \geq p \geq q \geq 0$). If q is omitted it is assumed to be 0.
- FLOAT: signed doubleword floating point number.
- CHAR(n): fixed length character string of length n.
- VARCHAR(n): varying length character string of maximum length n.

70.4.2.3. Create Index

Indices are used to speed up access to a relation. If a relation `R` has an index on attribute `A` then we can retrieve all tuples t having t(`A`) = a in time roughly proportional to the number of such tuples t rather than in time proportional to the size of `R`.

To create an index in SQL the CREATE INDEX command is used. The syntax is:

```
CREATE INDEX index_name
    ON table_name ( name_of_attribute );
```

Example 70.11. Create Index

To create an index named I on attribute SNAME of relation SUPPLIER we use the following statement:

```
CREATE INDEX I ON SUPPLIER (SNAME);
```

The created index is maintained automatically, i.e. whenever a new tuple is inserted into the relation SUPPLIER the index I is adapted. Note that the only changes a user can percept when an index is present are an increased speed.

70.4.2.4. Create View

A view may be regarded as a *virtual table*, i.e. a table that does not *physically* exist in the database but looks to the user as if it does. By contrast, when we talk of a *base table* there is really a physically stored counterpart of each row of the table somewhere in the physical storage.

Views do not have their own, physically separate, distinguishable stored data. Instead, the system stores the definition of the view (i.e. the rules about how to access physically stored base tables in order to materialize the view) somewhere in the system catalogs (see System Catalogs). For a discussion on different techniques to implement views refer to *SIM98*.

In SQL the `CREATE VIEW` command is used to define a view. The syntax is:

```
CREATE VIEW view_name
    AS select_stmt
```

where *`select_stmt`* is a valid select statement as defined in Select. Note that *`select_stmt`* is not executed when the view is created. It is just stored in the *system catalogs* and is executed whenever a query against the view is made.

Let the following view definition be given (we use the tables from The Suppliers and Parts Database again):

```
CREATE VIEW London_Suppliers
    AS SELECT S.SNAME, P.PNAME
        FROM SUPPLIER S, PART P, SELLS SE
        WHERE S.SNO = SE.SNO AND
              P.PNO = SE.PNO AND
```

```
                S.CITY = 'London';
```

Now we can use this *virtual relation* `London_Suppliers` as if it were another base table:

```
SELECT * FROM London_Suppliers
    WHERE P.PNAME = 'Screw';
```

which will return the following table:

```
 SNAME | PNAME
-------+-------
 Smith | Screw
```

To calculate this result the database system has to do a *hidden* access to the base tables SUPPLIER, SELLS and PART first. It does so by executing the query given in the view definition against those base tables. After that the additional qualifications (given in the query against the view) can be applied to obtain the resulting table.

70.4.2.5. Drop Table, Drop Index, Drop View

To destroy a table (including all tuples stored in that table) the DROP TABLE command is used:

```
DROP TABLE table_name;
```

To destroy the SUPPLIER table use the following statement:

```
DROP TABLE SUPPLIER;
```

The DROP INDEX command is used to destroy an index:

```
DROP INDEX index_name;
```

Finally to destroy a given view use the command DROP VIEW:

```
DROP VIEW view_name;
```

70.4.3. Data Manipulation

70.4.3.1. Insert Into

Once a table is created (see Create Table), it can be filled with tuples using the command `INSERT INTO`. The syntax is:

```
INSERT INTO table_name (name_of_attr_1
    [, name_of_attr_2 [,...]])
    VALUES (val_attr_1 [, val_attr_2 [, ...]]);
```

To insert the first tuple into the relation SUPPLIER (from The Suppliers and Parts Database) we use the following statement:

```
INSERT INTO SUPPLIER (SNO, SNAME, CITY)
    VALUES (1, 'Smith', 'London');
```

To insert the first tuple into the relation SELLS we use:

```
INSERT INTO SELLS (SNO, PNO)
    VALUES (1, 1);
```

70.4.3.2. Update

To change one or more attribute values of tuples in a relation the UPDATE command is used. The syntax is:

```
UPDATE table_name
    SET name_of_attr_1 = value_1
        [, ... [, name_of_attr_k = value_k]]
    WHERE condition;
```

To change the value of attribute PRICE of the part 'Screw' in the relation PART we use:

```
UPDATE PART
    SET PRICE = 15
    WHERE PNAME = 'Screw';
```

The new value of attribute PRICE of the tuple whose name is 'Screw' is now 15.

70.4.3.3. Delete

To delete a tuple from a particular table use the command DELETE FROM. The syntax is:

```
DELETE FROM table_name
    WHERE condition;
```

To delete the supplier called 'Smith' of the table SUPPLIER the following statement is used:

```
DELETE FROM SUPPLIER
    WHERE SNAME = 'Smith';
```

70.4.4. System Catalogs

In every SQL database system *system catalogs* are used to keep track of which tables, views indexes etc. are defined in the database. These system catalogs can be queried as if they were normal relations. For example there is one catalog used for the definition of views. This catalog stores the query from the view definition. Whenever a query against a view is made, the system first gets the *view definition query* out of the catalog and materializes the view before proceeding with the user query (see Simkovics, 1998 for a more detailed description). For more information about system catalogs refer to Date, 1994 .

70.4.5. Embedded SQL

In this section we will sketch how SQL can be embedded into a host language (e.g. C). There are two main reasons why we want to use SQL from a host language:

- There are queries that cannot be formulated using pure SQL (i.e. recursive queries). To be able to perform such queries we need a host language with a greater expressive power than SQL.

- We simply want to access a database from some application that is written in the host language (e.g. a ticket reservation system with a graphical user interface is written in C and the information about which tickets are still left is stored in a database that can be accessed using embedded SQL).

A program using embedded SQL in a host language consists of statements of the host language and of *embedded SQL* (ESQL) statements. Every ESQL statement begins with the keywords `EXEC SQL`. The ESQL statements are transformed to statements of the host language by a *precompiler* (which usually inserts calls to library routines that perform the various SQL commands).

When we look at the examples throughout Select we realize that the result of the queries is very often a set of tuples. Most host languages are not designed to operate on sets so we need a mechanism to access every single tuple of the set of tuples returned by a SELECT statement. This mechanism can be provided by declaring a *cursor*. After that we can use the FETCH command to retrieve a tuple and set the cursor to the next tuple.

For a detailed discussion on embedded SQL refer to Date and Darwen, 1997 , Date, 1994 , or Ullman, 1988 .

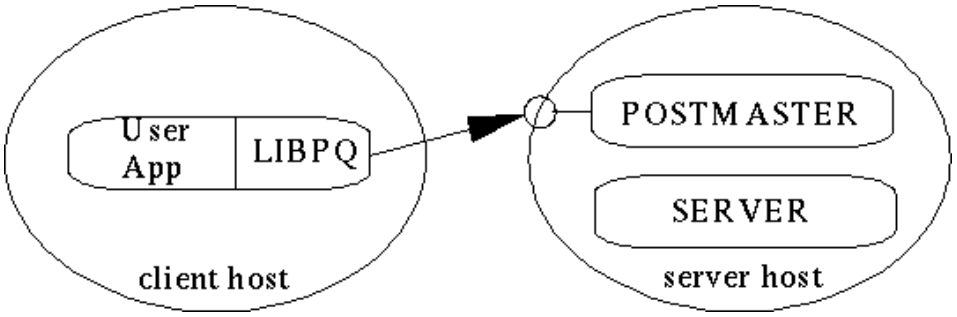
Chapter 71. Architecture

71.1. Postgres Architectural Concepts

Before we begin, you should understand the basic Postgres system architecture. Understanding how the parts of Postgres interact will make the next chapter somewhat clearer. In database jargon, Postgres uses a simple "process per-user" client/server model. A Postgres session consists of the following cooperating Unix processes (programs):

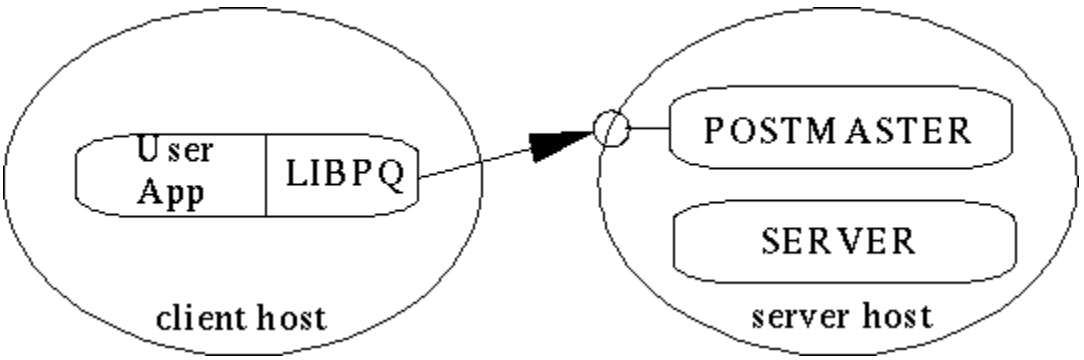
- A supervisory daemon process (postmaster),
- the user's frontend application (e.g., the psql program), and
- the one or more backend database servers (the postgres process itself).

A single postmaster manages a given collection of databases on a single host. Such a collection of databases is called an installation or site. Frontend applications that wish to access a given database within an installation make calls to the library. The library sends user requests over the network to the postmaster (



), which in turn starts a new backend server process

Figure 71.1. How a connection is established



and connects the frontend process to the new server. From that point on, the frontend process and the backend server communicate without intervention by the postmaster. Hence, the postmaster is always running, waiting for requests, whereas frontend and backend processes come and go.

The `libpq` library allows a single frontend to make multiple connections to backend processes. However, the frontend application is still a single-threaded process. Multithreaded frontend/backend connections are not currently supported in `libpq`. One implication of this architecture is that the postmaster and the backend always run on the same machine (the database server), while the frontend application may run anywhere. You should keep this in mind, because the files that can be accessed on a client machine may not be accessible (or may only be accessed using a different filename) on the database server machine.

You should also be aware that the postmaster and postgres servers run with the user-id of the Postgres "superuser." Note that the Postgres superuser does not have to be a special user (e.g., a user named "postgres"). Furthermore, the Postgres superuser should definitely not be the Unix superuser ("root")! In any case, all files relating to a database should belong to this Postgres superuser.

Chapter 72. Getting Started

How to begin work with Postgres for a new user.

Some of the steps required to use Postgres can be performed by any Postgres user, and some must be done by the site database administrator. This site administrator is the person who installed the software, created the database directories and started the postmaster process. This person does not have to be the Unix superuser ("root") or the computer system administrator; a person can install and use Postgres without any special accounts or privileges.

If you are installing Postgres yourself, then refer to the Administrator's Guide for instructions on installation, and return to this guide when the installation is complete.

Throughout this manual, any examples that begin with the character "`%`" are commands that should be typed at the Unix shell prompt. Examples that begin with the character "`*`" are commands in the Postgres query language, Postgres SQL.

72.1. Setting Up Your Environment

This section discusses how to set up your own environment so that you can use frontend applications. We assume Postgres has already been successfully installed and started; refer to the Administrator's Guide and the installation notes for how to install Postgres.

Postgres is a client/server application. As a user, you only need access to the client portions of the installation (an example of a client application is the interactive monitor psql). For simplicity, we will assume that Postgres has been installed in the directory `/usr/local/pgsql`. Therefore, wherever you see the directory `/usr/local/pgsql` you should substitute the name of the directory where Postgres is actually installed. All Postgres commands are installed in the directory `/usr/local/pgsql/bin`. Therefore, you should add this directory to your shell command path. If you use a variant of the Berkeley C shell, such as csh or tcsh, you would add

```
% set path = ( /usr/local/pgsql/bin path )
```

in the `.login` file in your home directory. If you use a variant of the Bourne shell, such as sh, ksh, or bash, then you would add

```
% PATH=/usr/local/pgsql/bin:$PATH
% export PATH
```

to the .profile file in your home directory. From now on, we will assume that you have added the Postgres bin directory to your path. In addition, we will make frequent reference to “setting a shell variable” or “setting an environment variable” throughout this document. If you did not fully understand the last paragraph on modifying your search path, you should consult the Unix manual pages that describe your shell before going any further.

If your site administrator has not set things up in the default way, you may have some more work to do. For example, if the database server machine is a remote machine, you will need to set the PGHOST environment variable to the name of the database server machine. The environment variable PGPORT may also have to be set. The bottom line is this: if you try to start an application program and it complains that it cannot connect to the postmaster, you should immediately consult your site administrator to make sure that your environment is properly set up.

72.2. Starting the Interactive Monitor (psql)

Assuming that your site administrator has properly started the postmaster process and authorized you to use the database, you (as a user) may begin to start up applications. As previously mentioned, you

should add `/usr/local/pgsql/bin` to your shell search path. In most cases, this is all you should have to do in terms of preparation.

Two different styles of connections are supported. The site administrator will have chosen to allow TCP/IP network connections or will have restricted database access to local (same-machine) socket connections only. These choices become significant if you encounter problems in connecting to a database, since you will want to confirm that you are choosing an allowed connection option.

If you get the following error message from a Postgres command (such as psql or createdb):

```
% psql template1
Connection to database 'postgres' failed.
connectDB() failed: Is the postmaster running and accepting
 connections
    at 'UNIX Socket' on port '5432'?
```

or

```
% psql -h localhost template1
Connection to database 'postgres' failed.
connectDB() failed: Is the postmaster running and accepting TCP/IP
    (with -i) connections at 'localhost' on port '5432'?
```

it is usually because

- the postmaster is not running, or
- you are attempting to connect to the wrong server host.

If you get the following error message:

```
FATAL 1:Feb 17 23:19:55:process userid (2360) != database owner
 (268)
```

it means that the site administrator started the postmaster as the wrong user. Tell him to restart it as the Postgres superuser.

72.3. Managing a Database

Now that Postgres is up and running we can create some databases to experiment with. Here, we describe the basic commands for managing a database.

Most Postgres applications assume that the database name, if not specified, is the same as the name on your computer account.

If your database administrator has set up your account without database creation privileges, then she should have told you what the name of your database is. If this is the case, then you can skip the sections on creating and destroying databases.

72.3.1. Creating a Database

Let's say you want to create a database named mydb. You can do this with the following command:

```
% createdb mydb
```

If you do not have the privileges required to create a database, you will see the following:

```
% createdb mydb
NOTICE:user "your username" is not allowed to create/destroy
 databases
createdb: database creation failed on mydb.
```

Postgres allows you to create any number of databases at a given site and you automatically become the database administrator of the database you just created. Database names must have an alphabetic first character and are limited to 32 characters in length. Not every user has authorization to become a database administrator. If Postgres refuses to create databases for you, then the site administrator needs to grant you permission to create databases. Consult your site administrator if this occurs.

72.3.2. Accessing a Database

Once you have constructed a database, you can access it by:

- Running the Postgres terminal monitor programs (e.g. psql) which allows you to interactively enter, edit, and execute SQL commands.
- Using an existing native frontend tool like pgaccess or ApplixWare (via ODBC) to create and manipulate a database.
- Using a language like perl or tcl which has a supported interface for Postgres. Some of these languages also have convenient and powerful GUI toolkits which can help you construct custom applications. pgaccess, mentioned above, is one such application written in tk/tcl and can be used as an example.
- Writing a C program using the LIBPQ subroutine library. This allows you to submit SQL commands from C and get answers and status messages back to your program. This interface is discussed further in *The PostgreSQL Programmer's Guide*.

You might want to start up psql, to try out the examples in this manual. It can be activated for the mydb database by typing the command:

```
% psql mydb
```

You will be greeted with the following message:

```
Welcome to the POSTGRESQL interactive sql monitor:
  Please read the file COPYRIGHT for copyright terms of POSTGRESQL

   type \? for help on slash commands
   type \q to quit
   type \g or terminate with semicolon to execute query
 You are currently connected to the database: template1

mydb=>
```

This prompt indicates that the terminal monitor is listening to you and that you can type SQL queries into a workspace maintained by the terminal monitor. The psql program responds to escape codes that begin with the backslash character, "\" For example, you can get help on the syntax of various Postgres SQL commands by typing:

```
mydb=> \h
```

Once you have finished entering your queries into the workspace, you can pass the contents of the workspace to the Postgres server by typing:

```
mydb=> \g
```

This tells the server to process the query. If you terminate your query with a semicolon, the "\g" is not necessary. psql will automatically process semicolon terminated queries. To read queries from a file, say myFile, instead of entering them interactively, type:

```
mydb=> \i fileName
```

To get out of psql and return to Unix, type

```
mydb=> \q
```

and psql will quit and return you to your command shell. (For more escape codes, type \h at the monitor prompt.) White space (i.e., spaces, tabs and newlines) may be used freely in SQL queries. Single-line comments are denoted by "--". Everything after the dashes up to the end of the line is ignored. Multiple-line comments, and comments within a line, are denoted by "/* ... */".

72.3.3. Destroying a Database

If you are the database administrator for the database mydb, you can destroy it using the following Unix command:

```
% dropdb mydb
```

This action physically removes all of the Unix files associated with the database and cannot be undone, so this should only be done with a great deal of forethought.

Chapter 73. The Query Language

The Postgres query language is a variant of the SQL3 draft next-generation standard. It has many extensions to SQL92 such as an extensible type system, inheritance, functions and production rules. These are features carried over from the original Postgres query language, PostQuel. This section provides an overview of how to use Postgres SQL to perform simple operations. This manual is only intended to give you an idea of our flavor of SQL and is in no way a complete tutorial on SQL. Numerous books have been written on SQL92, including Melton and Simon, 1993 and Date and Darwen, 1997 . You should be aware that some language features are extensions to the ANSI standard.

73.1. Interactive Monitor

In the examples that follow, we assume that you have created the mydb database as described in the previous subsection and have started psql. Examples in this manual can also be found in `/usr/local/pgsql/src/tutorial/`. Refer to the `README` file in that directory for how to use them. To start the tutorial, do the following:

```
% cd /usr/local/pgsql/src/tutorial
% psql -s mydb
Welcome to the POSTGRESQL interactive sql monitor:
  Please read the file COPYRIGHT for copyright terms of POSTGRESQL

   type \? for help on slash commands
   type \q to quit
   type \g or terminate with semicolon to execute query
 You are currently connected to the database: postgres

mydb=> \i basics.sql
```

The `\i` command read in queries from the specified files. The `-s` option puts you in single step mode which pauses before sending a query to the backend. Queries in this section are in the file `basics.sql`.

psql has a variety of `\d` commands for showing system information. Consult these commands for more details; for a listing, type `\?` at the psql prompt.

73.2. Concepts

The fundamental notion in Postgres is that of a class, which is a named collection of object instances. Each instance has the same collection of named attributes, and each attribute is of a specific type. Furthermore, each instance has a permanent *object identifier* (OID) that is unique throughout the installation. Because SQL syntax refers to tables, we will use the terms *table* and *class* interchangeably. Likewise, an SQL *row* is an *instance* and SQL *columns* are *attributes*. As previously discussed, classes are grouped into databases, and a collection of databases managed by a single postmaster process constitutes an installation or site.

73.3. Creating a New Class

You can create a new class by specifying the class name, along with all attribute names and their types:

```
CREATE TABLE weather (
    city            varchar(80),
    temp_lo         int,           -- low temperature
    temp_hi         int,           -- high temperature
```

```
    prcp            real,           -- precipitation
    date            date
);
```

Note that both keywords and identifiers are case-insensitive; identifiers can preserve case by surrounding them with double-quotes as allowed by SQL92. Postgres SQL supports the usual SQL types `int`, `float`, `real`, `smallint`, `char(N)`, `varchar(N)`, `date`, `time`, and `timestamp`, as well as other types of general utility and a rich set of geometric types. As we will see later, Postgres can be customized with an arbitrary number of user-defined data types. Consequently, type names are not syntactical keywords, except where required to support special cases in the SQL92 standard. So far, the Postgres `CREATE` command looks exactly like the command used to create a table in a traditional relational system. However, we will presently see that classes have properties that are extensions of the relational model.

73.4. Populating a Class with Instances

The `INSERT` statement is used to populate a class with instances:

```
INSERT INTO weather
    VALUES ('San Francisco', 46, 50, 0.25, '11/27/1994');
```

You can also use `COPY` to load large amounts of data from flat (ASCII) files. This is usually faster because the data is read (or written) as a single atomic transaction directly to or from the target table. An example would be:

```
COPY weather FROM '/home/user/weather.txt'
    USING DELIMITERS '|';
```

where the path name for the source file must be available to the backend server machine, not the client, since the backend server reads the file directly.

73.5. Querying a Class

The weather class can be queried with normal relational selection and projection queries. A SQL `SELECT` statement is used to do this. The statement is divided into a target list (the part that lists the attributes to be returned) and a qualification (the part that specifies any restrictions). For example, to retrieve all the rows of weather, type:

```
SELECT * FROM weather;
```

and the output should be:

```
+--------------+---------+---------+------+------------+
|city          | temp_lo | temp_hi | prcp | date       |
+--------------+---------+---------+------+------------+
|San Francisco | 46      | 50      | 0.25 | 11-27-1994 |
+--------------+---------+---------+------+------------+
|San Francisco | 43      | 57      | 0    | 11-29-1994 |
+--------------+---------+---------+------+------------+
|Hayward       | 37      | 54      |      | 11-29-1994 |
+--------------+---------+---------+------+------------+
```

You may specify any arbitrary expressions in the target list. For example, you can do:

```
SELECT city, (temp_hi+temp_lo)/2 AS temp_avg, date FROM weather;
```

Arbitrary Boolean operators (AND, OR and NOT) are allowed in the qualification of any query. For example,

```
SELECT * FROM weather
    WHERE city = 'San Francisco'
    AND prcp > 0.0;
```

results in:

```
+--------------+---------+---------+------+------------+
|city          | temp_lo | temp_hi | prcp | date       |
+--------------+---------+---------+------+------------+
|San Francisco | 46      | 50      | 0.25 | 11-27-1994 |
+--------------+---------+---------+------+------------+
```

As a final note, you can specify that the results of a select can be returned in a *sorted order* or with *duplicate instances* removed.

```
SELECT DISTINCT city
    FROM weather
    ORDER BY city;
```

73.6. Redirecting SELECT Queries

Any SELECT query can be redirected to a new class

```
SELECT * INTO TABLE temp FROM weather;
```

This forms an implicit CREATE command, creating a new class temp with the attribute names and types specified in the target list of the SELECT INTO command. We can then, of course, perform any operations on the resulting class that we can perform on other classes.

73.7. Joins Between Classes

Thus far, our queries have only accessed one class at a time. Queries can access multiple classes at once, or access the same class in such a way that multiple instances of the class are being processed at the same time. A query that accesses multiple instances of the same or different classes at one time is called a join query. As an example, say we wish to find all the records that are in the temperature range of other records. In effect, we need to compare the temp_lo and temp_hi attributes of each EMP instance to the temp_lo and temp_hi attributes of all other EMP instances.

Note

This is only a conceptual model. The actual join may be performed in a more efficient manner, but this is invisible to the user.

We can do this with the following query:

```
SELECT W1.city, W1.temp_lo AS low, W1.temp_hi AS high,
    W2.city, W2.temp_lo AS low, W2.temp_hi AS high
    FROM weather W1, weather W2
    WHERE W1.temp_lo < W2.temp_lo
```

```
    AND W1.temp_hi > W2.temp_hi;

+--------------+-----+------+--------------+-----+------+
|city          | low | high | city         | low | high |
+--------------+-----+------+--------------+-----+------+
|San Francisco | 43  | 57   | San Francisco | 46  | 50   |
+--------------+-----+------+--------------+-----+------+
|San Francisco | 37  | 54   | San Francisco | 46  | 50   |
+--------------+-----+------+--------------+-----+------+
```

Note

The semantics of such a join are that the qualification is a truth expression defined for the Cartesian product of the classes indicated in the query. For those instances in the Cartesian product for which the qualification is true, Postgres computes and returns the values specified in the target list. Postgres SQL does not assign any meaning to duplicate values in such expressions. This means that Postgres sometimes recomputes the same target list several times; this frequently happens when Boolean expressions are connected with an "or". To remove such duplicates, you must use the `SELECT DISTINCT` statement.

In this case, both `W1` and `W2` are surrogates for an instance of the class weather, and both range over all instances of the class. (In the terminology of most database systems, `W1` and `W2` are known as *range variables*.) A query can contain an arbitrary number of class names and surrogates.

73.8. Updates

You can update existing instances using the `UPDATE` command. Suppose you discover the temperature readings are all off by 2 degrees as of Nov 28, you may update the data as follow:

```
UPDATE weather
    SET temp_hi = temp_hi - 2,  temp_lo = temp_lo - 2
    WHERE date > '11/28/1994';
```

73.9. Deletions

Deletions are performed using the `DELETE` command:

```
DELETE FROM weather WHERE city = 'Hayward';
```

All weather recording belongs to Hayward is removed. One should be wary of queries of the form

```
DELETE FROM classname;
```

Without a qualification, `DELETE` will simply remove all instances of the given class, leaving it empty. The system will not request confirmation before doing this.

73.10. Using Aggregate Functions

Like most other relational database products, PostgreSQL supports aggregate functions. An aggregate function computes a single result from multiple input rows. For example, there are aggregates to compute the `count`, `sum`, `avg` (average), `max` (maximum) and `min` (minimum) over a set of instances.

It is important to understand the interaction between aggregates and SQL's WHERE and HAVING clauses. The fundamental difference between WHERE and HAVING is this: WHERE selects input rows before groups and aggregates are computed (thus, it controls which rows go into the aggregate computation), whereas HAVING selects group rows after groups and aggregates are computed. Thus, the WHERE clause may not contain aggregate functions; it makes no sense to try to use an aggregate to determine which rows will be inputs to the aggregates. On the other hand, HAVING clauses always contain aggregate functions. (Strictly speaking, you are allowed to write a HAVING clause that doesn't use aggregates, but it's wasteful; the same condition could be used more efficiently at the WHERE stage.)

As an example, we can find the highest low-temperature reading anywhere with

```
SELECT max(temp_lo) FROM weather;
```

If we want to know which city (or cities) that reading occurred in, we might try

```
SELECT city FROM weather WHERE temp_lo = max(temp_lo);
```

but this will not work since the aggregate max can't be used in WHERE. However, as is often the case the query can be restated to accomplish the intended result; here by using a *subselect*:

```
SELECT city FROM weather
    WHERE temp_lo = (SELECT max(temp_lo) FROM weather);
```

This is OK because the sub-select is an independent computation that computes its own aggregate separately from what's happening in the outer select.

Aggregates are also very useful in combination with GROUP BY clauses. For example, we can get the maximum low temperature observed in each city with

```
SELECT city, max(temp_lo)
    FROM weather
    GROUP BY city;
```

which gives us one output row per city. We can filter these grouped rows using HAVING:

```
SELECT city, max(temp_lo)
    FROM weather
    GROUP BY city
    HAVING min(temp_lo) < 0;
```

which gives us the same results for only the cities that have some below-zero readings. Finally, if we only care about cities whose names begin with "P", we might do

```
SELECT city, max(temp_lo)
    FROM weather
    WHERE city like 'P%'
    GROUP BY city
    HAVING min(temp_lo) < 0;
```

Note that we can apply the city-name restriction in WHERE, since it needs no aggregate. This is more efficient than adding the restriction to HAVING, because we avoid doing the grouping and aggregate calculations for all rows that fail the WHERE check.

Chapter 74. Advanced Postgres SQL Features

Having covered the basics of using Postgres SQL to access your data, we will now discuss those features of Postgres that distinguish it from conventional data managers. These features include inheritance, time travel and non-atomic data values (array- and set-valued attributes). Examples in this section can also be found in `advance.sql` in the tutorial directory. (Refer to Chapter 73, *The Query Language* for how to use it.)

74.1. Inheritance

Let's create two classes. The capitals class contains state capitals which are also cities. Naturally, the capitals class should inherit from cities.

```
CREATE TABLE cities (
    name            text,
    population      float,
    altitude        int     -- (in ft)
);

CREATE TABLE capitals (
    state           char(2)
) INHERITS (cities);
```

In this case, an instance of capitals *inherits* all attributes (name, population, and altitude) from its parent, cities. The type of the attribute name is `text`, a native Postgres type for variable length ASCII strings. The type of the attribute population is `float`, a native Postgres type for double precision floating point numbers. State capitals have an extra attribute, state, that shows their state. In Postgres, a class can inherit from zero or more other classes, and a query can reference either all instances of a class or all instances of a class plus all of its descendants.

Note

The inheritance hierarchy is a directed acyclic graph.

For example, the following query finds all the cities that are situated at an attitude of 500ft or higher:

```
SELECT name, altitude
    FROM cities
    WHERE altitude > 500;

+----------+----------+
|name      | altitude |
+----------+----------+
|Las Vegas | 2174     |
+----------+----------+
|Mariposa  | 1953     |
+----------+----------+
```

On the other hand, to find the names of all cities, including state capitals, that are located at an altitude over 500ft, the query is:

```
SELECT c.name, c.altitude
    FROM cities* c
```

```
    WHERE c.altitude > 500;
```

which returns:

```
+----------+----------+
|name      | altitude |
+----------+----------+
|Las Vegas | 2174     |
+----------+----------+
|Mariposa  | 1953     |
+----------+----------+
|Madison   | 845      |
+----------+----------+
```

Here the "*" after cities indicates that the query should be run over cities and all classes below cities in the inheritance hierarchy. Many of the commands that we have already discussed (`SELECT`, `UPDATE` and `DELETE`) support this inheritance notation using "*" as do other commands like `ALTER`.

74.2. Non-Atomic Values

One of the tenets of the relational model is that the attributes of a relation are atomic. Postgres does not have this restriction; attributes can themselves contain sub-values that can be accessed from the query language. For example, you can create attributes that are arrays of base types.

74.2.1. Arrays

Postgres allows attributes of an instance to be defined as fixed-length or variable-length multi-dimensional arrays. Arrays of any base type or user-defined type can be created. To illustrate their use, we first create a class with arrays of base types.

```
CREATE TABLE SAL_EMP (
    name            text,
    pay_by_quarter  int4[],
    schedule        text[][]
);
```

The above query will create a class named SAL_EMP with a *text* string (name), a one-dimensional array of *int4* (pay_by_quarter), which represents the employee's salary by quarter and a two-dimensional array of *text* (schedule), which represents the employee's weekly schedule. Now we do some *INSERTS*s; note that when appending to an array, we enclose the values within braces and separate them by commas. If you know *C*, this is not unlike the syntax for initializing structures.

```
INSERT INTO SAL_EMP
    VALUES ('Bill',
    '{10000, 10000, 10000, 10000}',
    '{{"meeting", "lunch"}, {}}');

INSERT INTO SAL_EMP
    VALUES ('Carol',
    '{20000, 25000, 25000, 25000}',
    '{{"talk", "consult"}, {"meeting"}}');
```

By default, Postgres uses the "one-based" numbering convention for arrays -- that is, an array of n elements starts with array[1] and ends with array[n]. Now, we can run some queries on SAL_EMP.

First, we show how to access a single element of an array at a time. This query retrieves the names of the employees whose pay changed in the second quarter:

```
SELECT name
    FROM SAL_EMP
    WHERE SAL_EMP.pay_by_quarter[1] <>
    SAL_EMP.pay_by_quarter[2];

+------+
|name  |
+------+
|Carol |
+------+
```

This query retrieves the third quarter pay of all employees:

```
SELECT SAL_EMP.pay_by_quarter[3] FROM SAL_EMP;


+---------------+
|pay_by_quarter |
+---------------+
|10000          |
+---------------+
|25000          |
+---------------+
```

We can also access arbitrary slices of an array, or subarrays. This query retrieves the first item on Bill's schedule for the first two days of the week.

```
SELECT SAL_EMP.schedule[1:2][1:1]
    FROM SAL_EMP
    WHERE SAL_EMP.name = 'Bill';

+-------------------+
|schedule           |
+-------------------+
|{{"meeting"},{""}} |
+-------------------+
```

74.3. More Advanced Features

Postgres has many features not touched upon in this tutorial introduction, which has been oriented toward newer users of SQL. These are discussed in more detail in both the User's and Programmer's Guides.

Part VII. Appendices

Additional related information.

Table of Contents

UG1. Date/Time Support 708
UG1.1. Time Zones 708
UG1.1.1. Australian Time Zones 710
UG1.1.2. Date/Time Input Interpretation 710
UG1.2. History of Units 711
DG1. The CVS Repository 713
DG1.1. CVS Tree Organization 713
DG1.2. Getting The Source Via Anonymous CVS 714
DG1.3. Getting The Source Via CVSup 715
DG1.3.1. Preparing A CVSup Client System 716
DG1.3.2. Running a CVSup Client 716
DG1.3.3. Installing CVSup 718
DG1.3.4. Installation from Sources 718
DG2. Documentation 720
DG2.1. Documentation Roadmap 720
DG2.2. The Documentation Project 720
DG2.3. Documentation Sources 721
DG2.3.1. Document Structure 721
DG2.3.2. Styles and Conventions 722
DG2.3.3. SGML Authoring Tools 722
DG2.4. Building Documentation 723
DG2.5. Manpages 724
DG2.6. Hardcopy Generation for v7.0 724
DG2.6.1. Text Hardcopy 724
DG2.6.2. Postscript Hardcopy 725
DG2.7. Toolsets 726
DG2.7.1. Linux RPM Installation 727
DG2.7.2. FreeBSD Installation 727
DG2.7.3. Debian Installation 728
DG2.7.4. Manual Installation of Tools 728
DG2.8. Alternate Toolsets 733
Bibliography 734

Appendix UG1. Date/Time Support

UG1.1. Time Zones

Postgres must have internal tabular information for time zone decoding, since there is no *nix standard system interface to provide access to general, cross-timezone information. The underlying OS *is* used to provide time zone information for *output*.

Table UG1.1. Postgres Recognized Time Zones

Time Zone	Offset from UTC	Description
NZDT	+13:00	New Zealand Daylight Time
IDLE	+12:00	International Date Line, East
NZST	+12:00	New Zealand Std Time
NZT	+12:00	New Zealand Time
AESST	+11:00	Australia Eastern Summer Std Time
ACSST	+10:30	Central Australia Summer Std Time
CADT	+10:30	Central Australia Daylight Savings Time
SADT	+10:30	South Australian Daylight Time
AEST	+10:00	Australia Eastern Std Time
EAST	+10:00	East Australian Std Time
GST	+10:00	Guam Std Time, USSR Zone 9
LIGT	+10:00	Melbourne, Australia
ACST	+09:30	Central Australia Std Time
CAST	+09:30	Central Australia Std Time
SAT	+9:30	South Australian Std Time
AWSST	+9:00	Australia Western Summer Std Time
JST	+9:00	Japan Std Time,USSR Zone 8
KST	+9:00	Korea Standard Time
WDT	+9:00	West Australian Daylight Time
MT	+8:30	Moluccas Time
AWST	+8:00	Australia Western Std Time
CCT	+8:00	China Coastal Time
WADT	+8:00	West Australian Daylight Time
WST	+8:00	West Australian Std Time
JT	+7:30	Java Time
WAST	+7:00	West Australian Std Time
IT	+3:30	Iran Time
BT	+3:00	Baghdad Time
EETDST	+3:00	Eastern Europe Daylight Savings Time

Time Zone	Offset from UTC	Description
CETDST	+2:00	Central European Daylight Savings Time
EET	+2:00	Eastern Europe, USSR Zone 1
FWT	+2:00	French Winter Time
IST	+2:00	Israel Std Time
MEST	+2:00	Middle Europe Summer Time
METDST	+2:00	Middle Europe Daylight Time
SST	+2:00	Swedish Summer Time
BST	+1:00	British Summer Time
CET	+1:00	Central European Time
DNT	+1:00	Dansk Normal Tid
DST	+1:00	Dansk Standard Time (?)
FST	+1:00	French Summer Time
MET	+1:00	Middle Europe Time
MEWT	+1:00	Middle Europe Winter Time
MEZ	+1:00	Middle Europe Zone
NOR	+1:00	Norway Standard Time
SET	+1:00	Seychelles Time
SWT	+1:00	Swedish Winter Time
WETDST	+1:00	Western Europe Daylight Savings Time
GMT	0:00	Greenwish Mean Time
WET	0:00	Western Europe
WAT	-1:00	West Africa Time
NDT	-2:30	Newfoundland Daylight Time
ADT	-03:00	Atlantic Daylight Time
NFT	-3:30	Newfoundland Standard Time
NST	-3:30	Newfoundland Standard Time
AST	-4:00	Atlantic Std Time (Canada)
EDT	-4:00	Eastern Daylight Time
ZP4	-4:00	GMT +4 hours
CDT	-5:00	Central Daylight Time
EST	-5:00	Eastern Standard Time
ZP5	-5:00	GMT +5 hours
CST	-6:00	Central Std Time
MDT	-6:00	Mountain Daylight Time
ZP6	-6:00	GMT +6 hours
MST	-7:00	Mountain Standard Time
PDT	-7:00	Pacific Daylight Time
PST	-8:00	Pacific Std Time
YDT	-8:00	Yukon Daylight Time

Time Zone	Offset from UTC	Description
HDT	-9:00	Hawaii/Alaska Daylight Time
YST	-9:00	Yukon Standard Time
AHST	-10:00	Alaska-Hawaii Std Time
CAT	-10:00	Central Alaska Time
NT	-11:00	Nome Time
IDLW	-12:00	International Date Line, West

UG1.1.1. Australian Time Zones

Australian time zones and their naming variants account for fully one quarter of all time zones in the Postgres time zone lookup table. There are two naming conflicts with common time zones defined in the United States, `CST` and `EST`.

If the compiler option USE_AUSTRALIAN_RULES is set then `CST` and `EST` will be interpreted using Australian conventions.

Table UG1.2. Postgres Australian Time Zones

Time Zone	Offset from UTC	Description
CST	+10:30	Australian Central Standard Time
EST	+10:00	Australian Eastern Standard Time

UG1.1.2. Date/Time Input Interpretation

The date/time types are all decoded using a common set of routines.

Date/Time Input Interpretation

1. Break the input string into tokens and categorize each token as a string, time, time zone, or number.
    a. If the token contains a colon (":"), this is a time string.
    b. If the token contains a dash ("-"), slash ("/"), or dot ("."), this is a date string which may have a text month.
    c. If the token is numeric only, then it is either a single field or an ISO-8601 concatenated date (e.g. "19990113" for January 13, 1999) or time (e.g. 141516 for 14:15:16).
    d. If the token starts with a plus ("+") or minus ("-"), then it is either a time zone or a special field.
2. If the token is a text string, match up with possible strings.
    a. Do a binary-search table lookup for the token as either a special string (e.g. `today`), day (e.g. `Thursday`), month (e.g. `January`), or noise word (e.g. `on`).

       Set field values and bit mask for fields. For example, set year, month, day for `today`, and additionally hour, minute, second for `now`.
    b. If not found, do a similar binary-search table lookup to match the token with a time zone.
    c. If not found, throw an error.

3. The token is a number or number field.

    a. If there are more than 4 digits, and if no other date fields have been previously read, then interpret as a "concatenated date" (e.g. `19990118`). 8 and 6 digits are interpreted as year, month, and day, while 7 and 5 digits are interpreted as year, day of year, respectively.

    b. If the token is three digits and a year has already been decoded, then interpret as day of year.

    c. If four or more digits, then interpret as a year.

    d. If in European date mode, and if the day field has not yet been read, and if the value is less than or equal to 31, then interpret as a day.

    e. If the month field has not yet been read, and if the value is less than or equal to 12, then interpret as a month.

    f. If the day field has not yet been read, and if the value is less than or equal to 31, then interpret as a day.

    g. If two digits or four or more digits, then interpret as a year.

    h. Otherwise, throw an error.

4. If BC has been specified, negate the year and offset by one for internal storage (there is no year zero in the Gregorian calendar, so numerically 1BC becomes year zero).

5. If BC was not specified, and if the year field was two digits in length, then adjust the year to 4 digits. If the field was less than 70, then add 2000; otherwise, add 1900.

    ### Tip

    Gregorian years 1-99AD may be entered by using 4 digits with leading zeros (e.g. 0099 is 99AD). Previous versions of Postgres accepted years with three digits and with single digits, but as of v7.0 the rules have been tightened up to reduce the possibility of ambiguity.

UG1.2. History of Units

Note

Contributed by José Soares[1].

The Julian Day was invented by the French scholar Joseph Justus Scaliger (1540-1609) and probably takes its name from the Scaliger's father, the Italian scholar Julius Caesar Scaliger (1484-1558). Astronomers have used the Julian period to assign a unique number to every day since 1 January 4713 BC. This is the so-called Julian Day (JD). JD 0 designates the 24 hours from noon UTC on 1 January 4713 BC to noon UTC on 2 January 4713 BC.

"Julian Day" is different from "Julian Date". The Julian calendar was introduced by Julius Caesar in 45 BC. It was in common use until the 1582, when countries started changing to the Gregorian calendar. In the Julian calendar, the tropical year is approximated as 365 1/4 days = 365.25 days. This gives an error of 1 day in approximately 128 days. The accumulating calendar error prompted pope Gregory XIII to reform the calendar in accordance with instructions from the Council of Trent.

In the Gregorian calendar, the tropical year is approximated as 365 + 97 / 400 days = 365.2425 days. Thus it takes approximately 3300 years for the tropical year to shift one day with respect to the Gregorian calendar.

[1] jose@sferacarta.com

The approximation 365+97/400 is achieved by having 97 leap years every 400 years, using the following rules:

Every year divisible by 4 is a leap year.
However, every year divisible by 100 is not a leap year.
However, every year divisible by 400 is a leap year after all.
So, 1700, 1800, 1900, 2100, and 2200 are not leap years. But 1600, 2000, and 2400 are leap years. By contrast, in the older Julian calendar only years divisible by 4 are leap years.

The papal bull of February 1582 decreed that 10 days should be dropped from October 1582 so that 15 October should follow immediately after 4 October. This was observed in Italy, Poland, Portugal, and Spain. Other Catholic countries followed shortly after, but Protestant countries were reluctant to change, and the Greek orthodox countries didn't change until the start of this century. The reform was observed by Great Britain and Dominions (including what is now the USA) in 1752. Thus 2 Sep 1752 was followed by 14 Sep 1752. This is why Unix systems have cal produce the following:

```
% cal 9 1752
   September 1752
 S  M Tu  W Th  F  S
       1  2 14 15 16
17 18 19 20 21 22 23
24 25 26 27 28 29 30
```

Note

SQL92 states that “Within the definition of a `datetime literal`, the `datetime values` are constrained by the natural rules for dates and times according to the Gregorian calendar”. Dates between 1752-09-03 and 1752-09-13, although eliminated in some countries by Papal fiat, conform to “natural rules” and are hence valid dates.

Different calendars have been developed in various parts of the world, many predating the Gregorian system. For example, the beginnings of the Chinese calendar can be traced back to the 14th century BC. Legend has it that the Emperor Huangdi invented the calendar in 2637 BC. The People's Republic of China uses the Gregorian calendar for civil purposes. Chinese calendar is used for determining festivals.

Appendix DG1. The CVS Repository

Marc Fournier
Tom Lane
Thomas Lockhart

The Postgres source code is stored and managed using the CVS code management system.

At least two methods, anonymous CVS and CVSup, are available to pull the CVS code tree from the Postgres server to your local machine.

DG1.1. CVS Tree Organization

Author

Written by Marc G. Fournier[1] on 1998-11-05.

The command `cvs checkout` has a flag, `-r`, that lets you check out a certain revision of a module. This flag makes it easy to, for example, retrieve the sources that make up release 1.0 of the module `tc' at any time in the future:

```
$ cvs checkout -r REL6_4 tc
```

This is useful, for instance, if someone claims that there is a bug in that release, but you cannot find the bug in the current working copy.

Tip

You can also check out a module as it was at any given date using the `-D` option.

When you tag more than one file with the same tag you can think about the tag as "a curve drawn through a matrix of filename vs. revision number". Say we have 5 files with the following revisions:

```
file1   file2   file3   file4   file5

1.1     1.1     1.1     1.1  /--1.1*      <-*-  TAG
1.2*-   1.2     1.2    -1.2*-
1.3  \- 1.3*-   1.3   / 1.3
1.4          \  1.4  /  1.4
              \-1.5*-   1.5
                1.6
```

then the tag "`TAG`" will reference file1-1.2, file2-1.3, etc.

Note

For creating a release branch, other then a -b option added to the command, it's the same thing.

So, to create the v6.4 release I did the following:

```
$ cd pgsql
$ cvs tag -b REL6_4
```

[1] mailto:scrappy@hub.org

which will create the tag and the branch for the RELEASE tree.

Now, for those with CVS access, it's too simple. First, create two subdirectories, RELEASE and CURRENT, so that you don't mix up the two. Then do:

```
cd RELEASE
cvs checkout -P -r REL6_4 pgsql
cd ../CURRENT
cvs checkout -P pgsql
```

which results in two directory trees, `RELEASE/pgsql` and `CURRENT/pgsql`. From that point on, CVS will keep track of which repository branch is in which directory tree, and will allow independent updates of either tree.

If you are *only* working on the `CURRENT` source tree, you just do everything as before we started tagging release branches.

After you've done the initial checkout on a branch

```
$ cvs checkout -r REL6_4
```

anything you do within that directory structure is restricted to that branch. If you apply a patch to that directory structure and do a

```
cvs commit
```

while inside of it, the patch is applied to the branch and *only* the branch.

DG1.2. Getting The Source Via Anonymous CVS

If you would like to keep up with the current sources on a regular basis, you can fetch them from our CVS server and then use CVS to retrieve updates from time to time.

Anonymous CVS

1. You will need a local copy of CVS (Concurrent Version Control System), which you can get from http://www.cyclic.com/ or any GNU software archive site. We currently recommend version 1.10 (the most recent at the time of writing). Many systems have a recent version of cvs installed by default.

2. Do an initial login to the CVS server:

   ```
   $ cvs -d :pserver:anoncvs@postgresql.org:/usr/local/cvsroot
    login
   ```

   You will be prompted for a password; enter '`postgresql`'. You should only need to do this once, since the password will be saved in `.cvspass` in your home directory.

3. Fetch the Postgres sources:

   ```
   cvs -z3 -d :pserver:anoncvs@postgresql.org:/usr/local/cvsroot
    co -P pgsql
   ```

which installs the Postgres sources into a subdirectory `pgsql` of the directory you are currently in.

Note

If you have a fast link to the Internet, you may not need `-z3`, which instructs CVS to use gzip compression for transferred data. But on a modem-speed link, it's a very substantial win.

This initial checkout is a little slower than simply downloading a `tar.gz` file; expect it to take 40 minutes or so if you have a 28.8K modem. The advantage of CVS doesn't show up until you want to update the file set later on.

4. Whenever you want to update to the latest CVS sources, `cd` into the `pgsql` subdirectory, and issue

```
$ cvs -z3 update -d -P
```

This will fetch only the changes since the last time you updated. You can update in just a couple of minutes, typically, even over a modem-speed line.

5. You can save yourself some typing by making a file `.cvsrc` in your home directory that contains

```
cvs -z3
update -d -P
```

This supplies the `-z3` option to all cvs commands, and the `-d` and `-P` options to cvs update. Then you just have to say

```
$ cvs update
```

to update your files.

Caution

Some older versions of CVS have a bug that causes all checked-out files to be stored world-writable in your directory. If you see that this has happened, you can do something like

```
$ chmod -R go-w pgsql
```

to set the permissions properly. This bug is fixed as of CVS version 1.9.28.

CVS can do a lot of other things, such as fetching prior revisions of the Postgres sources rather than the latest development version. For more info consult the manual that comes with CVS, or see the online documentation at http://www.cyclic.com/.

DG1.3. Getting The Source Via CVSup

An alternative to using anonymous CVS for retrieving the Postgres source tree is CVSup. CVSup was developed by John Polstra[2] to distribute CVS repositories and other file trees for the FreeBSD project[3].

A major advantage to using CVSup is that it can reliably replicate the *entire* CVS repository on your local system, allowing fast local access to cvs operations such as `log` and `diff`. Other advantages

[2] mailto:jdp@polstra.com
[3] http://www.freebsd.org

include fast synchronization to the Postgres server due to an efficient streaming transfer protocol which only sends the changes since the last update.

DG1.3.1. Preparing A CVSup Client System

Two directory areas are required for CVSup to do it's job: a local CVS repository (or simply a directory area if you are fetching a snapshot rather than a repository; see below) and a local CVSup bookkeeping area. These can coexist in the same directory tree.

Decide where you want to keep your local copy of the CVS repository. On one of our systems we recently set up a repository in `/home/cvs/`, but had formerly kept it under a Postgres development tree in `/opt/postgres/cvs/`. If you intend to keep your repository in `/home/cvs/`, then put

```
setenv CVSROOT /home/cvs
```

in your `.cshrc` file, or a similar line in your `.bashrc` or `.profile` file, depending on your shell.

The cvs repository area must be initialized. Once `CVSROOT` is set, then this can be done with a single command:

```
$ cvs init
```

after which you should see at least a directory named `CVSROOT` when listing the `CVSROOT` directory:

```
$ ls $CVSROOT
CVSROOT/
```

DG1.3.2. Running a CVSup Client

Verify that cvsup is in your path; on most systems you can do this by typing

```
which cvsup
```

Then, simply run cvsup using:

```
$ cvsup -L 2 postgres.cvsup
```

where `-L 2` enables some status messages so you can monitor the progress of the update, and *`postgres.cvsup`* is the path and name you have given to your CVSup configuration file.

Here is a CVSup configuration file modified for a specific installation, and which maintains a full local CVS repository:

```
# This file represents the standard CVSup distribution file
# for the PostgreSQL ORDBMS project
# Modified by lockhart@alumni.caltech.edu 1997-08-28
# - Point to my local snapshot source tree
# - Pull the full CVS repository, not just the latest snapshot
#
# Defaults that apply to all the collections
*default host=postgresql.org
*default compress
*default release=cvs
*default delete use-rel-suffix
```

```
# enable the following line to get the latest snapshot
#*default tag=.
# enable the following line to get whatever was specified above or
 by default
# at the date specified below
#*default date=97.08.29.00.00.00

# base directory points to where CVSup will store its 'bookmarks'
 file(s)
# will create subdirectory sup/
#*default base=/opt/postgres # /usr/local/pgsql
*default base=/home/cvs

# prefix directory points to where CVSup will store the actual
 distribution(s)
*default prefix=/home/cvs

# complete distribution, including all below
pgsql

# individual distributions vs 'the whole thing'
# pgsql-doc
# pgsql-perl5
# pgsql-src
```

The following is a suggested CVSup config file from the Postgres ftp site[4] which will fetch the current snapshot only:

```
# This file represents the standard CVSup distribution file
# for the PostgreSQL ORDBMS project
#
# Defaults that apply to all the collections
*default host=postgresql.org
*default compress
*default release=cvs
*default delete use-rel-suffix
*default tag=.

# base directory points to where CVSup will store its 'bookmarks'
 file(s)
*default base=/usr/local/pgsql

# prefix directory points to where CVSup will store the actual
 distribution(s)
*default prefix=/usr/local/pgsql

# complete distribution, including all below
pgsql

# individual distributions vs 'the whole thing'
# pgsql-doc
# pgsql-perl5
# pgsql-src
```

[4] ftp://ftp.postgresql.org/pub/CVSup/README.cvsup

DG1.3.3. Installing CVSup

CVSup is available as source, pre-built binaries, or Linux RPMs. It is far easier to use a binary than to build from source, primarily because the very capable, but voluminous, Modula-3 compiler is required for the build.

CVSup Installation from Binaries

You can use pre-built binaries if you have a platform for which binaries are posted on the Postgres ftp site[5], or if you are running FreeBSD, for which CVSup is available as a port.

> **Note**
>
> CVSup was originally developed as a tool for distributing the FreeBSD source tree. It is available as a "port", and for those running FreeBSD, if this is not sufficient to tell how to obtain and install it then please contribute a procedure here.

At the time of writing, binaries are available for Alpha/Tru64, ix86/xBSD, HPPA/HPUX-10.20, MIPS/irix, ix86/linux-libc5, ix86/linux-glibc, Sparc/Solaris, and Sparc/SunOS.

1. Retrieve the binary tar file for cvsup (cvsupd is not required to be a client) appropriate for your platform.

   a. (Optional) If you are running FreeBSD, install the CVSup port.

   b. (Optional) If you have another platform, check for and download the appropriate binary from the Postgres ftp site[6].

2. Check the tar file to verify the contents and directory structure, if any. For the linux tar file at least, the static binary and man page is included without any directory packaging.

   a. If the binary is in the top level of the tar file, then simply unpack the tar file into your target directory:

   ```
   $ cd /usr/local/bin
   $ tar zxvf /usr/local/src/cvsup-16.0-linux-i386.tar.gz
   $ mv cvsup.1 ../doc/man/man1/
   ```

   b. If there is a directory structure in the tar file, then unpack the tar file within /usr/local/src and move the binaries into the appropriate location as above.

3. Ensure that the new binaries are in your path.

   ```
   $ rehash
   $ which cvsup
   $ set path=(path to cvsup $path)
   $ which cvsup
   /usr/local/bin/cvsup
   ```

DG1.3.4. Installation from Sources

Installing CVSup from sources is not entirely trivial, primarily because most systems will need to install a Modula-3 compiler first. This compiler is available as Linux RPM, FreeBSD package, or source code.

[5] ftp://postgresql.org/pub
[6] ftp://postgresql.org/pub

Note

A clean-source installation of Modula-3 takes roughly 200MB of disk space, which shrinks to roughly 50MB of space when the sources are removed.

Linux installation

1. Install Modula-3.

   a. Pick up the Modula-3 distribution from Polytechnique Montréal[7], who are actively maintaining the code base originally developed by the DEC Systems Research Center[8]. The PM3 RPM distribution is roughly 30MB compressed. At the time of writing, the 1.1.10-1 release installed cleanly on RH-5.2, whereas the 1.1.11-1 release is apparently built for another release (RH-6.0?) and does not run on RH-5.2.

      ## Tip

      This particular rpm packaging has *many* RPM files, so you will likely want to place them into a separate directory.

   b. Install the Modula-3 rpms:

      ```
      # rpm -Uvh pm3*.rpm
      ```

2. Unpack the cvsup distribution:

   ```
   # cd /usr/local/src
   # tar zxf cvsup-16.0.tar.gz
   ```

3. Build the cvsup distribution, suppressing the GUI interface feature to avoid requiring X11 libraries:

   ```
   # make M3FLAGS="-DNOGUI"
   ```

   and if you want to build a static binary to move to systems which may not have Modula-3 installed, try:

   ```
   # make M3FLAGS="-DNOGUI -DSTATIC"
   ```

4. Install the built binary:

   ```
   # make M3FLAGS="-DNOGUI -DSTATIC" install
   ```

[7] http://m3.polymtl.ca/m3
[8] http://www.research.digital.com/SRC/modula-3/html/home.html

Appendix DG2. Documentation

Thomas Lockhart
Tom Ivar Helbekkmo

The purpose of documentation is to make Postgres easier to learn, use, and extend.. The documentation set should describe the Postgres system, language, and interfaces. It should be able to answer common questions and to allow a user to find those answers on his own without resorting to mailing list support.

DG2.1. Documentation Roadmap

Postgres has four primary documentation formats:

- Plain text for pre-installation information.
- HTML, for on-line browsing and reference.
- Hardcopy (Postscript or PDF), for in-depth reading and reference.
- man pages, for quick reference.

Table DG2.1. Postgres Documentation Products

File	Description
./COPYRIGHT	Copyright notice
./INSTALL	Installation instructions (text from sgml->rtf->text)
./README	Introductory info
./register.txt	Registration message during make
./doc/bug.template	Bug report template
./doc/postgres.tar.gz	Integrated docs (HTML)
./doc/programmer.ps.gz	Programmer's Guide (Postscript)
./doc/programmer.tar.gz	Programmer's Guide (HTML)
./doc/reference.ps.gz	Reference Manual (Postscript)
./doc/reference.tar.gz	Reference Manual (HTML)
./doc/tutorial.ps.gz	Introduction (Postscript)
./doc/tutorial.tar.gz	Introduction (HTML)
./doc/user.ps.gz	User's Guide (Postscript)
./doc/user.tar.gz	User's Guide (HTML)

There are man pages available, as well as a large number of plain-text README-type files throughout the Postgres source tree.

DG2.2. The Documentation Project

Packaged documentation is available in both HTML and *Postscript* formats. These are available as part of the standard Postgres installation. We discuss here working with the documentation sources and generating documentation packages.

The documentation sources are written using SGML markup of plain text files. The purpose of DocBook SGML is to allow an author to specify the structure and content of a technical document (using the DocBook DTD), and to have a document style define how that content is rendered into a final form (e.g. using Norm Walsh's Modular Style Sheets).

See Introduction to DocBook[1] for a nice "quickstart" summary of DocBook features. DocBook Elements[2] provides a powerful cross-reference for features of DocBook.

This documentation set is constructed using several tools, including James Clark's jade[3] and Norm Walsh's Modular DocBook Stylesheets[4].

Currently, hardcopy is produced by importing *Rich Text Format* (RTF) output from jade into ApplixWare for minor formatting fixups, then exporting as a Postscript file.

TeX[5] is a supported format for jade output, but is not used at this time for several reasons, including the inability to make minor format fixes before committing to hardcopy and generally inadequate table support in the TeX stylesheets.

DG2.3. Documentation Sources

Documentation sources include plain text files, man pages, and html. However, most new Postgres documentation will be written using the *Standard Generalized Markup Language* (SGML) DocBook[6] *Document Type Definition* (DTD). Much of the existing documentation has been or will be converted to SGML.

The purpose of SGML is to allow an author to specify the structure and content of a document (e.g. using the DocBook DTD), and to have the document style define how that content is rendered into a final form (e.g. using Norm Walsh's stylesheets).

Documentation has accumulated from several sources. As we integrate and assimilate existing documentation into a coherent documentation set, the older versions will become obsolete and will be removed from the distribution. However, this will not happen immediately, and will not happen to all documents at the same time. To ease the transition, and to help guide developers and writers, we have defined a transition roadmap.

DG2.3.1. Document Structure

There are currently five separate documents written in DocBook. Each document has a container source document which defines the DocBook environment and other document source files. These primary source files are located in `doc/src/sgml/`, along with many of the other source files used for the documentation. The primary source files are:

postgres.sgml

> This is the integrated document, including all other documents as parts. Output is generated in HTML since the browser interface makes it easy to move around all of the documentation by just clicking. The other documents are available in both HTML and hardcopy.

tutorial.sgml

> The introductory tutorial, with examples. Does not include programming topics, and is intended to help a reader unfamiliar with SQL. This is the "getting started" document.

user.sgml

> The User's Guide. Includes information on data types and user-level interfaces. This is the place to put information on "why".

[1] http://nis-www.lanl.gov/~rosalia/mydocs/docbook-intro.html
[2] http://www.ora.com/homepages/dtdparse/docbook/3.0/
[3] http://www.jclark.com/jade/
[4] http://www.nwalsh.com/docbook/dsssl/
[5] http://sunsite.unc.edu/pub/packages/TeX/systems/unix/
[6] http://www.ora.com/davenport/

reference.sgml

The Reference Manual. Includes Postgres SQL syntax. This is the place to put information on "how".

programming.sgml

The Programmer's Guide. Includes information on Postgres extensibility and on the programming interfaces.

admin.sgml

The Administrator's Guide. Include installation and release notes.

DG2.3.2. Styles and Conventions

DocBook has a rich set of tags and constructs, and a suprisingly large percentage are directly and obviously useful for well-formed documentation. The Postgres documentation set has only recently been adapted to SGML, and in the near future several sections of the set will be selected and maintained as prototypical examples of DocBook usage. Also, a short summary of DocBook tags will be included below.

DG2.3.3. SGML Authoring Tools

The current Postgres documentation set was written using a plain text editor (or emacs/psgml; see below) with the content marked up using SGML DocBook tags.

SGML and DocBook do not suffer from an oversupply of open-source authoring tools. The most common toolset is the emacs/xemacs editing package with the psgml feature extension. On some systems (e.g. RedHat Linux) these tools are provided in a typical full installation.

DG2.3.3.1. emacs/psgml

emacs (and xemacs) have an SGML *major mode*. When properly configured, this will allow you to use emacs to insert tags and check markup consistancy.

Put the following in your `~/.emacs` environment file (adjusting the path names to be appropriate for your system):

```
; ********** for SGML mode (psgml)

(setq sgml-catalog-files "/usr/lib/sgml/CATALOG")
(setq sgml-local-catalogs "/usr/lib/sgml/CATALOG")

(autoload 'sgml-mode "psgml" "Major mode to edit SGML files." t )
```

and add an entry in the same file for SGML into the (existing) definition for auto-mode-alist:

```
(setq
  auto-mode-alist
  '(("\\.sgml$" . sgml-mode)
   ))
```

Each SGML source file has the following block at the end of the file:

```
!-- Keep this comment at the end of the file
Local variables:
mode: sgml
```

```
sgml-omittag:t
sgml-shorttag:t
sgml-minimize-attributes:nil
sgml-always-quote-attributes:t
sgml-indent-step:1
sgml-indent-data:t
sgml-parent-document:nil
sgml-default-dtd-file:"./reference.ced"
sgml-exposed-tags:nil
sgml-local-catalogs:("/usr/lib/sgml/catalog")
sgml-local-ecat-files:nil
End:
--
```

The Postgres distribution includes a parsed DTD definitions file `reference.ced`. You may find that

When using emacs/psgml, a comfortable way of working with these separate files of book parts is to insert a proper DOCTYPE declaration while you're editing them. If you are working on this source, for instance, it's an appendix chapter, so you would specify the document as an "appendix" instance of a DocBook document by making the first line look like this:

```
!doctype appendix PUBLIC "-//Davenport//DTD DocBook V3.0//EN"
```

This means that anything and everything that reads SGML will get it right, and I can verify the document with "nsgmls -s docguide.sgml".

DG2.4. Building Documentation

GNU make is used to build documentation from the DocBook sources. There are a few environment definitions which may need to be set or modified for your installation. The `Makefile` looks for `doc/../src/Makefile` and (implicitly) for `doc/../src/Makefile.custom` to obtain environment information. On my system, the `src/Makefile.custom` looks like

```
# Makefile.custom
# Thomas Lockhart 1998-03-01

POSTGRESDIR= /opt/postgres/current
CFLAGS+= -m486
YFLAGS+= -v

# documentation

HSTYLE= /home/lockhart/SGML/db143.d/docbook/html
PSTYLE= /home/lockhart/SGML/db143.d/docbook/print
```

where HSTYLE and PSTYLE determine the path to `docbook.dsl` for HTML and hardcopy (print) stylesheets, respectively. These stylesheet file names are for Norm Walsh's Modular Style Sheets; if other stylesheets are used then one can define HDSL and PDSL as the full path and file name for the stylesheet, as is done above for HSTYLE and PSTYLE. On many systems, these stylesheets will be found in packages installed in `/usr/lib/sgml/`, `/usr/share/lib/sgml/`, or `/usr/local/lib/sgml/`.

HTML documentation packages can be generated from the SGML source by typing

```
% cd doc/src
% make tutorial.tar.gz
```

```
% make user.tar.gz
% make admin.tar.gz
% make programmer.tar.gz
% make postgres.tar.gz
% make install
```

These packages can be installed from the main documentation directory by typing

```
% cd doc
% make install
```

DG2.5. Manpages

We use the docbook2man utility to convert DocBook `REFENTRY` pages to *roff output suitable for man pages. At the time of writing, the utility required patching to successfully run on the Postgres markup, and we added a small amount of new functionality to allow setting the man page section in the output file name.

docbook2man is written in perl, and requires the CPAN package `SGMLSpm` to run. Also, it requires nsgmls to be available, which is included in the jade distribution. After installing these packages, then simply run

```
$ cd doc/src
$ make man
```

which will result in a tar file being generated in the `doc/src` directory.

docbook2man Installation Procedure

1. Install the docbook2man package, available at http://shell.ipoline.com/~elmert/comp/docbook2X/
2. Install the SGMLSpm perl module, available from CPAN mirrors.
3. Install nsgmls if not already available from your jade installation.

DG2.6. Hardcopy Generation for v7.0

The hardcopy Postscript documentation is generated by converting the SGML source code to RTF, then importing into ApplixWare-4.4.1. After a little cleanup (see the following section) the output is "printed" to a postscript file.

DG2.6.1. Text Hardcopy

`INSTALL` and `HISTORY` are updated for each release. For historical reasons, these files are in plain text, but are derived from the newer SGML sources.

Plain Text Generation

Both `INSTALL` and `HISTORY` are generated from existing SGML sources. They are extracted from the same intermediate RTF file.

1. Generate RTF by typing:

   ```
   % cd doc/src/sgml
   % make installation.rtf
   ```

2. Import `installation.rtf` into Applix Words.
3. Set the page width and margins.
   a. Adjust the page width in File.PageSetup to 10 inches.
   b. Select all text. Adjust the right margin using the ruler to 9.5 inches. This will give a maximum column width of 79 characters, within the 80 columns upper limit goal.
4. Lop off the parts of the document which are not needed.

   For `INSTALL`, remove all release notes from the end of the text, except for those from the current release. For `HISTORY`, remove all text up to the release notes, preserving and modifying the title and ToC.
5. Export the result as "ASCII Layout".
6. Using emacs or vi, clean up the tabular information in `INSTALL`. Remove the "mailto" URLs for the porting contributors to shrink the column heights.

DG2.6.2. Postscript Hardcopy

Several areas are addressed while generating Postscript hardcopy, including RTF repair, ToC generation, and page break adjustments.

Applixware RTF Cleanup

jade, an integral part of the hardcopy procedure, omits specifying a default style for body text. In the past, this undiagnosed problem led to a long process of Table of Contents (ToC) generation. However, with great help from the ApplixWare folks the symptom was diagnosed and a workaround is available.

1. Generate the RTF input by typing (for example):

```
% cd doc/src/sgml
% make tutorial.rtf
```

2. Repair the RTF file to correctly specify all styles, in particular the default style. The field can be added using vi or the following small sed procedure:

```
#!/bin/sh
# fixrtf.sh
# Utility to repair slight damage in RTF files generated by
 jade
# Thomas Lockhart <lockhart@alumni.caltech.edu>
#
for i in $* ; do
  mv $i $i.orig
  cat $i.orig | sed 's#\\stylesheet#\\stylesheet{\\s0
 Normal;}#' > $i
done

exit
```

   where the script is adding `{\s0 Normal;}` as the zero-th style in the document. According to ApplixWare, the RTF standard would prohibit adding an implicit zero-th style, though M$Word happens to handle this case.

3. Open a new document in Applix Words and then import the RTF file.

4. Generate a new ToC using ApplixWare.

   a. Select the existing ToC lines, from the beginning of the first character on the first line to the last character of the last line.

   b. Build a new ToC using `Tools.BookBuilding.CreateToC`. Select the first three levels of headers for inclusion in the ToC. This will replace the existing lines imported in the RTF with a native ApplixWare ToC.

   c. Adjust the ToC formatting by using `Format.Style`, selecting each of the three ToC styles, and adjusting the indents for `First` and `Left`. Use the following values:

Table DG2.2. Indent Formatting for Table of Contents

Style	First Indent (inches)	Left Indent (inches)
`TOC-Heading 1`	`0.6`	`0.6`
`TOC-Heading 2`	`1.0`	`1.0`
`TOC-Heading 3`	`1.4`	`1.4`

5. Work through the document to:

   - Adjust page breaks.
   - Adjust table column widths.
   - Insert figures into the document. Center each figure on the page using the centering margins button on the ApplixWare toolbar.

> **Note**
>
> Not all documents have figures. You can grep the SGML source files for the string "`graphic`" to identify those parts of the documentation which may have figures. A few figures are replicated in various parts of the documentation.

6. Replace the right-justified page numbers in the Examples and Figures portions of the ToC with correct values. This only takes a few minutes per document.

7. If a bibliography is present, remove the *short form* reference title from each entry. The DocBook stylesheets from Norm Walsh seem to print these out, even though this is a subset of the information immediately following.

8. Save the document as native Applix Words format to allow easier last minute editing later.

9. "Print" the document to a file in Postscript format.

10. Compress the Postscript file using gzip. Place the compressed file into the `doc` directory.

DG2.7. Toolsets

We have documented experience with three installation methods for the various tools that are needed to process the documentation. One is installation from RPMs on Linux, the second is installation from FreeBSD *port*, and the last is a general installation from original distributions of the individual tools. These will be described below.

There may be some other packaged distributions for these tools. Please report package status to the docs mailing list and we will include that information here.

DG2.7.1. Linux RPM Installation

The simplest installation for a RedHat-compatible Linux system uses the RPM set developed by Mark Galassi at Cygnus. It should also be possible to install from sources, as described in a subsequent section.

Installing RPMs

1. Install RPMs[7] for Jade and related packages.

2. (Optional) Install Norm Walsh's latest style sheets. Depending on the age of the RPMs, the latest style sheets may be substantially improved from those contained in the RPMs.

3. Update your `src/Makefile.custom` to include HSTYLE and PSTYLE definitions pointing to the style sheets.

DG2.7.2. FreeBSD Installation

There is a full set of *ports* of the documentation tools available on FreeBSD. In fact, postgresql.org, on which documentation is automatically updated every evening, is a FreeBSD machine.

Installing FreeBSD Ports

1. To build the documentation on FreeBSD a number of ports will need to be installed.

```
% cd /usr/ports/devel/gmake && make install
% cd /usr/ports/textproc/docproj && make install
% cd /usr/ports/textproc/docbook && make install
% cd /usr/ports/textproc/dsssl-docbook-modular && make install
```

2. (Optional) Set environment variables to access the jade toolset.

   Note

   This was not required for the FreeBSD machine at postgresql.org, so you may not have to do this.

```
export SMGL_ROOT=/usr/local/share/sgml

SGML_CATALOG_FILES=/usr/local/share/sgml/jade/catalog

SGML_CATALOG_FILES=/usr/local/share/sgml/html/catalog:
$SGML_CATALOG_FILES
SGML_CATALOG_FILES=/usr/local/share/sgml/iso8879/catalog:
$SGML_CATALOG_FILES
SGML_CATALOG_FILES=/usr/local/share/sgml/transpec/catalog:
$SGML_CATALOG_FILES
SGML_CATALOG_FILES=/usr/local/share/sgml/docbook/catalog:
$SGML_CATALOG_FILES
export SGML_CATALOG_FILES
```

   (this is sh/bash syntax; adjust accordingly for csh/tcsh).

3. Make needs some special arguments, or these need to be added to your Makefile.custom:

[7] ftp://ftp.cygnus.com/pub/home/rosalia/

```
HSTYLE=/usr/local/share/sgml/docbook/dsssl/modular/html/
PSTYLE=/usr/local/share/sgml/docbook/dsssl/modular/print/
```

Of course you'll need to use gmake rather than just plain 'make' to build.

DG2.7.3. Debian Installation

There is a full set of packages of the documentation tools available for Debian.

Installing Debian Packages

1. Install jade, docbook, and unzip:

   ```
   apt-get install jade
   apt-get install docbook
   apt-get install docbook-stylesheets
   ```

2. (Optional) Install the latest style sheets.

   a. (Optional) Verify that unzip is installed, or install the package:

      ```
      apt-get install unzip
      ```

   b. Grab the latest stylesheet zipballs from http://www.nwalsh.com/docbook/dsssl and unzip it somewhere (possibly /usr/share).

3. Edit src/Makefile.custom to add appropriate HSTYLE and PSTYLE definitions:

   ```
   HSTYLE= /usr/share/docbook/html
   PSTYLE= /usr/share/docbook/print
   ```

DG2.7.4. Manual Installation of Tools

This is a brief run-through of the process of obtaining and installing the software you'll need to edit DocBook source with Emacs and process it with Norman Walsh's DSSSL style sheets to create HTML and RTF.

The easiest way to obtain the SGML and DocBook tools may be to get sgmltools from sgmltools[8]. sgmltools requires the GNU version of m4. To confirm that you have the correct version of m4 available, try

```
gnum4 --version
```

If you install GNU m4, install it with the name gnum4 and sgmltools will find it. After the install, you will have sgmltools, jade, and Norm Walsh's DocBook style sheets. The instructions below are for installing these tools separately.

DG2.7.4.1. Prerequisites

What you need:

- A working installation of GCC 2.7.2

[8] http://www.sgmltools.org/

- A working installation of Emacs 19.19 or later
- An unzip program for Unix to unpack things

What you must fetch:

- James Clark's Jade[9] (version 1.1 in file `jade1_1.zip` was current at the time of writing)
- DocBook version 3.0[10]
- Norman Walsh's Modular Stylesheets[11] (version 1.19 was originally used to produce these documents)
- Lennart Staflin's PSGML[12] (version 1.0.1 in `psgml-1.0.1.tar.gz` was available at the time of writing)

Important URLs:

- The Jade web page[13]
- The DocBook web page[14]
- The Modular Stylesheets web page[15]
- The PSGML web page[16]
- Steve Pepper's Whirlwind Guide[17]
- Robin Cover's database of SGML software[18]

DG2.7.4.2. Installing Jade

Installing Jade

1. Read the installation instructions at the above listed URL.
2. Unzip the distribution kit in a suitable place. The command to do this will be something like

   ```
   unzip -aU jade1_1.zip
   ```

3. Jade is not built using GNU autoconf, so you'll need to edit a `Makefile` yourself. Since James Clark has been good enough to prepare his kit for it, it is a good idea to make a build directory (named for your machine architecture, perhaps) under the main directory of the Jade distribution, copy the file `Makefile` from the main directory into it, edit it there, and then run `make` there.

   However, the `Makefile` does need to be edited. There is a file called `Makefile.jade` in the main directory, which is intended to be used with `make -f Makefile.jade` when building Jade (as opposed to just SP, the SGML parser kit that Jade is built upon). We suggest that you don't do that, though, since there is more that you need to change than what is in `Makefile.jade`, so you'd have to edit one of them anyway.

[9] ftp://ftp.jclark.com/pub/jade/
[10] http://www.ora.com/davenport/docbook/current/docbk30.zip
[11] http://nwalsh.com/docbook/dsssl/
[12] ftp://ftp.lysator.liu.se/pub/sgml/
[13] http://www.jclark.com/jade/
[14] http://www.ora.com/davenport/
[15] http://nwalsh.com/docbook/dsssl/
[16] http://www.lysator.liu.se/projects/about_psgml.html
[17] http://www.infotek.no/sgmltool/guide.htm
[18] http://www.sil.org/sgml/publicSW.html

Go through the `Makefile`, reading James' instructions and editing as needed. There are various variables that need to be set. Here is a collected summary of the most important ones, with typical values:

```
prefix = /usr/local
XDEFINES = -DSGML_CATALOG_FILES_DEFAULT=\"/usr/local/share/
sgml/catalog\"
XLIBS = -lm
RANLIB = ranlib
srcdir = ..
XLIBDIRS = grove spgrove style
XPROGDIRS = jade
```

Note the specification of where to find the default catalog of SGML support files -- you may want to change that to something more suitable for your own installation. If your system doesn't need the above settings for the math library and the `ranlib` command, leave them as they are in the `Makefile`.

4. Type `make` to build Jade and the various SP tools.

5. Once the software is built, `make install` will do the obvious.

DG2.7.4.3. Installing the DocBook DTD Kit

Installing the DocBook DTD Kit

1. You'll want to place the files that make up the DocBook DTD kit in the directory you built Jade to expect them in, which, if you followed our suggestion above, is `/usr/local/share/sgml/`. In addition to the actual DocBook files, you'll need to have a `catalog` file in place, for the mapping of document type specifications and external entity references to actual files in that directory. You'll also want the ISO character set mappings, and probably one or more versions of HTML.

   One way to install the various DTD and support files and set up the `catalog` file, is to collect them all into the above mentioned directory, use a single file named `CATALOG` to describe them all, and then create the file `catalog` as a catalog pointer to the former, by giving it the single line of content:

   ```
   CATALOG /usr/local/share/sgml/CATALOG
   ```

2. The `CATALOG` file should then contain three types of lines. The first is the (optional) SGML declaration, thus:

   ```
   SGMLDECL docbook.dcl
   ```

   Next, the various references to DTD and entity files must be resolved. For the DocBook files, these lines look like this:

   ```
   PUBLIC "-//Davenport//DTD DocBook V3.0//EN" docbook.dtd
   PUBLIC "-//USA-DOD//DTD Table Model 951010//EN" cals-tbl.dtd
   PUBLIC "-//Davenport//ELEMENTS DocBook Information Pool V3.0//
   EN" dbpool.mod
   PUBLIC "-//Davenport//ELEMENTS DocBook Document Hierarchy
    V3.0//EN" dbhier.mod
   PUBLIC "-//Davenport//ENTITIES DocBook Additional General
    Entities V3.0//EN" dbgenent.mod
   ```

3. Of course, a file containing these comes with the DocBook kit. Note that the last item on each of these lines is a file name, given here without a path. You can put the files in subdirectories of your main SGML directory if you like, of course, and modify the reference in the `CATALOG` file. DocBook also references the ISO character set entities, so you need to fetch and install these (they are available from several sources, and are easily found by way of the URLs listed above), along with catalog entries for all of them, such as:

   ```
   PUBLIC "ISO 8879-1986//ENTITIES Added Latin 1//EN" ISO/ISOlat1
   ```

   Note how the file name here contains a directory name, showing that we've placed the ISO entity files in a subdirectory named `ISO`. Again, proper catalog entries should accompany the entity kit you fetch.

DG2.7.4.4. Installing Norman Walsh's DSSSL Style Sheets

Installing Norman Walsh's DSSSL Style Sheets

1. Read the installation instructions at the above listed URL.

2. To install Norman's style sheets, simply unzip the distribution kit in a suitable place. A good place to dot this would be `/usr/local/share`, which places the kit in a directory tree under `/usr/local/share/docbook`. The command will be something like

   ```
   unzip -aU db119.zip
   ```

3. One way to test the installation is to build the HTML and RTF forms of the *PostgreSQL User's Guide*.

   a. To build the HTML files, go to the SGML source directory, `doc/src/sgml`, and say

      ```
      jade -t sgml -d /usr/local/share/docbook/html/docbook.dsl -
      D ../graphics postgres.sgml
      ```

      `book1.htm` is the top level node of the output..

   b. To generate the RTF output, ready for importing into your favorite word processing system and printing, type:

      ```
      jade -t rtf -d /usr/local/share/docbook/print/docbook.dsl -
      D ../graphics postgres.sgml
      ```

DG2.7.4.5. Installing PSGML

Installing PSGML

1. Read the installation instructions at the above listed URL.

2. Unpack the distribution file, run configure, make and make install to put the byte-compiled files and info library in place.

3. Then add the following lines to your `/usr/local/share/emacs/site-lisp/site-start.el` file to make Emacs properly load PSGML when needed:

   ```
   (setq load-path
   ```

```
      (cons "/usr/local/share/emacs/site-lisp/psgml" load-
path))
(autoload 'sgml-mode "psgml" "Major mode to edit SGML files."
 t)
```

4. (Optional) If you want to use PSGML when editing HTML too, also add this:

```
(setq auto-mode-alist
      (cons '("\\.s?html?\\'" . sgml-mode) auto-mode-alist))
```

5. (Optional) There is one important thing to note with PSGML: its author assumed that your main SGML DTD directory would be `/usr/local/lib/sgml`. If, as in the examples in this chapter, you use `/usr/local/share/sgml`, you have to compensate for this.

   a. (Optional) You can set the `SGML_CATALOG_FILES` environment variable.

   b. (Optional) You can customize your PSGML installation (its manual tells you how).

   c. (Optional) You can even edit the source file `psgml.el` before compiling and installing PSGML, changing the hard-coded paths to match your own default.

DG2.7.4.6. Installing JadeTeX

If you want to, you can also install JadeTeX to use TeX as a formatting backend for Jade. Note that this is still quite unpolished software, and will generate printed output that is inferior to what you get from the RTF backend. Still, it works all right, especially for simpler documents that don't use tables, and as both JadeTeX and the style sheets are under continuous improvement, it will certainly get better over time.

To install and use JadeTeX, you will need a working installation of TeX and LaTeX2e, including the supported tools and graphics packages, Babel, AMS fonts and AMS-LaTeX, the PSNFSS extension and companion kit of "the 35 fonts", the dvips program for generating PostScript, the macro packages fancyhdr, hyperref, minitoc, url and ot2enc, and of course JadeTeX itself. All of these can be found on your friendly neighborhood CTAN site.

JadeTeX does not at the time of writing come with much of an installation guide, but there is a `makefile` which shows what is needed. It also includes a directory `cooked`, wherein you'll find some of the macro packages it needs, but not all, and not complete -- at least last we looked.

Before building the `jadetex.fmt` format file, you'll probably want to edit the `jadetex.ltx` file, to change the configuration of Babel to suit your locality. The line to change looks something like

```
\RequirePackage[german,french,english]{babel}[1997/01/23]
```

and you should obviously list only the languages you actually need, and have configured Babel for.

With JadeTeX working, you should be able to generate and format TeX output for the PostgreSQL manuals by giving the commands (as above, in the `doc/src/sgml` directory)

```
jade -t tex -d /usr/local/share/docbook/print/docbook.dsl -D ../
graphics postgres.sgml
jadetex postgres.tex
jadetex postgres.tex
dvips postgres.dvi
```

Of course, when you do this, TeX will stop during the second run, and tell you that its capacity has been exceeded. This is, as far as we can tell, because of the way JadeTeX generates cross referencing

information. TeX can, of course, be compiled with larger data structure sizes. The details of this will vary according to your installation.

DG2.8. Alternate Toolsets

sgml-tools v2.x supports jade and DocBook.

Bibliography

Selected references and readings for SQL and Postgres.

Some white papers and technical reports from the original Postgres development team are available at the University of California, Berkeley, Computer Science Department web site[1]

SQL Reference Books

Reference texts for SQL features.

The Practical SQL Handbook . Bowman et al, 1996 . Using Structured Query Language . 3. Judith Bowman, Sandra Emerson, and Marcy Darnovsky. 0-201-44787-8. 1996. Addison-Wesley. Copyright © 1996 Addison-Wesley Longman, Inc..

A Guide to the SQL Standard . Date and Darwen, 1997 . A user's guide to the standard database language SQL . 4. C. J. Date and Hugh Darwen. 0-201-96426-0. 1997. Addison-Wesley. Copyright © 1997 Addison-Wesley Longman, Inc..

An Introduction to Database Systems . Date, 1994 . 6. C. J. Date. 1. 1994. Addison-Wesley. Copyright © 1994 Addison-Wesley Longman, Inc..

Understanding the New SQL . Melton and Simon, 1993 . A complete guide. Jim Melton and Alan R. Simon. 1-55860-245-3. 1993. Morgan Kaufmann. Copyright © 1993 Morgan Kaufmann Publishers, Inc..

Principles of Database and Knowledge . Base Systems . Ullman, 1988 . Jeffrey D. Ullman. 1. Computer Science Press . 1988 .

PostgreSQL-Specific Documentation

This section is for related documentation.

The PostgreSQL. Administrator's Guide . The Administrator's Guide . Thomas Lockhart. 2000-05-01. The PostgreSQL Global Development Group.

The PostgreSQL. Developer's Guide . The Developer's Guide . Thomas Lockhart. 2000-05-01. The PostgreSQL Global Development Group.

The PostgreSQL. Programmer's Guide . The Programmer's Guide . Thomas Lockhart. 2000-05-01. The PostgreSQL Global Development Group.

The PostgreSQL. Tutorial Introduction . The Tutorial . Thomas Lockhart. 2000-05-01. The PostgreSQL Global Development Group.

The PostgreSQL. User's Guide . The User's Guide . Thomas Lockhart. 2000-05-01. The PostgreSQL Global Development Group.

Enhancement of the ANSI SQL Implementation of PostgreSQL . Simkovics, 1998 . Stefan Simkovics. with support by . O.Univ.Prof.Dr.. Georg. Gottlob. Univ.Ass. Mag.. Katrin. Seyr. . November 29, 1998. Department of Information Systems, Vienna University of Technology .

The Postgres95. User Manual . Yu and Chen, 1995 . A. Yu and J. Chen. . Sept. 5, 1995. University of California, Berkeley CA.

Proceedings and Articles

This section is for articles and newsletters.

[1] http://s2k-ftp.CS.Berkeley.EDU:8000/postgres/papers/

Partial indexing in POSTGRES: research project . Olson, 1993 . Nels Olson. 1993. UCB Engin T7.49.1993 O676. University of California, Berkeley CA.

A Unified Framework for Version Modeling Using Production Rules in a Database System . Ong and Goh, 1990 . L. Ong and J. Goh. April, 1990. ERL Technical Memorandum M90/33. University of California, Berkeley CA.

The Postgres. Data Model . Rowe and Stonebraker, 1987 . L. Rowe and M. Stonebraker. Sept. 1987. VLDB Conference, Brighton, England. 1987. .

Generalized partial indexes [2] . Seshardri, 1995 . P. Seshadri and A. Swami. March 1995. Eleventh International Conference on Data Engineering. . 1995. Cat. No.95CH35724. IEEE Computer Society Press.

The Design of Postgres. . Stonebraker and Rowe, 1986 . M. Stonebraker and L. Rowe. May 1986. Conference on Management of Data, Washington DC. ACM-SIGMOD. 1986. .

The Design of the Postgres. Rules System. Stonebraker, Hanson, Hong, 1987 . M. Stonebraker, E. Hanson, and C. H. Hong. Feb. 1987. Conference on Data Engineering, Los Angeles, CA. IEEE. 1987. .

The Postgres. Storage System . Stonebraker, 1987 . M. Stonebraker. Sept. 1987. VLDB Conference, Brighton, England. 1987. .

A Commentary on the Postgres. Rules System . Stonebraker et al, 1989. M. Stonebraker, M. Hearst, and S. Potamianos. Sept. 1989. Record 18(3). SIGMOD. 1989. .

The case for partial indexes (DBMS) [3] . Stonebraker, M, 1989b. M. Stonebraker. Dec. 1989. Record 18(no.4):4-11. SIGMOD. 1989. .

The Implementation of Postgres. . Stonebraker, Rowe, Hirohama, 1990 . M. Stonebraker, L. A. Rowe, and M. Hirohama. March 1990. Transactions on Knowledge and Data Engineering 2(1). IEEE. .

On Rules, Procedures, Caching and Views in Database Systems . Stonebraker et al, ACM, 1990 . M. Stonebraker and et al. June 1990. Conference on Management of Data. ACM-SIGMOD. .

[2] http://simon.cs.cornell.edu/home/praveen/papers/partindex.de95.ps.Z
[3] http://s2k-ftp.CS.Berkeley.EDU:8000/postgres/papers/ERL-M89-17.pdf