Table of Contents

<table>
<thead>
<tr>
<th>Section 1 — Introduction</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Section 2 — UNIX Commands (UNIX)</td>
<td>2</td>
</tr>
<tr>
<td>General Information</td>
<td>2</td>
</tr>
<tr>
<td>Security</td>
<td>3</td>
</tr>
<tr>
<td>Createdb</td>
<td>7</td>
</tr>
<tr>
<td>Createuser</td>
<td>9</td>
</tr>
<tr>
<td>Destroydb</td>
<td>11</td>
</tr>
<tr>
<td>Destroyuser</td>
<td>13</td>
</tr>
<tr>
<td>Icon</td>
<td>15</td>
</tr>
<tr>
<td>Initdb</td>
<td>17</td>
</tr>
<tr>
<td>Ipcclean</td>
<td>18</td>
</tr>
<tr>
<td>The POSTGRES Terminal Monitor</td>
<td>19</td>
</tr>
<tr>
<td>Newdb</td>
<td>22</td>
</tr>
<tr>
<td>Pagedoc</td>
<td>23</td>
</tr>
<tr>
<td>Pcat</td>
<td>24</td>
</tr>
<tr>
<td>Pcd</td>
<td>25</td>
</tr>
<tr>
<td>Pls</td>
<td>26</td>
</tr>
<tr>
<td>Pmkdir</td>
<td>27</td>
</tr>
<tr>
<td>Pmv</td>
<td>28</td>
</tr>
<tr>
<td>The POSTGRES Backend Server</td>
<td>29</td>
</tr>
<tr>
<td>The POSTGRES Postmaster</td>
<td>31</td>
</tr>
<tr>
<td>Ppwd</td>
<td>34</td>
</tr>
<tr>
<td>Pnm</td>
<td>35</td>
</tr>
<tr>
<td>Pmdir</td>
<td>36</td>
</tr>
<tr>
<td>Reindexdb</td>
<td>37</td>
</tr>
<tr>
<td>Shmemdoc</td>
<td>39</td>
</tr>
<tr>
<td>Section 3 — What comes with POSTGRES (BUILT-INS)</td>
<td>41</td>
</tr>
<tr>
<td>Built-in and System Types</td>
<td>41</td>
</tr>
<tr>
<td>List of built-in types</td>
<td>41</td>
</tr>
<tr>
<td>Syntax of date and time types</td>
<td>42</td>
</tr>
<tr>
<td>Built-in operators and functions</td>
<td>42</td>
</tr>
<tr>
<td>Binary operators</td>
<td>43</td>
</tr>
<tr>
<td>Unary operators</td>
<td>48</td>
</tr>
<tr>
<td>Built-in aggregate functions</td>
<td>48</td>
</tr>
<tr>
<td>Section 4 — POSTQUEL Commands (COMMANDS)</td>
<td>50</td>
</tr>
<tr>
<td>General Information</td>
<td>50</td>
</tr>
<tr>
<td>Constants</td>
<td>50</td>
</tr>
<tr>
<td>Fields and Attributes</td>
<td>52</td>
</tr>
<tr>
<td>Operators</td>
<td>53</td>
</tr>
<tr>
<td>Expressions</td>
<td>53</td>
</tr>
<tr>
<td>Commands</td>
<td>58</td>
</tr>
<tr>
<td>Abort</td>
<td>58</td>
</tr>
<tr>
<td>Addattr</td>
<td>59</td>
</tr>
<tr>
<td>Append</td>
<td>60</td>
</tr>
<tr>
<td>Attachas</td>
<td>63</td>
</tr>
<tr>
<td>Begin</td>
<td>64</td>
</tr>
<tr>
<td>Change ACL</td>
<td>65</td>
</tr>
<tr>
<td>Close</td>
<td>67</td>
</tr>
<tr>
<td>Cluster</td>
<td>68</td>
</tr>
<tr>
<td>Copy</td>
<td>69</td>
</tr>
<tr>
<td>Create</td>
<td>71</td>
</tr>
<tr>
<td>Createdb</td>
<td>73</td>
</tr>
<tr>
<td>Create Version</td>
<td>74</td>
</tr>
<tr>
<td>Define Aggregate</td>
<td>75</td>
</tr>
<tr>
<td>Define Function</td>
<td>77</td>
</tr>
<tr>
<td>Define Index</td>
<td>86</td>
</tr>
<tr>
<td>Define Operator</td>
<td>90</td>
</tr>
<tr>
<td>Define Rule</td>
<td>94</td>
</tr>
<tr>
<td>Define Type</td>
<td>97</td>
</tr>
</tbody>
</table>

03/11/94
SECTION 1 — INTRODUCTION

OVERVIEW
This document is the reference manual for the POSTGRES database management system under development at the University of California at Berkeley. The POSTGRES project, led by Professor Michael Stonebraker, has been sponsored by the Defense Advanced Research Projects Agency (DARPA), the Army Research Office (ARO), the National Science Foundation (NSF), and ESL, Inc.

POSTGRES is distributed in source code format and is the property of the Regents of the University of California. However, the University will grant unlimited commercialization rights for any derived work on the condition that it obtain an educational license to the derived work. For further information, consult the Berkeley Campus Software Office, 295 Evans Hall, University of California, Berkeley, CA 94720. Note that there is no organization who can help you with any bugs you may encounter or with any other problems. In other words, this is unsupported software.

POSTGRES DISTRIBUTION
This reference describes Version 4.2 of POSTGRES. The POSTGRES software is about 200,000 lines of C code. Information on obtaining the source code is available from:

Claire Mosher
Computer Science Division
521 Evans Hall
University of California
Berkeley, CA 94720
(510) 642-4662

Version 4.2 has been tuned modestly. Hence, on the Wisconsin benchmark, one should expect performance about twice that of the public domain, University of California version of INGRES, a relational prototype from the late 1970s.

As distributed, POSTGRES runs on Digital Equipment Corporation computers based on MIPS R2000 and R3000 processors (under Ultrix 4.2A and 4.3A), Digital Equipment Corporation computers based on Alpha AXP (DECchip 21064) processors (under OSF/1 1.3), Sun Microsystems computers based on SPARC processors (under SunOS 4.1.3), Hewlett-Packard Model 9000 Series 700 and 800 computers based on PA-RISC processors (under HP-UX 9.00 and 9.01), and International Business Machines RS/6000 computers based on POWER processors (under AIX 3.2.5). POSTGRES users have ported previous releases of the system to many other architectures and operating systems, including NeXTSTEP, Solaris 2.2, IRIX, Intel System V Release 4, Linux and NetBSD.

POSTGRES DOCUMENTATION
This reference manual describes the functionality of Version 4.2 and contains notations where appropriate to indicate which features are not implemented in Version 4.2. Application developers should note that this reference contains only the specification for the low-level call-oriented application program interface, LIBPQ. A companion volume, the POSTGRES User Manual, contains tutorial examples of the ways in which the system can be extended.

The remainder of this reference manual is structured as follows. In Section 2 (UNIX), we discuss the POSTGRES capabilities that are available directly from the operating system. Section 3 (BUILT-INS) describes POSTGRES internal data types, functions, and operators. Section 4 (COMMANDS) then describes POSTQUEL, the language by which a user interacts with a POSTGRES database. Then, Section 5 (LIBPQ) describes a library of low level routines through which a user can formulate POSTQUEL queries.
from a C program and get appropriate return information back to his program. Next, Section 6 (FAST
PATH) continues with a description of a method by which applications may execute functions in POST-
GRES with very high performance. Section 7 (LARGE OBJECTS) describes the internal POSTGRES in-
terface for accessing large objects. Section 8 (SYSTEM CATALOGS) gives a brief explanation of the tables
used internally by POSTGRES. The reference concludes with Section 9 (FILES), a collection of file format
descriptions for files used by POSTGRES.

ACKNOWLEDGEMENTS

POSTGRES has been constructed by a team of undergraduate, graduate, and staff programmers. The con-
tributors (in alphabetical order) consisted of: Jeff Anton, Paul Aoki, James Bell, Jennifer Caetta, Philip
Chang, Jolly Chen, Ron Choi, Matt Dillon, Zelaine Fong, Adam Glass, Jeffrey Goh, Steven Grady, Serge
Granik, Marti Hearst, Joey Hellerstein, Michael Hirohama, Chin-heng Hong, Wei Hong, Anant Jhingran,
Greg Kemaitz, Marcel Kornacker, Case Larsen, Boris Livshitz, Jeff Meredith, Ginger Ogle, Michael
Olson, Nels Olson, Lay-Peng Ong, Carol Paxson, Avi Pfeffer, Spyros Potamianos, Sumita Sarawagi, David
Muir Sharoff, Mark Sullivan, Cimarron Taylor, Marc Teitelbaum, Yongdong Wang, Kristin Wright and
Andrew Yu. The HP-UX port is courtesy of Richard Turnbull (University of Liverpool) and Sebastian Ferna-
andez (University of California at Berkeley). The initial AIX port was performed by Rafael Morales
Gamboa (ITESM Campus Morelos, Cuernavaca). Carl Staelin of H-P Laboratories and Steve Miley of
UCSB/CRSEO provided the computing resources that enabled us to integrate these ports into the POST-
GRES distribution.

Marc Teitelbaum served as chief programmer for Version 4.2 and was responsible for overall coordinat-
ion of the project.

This reference was collectively written by the above implementation team, assisted by Bob Devine, Jim
Frew, Chandra Ghosh, Claire Mosher and Michael Stonebraker.

LEGAL NOTICES

POSTGRES is copyright © 1989, 1994 by the Regents of the University of California. Permission to use,
copy, modify, and distribute this software and its documentation for educational, research, and non-profit
purposes and without fee is hereby granted, provided that the above copyright notice appear in all copies
and that both that copyright notice and this permission notice appear in supporting documentation, and that
the name of the University of California not be used in advertising or publicity pertaining to distribution of
the software without specific, written prior permission. Permission to incorporate this software into com-
mercial products can be obtained from the Campus Software Office, 295 Evans Hall, University of Califor-
nia, Berkeley, Ca., 94720. The University of California makes no representations about the suitability of
this software for any purpose. It is provided "as is" without express or implied warranty.

UNIX is a trademark of Unix Systems Laboratories. Sun4, SPARC, SunOS and Solaris are trademarks of
Sun Microsystems, Inc. DEC, DECstation, Alpha AXP and ULTRIX are trademarks of Digital Equipment
Corp. PA-RISC and HP-UX are trademarks of Hewlett-Packard Co. RS/6000, POWER and AIX are
trademarks of International Business Machines Corp. OSF/1 is a trademark of the Open Systems Fou-
dation. NeXTSTEP is a trademark of NeXT Computer, Inc. MIPS and IRIX are trademarks of Silicon
Graphics, Inc.
SECTION 2 — UNIX COMMANDS (UNIX)

OVERVIEW

This section contains information on the interaction between POSTGRES and the operating system. In particular, the pages of this section describe the POSTGRES support programs that are executable as UNIX commands.

TERMINOLOGY

In the following documentation, the term site may be interpreted as the host machine on which POSTGRES is installed. However, 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 super-user is the user named “postgres” (usually) who owns the POSTGRES binaries and database files. As the database super-user, all protection mechanisms may be bypassed and any data accessed arbitrarily. In addition, the POSTGRES super-user is allowed to execute some support programs which are generally not available to all users. Note that the POSTGRES super-user is not the same as the UNIX super-user, root, and should have a non-zero userid.

The database base administrator or DBA, is the person who is responsible for installing POSTGRES to enforce a security policy for a site. The DBA will add new users by the method described below, change the status of user-defined functions from untrusted to trusted as explained in define function commands, and maintain a set of template databases for use by createdb commands.

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 track of any system errors and communication between the backend processes. The postmaster can take several command-line arguments to tune its behavior, but supplying arguments is necessary only if you intend to run multiple sites or a non-default site. See postmaster(1) for details.

The POSTGRES backend (.../bin/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, so this is not recommended in a multiuser site.

NOTATION

"..." at the front of a file name is used to represent the path to the POSTGRES super-user's home directory. Anything in brackets ("[" and "]") is optional. Anything in braces ("{" and "}") can be repeated 0 or more times. Parentheses ("(" and ")") are used to group boolean expressions. "I" is the boolean operator OR.

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.

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 (real) user-id before performing a setuid(3) to the user-id of user "postgres". The real 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 completely unrestricted. That is, any user may connect to this port, spoof the postmaster, pose as an authorized user and issue any commands desired. However, since this port is configurable and not normally advertised in any public files (e.g., /etc/services), some administrators may be satisfied with security-by-obscurity.

If greater security is desired, POSTGRES and its clients may be modified to use a network authentication system. For example, the postmaster, monitor and libpq have already been configured to use either Version 4 or Version 5 of the Kerberos authentication system from the Massachusetts Institute of Technology. For more information on using Kerberos with POSTGRES, see the appendix below.

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 control lists

The use of access control lists to limit reading, writing and setting of rules on classes is covered in change acl(commands).

Class removal and schema modification

Commands that destroy or modify the structure of an existing class, such as addattr, destroy, rename, and remove index, only operate for the owner of the class. As mentioned above, these operations are never permitted on system catalogs.

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 POSTQUEL fields) and rules. Audit trails and alerters on "pg_class", "pg_user" and "pg_group" are also recommended.

Functions

Functions written in any language except POSTQUEL with "pg_proc.proistrusted" set 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 trusted functions.

Non-POSTQUEL functions that do not have "pg_proc.proistrusted" set run in a separate untrusted-function process spawned by the backend server. If correctly installed, this process runs with real and effective user-id set to "nobody" (or some other user with strictly limited permissions). It should be noted, however, that
the primary purpose of untrusted functions is actually to simplify debugging of user-defined functions
(since buggy functions will only crash or corrupt the untrusted-function process instead of the server pro-
cess). The current RPC protocol only works in one direction, so any function that make function-manager
calls (e.g., access method calls) or performs other database file operations must be trusted.

Since untrusted functions are a new feature in Version 4.2, the define function command still defaults to
making new functions trusted. This is a massive security hole that will be removed in a later release, once
the (mis)features and interface of untrusted functions have stabilized. (An additional access control will be
added for defining functions, analogous to the access control on defining rules.)

Like other functions that perform database file operations, POSTQUEL functions must run in the same
address space as the backend server. The "pg_proc.proistrusted" field has no effect for POSTQUEL func-
tions, which always run with the permissions of the user who invoked the backend server. (Otherwise,
users could circumvent access controls — the "nobody" user may well be allowed greater access to a par-
ticular object than a given user.)

Rules
Like POSTQUEL functions, rules always run with the identity and permissions of the user who invoked the
backend server.

SEE ALSO
postmaster(uni), addattr(commands), append(commands), change acl(commands), copy(commands),
define rule(commands), delete(commands), destroy(commands), remove index(commands), remove
rule(commands), rename(commands), replace(commands), retrieve(commands), kerberos(1), kinit(1), ker-
beros(3)

CAVEATS
There are no plans to explicitly support encrypted data inside of POSTGRES (though there is nothing to pre-
vent 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.

User system identifiers are currently UNIX user-ids.

APPENDIX: USING KERBEROS

Availability
The Kerberos authentication system is not distributed with POSTGRES, nor is it available from the University
of California at Berkeley. Versions of Kerberos are typically available as optional software from oper-
ating system vendors. In addition, a source code distribution may be obtained through MIT Project Athena
by anonymous FTP from ATHENA-DIST.MIT.EDU (18.71.0.38). (You may wish to obtain the MIT ver-
sion even if your vendor provides a version, since some vendor ports have been deliberately crippled or ren-
dered 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. govern-
ment export regulations.

Any additional inquiries should be directed to your vendor or MIT Project Athena ("info-
kerberos@ATHENA,MIT.EDU"). Note that FAQLs (Frequently-Asked Questions Lists) are periodically
posted to the Kerberos mailing list, "kerberos@ATHENA,MIT.EDU" (send mail to "kerberos-
request@ATHENA,MIT.EDU" to subscribe), and USENET news group, "comp.protocols.kerberos".

Installation
Installation of Kerberos itself is covered in detail in the Kerberos Installation Notes. Make sure that the
server key file (the srrvtab or keytab) is somehow readable by user "postgres".

03/12/94
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.

Operation

After initial installation, POSTGRES should operate in all ways as a normal Kerberos service. For details on the use of authentication, see the manual pages for postmaster(unix) and monitor(unix).

In the Kerberos Version 5 hooks, the following assumptions are made about user and service naming: (1) user principal names (anames) are assumed to contain the actual UNIX/POSTGRES user name in the first component; (2) the POSTGRES service is assumed to have two components, the service name and a hostname, canonicalized as in Version 4 (i.e., all domain suffixes removed).

user example: frew@S2K.ORG
user example: aoki/HOST=miyu.S2K.Berkeley.EDU@S2K.ORG
host example: postgres_dbms/ucbvax@S2K.ORG

Support for Version 4 will disappear sometime after the production release of Version 5 by MIT.

03/12/94
NAME
createdb — create a database

SYNOPSIS
createdb [-a system] [-h host] [-p port] [dbname]

DESCRIPTION
createdb creates a new database. The person who executes this command becomes the database administrator, or DBA, for this database and is the only person, other than the POSTGRES super-user, who can destroy it.

createdb is a shell script that invokes the POSTGRES monitor. Hence, a postmaster process must be running on the database server host before createdb is executed. In addition, the PGOPTION and PGREALM environment variables will be passed on to monitor and processed as described in monitor(unix).

The optional argument dbname specifies the name of the database to be created. The name must be unique among all POSTGRES databases. Dname defaults to the value of the USER environment variable.

createdb understands the following command-line options:
-a system
  Specifies an authentication system system (see introduction(unix)) to use in connecting to the postmaster process. The default is site-specific.
-h host
  Specifies the hostname of the machine on which the postmaster is running. Defaults to the name of the local host, or the value of the PGHOST environment variable (if set).
-p port
  Specifies the Internet TCP port on which the postmaster is listening for connections. Defaults to 4321, or the value of the PGPORT environment variable (if set).

EXAMPLES
# create the demo database
createdb demo

# create the demo database using the postmaster on host eden,
# port 1234, using the Kerberos authentication system.
createdb -a kerberos -p 1234 -h eden demo

FILES
  SPGDATA/base/dbname
  The location of the files corresponding to the database dbname.

SEE ALSO
createdb(commands), destroydb(unix), initdb(unix), monitor(unix), postmaster(unix).

DIAGNOSTICS
Error: Failed to connect to backend (host=x.x.x, port=x.x.x)
createdb 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.
user "username" is not in "pg_user"
    You do not have a valid entry in the relation "pg_user" and cannot do anything with POSTGRES at all;
    contact your POSTGRES site administrator.

user "username" is not allowed to create/destroy databases
    You do not have permission to create new databases; contact your POSTGRES site administrator.

dbname already exists
    The database already exists.

database creation failed on dbname
    An internal error occurred in monitor or the backend server. Ensure that your POSTGRES site admin-
    istrator has properly installed POSTGRES and initialized the site with initdb.
NAME
createuser — create a POSTGRES user

SYNOPSIS
createuser [-a system] [-h host] [-p port] [username]

DESCRIPTION
createuser creates a new POSTGRES user. Only users with "usesuper" set in the "pg_user" class can create new POSTGRES users. As shipped, the user "postgres" can create users.

createuser is a shell script that invokes monitor. Hence, a postmaster process must be running on the database server host before createuser is executed. In addition, the PGOPTION and PGREALM environment variables will be passed on to monitor and processed as described in monitor(unix).

The optional argument username specifies the name of the POSTGRES user to be created. (The invoker will be prompted for a name if none is specified on the command line.) This name must be unique among all POSTGRES users.

createuser understands the following command-line options:
-a system
  Specifies an authentication system system (see introduction(unix)) to use in connecting to the postmaster process. The default is site-specific.
-h host
  Specifies the hostname of the machine on which the postmaster is running. Defaults to the name of the local host, or the value of the PGHOST environment variable (if set).
-p port
  Specifies the Internet TCP port on which the postmaster is listening for connections. Defaults to 4321, or the value of the PGPORT environment variable (if set).

INTERACTIVE QUESTIONS
Once invoked with the above options, createuser will ask a series of questions. The new user's login name (if not given on the command line) and user-id must be specified. (Note that the POSTGRES user-id must be the same as the user's UNIX user-id.) In addition, you must describe the security capabilities of the new user. Specifically, you will be asked whether the new user should be able to act as POSTGRES super-user, create new databases and update the system catalogs manually.

SEE ALSO
destroyuser(unix), monitor(unix), postmaster(unix).

DIAGNOSTICS
Error: Failed to connect to backend (host=xxx, port=xxx)
createuser 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.

user "username" is not in "pg_user"
You do not have a valid entry in the relation "pg_user" and cannot do anything with POSTGRES at all; contact your POSTGRES site administrator.

username cannot create users.
You do not have permission to create new users; contact your POSTGRES site administrator.

user "username" already exists
The user to be added already has an entry in the "pg_user" class.
database access failed

An internal error occurred in monitor or the backend server. Ensure that your POSTGRES site administrator has properly installed POSTGRES and initialized the site with initdb.

BUGS

POSTGRES user-id's and user names should not have anything to do with the constraints of UNIX.
DESTROYDB(UNIX)

NAME
destroydb — destroy an existing database

SYNOPSIS
destroydb [-a system] [-h host] [-p port] [dbname]

DESCRIPTION
Destroydb destroys an existing database. To execute this command, the user must be the database administrator, or DBA, for this database. The program runs silently; no confirmation message will be displayed. After the database is destroyed, a UNIX shell prompt will reappear.

Destroydb is a shell script that invokes monitor. Hence, a postmaster process must be running on the database server host before destroydb is executed. In addition, the PGOPTION and PGREALM environment variables will be passed on to monitor and processed as described in monitor(unix).

The optional argument dbname specifies the name of the database to be destroyed. All references to the database are removed, including the directory containing this database and its associated files. dbname defaults to the value of the USER environment variable.

Destroydb understands the following command-line options:

-a system
Specifies an authentication system system (see introduction(unix)) to use in connecting to the postmaster process. The default is site-specific.

-h host
Specifies the hostname of the machine on which the postmaster is running. Defaults to the name of the local host, or the value of the PGHOST environment variable (if set).

-p port
Specifies the Internet TCP port on which the postmaster is listening for connections. Defaults to 4321, or the value of the PGPORT environment variable (if set).

EXAMPLES
# destroy the demo database
destroydb demo

# destroy the demo database using the postmaster on host eden,
# port 1234, using the Kerberos authentication system.
destroydb -a kerberos -p 1234 -h eden demo

FILES
$PGDATA/base/dbname
The location of the files corresponding to the database dbname.

SEE ALSO
destroydb(commands), createdb(unix), initdb(unix), monitor(unix), postmaster(unix).

DIAGNOSTICS
Error: Failed to connect to backend (host=xxx, port=xxx)
Destroydb 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.
user "username" is not in "pg_user"
   You do not have a valid entry in the relation "pg_user" and cannot do anything with POSTGRES at all;
   contact your POSTGRES site administrator.

user "username" is not allowed to create/destroy databases
   You do not have permission to destroy databases; contact your POSTGRES site administrator.

database "dbname" does not exist
   The database to be removed does not have an entry in the "pg_database" class.

database "dbname" is not owned by you
   You are not DBA for the specified database.

database destroy failed on dbname
   An internal error occurred in monitor or the backend server. Contact your POSTGRES site administrator to ensure that ensure that the files and database entries associated with the database are completely removed.
NAME

destroyuser — destroy a POSTGRES user and associated databases

SYNOPSIS

destroyuser [-a system] [-h host] [-p port] [username]

DESCRIPTION

Destroyuser destroys an existing POSTGRES user and the databases for which that user is database administrator. Only users with "usesuper" set in the "pg_user" class can destroy new POSTGRES users. As shipped, the user "postgres" can destroy users.

Destroyuser is a shell script that invokes monitor. Hence, a postmaster process must be running on the database server host before destroyuser is executed. In addition, the PGOPTION and PGREALM environment variables will be passed on to monitor and processed as described in monitor(1x).

The optional argument username specifies the name of the POSTGRES user to be destroyed. (The invoker will be prompted for a name if none is specified on the command line.)

Destroyuser understands the following command-line options:

-a system
   Specifies an authentication system system (see introduction(1x)) to use in connecting to the postmaster process. The default is site-specific.

-h host
   Specifies the hostname of the machine on which the postmaster is running. Defaults to the name of the local host, or the value of the PGHOST environment variable (if set).

-p port
   Specifies the Internet TCP port on which the postmaster is listening for connections. Defaults to 4321, or the value of the PGPORT environment variable (if set).

INTERACTIVE QUESTIONS

Once invoked with the above options, destroyuser will warn you about the databases that will be destroyed in the process and permit you to abort the removal of the user if desired.

SEE ALSO

createuser(1x), monitor(1x), postmaster(1x).

DIAGNOSTICS

Error: Failed to connect to backend (host=xxx, port=xxx)
   Destroyuser 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.

user "username" is not in "pg_user"
   You do not have a valid entry in the relation "pg_user" and cannot do anything with POSTGRES at all; contact your POSTGRES site administrator.

username cannot delete users.
   You do not have permission to delete users: contact your POSTGRES site administrator.

user "username" does not exist
   The user to be removed does not have an entry in the "pg_user" class.

database access failed

01/23/93
destroydb on dbname failed - exiting

delete of user username was UNSUCCESSFUL

An internal error occurred in monitor or the backend server. Contact your POSTORES site administrator to ensure that the files and database entries associated with the user and his/her associated databases are completely removed.
NAME
icopy - copy files between Unix and Inversion file systems

SYNOPSIS
icopy direction -d dbname -s smgr

DESCRIPTION
Icopy copies files between the Inversion file system and the UNIX file system. This program is a libpq client program, and the Inversion file system is a transaction-protected file system used by the Sequoia 2000 research project at UC Berkeley. Inversion provides the same file system services provided by the UNIX fast file system, but does not support an NFS interface at present. In order to make it easier to use Inversion, a suite of utility programs, including icopy, has been written to manage files.

The user specifies the host and port on which POSTGRES is running, and the database and storage manager to use for file storage. The direction of the copy specifies whether files should be copied from UNIX to Inversion (in), or from Inversion to UNIX (out). The user also supplies two file names for the source and destination of the copy.

ARGUMENTS
The first five arguments listed here are required.

direction The direction of the copy. If the direction is in, then the file is copied from UNIX into Inversion. If the direction is out, then the file is copied out of Inversion to UNIX. The direction argument affects the interpretation of the source and destination file names, and may make some other flags (such as -s) optional (see below). This argument must immediately follow the program name.

-d dbname
The database to use for file storage. The user should have permission to create objects in dbname. The database name must be supplied; there is no default.

-s smgr
Use smgr as the storage manager for the file. Storage managers in POSTGRES manage physical devices, so this flag gives the user a way of controlling the device on which his file should be stored. If the direction of the copy is in, then the storage manager must be specified. If the direction of the copy is out, then the storage manager flag is optional, and is ignored if it is supplied.

The list of available storage managers may be obtained by typing

icopy

with no options; the resulting usage message includes a list of storage managers supported.

srcfile
The file from which to copy. If direction is in, then this is the name of a file or directory on the UNIX file system. If direction is out, then this is the name of a file or directory on the Inversion file system.

If srcfile is a directory and the -R flag is supplied, then the tree rooted at srcfile is copied. It is an error to specify a directory to copy without supplying the -R flag.

destfile
The file to which to copy. If direction is in, then this is the name of an Inversion file or directory. If direction is out, then destfile is the name of a UNIX file or directory.

If destfile already exists and is a directory, then srcfile will be created in the directory destfile.

The following arguments are optional.
ICOPY(UNIX)

-h host  Specifies the hostname of the machine on which the postmaster is running. Defaults to the name of the local host, or the value of the PGHOST environment variable (if set).

-p port  Specifies the Internet TCP port on which the postmaster is listening for connections. Defaults to 4321, or the value of the PGPORT environment variable (if set).

-R      Copy a directory tree recursively. Rather than copying a single file, the tree rooted at srcfile is copied to a tree rooted at destfile.

-a      Copy all files, including those beginning with a dot. This flag is useful only in conjunction with -R. Normally, recursive copies of a directory tree will not copy files or directories whose names begin with a dot.

-v      Turn verbose mode on. Icopy will report its progress as it moves files to or from Inversion.

EXAMPLES

The command

    icopy in -h myhost -p 4321 -d mydb -s d /vmunix /inv_vmunix

Copies the UNIX file "/vmunix" to the Inversion file "/inv_vmunix". The Inversion file is stored in the database "mydb" by the POSTGRES backend running on machine "myhost" and listening on port number 4321.

The command

    icopy out -h myhost -p 4321 -d mydb /inv_vmunix /vmunix.dup

Copies it back out again, putting the copy in the UNIX file "/vmunix.dup".

BUGS

The POSTGRES file system code should support operations via NFS, so this program actually has no right to exist.

See introduction (large objects) for filename and path limitations imposed by the Inversion file system.
NAME

initdb — initialize the database templates and primary directories

SYNOPSIS

initdb [-v] [-d] [-n]

DESCRIPTION

`initdb` sets up the initial template databases and is normally executed as part of the installation process. The template database is created under the directory specified by the environment variable `PGDATA`, or to a default specified at compile-time. The template database is then vacuumed.

`initdb` is a shell script that invokes the backend server directly. Hence, it must be executed by the PostgreSQL super-user.

`initdb` understands the following command-line options:

- `-v`  Produce verbose output, printing messages stating where the directories are being created, etc.
- `-d`  Print debugging output from the backend server. This option generates a tremendous amount of information. This option also turns off the final vacuuming step.
- `-n`  Run in “noclean” mode. By default, `initdb` cleans up (recursively unlinks) the data directory if any error occurs, which also removes any core files left by the backend server. This option inhibits any tidying-up.

FILES

`$PGDATA/base`

The location of global (shared) classes.

`$PGDATA/base/template1`

The location of the template database.

`$PGDATA/files/{global1,local1,template1}.bki`

Command files used to generate the global and template databases, generated and installed by the initial compilation process.

SEE ALSO

createdb(unix), vacuum(commands), bki(files), template(files).
NAME
ipcclean — clean up shared memory and semaphores from aborted backends

SYNOPSIS
ipcclean

DESCRIPTION
Ipclean 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.

BUGS
If this command is executed while a 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 effec-
tive and portable solution. Suggestions are welcome.
NAME

monitor — run the interactive terminal monitor

SYNOPSIS

monitor [-N] [-Q] [-T] [-a system] [-c query] [-d path]
       [-b hostname] [-p port] [-q] [-t tty_device] [dbname]

DESCRIPTION

The interactive terminal monitor is a simple frontend to POSTGRES. It enables one to formulate, edit and
review queries before issuing them to POSTGRES. If changes must be made, a UNIX editor may be called
to edit the query buffer managed by the terminal monitor. The editor used is determined by the value of
the EDITOR environment variable. If EDITOR is not set, then vi is used by default.

Monitor is a frontend application, like any other. Hence, a postmaster process must be running on the
database server host before monitor is executed. In addition, the correct postmaster port number must be
specified as described below.

The optional argument dbname specifies the name of the database to be accessed. This database must
already have been created using createdb. Dname defaults to the value of the USER environment variable.

Monitor understands the following command-line options:

-N   Specifies that query results will be dumped to the screen without any attempt at formatting. This is
     useful in conjunction with the -c option in shell scripts.

-Q   Produces extremely unverbose output. This is useful in conjunction with the -c option in shell
     scripts.

-T   Specifies that attribute names will not be printed. This is useful in conjunction with the -c option in
     shell scripts.

-a system
     Specifies an authentication system system (see introduction(UNIX)) to use in connecting to the post-
     master process. The default is site-specific.

-c query
     Specifies that monitor is to execute one query string, query, and then exit. This is useful for shell
     scripts, typically in conjunction with the -N and -T options. Examples of shell scripts in the POST-
     GRES distribution using monitor -c include createdb, destroydb, createuser, destroyuser, and vac-
     uum.

-d path
     path specifies the path name of the file or tty to which frontend (i.e., monitor) debugging messages
     are to be written; the default is not to generate any debugging messages.

-b hostname
     Specifies the hostname of the machine on which the postmaster is running. Defaults to the name of
     the local host, or the value of the PGHOST environment variable (if set).

-p port
     Specifies the Internet TCP port on which the postmaster is listening for connections. Defaults to
     4321, or the value of the PGPORT environment variable (if set).

-q   Specifies that the monitor should do its work quietly. By default, it prints welcome and exit mes-
     sages and the queries it sends to the backend. If this option is used, none of this happens.

-t tty_device
     tty_device specifies the path name to the file or tty to which backend (i.e., postgres) debugging mes-
     sages are to be written; the default is /dev/null.
You may set environment variables to avoid typing some of the above options. See the ENVIRONMENT VARIABLES section below.

MESSAGES AND PROMPTS
The terminal monitor gives a variety of messages to keep the user informed of the status of the monitor and the query buffer.

When the terminal monitor is executed, it displays the current date and time as well as a prompt.

The terminal monitor displays two kinds of messages:

- go  The query buffer is empty and the terminal monitor is ready for input. Anything typed will be added to the buffer.
- *  This prompt is typed at the beginning of each line when the terminal monitor is waiting for input.

TERMINAL MONITOR COMMANDS

- \e  Enter the editor to edit the query buffer.
- \g  Submit query buffer to POSTGRES for execution.
- \h  Get on-line help.
- \i filename  Include the file filename into the query buffer.
- \p  Print the current contents of the query buffer.
- \q  Exit from the terminal monitor.
- \r  Reset (clear) the query buffer.
- \s  Escape to a UNIX subshell. To return to the terminal monitor, type "exit" at the shell prompt.
- \t  Print the current time.
- \w filename  Store (write) the query buffer to an external file filename.
- \  Produce a single backslash at the current location in query buffer.

ENVIRONMENT VARIABLES
You may set any of the following environment variables to avoid specifying command-line options:

- hostname: PGHOST
- port: PGPORT
- tty: PGTTY
- options: PGOPTION
- realm: PGREALM

If PGOPTION is specified, then the options it contains are parsed before any command-line options.

PGREALM only applies if Kerberos authentication is in use. If this environment variable is set, POSTGRES will attempt authentication with servers for this realm and use separate ticket files to avoid conflicts with local ticket files. See introduction(unix) for additional information on Kerberos.

See introduction(libpq) for additional details.

RETURN VALUE
When executed with the -c option, monitor returns 0 to the shell on successful query completion, 1 otherwise.

02/12/94
SEE ALSO

introduction(libpq), createdb(unix), createuser(unix), postgres(unix), postmaster(unix).

BUGS

Does not poll for asynchronous notification events generated by listen(commands) and notify(commands).

Escapes (backslash characters) cannot be commented out.
NAME
newbki — change the POSTGRES superuser in the database template files

SYNOPSIS
newbki username

DESCRIPTION
Newbki is a script that changes the UNIX user name and user ID of the POSTGRES superuser in the database template files.

As packaged, POSTGRES assumes that there exists a user named "postgres" on your system with the same user ID as on our systems. This will not (in general) be the case. Before trying to create any databases, you should run newbki to update the template files.

Note that this only updates the files from which the template database will be built if you run the initdb(unix) command. This in turn implies that you will have to run cleardbdir(unix) to destroy the existing template database and any existing user databases — initdb will not run unless this has been done.

FILES
$PGDATA/base
  The location of global (shared) classes.

$PGDATA/base/template1
  The location of the template database.

$PGDATA/files/[(gloabal, local)_template1].bki
  Command files used to generate the global and template databases, generated and installed by the initial compilation process. These are the only files modified by newbki.

SEE ALSO
initdb(cleardbdir), createdb(unix), initdb(unix), bki(files), template(files).

CAVEATS
There is no good way to change the POSTGRES user ID after you have started creating new databases. newbki is definitely not the recommended way to try to do this. You might think that you can save your databases in flat files using copy(commands) and then restore them after initdb has been executed. However, there is the additional problem that the POSTGRES user ID is embedded in the system catalog data itself.
NAME
pagedoc — POSTGRES data page editor

SYNOPSIS
pagedoc [-h|b|r ] [-d level] [-s start] [-n count] filename

DESCRIPTION
The *pagedoc* program understands the layout of data on POSTGRES data pages and can be used to view contents of a relation *filename* if it becomes corrupted. Contents are printed to standard output, and probable errors are flagged with four asterisks ("****") and a description of the problem.

Several levels of detail are available. Level zero prints only a single summary line per data page in the relation. The summary line includes the number of items on the page, some allocation information, and whatever additional detail is appropriate for the relation type being examined. Level one also prints a single summary line for each tuple that appears on each page. The tuple summary includes the tuple's position on the page, its length, and some allocation information. Level two (or higher) prints all of the information printed by level one, and prints tuple headers for every tuple on the page. The header information displayed depends on the type of relation being viewed; either HeapTuple or IndexTuple structure entries are possible.

If the relation's contents are badly damaged, then only level zero is likely to work. Finer levels of detail assume that more page structure is correct, and so are more sensitive to corruption.

*Pagedoc* understands the following command-line options:

- `-h|b|r`  The type of the relation. Type *h* is heap, *b* is btree, and *r* is rtree. The default is *h*.
- `-d level`  The detail level to use in displaying pages.
- `-s start`  Start at page number *start* (zero-based) rather than on page zero.
- `-n count`  Display data for *count* pages rather than all of them.

EXAMPLES
Print page and line pointer summaries and tuple headers for a btree index "pg_typeidind":

```
pagedoc -b -d2 pg_typeidind
```

Show the default (level zero) summary of a heap relation "pg_user":

```
pagedoc pg_user
```

SEE ALSO
page(files).

BUGS
Finer levels of detail produce a lot of output.

There's no way to skip forward to a page that shows some corruption.

You can only examine contents, you can't actually fix them.
NAME
pcat – cat an Inversion file to stdout

SYNOPSIS
pcat [-D database] [-H host] [-P port] filename (, filename ... )

DESCRIPTION
Pcat catenates files from the Inversion file system to standard output.

ARGUMENTS
filename The name of the Inversion file to copy to standard output. If filename is "-" (a single dash),
then standard input is copied.

-D database
Specifies the database to use. Defaults to the value of the environment variable DATABASE
(see below).

-H host
Specifies the hostname of the machine on which the postmaster is running. Defaults to the
name of the local host, or the value of the PGHOST environment variable (if set).

-P port
Specifies the Internet TCP port on which the postmaster is listening for connections. Defaults
to 4321, or the value of the PGPORT environment variable (if set).

EXAMPLES
The command

pcat /myfile1 - /myfile2

copies the contents of Inversion file "/myfile1", standard input, and the contents of Inversion file "/myfile2"
to standard output.

ENVIRONMENT
If no database is given on the command line, the environment variable DATABASE is checked. If no envi-
ronment variable DATABASE is present, the command exits with an error status.

BUGS
See introduction(large objects) for filename and path limitations imposed by the Inversion file system.
NAME
pod - change directories in an Inversion file system

SYNOPSIS
pod [-D database] [-H host] [-P port] [pathname]

DESCRIPTION
Pod updates the current working directory environment variable.

ARGUMENTS
pathname The name of the directory to change to. If no pathname is given, the path is assumed to be "/".

-D database
Specifies the database to use. Defaults to the value of the environment variable DATABASE (see below).

-H host
Specifies the hostname of the machine on which the postmaster is running. Defaults to the name of the local host, or the value of the PGHOST environment variable (if set).

-P port
Specifies the Internet TCP port on which the postmaster is listening for connections. Defaults to 4321, or the value of the PGPORT environment variable (if set).

ENVIRONMENT
The environment variable PFCWD is checked and updated.

If no database is given on the command line, the environment variable DATABASE is checked. If no environment variable DATABASE is present, the command exits with an error status.

BUGS
See introduction (large objects) for filename and path limitations imposed by the Inversion file system.
NAME

`pis` - list contents of the Inversion file system

SYNOPSIS

`pis < ls flags>`

DESCRIPTION

`pis` prints directory listings of the Inversion file system. It takes the same arguments as the UNIX `ls` command.

EXAMPLES

The command

```
pis -laga /
```

prints a long-format listing of all the files in the root directory of Inversion, including size and ownership information.

ENVIRONMENT

The environment variable `DATABASE` is checked to determine the name of the database to use to find Inversion files. `PGHOST` and `PGPORT` must be used to specify the hostname of the machine on which the `postmaster` is running (defaults to the name of the local host) and the Internet TCP port on which the `postmaster` is listening for connections (defaults to 4321), respectively.

BUGS

The database name, port number, and host name to use for database accesses should be passed on the command line. Unfortunately, almost all the available option letters are already used by `ls`.

See `introduction` (large objects) for filename and path limitations imposed by the Inversion file system.
NAME
politdir - create a new Inversion file system directory

SYNOPSIS
politdir [-D database] [-H host] [-P port] path { path ... }

DESCRIPTION
Politdir creates new directories on the Inversion file system. The Inversion file system has a hierarchical
namespace with the same rules as that of the Unix filesystem: components in a pathname are separated by
slashes, and an initial slash refers to the root directory of the file system.

ARGUMENTS
pathname The name of the directory to create.

-D database
Specifies the database to use. Defaults to the value of the environment variable DATABASE
(see below).

-H host Specifies the hostname of the machine on which the postmaster is running. Defaults to the
name of the local host, or the value of the PGHOST environment variable (if set).

-P port Specifies the Internet TCP port on which the postmaster is listening for connections. Defaults
to 4321, or the value of the PGPORT environment variable (if set).

EXAMPLES
The command

politdir /a/b/c/d

creates a new directory "d" as a child of "/a/b/c". "/a/b/c" must already exist.

ENVIRONMENT
If no database is given on the command line, the environment variable DATABASE is checked. If no envi-
ronment variable DATABASE is present, the command exits with an error status.

The environment variable PFCWD is used for the current directory if the pathname specified is relative.

BUGS
See introduction (large objects) for filename and path limitations imposed by the Inversion file system.
NAME

pmv - rename an Invenio file or directory

SYNOPSIS

pmv [-D database] [-H host] [-P port] oldpath newpath

DESCRIPTION

Pmv changes the name of an existing file or directory on the Inversion file system. In the case that a directory is moved, the children of the original directory remain children of the directory under its new name.

ARGUMENTS

oldpath  The path name of the file or directory to rename. This must be a fully-qualified path rooted at “/”, and the named file or directory must exist.

newpath  The new pathname for the file or directory. Again, this must be fully qualified, and intermediate components must exist — that is, you cannot move a file to a directory which does not yet exist.

-D database  Specifies the database to use. Defaults to the value of the environment variable DATABASE (see below).

-H host  Specifies the hostname of the machine on which the postmaster is running. Defaults to the name of the local host, or the value of the PGHOST environment variable (if set).

-P port  Specifies the Internet TCP port on which the postmaster is listening for connections. Defaults to 4321, or the value of the PGPORT environment variable (if set).

EXAMPLES

The command

```
  pmv c/d b/c/longname
```

renames the Inversion file “d” in directory “c” to “b/c/longname”.

ENVIRONMENT

If no database is given on the command line, the environment variable DATABASE is checked. If no environment variable DATABASE is present, the command exits with an error status.

The environment variable PFCWD is used for the current directory if the pathname specified is relative.

BUGS

See introduction (large objects) for filename and path limitations imposed by the Inversion file system.

01/23/93
NAME
postgres — the POSTGRES backend server

SYNOPSIS
          [-d debug_level] [-o output_file] [-s] [dbname]

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.

The optional argument dbname specifies the name of the database to be accessed. Dbname defaults to the
value of the USER environment variable.

The postgres server understands the following command-line options:

-B n_buffers
   If the backend is running under the postmaster, n_buffers is the number of shared-memory buffers
   that the postmaster has allocated for the backend server processes that it starts. If the backend is run-
   ning standalone, this specifies the number of buffers to allocate. This value defaults to 64.

-E    Echo all queries.

-P filedes
   filedes specifies the file descriptor that corresponds to the socket (port) on which to communicate to
   the frontend process. This option is not useful for interactive use.

-Q    Specifies "quiet" mode.

-d debug_level
   Turns on debugging at the numeric level debug_level. Turning on debugging will cause query parse
trees and query plans to be displayed.

-o output_file
   Sends all debugging and error output to output_file. If the backend is running under the postmaster,
   error messages are still sent to the frontend process as well as to output_file, 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.

DEPRECATED COMMAND OPTIONS
There are several other options that may be specified, used mainly for debugging purposes. These are listed
here only for the use of POSTGRES system developers. Use of any of these options is highly discouraged.
Furthermore, any of these options may disappear or change at any time.

-AnrbiQnlXn
   Turns on memory manager tracing; An prints allocations/deallocation events when they occur, Ar
   enables silent record-collection, Ab enables both record-collection and event-printing, AQn prints
   recorded events each n tuples processed, and AXn prints recorded events each n transactions pro-
   cessed.

   This option generates a tremendous amount of output.

-C    Don't check whether database metadescrimptions (i.e., PG_VERSION files) are consistent.

-L    Turns off the locking system.
-N  Disables use of newline as a query delimiter.

-S  Indicates that the transaction system can run with the assumption of stable main memory, thereby avoiding the necessary flushing of data and log pages to disk at the end of each transaction system. This is only used for performance comparisons for stable vs. non-stable storage. Do not use this in other cases, as recovery after a system crash may be impossible when this option is specified in the absence of stable main memory.

-b  Enables generation of bushy query plan trees (as opposed to left-deep query plans trees). These query plans are not intended for actual execution; in addition, this flag often causes POSTGRES to run out of memory.

-f  Forbids the use of particular scan and join methods: s and l disable sequential and index scans respectively, while n, m and h disable nested-loop, merge and hash joins respectively. This is another feature that may not necessarily produce executable plans.

-p  Indicates to the backend server that it has been started by a postmaster and make different assumptions about buffer pool management, file descriptors, etc.

-tpl(parser|planner|executor)
Print timing statistics for each query relating to each of the major system modules. This option cannot be used with -s.

SEE ALSO
ipcclean(unix), monitor(unix), postmaster(unix).

DIAGNOSTICS
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 postgres 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.
NAME
postmaster — run the POSTGRES postmaster

SYNOPSIS
postmaster [-B n_buffers] [-D data_dir] [-S] [-a system]
   [-b backend_pathname] [-d debug_level] [-n]
   [-o backend_options] [-p port] [-s]

DESCRIPTION
The postmaster manages the communication between frontend and backend processes, as well as allocating
the shared buffer pool and semaphores (on machines without a test-and-set instruction). The postmaster
does not itself interact with the user and should be started as a background process. Only one postmaster
should be run on a machine.

The postmaster understands the following command-line options:

-B n_buffers
   n_buffers is the number of shared-memory buffers for the postmaster to allocate and manage for the
   backend server processes that it starts. This value defaults to 64.

-D data_dir
   Specifies the directory to use as the root of the tree of database directories. This directory uses the
   value of the environment variable PGDATA. If PGDATA is not set, then the directory used is $POST-
   GRESHOME/data. If neither environment variable is set and this command-line option is not speci-
   fied, the default directory that was set at compile-time is used.

-S
   Specifies that the postmaster process should start up in silent mode. That is, it will disassociate from
   the user’s (controlling) tty and start its own process group. This should not be used in combination
   with debugging options because any messages printed to standard output and standard error are dis-
   carded.

-a system
   Specifies whether or not to use the authentication system system (see introduction(unix)) for frontend
   applications to use in connecting to the postmaster process. Specify system to enable a system, or
   nosystem to disable a system. For example, to permit users to use Kerberos authentication, use -a
   kerberos; to deny any unauthenticated connections, use -a noneauth. The default is site-specific.

-b backend_pathname
   backend_pathname is the full pathname of the POSTGRES backend server executable file that the
   postmaster will invoke when it receives a connection from a frontend application. If this option is
   not used, then the postmaster tries to find this executable file in the directory in which its own
   executable is located (this is done by looking at the pathname under which the postmaster was
   invoked. If no pathname was specified, then the PATH environment variable is searched for an
   executable named "postgres").

-d debug_level
   The optional argument debug_level determines the amount of debugging output the backend servers
   will produce. If debug_level 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 spec-
   ified for backend servers to send their debugging output (e.g., using the -t option of monitor or the -o
   option of postgres) then this output will appear on the controlling tty of their parent postmaster.

-n, -s
   The -s and -n options control the behavior of the postmaster when a backend dies abnormally. Nei-
   ther 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 reinitalize the shared memory and semaphores. This is because an errant backend could have corrupted some shared state before terminating.

If the -s option is supplied, then the 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.

If the -n option is supplied, then the postmaster does not reinitalize shared data structures. A knowledgeable system programmer can then use the shmemdoc program to examine shared memory and semaphore state.

-o backend_options
The postgres(unix) options specified in backend_options 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 Internet TCP port on which the postmaster is to listen for connections from frontend applications. Defaults to 4321, or the value of the PGPORT environment variable (if set). If you specify a port other than the default port then all frontend application users must specify the same port (using command-line options or PGPORT) when starting any libpq application, including the terminal monitor.

WARNINGS
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. Hence, avoid

```
kill -KILL
```

or its alternative form

```
kill -9
```

as this will prevent the postmaster from freeing the system resources (e.g., shared memory and semaphores) that it holds before dying. This prevents you from having to deal with the problem with shmat(2) described below.

EXAMPLES

```
# start postmaster using default values
postmaster &
```

This command will start up postmaster on the default port (4321) and will search $PATH to find an executable file called "postgres". This is the simplest and most common way to start the postmaster.

```
# start with specific port and executable name
postmaster -p 1234 -b /usr/postgres/bin/postgres &
```

This command will start up a postmaster communicating through the port 1234, and will attempt to use the backend located at "/usr/postgres/bin/postgres". In order to connect to this postmaster using the terminal monitor, you would need to either specify -p 1234 on the monitor command-line or set the environment variable PGPORT to 1234.

SEE ALSO

ipcs(1), ipcrm(1), ipcclean(unix), monitor(unix), postgres(unix), shmemdoc(unix).
DIAGNOSTICS

semget: No space left on device

If you see this message, you should run the `ipcclean` command. After doing this, try starting the `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 `postmasters` on a single host, or have reduced the shared memory and semaphore parameters from the defaults in the generic kernel, you may have to go back and increase the shared memory and semaphores configured into your kernel.

StreamServerPort: cannot bind to port

If you see this message, you should be certain that there is no other `postmaster` process already running. The easiest way to determine this is by using the command

```
ps -ax | grep postmaster
```

on BSD-based systems (the equivalent syntax is

```
ps -e | grep postmast
```

on 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 `postmasters` 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.
NAME
   ppwd — return Inversion file system working directory name

SYNOPSIS
   ppwd

DESCRIPTION
   Ppwd writes the absolute pathname of the current working directory to the standard output.
   Ppwd exits with status 0 on success, and >0 if an error occurs.

ENVIRONMENT
   The environment variable PFCWD stores the current Inversion working directory.

SEE ALSO
   pcd(unix), p_getwd(large_objects).
NAME
prm – remove an Inversion file

SYNOPSIS
prm [-D database] [-H host] [-P port] pathname

DESCRIPTION
Prm removes a file stored by the Inversion file system. Directories must be removed using the prmdir command.

ARGUMENTS
- pathname  The fully-qualified pathname of the file to remove, rooted at “/”.
- -D database  Specifies the database to use. Defaults to the value of the environment variable DATABASE (see below).
- -H host  Specifies the hostname of the machine on which the postmaster is running. Defaults to the name of the local host, or the value of the PGHOST environment variable (if set).
- -P port  Specifies the Internet TCP port on which the postmaster is listening for connections. Defaults to 4321, or the value of the PGPORT environment variable (if set).

EXAMPLES
The command

    prm b/c/d

removes file “d” from directory “b/c”.

ENVIRONMENT
If no database is given on the command line, the environment variable DATABASE is checked. If no environment variable DATABASE is present, the command exits with an error status.

The environment variable PFCWD is used for the current directory if the pathname specified is relative.

BUGS
It is not possible to remove files stored on write-once storage managers (e.g., the Sony optical disk jukebox at Berkeley).

See introduction(large objects) for filename and path limitations imposed by the Inversion file system.
PRMDIR(UNIX)

NAME
prmdir – remove an Inversion directory

SYNOPSIS
prmdir [-D database] [-H host] [-P port] pathname

DESCRIPTION
Prmdir removes a directory from the Inversion file system. The directory must be empty. Files in director-
ies may be removed by using the prn command.

ARGUMENTS
pathname The fully-qualified pathname of the directory to remove, rooted at “/”.

-D database
Specifies the database to use. Defaults to the value of the environment variable DATABASE
(see below).

-H host
Specifies the hostname of the machine on which the postmaster is running. Defaults to the
name of the local host, or the value of the PGHOST environment variable (if set).

-P port
Specifies the Internet TCP port on which the postmaster is listening for connections. Defaults
to 4321, or the value of the PGPORT environment variable (if set).

EXAMPLES
The command

    prmdir b/c

removes directory “b/c” from the Inversion file system.

ENVIRONMENT
If no database is given on the command line, the environment variable DATABASE is checked. If no envi-
ronment variable DATABASE is present, the command exits with an error status.

The environment variable PPCWD is used for the current directory if the pathname specified is relative.

BUGS
It is not possible to remove files stored on write-once storage managers (e.g., the Sony optical disk jukebox
at Berkeley).

See introduction(large objects) for filename and path limitations imposed by the Inversion file system.
NAME
  reindexdb – reconstruct damaged system catalog indices

SYNOPSIS
  reindexdb dbname

DESCRIPTION
  In normal processing mode, POSTGRES requires secondary indices on certain system catalog classes. It is possible that these indices can be damaged during updates, e.g., if the backend server is killed during a query that creates a new class. Once the indices are damaged, it becomes impossible to access the database. Reindexdb removes the old indices and attempts to reconstruct them from the base class data.

  Before running reindexdb, make sure that the postmaster process is not running on the database server host.

  Reindexdb is a shell script that invokes the backend server directly. Hence, it must be executed by the POSTGRES super-user.

SEE ALSO
  initdb(unix), postmaster(unix).

CAVEATS
  Should only be used as a last resort. Many problems are better solved by simply shutting down the postmaster process and restarting it.

  If the base system catalog classes are damaged, reindexdb will generally print a cryptic message and fail. In this case, there is very little recourse but to reload the data.
NAME
s2kutils — scripts to allow operation with a different Kerberos realm

SYNOPSIS
s2kinit
s2klist
s2kdestroy

DESCRIPTION
s2kinit, s2klist and s2kdestroy are wrappers around the Kerberos programs kinit(1), klist(1) and kdestroy(1) that cause them to operate in the realm indicated by the environment variable PGREALM. This includes the use of ticket files distinct from those obtained for use in the local realm.

The PGREALM environment variable is also understood by the authentication code invoked by LIBPQ applications. Hence, if PGREALM is set, tickets obtained using s2kinit are used by monitor and the Inversion file system utilities. If PGREALM is not set, then the programs display the usual Kerberos behavior.

SEE ALSO
monitor(UNIX), kerberos(1), kinit(1), klist(1), kdestroy(1)

BUGS
These have almost nothing to do with POSTGRES. They are here as a convenience to Sequoia 2000 researchers who do not work in the Sequoia 2000 realm except to use POSTGRES.

You still have to insert the correct realm-server mapping into /etc/krb.conf.
NAME
shmemdoc — POSTGRES shared memory editor

SYNOPSIS
shmemdoc [-p port] [-B nbuffers]

DESCRIPTION
The shmemdoc program understands the layout of POSTGRES data in shared memory and can be used to
examine these shared structures. This program is intended only for debugging POSTGRES, and should not
be used in normal operation.

When some backend server dies abnormally, the postmaster normally reinitializes shared memory and
semaphores and forces all peers of the dead process to exit. If postmaster is started with the -n flag, then
shared memory will not be reinitialized and shmemdoc can be used to examine shared state after the crash.

Shmemdoc understands the following command-line options:

-B nbuffers
The number of buffers used by the backend. This value is ignored in the present implementation of
shmemdoc, but is important if you choose to change the number allocated by POSTGRES. In that
case, you're out of luck for now.

-p port
The port on which the postmaster was listening. This value is used to compute the shared memory
key used by the postmaster when shared memory was initialized.

A simple command interpreter reads user commands from standard input and prints results on standard output. The available commands are:

semstat
Show the status of system semaphores. Status includes semaphore names and values, the process id
of the last process to change each semaphore, and a count of processes sleeping on each semaphore.

semset n val
Set the value of semaphore number n (with zero being the first semaphore named by semstat) to val.
This is really only useful for resetting system state manually after a crash, which is something you
don't really want to do.

bufdescs
Print the contents of the shared buffer descriptor table.

bufdesc n
Print the shared buffer descriptor table entry for buffer n.

buffer n type level
Print the contents of buffer number n in the shared buffer table. The buffer is interpreted as a page
from a type relation, where type may be heap, btree, or rtree. The level argument controls the
amount of detail presented. Level zero prints only page headers, level one prints page headers and
line pointer tables, and level two (or higher) prints headers, line pointer tables, and tuples.

linp n which
Print line pointer table entry which of buffer n.

tuple n type which
Print tuple which of buffer n. The buffer is interpreted as a page from a type relation, where type may
be heap, btree, or rtree.
setbase *ptr*
Set the logical base address of shared memory for `shmemdoc` to *ptr*. Normally, `shmemdoc` uses the address of each structure in its own address space when interpreting commands and printing results. If `setbase` is used, then on input and output, addresses are translated so that the shared memory segment appears to start at address *ptr*. This is useful when a debugger is examining a core file produced by POSTGRES and you want to use the shared memory addresses that appear in the core file. The base of shared memory in POSTGRES is stored in the variable `ShmemBase`, which may be examined by a debugger. *Ptr* may be expressed in octal (leading zero), decimal, or hexadecimal (leading 0x).

shmemstat
Print shared memory layout and allocation statistics.

whatis *ptr*
Identify the shared memory structure pointed at by *ptr*.

help
Print a brief command summary.

quit
Exit `shmemdoc`.

SEE ALSO
`ipcclean(unix)`.

BUGS
All of the sizes, offsets, and values for shared data are hardwired into this program; it shares no code with the ordinary POSTGRES system, so changes to shared memory layout will require changes to this program, as well. This hasn't been done recently, so as of Version 4.2 this program doesn't work correctly for many structures (most notably the shared memory buffer pool). Use of this command is highly discouraged.
## SECTION 3 — WHAT COMES WITH POSTGRES (BUILT-INS)

### DESCRIPTION
This section describes the data types, functions and operators available to users in POSTGRES as it is distributed.

### BUILT-IN AND SYSTEM TYPES
This section describes both built-in and system data types. Built-in types are required for POSTGRES to run. System types are installed in every database, but are not strictly required. Built-in types are marked with asterisks in the table below.

Users may add new types to POSTGRES using the `define type` command described in this manual. User-defined types are not described in this section.

<table>
<thead>
<tr>
<th>POSTGRES Type</th>
<th>Meaning</th>
<th>Required</th>
</tr>
</thead>
<tbody>
<tr>
<td>abstime</td>
<td>absolute date and time</td>
<td>*</td>
</tr>
<tr>
<td>aclitem</td>
<td>access control list item</td>
<td>*</td>
</tr>
<tr>
<td>bool</td>
<td>boolean</td>
<td>*</td>
</tr>
<tr>
<td>box</td>
<td>2-dimensional rectangle</td>
<td>*</td>
</tr>
<tr>
<td>bytea</td>
<td>variable length array of bytes</td>
<td>*</td>
</tr>
<tr>
<td>char</td>
<td>character</td>
<td>*</td>
</tr>
<tr>
<td>char2</td>
<td>array of 2 characters</td>
<td>*</td>
</tr>
<tr>
<td>char4</td>
<td>array of 4 characters</td>
<td>*</td>
</tr>
<tr>
<td>char8</td>
<td>array of 8 characters</td>
<td>*</td>
</tr>
<tr>
<td>char16</td>
<td>array of 16 characters</td>
<td>*</td>
</tr>
<tr>
<td>cid</td>
<td>command identifier type</td>
<td>*</td>
</tr>
<tr>
<td>filename</td>
<td>large object filename</td>
<td>*</td>
</tr>
<tr>
<td>int2</td>
<td>two-byte signed integer</td>
<td>*</td>
</tr>
<tr>
<td>int28</td>
<td>array of 8 int2</td>
<td>*</td>
</tr>
<tr>
<td>int4</td>
<td>four-byte signed integer</td>
<td>*</td>
</tr>
<tr>
<td>float4</td>
<td>single-precision floating-point number</td>
<td>*</td>
</tr>
<tr>
<td>float8</td>
<td>double-precision floating-point number</td>
<td>*</td>
</tr>
<tr>
<td>lseg</td>
<td>2-dimensional line segment</td>
<td>*</td>
</tr>
<tr>
<td>oid</td>
<td>object identifier type</td>
<td>*</td>
</tr>
<tr>
<td>oid8</td>
<td>array of 8 oid</td>
<td>*</td>
</tr>
<tr>
<td>oidchar16</td>
<td>oid and char16 composed</td>
<td>*</td>
</tr>
<tr>
<td>oidint2</td>
<td>oid and int2 composed</td>
<td>*</td>
</tr>
<tr>
<td>oidint4</td>
<td>oid and int4 composed</td>
<td>*</td>
</tr>
<tr>
<td>path</td>
<td>variable-length array of lseg</td>
<td>*</td>
</tr>
<tr>
<td>point</td>
<td>2-dimensional geometric point</td>
<td>*</td>
</tr>
<tr>
<td>polygon</td>
<td>2-dimensional polygon</td>
<td>*</td>
</tr>
<tr>
<td>regproc</td>
<td>registered procedure</td>
<td>*</td>
</tr>
<tr>
<td>reltime</td>
<td>relative date and time</td>
<td>*</td>
</tr>
<tr>
<td>smgr</td>
<td>storage manager</td>
<td>*</td>
</tr>
<tr>
<td>text</td>
<td>variable length array of characters</td>
<td>*</td>
</tr>
<tr>
<td>tid</td>
<td>tuple identifier type</td>
<td>*</td>
</tr>
<tr>
<td>tinterval</td>
<td>time interval</td>
<td>*</td>
</tr>
<tr>
<td>xid</td>
<td>transaction identifier type</td>
<td>*</td>
</tr>
</tbody>
</table>

As a rule, the built-in types are all either (1) internal types, in which case the user should not worry about
their external format, or (2) have obvious formats. The exceptions to this rule are the three time types.

**ABSOLUTE TIME**

Absolute time is specified using the following syntax:

```
Month Day [ Hour : Minute : Second ] Year [ Timezone ]
```

where

- **Month** is Jan, Feb, ..., Dec
- **Day** is 1, 2, ..., 31
- **Hour** is 01, 02, ..., 24
- **Minute** is 00, 01, ..., 59
- **Second** is 00, 01, ..., 59
- **Year** is 1901, 1902, ..., 2038

Valid dates are from Dec 13 20:45:53 1901 GMT to Jan 19 03:14:04 2038 GMT. As of Version 3.0, times are no longer read and written using Greenwich Mean Time; the input and output routines default to the local time zone.

The special absolute time values “current”, “infinity” and “-infinity” are also provided. “infinity” specifies a time later than any valid time, and “-infinity” specifies a time earlier than any valid time. “current” indicates that the current time should be substituted whenever this value appears in a computation.

The strings “now” and “epoch” can be used to specify time values. “now” means the current time, and differs from “current” in that the current time is immediately substituted for it. “epoch” means Jan 1 00:00:00 1970 GMT.

**RELATIVE TIME**

Relative time is specified with the following syntax:

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

where

- **Quantity** is ‘1’, ‘2’, ...
- **Unit** is “second”, “minute”, “hour”, “day”, “week”, “month” (30-days), or “year” (365-days), or PLURAL of these units.
- **Direction** is “ago”

(Note: Valid relative times are less than or equal to 68 years.) In addition, the special relative time “Undefined RelTime” is provided.

**TIME RANGES**

Time ranges are specified as:

```
[ 'abstime' 'abstime' ]
```

where **abstime** is a time in the absolute time format. Special abstime values such as “current”, “infinity” and “-infinity” can be used.

**OPERATORS**

PostgreSQL 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 object ID of the procedure that implements the operator.

Users may invoke operators using the operator name, as in

```
retrieve (emp.all) where emp.salary < 40000
```
Alternatively, users may call the functions that implement the operators directly. In this case, the query above would be expressed as

\[
\text{retrieve (emp.all) where int4lt(emp.salary, 40000)}
\]

The rest of this section provides a list of the built-in operators and the functions that implement them. Binary operators are listed first, followed by unary operators.

**BINARY OPERATORS**

This list was generated from the POSTGRES system catalogs with the query

\[
\text{retrieve (argtype = t1.typname, o.oprname, t0.typname, p.proname, ltype=t1.typname, rtype=t2.typname) from p in pg_proc, t0 in pg_type, t1 in pg_type, t2 in pg_type, o in pg_operator where p.prorettype = t0.oid and RegprocToOid(o.oprcode) = p.oid and p.pronargs = 2 and o.oprleft = t1.oid and o.oprright = t2.oid}
\]

The list is sorted by the built-in type name of the first operand. The function prototype column gives the return type, function name, and argument types for the procedure that implements the operator. Note that these function prototypes are cast in terms of POSTQUEL types and so are not directly usable as C function prototypes.

<table>
<thead>
<tr>
<th>Type</th>
<th>Operator</th>
<th>POSTGRES Function Prototype</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>abstime</td>
<td>!=</td>
<td>bool abstimene(abstime, abstime)</td>
<td>inequality</td>
</tr>
<tr>
<td></td>
<td>+</td>
<td>abstime timepl(abstime, reltime)</td>
<td>addition</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>abstime timemn(abstime, reltime)</td>
<td>subtraction</td>
</tr>
<tr>
<td></td>
<td>&lt;=</td>
<td>bool abstimel(abstime, abstime)</td>
<td>less or equal</td>
</tr>
<tr>
<td></td>
<td>&lt;!&gt;</td>
<td>bool intval(abstime, tinterval)</td>
<td>abstime in tinterval?</td>
</tr>
<tr>
<td></td>
<td>&lt;</td>
<td>bool abstimelt(abstime, abstime)</td>
<td>less than</td>
</tr>
<tr>
<td></td>
<td>=</td>
<td>bool abstimeq(abstime, abstime)</td>
<td>equality</td>
</tr>
<tr>
<td></td>
<td>&gt;=</td>
<td>bool abstimge(abstime, abstime)</td>
<td>greater or equal</td>
</tr>
<tr>
<td></td>
<td>&gt;</td>
<td>bool abstimgt(abstime, abstime)</td>
<td>greater than</td>
</tr>
<tr>
<td>bool</td>
<td>=</td>
<td>bool booleq(bool, bool)</td>
<td>equality</td>
</tr>
<tr>
<td></td>
<td>!=</td>
<td>bool boolne(bool, bool)</td>
<td>inequality</td>
</tr>
<tr>
<td>box</td>
<td>&amp;</td>
<td>bool box_overlap(box, box)</td>
<td>boxes overlap</td>
</tr>
<tr>
<td></td>
<td>&amp;&lt;</td>
<td>bool box_olpleft(box, box)</td>
<td>box A overlaps box B, but does not extend to right of box B</td>
</tr>
<tr>
<td></td>
<td>&amp;&gt;</td>
<td>bool box_olpright(box, box)</td>
<td>box A overlaps box B, but does not extend to left of box B</td>
</tr>
<tr>
<td></td>
<td>&lt;&lt;</td>
<td>bool box_left(box, box)</td>
<td>A is left of B</td>
</tr>
<tr>
<td></td>
<td>&lt;=</td>
<td>bool box_le(box, box)</td>
<td>area less or equal</td>
</tr>
<tr>
<td></td>
<td>&lt;</td>
<td>bool box_lt(box, box)</td>
<td>area less than</td>
</tr>
<tr>
<td></td>
<td>=</td>
<td>bool box_eq(box, box)</td>
<td>area equal</td>
</tr>
<tr>
<td></td>
<td>&gt;=</td>
<td>bool box_ge(box, box)</td>
<td>area greater or equal</td>
</tr>
<tr>
<td>Symbol</td>
<td>Description</td>
<td></td>
<td></td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&gt;</code></td>
<td><code>bool box_right(box, box)</code> A is right of B</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&gt;</code></td>
<td><code>bool box.gt(box, box)</code> area greater than</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>@</code></td>
<td><code>bool box.contained(box, box)</code> A is contained in B</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>^</code></td>
<td><code>bool box.same(box, box)</code> box equality</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>~</code></td>
<td><code>bool box.contains(box, box)</code> A contains B</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>=</code></td>
<td><code>bool char.same(char, char)</code> box equality</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>*</code></td>
<td><code>bool char.mul(char, char)</code> multiplication</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>+</code></td>
<td><code>bool char.add(char, char)</code> addition</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>-</code></td>
<td><code>bool char.sub(char, char)</code> subtraction</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>/</code></td>
<td><code>bool char.div(char, char)</code> division</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;=</code></td>
<td><code>bool char.le(char, char)</code> less or equal</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;</code></td>
<td><code>bool char.lt(char, char)</code> less than</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>==</code></td>
<td><code>bool char.eq(char, char)</code> equality</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&gt;=</code></td>
<td><code>bool char.ge(char, char)</code> greater or equal</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&gt;</code></td>
<td><code>bool char.gt(char, char)</code> greater than</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Legend:

- `char`: A and B are characters.
- `char2`: A and B are strings.
- `char4`, `char8`: A and B are strings.

Note: `char2`, `char4`, `char8` are classes that represent strings.
### INTRODUCTION (BUILT-INS)

- **bool char8regexeq(char8, text)**
  - A matches regular expression B (POSTGRES uses the libc regexp calls for this operation)

<table>
<thead>
<tr>
<th>char16</th>
<th>!=</th>
<th>bool char16ne(char16, char16)</th>
<th>inequality</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>!~</td>
<td>bool char16regexne(char16, text)</td>
<td>A does not match regular expression B (POSTGRES uses the libc regexp calls for this operation)</td>
</tr>
<tr>
<td>&lt;=</td>
<td>bool char16le(char16, char16)</td>
<td>less or equal</td>
<td></td>
</tr>
<tr>
<td>&lt;</td>
<td>bool char16lt(char16, char16)</td>
<td>less than</td>
<td></td>
</tr>
<tr>
<td>==</td>
<td>bool char16eq(char16, char16)</td>
<td>equality</td>
<td></td>
</tr>
<tr>
<td>&gt;=</td>
<td>bool char16ge(char16, char16)</td>
<td>greater or equal</td>
<td></td>
</tr>
<tr>
<td>&gt;</td>
<td>bool char16gt(char16, char16)</td>
<td>greater than</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>float4</th>
<th>!=</th>
<th>bool float4ne(float4, float4)</th>
<th>inequality</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>*</td>
<td>float4 float4mul(float4, float4)</td>
<td>multiplication</td>
</tr>
<tr>
<td></td>
<td>+</td>
<td>float4 float4pl(float4, float4)</td>
<td>addition</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>float4 float4mi(float4, float4)</td>
<td>subtraction</td>
</tr>
<tr>
<td></td>
<td>/</td>
<td>float4 float4div(float4, float4)</td>
<td>division</td>
</tr>
<tr>
<td>&lt;=</td>
<td>bool float4le(float4, float4)</td>
<td>less or equal</td>
<td></td>
</tr>
<tr>
<td>&lt;</td>
<td>bool float4lt(float4, float4)</td>
<td>less than</td>
<td></td>
</tr>
<tr>
<td>==</td>
<td>bool float4eq(float4, float4)</td>
<td>equality</td>
<td></td>
</tr>
<tr>
<td>&gt;=</td>
<td>bool float4ge(float4, float4)</td>
<td>greater or equal</td>
<td></td>
</tr>
<tr>
<td>&gt;</td>
<td>bool float4gt(float4, float4)</td>
<td>greater than</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>float8</th>
<th>!=</th>
<th>bool float8ne(float8, float8)</th>
<th>inequality</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>*</td>
<td>float8 float8mul(float8, float8)</td>
<td>multiplication</td>
</tr>
<tr>
<td></td>
<td>+</td>
<td>float8 float8pl(float8, float8)</td>
<td>addition</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>float8 float8mi(float8, float8)</td>
<td>subtraction</td>
</tr>
<tr>
<td></td>
<td>/</td>
<td>float8 float8div(float8, float8)</td>
<td>division</td>
</tr>
<tr>
<td>&lt;=</td>
<td>bool float8le(float8, float8)</td>
<td>less or equal</td>
<td></td>
</tr>
<tr>
<td>&lt;</td>
<td>bool float8lt(float8, float8)</td>
<td>less than</td>
<td></td>
</tr>
<tr>
<td>==</td>
<td>bool float8eq(float8, float8)</td>
<td>equality</td>
<td></td>
</tr>
<tr>
<td>&gt;=</td>
<td>bool float8ge(float8, float8)</td>
<td>greater or equal</td>
<td></td>
</tr>
<tr>
<td>&gt;</td>
<td>bool float8gt(float8, float8)</td>
<td>greater than</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>int2</th>
<th>!=</th>
<th>bool int2ne(int2, int2)</th>
<th>inequality</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>!=</td>
<td>bool int2ne(int2, int4)</td>
<td>inequality</td>
</tr>
<tr>
<td></td>
<td>%</td>
<td>int2 int2mod(int2, int2)</td>
<td>modulus</td>
</tr>
<tr>
<td></td>
<td>%</td>
<td>int4 int24mod(int2, int4)</td>
<td>modulus</td>
</tr>
<tr>
<td></td>
<td>*</td>
<td>int2 int2mul(int2, int2)</td>
<td>multiplication</td>
</tr>
<tr>
<td></td>
<td>*</td>
<td>int4 int24mul(int2, int4)</td>
<td>multiplication</td>
</tr>
<tr>
<td></td>
<td>+</td>
<td>int2 int2pl(int2, int2)</td>
<td>addition</td>
</tr>
<tr>
<td></td>
<td>+</td>
<td>int4 int24pl(int2, int4)</td>
<td>addition</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>int2 int2mi(int2, int2)</td>
<td>subtraction</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>int4 int24mi(int2, int4)</td>
<td>subtraction</td>
</tr>
<tr>
<td></td>
<td>/</td>
<td>int2 int2div(int2, int2)</td>
<td>division</td>
</tr>
<tr>
<td></td>
<td>/</td>
<td>int4 int24div(int2, int4)</td>
<td>division</td>
</tr>
<tr>
<td>&lt;=</td>
<td>bool int2le(int2, int2)</td>
<td>less or equal</td>
<td></td>
</tr>
</tbody>
</table>
## INTRODUCTION (BUILT-INS)

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;=</code></td>
<td>Less or equal</td>
</tr>
<tr>
<td><code>&lt;</code></td>
<td>Less than</td>
</tr>
<tr>
<td><code>=</code></td>
<td>Equality</td>
</tr>
<tr>
<td><code>&gt;=</code></td>
<td>Greater or equal</td>
</tr>
<tr>
<td><code>&gt;</code></td>
<td>Greater than</td>
</tr>
<tr>
<td><code>int2inc(int2)</code></td>
<td>Increment</td>
</tr>
</tbody>
</table>

#### INTRODUCTION (BUILT-INS)

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>!=</code></td>
<td>Inequality</td>
</tr>
<tr>
<td><code>%</code></td>
<td>Modulus</td>
</tr>
<tr>
<td><code>*</code></td>
<td>Multiplication</td>
</tr>
<tr>
<td><code>/</code></td>
<td>Division</td>
</tr>
<tr>
<td><code>+</code></td>
<td>Addition</td>
</tr>
<tr>
<td><code>-</code></td>
<td>Subtraction</td>
</tr>
<tr>
<td><code>int4inc(int4)</code></td>
<td>Increment</td>
</tr>
</tbody>
</table>

#### INTRODUCTION (BUILT-INS)

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>!=</code></td>
<td>Inequality</td>
</tr>
<tr>
<td><code>%</code></td>
<td>Modulus</td>
</tr>
<tr>
<td><code>*</code></td>
<td>Multiplication</td>
</tr>
<tr>
<td><code>/</code></td>
<td>Division</td>
</tr>
<tr>
<td><code>+</code></td>
<td>Addition</td>
</tr>
<tr>
<td><code>-</code></td>
<td>Subtraction</td>
</tr>
<tr>
<td><code>int4inc(int4)</code></td>
<td>Increment</td>
</tr>
</tbody>
</table>

#### INTRODUCTION (BUILT-INS)

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>!=</code></td>
<td>Inequality</td>
</tr>
<tr>
<td><code>%</code></td>
<td>Modulus</td>
</tr>
<tr>
<td><code>*</code></td>
<td>Multiplication</td>
</tr>
<tr>
<td><code>/</code></td>
<td>Division</td>
</tr>
<tr>
<td><code>+</code></td>
<td>Addition</td>
</tr>
<tr>
<td><code>-</code></td>
<td>Subtraction</td>
</tr>
<tr>
<td><code>int4inc(int4)</code></td>
<td>Increment</td>
</tr>
</tbody>
</table>

#### INTRODUCTION (BUILT-INS)

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>!=</code></td>
<td>Inequality</td>
</tr>
<tr>
<td><code>%</code></td>
<td>Modulus</td>
</tr>
<tr>
<td><code>*</code></td>
<td>Multiplication</td>
</tr>
<tr>
<td><code>/</code></td>
<td>Division</td>
</tr>
<tr>
<td><code>+</code></td>
<td>Addition</td>
</tr>
<tr>
<td><code>-</code></td>
<td>Subtraction</td>
</tr>
<tr>
<td><code>int4inc(int4)</code></td>
<td>Increment</td>
</tr>
</tbody>
</table>

#### INTRODUCTION (BUILT-INS)

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>!=</code></td>
<td>Inequality</td>
</tr>
<tr>
<td><code>%</code></td>
<td>Modulus</td>
</tr>
<tr>
<td><code>*</code></td>
<td>Multiplication</td>
</tr>
<tr>
<td><code>/</code></td>
<td>Division</td>
</tr>
<tr>
<td><code>+</code></td>
<td>Addition</td>
</tr>
<tr>
<td><code>-</code></td>
<td>Subtraction</td>
</tr>
<tr>
<td><code>int4inc(int4)</code></td>
<td>Increment</td>
</tr>
</tbody>
</table>

This is the relational "not in" operator, and is not intended for public use.
<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;=</td>
<td>bool oidchar16le(oidchar16, oidchar16)</td>
<td>less or equal</td>
</tr>
<tr>
<td>==</td>
<td>bool oidchar16eq(oidchar16, oidchar16)</td>
<td>equality</td>
</tr>
<tr>
<td>&gt;</td>
<td>bool oidchar16gt(oidchar16, oidchar16)</td>
<td>greater than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>bool oidchar16ge(oidchar16, oidchar16)</td>
<td>greater or equal</td>
</tr>
<tr>
<td>!=</td>
<td>bool oidchar16ne(oidchar16, oidchar16)</td>
<td>inequality</td>
</tr>
<tr>
<td>&lt;</td>
<td>bool oidchar16lt(oidchar16, oidchar16)</td>
<td>less than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>bool oidchar16le(oidchar16, oidchar16)</td>
<td>less or equal</td>
</tr>
<tr>
<td>==</td>
<td>bool oidchar16eq(oidchar16, oidchar16)</td>
<td>equality</td>
</tr>
<tr>
<td>&gt;</td>
<td>bool oidchar16gt(oidchar16, oidchar16)</td>
<td>greater than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>bool oidchar16ge(oidchar16, oidchar16)</td>
<td>greater or equal</td>
</tr>
<tr>
<td>!=</td>
<td>bool oidchar16ne(oidchar16, oidchar16)</td>
<td>inequality</td>
</tr>
<tr>
<td>&lt;</td>
<td>bool oidchar16lt(oidchar16, oidchar16)</td>
<td>less than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>bool oidchar16le(oidchar16, oidchar16)</td>
<td>less or equal</td>
</tr>
<tr>
<td>==</td>
<td>bool oidchar16eq(oidchar16, oidchar16)</td>
<td>equality</td>
</tr>
<tr>
<td>&gt;</td>
<td>bool oidchar16gt(oidchar16, oidchar16)</td>
<td>greater than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>bool oidchar16ge(oidchar16, oidchar16)</td>
<td>greater or equal</td>
</tr>
<tr>
<td>!=</td>
<td>bool oidchar16ne(oidchar16, oidchar16)</td>
<td>inequality</td>
</tr>
<tr>
<td>&lt;</td>
<td>bool oidchar16lt(oidchar16, oidchar16)</td>
<td>less than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>bool oidchar16le(oidchar16, oidchar16)</td>
<td>less or equal</td>
</tr>
<tr>
<td>==</td>
<td>bool oidchar16eq(oidchar16, oidchar16)</td>
<td>equality</td>
</tr>
<tr>
<td>&gt;</td>
<td>bool oidchar16gt(oidchar16, oidchar16)</td>
<td>greater than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>bool oidchar16ge(oidchar16, oidchar16)</td>
<td>greater or equal</td>
</tr>
<tr>
<td>point</td>
<td>![ point_left(point, point) ]</td>
<td>A is left of B</td>
</tr>
<tr>
<td>!&gt;</td>
<td>bool point_right(point, point)</td>
<td>A is right of B</td>
</tr>
<tr>
<td>!=</td>
<td>bool point_above(point, point)</td>
<td>A is above B</td>
</tr>
<tr>
<td>==</td>
<td>bool point_below(point, point)</td>
<td>A is below B</td>
</tr>
<tr>
<td>=&gt;</td>
<td>bool point_eq(point, point)</td>
<td>equality</td>
</tr>
<tr>
<td>---&gt;</td>
<td>bool on_pb(point, box)</td>
<td>point inside box</td>
</tr>
<tr>
<td>----</td>
<td>bool on_pset(point, path)</td>
<td>point on path</td>
</tr>
<tr>
<td>&lt;&lt;&lt;</td>
<td>int4 pointdist(point, point)</td>
<td>distance between points</td>
</tr>
<tr>
<td>polygon</td>
<td>&amp; &amp; bool poly_overlap(polygon, polygon)</td>
<td>polygons overlap</td>
</tr>
<tr>
<td>&amp; &lt;</td>
<td>bool poly_overleft(polygon, polygon)</td>
<td>A overlaps B but does not extend to right of B</td>
</tr>
<tr>
<td>&amp; &gt;</td>
<td>bool poly_overright(polygon, polygon)</td>
<td>A overlaps B but does not extend to left of B</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>bool poly_left(polygon, polygon)</td>
<td>A is left of B</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>bool poly_right(polygon, polygon)</td>
<td>A is right of B</td>
</tr>
<tr>
<td>@</td>
<td>bool polycontained(polygon, polygon)</td>
<td>A is contained by B</td>
</tr>
<tr>
<td>=~</td>
<td>bool poly_same(polygon, polygon)</td>
<td>equality</td>
</tr>
<tr>
<td>-</td>
<td>bool poly_contain(polygon, polygon)</td>
<td>A contains B</td>
</tr>
<tr>
<td>reltime</td>
<td>!= bool reltime_ne(reltime, reltime)</td>
<td>inequality</td>
</tr>
<tr>
<td>&lt;=</td>
<td>bool reltime_le(reltime, reltime)</td>
<td>less or equal</td>
</tr>
<tr>
<td>&lt;</td>
<td>bool reltime_lt(reltime, reltime)</td>
<td>less than</td>
</tr>
<tr>
<td>==</td>
<td>bool reltime_eq(reltime, reltime)</td>
<td>equality</td>
</tr>
<tr>
<td>&gt;=</td>
<td>bool reltime_ge(reltime, reltime)</td>
<td>greater or equal</td>
</tr>
<tr>
<td>&gt;</td>
<td>bool reltime_gt(reltime, reltime)</td>
<td>greater than</td>
</tr>
<tr>
<td>text</td>
<td>!= bool text_ne(text, text)</td>
<td>inequality</td>
</tr>
<tr>
<td>!~</td>
<td>bool text_regex_ne(text, text)</td>
<td>A does not contain the regular expression B. POSTGRES uses the libc regexp interface for this operator.</td>
</tr>
<tr>
<td>&lt;=</td>
<td>bool text_le(text, text)</td>
<td>less or equal</td>
</tr>
<tr>
<td>&lt;</td>
<td>bool text_lt(text, text)</td>
<td>less than</td>
</tr>
<tr>
<td>==</td>
<td>bool text_eq(text, text)</td>
<td>equality</td>
</tr>
</tbody>
</table>
UNARY OPERATORS

The tables below give right and left unary operators. Left unary operators have the operator precede the operand; right unary operators have the operator follow the operand.

Right Unary Operators

<table>
<thead>
<tr>
<th>Type</th>
<th>Operator</th>
<th>POSTGRES Function Prototype</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>float8</td>
<td>%</td>
<td>float8 dround(float8)</td>
<td>round to nearest integer</td>
</tr>
</tbody>
</table>

Left Unary Operators

<table>
<thead>
<tr>
<th>Type</th>
<th>Operator</th>
<th>POSTGRES Function Prototype</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>box</td>
<td>@@</td>
<td>point_box_center(box)</td>
<td>center of box</td>
</tr>
<tr>
<td>float4</td>
<td>@</td>
<td>float4 float4abs(float4)</td>
<td>absolute value</td>
</tr>
<tr>
<td>float8</td>
<td>@</td>
<td>float8 float8abs(float8)</td>
<td>absolute value</td>
</tr>
<tr>
<td></td>
<td>%</td>
<td>float8 dtrunc(float8)</td>
<td>truncate to integer</td>
</tr>
<tr>
<td></td>
<td>(/)</td>
<td>float8 dsqrt(float8)</td>
<td>square root</td>
</tr>
<tr>
<td></td>
<td>(l/)</td>
<td>float8 dcbrt(float8)</td>
<td>cube root</td>
</tr>
<tr>
<td></td>
<td>:</td>
<td>float8 dexp(float8)</td>
<td>exponential function</td>
</tr>
<tr>
<td></td>
<td>;</td>
<td>float8 dlog1(float8)</td>
<td>natural logarithm</td>
</tr>
<tr>
<td>tinterval</td>
<td>(l)</td>
<td>abstime intervalstart(tinterval)</td>
<td>start of interval</td>
</tr>
</tbody>
</table>

AGGREGATE FUNCTIONS

The table below gives the aggregate functions that are normally registered in the system catalogs. None of them are required for POSTGRES to operate.

<table>
<thead>
<tr>
<th>Name</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>int2ave</td>
<td>int2 average</td>
</tr>
<tr>
<td>int4ave</td>
<td>int4 average</td>
</tr>
<tr>
<td>float4ave</td>
<td>float4 average</td>
</tr>
<tr>
<td>float8ave</td>
<td>float8 average</td>
</tr>
</tbody>
</table>
INTRODUCTION (BUILT-INS)

int2sum
int4sum
float4sum
float8sum
int2max
int4max
float4max
float8max
int2min
int4min
float4min
float8min
count

SEE ALSO
For examples on specifying literals of built-in types, see postscript(commands).

BUGS
The lists of types, functions, and operators are accurate only for Version 4.2. The lists will be incomplete and contain extraneous entries in future versions of POSTGRES.

Although most of the input and output functions corresponding to the base types (e.g., integers and floating point numbers) do some error-checking, none of them are particularly rigorous about it. More importantly, almost none of the operators and functions (e.g., addition and multiplication) perform any error-checking at all. Consequently, many of the numeric operations will (for example) 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.
SECTION 4 — POSTQUEL COMMANDS (COMMANDS)

DESCRIPTION
The following is a description of the general syntax of POSTQUEL. Individual POSTQUEL statements and commands are treated separately in the document; this section describes the syntactic classes from which the constituent parts of POSTQUEL statements are drawn.

Comments
A comment is an arbitrary sequence of characters bounded on the left by “/*” and on the right by “*/”, e.g:

    /* This is a comment */

Names
Names in POSTQUEL are sequences of not more than 16 alphanumeric characters, starting with an alphabetic character. Underscore (“_”) is considered an alphabetic character.

Keywords
The following identifiers are reserved for use as keywords and may not be used otherwise:

    abort    define    is    quel    relation
    acl      delete    ISNULL    remove
    addattr  demand    key    leftouter
    after    descending    light
    aggregate  destroy    listen
    all      destroydb    load
    always   do    merge
    and      empty    move
    append   end    never
    archive  execute    new
    arch_store  extend    none
    arg      fetch    nonnulls
    as      forward    not
    ascending  from    notify
    attachas  function    NOTNULL
    backward  group    NULL
    before    heavy   .transaction
    begin    in    on
    binary    index    once
    by      indexable    operator
    cfunction  inherits    or
    change    input_proc    output_proc
    close    instance    parallel
    cluster  instead    pfuction
    copy      intersect    portal
    create    into    postquel
    createdb    intotemp    priority
    current    iportal    purge

In addition, all POSTGRES classes have several predefined attributes used by the system. For a list of these, see the section Fields, below.

03/12/94
Constants

There are six types of constants for use in POSTQUEL. They are described below.

Character Constants

Single character constants may be used in POSTQUEL by surrounding them by single quotes, e.g., 'n'.

String Constants

Strings in POSTQUEL are arbitrary sequences of ASCII characters bounded by double quotes (" "). Upper case alphabets within strings are accepted literally. Non-printing characters may be embedded within strings by prepending them with a backslash, e.g., "\n". Also, in order to embed quotes within strings, it is necessary to prefix them with "\". The same convention applies to \" itself. Because of the limitations on instance sizes, string constants are currently limited to a length of a little less than 8192 bytes. Larger objects may be created using the POSTGRES Large Object interface.

Integer Constants

Integer constants in POSTQUEL are collections of ASCII digits with no decimal point. Legal values range from \(-2147483647\) to \(2147483647\). This will vary depending on the operating system and host machine.

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:

\( (<\text{dig}>) . (<\text{dig}>) \ [e \ [\pm \] \ (<\text{dig}>) ] \)

Where \(<\text{dig}>\) is a digit. You must include at least one \(<\text{dig}>\) after the period and after the \([\pm]\) 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 constants are taken to be double-precision quantities with a range of approximately \(-10^{38}\) to \(10^{38}\) and a precision of 17 decimal digits. This will vary depending on the operating system and host machine.

Constants of POSTGRES User-Defined Types

A constant of an arbitrary type can be entered using the notation:

"string"::type-name

In this case the value inside the string is passed to the input conversion routine for the type called type-name. 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.

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:

"(<val1><delim><val2><delim>"

Where \(<\text{delim}>\) 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.
Arrays of fixed-length types may also be stored as POSTGRES large objects (see introduction(large objects)). The syntax for an array constant of this form is

```
"large_object [-unix | -inverT] [-chunk (DEFAULT | acc_patt_file)]"
```

That is, any array constant that does not begin and end in curly braces is assumed to be an Inversion file system filename that contains the appropriate array data. The Inversion file will be created if it does not already exist. The flag "unix" or "invert" is used to indicate the type of the large object. The default type is "unix". An array stored in large object can be chunked to optimize retrievals by using the "-chunk" flag. The array can be chunked using a default chunk size (by using the keyword DEFAULT) or by using an access pattern stored in a native file "acc_patt_file". The access pattern is expected to be in the following format.

```
<n> <A_11 A_12 .. A_1d P_1> ... <A_n1 A_n2 .. A_nd P_n>
```

where n is the number of tuples in the access pattern and d is the number of dimensions of the array. For each i, <A_i1 A_i2 .. A_id> is the dimension of an access request on the array and P_i is the relative frequency of the access.

**Fields**

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

- `all`
- `oid`
- `tmin`
- `tmax`
- `xmin`
- `xmax`
- `cmin`
- `cmax`
- `vtype`

As in INGRES, `all` is a shorthand for all normal attributes in a class, and may be used profitably in the target list of a retrieve statement.

`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.

`tmin, tmax, xmin, xmax, cmin, cmax` stand respectively for the time that the instance was inserted, the time the instance was deleted, the identity of the inserting transaction, the command identifier within the transaction, the identity of the deleting transaction and its associated deleting command. For further information on these fields consult [STON87]. Times are represented internally as instances of the "abstime" data type. Transaction identifiers are 32 bit quantities which are assigned sequentially starting at 512. Command identifiers are 16 bit objects; hence, it is an error to have more than 65535 POSTQUEL commands within one transaction.

**Attributes**

An attribute is a construct of the form:

```
Instance-variable(.composite_field).field ['number']
```

`Instance-variable` 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 POSTQUEL command. Composite field is a field of one of the POSTGRES composite types indicated in the information(commands) section, 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.

Operators

Any built-in system, or user-defined operator may be used in POSTQUEL. For the list of built-in and system operators consult introduction(built-ins). 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.

Expressions (a_expr)

An expression is one of the following:

- \(( \text{a_expr} \)\)
- constant
- attribute
- \(\text{a_expr binary_operator a_expr}\)
- \(\text{a_expr right_unary_operator}\)
- \(\text{left_unary_operator a_expr}\)
- parameter
- functional expressions
- aggregate expressions
- set expressions (no general implementation in Version 4.2)
- class expression (no general implementation in Version 4.2)

We have already discussed constants and attributes. The two kinds of operator expressions indicate respectively binary and left_unary expressions. The following sections discuss the remaining options.

Parameters

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

- \(\$\) number

For example, consider the definition of a function, DEPT, as

```sql
define function DEPT
  (language="postquel", returntype = dept)
  arg is (char16) as
  retrieve (dept.all) where dept.name = $1
```

Functional Expressions

A functional expression is the name of a legal POSTQUEL function, followed by its argument list enclosed in parentheses, e.g.:

```sql
fn-name (a_expr( , a_expr))
```

For example, the following computes the square root of an employee salary.
Aggregate Expression

An aggregate expression represents a simple aggregate (i.e., one that computes a single value) or an aggregate function (i.e., one that computes a set of values). The syntax is the following:

```
aggregate_name '(' [unique [using] opr] a_expr 
    [from from_list] 
    [where qualification'] 
'
```

Here, aggregate_name must be a previously defined aggregate. The from_list indicates the class to be aggregated over while qualification gives restrictions which must be satisfied by the instances to be aggregated. Next, the a_expr gives the expression to be aggregated, while the unique tag indicates whether all values should be aggregated or just the unique values of a_expr. Two expressions, a_expr1 and a_expr2 are the same if a_expr1 opr a_expr2 evaluates to true.

In the case that all instance variables used in the aggregate expression are defined in the from list, a simple aggregate has been defined. For example, to sum employee salaries whose age is greater than 30, one would write:

```
retrieve (total = sum (e.salary from e in emp 
    where e.age > 30) )
```

or

```
retrieve (total = sum (emp.salary where emp.age > 30))
```

In either case, POSTGRES is instructed to find the instances in the from list which satisfy the qualification and then compute the aggregate of the a_expr indicated.

On the other hand, if there are variables used in the aggregate expression that are not defined in the from list, e.g:

```
avg (emp.salary where emp.age = e.age)
```

then this aggregate function has a value for each possible value taken on by “e.age”. For example, the following complete query finds the average salary of each possible employee age over 18:

```
retrieve (e.age, eavg = avg (emp.salary where emp.age = e.age)) 
    from e in emp 
    where e.age > 18
```

Aggregate functions are not supported in Version 4.2.

In general, the following aggregates (i.e., the expression within the braces) will not work:

- Aggregate functions of any kind.
- Aggregates containing more than one range variable.
- Aggregates that refer to range variables that use class inheritance (e.g., “e from emp”).
- Aggregates containing clauses other than a_expr and where-qualification clauses. (In other words, from clauses within aggregates are not supported.)
In addition, aggregate expressions may only appear within the target list of a query — that is, no aggregate expression may appear in a query qualification (or where clause).

Therefore, of the three example queries given, only the second is actually supported.

Set Expressions

Generalized set expressions are not supported in Version 4.2. For information on sets as attributes, see the manual pages for the create(commands), append(commands) and retrieve(commands) commands.

A set expression defines a collection of instances from some class and uses the following syntax:

\[
\text{(target\_list from from\_list where qualification)}
\]

For example, the set of all employee names over 40 is:

\[
\text{(emp.name where emp.age > 40)}
\]

In addition, it is legal to construct set expressions which have an instance variable which is defined outside the scope of the expression. For example, the following expression is the set of employees in each department:

\[
\text{(emp.name where emp.dept = dept.dname)}
\]

Set expressions can be used in class expressions which are defined below.

Class Expressions

Generalized class expressions are not supported in Version 4.2. For information on classes as attributes, see the manual pages for the create(commands), append(commands) and retrieve(commands) commands.

A class expression is an expression of the form:

\[
\text{class\_constructor binary\_class\_operator class\_constructor}
\]

where binary\_class\_operator is one of the following:

- union
- intersect
- 
- difference of two classes
- >>
- <=
- ==

and unary\_class\_operator can be:

- empty

A class\_constructor is either an instance variable, a class name, the value of a composite field or a set expression.

An example of a query with a class expression is one to find all the departments with no employees:

\[
\text{retrieve (dept.dname)}
\quad \text{where empty (emp.name where emp.dept = dept.dname)}
\]
**Target List**

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

\[
\text{[result_attname =]} \ a\_\text{expr}
\]

Here, 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 Version 4.2 default naming is only used if a_expr is an attribute.

**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.

**From List**

The from list is a comma-separated list of from expressions.

Each from expression is of the form:

\[
\text{instance_variable-1} \ (, \ \text{instance_variable-2}) \\
\text{in class_reference}
\]

where class_reference is of the form

\[
\text{class_name [time_expression] [^]}
\]

The from expression defines one or more instance variables to range over the class indicated in class_reference. Adding a time_expression will indicate that a historical class is desired. 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 "^".

**Time Expressions**

A time expression is in one of two forms:

\[
\{\text{*date*} \}
\]

\[
\{\text{*date-1*}, \text{*date-2*}\}
\]

The first case requires instances that are valid at the indicated time. The second case requires instances that are valid at some time within the date range specified. If no time expression is indicated, the default is "now".

In each case, the date is a character string of the form

\[
\{\text{MON-FRI} \ *\text{MMM DD [HH:MM:SS] YYYY [Timezone]} \}
\]

where MMM is the month (Jan – Dec), DD is a legal day number in the specified month, HH:MM:SS is an optional time in that day (24-hour clock), and YYYY is the year. If the time of day HH:MM:SS is not
specified, it defaults to midnight at the start of the specified day. As of Version 3.0, times are no longer read and written using Greenwich Mean Time; the input and output routines default to the local time zone. For example,

```
["Jan 1 1990"
["Mar 3 00:00:00 1980", "Mar 3 23:59:59 1981"]
```

are valid time specifications.

Note that this syntax is slightly different than that used by the time-range type.

SEE ALSO

BUGS

The following constructs are not available in Version 4.2:

- class expressions
- set expressions
NAME
abort — abort the current transaction

SYNOPSIS
abort

DESCRIPTION
This command aborts the current transaction and causes all the updates made by the transaction to be discarded.

SEE ALSO
begin(commands), end(commands).
NAME
addattr — add attributes to a class

SYNOPSIS
addattr (atname1 = type1, atname-i = type-i) to classname [*]

DESCRIPTION
The addattr command causes new attributes to be added to an existing class, classname. The new
attributes and their types are specified in the same style and with the same restrictions as in create(commands).

In order to add an attribute to each class in an entire inheritance hierarchy, use the classname of the super-
class and append a "*". (By default, the attribute will not be added to any of the subclasses.) This should
always be done when adding an attribute to a superclass. If it is not, queries on the inheritance hierarchy
such as

retrieve (s.all) from s in super*

will not work because the subclasses will be missing an attribute found in the superclass.

For efficiency reasons, default values for added attributes are not placed in existing instances of a class.
That is, existing instances will have NULL values in the new attributes. If non-NULL values are desired, a
subsequent replace(commands) query should be run.

You must own the class in order to change its schema.

EXAMPLE
/*
 * add the date of hire to the emp class
 */
addattr (hiredate = abstime) to emp

/*
 * add a health-care number to all persons
 * (including employees, students, ...)
 */
addattr (health_care_id = int4) to person*

SEE ALSO
create(commands), rename(commands), replace(commands).
NAME
append — append tuples to a relation

SYNOPSIS
append classname
( att_expr = expression , att_expr = expression )
[ from from_list ] [ where qual ]

DESCRIPTION
Append adds instances that satisfy the qualification, qual, to classname. Classname must be the name of
an existing class. The target list specifies the values of the fields to be appended to classname. That is,
each att_expr specifies a field (either an attribute name or an attribute name plus an array specification) to
which the corresponding expression should be assigned. The fields in the target list may be listed in any
order. Fields of the result class which do not appear in the target list default to NULL. If the expression for
each field is not of the correct data type, automatic type coercion will be attempted.

An array initialization may take exactly one of the following forms:

/*
 * Specify a lower and upper index for each dimension
*/
att_name[1:1][1:1] = array_str

/*
 * Specify only the upper index for each dimension
 * (each lower index defaults to 1)
*/
att_name[1..1] = array_str

/*
 * Use the upper index bounds as specified within array_str
 * (each lower index defaults to 1)
*/
att_name = array_str

where each Index or uIndex is an integer constant and array_str is an array constant (see introduction(commands)).

If the user does not specify any array bounds (as in the third form) then POSTGRES will attempt to deduce
the actual array bounds from the contents of array_str.

If the user does specify explicit array bounds (as in the first and second forms) then the array may be initialized
partly or fully using a C-like syntax for array initialization. However, the uninitialized array elements
will contain garbage.

The keyword all can be used when it is desired to append all fields of a class to another class.
If the attribute is a complex type, its contents are specified as a query which will return the tuples in the set.
See the examples below.

You must have write or append access to a class in order to append to it, as well as read access on any class
whose values are read in the target list or qualification (see change acl(commands)).
EXAMPLES

/*
* Make a new employee Jones work for Smith
*/
append emp (newemp.name, newemp.salary, mgr = "Smith",
        bdate = 1990 - newemp.age)
        where newemp.name = "Jones"

/*
* Same command using the from list clause
*/
append emp (n.name, n.salary, mgr = "Smith",
        bdate = 1990 - n.age)
        from n in newemp
        where n.name = "Jones"

/*
* Append the newempl class to newemp
*/
append newemp (newempl.all)

/*
* Create an empty 3x3 gameboard for noughts-and-crosses
* (all of these queries create the same board attribute)
*/
append tictactoe (game = 1, board[1:3][1:3] =
        "(("","",""),(,"",")")")
append tictactoe (game = 2, board[3][3] =
        ")")")
append tictactoe (game = 3, board =
        "((,,),{,,}),{,,})")")

/*
* Create a 3x3 noughts-and-crosses board that is
* completely filled-in
*/
append tictactoe (game = 4, board =
        "((X,O,X),(O,X,O),(X,X,X))")

/*
* Create a 3x3 noughts-and-crosses board that has
* only 1 place filled-in
*/
append tictactoe (game = 4, board[3][3] =
        "((,,),(X,))")")

03/12/94 61
/*
 * Create a tuple containing a large-object array.
 * The large object "/large/tictactoe/board" will be
 * created if it does not already exist. The flag "-invert"
 * indicates that the large object is of type Inversion
 * (the default type is Unix).
 */
append tictactoe (board[3][3] =
    "/large/tictactoe/board -invert")

/*
 * Create a tuple containing a large-object array and "chunk"
 * it. The Inversion file "/large/tictactoe/board" must already
 * exist. The external file "/etc/acc_patt" holds the access
 * pattern used to cluster (chunk) the array elements. A new
 * large object is created to hold the chunked array.
 * (See "src/doc/papers/arrays/paper.ps" for more information)
 */
append tictactoe (board[3][3] =
    "/large/tictactoe/board -chunk /etc/acc_patt")

/*
 * Append a tuple with a set attribute "mgr" of type emp. The
 * query to produce the manager of "carol" (specified dynamically
 * here) will be stored as a POSTQUEL function in the system
 * catalog "pg_proc". The object ID of this tuple in "pg_proc"
 * will be used in the name of the procedure, resulting in a
 * procedure name of the form "set<OID of the tuple>". Two
 * backslashes are needed here to escape the inner quotes when
 * entering this query from the monitor.
 */
append emp (name = "carol",
    mgr = *retrieve (emp.all)
        where emp.name = \"mike\"")

SEE ALSO
postquel(commands), create(commands), define type(commands), replace(commands), retrieve(commands)
introduction(large objects).

BUGS
Once an array is created by an append query, its size (in bytes) cannot be changed. This has several impli-
cations.

First, there is no longer any notion of a "variable-length array." In fact, since variable-length arrays
were not actually supported in previous versions of POSTGRES, this is not much of a change.

Second, arrays of variable-length types (e.g., text) cannot be updated. Since the array cannot
grow, replacement of individual array elements cannot be supported in general.
NAME
attachas — reestablish communication using an existing portal

SYNOPSIS
attachas name

DESCRIPTION
This command allows application programs to use a logical name, name, in interactions with POSTORES. Suppose the user of an application program specifies a collection of rules that retrieve data and that the program fails for some reason. Then, under ordinary circumstances, all the rules would need to be reentered when the program is restored. Alternatively, the attachas command may be used before defining the rules the first time. Then, upon restoring the program, the attachas command will reattach the user to the active rules.

BUGS
Attachas is not implemented in Version 4.2.
NAME
    begin — begins a transaction

SYNOPSIS
    begin

DESCRIPTION
    This command begins a user transaction which POSTGRES will guarantee is serializable with respect to all concurrently executing transactions. POSTGRES uses two-phase locking to perform this task. If the transaction is committed, POSTGRES will ensure that all updates are done or none of them are done. Transactions have the standard ACID (atomic, consistent, isolatable, and durable) property.

SEE ALSO
    abort(commands), end(commands).
NAME
change acl — change access control list(s)

SYNOPSIS
change acl (group|user) [name]+[a|rw|wr|] class-1 [, class-i]
change acl (group|user) [name]-[a|rw|wr|] class-1 [, class-i]
change acl (group|user) [name]=[a|rw|wr|] class-1 [, class-i]

DESCRIPTION
Introduction
An access control list (ACL) specifies the access modes that are permitted on a given class for a set of users and groups of users. These modes are:
- a - append data to a class
- r - read data from a class
- w - write data (append, delete, replace) to a class
- R - define rules on a class

Application of ACLs to users
Each entry in an ACL consists of an identifier and a set of permitted access modes. The identifier may apply to a single user, a group of users, or all other users. If a user has a personal entry in an ACL, then only the listed access modes are permitted. If a user does not have a personal entry but is a member of some group(s) listed in the ACL, then access is permitted if all of the listed groups of which the user is a member have the desired access mode. Finally, if a user does not have a personal entry and is not a member of any listed groups, then the desired access mode is checked against the "other" entry.

Database superusers (i.e., users who have pg_user.usesuper set) silently bypass all access controls with one exception: manual system catalog updates are never permitted if the user does not have pg_user.usecatupd set. This is intended as a convenience (safety net) for careless superusers.

Application of ACLs through time
The access control system always uses the ACLs that are currently valid, i.e., time travel is not supported. This may change if/when a more general notion of time-travel is documented.

CHANGING ACLS
In the syntax shown above, name is a user or group identifier. If the user or group keywords are left out, name is assumed to be a user name. If no name is listed at all, then the ACL entry applies to the "other" category.

Access modes are added, deleted or explicitly set using exactly one of the +, - and = mode-change flags. The access modes themselves are specified using any number of the single-letter mode flags listed above.

Only the owner of a class (or a database superuser) may change an ACL.

By default, classes start without any ACLs. Classes created using the inheritance mechanism do not inherit ACLs.

EXAMPLES
/*
 * Deny any access to "other" to classes "gcmdata" and "btdata".
 */
change acl = gcmdata, btdata

/*
 * Grant "dozier" all permissions to "gcmdata" and "btdata".
*/
CHANGE ACL COMMANDS

```c
/*
  * Allow group *sequoia* to read and append data to *gcmdata*.
  */
change acl group sequoia+r gcmdata
```

```c
/*
  * Deny *frew* the ability to define rules on *gcmdata*.
  */
change acl frew-R gcmdata
```

SEE ALSO

- `introduction(unix)`, `append(commands)`, `copy(commands)`, `delete(commands)`, `define rule(commands)`, `replace(commands)`, `retrieve(commands)`.

CAVEATS

The command syntax, patterned after `chmod(1)`, is admittedly somewhat cryptic.

A facility like `umask(2)` will be added in the future.

User authentication is only conducted if the frontend process and backend server have been compiled with the `kerberos(5)` libraries. See `introduction(unix)`.

As shipped, the system does not have any installed ACLs.

An access control mode for defining trusted functions (analogous to the access control on defining rules) will be added after the (mis)features and interface of untrusted functions have stabilized.

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.

User system identifiers, as mentioned in a previous section of the manual, are currently UNIX user-id’s. This may change at some time in the future.

It is possible for users to change the server’s internal data structures from inside of trusted (fast path) C functions. Hence, among many other things, such functions can circumvent any system access controls. This is an inherent problem with trusted functions.

No POSTQUEL command is provided to clean up ACLs by removing entries (as opposed to removing the associated permissions). However, the built-in ACL functions provided make most administrative tasks fairly trivial. For example, to remove all ACL references to a user "mao" who is about to be fired, use:

```sql
replace pg_class (relacl = pg_class.relacl - 'mao'::aclitem)
```

Security should be implemented with a clever query modification or rule-based scheme.

01/23/93
NAME
  close — close a portal

SYNOPSIS
  close [ portal_name ]

DESCRIPTION
  Close frees the resources associated with a portal, portal_name. After this portal is closed, no subsequent
  operations are allowed on it. A portal should be closed when it is no longer needed. If portal_name is not
  specified, then the blank portal is closed.

EXAMPLE
  /*
   * close the portal FOO
   */
  close FOO

SEE ALSO
  fetch(commands), move(commands), retrieve(commands).
NAME
cluster — give storage clustering advice to POSTGRES

SYNOPSIS
cluster classname on attname [ using operator ]

DESCRIPTION
This command instructs POSTGRES to keep the class specified by classname approximately sorted on
attname using the specified operator to determine the sort order. The operator must be a binary operator
and both operands must be of type attname and the operator must produce a result of type boolean. If no
operator is specified, then "<" is used by default.

A class can be reclustered at any time on a different attname and/or with a different operator.

POSTGRES will try to keep the heap data structure which stores the instances of this class approximately in
sorted order. If the user specifies an operator which does not define a linear ordering, this command will
produce unpredictable orderings.

Also, if there is no index for the clustering attribute, then this command will have no effect.

EXAMPLE
/*
 * cluster employees in salary order
 */
cluster emp on salary

BUGS
Cluster has no effect in Version 4.2.
NAME
copy — copy data to or from a class from or to a UNIX file.

SYNOPSIS
copy [binary] [nonulls] classnamen
to/from "filename" stdin stdout

DESCRIPTION
Copy moves data between POSTGRES classes and standard UNIX files. The keyword binary changes the
behavior of field formatting, as described below. Classname is the name of an existing class. Filename is
the UNIX pathname of the file. In place of a filename, the keywords stdin and stdout can be used so that
input to copy can be written by a LIBPQ application and output from the copy command can be read by a
LIBPQ application. The binary keyword will force all data to be stored/read as binary objects rather than as
ASCII 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.

You must have read access on any class whose values are read by the copy command, and either write or
append access to a class to which values are being appended by the copy command.

FORMAT OF OUTPUT FILES

ASCII COPY FORMAT
When copy is used without the binary keyword, the file generated will have each instance on a line, with
each attribute separated by tabs (\t). Embedded tabs 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.

Note that on input to copy, backslashes are considered to be special control characters, and should be dou-
bled if you want to embed a backslash, i.e., the string "1\2\19\88" will be converted by copy to "1\2\19\88".
The actual format for each instance is

<attr1><tab><attr2><tab>...<tab><attrm><newline>

If copy is sending its output to standard output instead of a file, it will send a period (.) followed immedi-
ately by a newline, on a line by themselves, when it is done. Similarly, if copy is reading from standard
input, it will expect a period (.) followed by a newline, as the first two characters on a line, to denote end-
of-file. However, copy will terminate (followed by the backend itself) if a true EOF is encountered.

NULL attributes are handled simply as null strings, that is, consecutive tabs in the input file denote a
NULL attribute.

BINARY COPY 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 below description.

  uint32 total length (not including itself),
  uint32 number of null attributes
  [uint32 attribute number of first null attribute
   ...
  uint32 attribute number of nth null attribute],
  <data>

ALIGNMENT OF BINARY DATA
On Sun-3's, 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 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.

SEE ALSO
append(commands), create(commands), vacuum(commands), libpq.

BUGS
Files used as arguments to the copy command must reside on or be accessible to the the database server machine by being either on local disks or a networked file system.

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. The vacuum(commands) query should be used to clean up after a failed copy.

Because POSTGRES operates out of a different directory than the user's working directory at the time POSTGRES is invoked, 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 path-name should be used when specifying files to be copied.

Copy has virtually no error checking, and a malformed input file will likely cause the backend to crash. Humans should avoid using copy for input whenever possible.
NAME
create — create a new class

SYNOPSIS
create classname (attname-1 = type-1 (, attname-i = type-i))
  [key (attname-1 [using operator-1]
  , attname-i [using operator-i]])
  [inherits (classname-1 (, classname-i))]
  [archive = archive_mode]
  [store = "smgr_name"]
  [arch_store = "smgr_name"]

DESCRIPTION
Create will enter a new class into the current database. The class will be "owned" by the user issuing the
command. The name of the class is classname and the attributes are as specified in the list of attnames.
The ith attribute is created with the type specified by type-i. Each type may be a simple type, a complex
type (set) or an array type.

Each array attribute stores arrays that must have the same number of dimensions but may have different
sizes and array index bounds. An array of dimension n is specified by appending n pairs of square brackets:

att_name = type[1][1]...[1]

The optional key clause is used to specify that a field or a collection of fields is unique. If no key clause is
specified, POSTGRES will still give every instance a unique object-id (OID). This clause allows other fields
to be additional keys. The using part of the clause allows the user to specify what operator should be used
for the uniqueness test. For example, integers are all unique if "=" is used for the check, but not if "<>" is
used instead. If no operator is specified, "=" is used by default. Any specified operator must be a binary
operator returning a boolean. If there is no compatible index to allow the key clause to be rapidly checked,
POSTGRES defaults to not checking rather than performing an exhaustive search on each key update.

The optional inherits clause specifies a collection of class names from which this class automatically inher-
ts 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 class classname is automatically created as a type. Therefore, one or more instances from the
class are automatically a type and can be used in addattr(commands) or other create statements. See intro-
duction(commands) for a further discussion of this point.

The optional store and arch_store keywords may be used to specify a storage manager to use for the new
class. The released version of POSTGRES supports only "magnetic disk" as a storage manager name; the
research system at Berkeley provides additional storage managers. Store controls the location of current
data, and arch_store controls the location of historical data. Arch_store may only be specified if archive
is also specified. If either store or arch_store is not declared, it defaults to "magnetic disk".

The new class is created as a heap with no initial data. A class can have no more than 1600 domains (realis-
tically, this is limited by the fact that tuple sizes must be less than 8192 bytes), but this limit may be con-
figured lower at some sites. A class cannot have the same name as a system catalog class.

The archive keyword specifies whether historical data is to be saved or discarded. Arch_mode may be one of:
none No historical access is supported.
Historical access is allowed and optimized for light update activity.
Heavy Historical access is allowed and optimized for heavy update activity.

Arch_mode defaults to "none". Once the archive status is set, there is no way to change it. For details of the optimization, see [STON87].

EXAMPLES

/*
 * Create class emp with attributes name, sal and bdate
 */
create emp (name = char16, salary = float4, bdate = abstime)

/*
 * Create class permemp with pension information that
 * inherits all fields of emp
 */
create permemp (plan = char16) inherits (emp)

/*
 * Create class foo on magnetic disk and archive historical data
 */
create foo (bar = int4) archive = heavy
   store = "magnetic disk"

/*
 * Create class tictactoe to store noughts-and-crosses
 * boards as a 2-dimensional array
 */
create tictactoe (game = int4, board = char[][])

/*
 * Create a class newemp with a set attribute "manager". A
 * set (complex) attribute may be of the same type as the
 * relation being defined (as here) or of a different complex
 * type. The type must exist in the "pg_type" catalog or be
 * the one currently being defined.
 */
create newemp (name = text, manager = newemp)

SEE ALSO
destroy(commands).

BUGS
The key clause is not implemented in Version 4.2.

Optional specifications (i.e., inherits, archive and store) must be supplied in the order given above, if they are supplied at all.
NAME
createdb — create a new database

SYNOPSIS
createdb dbname

DESCRIPTION
Createdb creates a new POSTGRES database. The creator becomes the administrator of the new database.

SEE ALSO
createdb(unix), destroydb(commands), destroydb(unix), initdb(unix).

BUGS
This command should NOT be executed interactively. The createdb(unix) script should be used instead.
NAME
create version --- construct a version class

SYNOPSIS
create version classname1 from classname2 [abstime]

DESCRIPTION
This command creates a version class classname1 which is related to its parent class, classname2. Initially, classname1 has the same contents as classname2. As updates to classname1 occur, however, the content of classname1 diverges from classname2. On the other hand, any updates to classname2 show transparently through to classname1, unless the instance in question has already been updated in classname1.

If the optional abstime clause is specified, then the version is constructed relative to a snapshot of classname2 as of the time specified.

POSTGRES uses the query rewrite rule system to ensure that classname1 is differentially encoded relative to classname2. Moreover, classname1 is automatically constructed to have the same indexes as classname2. It is legal to cascade versions arbitrarily, so a tree of versions can ultimately result. The algorithms that control versions are explained in [ONG90].

EXAMPLE
/ *
 * create a version foobar from a snapshot of
 * barfoo as of January 17, 1990
 */
create version foobar from barfoo ['Jan 17 1990']

SEE ALSO
define view(commands), merge(commands), postquel(commands).

BUGS
Snapshots (i.e., the optional abstime clause) are not implemented in Version 4.2.
NAME

define aggregate — define a new aggregate

SYNOPSIS

define aggregate agg-name [as]
   ([sffunc1 = state-transition-function-1
      , basetype = data-type
      , stype1 = sffunc1-return-type]
    [, sffunc2 = state-transition-function-2
      , stype2 = sffunc2-return-type]
    [, finalfunc = final-function]
    [, initcond1 = initial-condition-1]
    [, initcond2 = initial-condition-2])

DESCRIPTION

An aggregate function can use up to three functions, two state transition functions, X1 and X2:
   X1( internal-state1, next-data_item ) —> next-internal-state1
   X2( internal-state2 ) —> next-internal-state2
and a final calculation function, F:
   F( internal-state1, internal-state2 ) —> aggregate-value
These functions are required to have the following properties:

The arguments to state-transition-function-1 must be (stype1,basetype), and its return value must be stype1.

The argument and return value of state-transition-function-2 must be stype2.

The arguments to the final-calculation-function must be (stype1,stype2), and its return value must be a POSTGRES base type (not necessarily the same as basetype).

The final-calculation-function should be specified if and only if both state-transition functions are specified.

Note that it is possible to specify aggregate functions that have varying combinations of state and final functions. For example, the “count” aggregate requires sffunc2 (an incrementing function) but not sffunc1 or finalfunc, whereas the “sum” aggregate requires sffunc1 (an addition function) but not sffunc2 or finalfunc and the “average” aggregate requires both of the above state functions as well as a finalfunc (a division function) to produce its answer. In any case, at least one state function must be defined, and any sffunc2 must have a corresponding initcond2.

Aggregates also require two initial conditions, one for each transition function. These are specified and stored in the database as fields of type text.

EXAMPLE

This avg aggregate consists of two state transition functions, a addition function and a incrementing function. These modify the internal state of the aggregate through a running sum and and the number of values seen so far. It accepts a new employee salary, increments the count, and adds the new salary to produce the next state. The state transition functions must be passed correct initialization values. The final calculation then divides the sum by the count to produce the final answer.

```c
/*
 * Define an aggregate for int4 average
 */
define aggregate avg (sffunc1 = int4add, basetype = int4,
   stype1 = int4, sffunc2 = int4inc, stype2 = int4,
   finalfunc = int4div, initcond1 = "0", initcond2 = "0")
```

01/23/93 75
DEFINE AGGREGATE(COMMANDS)

SEE ALSO

define function(commands), remove aggregate(commands).
NAME
define function — define a new function

SYNOPSIS
define function function_name (  
  language = ["c" | "postquel"],  
  returntype = type-r  
  [, iscachable ]  
  [, trusted = ["i" | "f"] ]  
  [, percall_cpu = "costly[!]"]  
  [, perbyte_cpu = "costly[!]"]  
  [, outin_ratio = percentage ]  
  [, byte_pct = percentage ]  
  )  
  arg is ([ type-l [, type-n ]])  
  as {"full/path/to/objectfile" | "list-of-postque1-queries"}

DESCRIPTION
With this command, a POSTGRES user can register a function with POSTGRES. Subsequently, this user is  
treated as the owner of the function.

When defining a function with arguments, the input data types, type-1, type-2, ..., type-n, and the return data  
type, type-r must be specified, along with the language, which may be "c" or "postquel". (The arg is  
clause may be left out if the function has no arguments, or alternatively the argument list may be left  
empty.) The input types may be base or complex types, or any. Any indicates that the function accepts  
arguments of any type, or takes an invalid POSTQUEL type such as (char *). The output type may be speci-  
fied as a base type, complex type, setof <type>, or any. The setof modifier indicates that the function will  
return a set of items, rather than a single item. The as clause of the command is treated differently for C  
and POSTQUEL functions, as explained below.

C FUNCTIONS
Functions written in C can be defined to POSTGRES, which will dynamically load them into its address  
space. The loading happens either using load(commands) or automatically the first time the function is  
necessary for execution. Repeated execution of a function will cause negligible additional overhead, as the  
function will remain in a main memory cache.

The iscachable flag indicates to the system that the return value of the function can be associatively cached.

The trusted flag specifies that the function can run inside the POSTGRES server's address space with the  
user-id of the POSTGRES super-user. If this flag is not specified, the function will be run in a separate pro-  
cess.

The percall_cpu, perbyte_cpu, outin_ratio, and byte_pct flags are provided for C functions to give a  
rough estimate of the function's running time, allowing the query optimizer to postpone applying expensive  
functions used in a query's where clause. The percall_cpu flag captures the overhead of the function's  
invocation (regardless of input size), while the perbyte_cpu flag captures the sensitivity of the function's  
running time to the size of its inputs. The magnitude of these two parameters is determined by the number  
of exclamation points appearing after the word costly: specifically, each exclamation point can be thought  
of as another order of magnitude in cost, i.e.,

\[
\text{cost} = 10^{\text{number-of-exclamation-points}}
\]

The default value for percall_cpu and perbyte_cpu is 0. Examples of reasonable cost values may be  
found in the system catalog "pg_proc": most simple functions on base types have costs of 0.

03/12/94
The output ratio is provided for functions which return variable-length types, such as text or bytea. It should be set to the size of the function's output as a percentage of the size of the input. For example, a function which compresses its operands by 2 should have output_ratio = 50. The default value is 100.

The byte_pct flag should be set to the percentage of the bytes of the arguments that actually need to be examined in order to compute the function. This flag is particularly useful for functions which generally take a large object as an argument, but only examine a small fixed portion of the object. The default value is 100.

Writing C Functions

The body of a C function following as should be the FULL PATH of the object code (.o file) for the function, bracketed by quotation marks. (POSTGRES will not compile a function automatically — it must be compiled before it is used in a define function command.)

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

```c
.../src/backend/utils/builtins.h
```

is included as a header file. This can be achieved by having

```c
#include <utils/builtins.h>
```

at the top of the C source file and by compiling all C files with the following include options:

```c
-I.../src/backend
-I.../src/backend/port/<portname>
-I.../src/backend/obj
```

before any "c" programs in the cc command line, e.g.:

```c
cc -I.../src/backend \
   -I.../src/backend/port/<portname> \
   -I.../src/backend/obj \
   -c progranme.c
```

where "..." is the path to the installed POSTGRES source tree and "<portname>" is the name of the port for which the source tree has been built.

The convention for passing arguments to and from the user's C functions is to use pass-by-value for data types that are 32 bits (4 bytes) or smaller, and pass-by-reference for data types that require more than 32 bits.

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

```c
.../src/backend
```

directory) that the equivalent C type is defined. However, if you include "utils/builtins.h", these files will automatically be included.
Complex arguments to C functions are passed into the C function as a special C type, TUPLE, defined in .../src/libpq/libpq-fe.h.

Given a variable \( t \) of this type, the C function may extract attributes from the function using the function call:

\[
\text{GetAttributeByName}(t, \text{"fieldname"}, \text{&isnull})
\]

where \( \text{isnull} \) is a pointer to a bool, which the function sets to \( \text{true} \) if the field is null. The result of this function should be cast appropriately as shown in the examples below.

### Compiling Dynamically-Loaded C Functions

Different operating systems require different procedures for compiling C source files so that POSTGRES can load them dynamically. This section discusses the required compiler and loader options on each system.

Under Ultrix, all object files that POSTGRES is expected to load dynamically must be compiled using \( /bin/cc \) with the “-G 0” option turned on. The object file name in the \( \text{as} \) clause should end in “.o”.

Under HP-UX, DEC OSF/1, AIX and SunOS 4, all object files must be turned into shared libraries using the operating system’s native object file loader, \( ld(1) \).

Under HP-UX, an object file must be compiled using the native HP-UX C compiler, \( /bin/cc \), with both the “-z” and “-u” flags turned on. The first flag turns the object file into “position-independent code” (PIC);
DEFINE FUNCTION(COMMANDS)

the second flag removes some alignment restrictions that the PA-RISC architecture normally enforces. The
object file must then be turned into a shared library using the HP-UX loader, /bin/ld. The command lines to
compile a C source file, “foo.c”, look like:

cc <other flags> +z +u -c foo.c
ld <other flags> -b -o foo.sl foo.o

The object file name in the as clause should end in “.sl”.
An extra step is required under versions of HP-UX prior to 9.00. If the POSTGRES header file

/tmp/c.h

is not included in the source file, then the following line must also be added at the top of every source file:

#pragma HP_ALIGN HPUX_NATURAL_S500

However, this line must not appear in programs compiled under HP-UX 9.00 or later.
Under DEC OSF/1, an object file must be compiled and then turned into a shared library using the OSF/1
loader, /bin/ld. In this case, the command lines look like:

cc <other flags> -c foo.c
ld <other flags> -shared -expect_unresolved '*' -o foo.so foo.o

The object file name in the as clause should end in “.so”.
Under SunOS 4, an object file must be compiled and then turned into a shared library using the SunOS 4
loader, /bin/ld. The command lines look like:

cc <other flags> -PIC -c foo.c
ld <other flags> -dc -dp -Bdynamic -o foo.so foo.o

The object file name in the as clause should end in “.so”.
Under AIX, object files are compiled normally but building the shared library requires a couple of steps.
First, create the object file:

cc <other flags> -c foo.c

You must then create a symbol “exports” file for the object file:

mkldeexport foo.o ‘pwd’ > foo.exp

Finally, you can create the shared library:

ld <other flags> -H512 -T512 -o foo.so -e _nostart -bI:.../lib/postgres.exp -bE:foo.exp foo.o -lM -lc 2>/dev/null

You should look at the POSTGRES User Manual for an explanation of this procedure.

03/12/94
POSTQUEL FUNCTIONS

POSTQUEL functions execute an arbitrary list of POSTQUEL queries, returning the results of the last query in the list. POSTQUEL functions in general return sets. If their return type is not specified as a setof, then an arbitrary element of the last query’s result will be returned. The expensive function parameters per-call_cpu, perbyte_cpu, outin_ratio, and byte_pct are not used for POSTQUEL functions; their costs are determined dynamically by the query optimizer.

The body of a POSTQUEL 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 (i.e. \").

Arguments to the POSTQUEL 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 may be used to access attributes of the argument (e.g. “$1.emp”), or to invoke functions via a nested-dot syntax.

EXAMPLES: C Functions

The following command defines a C function, overpaid, of two basetype arguments.

```c
define function overpaid
  (language = "c", returntype = bool)
  arg is (float8, int4)
  as "/usr/postgres/src/adt/overpaid.o"
```

The C file "overpaid.c" might look something like:

```c
#include <utils/builtins.h>

bool overpaid(salary, age)
  float8 *salary;
  int4 age;
{
  if (*salary > 200000.00)
    return(TRUE);
  if ((age < 30) && (*salary > 100000.00))
    return(TRUE);
  return(FALSE);
}
```

The overpaid function can be used in a query, e.g:

```sql
retrieve (EMP.name)
  where overpaid(EMP.salary, EMP.age)
```

One can also write this as a function of a single argument of type EMP:

```c
define function overpaid_2
  (language = "c", returntype = bool)
  arg is (EMP)
  as "/usr/postgres/src/adt/overpaid_2.o"
```

The following query is now accepted:
DEFINE FUNCTION(COMMANDS)

retrieve (EMP.name) where overpaid_2(EMP)

In this case, in the body of the overpaid_2 function, the fields in the EMP record must be extracted. The C
file "overpaid_2.c" might look something like:

#include <utils/builtins.h>
#include <tmp/libpq-fe.h>

bool overpaid_2(t)

TUPLE t;

{
  float8 *salary;
  int4 age;
  bool salnull, agenull;

  salary = (float8 *)GetAttributeByName(t, "salary", &salnull);
  age = (int4)GetAttributeByName(t, "age", &agenull);
  if (!salnull && *salary > 200000.00)
    return(TRUE);
  if (!agenull && (age<30) && (*salary > 100000.00))
    return(TRUE);
  return(FALSE)
}

EXAMPLES: POSTQUEL Functions

To illustrate a simple POSTQUEL function, consider the following, which might be used to debit a bank
account:

define function TPl

(language = "postquel", returntype = int4)

arg is (int4, floats)

as "replace BANK (balance = BANK.balance - $2)

  where BANK.accountno = $1

  retrieve(x = 1)"

A user could execute this function to debit account 17 by $100.00 as follows:

retrieve (x = TPl( 17,100.0))

The following more interesting examples take a single argument of type EMP, and retrieve multiple results:

define function hobbies

(language = "postquel", returntype = setof HOBBIES)

arg is (EMP)

as "retrieve (HOBBIES.all)

  where $1.name = HOBBIES.person"

define function children

(language = "postquel", returntype = setof KIDS)

03/12/94

82
DEFINE FUNCTION(COMMANDS)

arg is (EMP)
as "retrieve (KIDS.all)
where $1.name = KIDS.dad
or $1.name = KIDS.mom"

Then the following query retrieves overpaid employees, their hobbies, and their children:

retrieve (name=name(EMP), hobby=name(hobbies(EMP)),
        kid=name(children(EMP)))
where overpaid_2(EMP)

Note that attributes can be projected using function syntax (e.g. name(EMP)), as well as the traditional dot syntax (e.g. EMP.name).

An equivalent expression of the previous query is:

retrieve (EMP.name, hobby=EMP.hobbies.name,
        kid=EMP.children.name)
where overpaid_2(EMP)

This "nested dot" notation for functions can be used to cascade functions of single arguments. Note that the function after a dot must have only one argument, of the type returned by the function before the dot.

POSTGRES flattens the target list of the queries above. That is, it produces the cross-product of the hobbies and the children of the employees. For example, given the schema:

create BANK (accountno = int4, balance = float8)
append BANK (accountno = 17,
            balance = "10000.00"::float8)
create EMP (name = char16, salary = float8,
        dept = char16, age = int4)
create HOBBIES (name = char16, person = char16)
create KIDS (name = char16, dad = char16, mom = char16)
append EMP (name = "joey", salary = "100000.01"::float8,
            dept = "toy", age = 24)
append EMP (name = "jeff", salary = "100000.01"::float8,
            dept = "shoe", age = 23)
append EMP (name = "wei", salary = "100000"::float8,
            dept = "tv", age = 30)
append EMP (name = "mike", salary = "500000"::float8,
            dept = "appliances", age = 30)
append HOBBIES (name = "biking", person = "jeff")
append HOBBIES (name = "jamming", person = "joey")
append HOBBIES (name = "basketball", person = "wei")
append HOBBIES (name = "swimming", person = "mike")
append HOBBIES (name = "philately", person = "mike")
append KIDS (name = "matthew", dad = "mike",
            mom = "teresa")
append KIDS (name = "calvin", dad = "mike",
            mom = "teresa")

The query above returns

03/12/94
NAME

MAY BE GIVEN AS AN ARGUMENT TO A POSTQUEL FUNCTION.

BUGS

THE ISCACHABLE FLAG DOES NOT DO ANYTHING IN VERSION 4.2.

UNTRUSTED FUNCTIONS CANNOT MAKE ANY FUNCTION CALLS USING ACCESS METHODS OR BUILT-IN FUNCTIONS THAT HAVE NOT BEEN LOADED INTO THE UNTRUSTED-FUNCTION PROCESS.

UNTRUSTED FUNCTIONS MUST BE EXPLICITLY DESIGNATED AS SUCH IN A SEPARATE QUERY, E.G.:

    replace pg_proc (proistrusted = "f"::bool)
    where pg_proc.proname = "mynewfunction"

03/12/94  84
C functions cannot return a set of values.
NAME
define index — construct a secondary index

SYNOPSIS
define [archive] index index-name
   on classname using am-name
   ( attname type_class )
   [where qual]

define [archive] index index-name
   on classname using am-name
   ( funcname (attname-1 {, attname-i} ) type_class )

DESCRIPTION
This command constructs an index called index-name. If the archive keyword is absent, the classname class is indexed. When archive is present, an index is created on the archive class associated with the classname class.

Am-name is the name of the access method which is used for the index.

In the first syntax shown above, the key field for the index is specified as an attribute name and an associated operator class. An operator class is used to specify the operators to be used for a particular index. 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.

If a qual is given, the index will be a partial index, which will index only those instances in classname for which the predicate specified by qual is true. Note that the predicate may only refer to attributes of the indexed class, classname. POSTGRES may use a partial index as an access path only for queries that include a restriction that implies that the predicate is true. For example, if the index predicate is

emp.age < 30

then the index can be used for a query with the restriction

where emp.age < 25

but not for a query with the restriction

where emp.age < 40

and so forth. Although partial indexes cannot be used to satisfy as wide a range of queries as complete indexes, they can be constructed more quickly and extended incrementally (see extend index(commands)).

In the second syntax shown above, an index can be defined on the result of a user-defined function funcname applied to one or more attributes of a single class. These functional indices are primarily useful in two situations. First, functional indices can be used to simulate multi-key indices. That is, the user can define a new base type (a simple combination of, say, “oid” and “int2”) and the associated functions and operators on this new type such that the access method can use it. Once this has been done, the standard techniques for interfacing new types to access methods (described in the POSTGRES user manual) can be applied. Second, functional indices can be used to obtain fast access to data based on operators that would normally require some transformation to be applied to the base data. For example, say you have an attribute in class “myclass” called “pt” that consists of a 2D point type. Now, suppose that you would like to index this attribute but you only have index operator classes for 2D polygon types. You can define an index on the point attribute using a function that you write (call it “point_to_polygon”) and your existing
polygon operator class; after that, queries using existing polygon operators that reference "point_to_polygon(myclass.pt)" on one side will use the precomputed polygons stored in the functional index instead of computing a polygon for each and every instance in "myclass" and then comparing it to the value on the other side of the operator. Obviously, the decision to build a functional index represents a tradeoff between space (for the index) and execution time.

POSTGRES Version 4.2 provides btree, rtree and hash access methods for secondary indices. The btree access method is an implementation of the 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 operator classes defined on btrees are

```
int2_ops  char2_ops  oidint2_ops
int4_ops  char4_ops  oidint4_ops
int24_ops char8_ops  oidchar16_ops
int42_ops char16_ops
float4_ops oid_ops
float8_ops text_ops
char_ops  abstime_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 operator classes `oidint2_ops`, `oidint4_ops`, and `oidchar16_ops` represent the use of functional indices to simulate multi-key indices.

The POSTGRES query optimizer will consider using btree indices in a scan whenever an indexed attribute is involved in a comparison using one of

```
<  <=  =  >=  >
```

The operator classes defined on rtrees are

```
box_ops
bigbox_ops
poly_ops
```

Both box classes support indices on the "box" datatype in POSTGRES. 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 `poly_ops` operator class supports rtree indices on "polygon" data.

The POSTGRES query optimizer will consider using an rtree index whenever an indexed attribute is involved in a comparison using one of

```
<<  &<  &>  >>  @  "="  &
```

The operator classes defined on the hash access method are

```
03/12/94
```
The POSTGRES query optimizer will consider using a hash index whenever an indexed attribute is involved in a comparison using the = operator.

EXAMPLES

/*
 * Create a btree index on the emp class using the age attribute.
 */
define index empindex on emp using btree (age int4_ops)

/*
 * Create a btree index on employee name.
 */
define index empname
    on emp using btree (name char16_ops)

/*
 * Create an rtree index on the bounding rectangle of cities.
 */
define index cityrect
    on city using rtree (boundbox box_ops)

/*
 * Create a rtree index on a point attribute such that we can efficiently use box operators on the result of the conversion function. Such a qualification might look like "where point2box(points.pointloc) = boxes.box".
 */
define index pointloc
    on points using rtree (point2box(location) box_ops)

/*
 * Create a partial btree index on employee salaries for employees over age 50
 */
define index empsal
    on emp using btree (salary int4_ops) where emp.age > 49
Note: if the partial-index predicate refers to an attribute of a discrete-valued type (such as integers), it is slightly preferable to express the predicate as, e.g., "emp.age > 49" rather than as "emp.age >= 50", because even though both indexes would, in theory, be equally usable, POSTGRES would only be able to use a partial index with the former predicate in the event of a query that had the exact restriction "emp.age > 49".

**BUGS**

Archive indices are not supported in Version 4.2.

There should be an access method designer's guide.

Indices may only be defined on a single key. This can be hacked around by defining special types and using the POSTGRES support for indices on functional values of attributes.

The only kind of partial index predicates POSTGRES Version 4.2 understands are those made up of boolean combinations of simple clauses of the form

```
ATTR OP CONST
```

where ATTR is a single attribute of the indexed class, and OP is an operator in a btree operator class defined on the types of ATTR and CONST. If some other form of predicate is specified, Version 4.2 will never use the resulting partial index.
NAME

define operator — define a new user operator

SYNOPSIS

define operator operator_name

(('arg1 = type-1')

('arg2 = type-2')

procedure = func_name

precedence = number

associativity = (left | right | none | any)

commutator = com_op

gle_op = neg_op

restrict = res_proc

hashes

join = join_proc

sort = sort_op1 {, sort_op2 }
)

DESCRIPTION

This command defines a new user operator, operator_name. The user who defines an operator becomes its owner.

The operator_name is a sequence of up to sixteen punctuation characters. The following characters are valid for single-character operator names:

~ ! @ # $ % ^ & * ?

If the operator name is more than one character long, it may consist of any combination of the above characters or the following additional characters:

| $ : + - * / < > =

At least one of arg1 and arg2 must be defined. For binary operators, both should be defined. For right unary operators, only arg1 should be defined, while for left unary operators only arg2 should be defined.

The name of the operator, operator_name, can be composed of symbols only. Also, the func_name procedure must have been previously defined using define function(commands) and must have one or two arguments. The types of the arguments for the operator and the type of the answer are as defined by the function. Precedence refers to the order that multiple instances of the same operator are evaluated. The next several fields are primarily for the use of the query optimizer.

The associativity value is used to indicate how an expression containing this operator should be evaluated when precedence and explicit grouping are insufficient to produce a complete order of evaluation. Left and right indicate that expressions containing the operator are to be evaluated from left to right or from right to left, respectively. None means that it is an error for this operator to be used without explicit grouping when there is ambiguity. And any, the default, indicates that the optimizer may choose to evaluate an expression which contains this operator arbitrarily.

The commutator operator is present so that POSTGRES can reverse the order of the operands if it wishes. For example, the operator area-less-than, >>, would have a commutator operator, area-greater-than, <<<. Suppose that an operator, area-equal, ===, exists, as well as an area not equal, !=. Hence, the query optimizer could freely convert:

"0, 0, 1, 1": box >>> MYBOXES.description

03/12/94 90
to

MYBOXES.description <<< "0,0,1,1*::box"

This allows the execution code to always use the latter representation and simplifies the query optimizer somewhat.

The negator operator allows the query optimizer to convert

not MYBOXES.description == "0,0,1,1*::box"

to

MYBOXES.description != "0,0,1,1*::box"

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 current (new) 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.

The next two specifications 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 annotated by an operator that returns a boolean) by iterative substitution [WONG76]. In addition, POSTGRES is planning on implementing a hash-join algorithm along the lines of [SHAP86]; however, it must know whether this strategy is applicable. For example, a hash-join algorithm is usable for a clause of the form:

MYBOXES.description == MYBOXES2.description

but not for a clause of the form:

MYBOXES.description <<< MYBOXES2.description.

The hashes flag gives the needed information to the query optimizer concerning whether a hash join strategy is usable for the operator in question.

Similarly, the two sort operators indicate to the query optimizer whether merge-sort is a usable join strategy and what operators should be used to sort the two operand classes. For the == clause above, the optimizer must sort both relations using the operator, <<<. On the other hand, merge-sort is not usable with the clause:

MYBOXES.description <<< MYBOXES2.description

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 last two pieces of the specification are present so the query optimizer can estimate result sizes. If a clause of the form:
MYBOXES.description \lll *0,0,1,1*:::box

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 define function(commands)) which accepts one argument of the correct data type 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 desired expected number of instances.

Similarly, when the operands of the operator both contain instance variables, the query optimizer must esti-
mate 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 desired expected result
size.

The difference between the function

\texttt{my\_procedure\_1 (MYBOXES.description, *0,0,1,1*:::box)}

and the operator

\texttt{MYBOXES.description === *0,0,1,1*:::box}

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.

EXAMPLE

/*
 * The following command defines a new operator,
 * area-equality, for the BOX data type.
 */
define operator === ( 
    arg1 = box, 
    arg2 = box, 
    procedure = area_equal_procedure, 
    precedence = 30, 
    associativity = left, 
    commutator = ===, 
    negator = !==, 
    restrict = area_restriction_procedure, 
    hashes, 
    join = area-join-procedure, 
    sort = \lll, \lll)

SEE ALSO

define function(commands), remove operator(commands).
BUGS

Operator names cannot be composed of alphabetic characters in Version 4.2.

Operator precedence is not implemented in Version 4.2.

If an operator is defined before its commuting operator has been defined (a case specifically warned against above), a dummy operator with invalid fields will be placed in the system catalogs. This may interfere with the definition of later operators.
NAME
define rule — define a new rule

SYNOPSIS
define [instance | rewrite] rule rule_name
[as exception to rule_name_2]
is on event
to object [[from clause] where clause]
do [instead]
[action | nothing | [actions...]]

DESCRIPTION
Define rule is used to define a new rule. There are two implementations of the rules system, one based on
query rewrite and the other based on instance-level processing. In general, the instance-level system is
more efficient if there are many rules on a single class, each covering a small subset of the instances. The
rewrite system is more efficient if large scope rules are being defined. The user can optionally choose which
rule system to use by specifying rewrite or instance in the command. If the user does not specify which
system to use, POSTGRES defaults to using the instance-level system. In the long run POSTGRES will auto-
matically decide which rules system to use and the possibility of user selection will be removed.

Here, event is one of retrieve, replace, delete or append. Object is either:

a class name

or
class.column

The from clause, the where clause, and the action are respectively normal POSTQUEL from clauses, where
clauses and collections of POSTQUEL commands with the following change:

new or current can appear instead of an instance variable whenever an instance
variable is permissible in POSTQUEL.

The semantics of a rule is that at the time an individual instance is accessed, updated, inserted or deleted,
there is a current instance (for retrieves, replaces and deletes) and a new instance (for replaces and
appends). If the event specified in the on clause and the condition specified in the where clause are true for
the current instance, then the action part of the rule is executed. First, however, values from fields in the
current instance and/or the new instance are substituted for:

current.attribute-name

new.attribute-name

The action part of the rule executes with same command and transaction identifier as the user command
that caused activation.

A note of caution about POSTQUEL rules is in order. If the same class name or instance variable appears in
the event, where clause and the action parts of a rule, they are all considered different tuple variables. More
accurately, new and current are the only tuple variables that are shared between these clauses. For
example, the following two rules have the same semantics:

on replace to EMP.salary where EMP.name = "Joe"
do replace EMP ( ... ) where ...

on replace to EMP-1.salary where EMP-2.name = "Joe"
do replace EMP-3 ( ... ) 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.
When choosing between the rewrite and instance rule systems for a particular rule application, remember that in the rewrite system current refers to a relation and some qualifiers whereas in the instance system it refers to an instance (tuple).

It is very important to note that the rewrite rule system will neither detect nor process circular rules. For example, though each of the following two rule definitions are accepted by POSTGRES, the retrieve command will cause POSTGRES to crash:

```sql
/*
 * Example of a circular rewrite rule combination.
 */
define rewrite rule bad_rule_combination_1 is
  on retrieve to EMP
  do instead retrieve to TOYEMP

define rewrite rule bad_rule_combination_2 is
  on retrieve to TOYEMP
  do instead retrieve to EMP

/*
 * This attempt to retrieve from EMP will cause POSTGRES to crash.
 */
retrieve (EMP.all)
```

You must have rule definition access to a class in order to define a rule on it (see change acl(commands).

**EXAMPLES**

```sql
/*
 * Make Sam get the same salary adjustment as Joe
 */
define rule example_1 is
  on replace to EMP.salary where current.name = "Joe"
  do replace EMP (salary = new.salary)
    where EMP.name = "Sam"

At the time Joe receives a salary adjustment, the event will become true and Joe’s current 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
 */
define rule example_2 is
  on retrieve to EMP.salary
    where current.name = "Bill"
  do instead
    retrieve (EMP.salary) where EMP.name = "Joe"

/*
 * Deny Joe access to the salary of employees in the shoe
```
/*
 * department. (pg_username() returns the name of the current user)
 */
define rule example_3 is
  on retrieve to EMP.salary
  where current.dept = "shoe"
       and pg_username() = "Joe"
  do instead nothing

/*
 * Create a view of the employees working in the toy department.
 */
cREATE TOYEMP(name = char16, salary = int4)
define rule example_4 is
  on retrieve to TOYEMP
  do instead retrieve (EMP.name, EMP.salary)
       where EMP.dept = "toy"

/*
 * All new employees must make 5,000 or less
 */
define rule example_5 is
  on append to EMP where new.salary > 5000
  do replace new(salary = 5000)

SEE ALSO
  postquel(commands), remove rule(commands), define view(commands).

BUGS
  Exceptions are not implemented in Version 4.2.
  The object in a POSTQUEL 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 where clause cannot have a from clause.
  Only one POSTQUEL command can be specified in the action part of a tuple rule and it can only be a
  replace, append, retrieve or delete command.
  The rewrite rule system does support multiple action rules as long as event is not retrieve.
  The query rewrite rule system now supports most rule semantics, and closely parallels the tuple system. It
  also attempts to avoid odd semantics by running instead rules before non-instead rules.
  Both rule systems store 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).
NAME
define type — define a new base data type

SYNOPSIS
define type typename (internallength = (number | variable),
   [ externallength = (number | variable), ]
   input = input_function,
   output = output_function
   [, element = typename]
   [, delimiter = <character>]
   [, default = "string"]
   [, send = send_function]
   [, receive = receive_function]
   [, passedbyvalue))

DESCRIPTION
Define 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.

Define type requires the registration of two functions (using define function(commands)) 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 **any**.

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 (","), character.

A default value is optionally available in case a user wants some specific bit pattern to mean "data not pre-
sent."

The optional functions send function and receive function are used when the application program request-
ing POSTGRES services resides on a different machine. In this case, the machine on which POSTGRES runs
may use a different format for the data type than 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 rele-
vant 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 passedbyvalue flag indicates that operators and functions which use this data type should be
passed an argument by value rather than by reference. Note that only types whose internal representation is
at most four bytes may be passed by value.
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 Section 7, the large object interface. The length of all large object types is always variable, meaning the internallength for large objects is always -1.

EXAMPLES
*/
* This command creates the box data type and then uses the
type in a class definition
*/
define type box (internallength = 8,
    input = my_procedure_1, output = my_procedure_2)
create MYBOXES (id = int4, description = box)

/
* This command creates a variable length array type with
* integer elements.
*/
define type int4array
    (input = array_in, output = array_out,
    internallength = variable, element = int4)
create MYARRAYS (id = int4, numbers = int4array)

/
* This command creates a large object type and uses it in
* a class definition.
*/
define type bigobj
    (input = lo_filein, output = lo_fileout,
    internallength = variable)
create BIG_OBJS (id = int4, obj = bigobj)

RESTRICTIONS
Type names cannot begin with the underscore character (“_”) and can only be 15 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.
DEFINE TYPE (COMMANDS)

SEE ALSO

define function(commands), define operator(commands), remove type(commands), introduction(large objects).
NAME
define view — construct a virtual class

SYNOPSIS
define view view_name
   ([ dom_name_1 =] expression_1
     , [ dom_name_i =] expression_i )
   [ from from_list ]
   [ where qual ]

DESCRIPTION
Define view will define a view of a class. This view is not physically materialized; instead the rule system
is used to support view processing as in [STON90]. Specifically, a query rewrite retrieve rule is automati-
cally generated to support retrieve operations on views. Then, the user can add as many update rules as he
wishes to specify the processing of update operations to views. See [STON90] for a detailed discussion of
this point.

EXAMPLE
/*
 * define a view consisting of toy department employees
 */
define view toyemp (e.name)
   from e in emp
   where e.dept = *toy*

/*
 * Specify deletion semantics for toyemp
 */
define rewrite rule example1 is
   on delete to toyemp
       then do instead delete emp where emp.OID = current.OID

SEE ALSO
create(commands), define rule(commands), postquel(commands).
NAME
delete — delete instances from a class

SYNOPSIS
delete instance_variable [ from from_list ] [ where qual ]

DESCRIPTION
Delete removes instances which satisfy the qualification, qual, from the class specified by instance_variable. Instance_variable is either a class name or a variable assigned by from_list. If the qualification is absent, the effect is to delete all instances in the class. The result is a valid, but empty class.

You must have write access to the class in order to modify it, as well as read access to any class whose values are read in the qualification (see change aid(commands).

EXAMPLE
/*
 * Remove all employees who make over $30,000
 */
delete emp where emp.sal > 30000

/
 * Clear the hobbies class
 */
delete hobbies

SEE ALSO
destroy(commands).
NAME
  destroy — destroy existing classes
SYNOPSIS
  destroy classname-1 (, classname-i )
DESCRIPTION
  Destroy removes classes from the data base. Only its owner may destroy a class. A class may be emptied
  of instances, but not destroyed, by using delete(commands).

  If a class being destroyed has secondary indices on it, then they will be removed first. The removal of just a
  secondary index will not affect the indexed class.

  This command may be used to destroy a version class which is not a parent of some other version.
  Destroying a class which is a parent of a version class is disallowed. Instead, merge(commands) should be
  used. Moreover, destroying a class whose fields are inherited by other classes is similarly disallowed. An
  inheritance hierarchy must be destroyed from leaf level to root level.

  The destruction of classes is not reversible. Thus, a destroyed class will not be recovered if a transaction
  which destroys this class fails to commit. In addition, historical access to instances in a destroyed class is
  not possible.

EXAMPLE
  /*
   * Destroy the emp class
   */
  destroy emp

  /*
   * Destroy the emp and parts classes
   */
  destroy emp, parts

SEE ALSO
  delete(commands), merge(commands), remove index(commands).
NAME

destroydb — destroy an existing database

SYNOPSIS

destroydb dbname

DESCRIPTION

Destroydb removes the catalog entries for an existing database and deletes the directory containing the
data. It can only be executed by the database administrator (see createdb(commands) for details).

SEE ALSO

createdb(commands), destroydb(unix).

BUGS

This query should NOT be executed interactively. The destroydb(unix) script should be used instead.
NAME
   end — commit the current transaction

SYNOPSIS
   end

DESCRIPTION
   This command commits the current transaction. All changes made by the transaction become visible to
   others and are guaranteed to be durable if a crash occurs.

SEE ALSO
   abort(commands), begin(commands).
NAME
extend index — extend a partial secondary index

SYNOPSIS
extend index index-name [where qual]

DESCRIPTION
This command extends the existing partial index called index-name.

If a qual is given, the index will be extended to cover all instances that satisfy the predicate specified by qual (in addition to the instances the index already covers). If no qual is given, the index will be extended to be a complete index. Note that the predicate may only refer to attributes of the class on which the specified partial index was defined (see define index(commands)).

EXAMPLE

/*
 * Extend a partial index on employee salaries to include
 * all employees over 40
 */
extend index empsal where emp.age > 39

SEE ALSO
define index(commands), remove index(commands).
NAME
fetch — fetch instance(s) from a portal

SYNOPSIS
fetch [ (forward | backward) ] [ (number | all) ] [ in portal_name]

DESCRIPTION
Fetch allows a user to retrieve instances from a portal named portal_name. The number of instances
retrieved is specified by number. If the number of instances remaining in the portal is less than number,
then only those available are fetched. Substituting the keyword all in place of a number will cause all
remaining instances in the portal to be retrieved. Instances may be fetched in both forward and backward
directions. The default direction is forward.

Updating data in a portal is not supported by POSTGRES, because mapping portal updates back to base
classes is impossible in general as with view updates. Consequently, users must issue explicit replace com-
mands to update data.

Portals may only be used inside of transaction blocks marked by begin(commands) and end(commands)
because the data that they store spans multiple user queries.

EXAMPLE
/*
 * set up and use a portal
 */
begin \g
  retrieve portal myportal (pg_user.all) \g
  fetch 2 in myportal \g
  fetch all in myportal \g
  close myportal \g
end \g

/*
 * Fetch all the instances available in the portal FOO
 */
fetch all in FOO

/*
 * Fetch 5 instances backward in the portal FOO
 */
fetch backward 5 in FOO

SEE ALSO
begin(commands), end(commands), close(commands), move(commands), retrieve(commands).

BUGS
Currently, the smallest transaction in POSTGRES is a single POSTQUEL command. It should be possible for
a single fetch to be a transaction.
NAME
listen — listen for notification on a relation

SYNOPSIS
listen class name

DESCRIPTION
listen is used to register the current backend as a listener on the relation class name. When the command notify class name is called either from within a rule or at the query level, the frontend applications corresponding to the listening backends are notified. When the backend process exits, this registration is cleared.

This event notification is performed through the LIBPQ protocol and frontend application interface. The application program must explicitly poll a LIBPQ global variable, PQAsyncNotifyWaiting, and call the routine PQnotifies in order to find out the name of the class to which a given notification corresponds. If this code is not included in the application, the event notification will be queued and never be processed.

SEE ALSO
define rule(commands), notify(commands), retrieve(commands), libpq.

BUGS
There is no way to un-listen except to drop the connection (i.e., restart the backend server).

The monitor(unix) command does not poll for asynchronous events.
NAME
load — dynamically load an object file

SYNOPSIS
load "filename"

DESCRIPTION
Load loads an object (or ".o") file into POSTGRES's address space. Once a file is loaded, all functions in
that file can be accessed. This function is used in support of ADT's.

If a file is not loaded using the load command, the file will be loaded automatically the first time the func-
tion is called by POSTGRES. Load can also be used to reload an object file if it has been edited and recom-
piled. Only objects created from C language files are supported at this time.

EXAMPLE
/*
 * Load the file /usr/postgres/demo/circle.o
 */
load "/usr/postgres/demo/circle.o"

CAVEATS
Functions in loaded object files should not call functions in other object files loaded through the load com-
mand, meaning, for example, that 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 "smart" enough 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.

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, the load command will have to work in order to
support ADTs.
NAME
merge — merge two classes

SYNOPSIS
merge classname1 into classname2

DESCRIPTION
Merge will combine a version class, classname1, with its parent, classname2. If classname2 is a base class, then this operation merges a differently encoded offset, classname1, into its parent. On the other hand, if classname2 is also a version, then this operation combines two differentially encoded offsets together into a single one. In either case any children of classname1 become children of classname2.

A version class may not be merged into its parent class when the parent class is also the parent of another version class.

However, merging in the reverse direction is allowed. Specifically, merging the parent, classname1, with a version, classname2, causes classname2 to become disassociated from its parent. As a side effect, classname1 will be destroyed if is not the parent of some other version class.

EXAMPLE
/*
 * Combine office class and employee class
 */
merge office into employee

SEE ALSO
create version(commands), destroy(commands).

BUGS
Merge is not implemented in Version 4.2.
NAME
move — move the pointer in a portal

SYNOPSIS
move [ ( forward | backward ) ]
[ ( number | all | to ( number | record_qual ) ) ]
in portal_name

DESCRIPTION
Move allows a user to move the instance pointer within the portal named portal_name. Each portal has an instance pointer, which points to the previous instance to be fetched. It always points to before the first instance when the portal is first created. The pointer can be moved forward or backward. It can be moved to an absolute position or over a certain distance. An absolute position may be specified by using to; distance is specified by a number. Record_qual is a qualification without instance variables, aggregates, or set expressions which can be evaluated completely on a single instance in the portal.

EXAMPLE
/*
 * Move backwards 5 instances in the portal FOO
 */
move backward 5 in FOO

/*
 * Move to the 6th instance in the portal FOO
 */
move to 6 in FOO

SEE ALSO
close(commands), fetch(commands), retrieve(commands).

BUGS
Move is not implemented in Version 4.2. The portal pointer may be moved using fetch(commands) and ignoring its return values.
NAME

notify — signal all frontends and backends listening on a class

SYNOPSIS

notify class_name

DESCRIPTION

notify is used to awaken all backends and consequently all frontends that have executed listen(commands) on class_name. This can be used either within an instance-level rule as part of the action body or from a normal query. When used from within a normal query, this can be thought of as interprocess communication (IPC). When used from within a rule, this can be thought of as an alert mechanism.

Notice that the mere fact that a notify has been executed does not imply anything in particular about the state of the class (e.g., that it has been updated), nor does the notification protocol transmit any useful information other than the class name. Therefore, all notify does is indicate that some backend wishes its peers to examine class_name in some application-specific way.

This event notification is performed through the LIBPQ protocol and frontend application interface. The application program must explicitly poll a LIBPQ global variable, PQAsyncNotifyWaiting, and call the routine PQnotifies in order to find out the name of the class to which a given notification corresponds. If this code is not included in the application, the event notification will be queued and never be processed.

SEE ALSO

define rule(commands), listen(commands), libpq.
NAME
purge — discard historical data

SYNOPSIS
purge classname [ before abstime ] [ after reltime ]

DESCRIPTION
Purge allows a user to specify the historical retention properties of a class. If the date specified is an absolute time such as "Jan 1 1987", PostgreSQL will discard tuples whose validity expired before the indicated time. Purge with no before clause is equivalent to "purge before now". Until specified with a purge command, instance preservation defaults to "forever".

The user may purge a class at any time as long as the purge date never decreases. PostgreSQL will enforce this restriction, silently.

Note that the purge command does not do anything except set a parameter for system operation. Use vacuum(commands) to enforce this parameter.

EXAMPLE
/*
 * Always discard data in the EMP class
 * prior to January 1, 1989
 */
purge EMP before "Jan 1 1989"

/*
 * Retain only the current data in EMP
 */
purge EMP

SEE ALSO
vacuum(commands).

BUGS AND CAVEATS
Error messages are quite unhelpful. A complaint about "inconsistent times" followed by several nine-digit numbers indicates an attempt to "back up" a purge date on a relation.

You cannot purge certain system catalogs (namely, "pg_class", "pg_attribute", "pg_am", and "pg_amop") due to circularities in the system catalog code.

This definition of the purge command is really only useful for non-archived relations, since tuples will not be discarded from archive relations (they are never vacuumed).
NAME
remove aggregate — remove the definition of an aggregate

SYNOPSIS
remove aggregate argname

DESCRIPTION
Remove aggregate will remove all reference to an existing aggregate definition. To execute this command the current user must be the owner of the aggregate.

EXAMPLE
/*
 * Remove the average aggregate
 */
remove aggregate avg

SEE ALSO
define aggregate(commands).
NAME
   remove function — remove a user-defined C function

SYNOPSIS
   remove function function_name ( [ type-1 { , type-n } ])

DESCRIPTION
   Remove 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.

EXAMPLE
   /*
    * this command removes the square root function
    */
   remove function sqrt(int4)

SEE ALSO
   define function(commands).

BUGS
   No checks are made to ensure that types, operators or access methods that rely on the function have been
   removed first.
NAME
remove index — removes an index from POSTGRES

SYNOPSIS
remove index index_name

DESCRIPTION
This command drops an existing index from the POSTGRES system. To execute this command you must be
the owner of the index.

EXAMPLE
/*
 * this command will remove the "emp_index" index
 */
remove index emp_index

SEE ALSO
define index(commands).
NAME
remove operator — remove an operator from the system

SYNOPSIS
remove operator opr_desc

DESCRIPTION
This command drops an existing operator from the database. To execute this command you must be the
owner of the operator.

Opr_desc is the name of the operator to be removed followed by a parenthesized list of the operand types
for the operator. The left or right type of a left or right unary operator, respectively, may be specified as
none.

It is the user's responsibility to remove any access methods, operator classes, etc. that rely on the deleted
operator.

EXAMPLE
/*
 * Remove power operator a^n for int4
 */
remove operator (int4, int4)

/*
 * Remove left unary operator !a for booleans
 */
remove operator ! (none, bool)

/*
 * Remove right unary factorial operator a! for int4
 */
remove operator ! (int4, none)

/*
 * Remove right unary factorial operator a! for int4
 * (default is right unary)
 */
remove operator ! (int4)

SEE ALSO
define operator(commands).
NAME
remove rule – removes a current rule from POSTGRES

SYNOPSIS
remove [ instance | rewrite ] rule rule_name

DESCRIPTION
This command drops the rule named rule_name from the specified POSTGRES rule system. POSTGRES will immediately cease enforcing it and will purge its definition from the system catalogs.

EXAMPLE
/ *
    * This example drops the rewrite rule example_1
 */
    remove rewrite rule example_1

SEE ALSO
define rule(commands), remove view(commands).

BUGS
Once a rule is dropped, access to historical information the rule has written may disappear.
NAME
remove type — remove a user-defined type from the system catalogs

SYNOPSIS
remove type typename

DESCRIPTION
This command removes a user type from the system catalogs. Only the owner of a type can remove it.

It is the user’s responsibility to remove any operators, functions, aggregates, access methods, sub-types, classes, etc. that use a deleted type.

EXAMPLE
/*
 * remove the box type
 */
remove type box

SEE ALSO
introduction(commands), define type(commands), remove operator(commands).

BUGS
It is still possible to remove built-in types.
NAME
rename — rename a class or an attribute in a class

SYNOPSIS
rename classnamel to classname2
rename atname1 in classname [*] to atname2

DESCRIPTION
The rename command causes the name of a class or attribute to change without changing any of the data contained in the affected class. Thus, the class or attribute will remain of the same type and size after this command is executed.

In order to rename an attribute in each class in an entire inheritance hierarchy, use the classname of the superclass and append a "*". (By default, the attribute will not be renamed in any of the subclasses.) This should always be done when changing an attribute name in a superclass. If it is not, queries on the inheritance hierarchy such as

retrieve (s.all) from s in super*

will not work because the subclasses will be (in effect) missing an attribute found in the superclass.

You must own the class being modified in order to rename it or part of its schema. Renaming any part of the schema of a system catalog is not permitted.

EXAMPLE
/*
 * change the emp class to personnel
 */
rename emp to personnel

/*
 * change the sports attribute to hobbies
 */
rename sports in emp to hobbies

/*
 * make a change to an inherited attribute
 */
rename last_name in person* to family_name

BUGS
Execution of historical queries using classes and attributes whose names have changed will produce incorrect results in many situations.

Renaming of types, operators, rules, etc., should also be supported.
NAME
replace — replace values of attributes in a class

SYNOPSIS
replace instance_variable ( att_name-i = expression-i
{, att_name-i = expression-i })
[ from from_list ]
[ where qual ]

DESCRIPTION
Replace changes the values of the attributes specified in target_list for all instances which satisfy the qualification, qual. Only the attributes to be modified need appear in target_list.

Array references use the same syntax found in retrieve(commands). 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 class in order to modify it, as well as read access to any class whose values are mentioned in the target list or qualification (see change aci(commands).

EXAMPLES

/*
 * Give all employees who work for Smith a 10% raise
 */
replace emp(sal = 1.1 * emp.sal)
   where emp.mgr = "Smith"

/
 * Replace the middle element of a 3x3
 * noughts-and-crosses board with an O.
 */
replace tictactoe (board[2][2] = "O")
   where tictactoe.game = 1

/*
 * Replace the entire middle row of a 3x3
 * noughts-and-crosses board with Os.
 */
replace tictactoe (board[2:2] [1:3] = "(0,0,0)")
   where tictactoe.game = 2

/*
 * Replace the entire 3x3 noughts-and-crosses
 * board from game 2 with that of game 4
 */
replace tictactoe (board = ttt.board)
   from ttt in tictactoe
   where tictactoe.game = 2 and
       ttt.game = 4
SEE ALSO
    postquel(commands), create(commands), replace(commands), retrieve(commands).
NAME
retrieve — retrieve instances from a class

SYNOPSIS
retrieve
[ (into classname [ archive_mode ] )
portal portal_name !
!portal portal_name ] )
[unique]
( [ attr_name-i =] expression-i , [attr_name-i =] expression-i )
[ from from_list ]
[ where qual ]
[ sort by attr_name-i [using operator]
{ , attr_name-j [using operator] } ]

DESCRIPTION
Retrieve will get all instances which satisfy the qualification, qual, compute the value of each element in
the target list, and either (1) return them to an application program through one of two different kinds of
portals or (2) store them in a new class.

If classname is specified, the result of the query will be stored in a new class with the indicated name. If an
archive specification, archive_mode of light, heavy, or none is not specified, then it defaults to light archiv-
ing. (This default may be changed at a site by the DBA). The current user will be the owner of the new
class. The class will have attribute names as specified in the target list. A class with this name owned by
the user must not already exist. The keyword all can be used when it is desired to retrieve all fields of a
class.

If no result classname is specified, then the result of the query will be available on the specified portal and
will not be saved. If no portal name is specified, the blank portal is used by default. For a portal specified
with the !portal keyword, retrieve passes data to an application without conversion to external format. For
a portal specified with the portal keyword, retrieve passes data to an application after first converting it to
the external representation. For the blank portal, all data is converted to external format. Duplicate
instances are not removed when the result is displayed through a portal unless the optional unique tag is
appended, in which case the instances in the target list are sorted according to the sort clause and duplicates
are removed before being returned.

Instances retrieved into a portal may be fetched in subsequent queries by using the fetch command. Since
the results of a retrieve portal span queries, retrieve portal may only be executed inside of a begin/end
transaction block. Attempts to use named portals outside of a transaction block will result in a warning
message from the parser, and the query will be discarded.

The sort clause allows a user to specify that he wishes the instances sorted according to the corresponding
operator. This operator must be a binary one returning a boolean. Multiple sort fields are allowed and are
applied from left to right.

The target list specifies the fields to be retrieved. Each attr_name specifies the desired attribute or portion
of an array attribute. Thus, each attr_name takes the form

class_name.att_name

or, if the user only desires part of an array.

/*
 * Specify a lower and upper index for each dimension
 * (i.e., clip a range of array elements)
class_name.att_name[1Index-1:uIndex-1]..[1Index-i:uIndex-i]

class_name.att_name[uIndex-1]..[uIndex-i]

where each index or uindex is an integer constant.

When you retrieve an attribute which is of a complex type, the behavior of the system depends on whether you used "nested dots" to project out attributes of the complex type or not. See the examples below.

You must have read access to a class to read its values (see change acl(commands).

EXAMPLES

Retrieve all employees who make more than their manager

retrieve (e.name)
from e, m in emp
where e.mgr = m.name
and e.sal > m.sal

Retrieve all fields for those employees who make more than the average salary

retrieve into avgsal(ave = float8ave (emp.sal))
retrieve (e.all)
from e in emp
where e.sal > avgsal.ave

Retrieve all employee names in sorted order

retrieve unique (emp.name)
sort by name using <

Retrieve all employee names that were valid on 1/7/85 in sorted order

retrieve (e.name)
from e in emp["January 7 1985"]
sort by name using <
/*
 * Construct a new class, raise, containing 1.1
 * times all employee's salaries
 */
retrieve into raise (salary = 1.1 * emp.salary)

/*
 * Do a retrieve into a portal
 */
begin 
    retrieve portal myportal (pg_user.all)
    fetch 2 in myportal
    fetch all in myportal
    close myportal
end

/*
 * Retrieve an entire 3x3 array that represents
 * a particular noughts-and-crosses board.
 * This retrieves a 3x3 array of char.
 */
retrieve (tictactoe.board)
    where tictactoe.game = 2

/*
 * Retrieve the middle row of a 3x3 array that
 * represents a noughts-and-crosses board.
 * This retrieves a 1x3 array of char.
 */
retrieve (tictactoe.board[2:2][1:3])
    where tictactoe.game = 2

/*
 * Retrieve the middle element of a 3x3 array that
 * represents a noughts-and-crosses board.
 * This retrieves a single char.
 */
retrieve (tictactoe.board[2][2])
    where tictactoe.game = 2

/*
 * Retrieve all attributes of a class "newemp" that
 * contains two attributes, "name" and a complex type
 * "manager" which is of type "newemp". Since each
 * complex attribute represents a procedure recorded
in "pg_proc", the system will return the object IDs of each procedure. In this example, POSTGRES will return tuples of the form ("carol", 34562), ("sunita", 45662), and so on. The "manager" field is represented as an object ID.

```
retrieve (newemp.name, newemp.manager)
```

/*
 In order to see the attributes of a complex type, they must be explicitly projected. The following query will return tuples of the form ("carol", "mike", 23434), ("sunita", "mike", 23434)

```
retrieve (newemp.name, newemp.manager.name,
         newemp.manager.manager)
```

SEE ALSO
append(commands), close(commands), create(commands), fetch(commands), postquell(commands), replace(commands).

BUGS
Retrieve into does not delete duplicates.

Archive_mode is not implemented in Version 4.2.
If the backend crashes in the course of executing a retrieve into, the class file will remain on disk. It can be safely removed by the database DBA, but a subsequent retrieve into to the same name will fail with a cryptic error message about "BlockExtend".

Retrieve iportal returns data in an architecture dependent format, namely that of the server on which the backend is running. A standard data format, such as XDR, should be adopted.

Aggregate functions can only appear in the target list.
NAME
 vacuum — vacuum a database

SYNOPSIS
 vacuum

DESCRIPTION
 Vacuum is the POSTGRES vacuum cleaner. It opens every class in the database, moves deleted records to
 the archive for archived relations, cleans out records from aborted transactions, and updates statistics in the
 system catalogs. The statistics maintained include the number of tuples and number of pages stored in all
 classes. Running vacuum periodically will increase POSTGRES’s speed in processing user queries.

The open database is the one that is vacuumed. This is a new POSTQUEL command in Version 4.2; earlier
 versions of POSTGRES had a separate program for vacuuming databases. That program has been replaced
 by the vacuum(unix) shell script.

We recommend that production databases be vacuumed nightly, in order to keep statistics relatively current.
The vacuum query may be executed at any time, however. In particular, after copying a large class into
 POSTGRES or deleting a large number of records, it may be a good idea to issue a vacuum 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.

SEE ALSO
 vacuum(unix).
SECTION 5 — LIBPQ

DESCRIPTION

LIBPQ is the programmer's interface to POSTRES. LIBPQ is a set of library routines which allow queries to pass to the POSTGRES backend and instances to return through an IPC channel.

This version of the documentation is based on the 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 the following directories:

```
.../src/regress/demo
.../src/regress/regress
.../src/regress/video
.../src/bin/monitor
.../src/bin/fsutils
```

CONTROL AND INITIALIZATION

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:

- **PGHOST** sets the default server name.
- **PGDATABASE** sets the default POSTGRES database name.
- **PGPORT** sets the default communication port with the POSTGRES backend.
- **PGTTY** sets the file or tty on which debugging messages from the backend server are displayed.
- **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 enabled; see `introduction(unix)` for additional information on Kerberos.

Internal Variables

The following internal variables of LIBPQ can be accessed by the programmer:

```
char *PQhost;       /* the server on which POSTGRES
                     backend is running. */
char *PQport = NULL; /* The communication port with the
                      POSTGRES backend. */
char *PQtty;        /* The tty on the PQhost backend on
                      which backend messages are
                      displayed. */
char *PQoption;     /* Optional arguments to the backend */
char *PQdatabase;   /* backend database to access */
```
int PQportset = 0;  /* 1 if communication with
    backend is established */

int PQxactid = 0;    /* Transaction ID of the current
    transaction */

int PQtracep = 0;    /* 1 to print out front-end
    debugging messages */

int PQAsyncNotifyWaiting = 0; /* 1 if one or more asynchronous
    notifications have been
    triggered */

char PQerrormsg[];    /* null-delimited string containing the
    error message (usually from the backend)
    when the execution of a query or function
    fails */

QUERY EXECUTION FUNCTIONS
The following routines control the execution of queries from a C program.

PQsetdb Make the specified database the current database and reset communication using PQreset
(see below).

    void PQsetdb(char *dbname)
    char *dbname;

PQdb Returns the name of the POSTGRES database being accessed, or NULL if no database is
open. Only one database can be accessed at a time. The database name is a string limited
to 16 characters.

    char *PQdb()

PQreset Reset the communication port with the backend in case of errors. This function will
close the IPC socket connection to the backend thereby causing the next PQexec call to ask
for a new one from the postmaster. When the backend notices the socket was closed
it will exit, and when the postmaster is asked for the new connection it will start a new
backend.

    void PQreset()

PQfinish Close communication ports with the backend. Terminates communications and frees up
the memory taken up by the LIBPQ buffer.

    void PQfinish()

PQfn Send a function call to the POSTGRES backend. Provides access to the POSTGRES fast
path facility, a trapdoor into the system internals. See the FAST PATH section of the
manual.

03/12/94
The \texttt{fnid} argument is the object identifier of the function to be executed. \texttt{result_len} and \texttt{result_buf} specify the expected size (in bytes) of the function return value and a buffer in which to load the return value. The actual size of the returned value will be loaded into the space pointed to by \texttt{actual_result_len} if it is a valid pointer. \texttt{result_type} should be set to 1 if the return type is an integer and 2 in all other cases. \texttt{args} and \texttt{nargs} specify a pointer to a \texttt{PQArgBlock} structure (see 

\ldots ./src/backend/ctw/libpq.h

for more details) and the number of arguments, respectively.

\texttt{PQfn} returns a string containing the character “Q” when a return-value has been loaded into \texttt{result_buf}, or “V” if the function returned nothing. \texttt{PQfn} returns a NULL pointer and loads \texttt{PQerrormsg} if any error (fatal or non-fatal) occurs.

\texttt{PQfn} returns an error if \texttt{result_buf} is not large enough to contain the returned value.

\begin{verbatim}
char *PQfn(fnid, result_buf, result_len,
           actual_result_len,
           result_type, args, nargs)
  int fnid;
  int *result_buf;
  int result_len;
  int *actual_result_len;
  int result_type;
  PQArgBlock *args;
  int nargs;
\end{verbatim}

\textbf{PQexec} Submit a query to POSTGRES. Returns a status indicator or an error message.

If the query returns data (e.g., \texttt{fetch}), \texttt{PQexec} returns a string consisting of the character “P” followed by the name of the portal buffer.

If the query does not return any instances, as in the case with update queries, \texttt{PQexec} will return a string consisting of the character “C” followed by the command tag (e.g., “CREPLACE”).

If a “copy from stdin” or “copy to stdout” query is executed (see \texttt{copy(commands)} for more details), \texttt{PQexec} will return the strings “DCOPY” and “BCOPY”, respectively.

A string beginning with the character “I” indicates that the server has finished sending the results of a multi-query command (e.g., has finished processing an asynchronous portal command).

If a non-fatal error occurred during the execution of the query, \texttt{PQexec} will return (for historical reasons) the character “R” and load an error message into \texttt{PQerrormsg}. If a fatal error occurred (i.e., the backend crashed), \texttt{PQexec} returns the character “E” and loads an error message into \texttt{PQerrormsg}.

\begin{verbatim}
char *PQexec(query)
  char *query;
\end{verbatim}

\textbf{PQFlush} The frontend/backend protocol has a serious flaw in that the queries executed when using \texttt{PQfn} and \texttt{PQexec} can cause several query responses to come back to the frontend. For
example, during the definition of a view, the server actually executes several queries on its own to modify the system catalogs. Unfortunately, the implementation of this was botched and these queries return status messages to the frontend of their own. If the frontend application only reads one response and then goes on to execute more queries, these extra responses sit in the message queue and the frontend will read these leftovers instead of reading the responses from its latest queries.

If you aren’t completely positive that a call to PQexec won’t do something more complicated than a simple retrieve, you should probably wrap it in a loop that processes “P” and “C” responses in the usual way, but also performs

```c
result = PQexec(" "); /* dummy query */
++dummies_sent;
```

after receiving each good protocol result. When the first character of a PQexec result is ‘T’, you know you have received the last result and have started receiving responses to your dummy queries. To get rid of the “T” protocol responses that are now stuffed into your message buffer, call PQFlushI with the number of dummy queries you sent.

This is horrendously complicated and should be fixed. Meanwhile, you should look at...

`.../src/bin/monitor/monitor.c`

to see an example of a program that handles this problem correctly.

```c
int PQFlushI(i_count)
    int i_count;
```

PORTAL FUNCTIONS

A portal is a POSTGRES buffer from which instances can be fetched. Each portal has a string name (currently limited to 16 bytes). A portal is initialized by submitting a retrieve statement using the PQexec function, for example:

```
retrieve portal foo (EMP.all)
```

The programmer can then move data from the portal into LIBPQ by executing a fetch statement, e.g:

```
fetch 10 in foo
fetch all in foo
```

If no portal name is specified in a query, the default portal name is the string “blank”, known as the blank portal. All qualifying instances in a blank portal are fetched immediately, without the need for the programmer to issue a separate fetch command.

Data fetched from a portal into LIBPQ is moved into a portal buffer. Portal names are mapped to portal buffers through an internal table. Each instance in a portal buffer has an index number locating its position in the buffer. In addition, each field in an instance has a name (attribute name) and a field index (attribute number).

A single retrieve command can return multiple types of instances. This can happen if a POSTGRES function is executed in the evaluation of a query or if the query returns multiple instance types from an
inheritance hierarchy. Consequently, the instances in a portal are set up in groups. Instances in the same

Portals that are associated with normal user commands are called synchronous. In this case, the application program is expected to issue a retrieval followed by one or more fetch commands. The functions that follow can now be used to manipulate data in the portal.

PQnportals Return the number of open portals. If rule_p is not 0, then only return the number of asynchronous portals.

```
int PQnportals(rule_p)
  int rule_p;
```

PQpnames Return all portal names. If rule_p is not 0, then only return the names of asynchronous portals. The caller is responsible for allocating sufficient storage for pnames. The number of names returned can be determined with a call to PQnportals(). Each portal name is at most PortalNameLength characters long (see .../src/backend/tmp/libpq.h).

```
void PQpnames(pnames, rule_p)
  char **pnames;
  int rule_p;
```

PQparray Return the portal buffer given a portal name, pname.

```
PortalBuffer *PQparray(pname)
  char *pname;
```

PQclear Free storage claimed by portal pname.

```
void PQclear(pname)
  char *pname;
```

PQntuples Return the number of instances (tuples) in a portal buffer portal.

```
int PQntuples(portal)
  PortalBuffer *portal;
```

PQngroups Return the number of instance groups in a portal buffer portal.

```
int PQngroups(portal)
  PortalBuffer *portal
```

PQntuplesGroup Return the number of instances in an instance group group_index associated with a portal buffer portal.

```
int PQntuplesGroup(portal, group_index)
  PortalBuffer *portal;
```
int group_index;

PQnfieldsGroup: Return the number of fields (attributes) for the instances (tuples) in instance group

\[\text{group\_index} \]\n
associated with portal buffer \(\text{portal}\).

\[
\text{int PQnfieldsGroup}(\text{portal, group\_index}) \\
\text{PortalBuffer *portal; } \\
\text{int group\_index; }
\]

PQnameGroup: Return the field (attribute) name for the instances (tuples) in instance group \(\text{group\_index}\) (associated with portal buffer \(\text{portal}\)) and the field index \(\text{field\_number}\).

\[
\text{char *PQnameGroup}(\text{portal, group\_index, field\_number}) \\
\text{PortalBuffer *portal; } \\
\text{int group\_index; } \\
\text{int field\_number; }
\]

PQnumberGroup: Return the field index (attribute number) given the instance group \(\text{group\_index}\) (associated with portal buffer \(\text{portal}\)) and the field (attribute) name \(\text{field\_name}\).

\[
\text{int PQnumberGroup}(\text{portal, group\_index, field\_name}) \\
\text{PortalBuffer *portal; } \\
\text{int group\_index; } \\
\text{char *field\_name; }
\]

PQgetgroup: Returns the index of the instance group (associated with portal buffer \(\text{portal}\)) that contains a particular instance \(\text{tuple\_index}\).

\[
\text{int PQgetgroup}(\text{portal, tuple\_index}) \\
\text{PortalBuffer *portal; } \\
\text{int tuple\_index; }
\]

PQnfields: Returns the number of fields (attributes) in an instance \(\text{tuple\_index}\) contained in portal buffer \(\text{portal}\).

\[
\text{int PQnfields}(\text{portal, tuple\_index}) \\
\text{PortalBuffer *portal; } \\
\text{int tuple\_index; }
\]

PQnumber: Returns the field index (attribute number) of a given field name \(\text{field\_name}\) within an instance \(\text{tuple\_index}\) contained in portal buffer \(\text{portal}\).

\[
\text{int PQnumber}(\text{portal, tuple\_index, field\_name}) \\
\text{PortalBuffer *portal; } \\
\text{int tuple\_index; } \\
\text{char *field\_name; }
\]

03/12/94 132
PQname Returns the name of a field (attribute) field_number of instance tuple_index contained in portal buffer portal.

\[
\begin{align*}
\text{char } *\text{PQname}(\text{portal, tuple_index, field_number}) \\
\text{PortalBuffer } *\text{portal}; \\
\text{int tuple_index}; \\
\text{int field_number};
\end{align*}
\]

PQftype Returns the type of a field (attribute) field_number of instance tuple_index contained in portal buffer portal. The type returned is an internal coding of a type.

\[
\begin{align*}
\text{int } \text{PQftype}(\text{portal, tuple_index, field_number}) \\
\text{PortalBuffer } *\text{portal}; \\
\text{int tuple_index}; \\
\text{int field_number};
\end{align*}
\]

PQsametype Returns 1 if two instances tuple_index1 and tuple_index2, both contained in portal buffer portal, have the same field (attribute) types.

\[
\begin{align*}
\text{int } \text{PQsametype}(\text{portal, tuple_index1, tuple_index2}) \\
\text{PortalBuffer } *\text{portal}; \\
\text{int tuple_index1}; \\
\text{int tuple_index2};
\end{align*}
\]

PQgetvalue Returns a field (attribute) value.

\[
\begin{align*}
\text{char } *\text{PQgetvalue}(\text{portal, tuple_index, field_number}) \\
\text{PortalBuffer } *\text{portal}; \\
\text{int tuple_index}; \\
\text{int field_number};
\end{align*}
\]

PQgetlength Return the length of a field (attribute) value in bytes. If the field is a struct varlena, the length returned here does not include the size field of the varlena, i.e., it is 4 bytes less.

\[
\begin{align*}
\text{char } *\text{PQgetlength}(\text{portal, tuple_index, field_number}) \\
\text{PortalBuffer } *\text{portal}; \\
\text{int tuple_index}; \\
\text{int field_number};
\end{align*}
\]

If the portal is blank, or the portal was specified with the portal keyword, all values are returned as null-delimited strings. It is the programmer's responsibility to convert them to the correct type. If the portal is specified with the iportal keyword, all values are returned in an architecture-dependent internal (binary) format, namely, the format generated by the input function specified through define type(commands). Again, it is the programmer's responsibility to convert the data to the correct type.

**ASYNCHRONOUS PORTALS AND NOTIFICATION**

Asynchronous portals — query results of rules — are implemented using two mechanisms: relations and notification. The query result is transferred through a relation. The notification is done with special POSTQUEL commands and the frontend/backend protocol.
The first step in using asynchronous portals is to listen on a given class name. The fact that a process is listening on the class is shared with all backend servers running on a database; when one sets off the rule, it signals its peers. The backend server associated with the listening frontend process then sends its client an IPC message, which the frontend process must explicitly catch by polling the variable PQAsyncNotify. When this variable is non-zero, the frontend process must first issue a null (empty) query, i.e.,

```c
PQexec(" * ");
```

Then the frontend should check the variable, PQAsyncNotifyWaiting. When this variable is non-zero, the frontend can retrieve the notification data held using PQNotifies. The frontend must call PQNotifies in order to find out which classes the data corresponds to (i.e., which notification events have been set off). These events must then be individually cleared by calling PQRemoveNotify on each element of the list returned by PQNotifies.

Notice that the asynchronous notification process does not itself transfer any data, but only a class name. Hence the frontend and backend must come to agreement on the class to be used to pass any data prior to notification and data transfer (obviously, since the frontend must specify this table name in the corresponding listen command).

The second sample program gives an example of the use of asynchronous portals in which the frontend program retrieves the entire contents of the result class each time it is notified.

PQNotifies Return the list of relations on which notification has occurred.

```c
PQNotifyList *PQNotifies()
```

PQRemoveNotify Remove the notification from the list of unhandled notifications.

```c
PQNotifyList *PQRemoveNotify(pqNotify)
```

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.

For more information about the copy command, see copy(commands).

PQgetline Reads a newline-terminated line of characters (transmitted by the backend server) into a buffer `string` of size `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 single character "\n", which indicates that the backend server has finished sending the results of the copy command. 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 `PQgetline` very carefully.

03/12/94 134
The code in

```c
.../src/bin/monitor/monitor.c
```

contains routines that correctly handle the copy protocol.

```c
PQgetline(string, length)
    char *string;
    int length
```

**PQputline** Sends a null-terminated `string` to the backend server.

The application must explicitly send the single character `\0` to indicate to the backend that it has finished sending its data.

```c
PQputline(string)
    char *string;
```

**PQendcopy** Syncs with the backend. 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 `PQputline` or when the last string has been received from the backend using `PQgetline`. 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.

```c
int PQendcopy()
```

As an example:

```c
PQexec("create foo (a=int4, b=char16, d=float8)*");
PQexec("copy foo from stdin");
PQgetline("3<TAB>hello world<TAB>4.5\n");
PQgetline("4<TAB>goodbye world<TAB>7.11\n");
...
PQputline(".\n");
PQendcopy();
```

**LIBPQ TRACING FUNCTIONS**

**PQtrace** Enable tracing. The routine sets the `PQtracep` variable to 1 which causes debug messages to be printed. You should note that the messages will be printed to `stdout` by default. If you would like different behavior you must set the variable

```c
FILE *debug_port
```

to the appropriate stream.

```c
void PQtrace()
```
INTRODUCTION

PQuntrace Disable tracing started by PQtrace.

void PQuntrace()

USER AUTHENTICATION FUNCTIONS

If the user has generated the appropriate authentication credentials (e.g., obtaining Kerberos tickets), the frontend/backend authentication process is handled by PQexec without any further intervention. The following routines may be called by LIBPQ programs to tailor the behavior of the authentication process.

fe_getauthname Returns a pointer to static space containing whatever name the user has authenticated. Use of this routine in place of calls to getenv(3) or getpwuid(3) by applications is highly recommended, as it is entirely possible that the authenticated user name is not the same as value of the USER environment variable or the user's entry in /etc/passwd. This becomes an important issue if the user name is being used as a value in a database interaction (e.g., using the user name as the default database name, as is done by monitor(unix).

char *fe_getauthname()

fe_setautsvc Specifies that LIBPQ should use authentication service name rather than its compiled-in default. This value is typically taken from a command-line switch.

void fe_setautsvc(name)
    char *name;

BUGS

The query buffer is 8192 bytes long, and queries over that length will be silently truncated.
SAMPLE PROGRAM 1

/*
 * testlibpq.c —
 * Test the C version of LIBPQ, the POSTGRES frontend library.
 */
#include <stdio.h>
#include "tmp/libpq.h"

main ()
{
    int i, j, k, g, n, m, t;
    PortalBuffer *p;
    char pnames[MAXPORTALS][portal_name_length];

    /* Specify the database to access. */
PQsetdb ("pic_demo");

    /* Start a transaction block for eportal */
PQexec ("begin");

    /* Fetch instances from the EMP class. */
PQexec ("retrieve portal eportal (EMP.all)");
PQexec ("fetch all in eportal");

    /* Examine all the instances fetched. */
p = PQarray ("eportal");
g = PQngroups (p);
t = 0;

    for (k = 0; k < g; k++) {
        printf ("VnA new instance group:
        n = PQntuplesGroup (p, k);
m = PQnfieldsGroup (p, k);

        /* Print out the attribute names. */
        for (i = 0; i < m; i++) {
            printf ("%-15s", PQfnameGroup (p, k, i));
            printf ("n");

        /* Print out the instances. */
        for (i = 0; i < n; i++) {
            for (j = 0; j < m; j++) {
                printf("%-15s", PQgetvalue(p, t+i, j));
                printf ("n");
            }
            t += n;
        }
    }
    
    /* Close the portal. */
PQexec ("close eportal");

03/12/94
/* End the transaction block */
PQexec("end");

/* Try out some other functions. */

/* Print out the number of portals. */
printf ("Number of portals open: %d.
", PQnportals ());

/* If any tuples are returned by rules, print out
* the portal name. */
if (PQnportals (1)) {
    printf ("Tuples are returned by rules. \n");
PQpnames (pnames, 1);
    for (i = 0; i < MAXPORTALS; i++)
        if (pnames[i] != NULL)
            printf ("portal used by rules: %s\n", pnames[i]);
}

/* finish execution. */
PQfinish ();
SAMPLE PROGRAM 2
/*
 * Testing of asynchronous notification interface.
 * Do the following at the monitor:
 * * create test1 (i = int4) g
 * * create test1a (i = int4) g
 * * define rule r1 is on append to test1 do
 * [append test1a (i = new.i) notify test1a] g
 * *
 * Then start up this process.
 * * append test1 (i = 10) g
 * *
 * The value i = 10 should be printed by this process.
 */

#include <tmp/simplelists.h>
#include <tmp/libpq.h>
#include <tmp/postgres.h>

extern int PQAsyncNotifyWaiting;

void main() {
    PQNotifyList *l;
    PortalBuffer *portalBuf;
    char *res;
    int ngroups, tupno, grpno, ntups, nfields;
    PQsetdb(getenv("USER"));
    PQexec("listen test1a");
    
    while (1) {
        res = PQexec(" ");
        if (*res != 'I') {
            printf("Unexpected result from a null query --> %s", res);
            PQfinish();
            exit(1);
        }
        
        if (PQAsyncNotifyWaiting) {
            PQAsyncNotifyWaiting = 0;
            for (l = PQnotifies(); l != NULL; l = PQnotifies()) {
                PQremoveNotify(l);
                printf("Async. notification on relation %s, our backend pid is %d\n",}

            03/12/94
l->relname, l->be_pid);
res = PQexec("retrieve (testla.i)");

if (*res != 'P') {
    fprintf(stderr, "no portal", ++res);
PQfinish();
    exit(1);
}

portalBuf = PQparray(++res);
ngroups = PQnpgroups(portalBuf);
for (grpno = 0 ; grpno < ngroups ; grpno++) {
    ntups = PQntuplesGroup(portalBuf, grpno);
    nflds = PQnfieldsGroup(portalBuf, grpno);
    if (nflds != 1) {
        fprintf(stderr, "expected 1 attributes, got %d\n", nflds);
PQfinish();
        exit(1);
    }
    for (tupno = 0 ; tupno < ntups ; tupno++) {
        printf("i = %s\n", PQgetvalue(portalBuf, tupno, 0));
    }
    for ()
    PQfinish();
    exit(0);
}
sleep(1);
}
SAMPLE PROGRAM 3

/*
* Test program for the binary portal interface.
*
* Create a test database and do the following at the monitor:
* *
* * create test1 (i = int4, d = float4, p = polygon)
* * append test1 (i = 1, d = 3.567,
*      p = "(3.0,4.0,1.0,2.0)"::polygon)
* * append test1 (i = 2, d = 89.05,
*      p = "(4.0,3.0,2.0,1.0)"::polygon)
* *
* adding as many tuples as desired.
* *
* Start up this program. The contents of class "test1" should be
* printed, e.g.:
*
tuple 0: got
  i=(4 bytes) 1,
  d=(4 bytes) 3.567000,
  p=(72 bytes) 2 points,
      boundbox=(hi=3.000000,4.000000 / lo=1.000000,2.000000)

tuple 1: got
  i=(4 bytes) 2,
  d=(4 bytes) 89.050000,
  p=(72 bytes) 2 points,
      boundbox=(hi=4.000000,3.000000 / lo=2.000000,1.000000)
*/

#include "tmp/simplelists.h"
#include "tmp/libpq.h"
#include "utils/geo-decls.h"

void main()
{
  PortalBuffer *portalbuf;
  char *res;
  int ngroups, tupno, grpno, ntups, nflds;

  PQsetdb("test");  /* change this to your database name */
PQexec("begin");
  res = (char *) PQexec("retrieve iportal junk (test1.all)");
  if (*res == 'E') {
    fprintf(stderr,"\nError: %s\n",++res);
    goto exit_error;
  }

  res = (char *) PQexec("fetch all in junk");
  if (*res != 'P') {
    fprintf(stderr,"\nError: no portal\n");
    goto exit_error;
  }

  /* Do your processing here */
}

exit_error:
  return 0;

03/12/94 141
/* get tuples in relation */
portalbuf = PQparray(+-res);
ngroups = PQngroups(portalbuf);
for (grpno = 0; grpno < ngroups; grpno++) {
    ntups = PQntuplesGroup(portalbuf, grpno);
    if ((nflds = PQnfieldsGroup(portalbuf, grpno)) != 3) {
        fprintf(stderr, "XnError: expected 3 attributes, got %d\n", nflds);
        goto exit_error;
    }
    for (tupno = 0; tupno < ntups; tupno++) {
        int *ival; /* 4 bytes */
        float *fval; /* 4 bytes */
        unsigned plen;
        POLYGON *pval;
        ival = (int *) PQgetvalue(portalbuf, tupno, 0);
        fval = (float *) PQgetvalue(portalbuf, tupno, 1);
        plen = PQgetlength(portalbuf, tupno, 2);
        if (!(pval = (POLYGON *) palloc(plen + sizeof(long)))) {
            fprintf(stderr, "XnError: palloc returned zero bytes\n");
            goto exit_error;
        }
        pval->size = plen + sizeof(long);
        bcopy(PQgetvalue(portalbuf, tupno, 2), (char *) &pval->npts, plen);
        printf("tuple %d: got\n\t\ti=(%d bytes) %d,\n\t\td=(%d bytes) %f,\n\t\tp=(%d bytes) %d points,\n\t\t\tboundbox=hi=%f,%f / lo=%f,%f)\n",
            tupno,
            PQgetlength(portalbuf, tupno, 0),
            *ival,
            PQgetlength(portalbuf, tupno, 1),
            *fval,
            PQgetlength(portalbuf, tupno, 2),
            pval->npts,
            pval->boundbox.xh,
            pval->boundbox.yh,
            pval->boundbox.xl,
            pval->boundbox.yl);
        }
    }
}
PQexec("end");
PQfinish();
exit(0);
exit_error:
PQexec("end");
PQfinish();
exit(1);
SECTION 6 — FAST PATH

SYNOPSIS
retriev( retval = function([ arg 1, arg 2 ]))

DESCRIPTION
POSTGRES allows any valid POSTGRES function to be called in this way. Prior implementations of fast path allowed user functions to be called directly. For now, the above syntax should be used, with arguments cast into the appropriate types. By executing the above type of query, control transfers completely to the user function; any user function can access any POSTGRES function or any global variable in the POSTGRES address space.

There are six levels at which calls can be performed:
1) Traffic cop level
   If a function wants to execute a POSTGRES command and pass a string representation, this level is appropriate.
2) Parser
   A function can access the POSTGRES parser, passing a string and getting a parse tree in return.
3) Query optimizer
   A function can call the query optimizer, passing it a parse tree and obtaining a query plan in return.
4) Executor
   A function can call the executor and pass it a query plan to be executed.
5) Access methods
   A function can directly call the access methods if it wishes.
6) Function manager
   A function can call other functions using this level.

Documentation of layers 1-6 will appear at some future time. Meanwhile, fast path users must consult the source code for function names and arguments at each level.

It should be noted that users who are concerned with ultimate performance can bypass the query language completely and directly call functions that in turn interact with the access methods. On the other hand, a user can implement a new query language by coding a function with an internal parser that then calls the POSTGRES optimizer and executor. Complete flexibility to use the pieces of POSTGRES as a tool kit is thereby provided.

993/08/23 09:03:16 aoki Exp $
SECTION 7 — LARGE OBJECTS

DESCRIPTION

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.

POSTGRES supports three standard implementations of large objects: as files external to POSTGRES, as UNIX files managed by POSTGRES, and as data stored within the POSTGRES database. These implementations allow users to trade-off between access speed, recoverability, and security. The choice of implementation is specified when the large object is created or "registered" with POSTGRES. In all cases, the large object becomes associated with a path name within a file system name space managed by POSTGRES (see below).

Applications which can tolerate lost data may store large objects as conventional files which are fast to access, but cannot be recovered in the case of system crashes. For applications requiring stricter data integrity, the transaction-protected large object implementation is available. This section describes each implementation and the programmatic and query language interfaces to POSTGRES large object data.

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. As mentioned above, POSTGRES supports functional indices on large object data. In this example, the results of the beard function could be stored in a B-tree index to provide fast searches for people with red beards.

UNIX FILES AS LARGE OBJECT ADTS

The simplest large object interface supplied with POSTGRES is also the least robust. It does not support transaction protection, crash recovery, or time travel. On the other hand, it can be used on existing data files (such as word-processor files) that must be accessed simultaneously by the database system and existing application programs.

POSTGRES has two ways of handling UNIX files that store large objects. These correspond to the External and Unix large object interfaces.

The simplest way to create a large object is to register the external file containing the large object with the POSTGRES database. This leaves the actual file as-is, outside of the POSTGRES data directory, and allows other UNIX users to access it without going through POSTGRES. The file is, in general, only protected by the standard UNIX permissions mechanism. In the case of a system crash, or if the file is removed or deleted, POSTGRES provides no recovery mechanism.

In the second approach, the user registers the large object in the POSTGRES database and copies the specified file into the POSTGRES database directory structure. Copying the file takes time, so this is not as fast as the External large object creation process. Furthermore, like External large objects, UNIX large objects are not recoverable. However, placing the large object files in the POSTGRES data area gives them the security of POSTGRES data files.

External large objects provide POSTGRES users with the ability to share large objects between POSTGRES and other systems. The files can be read and written by other UNIX users, and POSTGRES can be made aware of the large object very quickly. However, because of the security implications of the External large
INTRODUCTION (LARGE OBJECTS)

objects approach, the facility is not provided by default. To enable External large objects, refer to the POSTGRES release notes.

INVERSION LARGE OBJECTS
In contrast to UNIX files as large objects, 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.

Only programs that use the POSTGRES data manager can read and write Inversion large objects. Inversion large objects are slower than storing large objects as UNIX files, and they require more space.

LARGE OBJECT 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 LIBPQ interface, are described below. As POSTGRES has evolved a newer set of functions providing a more coherent interface have replaced an older set. The most recent approach will be described first, and the historical information included at the very end for completeness.

LARGE OBJECTS: BACKEND INTERFACE
This section describes how large objects may be accessed from dynamically-loaded C functions.

Creating New Large Objects
The routine

```c
int LOcreatin(path, mode, objtype)
char *path;
int mode;
int objtype;
```

creates a new large object.

The pathname is a slash-separated list of components, and must be a unique pathname in the POSTGRES large object namespace. There is a virtual root directory "CD in which objects may be placed.

The `objtype` parameter can be one of Inversion, UNIX or External. These are symbolic constants defined in

`.../include/catalog/pg_lobj.h`

The interpretation of the `mode` argument depends on the `objtype` selected. (Note that the External type is conditionally compiled into the backend. Please refer to the Release Notes for information on enabling External large objects and to the introduction of this section for a discussion on External large objects.)

For UNIX large objects, the `mode` is the mode used to protect the file on the UNIX file system. On creation, the file is open for reading and writing.

For External large objects, `mode` specifies the desired access permissions. If the file exists, the file permissions on the external file are compared to the requested mode; both the user who is currently connected to the backend server and the "postgres" user must have the appropriate permissions. Unlike `creat(2)`, an existing external file is not truncated.

For Inversion large objects, `mode` is a bitmask describing several different attributes of the new object. The symbolic constants listed here are defined in

`.../include/tmp/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. In the distributed version of POSTGRES, only the magnetic disk storage manager is supported. For users running POSTGRES at UC Berkeley, additional storage managers are available. For sites other than Berkeley, these bits should always be zero. At Berkeley, storage manager zero is magnetic disk, storage manager one is a Sony optical disk jukebox, and storage manager two is main memory.

The commands below open two large objects for writing and reading. The Inv_access large object is not archived, and is located on magnetic disk:

```c
unix_fd = LOcreat("/my_uni.x Obj", 0600, Unix);
inv_fd = LOcreat("/my inv_obj",
INV_READ|INV_WRITE, Inversion);
```

Opening Large Objects

Large objects registered into the database by the LOcreat call described above, or p_open call described below may be opened by calling the routine

```c
int LOopen(path, mode)
    char *path;
    int mode;
```

where the path argument specifies the large object's pathname, and is the same as the pathname used to create the object. The mode argument is interpreted by the two implementations differently. For UNIX large objects, values should be chosen from the set of mode bits passed to the open system call; that is, O_CREAT, O_RDONLY, O_WRONLY, O_RDWR, and O_TRUNC. For Inversion large objects, only the bits INV_READ and INV_WRITE have any meaning.

To open the two large objects created in the last example, a programmer would issue the commands

```c
unix_fd = LOopen("/my_unix_obj", O_RDWR);
inv_fd = LOopen("/my_inv_obj", INV_READ|INV_WRITE);
```

If a large object is opened before it has been created, then a new large object is created using the UNIX implementation, and the new object is opened.

Seeking on Large Objects

The command

```c
int
    L01seek(fd, offset, whence)
    int fd;
    int offset;
    int whence;
```

moves the current location pointer for a large object to the specified position. The fd parameter is the file descriptor returned by either LOcreat or LOopen. Offset is the byte offset in the large object to which to seek.
Because UNIX large objects are simply UNIX files, they may have "holes" like any other UNIX file. That is, a program may seek well past the end of the object and write bytes. Intervening blocks will not be created and reading them will return zero-filled blocks. Inversion large objects do not support holes.

The following code seeks to byte location 100000 of the example large objects:

```c
unix_stat = L0Lseek(unix_fd, 100000, L_SET);
inv_stat = L0Lseek(inv_fd, 100000, L_SET);
```

On error, `L0Lseek` returns a value less than zero. On success, the new offset is returned.

### Writing to Large Objects

Once a large object has been created, it may be filled by calling

```c
int L0write(fd, wbuf)
int fd;
struct varlena *wbuf;
```

Here, `fd` is the file descriptor returned by `L0creat` or `L0open`, and `wbuf` describes the data to write. The `varlena` structure in POSTGRES consists of four bytes in which the length of the datum is stored, followed by the data itself. The length stored in the length field includes the four bytes occupied by the length field itself.

For example, to write 1024 bytes of zeroes to the sample large objects:

```c
struct varlena *vl;
vl = (struct varlena *) palloc(1028);
VARSIZE(vl) = 1028;
bzero(VARDATA(vl), 1024);
nwrite_unix = L0write(unix_fd, vl);
nwrite_inv = L0write(inv_fd, vl);
```

`L0write` returns the number of bytes actually written, or a negative number on error. For Inversion large objects, the entire write is guaranteed to succeed or fail. That is, if the number of bytes written is non-negative, then it equals `VARSIZE(vl)`.

The `VARSIZE` and `VARDATA` macros are declared in the file

```c
.../include/tmp/postgres.h
```

### Reading from Large Objects

Data may be read from large objects by calling the routine

```c
struct varlena *
L0read(fd, len)
int fd;
int len;
```
This routine returns the byte count actually read and the data in a varlena structure. For example,

```c
struct varlena *unix_vl, *inv_vl;
int nread_ux, nread_inv;
char *data_ux, *data_inv;

unix_vl = L0read(unix_fd, 100);
nread_ux = VARSIZE(unix_vl);
data_ux = VARDATA(unix_vl);

inv_vl = LOread(inv_fd, 100);
nread_inv = VARSIZE(inv_vl);
data_inv = VARDATA(inv_vl);
```

The returned varlena structures have been allocated by the POSTGRES memory manager `palloc`, and may be `pfree`d when they are no longer needed.

Closing a Large Object

Once a large object is no longer needed, it may be closed by calling

```c
int LOclose(fd)
    int fd;
```

where `fd` is the file descriptor returned by `LOpen` or `L0creat`. On success, `LOclose` returns zero. A negative return value indicates an error.

For example,

```c
if (LOclose(unix_fd) < 0)
    /* error */

if (LOclose(inv_fd) < 0)
    /* error */
```

Directory Operations

The routine

```c
int L0mkdir(path, mode)
    char *path;
    int mode;
```

creates directories in the POSTGRES virtual file system but does not create any physical directories. Naturally,

```c
int L0rmdir(path)
    char *path;
```

removes directories in the POSTGRES virtual file system. Both routines return zero or negative values on
success and failure, respectively.

Removing Large Objects
The routine to remove large objects works differently for the different large object types. A call to

```c
int LOunlink(path)
    char *path;
```

will always remove the specified path from the POSTGRES virtual file system. However, it will only unlink
the underlying data file in the case of a UNIX large object. Neither External nor Inversion large object files
are actually removed by this call. LOunlink returns zero on success, negative values on failure.

LARGE OBJECTS: LIBPQ INTERFACE
Large objects may also be accessed from database client programs that link the LIBPQ library. This library
provides a set of routines that support opening, reading, writing, closing, and seeking on large objects. The
interface is similar to that provided via the backend, but rather than using varlena structures, a more con-
ventional UNIX-style buffer scheme is used.

This section describes the LIBPQ interface in detail.

Creating a Large Object
The routine

```c
int p_croat(path, mode, objtype)
    char *path;
    int mode;
    int objtype;
```

creates a new large object. The path argument specifies a large-object system pathname.

The objtype parameter can be one of Inversion, Unix or External, which are symbolic constants defined in

```
./include/catalog/pg_lobj.h
```

The interpretation of the mode and files arguments depends on the objtype selected.

For UNIX files, mode is the mode used to protect the file on the UNIX file system. On creation, the file is
open for reading and writing. The path name is an internal convention relative to the specific database and
the actual files are stored in the directory of the database itself.

For External large objects, mode specifies the desired access permissions. If the file exists, the file permis-
sions on the external file are compared to the requested mode; both the user who is currently connected to
the backend server and the "postgres" user must have the appropriate permissions. Unlike creat(2), an
existing external file is not truncated.

For Inversion large objects, mode is a bitmask describing several different attributes of the new object. The
symbolic constants listed here are defined in

```
./include/tmp/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. At Berkeley, storage manager zero is magnetic disk, storage manager one is a Sony optical disk jukebox, and storage manager two is main memory.

The commands below open large objects of the two types for writing and reading. The Inversion large object is not archived, and is located on magnetic disk:

```c
unix_fd = p_creat("/my_unix_obj", 0600, Unix);
inv_fd = p_creat("/my_inv_obj",
               INV_READ|INV_WRITE, Inversion);
```

Opening an Existing Large Object

To open an existing large object, call

```c
int
p_open(path, mode)
  char *path;
  int mode;
```

The `path` argument specifies the large object pathname for the object to open. The `mode` bits control whether the object is opened for reading, writing, or both. For UNIX large objects, the appropriate flags are O_CREAT, O_RDONLY, O_WRONLY, O_RDWR, and O_TRUNC. For Inversion large objects, only INV_READ and INV_WRITE are recognized.

If a large object is opened before it is created, it is created by default using the UNIX file implementation.

Writing Data to a Large Object

The routine

```c
int
p_write(fd, buf, len)
  int fd;
  char *buf;
  int len;
```

writes `len` bytes from `buf` to large object `fd`. The `fd` argument must have been returned by a previous `p_creat` or `p_open`.

The number of bytes actually written is returned. In the event of an error, the return value is negative.

Seeking on a Large Object

To change the current read or write location on a large object, call

```c
int
p_lseek(fd, offset, whence)
  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`. For this release of POSTGRES, only L_SET is a legal value for `whence`. 
CLOSING A LARGE OBJECT

A large object may be closed by calling

```c
int
p_close(fd)
int fd;
```

where `fd` is a large object descriptor returned by `p_creat` or `p_open`. On success, `p_close` returns zero. On error, the return value is negative.

DIRECTORY OPERATIONS

The routines

```c
int
p_mkdir(path, mode)
char *path;  //\int mode;
```

and

```c
int
p_rmdir(path)
char *path;
```

are analogous to `LOmkdir` and `LOrmdir` in that they only modify the POSTGRES file system namespace and return zero or negative values on success or failure, respectively.

REMOVING LARGE OBJECTS

The

```c
int
p_unlink(path)
char *path;
```

routine removes the specified path from the POSTGRES file system namespace and, if the path corresponds to a UNIX large object, removes the underlying file. The files that store other large object types are not removed by this call. `p_unlink` returns zero or negative values on success or error, respectively.

SAMPLE LARGE OBJECT PROGRAMS

The POSTGRES large object implementation serves as the basis for a file system (the "Inversion file system") built on top of the data manager. This file system provides time travel, transaction protection, and fast crash recovery to clients of ordinary file system services. It uses the Inversion large object implementation to provide these services.

The programs that comprise the Inversion file system are included in the POSTGRES source distribution, in the directory

```
.../src/bin/fsutils
```

These directories contain a set of programs for manipulating files and directories. These programs are based on the Berkeley Software Distribution NET-2 release.

These programs are useful in manipulating Inversion files, but they also serve as examples of how to code.
large object accesses in LIBPQ. All of the programs are LIBPQ clients, and all use the interfaces that have been described in this section.

Interested readers should refer to the files in the directory

```
.../src/bin/fsutilg
```

for in-depth examples of the use of large objects. Below, a more terse example is provided. This code fragment creates a new large object managed by Inversion, fills it with data from a UNIX file, and closes it.
#include "tmp/c.h"
#include "tmp/libpq-fe.h"
#include "tmp/libpq-fs.h"
#include "catalog/pg_lobj.h"

#define MYBUFSIZ 1024

main()
{
    int inv_fd;
    int fd;
    char *qry_result;
    char buf[MYBUFSIZ];
    int nbytes;
    int tmp;

    PQsetdb("mydatabase");

    /* large object accesses must be */
    /* transaction-protected */
    qry_result = PQexec("begin");

    if (*qry_result == '\0') /* error */
        exit (1);

    /* open the UNIX file */
    fd = open("/my_unix_file", O_RDONLY, 0666);
    if (fd < 0) /* error */
        exit (1);

    /* create the Inversion file */
    inv_fd = p_creat("/inv_file", INV_WRITE, Inversion);
    if (inv_fd < 0) /* error */
        exit (1);

    /* copy the UNIX file to the Inversion */
    /* large object */
    while ((nbytes = read(fd, buf, MYBUFSIZ)) > 0)
    {
        tmp = p_write(inv_fd, buf, nbytes);
        if (tmp < nbytes) /* error */
            exit (1);
    }

    (void) close(fd);
    (void) close(inv_fd);

    /* commit the transaction */
    qry_result = PQexec("end");

    if (*qry_result == '\0') /* error */
}
exit (1);
/* by here, success */
exit (0);

BUGS

Shouldn't have to distinguish between Inversion and UNIX large objects when you open an existing large object. The system knows which implementation was used. The flags argument should be the same in these two cases.

All large object file names (paths) are limited to 256 characters.

In the Inversion file system, file name components (the sections of the path preceding, following or in between "/") are limited to 16 characters each. The maximum path length is still 256 characters.

The unlink routines do not always remove the underlying data files because they do not implement reference counts.

THE lo_filein() and lo_fileout() INTERFACE

As POSTGRES has evolved, the backend large object interface described above has replaced an earlier backend large object interface. The previous interface required users to store internal large object descriptors in their attributes; this worked, but required users to call internal POSTGRES routines directly in order to access their data. The interface documented above is clearer and more consistent, so the interface about to be described is deprecated and documented only for historical reasons.

The functions lo_filein and lo_fileout convert between UNIX filenames and internal large object descriptors. These functions are POSTGRES registered functions, meaning they can be used directly in POSTQUEL queries as well as from dynamically-loaded C functions.

The routine

```c
LargeObject *lo_filein(filename)
char *filename;
```

associates a new UNIX file storing large object data with the database system. This routine stores the filename in a abstract data structure suitable for inclusion as an attribute of a tuple.

The converse routine,

```c
char *lo_fileout(object)
LargeObject *object;
```

returns the UNIX filename associated with a large object.

If you are defining a simple large object ADT, these functions can be used as your "input" and "output" functions (see define type (commands)). A suitable declaration would be

```c
define type LargeObject (internallength = variable,
input = lo_filein, output = lo_fileout)
```

The file storing the large object must be accessible on the machine on which POSTGRES is running. The data is not copied into the database system, so if the file is later removed, it is unrecoverable.

The data in large objects imported in this manner are only accesible from the POSTGRES backend using
dynamically-loaded functions. However, the internal large object descriptors cannot be used with the 
*LOpen* backend interface. Instead, these descriptors can only be used by making direct calls to a set of 
undocumented routines within the POSTGRES storage manager. Furthermore, it becomes the user's respon-
sibility to make calls to the correct set of routines for UNIX or Inversion large objects.

SEE ALSO

introduction(commands), define function(commands), define type(commands), load(commands).
SECTION 8 — SYSTEM CATALOGS

DESCRIPTION
Thus far we have made many allusions to the system catalogs and their role in the POSTGRES extensibility architecture but have managed to avoid a systematic specification of their layout and contents. In this section we list each of the attributes of the system catalogs and define their meanings.

CLASS/TYPES SYSTEM CATALOGS
These catalogs form the core of the extensibility system:

<table>
<thead>
<tr>
<th>name</th>
<th>shared/local</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pg_aggregate</td>
<td>local</td>
<td>aggregate functions</td>
</tr>
<tr>
<td>pg_am</td>
<td>local</td>
<td>access methods</td>
</tr>
<tr>
<td>pg_amop</td>
<td>local</td>
<td>operators usable with specific access methods</td>
</tr>
<tr>
<td>pg_amproc</td>
<td>local</td>
<td>procedures used with specific access methods</td>
</tr>
<tr>
<td>pg_attribute</td>
<td>local</td>
<td>class attributes</td>
</tr>
<tr>
<td>pg_class</td>
<td>local</td>
<td>classes</td>
</tr>
<tr>
<td>pg_index</td>
<td>local</td>
<td>secondary indices</td>
</tr>
<tr>
<td>pg_inherits</td>
<td>local</td>
<td>class inheritance hierarchy</td>
</tr>
<tr>
<td>pg_language</td>
<td>local</td>
<td>procedure implementation languages</td>
</tr>
<tr>
<td>pg_opclass</td>
<td>local</td>
<td>operator classes</td>
</tr>
<tr>
<td>pg_operator</td>
<td>local</td>
<td>query language operators</td>
</tr>
<tr>
<td>pg_proc</td>
<td>local</td>
<td>procedures (functions)</td>
</tr>
<tr>
<td>pg_type</td>
<td>local</td>
<td>data types</td>
</tr>
</tbody>
</table>

ENTITIES
These catalogs deal with identification of entities known throughout the site:

<table>
<thead>
<tr>
<th>name</th>
<th>shared/local</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pg_database</td>
<td>shared</td>
<td>current databases</td>
</tr>
<tr>
<td>pg_group</td>
<td>shared</td>
<td>user groups</td>
</tr>
<tr>
<td>pg_user</td>
<td>shared</td>
<td>valid users</td>
</tr>
</tbody>
</table>

RULE SYSTEM CATALOGS

<table>
<thead>
<tr>
<th>name</th>
<th>shared/local</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pg_listener</td>
<td>local</td>
<td>processes waiting on alerters</td>
</tr>
<tr>
<td>pg_prs2plans</td>
<td>local</td>
<td>instance system procedures</td>
</tr>
<tr>
<td>pg_prs2rule</td>
<td>local</td>
<td>instance system rules</td>
</tr>
<tr>
<td>pg_prs2stub</td>
<td>local</td>
<td>instance system &quot;stubs&quot;</td>
</tr>
<tr>
<td>pg_rewrite</td>
<td>local</td>
<td>rewrite system information</td>
</tr>
</tbody>
</table>

LARGE OBJECT CATALOGS
These catalogs are specific to the Inversion file system and large objects in general:

<table>
<thead>
<tr>
<th>name</th>
<th>shared/local</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pg_lobj</td>
<td>local</td>
<td>description of a large object</td>
</tr>
<tr>
<td>pg_naming</td>
<td>local</td>
<td>Inversion name space mapping</td>
</tr>
<tr>
<td>pg_platter</td>
<td>local</td>
<td>jukebox platter inventory</td>
</tr>
<tr>
<td>pg_plmap</td>
<td>local</td>
<td>jukebox platter extent map</td>
</tr>
</tbody>
</table>

INTERNAL CATALOGS
These catalogs are internal classes that are not stored as normal heaps and cannot be accessed through normal means (attempting to do so causes an error).
There are several other classes defined with "pg_" names. Aside from those that end in "ind*" (secondary indices), these are all obsolete or otherwise deprecated.

**CLASS/TYP**E **SYSTEM CATALOGS**

The following catalogs relate to the class/type system.

```sql
/*
 * aggregates
 * see DEFINE AGGREGATE for an explanation of transition functions
 */
pg_aggregate
    char16  aggname  /* aggregate name (e.g., "count") */
    oid     aggowner /* usesysid of creator */
    regproc aggtransfn1 /* first transition function */
    regproc aggtransfn2 /* second transition function */
    regproc aggfinalfn /* final function */
    oid     aggbasetype /* type of data on which aggregate operates */
    oid     aggtransnippet1 /* type returned by aggtransfn1 */
    oid     aggtransnippet2 /* type returned by aggtransfn2 */
    oid     aggfinalnippet /* type returned by aggfinalfn */
    text    agginitval1 /* external format of initial (starting) value of aggtransfn1 */
    text    agginitval2 /* external format of initial (starting) value of aggtransfn2 */
```

```sql
pg_am
    char16  amname  /* access method name */
    oid     amowner /* usesysid of creator */
    char    amkind /* - deprecated */
    int2    amstrategies /* total NUMBER of strategies by which we can traverse/search this AM */
    int2    amsupport /* total NUMBER of support functions that this AM uses */
    regproc amgettuple /* "next valid tuple" function */
    regproc aminsert /* "insert this tuple" function */
    regproc amdelete /* "delete this tuple" function */
    regproc amgetattr /* - deprecated */
    regproc amsetlock /* - deprecated */
    regproc amsettid /* - deprecated */
```

03/13/94
### INTRODUCTION (SYSTEM CATALOGS)

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>regproc</td>
<td>amfreetuple /* - deprecated */</td>
</tr>
<tr>
<td>regproc</td>
<td>ambeginscan /* &quot;start new scan&quot; function */</td>
</tr>
<tr>
<td>regproc</td>
<td>amrescan /* &quot;restart this scan&quot; function */</td>
</tr>
<tr>
<td>regproc</td>
<td>amendscan /* &quot;end this scan&quot; function */</td>
</tr>
<tr>
<td>regproc</td>
<td>ammarkpos /* &quot;mark current scan position&quot; function */</td>
</tr>
<tr>
<td>regproc</td>
<td>amrestrpos /* &quot;restore marked scan position&quot; function */</td>
</tr>
<tr>
<td>regproc</td>
<td>amopen /* - deprecated */</td>
</tr>
<tr>
<td>regproc</td>
<td>amclose /* - deprecated */</td>
</tr>
<tr>
<td>regproc</td>
<td>ambuild /* &quot;build new index&quot; function */</td>
</tr>
<tr>
<td>regproc</td>
<td>amcreate /* - deprecated */</td>
</tr>
<tr>
<td>regproc</td>
<td>amdestroy /* - deprecated */</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pg_amop</td>
<td>oid amopid /* access method with which this operator be used */</td>
</tr>
<tr>
<td>pg_amop</td>
<td>oid amopclaid /* operator class with which this operator can be used */</td>
</tr>
<tr>
<td>pg_amop</td>
<td>oid amopopr /* the operator */</td>
</tr>
<tr>
<td>pg_amop</td>
<td>int2 amopstrategy /* traversal/search strategy number to which this operator applies */</td>
</tr>
<tr>
<td>regproc</td>
<td>amopselect /* function to calculate the operator selectivity */</td>
</tr>
<tr>
<td>regproc</td>
<td>amopnpages /* function to calculate the number of pages that will be examined */</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pg_amproc</td>
<td>oid amid /* access method with which this procedure is associated */</td>
</tr>
<tr>
<td>pg_amproc</td>
<td>oid amopclaid /* operator class with which this operator can be used */</td>
</tr>
<tr>
<td>pg_amproc</td>
<td>oid amproc /* the procedure */</td>
</tr>
<tr>
<td>pg_amproc</td>
<td>int2 amprocnvun /* support function number to which this operator applies */</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pg_class</td>
<td>char16 relname /* class name */</td>
</tr>
<tr>
<td>pg_class</td>
<td>oid relowner /* usesysid of owner */</td>
</tr>
<tr>
<td>pg_class</td>
<td>oid relam /* access method */</td>
</tr>
<tr>
<td>pg_class</td>
<td>int4 reltuples /* # of instances */</td>
</tr>
<tr>
<td>pg_class</td>
<td>int4 reltuples /* # of instances */</td>
</tr>
<tr>
<td>pg_class</td>
<td>abstime relextpires /* time after which instances are deleted from non-archival storage */</td>
</tr>
<tr>
<td>pg_class</td>
<td>reltime relpreserved /* timespan after which instances are deleted from non-archival storage */</td>
</tr>
<tr>
<td>pg_class</td>
<td>bool relhasindex /* does the class have a secondary */</td>
</tr>
</tbody>
</table>

03/13/94
### Introduction (System Catalogs)

<table>
<thead>
<tr>
<th>Type</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>relisshared</td>
<td>/* is the class shared or local? */</td>
</tr>
<tr>
<td>char</td>
<td>relkind</td>
<td>/* type of relation:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>i=index</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r=relation (heap)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>s=special</td>
</tr>
<tr>
<td></td>
<td></td>
<td>u=uncatalogued (temporary) */</td>
</tr>
<tr>
<td>char</td>
<td>relarch</td>
<td>/* archive mode:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>h=heavy</td>
</tr>
<tr>
<td></td>
<td></td>
<td>l=light</td>
</tr>
<tr>
<td></td>
<td></td>
<td>n=none */</td>
</tr>
<tr>
<td>int2</td>
<td>relnatts</td>
<td>/* current # of non-system attributes */</td>
</tr>
<tr>
<td>int2</td>
<td>relsmgr</td>
<td>/* storage manager:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0=magnetic disk</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1=sony WORM jukebox</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2=main memory */</td>
</tr>
<tr>
<td>int28</td>
<td>relkey</td>
<td>/* unused */</td>
</tr>
<tr>
<td>oid8</td>
<td>relkeyop</td>
<td>/* unused */</td>
</tr>
<tr>
<td>aclitem</td>
<td>relacl[1]</td>
<td>/* access control lists */</td>
</tr>
</tbody>
</table>

#### pg_attribute

<table>
<thead>
<tr>
<th>Type</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>oid</td>
<td>attrelid</td>
<td>/* class containing this attribute */</td>
</tr>
<tr>
<td>char16</td>
<td>attname</td>
<td>/* attribute name */</td>
</tr>
<tr>
<td>oid</td>
<td>atttypid</td>
<td>/* attribute type */</td>
</tr>
<tr>
<td>oid</td>
<td>attdefrel</td>
<td>/* deprecated */</td>
</tr>
<tr>
<td>int4</td>
<td>attvals</td>
<td>/* deprecated */</td>
</tr>
<tr>
<td>oid</td>
<td>atttyparg</td>
<td>/* deprecated */</td>
</tr>
<tr>
<td>int2</td>
<td>attlen</td>
<td>/* attribute length, in bytes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-1=variable */</td>
</tr>
<tr>
<td>int2</td>
<td>attnum</td>
<td>/* attribute number</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&gt;0=user attribute</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;0=system attribute */</td>
</tr>
<tr>
<td>int2</td>
<td>attbound</td>
<td>/* deprecated */</td>
</tr>
<tr>
<td>bool</td>
<td>attbyval</td>
<td>/* type passed by value? */</td>
</tr>
<tr>
<td>bool</td>
<td>attcanindex</td>
<td>/* deprecated */</td>
</tr>
<tr>
<td>oid</td>
<td>attproc</td>
<td>/* deprecated */</td>
</tr>
<tr>
<td>int4</td>
<td>attnelems</td>
<td>/* # of array dimensions */</td>
</tr>
<tr>
<td>int4</td>
<td>attcacheoff</td>
<td>/* cached offset into tuple */</td>
</tr>
<tr>
<td>bool</td>
<td>attisset</td>
<td>/* is attribute set-valued? */</td>
</tr>
</tbody>
</table>

#### pg_inherits

<table>
<thead>
<tr>
<th>Type</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>oid</td>
<td>inhrel</td>
<td>/* child class */</td>
</tr>
<tr>
<td>oid</td>
<td>inparent</td>
<td>/* parent class */</td>
</tr>
<tr>
<td>int4</td>
<td>inhseqno</td>
<td>/* deprecated */</td>
</tr>
<tr>
<td>oid</td>
<td>indexrelid</td>
<td>/* oid of secondary index class */</td>
</tr>
</tbody>
</table>
### INTRODUCTION(SYSTEM CATALOGS)

<table>
<thead>
<tr>
<th>oid</th>
<th>indrelid</th>
<th>/* oid of indexed heap class */</th>
</tr>
</thead>
<tbody>
<tr>
<td>oid</td>
<td>indproc</td>
<td>/* function to compute index key from attribute(s) in heap */</td>
</tr>
<tr>
<td>int28</td>
<td>indkey</td>
<td>/* attribute numbers of key attribute(s) */</td>
</tr>
<tr>
<td>oid8</td>
<td>indclass</td>
<td>/* opclass of each key */</td>
</tr>
<tr>
<td>bool</td>
<td>indisclastered</td>
<td>/* is the index clustered? */</td>
</tr>
<tr>
<td></td>
<td>indisarchived</td>
<td>/* is the index archival? */</td>
</tr>
<tr>
<td>text</td>
<td>indpred</td>
<td>/* query plan for partial index predicate */</td>
</tr>
</tbody>
</table>

### pg_type

<table>
<thead>
<tr>
<th>char16</th>
<th>typename</th>
<th>/* type name */</th>
</tr>
</thead>
<tbody>
<tr>
<td>oid</td>
<td>typowner</td>
<td>/* usesysid of owner */</td>
</tr>
<tr>
<td>int2</td>
<td>typplen</td>
<td>/* length in internal form */</td>
</tr>
<tr>
<td></td>
<td></td>
<td>-1=variable-length */</td>
</tr>
<tr>
<td>int2</td>
<td>typbyrntenlen</td>
<td>/* length in external form */</td>
</tr>
<tr>
<td>bool</td>
<td>typbyval</td>
<td>/* type passed by value? */</td>
</tr>
<tr>
<td>char</td>
<td>typbyval</td>
<td>/* kind of type: */</td>
</tr>
<tr>
<td></td>
<td></td>
<td>c=catalog (composite)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>b=base */</td>
</tr>
<tr>
<td>bool</td>
<td>typisdefined</td>
<td>/* defined or still a shell? */</td>
</tr>
<tr>
<td>char</td>
<td>typdelim</td>
<td>/* delimiter for array external form */</td>
</tr>
<tr>
<td>oid</td>
<td>typrelid</td>
<td>/* class (if composite) */</td>
</tr>
<tr>
<td>oid</td>
<td>typelem</td>
<td>/* type of each array element */</td>
</tr>
<tr>
<td>regproc</td>
<td>typinput</td>
<td>/* external-internal conversion function */</td>
</tr>
<tr>
<td>regproc</td>
<td>typoutput</td>
<td>/* internal-external conversion function */</td>
</tr>
<tr>
<td>regproc</td>
<td>typreceive</td>
<td>/* client-server conversion function */</td>
</tr>
<tr>
<td>regproc</td>
<td>typsend</td>
<td>/* server-client conversion function */</td>
</tr>
<tr>
<td>text</td>
<td>typdefault</td>
<td>/* default value */</td>
</tr>
</tbody>
</table>

### pg_operator

<table>
<thead>
<tr>
<th>char16</th>
<th>oprname</th>
<th>/* operator name */</th>
</tr>
</thead>
<tbody>
<tr>
<td>oid</td>
<td>oprrowner</td>
<td>/* usesysid of owner */</td>
</tr>
<tr>
<td>int2</td>
<td>oprrprec</td>
<td>/* - deprecated */</td>
</tr>
<tr>
<td>char</td>
<td>oprrkind</td>
<td>/* kind of operator: */</td>
</tr>
<tr>
<td></td>
<td></td>
<td>b=binary</td>
</tr>
<tr>
<td></td>
<td></td>
<td>l=left unary</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r=right unary */</td>
</tr>
<tr>
<td>bool</td>
<td>oprisleft</td>
<td>/* is operator left/right associative? */</td>
</tr>
<tr>
<td>bool</td>
<td>oprcanhash</td>
<td>/* is operator usable for hashjoin? */</td>
</tr>
<tr>
<td>oid</td>
<td>oprleft</td>
<td>/* left operand type */</td>
</tr>
<tr>
<td>oid</td>
<td>oprright</td>
<td>/* right operand type */</td>
</tr>
</tbody>
</table>
### Introduction (System Catalogs)

#### oid
- oprresult /* result type */
- oprcom /* commutator operator */
- oprnegate /* negator operator */
- oprlsortop /* sort operator for left operand */
- oprrsortop /* sort operator for right operand */
- oprcode /* function implementing this operator */
- oprrest /* function to calculate operator restriction selectivity */
- oprjoin /* function to calculate operator join selectivity */

#### char16
- opcname /* operator class name */

#### oid
- proname /* function name */
- proowner /* usesysid of owner */
- prolang /* function implementation language */
- proisinh /* deprecated */
- proistrusted /* run in server or untrusted function process? */
- proiscachable /* can the function return values be cached? */
- pronargs /* # of arguments */
- proretset /* does the function return a set? */
- prorettype /* return type */
- proargtypes /* argument types */
- probyte_pct /* % of argument size (in bytes) that needs to be examined in order to compute the function */
- properbyte_cpu /* sensitivity of the function's running time to the size of its inputs */
- propercall_cpu /* overhead of the function's invocation (regardless of input size) */
- prooutin_ratio /* size of the function's output as a percentage of the size of the input */
- prosrc /* function definition (postquel only) */
- probin /* path to object file (C only) */

#### char16
- lanname /* language name */
- lancompiler /* deprecated */
INTRODUCTION (SYSTEM CATALOGS)

ENTITIES

pg_database

  char16  datname       /* database name */
  oid     datdba        /* usesysid of database administrator */
  text    datapath      /* directory of database under $PGDATA */

pg_group

  char16  groname       /* group name */
  int2    grosysid      /* group's UNIX group id */
  int2    grolist[1]    /* list of usesysids of group members */

pg_user

  char16  usename       /* user's name */
  int2    usesysid      /* user's UNIX user id */
  bool    usecreatedb   /* can user create databases? */
  bool    usetrace      /* can user set trace flags? */
  bool    usesuper      /* can user be POSTGRES superuser? */
  bool    usecatupd     /* can user update catalogs? */

RULE SYSTEM CATALOGS

pg_listener

  char16  relname       /* class for which asynchronous notification is desired */
  int4    listenerpid   /* process id of server corresponding to a frontend program waiting for asynchronous notification */
  int4    notification  /* whether an event notification for this process id still pending */

pg_prs2rule

  char16  prs2name      /* rule name */
  char    prs2eventtype /* rule event type: */
  oid     prs2eventrel  /* class to which event applies */
  int2    prs2eventattr /* attribute to which event applies */
  float8  necessary     /* - deprecated */
  float8  sufficient    /* - deprecated */
  text    prs2text      /* text of original rule definition */

pg_prs2plans

  oid     prs2ruleid    /* prs2rule instance for which this
INTRODUCTION(SYSTEM CATALOGS)

int2  prs2planno  /* plan number (one rule may invoke multiple plans) */
text  prs2code    /* external representation of the plan */

pg_prs2stub
oid  prs2relid  /* class to which this rule applies */
bool prs2islast /* is this the last stub fragment? */
int4 prs2no     /* stub fragment number */
stub prs2stub   /* stub fragment */

pg_rewrite
char16 rulename /* rule name */
char  ev_type   /* event type: RETRIEVE, REPLACE, APPEND, DELETE codes are parser-dependent (1?) */
oid  ev_class  /* class to which this rule applies */
int2 ev_attr   /* attribute to which this rule applies */
bool is_instead /* is this an "instead" rule? */
text ev_qual   /* qualification with which to modify (rewrite) the plan that triggered this rule */
text action    /* parse tree of action */

LARGE OBJECT CATALOGS

pg_lobj
oid  ourid  /* 'ourid' from pg_naming that identifies this object in the Inversion file system namespace */
int4 objtype /* storage type code:
  0=Inversion
  1=Unix
  2=External
  3=Jaquith */
bytea object_descripto/* opaque object-handle structure */

pg_naming
char16 filename /* filename component */
oid  ourid  /* random oid used to identify this instance in other instances (can't use the actual oid for obscure reasons */
oid  parentid /* pg_naming instance of parent Inversion file system directory */

pg_platter

03/13/94  164
INTRODUCTION (SYSTEM CATALOGS)

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>char16</td>
<td>plname</td>
<td>/* platter name */</td>
</tr>
<tr>
<td>int4</td>
<td>pled</td>
<td>/* the highest OCCUPIED extent */</td>
</tr>
<tr>
<td>pg_plmap</td>
<td>plid</td>
<td>/* platter (in pg_platter) on which this extent (of blocks) resides */</td>
</tr>
<tr>
<td></td>
<td>pldbid</td>
<td>/* database of the class to which this extent (of blocks) belongs */</td>
</tr>
<tr>
<td></td>
<td>plrelid</td>
<td>/* class to which this extend (of blocks) belongs */</td>
</tr>
<tr>
<td>int4</td>
<td>plblkno</td>
<td>/* starting block number within the class */</td>
</tr>
<tr>
<td>int4</td>
<td>ploffset</td>
<td>/* offset within the platter at which this extent begins */</td>
</tr>
<tr>
<td>int4</td>
<td>plextentsz</td>
<td>/* length of this extent */</td>
</tr>
</tbody>
</table>
SECTION 8 — FILES

OVERVIEW

This section describes some of the important files used by POSTGRES.

NOTATION

".../" at the front of file names represents the path to the postgres user's home directory. Anything in square brackets ("[" and "]") is optional. Anything in braces ("{" and "}") can be repeated 0 or more times. Parentheses ("(" and ")") are used to group boolean expressions. | is the boolean operator OR.

BUGS

The descriptions of

\[ .../data/PG_VERSION, \]
\[ .../data/base/*/PG_VERSION, \]

the temporary sort files, and the database debugging trace files are absent.
NAME

.../src/backend/obj/{local,global}.bki — template scripts

DESCRIPTION

Backend Interface (BKI) files are scripts that describe the contents of the initial POSTGRES database. This database is constructed during system installation, by the *initdb* command. *Initdb* executes the POSTGRES backend with a special set of flags, that cause it to consume the BKI scripts and bootstrap a database.

These files are automatically generated from system header files during installation. They are not intended for use by humans, and you do not need to understand their contents in order to use POSTGRES. These files are copied to

```plaintext
.../files/{global,local}.bki
```
during system installation.

All new user databases will be created by copying the template database that POSTGRES constructs from the BKI files. Thus, a simple way to customize the template database is to let the POSTGRES initialization script create it for you, and then to run the terminal monitor to make the changes you want.

The POSTGRES backend interprets BKI files as described below. This description will be easier to understand if the example in ".../files/global.bki" is at hand.

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, 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

```plaintext
define macro macro_name = macro_value
```

and are undefined using

```plaintext
undefine macro macro_name
```

and redefined using the same syntax as define.

Lists of general commands and macro commands follow.

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 "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 index-name on class-name using am-name

Create an index named index-name on the class named class-name using the am-name access method.
The fields to index are called name1, name2, etc., and the operator collections to use are collection_1, collection_2, etc., respectively.

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.

EXAMPLE

The following set of commands will create the "pg_opclass" class containing the int_ops collection as
object 421, print out the class, and then close it.

create pg_opclass (opcname=char16)
open pg_opclass
insert oid=421 (int_ops)
print
close pg_opclass

SEE ALSO

initdb(unix), createdb(unix), createdb(commands), template(files).
NAME
page structure — POSTGRES database file default page format

DESCRIPTION
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.

Diagram 1 shows how pages in both normal POSTGRES classes and POSTGRES index classes (e.g., a B-tree index) are structured.

![Sample Page Layout Diagram](image)

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.

**FILES**

.../data/...
Location of shared (global) database files.

.../data/base/...
Location of local database files.

**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.
NAME

.../data/files/global1.bki — global database template
.../data/files/local1_XXX.bki — local database template
.../data/files/template1/* — default database template

DESCRIPTION

These files contain scripts which direct the construction of databases. Note that the “global1.bki” and “template1_local.bki” files are installed automatically when the POSTGRES super-user runs initdb. These files are copied from

.../src/backend/obj/({dbdb,local}).bki

The databases which are generated by the template scripts are normal databases. Consequently, you can use the terminal monitor or some other frontend on a template database to simplify the customization task. That is, there is no need to express everything about your desired initial database state using a BKI template script, because the database state can be tuned interactively.

The system catalogs consist of classes of two types: global and local. There is one copy of each global class that is shared among all databases at a site. Local classes, on the other hand, are not accessible except from their own database.

The file

.../data/files/global1.bki

specifies the process used in the creation of global (shared) classes by createdb. Similarly, the

.../files/local1_XXX.bki

files specify the process used in the creation of local (unshared) catalog classes for the “XXX” template database. “XXX” may be any string of 16 or fewer printable characters. If no template is specified in a createdb command, then the template in

.../files/local1_template1.bki

is used.

The .bki files are generated from C source code by an inscrutable set of AWK scripts.

BUGS

POSTGRES Version 4.2 does not permit users to have separate template databases.

SEE ALSO

bki(files), initdb(unix), createdb(unix).
REFERENCES

The following technical reports are referred to in this document. For information on ordering technical reports, see the installation notes that accompany the POSTGRES distribution.


