SECTION 1 — INTRODUCTION

OVERVIEW

This document is the reference for the POSTGRES database system under development at the University of California, Berkeley. It is intended to be a supplement to the POSTGRES Manual, which is included in this distribution. The POSTGRES project, led by Professor Michael Stonebraker is 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. Moreover, 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 4.0 of POSTGRES. The POSTGRES software is about 200,000 lines of C code, and is available for SUN 4 class machines, for DECstation 3100 and 5000 machines and for the SEQUENT Symmetry machine. Information on obtaining the source code for these computers is available from:

Chandra Ghosh
Computer Science Division
521 Evans Hall
University of Califiornia
Berkeley, CA 94720
(510) 642-4662

Version 4.0 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.

POSTGRES DOCUMENTATION

This reference describes the functionality of Version 4.0 and contains notations where appropriate to indicate which features are not implemented in Version 4.0. Application developers should note that this reference contains only the specification for the low-level call-oriented application program interface, LIBPQ.

The remainder of this reference 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 POSTGRES with very high performance. Section 7 (LARGE OBJECTS) describes the internal POSTGRES interface for accessing large objects. The reference concludes with Section 8 (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 contributors (in alphabetical order) consisted of James Bell, Jennifer Caetta, Jolly Chen, Ron Choi, Jeffrey Goh, Joey Hellerstein, Wei Hong, Anant Jhingran, Greg Kemnitz, Case Larsen Jeff Meredith, Michael Olson, Lay-Peng Ong, Spyros Potamianos, Sunita Sarawagi, and Cimarron Taylor.

For Version 4.0, Jeff Meredith served as chief programmer and was responsible for overall coordination of the project and for individually implementing the "everything else" portion of the system.

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

FOOTNOTES

UNIX is a trademark of AT&T.

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 which 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. But 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 is the owner of the POSTGRES binaries and database files. As the 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 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**(unix).

The *postmaster is a process which acts as a clearing house for requests to* the POSTGRES system. Basically, frontend applications connect with the postmaster which keeps tracks of any system errors and communication between the backend processes. The postmaster (POSTMASTER (UNIX)) takes from zero to seven arguments to tune its behavior. Supplying arguments is necessary only if you intend to run multiple sites or a non-default site.

The *POSTGRES backend* (.../bin/postgres) may be executed directly from the 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 file names is used to represent the path to the postgres 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. “|” is the boolean operator OR.

USING POSTGRES FROM UNIX

All POSTGRES commands which 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.

There is a collection of system catalogs that exist at each site. These include a USER class which contains an instance for each valid POSTGRES user. In the instance is a collection of POSTGRES privileges, the most relevant of which is whether or not creation of POSTGRES databases is allowed. A UNIX user can do nothing at all with POSTGRES until an appropriate record is installed in this system catalog class. Further information on the system catalogs is available by running queries on the appropraiate classes.

NAME

createdb — create a database

SYNOPSIS

createdb [**-p** port] [**-h** host] username

DESCRIPTION

Createdb creates a new database. The person who executes this command becomes the database administrator (DBA) for this database. The DBA has special powers not granted to ordinary users. Namely, they can destroy the database they created.

Dbname is the name of the database to be created. The name must be unique among all POSTGRES databases.

The argument *port* and *hostname* are the same as in the terminal monitor - they are used to connect to the postmaster using the TCP/IP port *port* running on the database server *hostname.* The defaults are to the local machine (localhost) and to the default port (4321).

SEE ALSO

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

DIAGNOSTICS

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

createdb could not attach to the postmaster on the specified host and port. If you see this message, check that the postmaster is running on the proper host and that the proper port is specified.

You are not a valid POSTGRES user

You do not have a users file entry, and can not do anything with POSTGRES at all.

<dbname> **already exists**

The database already exists.

NAME

createuser — create a POSTGRES user

SYNOPSIS

createuser [**-p** port] [**-h** host] 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.

Username is the name of the POSTGRES user to be created. The name must be unique among all POSTGRES users.

The arguments *port* and *hostname* are the same as in the terminal monitor - they are used to connect to the postmaster using the TCP/IP port *port* running on the database server *hostname.* The defaults are to the local machine (localhost) and to the default port (4321).

INTERACTIVE QUESTIONS

Once invoked with the above options **createuser** will guide the person adding the new POSTGRES user through a series of questions. These questions describe the security capabilities of the POSTGRES user. The new user’s POSTGRES userid must be the user’s Unix userid.

SEE ALSO

destroyuser(unix).

DIAGNOSTICS

You are not a valid POSTGRES user

You do not have a users file entry, and can not do anything with POSTGRES at all.

<user> **already exists**

The user already exists.

BUGS

POSTGRES userid’s and usernames should not have anything to do with the constraints of UNIX.

NAME

destroydb — destroy an existing database

SYNOPSIS

destroydb [**-p** port] [**-h** host] username

DESCRIPTION

Destroydb removes all reference to an existing database named *dbname.* The directory containing this database and all associated files are removed.

To execute this command, the user must be the DBA for this database. After the database is destroyed, a UNIX shell prompt will reappear; no confirmation message will be displayed.

Destroydb needs to connect to a running postmaster to accomplish its tasks. If no postmaster is running then one must be started before destroydb is run.

COMMAND OPTIONS

-p port indicates that destroydb should attempt to connect to a postmaster listening to the specified port.

-h hostname indicates that destroydb should attempt to connect to a postmaster running on the specified host machine.

EXAMPLE

```
/* destroy the demo database */
destroydb demo
```

```
/* destroy the demo database using the postmaster on host
eden, port 1234 */
destroydb -p 1234 -h eden demo
```

DIAGNOSTICS

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

destroydb could not attach to the postmaster on the specified host and port. If you see this message, check that the postmaster is running on the proper host and that the proper port is specified.

FILES

```
.../data/base/*
```

SEE ALSO

createdb(unix), postmaster(unix), destroydb(commands).

NAME

destroyuser — destroy a POSTGRES user

SYNOPSIS

destroyuser [**-p** port] [**-h** host] username

DESCRIPTION

Destroyuser destroys an existing POSTGRES user. Only users with ‘‘usesuper’’ set in the pg_user class can destroy POSTGRES users. As shipped, the user ‘‘postgres’’ can destroy users.

Username is the name of the POSTGRES user to be destroyed.

The argument *port* and *hostname* are the same as in the terminal monitor - they are used to connect to the postmaster using the TCP/IP port *port* running on the database server *hostname.* The defaults are to the local machine (localhost) and to the default port (4321).

INTERACTIVE QUESTIONS

Once invoked with the above options **destroyuser** will warn the person destroying the POSTGRES user about the databases that will be destroyed in the process. If the databases should not be destroyed, **destroyuser** can be aborted.

SEE ALSO

createuser(unix).

DIAGNOSTICS

You are not a valid POSTGRES user

You do not have a users file entry, and can not do anything with POSTGRES at all.

<user> **does not exist**

The user does not exist.

NAME

initdb — initalize the database templates and primary directories

SYNOPSIS

initdb [-v]

DESCRIPTION

initdb sets up the initial template databases. It is normally executed as part of the installation process. The template database is created under the directory specified by the the environment variable, POSTGRESHOME. For example,

```
setenv POSTGRESHOME /usr/postgres
```

POSTGRES also supports data striping by allowing a database spread across multiple directories. The user can specify multiple directories in POSTGRESHOME separated by

```
setenv POSTGRESHOME /usr1/postgres:/usr2/postgres
```

The *-v* option specifies that **initdb** should be run in "verbose mode", meaning that it will print messages stating where the directories are being created, etc.

SEE ALSO

createdb(unix).

NAME

ipcclean — clean up shared memory and semaphores from aborted backends

SYNOPSIS

ipcclean

DESCRIPTION

Ipcclean cleans up shared memory and semaphore space from aborted backends. Only the DBA should execute this program, as it can cause bizarre behavior if run during multi-user execution. This program should be ran if errors such as **semget: No space left on device** are encountered in starting up programs like the Postmaster or POSTGRES backend.

BUGS

If this command is run while a Postmaster or backend is running, the shared memory and semaphores allocated by the postmaster will be deleted. This will result in a general failure of the backends which are currently running.

NAME

monitor — run the interactive terminal monitor

SYNOPSIS

monitor [-h hostname] [-p port] [-t tty_device] [-N] [-T]
[-c query] [-d path] [-q] [-o options] 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 called to edit the **query buffer**, which the terminal monitor manages. The editor used is determined by the value of the EDITOR environment variable. If EDITOR is not set, then **vi** is used by default.

The terminal monitor requires that the postmaster be running, and the ports (specified with the "-p" option or by the PGPORT environment variable) must be identical to those specified to the postmaster.

COMMAND OPTIONS

-h host specifies host machine on which the POSTGRES backend is running; default is your local machine (localhost).

-p port specifies the well known TCP/IP port used for network communication between the terminal monitor and the postmaster.

-t tty_device specifies the path name to the tty device (or regular UNIX file) which you want the backend debugging messages to be sent to; default is /dev/null. *-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.

-T specifies that attribute names will not be printed - only the data itself. This is useful in conjunction with the **-c** option in shell scripts.

-c query specifies that the monitor is to run one query and exit. This is useful for shell scripts, typically in conjunction with the **-N** and **-T** options. Examples of shell scripts in the POSTGRES distribution using **monitor**-c are createdb, destroydb, createuser, and destroyuser.

-d path specifies the path name of the file or tty which you want the frontend debugging messages to be written to; the default is not to generate any debugging messages.

-q specifies that the monitor should do its work quietly. By default, it prints welcome and exit messages and the queries it sends to the backend. If the *-q* flag is used, none of this happens.

-o options specifies additonal options for the POSTGRES backend. This is only intended

for use by POSTGRES developers.

You may set environment variables to avoid typing 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 gives the current date and time, usually followed by the information in the **dayfile** (files).

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 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 current time
\w *filename*	Store the query buffer to an external file
\\	Produce a single backslash at the current location in query buffer

ENVIRONMENT VARIABLES

You may set environment variables to avoid specifying command line options. These are as follows:

hostname:	PGHOST
port:	PGPORT
tty:	PGTTY
options:	PGOPTION

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

RETURN VALUE

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

SEE ALSO

backend(unix), postmaster(unix), createdb(unix), destroydb(unix), createuser(unix), destroyuser(unix).

NAME

pagedoc − POSTGRES page doctor

SYNOPSIS

pagedoc [−*h*|*b*|*r*] [−*d level*] *filename*

DESCRIPTION

The **pagedoc** program understands the layout of data on POSTGRES pages, and can be used to view contents of relations in case a database gets 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.

ARGUMENTS

−*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.

filename
    The name of the file containing the relation.

EXAMPLES

To print page and line pointer summaries and tuple headers for a btree index named *pg_typeidind*,

```
pagedoc −b −d2 pg_typeidind
```

To show the default (level zero) summary of a heap relation named *pg_user*,

```
pagedoc pg_user
```

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

postgres — run the Postgres backend directly

SYNOPSIS

postgres [**-Q**] [databasename] [**-d** debug_level]

DESCRIPTION

This command executes the POSTGRES backend directly. 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.

COMMAND OPTIONS

-*Q* indicates "Quiet" mode. By default, the POSTGRES backend prints the parse tree generated by the parser, the plan generated by the planner and many debugging message. Specifying this flag eliminates much of this.

databasename is the name of the database to be used. If this is not specified, databasename defaults to the value of the environment variable USER.

UNDOCUMENTED 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.

-*O* indicates that the backend should not use the transaction system. All commands run in the same transaction and all commands can see the results of prior commands.

-*M nnn* indicates that the backend should fork *nnn* slave backend processes and then execute queries in parallel. This is only useful on shared-memory multiprocessor systems (e.g. Sequent) and is still in experimental and research stage. It is especially unreliable for large joins. Full support for intra-query as well inter-query parallelism have been planned for Version 4.

-*S* indicates that the transaction system can run with the assumption of stable main memory thus 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 -S is specified in the absence of stable main memory.

-*s* indicates that time information and other statistics are to be displayed at the end of each query. This is useful for benchmarking or for use in tuning the number of buffers.

-*B nnn* indicates the number of shared buffers to use. The default number of buffers is 64.

-s POSTGRES will report execution statistics such as elapse time, buffer hit rate, etc for each query execution.

-L this flag will turn off locking.

DIAGNOSTICS

semget: No space left on device

If you see this message, you should run the *ipcclean* command. After doing this, try running POSTGRES again. If this still doesn't work, you will need to configure your kernel for shared memory and semaphores as described in the installation notes.

SEE ALSO

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

NAME

postmaster — run the POSTGRES postmaster

SYNOPSIS

postmaster [-p port] [-b backend_pathname] [-d debug_level] [-s] [-n] &

DESCRIPTION

The postmaster manages the communication between frontends and backends, as well as allocating the shared buffer pool and semaphores (on machines without TAS). The postmaster does not itself interact with the user so it should be started as a background process. **Only one postmaster should be run on a machine!**

COMMAND OPTIONS

port is the well known TCP/IP port used for network communication between a libpq application and the backend. If you specify a port other than the default port then you must specify the same port when starting any libpq application including the terminal monitor. Alternatively you may set the environment variable PGPORT to the specified port and all libpq applications will use it instead of the default.

backend_pathname is the full pathname of the POSTGRES backend you wish to use.

deug_level determines the amount of debugging output the backend will produce. Specifying any level will cause the postmaster to print out a few terse debugging output messages to the tty on which it was started.

The −*s* and −*n* options control the behavior of the postmaster when a backend dies abnormally. The ordinary strategy for this situation is to notify all other backends that they must terminate, and to reinitialize shared memory and semaphores. This is because an errant backend, before dumping core, could have contaminated some shared state.

If the −*s* option is supplied, then the postmaster will stop all other backends, but will not cause them to terminate. This permits system programmers to collect core dumps from all concurrent backends by hand.

If the −*n* command-line option is supplied, then the postmaster does not reinitialize shared data structures. A knowledgable system programmer can use the *shmemdoc* program to examine shared memory and semaphore state, in order to debug the problem.

Neither −*s* nor −*n* is intended for use in ordinary operation.

EXAMPLES

```
postmaster &
```

This command will start up a postmaster on the default port (4321) and will expect to use the default path to the POSTGRES backend ($POSTGRESHOME/bin/postgres) or /usr/postgres/bin/postgres. This is the simplest and most common way to start the postmaster.

```
postmaster -p 1234 -b /a/postgres/bin/postgres &
```

This command will start up a postmaster communicating through the port 1234, and will expect to use the backend located at /a/postgres/bin/postgres. Note: to connect to this postmaster using the terminal monitor, you would need to specify **-p 1234** on the command line invoking the terminal monitor.

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 will need to configure your kernel for shared memory and semaphores as described in the installation notes.

StreamServerPort: cannot bind to port

If you see this message, you should be certain that there is no other postmaster program already running. The easiest way to determine this is by the command "ps -ax | grep postmaster". If you are sure there is no other postmaster 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 should simply wait until the operating system closes the port.

SEE ALSO

postgres(unix), monitor(unix), ipcclean(unix), shmemdoc(unix).

NAME

shmemdoc − POSTGRES shared memory doctor

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 shared structures. This program is intended only for debugging POSTGRES, and should not be used in normal operation.

When some backend dies abnormally, the postmaster normally reinitializes shared memory and semaphores, and forces all peers of the dead backend to exit. If the 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.

A simple command interpreter reads user commands from standard input and prints results on standard output. The available commands, and their actions, 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 maually after a crash, and you don’t want to do it.

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

help Print a brief command summary.

quit Exit *shmemdoc*.

ARGUMENTS

−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.

BUGS

Probably doesn't work on anything but DECstations.

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.

SECTION 3 — WHAT COMES WITH POSTGRES (BUILT-INS)

DESCRIPTION

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.

POSTGRES Type	Meaning	Required
abstime	absolute date and time	*
bool	boolean	*
box	2-dimensional rectangle	
bytea	variable length array of bytes	
char	character	*
char16	array of 16 characters	*
cid	command identifier type	*
int2	two-byte signed integer	*
int28	array of 8 int2	*
int4	four-byte signed integer	*
float4	single-precision floating-point number	*
float8	double-precision floating-point number	*
lseg	2-dimensional line segment	
oid	object identifier type	*
oid8	array of 8 oid	*
path	variable-length array of lseg	
point	2-dimensional geometric point	
regproc	registered procedure	*
reltime	relative date and time	*
text	variable length array of characters	
tid	tuple identifier type	*
tinterval	time interval	*
uint2	two-byte unsigned integer	*
uint4	four-byte unsigned integer	*
xid	transaction identifier type	*

These types all have obvious formats except for the three time types, explained in the following.

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 1970, 1971, ..., 2038

Valid dates are, therefore, Jan 1 00:00:00 1970 GMT to Jan 1 00:00:00 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 "now" is provided as a convenience. The special absolute time ‘‘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]
[ , abstime]
[abstime, ""]
["", ""]
```

where *abstime* is a time in the absolute time format. "" will cause the time interval to either start or end at the least or greatest time allowable, that is, either Jan 1 00:00:00 1902 or Jan 1 00:00:00 2038, respectively.

OPERATORS

POSTGRES provides a large number of built-in operators on system types. These operators are declared in the system catalog *pg_operator*. Every entry in *pg_operator* includes the 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

```
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

```
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 POSTGRES function prototypes and they are **not equivalent** to C function prototypes.)

Type	Operator	POSTGRES Function Prototype	Operation
abstime	!=	bool abstimene(abstime, abstime)	inequality
	+	abstime timepl(abstime, reltime)	addition
	−	abstime timemi(abstime, reltime)	subtraction
	<=	bool abstimele(abstime, abstime)	less or equal
	<?>	bool ininterval(abstime, tinterval)	abstime in tinterval?
	<	bool abstimelt(abstime, abstime)	less than
	=	bool abstimeeq(abstime, abstime)	equality
	>=	bool abstimege(abstime, abstime)	greater or equal
	>	bool abstimegt(abstime, abstime)	greater than
bool	=	bool booleq(bool, bool)	equality
box	&&	bool box_overlap(box, box)	boxes overlap

	&<	bool box_overleft(box, box)	box A overlaps box B, but does not extend to right of box B
	&>	bool box_overright(box, box)	box A overlaps box B, but does not extend to left of box B
	<<	bool box_left(box, box)	A is left of B
	<=	bool box_le(box, box)	area less or equal
	<	bool box_lt(box, box)	area less than
	=	bool box_eq(box, box)	area equal
	>=	bool box_ge(box, box)	area greater or equal
	>>	bool box_right(box, box)	A is right of B
	>	bool box_gt(box, box)	area greater than
	@	bool box_contained(box, box)	A is contained in B
	˜=	bool box_same(box, box)	box equality
	˜	bool box_contain(box, box)	A contains B
char16	!=	bool char16ne(char16, char16)	inequality
	!˜	bool char16regexne(char16, char16)	A does not match regular expression B (POSTGRES uses the libc regexp calls for this operation)
	<=	bool char16le(char16, char16)	less or equal
	<	bool char16lt(char16, char16)	less than
	=	bool char16eq(char16, char16)	equality
	>=	bool char16ge(char16, char16)	greater or equal
	>	bool char16gt(char16, char16)	greater than
	˜	bool char16regexeq(char16, char16)	A matches regular expression B (POSTGRES uses the libc regexp calls for this operation)
char	!=	bool charne(char, char)	inequality
	*	bool charmul(char, char)	multiplication
	+	bool charpl(char, char)	addition
	−	bool charmi(char, char)	subtraction
	/	bool chardiv(char, char)	division
	<=	bool charle(char, char)	less or equal
	<	bool charlt(char, char)	less than
	=	bool chareq(char, char)	equality
	>=	bool charge(char, char)	greater or equal
	>	bool chargt(char, char)	greater than
float4	!=	bool float4ne(float4, float4)	inequality
	*	float4 float4mul(float4, float4)	multiplication
	+	float4 float4pl(float4, float4)	addition
	−	float4 float4mi(float4, float4)	subtraction
	/	float4 float4div(float4, float4)	division
	<=	bool float4le(float4, float4)	less or equal
	<	bool float4lt(float4, float4)	less than
	=	bool float4eq(float4, float4)	equality
	>=	bool float4ge(float4, float4)	greater or equal
	>	bool float4gt(float4, float4)	greater than

float8	!=	bool float8ne(float8, float8)	inequality
	*	float8 float8mul(float8, float8)	multiplication
	+	float8 float8pl(float8, float8)	addition
	−	float8 float8mi(float8, float8)	subtraction
	/	float8 float8div(float8, float8)	division
	<=	bool float8le(float8, float8)	less or equal
	<	bool float8lt(float8, float8)	less than1
	=	bool float8eq(float8, float8)	equality
	>=	bool float8ge(float8, float8)	greater or equal
	>	bool float8gt(float8, float8)	greater than
	^	float8 dpow(float8, float8)	exponentiation
int2	!=	bool int2ne(int2, int2)	inequality
	!=	int4 int24ne(int2, int4)	inequality
	%	int2 int2mod(int2, int2)	modulus
	%	int4 int24mod(int2, int4)	modulus
	*	int2 int2mul(int2, int2)	multiplication
	*	int4 int24mul(int2, int4)	multiplication
	+	int2 int2pl(int2, int2)	addition
	+	int4 int24pl(int2, int4)	addition
	−	int2 int2mi(int2, int2)	subtraction
	−	int4 int24mi(int2, int4)	subtraction
	/	int2 int2div(int2, int2)	division
	/	int4 int24div(int2, int4)	division
	<=	bool int2le(int2, int2)	less or equal
	<=	int4 int24le(int2, int4)	less or equal
	<	bool int2lt(int2, int2)	less than
	<	int4 int24lt(int2, int4)	less than
	=	bool int2eq(int2, int2)	equality
	=	int4 int24eq(int2, int4)	equality
	>=	bool int2ge(int2, int2)	greater or equal
	>=	int4 int24ge(int2, int4)	greater or equal
	>	bool int2gt(int2, int2)	greater than
	>	int4 int24gt(int2, int4)	greater than
		int2 int2inc(int2)	increment
int4	!!=	bool int4notin(int4, char16)	This is the relational “not in” operator, and is not intended for public use.
	!=	bool int4ne(int4, int4)	inequality
	!=	int4 int42ne(int4, int2)	inequality
	%	int4 int42mod(int4, int2)	modulus
	%	int4 int4mod(int4, int4)	modulus
	*	int4 int42mul(int4, int2)	multiplication
	*	int4 int4mul(int4, int4)	multiplication
	+	int4 int42pl(int4, int2)	addition
	+	int4 int4pl(int4, int4)	addition
	−	int4 int42mi(int4, int2)	subtraction
	−	int4 int4mi(int4, int4)	subtraction

	/	int4 int42div(int4, int2)	division
	/	int4 int4div(int4, int4)	division
	<=	bool int4le(int4, int4)	less or equal
	<=	int4 int42le(int4, int2)	less or equal
	<	bool int4lt(int4, int4)	less than
	<	int4 int42lt(int4, int2)	less than
	=	bool int4eq(int4, int4)	equality
	=	int4 int42eq(int4, int2)	equality
	>=	bool int4ge(int4, int4)	greater or equal
	>=	int4 int42ge(int4, int2)	greater or equal
	>	bool int4gt(int4, int4)	greater than
	>	int4 int42lt(int4, int2)	less than
		int4 int4inc(int4)	increment
oid	!!=	bool oidnotin(oid, char16)	This is the relational “not in” operator, and is not intended for public use.
	!=	bool oidneq(oid, oid)	inequality
	!=	bool oidneq(oid, regproc)	inequality
	<=	bool int4le(oid, oid)	less or equal
	<	bool int4lt(oid, oid)	less than
	=	bool oideq(oid, oid)	equality
	=	bool oideq(oid, regproc)	equality
	>=	bool int4ge(oid, oid)	greater or equal
	>	bool int4gt(oid, oid)	greater than
point	!<	bool point_left(point, point)	A is left of B
	!>	bool point_right(point, point)	A is right of B
	!ˆ	bool point_above(point, point)	A is above B
	!|	bool point_below(point, point)	A is below B
	=|=	bool point_eq(point, point)	equality
	--->	bool on_pb(point, box)	point inside box
	---‘	bool on_ppath(point, path)	point on path
	<--->	int4 pointdist(point, point)	distance between points
polygon	&&	bool poly_overlap(polygon, polygon)	polygons overlap
	&<	bool poly_overleft(polygon, polygon)	A overlaps B but does not extend to right of B
	&>	bool poly_overright(polygon, polygon)	A overlaps B but does not extend to left of B
	<<	bool poly_left(polygon, polygon)	A is left of B
	>>	bool poly_right(polygon, polygon)	A is right of B
	@	bool poly_contained(polygon, polygon)	A is contained by B
	˜=	bool poly_same(polygon, polygon)	equality
	˜	bool poly_contain(polygon, polygon)	A contains B
regproc	!=	bool oidneq(regproc, oid)	inequality
	=	bool oideq(regproc, oid)	equality
reltime	!=	bool reltimene(reltime, reltime)	inequality
	<=	bool reltimele(reltime, reltime)	less or equal

	<	bool reltimelt(reltime, reltime)	less than
	=	bool reltimeeq(reltime, reltime)	equality
	>=	bool reltimege(reltime, reltime)	greater or equal
	>	bool reltimegt(reltime, reltime)	greater than
text	!=	bool textne(text, text)	inequality
	!~	bool textregexne(text, text)	A does not contain the regular expression B. POSTGRES uses the libc regexp interface for this operator.
	<=	bool text_le(text, text)	less or equal
	<	bool text_lt(text, text)	less than
	=	bool texteq(text, text)	equality
	>=	bool text_ge(text, text)	greater or equal
	>	bool text_gt(text, text)	greater than
	~	bool textregexeq(text, text)	A contains the regular expression B. POSTGRES uses the libc regexp interface for this operator.
tinterval	#!=	bool intervallenne(tinterval, reltime)	interval length not equal to reltime.
	#<=	bool intervallenle(tinterval, reltime)	interval length less or equal reltime
	#<	bool intervallenlt(tinterval, reltime)	interval length less than reltime
	#=	bool intervalleneq(tinterval, reltime)	interval length not equal to reltime
	#>=	bool intervallenge(tinterval, reltime)	interval length greater or equal reltime
	#>	bool intervallengt(tinterval, reltime)	interval length greater than reltime
	&&	bool intervalov(tinterval, tinterval)	intervals overlap
	<<	bool intervalct(tinterval, tinterval)	A contains B
	=	bool intervaleq(tinterval, tinterval)	equality

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

Type	Operator	POSTGRES Function Prototype	Operation
float8	%	float8 dround(float8)	round to nearest integer

Left Unary Operators

Type	Operator	POSTGRES Function Prototype	Operation
box	@@	point box_center(box)	center of box
float4	@	float4 float4abs(float4)	absolute value
float8	@	float8 float8abs(float8)	absolute value
	%	float8 dtrunc(float8)	truncate to integer
	|/	float8 dsqrt(float8)	square root
	||/	float8 dcbrt(float8)	cube root
	:	float8 dexp(float8)	exponential function
	;	float8 dlog1(float8)	natural logarithm
tinterval	|	abstime intervalstart(tinterval)	start of interval

SEE ALSO

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

BUGS

The lists of types, functions, and operators are accurate only for Version 4.0. The lists will be incomplete and contain extraneous entries in future versions of POSTGRES.

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. Underscore (_) is considered an alphabetic.

Keywords

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

abort	**delete**	**key**	**remove**
addattr	**demand**	**leftouter**	**rename**
after	**descending**	**light**	**replace**
all	**destroy**	**load**	**retrieve**
always	**destroydb**	**merge**	**returns**
and	**do**	**move**	**rewrite**
append	**empty**	**never**	**rightouter**
arch_store	**end**	**new**	**rule**
archive	**execute**	**none**	**sort**
arg	**fetch**	**nonulls**	**stdin**
ascending	**forward**	**not**	**stdout**
attachas	**from**	**NULL**	**store**
backward	**function**	**on**	**to**
before	**heavy**	**once**	**transaction**
begin	**in**	**operator**	**type**
binary	**index**	**or**	**union**
by	**indexable**	**output_proc**	**unique**
cfunction	**inherits**	**parallel**	**using**
close	**input_proc**	**pfunction**	**vacuum**
cluster	**instance**	**portal**	**variable**
copy	**instead**	**postquel**	**version**
create	**intersect**	**priority**	**view**
createdb	**into**	**purge**	**where**

current	**intotemp**	**quel**	**with**
define	**is**	**relation**	

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

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 alphabetics 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 8K bytes. Larger objects may be created using the POSTGRES Large Object interface.

Integer Constants

Integer constants in POSTQUEL are collection 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:

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

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

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>}"
```

An example of an array constant is

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

This constant is an array consisting of three sub-arrays of integers.

Fields

A *field* is one of the following:

attribute name in a given class
```
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, cmin, xmax and *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 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 **built-in** types (commands) and b. system types (commands). 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:

```
( a_expr )
constant
attribute
a_expr  binary_operator  a_expr
left_unary_operator  a_expr
parameter
functional expressions
aggregate expressions
set expressions
class expression (not in Version 4.0)
```

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

```
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.:

```
fn-name (a_expr{ , a_expr})
```

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

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

Aggregate Expression

An *aggregate expression* represents a simple aggregate (i.e one which computes a single value) or an aggregate function (i.e. one which 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 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, avg {emp.salary where emp.age = e.age})
     from e in emp
     where e.age > 18
```

Such aggregate functions are not supported in Version 4.0. Furthermore, in this version, only the a_expr and the where-qualification clause are supported. Therefore, for the above simple sum aggregate, the supported query would be the latter. One other note: the qualification will support inheritance, but the expression to be aggregated will not.

Set Expressions

Set expressions are not supported in Version 4.0.

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

```
{target_list from from_list where qualification}
```

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

```
{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:

```
{emp.name where emp.dept = dept.dname}
```

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

Class Expression

Class expressions are not supported in Version 4.0.

A *class expression* is an expression of the form:

class_constructor binary_class_operator class_constructor
unary_class_operator class_constructor

where binary_class_operator is one of the following:

union	union of two classes
intersect	intersection of two classes
–	difference of two classes
>>	left class contains right class
<<	right class contains left class
==	right class equals left class

and unary_class_operator can be:

empty	right class is 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:

```
retrieve (dept.dname)
    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:

```
[result_attname  =] a_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 4.0 default naming is only used if the 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:

```
instance_variable-1 {, instance_variable-2}
     in class_reference
```

where *class_reference* is of the form

```
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:

```
 [date]
 [date-1, 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

```
[MON-FRI] "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. In addition, all times are interpreted as GMT.

For example,

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

are valid time specifications.

SEE ALSO

append(commands), delete(commands), execute(commands), replace(commands), retrieve(commands), monitor(unix).

BUGS

The following constructs are not available in Version 4.0:

```
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 (attname1 **=** type1 {**,** attname-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 the same restrictions as in **create**(commands).

The new attributes will not be added to any classes which inherit attributes from *classname*, unless the "*" is present.

The initial value of each added attribute for all instances is "null."

For efficiency reasons, default values for added attributes are not placed in existing instances of a class. If default values are desired, a subsequent **replace**(commands) query should be run.

EXAMPLE

```
/* add the date of hire to the emp class */

addattr (hiredate = abstime) to emp
```

SEE ALSO

create(commands).

BUGS

"*" is not supported in Version 4.0.

NAME

append — append tuples to a relation

SYNOPSIS

append classname
(att_name1 **=** expression1 {**,** att_name-i **=** expression-i})
[**from** from_list] [**where** qual]

DESCRIPTION

Append adds instances which 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*. The fields may be listed in any order. Fields of the result class which do not appear in the target list are default a null value. If the expression for each field is not of the correct data type, automatic type coercion will be attempted.

The keyword **all** can be used when it is desired to append all domains of a class to another class.

EXAMPLE

```
/* 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 newemp1 class to newemp */

append newemp (newemp1.all)
```

SEE ALSO

postquel(commands), retrieve(commands), define type(commands).

NAME

attachas — reestablish communication using an exising portal

SYNOPSIS

attachas name

DESCRIPTION

This command allows application programs to use a logical name, *name*, in interactions with POSTGRES. 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

This command is not implemented in Version 4.0.

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

end(commands), abort(commands).

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

retrieve(commands), fetch(commands), move(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 *attribute* 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 4.0.

NAME

copy — copy data to or from a class from or to a UNIX file.

SYNOPSIS

copy [**binary**] classname direction ("filename" **|** **stdin** **|** **stdout**)

DESCRIPTION

Copy moves data between POSTGRES classes and standard UNIX files. The keyword **binary** change the behavior of field formatting, as described below. *Classname* is the name of an existing class. *Direction* is either **to** or **from**. *Filename* is the UNIX pathname of the file. In place of a filename, 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.

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 (). Embedded tabs will be preceeded 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 doubled if you want to embed a backslash, ie, the string "12\19\88" will be converted by **copy** to "121988". The actual format for each instance is

<attr1><tab><attr2><tab>...<tab><attrn><newline>

If **copy** is sending its output to standard output instead of a file, it will send a period (.) followed immediately 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.

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 totallength (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 preceeded 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 accessable 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’’ 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. The full pathname 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 (attributename = type { , attributename = type})
[**key** (attributename [[**using**] **operator**]
{ , attributename [[**using**] **operator**] })]
[**inherits** (classname {**,** classname})]
[**archive =** archive_mode]
[**store =** “smgr name”]
[**arch_store =** “smgr name”]

DESCRIPTION

Create will enter a new class into the current data base. 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 *attributenames: attributename, attributename,* etc. The attributes are created with the type specified by *type.*

The *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. Moreover, the “using operator” 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 *inherits* clause specifies a collection of class names from which this class automatically inherits all fields. If any inherited field name appears more than once, POSTGRES reports an error. Moreover, 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).

In addition, *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 or other create statements. See **introduction** (commands) for a further discussion of this point.

The *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 class is created as a heap with no initial data. A class can have no more than 1600 domains (realistically, this is limited by the fact that tuple sizes must be less than 8K), but this limit may be configured lower at some sites. A class cannot have the same name as a system catalog class.

Archive specifies whether historical data is to be saved or discarded. *Arch_mode* may be one of:

none:	no historical access is supported
light:	historical access is allowed and optimized for light update activity
heavy:	historical access is allowed and optimized for heavy update activity

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

EXAMPLE

```
/* Create class emp with attributes name, sal and bdate */

create emp (name = char16, salary = float4, bdate = abstime)

/* Create class permemp with pension information
 * inheriting all fields of emp */

create permemp (plan = char16) inherits (emp)

/* Create a class foo on mag disk,
 * and archive historical data */

create foo (bar = int4) archive = heavy
    store = "magnetic disk"
```

SEE ALSO

destroy(commands).

BUGS

Key is not implemented in Version 4.0.

Optional specifications (inherits, archive, 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. This command was added to POSTQUEL in Version 4.0 and is intended to be used by the createdb script.

SEE ALSO

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

BUGS

This command should NOT be executed by humans. 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 contents 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

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

BUGS

Snapshots (i.e. the optional abstime specifications) have not yet been implemented.

NAME

define aggregate — define a new aggregate

SYNOPSIS

define aggregate agg-name [**as**]
  (sfunc1 = state-transition-function1,
  sfunc2 = state-transition-func2,
  finalfunc = final-function,
  initcond1 = initial-condition1,
  initcond2 = initial-condition2 **)**

DESCRIPTION

An aggregate requires 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 three properties:

(1) The return type of each state-transition-function and the arguments of the final-calculation-function must be the same type (*t*).

(2) The return type of the final-calculation-function must be a POSTGRES base type.

(3) The first argument to state-transition-function1 must be of type *t*, while the second argument must match the data type of the object being aggregated.

Aggregates also require two initial conditions, one for each transition function.

EXAMPLE

The *average* aggregate would consist of two state transition functions, a summer and an incrementer. These would hold the internal state of the aggregate through a running sum and and the number of values seen so far. It might accept a new employee salary, increment the count, and add 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.

```
/* Define an aggregate for int4average */

define aggregate avg (sfunc1 = int4add, sfunc2 = int4inc
     finalfunc = int4div, initcond1 = "0", initcond2 = "0")
```

NAME

define function — define a new function

SYNOPSIS

```
define function function_name (
      language = {"c" | "postquel"},
      returntype = type-r
      [ , percall_cpu = "costly{!*}" ]
      [ , perbyte_cpu = "costly{!*}" ]
      [ , outin_ratio = percentage]
      [ , byte_pct = percentage]
    )
  arg is ( type-1 { , type-n } )
  as {"/full/path/filename.o" | "list-of-postquel-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 the function, 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 input types may be base or complex types. The output type may be specified as a base type, complex type, or *setof <type>*. 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 via the **load** command, 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 *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., $cost = 10^{number\text{-}of\text{-}exclamation\text{-}points}$. 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.

The *outin_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 *outin_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.

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 `$POSTGRESHOME/src/lib/H/utils/builtins.h` is included as a header file. This can be achieved by having

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

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

```
 -I$POSTGRESHOME/src/lib/H
 -I$POSTGRESHOME/src/port/$PORTNAME
 -I$POSTGRESHOME/O/lib/H
```

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

```
cc -I$POSTGRESHOME/src/lib/H \
        -I$POSTGRESHOME/src/port/$PORTNAME \
        -I$POSTGRESHOME/O/lib/H \
        -c progname.c
```

The directory `$POSTGRESHOME/O/lib/H` contains "tags.h", which is generated in the build process. The directory `$POSTGRESHOME/src/port/$PORTNAME` contains "machine.h". Typical values for PORTNAME are sunos4 and ultrix4.

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 `$POSTGRESHOME/src/lib/H` directory) that the equivalent C type is defined. However, if you include "utils/builtins.h", these files will automatically be included.

Equivalent C Types for Built-In POSTGRES Types

Built-In Type	C Type	Defined In
abstime	AbsoluteTime	utils/nabstime.h
bool	bool	tmp/c.h
box	(BOX *)	utils/geo-decls.h
bytea	(bytea *)	tmp/postgres.h
char	char	N/A

char16	Char16 or (char16 *)	tmp/postgres.h
cid	CID	tmp/postgres.h
int2	int2	tmp/postgres.h
int28	(int28 *)	tmp/postgres.h
int4	int4	tmp/postgres.h
float4	float32 or (float4 *)	tmp/c.h or tmp/postgres.h
float8	float64 or (float8 *)	tmp/c.h or tmp/postgres.h
lseg	(LSEG *)	tmp/geo-decls.h
oid	oid	tmp/postgres.h
oid8	(oid8 *)	tmp/postgres.h
path	(PATH *)	utils/geo-decls.h
point	(POINT *)	utils/geo-decls.h
regproc	regproc or REGPROC	tmp/postgres.h
reltime	RELTIME	tmp/postgres.h
text	(text *)	tmp/postgres.h
tid	ItemPointer	storage/itemptr.h
tinterval	TimeInterval	tmp/nabstime.h
uint2	uint16	tmp/c.h
uint4	uint32	tmp/c.h
xid	(XID *)	tmp/postgres.h

Complex arguments to C functions are passed into the C function as a special C type, TUPLE, defined in `$POSTGRESHOME/src/lib/H/tmp/libpq-fe.h.` Given a variable t of this type, the C function may extract attributes from the function using the function call:

```
GetAttributeByName(t, "fieldname", &isnull)
```

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

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 returntype is not specified as a *setof*, then an arbitrary element of the last query's result will be returned. The expensive function parameters *percall_cpu, perbyte_cpu, outin_ratio,* and *byte_pct* are not used for POSTQUEL functions; their costs are determined dynamically by the planner.

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.

```
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:

```
#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:

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

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

```
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:

```
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 TP1
                (language = "postquel", returntype = int4)
                arg is (int4, float8)
                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 = TP1( 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)
            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

```
name   hobby       kid
-------------------------
jeff   biking      (null)
joey   jamming     (null)
mike   swimming    matthew
mike   philately   matthew
mike   swimming    calvin
mike   philately   calvin
```

Note that flattening preserves the name and hobby fields even when the kid field is null.

SEE ALSO

information(unix), load(commands), remove function(commands).

NOTES

The *percall_cpu* and *perbyte_cpu* flags can take integers surrounded by quotes instead of the *"costly{!*}"* syntax described above. This allows a finer grain of distinction between function costs, but is not encouraged since such distinctions are difficult to estimate accurately.

On Ultrix, all .o files that POSTGRES is expected to load dynamically must be compiled under *cc* with the `"-G 0"` option turned on.

RESTRICTIONS

The name of the C function must be a legal C function name, and the name of the function in C code must be exactly the same as the name used in define function.

BUGS

Function names must be unique per database, except for the fact that there may be attributes of the same name as a function. In the case that a there is an ambiguity between a function on a complex type and an attribute of the complex type, the attribute will always be used.

C functions cannot return a set of values.

The dynamic loader for DECstation Ultrix has exceedingly bad performance.

NAME

define index — construct a secondary index

SYNOPSIS

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

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.

The key fields for the index are specified as a collection of attribute names and associated *operator classes*. 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.

POSTGRES Version 4.0 provides btree and rtree access methods for secondary indices. The operator classes defined on btrees are

```
int2_ops        char_ops
int4_ops        char16_ops
int24_ops       oid_ops
int42_ops       text_ops
floag4_ops      abstime_ops
float8_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 POSTGRES query optimizer will consider using b-tree 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         poly_ops
bigbox_ops
```

Both of these 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 r-tree index whenever an indexed attribute is involved in a comparison using one of

```
<<     &<     &>     >>     @     ~=     &&
```

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)
```

BUGS

Archive indices are not supported in Version 4.0.

There should be an access method designers guide.

Indices may only be defined on a single key.

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]
  [**, negator =** neg_op]
  [**, restrict =** res_proc]
  [**, hashes**]
  [**, join =** join_proc]
  [**, sort =** sor_op1 {**,** sor_op2 }]
)

DESCRIPTION

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

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** 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
```

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 separated 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 <<< "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**) 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 estimate the size of the resulting join. The function join_proc will return another floating point number which will be multiplied by the cardinalities of the two classes involved to compute the desired expected result size.

The difference between the function

```
my_procedure_1 (MYBOXES.description, "0,0,1,1"::box)
```

and the operator

```
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 = <<<, <<<)
```

SEE ALSO

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

BUGS

Operator names cannot be composed of alphabetic characters in Version 4.0.

Operator precedence and associativity are not implemented in Version 4.0.

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 | '[' action ... ']']

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 automatically decide which rules system to use and the possibility of user selection will be removed.

Here, event is one of:

```
retrieve
replace
delete
append
```

Moreover, 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 the 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 (read tuple).

EXAMPLES

```
/* 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.  Note: 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).

BUGS

Exceptions are not implemented in Version 4.0.

The object in a POSTQUEL rule cannot be an array reference and cannot have parameters.

The WHERE clause can not 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 surrouas long as the 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.

NAME

define type — define a new base data type

SYNOPSIS

define type typename **(externallength** = (**number** | **variable**),
    [**externallength** = (**number** | **variable**),]
    input = input_function,
    output = output_function
    [**, element** = typename]
    [**, delimiter** = <character>]
    [**, default** = "string"]
    [**, send** = procedure]
    [**, receive** = procedure]
    [**, 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**) before defining the type. The representation of a new base type is determined by the function *input*, which converts the type's external representation to an internal representation usable by the operators and functions defined for the type. Naturally, *output* performs the reverse transformation.

New base data types can be fixed length, in which case *internal length* 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 *internal length* to **-1** Moreover, the external representation is similarly specified using *external length.*

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* attribute. For example, to define an array of 4 byte integers (int4), set the *element* attribute equal to **int4**.

To indicate the delimiter to be used on arrays of this type, the *delimiter* attribute 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 present.”

The optional functions *send* and *receive* are used when the application program requesting POSTGRES services resides on a different machine. In this case, the machine on which POSTGRES runs may use a 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 on output *send* and convert from the standard format to the machine specific format on input *receive.* If these functions are not specified, then it is assumed that the internal format of the type is acceptable on all relevant machine architectures (for example, single characters do not have to be converted if passed from a Sun 3 to a DECstation).

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 smaller than **sizeof**(*char* *), which is typically 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 builtin functions, **array_in** and **array_out,** exist for quick creation of variable length array types. These functions operate on any existing POSTGRES type.

LARGE OBJECT TYPES

A "regular" POSTGRES type can only be 8K bytes in length. If you need a larger type, then you will want to create a Large Object type. The interface for these types is discussed at length in Section 7, the Large Object Backend 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)
```

SEE ALSO

define function(commands), define operator(commands), remove type(commands), Large Object Backend Interface.

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 automatically 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

postquel(commands), create(commands), define rule(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.

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 the **delete** statement.

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, the **merge** command should be used. Moreover, destroying a qclass 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 reversable. 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), remove index(commands), merge(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 by humans. The destroydb(unix) script, should be used instead.

NAME

end — commit the current transaction

SYNOPSIS

end

DESCRIPTION

This commands 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

begin(commands), abort(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 commands to update data.

Portals may only be used inside of **begin/end** transaction blocks, since 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

retrieve(commands), close(commands), move(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 relation_name

DESCRIPTION

listen is used to register the current backend as a listener on the relation **relation_name.** When the command **notify relation_name** is called either from within a rule or at the query level, the frontends corresponding to the listening backends are notified.

SEE ALSO

retrieve(commands), notify(commands), definerule(commands), libpq.

BUGS

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 function is called by POSTGRES. **Load** can also be used to reload an object file if it has been edited and recompiled. Only objects created from C language files are supported at this time.

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** command, 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 diskless platforms or when running across NFS, **load** can take two or three minutes or more, depending on network traffic. On diskful platforms, **load** takes from a few seconds on Suns and Sparcs to several minutes on DECstations.

On DECstations, you must use the "-G O" 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 ADT's.

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* becomes children of *classname2*.

It is disallowed for a version class to be merged into its parent class when the parent class is also the parent of another version class.

Moreover, merging in the reverse direction is also 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

destroy(commands), create version(commands).

BUGS

Merge has not yet been implemented.

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 with no 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

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

BUGS

This command is not yet available. The pointer may be moved using the **fetch** command, and ignoring its return values.

NAME

notify — Signal all frontends and backends listening on a relation

SYNOPSIS

notify relation_name

DESCRIPTION

notify is used to awaken all backends and consequently all frontends listening on the relation **relation_name.**

This can be used either within a tuple 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 IPC. When used from within a rule, this can be thought of as the alerter mechanism.

SEE ALSO

definerule(commands), listen(commands), libpq.

BUGS

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. The date specified is an absolute time such as Jan 1 1987, and POSTGRES 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. POSTGRES will enforce this restriction, silently.

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
```

BUGS

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.

NAME

remove aggregate — remove the definition of an aggregate

SYNOPSIS

remove aggregate aggname

DESCRIPTION

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

DESCRIPTION

Remove function will remove all references to an existing C function. To execute this command the user must be the owner of the function.

EXAMPLE

```
/* this command removes the square root function */

remove function sqrt
```

SEE ALSO

define function(commands).

BUGS

No support is provided for removing POSTQUEL functions.

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
```

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.

EXAMPLE

```
/* Remove power operator a^n for 4 byte integers */

remove operator ^ (int4, 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).

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. Anyone is allowed to remove a type, and removal of types in use by a class will not be refused. Be careful not to remove a built-in type.

It is the user's responsibility to remove any operators and functions that use a deleted type.

EXAMPLE

```
/* remove the box type */

remove type box
```

SEE ALSO

introduction(commands), definetype(commands), removeoperator(commands).

BUGS

This command should only be available to the definer of the type.

NAME

rename — rename a class or an attribute in a class

SYNOPSIS

rename classname1 **to** classname2
rename attname1 **in** classname **to** attname2

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.

EXAMPLE

```
/* change the emp class to personnel */

rename emp to personnel

/* change the sports attribute to hobbies */

rename sports in emp to hobbies
```

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_name1 **=** expression1 {**,** att_name-i **=** expression-i })
[**from** from_list]
[**where** qual]

DESCRIPTION

Replace changes the values of the attributes specified in the target_list for all instances which satisfy the qualification, *qual*. Only attributes which are to be modified need appear in the target_list.

EXAMPLE

```
/* Give all employees who work for Smith a 10% raise */

replace emp(sal = 1.1 * emp.sal)
    where emp.mgr = "Smith"
```

NAME

retrieve — retrieve instances from a class

SYNOPSIS

retrieve
[(**into** classname [archive_mode] | **portal** portal_name) | **iportal** portal_name]
[**unique**]
([attr_name1 **=**] expression1 {**,** [attr_name-i **=**] expression-i} **)**
[**from** from_list]
[**where** qual]
[**sort by** attr_name−1 [**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 return them to an application program through one of two different kinds of portals or 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 specifed, then it defaults to **light** archiving. (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 res_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 **iportal** 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 res_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.

EXAMPLE

```
/* Find 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}) \g

retrieve (e.all)
   from e in emp
   where e.sal > avgsal.ave \g

/* retrieve employees's names sorted */

retrieve unique (emp.name)
   sort by name using <

/* retrieve all employees's 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 \g
   retrieve portal myportal (pg_user.all) \g
   fetch 2 in myportal \g
   fetch all in myportal \g
   close myportal \g
end \g
```

SEE ALSO

postquel(commands), fetch(commands), close(commands), create(commands).

BUGS

“Retrieve into” does not delete duplicates in Version 4.0.

“Archive_mode” is not implemented in Version 4.0.

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". A solution to this problem is being investigated and will be released in later version.

“Retrieve iportal” returns data in an architecture dependent format, namely that of the server on which the backend is running. A standard data format should be adopted, most likely XDR. At that point, there will be no need to distinguish among external and internal data.

Aggregate functions must 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' speed in processing user queries.

The open database is the one that is vacuumed. This is a new POSTQUEL command in Version 4.0; earlier versions of POSTGRES had a separate program for vacuuming databases. That program has been replaced by the **vacuum** shell script; see **vacuum**(unix) for details.

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

NAME

libpq — programmer's interface to POSTGRES

DESCRIPTION

LIBPQ is the programmer's interface to POSTGRES. LIBPQ is a set of library routines which allow queries to pass to the POSTGRES back-end and instances to return through an IPC channel.

This version of the documentation is based on the C library.

CONTROL AND INITIALIZATION

VARIABLES

The following five 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 back-end.
- **PGTTY** sets the tty on the PQhost back-end on which debugging messages are displayed.

The following internal variables of libpq can be accessed by the programmer:

```
char *PQhost;           /* the server on which POSTGRES
                           back-end is running. */

char *PQport = NULL;    /* The communication port with the
                           POSTGRES back-end. */

char *PQtty;            /* The tty on the PQhost back-end on
                           which back-end messages are
                           displayed. */

char *PQoption;         /* Optional arguements to the back-end */

char *PQdatabase;       /* Back-end database to access */

int  PQportset = 0;     /* 1 if communication with
                           back-end 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 */
```

QUERY EXECUTION FUNCTIONS

The following routines control the execution of queries from a C program.

PQsetdb — Make the specified database the current database,

```
void PQsetdb ( dbname )
      char  *dbname;
```

PQsetdb also resets communication via PQreset (see below).

PQdb — Return the current database being accessed.

```
char * PQdb ()
```

Returns the name of the POSTGRES database being accessed, or NIL if no database is open. Only one database can be accessed at a time. The database name is a string limited to 16 characters.

PQreset — Reset the communication port with the back-end in case of errors.

```
void PQreset()
```

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 back-end.

PQfinish — Close communication ports with the back-end.

```
void PQfinish ()
```

Terminates communications and frees up the memory taken up by the libpq buffer.

PQfn — Send a function call to the POSTGRES backend.

```
char *PQfn(fnid, result_buf, result_len,
           result_is_int, args, nargs)
      int fnid;
      int *result_buf;    /* can't use void, the */
      int result_len;     /* compiler complains  */
      int result_is_int;
      PQArgBlock *args;
```

```
     int nargs;
```

PQfn provides access to the POSTGRES fastpath facility, a trapdoor into the system internals. See **FASTPATH**.

PQexec — Submit a query to POSTGRES.

```
char *     PQexec (query)
     char  * query;
```

This function returns a status indicator or an error message. If the query returns data (e.g. fetch), PQexec returns a string consisting of 'P' followed by the name of the portal buffer. When the query does not return instances, PQexec will return a string consisting of 'C' followed by the command tag (e.g. "CREPLACE"). If an error occured during the execution of the query PQexec will return (for historical reasons) an "R".

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 seperate 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 and a field 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 group are guaranteed to have the same instance format.

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.

```
int PQnportals ( rule_p )
    int         rule_p ;
```

If rule_p is not 0, then only return the number of asynchronous portals.

PQpnames — Return all portal names.

```
void PQpnames  ( pnames, rule_p)
    char        *pnames [MAXPORTALS];
    int         rule_p ;
```

If rule_p is not 0, then only return the names of asynchronous portals.

PQparray — Return the portal buffer given a portal name.

```
PortalBuffer * PQparray ( pname )
    char        *pname;
```

PQclear — free storage claimed by named portal.

```
void PQclear ( pname )
    char        *pname;
```

PQntuples — Return the number of instances in a portal buffer.

```
int PQntuples (portal)
    PortalBuffer         *portal;
```

PQngroups — Return the number of instance groups in a portal buffer.

```
int PQngroups (portal)
    PortalBuffer *portal
```

PQntuplesGroup — Return the number of instances in an instance group.

```
int PQntuplesGroup (portal, group_index)
    PortalBuffer         *portal;
    int         group_index;
```

PQnfieldsGroup — Return the number of fields in an instance group.

```
int PQnfieldsGroup ( portal, group_index)
    PortalBuffer         *portal;
    int         group_index;
```

PQfnameGroup — Return the field name given the group and field index.

```
char * PQfnameGroup (portal, group_index, field_number )
    PortalBuffer         *portal;
    int         group_index;
    int         field_number;
```

PQfnumberGroup — Return the field number (index) given the group index and field name.

```
int PQfnumberGroup (portal, group_index, field_name)
    PortalBuffer         *portal;
    int         group_index;
    char        *field_name;
```

PQgetgroup — Returns the index of the group that a particular instance is in.

```
int PQgetgroup ( portal, tuple_index )
    PortalBuffer         *portal;
    int         tuple_index;
```

PQnfields — Return the number of fields in an instance.

```
int PQnfields (portal, tuple_index )
    PortalBuffer         *portal;
    int         tuple_index;
```

PQfnumber — Return the field index of a given field name within an instance.

```
int PQfnumber ( portal, tuple_index, field_name)
    PortalBuffer         *portal;
    int         tuple_index;
    char        *field_name;
```

PQfname — Return the name of a field.

```
char * PQfname ( portal, tuple_index, field_number )
    PortalBuffer         *portal;
    int         tuple_index;
    int         field_number;
```

PQftype — Return the type of a field.

```
int PQftype ( portal, tuple_index, field_number )
    PortalBuffer         *portal;
    int         tuple_index;
    int         field_number;
```

The type returned is an internal coding of a type.

PQsametype — Return 1 if two instances have the same attributes.

```
int PQsametype ( portal, tuple_index1, tuple_index2 )
    PortalBuffer         *portal;
    int         tuple_index1, tuple_index2;
```

PQgetvalue — Return an attribute (field) value.

```
char * PQgetvalue ( portal, tuple_index, field_number )
    PortalBuffer         *portal;
    int         tuple_index;
    int         field_number;


PQgetlength — Return the length of an attribute (field) value in bytes
If the field is a varlena, the length of the attribute returned here
does not include the longword size field of the varlena, e.g. it is 4
bytes less.

char * PQgetlength ( portal, tuple_index, field_number )
    PortalBuffer         *portal;
    int         tuple_index;
    int         field_number;

PQNotifies — Return the list of relations on which notification has occ

PQNotifyList *PQNotifies()

PQRemoveNotify — Remove the notification from the list of unhandled
                  notifications.

PQNotifyList *PQRemoveNotify(pqNotify)
    PQNotifyList *pqNotify;
```

If the portal is blank, or specified with the **portal** keyword, all values are returned as 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 internal format, namely in the format generated by the *input* function specified through the **definetype** command. Again, it is the programmer's responsibility to convert the data to the correct type.

ASYNCHRONOUS PORTALS/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 frontend/backend protocol.

Referring to the second sample program, after executing "listen relation_name" in the frontend process, periodically check PQAsyncNotifyWaiting. If it is non-zero, then the "notify relation_name" command has been executed by some backend. Immediately clear PQAsyncNotifyWaiting, then do a NULL query, i.e. PQexec(" "), to retrieve the actual notification data. Then call PQNotifies() to get the list of relations on which notification has occurred. After handling the notification, do PQRemoveNotify on each element of the list that has been handled to prevent further handling by you.

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(string, length)` — Reads a null-terminated line into string.

```
char *string; int length
```

`PQputline(string)` — Sends a null-terminated string.

```
char *string;
```

`int PQendcopy()` — Syncs with the back-end.

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 PGgetline(). It must be issued or the backend may get "out of sync" with the frontend. Upon return from this function, the backend is ready to receive the next query.

The return value is 0 on successful completion, nonzero otherwise.

For Example:

```
PQexec("create foo (a=int4, b=char16, d=float8)");
PQexec("copy foo from stdin");
PQputline("3<TAB>hello world<TAB>4.50);
PQputline("4<TAB>goodbye world<TAB>7.11");
PQputline(".\n");
PQendcopy();
```

TRACING FUNCTIONS

PQtrace — Enable tracing.

```
void PQtrace ()
```

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 FILE *debug_port to the appropriate stream.

PQuntrace — Disable tracing.

```
void PQuntrace ()
```

BUGS

The query buffer is only 8192 bytes long, and queries over that length will be silently truncated.

SAMPLE PROGRAM

```
/*
 * testlibpq.c —
 *    Test the C version of Libpq, the POSTGRES frontend library.
 */
#include <stdio.h>
#include "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 = PQparray ("eportal");
    g = PQngroups (p);
    t = 0;

    for (k = 0; k < g; k++) {
     printf ("\nA new instance group:\n");
     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");

       /* End the transaction block */
       PQexec("end");

       /* Try out some other functions. */

       /* Print out the number of portals. */
       printf ("\nNumber of portals open: %d.\n",
               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 portal 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"

   void main()
   {
       extern int PQAsyncNotifyWaiting;
       PQNotifyList *l;
       PortalBuffer *portalbuf;
       char *res;
       int ngroups,tupno, grpno, ntups, nflds;
       PQsetdb(getenv("USER"));

       PQexec("listen test1a");

       while(1) {
           sleep(1);
           if (PQAsyncNotifyWaiting) {
             PQAsyncNotifyWaiting = 0;
               PQexec(" ");
               l = PQnotifies();
               if (l != NULL) {
                   printf("notification on relation %s\n",
                     l->relname);
                   res = PQexec("retrieve (test1a.i)");
                   if (*res == 'E') {
                       fprintf(stderr,"%s\nfailed",++res);
                       goto exit_error;
                   }
                   if (*res != 'P') {
                       fprintf(stderr,"%s\nno portal",++res);
                   }
                   /* get tuples in relation */
                   portalbuf = PQparray(++res);
                   ngroups = PQngroups(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);
                           goto exit_error;
                       }
                       for (tupno = 0; tupno < ntups; tupno++) {
                           printf("got i=%s\n",
                     PQgetvalue(portalbuf,tupno,0));
                       }
                   }
                   break;
               }
           }
       }

       PQfinish();
       exit(0);
     exit_error:
       PQfinish();
       exit(1);

   }
```

SAMPLE PROGRAM 3

```
      /*
       * Testing of new binary portal interface.
       *
       * Do the following at the monitor:
       *

       create test1 (i = int4,d = float4,p = polygon) \g
       append test1 (i = 7, d=3.567,p="(1.0,2.0,3.0,4.0)"::polygon) \g

       -- Anything else you can think of.
       * Start up this program.
       * The correct contents of test1 should be printed
       */

      #include "tmp/simplelists.h"
      #include "tmp/libpq.h"
      #include "utils/geo-decls.h"

      void main()
      {
          extern int PQAsyncNotifyWaiting;
          PQNotifyList *l;
          PortalBuffer *portalbuf;
```

```
        char *res;
        int ngroups,tupno, grpno, ntups, nflds;
        PQsetdb(getenv("USER"));

        PQexec("begin");
        res = (char *)PQexec("retrieve iportal junk (test1.all)");
        if (*res == 'E') {
          fprintf(stderr,"%s\nfailed",++res);
          goto exit_error;
        }
        res = (char *)PQexec("fetch all in junk");
        if (*res != 'P') {
          fprintf(stderr,"\nno portal");
          goto exit_error;
        }
        /* 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, "expected 3 attributes, got %d\n", nflds);
               goto exit_error;
          }
          for (tupno = 0; tupno < ntups; tupno++) {
               int *bla1;
               char *bla2;
               POLYGON *bla3;
               bla1 = (int *)PQgetvalue(portalbuf,tupno,0);
               bla2 = PQgetvalue(portalbuf,tupno,1);
               bla3 = PQgetvalue(portalbuf,tupno,2)-4;

               printf ("got i=%d(%d bytes), d=(%f)(%d bytes)|%x|%x|%x|%x\n\
    Polygon(%d bytes)\
     %d points (%f,%f,%f,%f)\n",
                       *bla1,PQgetlength(portalbuf,tupno,0),
                       *((float *)bla2),
                       PQgetlength(portalbuf,tupno,1),
                       *bla2,*(bla2+1),*(bla2+2),*(bla2+3),
                       PQgetlength(portalbuf,tupno,2),
                       bla3->npts,
                       bla3->boundbox.xh,bla3->boundbox.yh,
                       bla3->boundbox.xl,bla3->boundbox.yl);
          }
        }

        PQexec("end");
        PQfinish();
```

```
          exit(0);
        exit_error:
          PQexec("end");
          PQfinish();
          exit(1);

    }
```

SECTION 6 — FAST PATH

NAME

fast path — trap door into system internals

SYNOPSIS

“retrieve (retval = function([arg { , arg }])”

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.

SECTION 7 — LARGE OBJECTS

NAME

Large Object Interface — interface to POSTGRES large objects

DESCRIPTION

In POSTGRES, data values are stored in tuples, and individual tuples cannot span multiple 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 explicitly declared to be a large type.

Version 4 of POSTGRES supports two different implementations of large objects. These two implementations allow users to trade off speed of access against transaction protection and crash recovery on large object data. Applications that can tolerate lost data may store object data in conventional files that are fast to access, but cannot be recovered in the case of system crashes. For applications that require stricter guarantees of durability, a transaction-protected large object implementation is available. This section describes the two implementations and the programmatic and query language interfaces to large object data.

Unlike the BLOB support provided by most commercial relational database management systems, POSTGRES allows users to define specific large object types. POSTGRES large objects are first-class objects in the database, and any operation that can be applied to a conventional (small) abstract data type (ADT) may also be applied to a large one. For example, two different large object types, such as *image* and *voice*, may be created. Functions that operate on image data, and other functions that operate on voice data, may be declared to the database system. The data manager will distinguish between image and voice data automatically, and will allow users to invoke the appropriate functions on values of each of these types. In addition, indices may be created large data values, or on functions of them. Finally, operators may be defined that operate on large values. Users may invoke these functions and operators from the query language. The database system will enforce type restrictions on large object data values.

The POSTGRES large object interface is modeled after the Unix file system interface, with analogs of open(), read(), write(), lseek(), 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 existed that stored photographs of faces, then a function called could be declared on data. 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 function. As mentioned above, POSTGRES supports functional indices on large object data. In this example, the results of the 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.

This implementation stores large object data in a UNIX file, and stores only the file name in the database. Importing a large object into the database is as simple as storing the file name in a distinguished "large object name" relation. Interface routines allow the database system to open, seek, read, write, and close these UNIX files by an internal large object identifier.

The functions and convert between UNIX filenames and internal large object identifiers. These functions are POSTGRES registered functions, meaning they can be used directly in Postquel queries as well as from dynamically loaded C functions. If you are defining a simple large object ADT, these functions can be used as your "input" and "output" functions (see **define type** and the POSTGRES Manual sections concerning user-defined types for details).

```
char *lo_filein(filename)
        char *filename;
```

Import a new UNIX file storing large object data into the database system. This routine stores the filename in a large object naming relation and assigns it a unique large object identifier.

```
char * lo_fileout (object)
        LargeObject *object;
```

This routine returns the UNIX filename associated with a large object.

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.

Large objects are accessible from both the POSTGRES backend, using dynamically-loaded functions, and from the front-end, using the LIBPQ interface. These interfaces will be described in detail below.

INVERSION LARGE OBJECTS

In contrast to UNIX files as large objects, the Inversion large object implementation guarantees transaction protection, crash recovery, and time travel on user large object data. This 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.

If a transaction that has made changes to an Inversion large object subsequently aborts, the changes are backed out in the normal way. Inversion large objects are stored in the database, and so are not directly accessible to other programs. Only programs that use the POSTGRES data manager can read and write Inversion large objects.

To use Inversion large objects, a new large object should be created using the LOcreat() interface, defined below. Afterwards, the name of the large object can be stored in an ordinary tuple.

The next section describes the programmatic interface to both UNIX and Inversion large objects.

BACKEND INTERFACE TO LARGE OBJECTS

Large object data is accessible from front-end programs linked with the LIBPQ library, and from dynamically-loaded routines that execute in the POSTGRES backend. This section describes access from dynamically loaded C functions.

Creating New Large Objects

The routine

```
int LOcreat(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 ("/") in which objects may be placed.

The parameter can be one of or which are symbolic constants defined in

```
~postgres/src/lib/H/catalog/pg_lobj.h
```

The interpretation of the argument depends on the selected.

For UNIX files, is the mode used to protect the file on the UNIX file system. On creation, the file is open for reading and writing.

For Inversion large objects, is a bitmask describing several different attributes of the new object. The symbolic constants listed here are defined in

```
~postgres/src/lib/H/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 are the storage manager number on which the large object should reside[1]. 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.

[1] 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.

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:

```
unix_fd = LOcreat("/my_unix_obj", 0600, Unix);

inv_fd = LOcreat("/my_inv_obj",
                 INV_READ|INV_WRITE, Inversion);
```

Opening Large Objects

Existing large objects may be opened for reading or writing by calling the routine

```
int LOopen(path, mode)
    char *path;
    int mode;
```

The argument specifies the large object's pathname, and is the same as the pathname used to create the object. The 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 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

```
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

```
int
LOlseek(fd, offset, whence)
    int fd;
    int offset;
    int whence;
```

moves the current location pointer for a large object to the specified position. The parameter is the file descriptor returned by either or is the byte offset in the large object to which to seek. The only legal value for in the current release of the system is as defined in <sys/files.h>.

UNIX large objects allow holes to exist in objects; that is, a program may seek well past the end of the object and write bytes. Intervening blocks will not be created; 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:

```
unix_status = LOlseek(unix_fd, 100000, L_SET);

inv_status = LOlseek(inv_fd, 100000, L_SET);
```

On error, 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

```
int
LOwrite(fd, wbuf)
    int fd;
    struct varlena *wbuf;
```

Here, is the file descriptor returned by or and describes the data to write. The structure in POSTGRES consists of four bytes in which the length of the datum is stored, followed by the data itself. The four length bytes include themselves.

For example, to write 1024 bytes of zeroes to the sample large objects:

```
struct varlena *vl;

vl = (struct varlena *) palloc(1028);
VARSIZE(vl) = 1028;
bzero(VARDATA(vl), 1024);

nwrite_unix = LOwrite(unix_fd, vl);

nwrite_inv = LOwrite(inv_fd, vl);
```

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

```
~postgres/src/lib/H/tmp/postgres.h
```

Reading from Large Objects

Data may be read from large objects by calling the routine

```
struct varlena *
LOread(fd, len)
    int fd;
    int len;
```

This routine returns the byte count actually read and the data in a varlena structure. For example,

```
        struct varlena *unix_vl, *inv_vl;
        int nread_ux, nread_inv;
        char *data_ux, *data_inv;

        unix_vl = LOread(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 and may be when they are no longer needed.

Closing a Large Object Once a large object is no longer needed, it may be closed by calling

```
        int
        LOclose(fd)
            int fd;
```

where is the file descriptor returned by or On success, returns zero. A negative return value indicates an error.

For example,

```
        if (LOclose(unix_fd) < 0)
            /* error */;

        if (LOclose(inv_fd) < 0)
            /* error */
```

LIBPQ LARGE OBJECT 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 conventional UNIX-style buffer scheme is used.

In version 4 of POSTGRES, large object operations must be enclosed in a transaction block. This is true even for UNIX large objects, which are not transaction-protected. This is due to a shortcoming in the memory management scheme for large objects, and will be rectified in version 4.1. The end of this section shows a short example program that correctly transaction-protects its file system operations.

This section describes the LIBPQ interface in detail.

Creating a Large Object

The routine

```
int
p_creat(path, mode, objtype)
    char *path;
    int mode;
    int objtype;
```

creates a new large object. The argument specifies a large-object system pathname.

The parameter can be one of or which are symbolic constants defined in

```
~postgres/src/lib/H/catalog/pg_lobj.h
```

The interpretation of the argument depends on the selected.

For UNIX files, is the mode used to protect the file on the UNIX file system. On creation, the file is open for reading and writing.

For Inversion large objects, is a bitmask describing several different attributes of the new object. The symbolic constants listed here are defined in

```
~postgres/src/lib/H/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 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:

```
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

```
int
p_open(path, mode)
    char *path;
    int mode;
```

The 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

```
int
p_write(fd, buf, len)
    int fd;
    char *buf;
    int len;
```

writes bytes from to large object The argument must have been returned by a previous or

The number of bytes actually written is returned. In the event of an error, the return value is negative.

Reading Data from a Large Object

The routine

```
int
p_read(fd, buf, nbytes)
    int fd;
    char *buf;
    int nbytes;
```

reads bytes into buffer from the large object descriptor The number of bytes actually read is returned. In the event of an error, the return value is less than zero.

Seeking on a Large Object

To change the current read or write location on a large object, call

```
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 to the new location specified by For this release of , only is a legal value for

Closing a Large Object

A large object may be closed by calling

```
int
p_close(fd)
    int fd;
```

where is a large object descriptor returned by or On success, returns zero. On error, the return value is negative.

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 directories

```
$POSTGRESHOME/test/postfs
$POSTGRESHOME/test/postfs.usr.bin
```

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 postfs directories 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 == ’E’)     /* error */
            exit (1);

      /* open the unix file */
      fd = open("/my_unix_file", O_RDONLY, 0666);
      if (fd < 0)      /* error */
            exit (1);

      /* open the inversion file */
      inv_fd = p_open("/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 == 'E')     /* 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.

SEE ALSO

define type(commands), define function(commands), load (commands).

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 `.../.postgresrc`, `.../data/PG_VERSION`, `.../data/*/PG_VERSION`, the temporary sort files, and the database debugging trace files are absent.

NAME

.../src/support/{local,dbdb}.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 .../files/local1.bki and .../files/global1.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/global1.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, then the first “$” is ignored and the argument is then processed normally. If the “$” is followed by space, then it is treated as a NULL value. Otherwise, the characters following the “$” are interpreted as the name of a macro causing the argument to be replaced with the macro’s value. It is an error for this macro to be undefined.

Macros are defined using "define macro macro_name = macro_value" and are undefined using "undefine macro macro_name" and redefined using the same syntax as define.

Lists of general commands and macro commands follow.

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 it's OID. If *oid* is not “0”, then this value will be used as the instance's object identifier. Otherwise, it is an error. To let the system generate a unique object identifier (as opposed to the "well-known" object identifiers which we specify) use insert '(' value1, value2, ... valuen ')' .

```
create classname '(' name1 = type1,
                     name2 = type2, ...name n = type n ')'
```

Create a class named *classname* with the attributes given in parentheses.

```
open '('name1 = type1, name2 = type2,...name n = type n ')'
    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 <amname>
    ( <opclass> <attr> | function({attr}) )
```

Create an index named *index_name* on the class named *classname* using the *amname* access method. The fields to index are called *name1, name2,* etc. and the operator collections to use are *collection_1, collection_2,* etc., respectively.

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 etc.

```
define macro macro_name from file filename
```

Define a macro named *macname* which has its value read from the file called *filename.*

EXAMPLE

The following set of commands will create the 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

.../data/... — database file default page format

DESCRIPTION

This section provides an overview of the page format used by POSTGRES classes. Diagram 1 shows how pages in both normal POSTGRES classes and POSTGRES index classes (eg., a B-tree index) are structured. 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 a sample page layout. Running ".../bin/dumpbpages" or ".../src/support/dumpbpages" as the postgres superuser with the file paths associated with (heap or B-tree index) classes, ".../data/base/<database-name>/<class-name>," will display the page structure used by the classes. Specifying the "-r" flag will cause the classes to be treated as heap classes and for more information to be displayed.

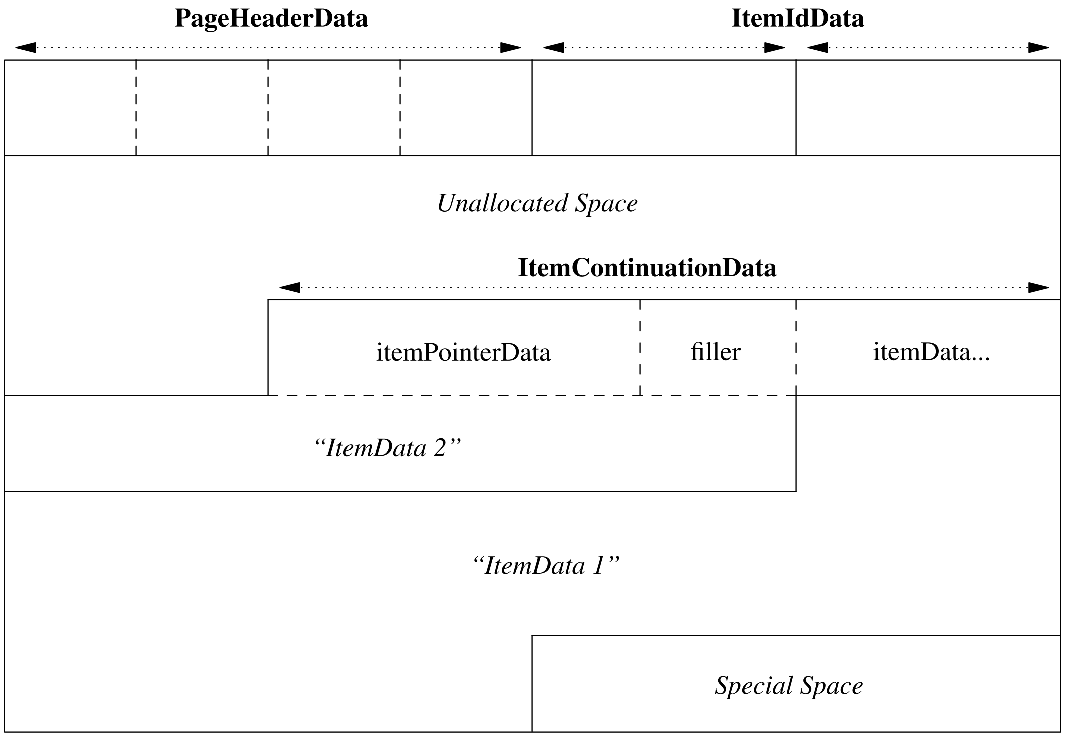

Diagram 1: Sample Page Layout

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.

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

.../files/global1.bki — global database template
.../files/local1_XXX.bki — local 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 superuser runs **initdb**. These files are copied from ".../src/support/{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.

.../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 C preprocessor macros.

BUGS

POSTGRES Version 4.0 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.

[ONG90]
Ong, L. and Goh, J., "A Unified Framework for Version Modeling Using Production Rules in a Database System," Electronics Research Laboratory, University of California, Berkeley, ERL Memo M90/33, April 1990.

[ROWE87]
Rowe, L. and Stonebraker, M., “The POSTGRES Data Model,” Proc. 1987 VLDB Conference, Brighton, England, Sept. 1987.

[SHAP86]
Shapiro, L., “Join Processing in Database Systems with Large Main Memories,” ACM-TODS, Sept. 1986.

[STON87]
Stonebraker, M., “The POSTGRES Storage System,” Proc. 1987 VLDB Conference, Brighton, England, Sept. 1987.

[STON90]
Stonebraker, M. et. al., "On Rules, Procedures, Caching and Views in Database Systems," Proc. 1990 ACM-SIGMOD Conference on Management of Data, Atlantic City, N.J., June 1990.

[WONG76]
Wong, E., “Decomposition: A Strategy for Query Processing,” ACM-TODS, Sept. 1976.

Table of Contents