POSTGRES REFERENCE MANUAL
VERSION 2.1

Edited by

Claire M. Mosher

Memorandum No. UCB/ERL M88/20

25 March 1988
(Revised April 15, 1991)

POSTGRES REFERENCE MANUAL
VERSION 2.1

Edited by

Claire M. Mosher

Memorandum No. UCB/ERL M88/20

25 March 1988
(Revised April 15, 1991)

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Title page

POSTGRES REFERENCE MANUAL
VERSION 2.1

Edited by

Claire M. Mosher

Memorandum No. UCB/ERL M88/20

25 March 1988
(Revised April 15, 1991)

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Table of Contents

OVERVIEW

This document is the reference manual for the POSTGRES database system under development at the University of California, Berkeley. This project, led by Professors Michael Stonebraker is sponsored by the Defense Advanced Research Projects Agency (DARPA), the Army Research Office (ARO), the National Science Foundation (NSF), 3M Corp, 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 manual describes Version 2.1 of POSTGRES. The POSTGRES software is about 170,000 lines of C code, and is available for SUN 3 and SUN 4 class machines, for DECstation 3100 machines and for the SEQUENT Symetry machine. Information on obtaining the source code for these computers is available from:

Claire Mosher
Computer Science Division
521 Evans Hall
University of Califiornia
Berkeley, Ca. 94720
(415) 642-4662

No attempt has been made to optimize Version 2.1; consequently, one should expect performance comparable to the public domain, University of California Version of INGRES, a relational prototype from the late 1970s.

This manual contains the description of a few capabilities which are not implemented in Version 2.1. We expect to support additional functionality in Version 3, currently scheduled for third quarter 1991. Moreover, Version 3 will be tuned to run as fast as possible.

POSTGRES DOCUMENTATION

This reference manual describes the functionality of Versions 2.1, and 3 and contains notations where appropriate to indicate which features are not implemented in Version 2.1. Application developers should note that this reference manual contains only the specification for the low-level call-oriented application program interface, LIBPQ. In addition, a companion project directed by Professor Rowe is building a collection of powerful development tools called **Picasso**, which will be the subject of a separate reference manual.

The remainder of this manual is structured as follows. In Section 2, we discuss the POSTGRES capabilities that are available directly from the operating system. Section 3 then describes POSTQUEL, the language by which a user interacts with a POSTGRES database. Then, Section 4 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 5 continues with a description of a method by which applications may execute functions in POSTGRES with very high performance. The manual concludes with Section 6, 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 Version 2.1 contributors (in alphabetical order) consisted of James Bell, Ron Choi, Jeffrey Goh, Wei Hong, Anant Jhingran, Greg Kemnitz, Michael Olson, Lay-Peng Ong, Spyros Potamianos, and Cimarron Taylor.

Greg Kemnitz 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 manual was collectively written by the above implementation team, assisted by Michael Stonebraker and Claire Mosher.

FOOTNOTES

UNIX is a trademark of AT&T.

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 C 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 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 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

initdb — initdb a database

SYNOPSIS

initdb [-v]

DESCRIPTION

initdb sets up the initial template databases. It is normally executed as part of the installation process. *-v* 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

createdb — create a database

SYNOPSIS

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

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.

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).

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.

<dbname> already exists

The database already exists.

NAME

destroydb — destroy an existing database

SYNOPSIS

destroydb [-p port] [-h hostname] dbname

DESCRIPTION

Destroydb removes all reference to an existing database named *dbname* and turns off the vacuum demon if running on this database. Normally, the directory containing this database and all associated files are removed. But when the database is placed elsewhere via the use of a "reference file," only the files contained in the referenced directory will be 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).

NAME

postmaster — run the Postgres postmaster

SYNOPSIS

postmaster [-p port] [-b backend_pathname] &

DESCRIPTION

The postmaster manages the communication between frontends and backends, as well as allocating the shared buffer pool and semaphores. 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 the terminal monitor and the backend. If you specify this then you must also specify them when starting the terminal monitor.

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

EXAMPLES

postmaster &

This command will start up a postmaster on the default ports (4321 and 4322) which will expect to use the default path to the postgres backend ($POSTGRESHOME/bin/postgres) or /usr/postgres/bin/postgres. This is the simplest way to start the postmaster.

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

This command will start up a postmaster communicating through ports 1234 and 1235, which 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.

SEE ALSO

postgres (unix), monitor (unix), ipcclean (unix)

NAME

monitor — run the interactive terminal monitor

SYNOPSIS

monitor [-h hostname] [-p port] [-t path] [-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 path specifies the path name of the file or tty which you want the backend debugging messages to be sent to; default is /dev/null.

-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 three 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.

continue	The terminal monitor is ready for input and the query buffer is not empty. Typing input will cause the query buffer to be silently cleared. Typing a terminal monitor command will cause the contents of any query buffer to be preserved. Further input will then be appended 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

SEE ALSO

backend(unix), postmaster(unix)

NAME

postgres — run the Postgres backend directly

SYNOPSIS

postgres [-Q] [databasename]

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 varuable 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 multiprocessor systems (e.g. sequent). Presently the slave processes are not used, so don't do this. Full support for heavyweight query parallism is not expected until version 3.

-D nnn indicates the degree of disk striping the backend should use. Again this functionality is only experimental at this stage.

-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.

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

vcontrol — control the vacuum daemon on a database

SYNOPSIS

vcontrol [**–h** host] [**–p** port] [–el] dbname

DESCRIPTION

Vcontrol controls the status of the vacuum daemon on a database.

COMMAND OPTIONS

–e
–k

Specifying the *–e* or *–k off* option enables the vacuum daemon for a database or kills it respectively. We suggest that you run a vacuum daemon on each active database. That way expired instances will be purged according to the criteria set in the **purge** command for each class. Also, this will ensure that the statistics kept in the CLASS class are updated periodically.

–s

Specifying the *–s* option shows whether a vacuum daemon is running or not on the specified database.

-p port
-h host

The vacuum daemon is associated with some postmaster process. (Note that a postmaster MUST be running to execute this command.) Specifying the port and host using *-p port* and *-h host* will cause the vacuum daemon to use the postmaster associated with the specified host and port.

ENVIRONMENT VARIABLES

PGPORT
The port on which the postmaster is running. This value is used if the *-p* option is not specified. If *-p* is not specified and PGPORT is not set, then the port defaults to 4321.

PGHOST
The host on which the postmaster is running. This value is used if the **-h** option is not specified. If **-h** is not specified and PGHOST is not set, then the host defaults to "localhost".

SEE ALSO

createdb(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.

<dbname> already exists

The database already exists.

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.

OVERVIEW

In this portion of the manual, we describe the components of the query language POSTQUEL which is available either from the terminal monitor or from an application program via LIBPQ. The main concepts in POSTQUEL are types, functions and rules. In this introduction we describe each of these constructs. Immediately following this introduction, we discuss the components of the POSTQUEL language, built-in types, and system types. In the next portion of the manual the individual POSTQUEL commands appear in alphabetical order.

KINDS OF TYPES

POSTGRES supports three kinds of types, namely **base** types, **array** types, and **composite** types. The query language capabilities for each are different, and we discuss them in turn.

Base types hold atomic data elements that appear to POSTGRES internals as uninterpreted byte strings. Example base types are integers and floating point numbers. Indexes can be constructed for attributes of classes containing base types and such attributes can be referenced using the conventional class-name.attribute addressing format. Moreover, functions and operators can be defined whose operands are base types. Lastly, base types can be added and dropped dynamically.

There are three kinds of **base** types available in POSTGRES.

(1) **Built-in types**
These are data types that are used in the system catalogs. Hence, they must exist as POSTGRES data types or the POSTGRES system will not run. Most of these types are ''hard wired'' into POSTGRES so the system can boot.

(2) **System types**
These are data types that are defined by the POSTGRES system administrator. They are automatically available for each data base that is created on a POSTGRES system. The built-in and system data types can be changed by a system administrator by making appropriate modifications to the file

.../files/local1_template1.bki

Each new data base automatically receives the collection of built-in and system types specified in the above file at the time the data base is created. System types which are defined subsequently must be inserted into pre-existing data bases one-by-one as user defined types.
Other template files may be constructed in files named

.../files/local1_template-name.bki

and then used by **createdb** (unix) with the -t flag. See **bki** (files) and **createdb** (unix) for more information.

(3) **User types**
These data types are defined dynamically by a user of a data base. Their scope is limited to the data base in which they are defined. See **define type** (commands) for details on creating and using these types. C functions, POSTQUEL functions, aggregate functions, and operators can be defined for user types using respectively the commands **define C** (commands), **define POSTQUEL function** (commands), **define aggregate** (commands), and **define operator** (commands).

In addition POSTGRES supports fixed and variable length **arrays** of base types. Whenever a new built-in, system or user type is constructed, POSTGRES automatically defines fixed and variable length arrays of this type as additional types. If B is a base type, then B[N] is an array of N instances of B, while B[] is a variable length array of instances of B, for example:

create emp (name = char16, age = int4, budget = int4[12], salary_history = float8[])

Here budget is an array of 12 integers while salary_history is a variable length array of floating point numbers. No sparse matrix techniques are applied to the storage of arrays; rather elements are stored contiguously in an instance.

All operations available for base types are also available for arrays of base types. Moreover, conventional array addressing is automatically provided in POSTQUEL. Hence, the i-th element of an array can be addressed as

class-name.instance[i]

For example the following query updates the April budget of joe.

replace emp (budget[4] = 95) where emp.name = "joe"

There are also two kinds of **composite** types in POSTQUEL.

(1) **One or more instances in a specific class**
Whenever a class is created, a type is automatically constructed of the same name whose value is one or more instances in the indicated class. For example, if "emp" is created as a class, then the type **emp** is automatically constructed. This new type can be used in other classes, for example:

create dept (name = char16, budget = int4, mgr = **emp**)

Here the field **mgr** is of type "emp" and refers to one or more instances from the "emp" class. The value of the **mgr** attribute for each instance is a function which returns the type, emp. For example, if f is a POSTQUEL function which accepts a character string argument and returns the type, emp, then the following is a valid insert to dept:

append dept (name = "toy", budget = 100000, mgr = f ("toy"))

In Version 2.1, only POSTQUEL functions have the power to return composite types. In the future C functions will be extended to have this capability.

(2) **Any set as a data type**
The type **set** is automatically available and allows the value of an attribute in a class to be an arbitrary collection of instances from arbitrary classes. For example, consider the following emp class:

create emp (name = char16, hobbies = set)

Here, the value of hobbies for any employee is any collection of instances from one or more classes. In fact, the actual value is a function which returns this type, Assuming that f has been defined to return the set type, the following insert works correctly.

append to emp (name = "joe", hobbies = f("joe"))

For composite data types POSTQUEL supports "nested dot" addressing. Hence, the following query will find the name of the manager of the shoe department:

retrieve (dept.mgr.name) where dept.name = "shoe"

Nested dot notation is explained in the **postquel** (postquel) section.

KINDS OF FUNCTIONS

In POSTGRES there are four kinds of functions that can be defined.

(1) **Normal functions**
Normal functions can be written either in C or in POSTQUEL and then defined to POSTGRES using the **define C function** (commands) and **define POSTQUEL function** (commands) respectively. Normal functions take base or array types as arguments and return base, array or composite types.
Queries can include normal functions using the standard notation, e.g.:

retrieve (emp.name) where overpaid (emp.salary, emp.age)

Here, overpaid is a normal function accepting a floating point number and an integer as arguments and returning a boolean. Clauses in a qualification containing normal functions cannot be optimized by POSTGRES, and a sequential scan of the associated class will typically result.

(2) **operators**
Consider a normal function which takes two operands of the same type and returns a boolean, e.g:

retrieve (emp.name) where greater (emp.age, 25)

An operator can be associated with this function, say >, using the **define operator** (commands) command. In this command, the information is specified that is needed by the optimizer to efficiently process queries including the operator token. Hence, the query:

retrieve (emp.name) where emp.age > 25

can be optimized to use an age index, whereas the one with the function notation cannot.

(3) **aggregate functions**
Aggregate functions allow a POSTGRES user to compute aggregates such as count, sum and average. Unfortunately, they do not work in Version 2.1.

(4) **Inheritable functions (methods)**
If a function has a first argument which is of type **instance** in some class, then this function is inheritable. Consider the following query:

retrieve (emp.name) where overpaid(emp)

Here overpaid takes an argument of type instance in emp and returns a boolean. Such functions can be written in C or POSTQUEL. If written in C, they must access fields in the argument tuple using special **accessor functions** as described **in the define C function** (commands) section. Inheritable functions can be referenced either using the functional notation above or using one of the attribute style notations as follows:

retrieve (emp.name) where emp.overpaid
retrieve (emp.name) where emp.overpaid()

These latter notations emphasise the fact that overpaid effectively defines a new attribute for the class emp containing the field, overpaid. Moreover, if any class inherits from the emp class, e.g: the pensionemp class, then any inheritable functions defined for emp are automatically defined for pensionemp. Hence, the following query automatically works:

retrieve (pensionemp.name) where overpaid (pensionemp)

Inheritable functions follow the conventions of the Common Lisp Object System (CLOS) when a function can be inherited from multiple parents.

RULES

The third major concept in POSTGRES is the notion of **rules.** They have the form:

```
on condition
then do action
```

Rules can be used to **trigger** DBMS actions e.g:

```
on update to emp.salary where emp.name = "mike"
then do replace emp (salary = new.salary) where emp.name = "joe"
```

When mike receives a salary adjustment, then this rule propagates the new salary on to Joe. An alternate rule which accomplishes the same thing is:

```
on retrieve to emp.salary where emp.name = "joe"
then do instead retrieve (emp.salary) where emp.name = "mike"
```

This rule will retrieve the salary of mike in place of whatever is stored in joe's record. Rules can be used to assist with the definition and maintenance of data in a class. Moreover, rules can sometimes be used in place of functions if the user wishes. Hence the following two commands have the effect of defining an attribute, overpaid.

```
add to emp (overpaid = boolean)

on retrieve to emp.overpaid
then do instead retrieve (overpaid = overpaid (current.salary, current.age))
```

This attribute will be inherited in the standard way, and the effect is the same as an inheritable function. The above solution allows the user to add additional rules to further define the column, e.g:

```
on update to emp.overpaid
then do ....
```

Such additional rules cannot be specified using the solution containing a function definition.

DESCRIPTION

This section describes the built-in data types and their associated functions and operators. A POSTGRES system **cannot run** without these types, so the POSTGRES system administrator is cautioned not to remove them.

bool	boolean
char	character
int2	two-byte signed integer
int4	four-byte signed integer
float4	single-precision floating-point number
float8	double-precision floating-point number
uint2	two-byte unsigned integer
uint4	four-byte unsigned integer
cid	command identifier type
oid	object identifier type
tid	tuple identifier type
xid	transaction identifier type

The following types are also required built-in types, but are expected to change or disappear between versions.

abstime	absolute date and time
bytea	variable length array of bytes
char16	array of 16 characters
datetime	timestamp
int28	array of 8 int2
oid8	array of 8 oid
regproc	registered procedure
reltime	relative date and time
text	variable length array of characters
tinterval	time interval

These types all have obvious formats except for the three time types, explained below:

ABSOLUTE TIME

Absolute time is specified using the following syntax:

Month Day [Hour : Minute : Second] Year

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 1902, 1903, ..., 2038

Valid dates are, therefore, Jan 1 00:00:00 1902 to Jan 1 00:00:00 2038. In Version 2, times are read and written using Greenwich Mean Time. The special absolute time ''now'' is also provided as a convenience. Similarly, the special absolute time ''epoch'' means Jan 1 00:00:00 1902.

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.

FUNCTIONS

The following functions are defined on built-in types and are essential to the operation of POSTGRES.

return type	*function name and argument types*	*meaning*
bool	boolin(external)	converts argument from external to internal form
char	charin(external)	"
int2	int2in(external)	"
int4	int4in(external)	"
float4	float4in(external)	"
float8	float8in(external)	"
cid	cidin(external)	"
oid	oidin(external)	"
tid	tidin(external)	"
xid	xidin(external)	"
abstime	abstimein(external)	"
reltime	reltimein(external)	"
tinterval	tintervalin(external)	"
bytea	byteain(external)	"
char16	char16in(external)	"
int28	int28in(external)	"

oid8	oid8in(external)	"
text	textin(external)	"
datetime	datetimein(external)	"
regproc	regprocin(external)	"
external	boolout(bool)	converts argument from internal to external form
external	charout(char)	"
external	int2out(int2)	"
external	int4out(int4)	"
external	float4out(float4)	"
external	float8out(float8)	"
external	cidout(cid)	"
external	oidout(oid)	"
external	tidout(tid)	"
external	xidout(xid)	"
external	abstimein(abstime)	"
external	reltimein(reltime)	"
external	tintervalin(tinterval)	"
external	byteaout(bytea)	"
external	char16out(char16)	"
external	int28out(int28)	"
external	oid8out(oid8)	"
external	textout(text)	"
external	datetimeout(datetime)	"
external	regprocout(regproc)	"
bool	booleq(bool,bool)	tests for equality
bool	chareq(char,char)	"
bool	int2eq(int2,int2)	"
bool	int4eq(int4,int4)	"
bool	float4eq(float4,float4)	"
bool	float8eq(float8,float8)	"
bool	cideq(cid,cid)	"
bool	oideq(oid,oid)	"
bool	tideq(tid,tid)	"
bool	xideq(xid,xid)	"
bool	abstimeeq(abstime,abstime)	"
bool	reltimeeq(reltime,reltime)	"
bool	tintervaleq(tinterval,tinterval)	"
bool	char16eq(char16,char16)	"
bool	texteq(text,text)	"
bool	datetimeeq(datetime,datetime)	"
bool	regproceq(regproc,regproc)	"
bool	int2ge(int2,int2)	tests for greater than or equal to, >=
bool	int4ge(int4,int4)	"
bool	float4ge(float4,float4)	"
bool	float8ge(float8,float8)	"
bool	int2gt(int2,int2)	tests for greater than, >
bool	int4gt(int4,int4)	"

bool	float4gt(float4,float4)	"
bool	float8gt(float8,float8)	"
bool	int2le(int2,int2)	tests for less than or equal to, <=
bool	int4le(int4,int4)	"
bool	float4le(float4,float4)	"
bool	float8le(float8,float8)	"
bool	int2lt(int2,int2)	tests for less than, <
bool	int4lt(int4,int4)	"
bool	float4lt(float4,float4)	"
bool	float8lt(float8,float8)	"
bool	ininterval(abstime, tinterval)	tests if time is in interval
bool	intervalct(tinterval, tinterval)	tests for contained-in
bool	intervalov(tinterval, tinterval)	tests for overlaps
abstime	intervalend(tinterval)	returns ending time of time interval
abstime	intervalstart(tinterval)	returns starting time for time interval
datetime	timenow()	returns current time

OPERATORS

The following operators are automatically defined on the built-in types. In practice, many of the functions named below can be the same function called with different argument types (depending on your compiler); thus, not all of the functions are actually distinct.

binary operator	*result type*	*supporting function*
=	bool	booleq(bool,bool)
equality		chareq(char,char)
		int2eq(int2,int2)
		int4eq(int4,int4)
		int24eq(int2,int4)
		int42eq(int4,int2)
		float4eq(float4,float4)
		float8eq(float8,float8)
		float48eq(float4,float8)
		float84eq(float8,float4)
		oideq(oid,oid)
		abstimeeq(abstime,abstime)
		reltimeeq(reltime,reltime)
		char16eq(bool,bool)
		texteq(text,text)
!=	bool	int2ne(int2,int2)
inequality		int4ne(int4,int4)
		int24ne(int2,int4)
		int42ne(int4,int2)
		float4ne(float4,float4)
		float8ne(float8,float8)
		float48ne(float4,float8)
		float84ne(float8,float4)
		oidne(oid,oid)
		abstimene(abstime,abstime)
		reltimene(reltime,reltime)
>=	bool	int2ge(int2,int2)
greater/equal		int4ge(int4,int4)
		int24ge(int2,int4)
		int42ge(int4,int2)
		float4ge(float4,float4)
		float8ge(float8,float8)
		float48ge(float4,float8)
		float84ge(float8,float4)
		abstimege(abstime, abstime)
		reltimege(reltime, reltime)
>	bool	int2gt(int2,int2)

greater		int4gt(int4,int4)
		int24gt(int2,int4)
		int42gt(int4,int2)
		float4gt(float4,float4)
		float8gt(float8,float8)
		float48gt(float4,float8)
		float84gt(float8,float4)
		abstimegt(abstime, abstime)
		reltimegt(reltime, reltime)
<=	bool	int2le(int2,int2)
less/equal		int4le(int4,int4)
		int24le(int2,int4)
		int42le(int4,int2)
		float4le(float4,float4)
		float8le(float8,float8)
		float48le(float4,float8)
		float84le(float8,float4)
		abstimele(abstime, abstime)
		reltimele(reltime, reltime)
<	bool	int2lt(int2,int2)
less		int4lt(int4,int4)
		int24lt(int2,int4)
		int42lt(int4,int2)
		float4lt(float4,float4)
		float8lt(float8,float8)
		float48lt(float4,float8)
		float84lt(float8,float4)
		abstimelt(abstime, abstime)
		reltimelt(reltime, reltime)
+	int2	int2pl(int2,int2)
addition	int4	int4pl(int4,int4)
	int4	int24pl(int2,int4)
	int4	int42pl(int4,int2)
	float4	float4pl(float4,float4)
	float8	float8pl(float8,float8)
	float8	float48pl(float4,float8)
	float8	float84pl(float8,float4)
	abstime	timepl(abstime,abstime)
–	int2	int2mi(int2,int2)
subtraction	int4	int4mi(int4,int4)
	int4	int24mi(int2,int4)
	int4	int42mi(int4,int2)
	float4	float4mi(float4,float4)
	float8	float8mi(float8,float8)
	float8	float48mi(float4,float8)
	float8	float84mi(float8,float4)
	abstime	timemi(abstime,abstime)

/	int2	int2div(int2,int2)
division	int4	int4div(int4,int4)
	int4	int24div(int2,int4)
	int4	int42div(int4,int2)
	float4	float4div(float4,float4)
	float8	float8div(float8,float8)
	float8	float48div(float4,float8)
	float8	float84div(float8,float4)
*	int2	int2mul(int2,int2)
multiplication	int4	int4mul(int4,int4)
	int4	int24mul(int2,int4)
	int4	int42mul(int4,int2)
	float4	float4mul(float4,float4)
	float8	float8mul(float8,float8)
	float8	float48mul(float4,float8)
	float8	float84mul(float8,float4)
%	int4	int4mod(int4,int4)
modulo	int2	int2mod(int2,int2)
		int4
		int4
^	float8	dpow(float8,float8)
power		
<<	bool	intervalct(tinterval, tinterval)
contained in		
&&	bool	intervalov(tinterval, tinterval)
overlaps		
#=	bool	intervalleneq(tinterval, reltime)
#!=	bool	intervallenne(tinterval, reltime)
#<	bool	intervallenlt(tinterval, reltime)
#>	bool	intervallengt(tinterval, reltime)
#<=	bool	intervallenle(tinterval, reltime)
#>=	bool	intervallenge(tinterval, reltime)
<?>	bool	ininterval(abstime, tinterval)
time comparison		

unary left operators	*result type*	*supporting procedure*
–	float4	float4um(float4)
unary minus	float8	float8um(float8)
@	float4	float4abs(float4)
absolute value	float8	float8abs(float8)
|/	float8	dsqrt(float8)

<table>
<tr><td>square root</td><td></td><td></td></tr>
<tr><td>||/
cube root</td><td>float8</td><td>dcbrt(float8)</td></tr>
<tr><td>%
round</td><td>float8</td><td>dround(float8)</td></tr>
<tr><td>:
exponent</td><td>float8</td><td>dexp(float8)</td></tr>
<tr><td>;
log</td><td>float8</td><td>dlog1(float8)</td></tr>
<tr><td>|</td><td>abstime</td><td>intervalstart(tinterval)</td></tr>
<tr><td>##
typecast</td><td>int2
int4
float4
float8</td><td>int4toint2(int4)
int2toint4(int2)
dtof(float8)
ftod(float4)</td></tr>
</table>

<table>
<tr><th>unary right operators</th><th>result type</th><th>supporting procedure</th></tr>
<tr><td>!
factorial</td><td>int4
int2</td><td>int4fac(int4)
int2fac(int2)</td></tr>
<tr><td>%
truncate</td><td>float8</td><td>dtrunc(float8)</td></tr>
<tr><td>|</td><td>abstime</td><td>intervalend(tinterval)</td></tr>
</table>

BUGS

The lists of types, functions, and operators are accurate only for Version 2.1. The lists will be incomplete and contain extraneous entries in future versions of POSTGRES.

DESCRIPTION

This section describes the available system data types and their associated functions and operators available in Version 2. These types are installed in every POSTGRES data base automatically. The POSTGRES system administrator can change them by editing

.../files/local1_template1.ami

as explained in local **template** (files).

The default system types are:

point	data point type
lseg	line segment type
path	variable length array of lseg
box	2d rectangle type

The intent of including these particular types and their associated functions and operators is to provide an example of a suite of user data types. No claim is made that they are useful or efficient.

FUNCTIONS

The following functions are defined on system types.

return type	*function name and argument types*	*meaning*
bool	box_overlap(box,box)	tests for overlapping boxes
bool	box_ge(box,box)	tests for area greater than or equal, >=
bool	box_gt(box, box)	tests for area greater than, >
bool	box_eq(box,box)	tests for area equality, =
bool	box_lt(box,box)	tests for area less than, <
bool	box_le(box,box)	tests for area less than or equal, <=
bool	point_above(point,point)	tests if point is above point
bool	point_left(point,point)	tests if point is left of point
bool	point_right(point,point)	tests if point is right of point
bool	point_below(point,point)	tests if point is below point
bool	point_eq(point,point)	tests for equality
bool	inside(point,box)	tests if point is in box
bool	on_ppath(point,path)	tests if point lies on path
external	point_out(point)	converts argument from internal to external form
external	lseg_out(lseg)	"
external	path_out(path)	"
external	box_out(box)	"
point	point_in(external)	converts argument from external to internal form
lseg	lseg_in(external)	"
path	path_in(external)	"
box	box_in(external)	"
int4	pointdist(point,point)	determines distance between two points

point	box_center(box)	locates center of box

OPERATORS

binary operator	*result type*	*supporting procedure*
=	bool	box_eq(box,box),
&&	bool	box_overlap(box,box)
=|=	bool	point_eq(point,point)
!^	bool	point_above(point,point)
!<	bool	point_left(point,point)
!>	bool	point_right(point,point)
!|	bool	point_below(point,point)
—>	bool	inside(point,box)
—'	bool	on_ppath(point,path)
spatial comparison		
<	bool	box_lt(box,box)
>=	bool	box_ge(box,box)
>	bool	box_gt(box,box)
<=	bool	box_le(box,box)
area comparison		
<—>	int4	point_dist(point,point)
distance		

unary left operators	*result type*	*supporting procedure*
@@	point	box_center(box)
center of box		

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	**addattr**	**after**
all	**always**	**and**
append	**archive**	**arg**
ascending	**attachas**	**backward**
before	**begin**	**binary**
by	**c**	**cfunction**
close	**cluster**	**copy**
create	**current**	**define**
delete	**demand**	**descending**
destroy	**do**	**empty**
end	**execute**	**fetch**
forward	**from**	**function**
heavy	**in**	**index**
indexable	**inherits**	**input_proc**
instead	**intersect**	**into**
is	**key**	**leftouter**
light	**merge**	**move**
never	**new**	**none**
nonulls	**not**	**null**
on	**once**	**operator**
or	**output_proc**	**pfunction**
portal	**postquel**	**priority**
purge	**quel**	**relation**
remove	**rename**	**replace**
retrieve	**returns**	**rewrite**
rightouter	**rule**	**sort**
to	**transaction**	**tuple**
type	**union**	**unique**
using	**version**	**view**
where	**with**	

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

Constants

There are five 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 current limitations on tuple sizes, string constants are currently limited to a length of a little less than 8K bytes. These size constraints will be removed when large-object support becomes stable (this is one of the goals of POSTGRES version 3).

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 Other 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.

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 40 bit quantities which are assigned sequentially starting at 1. Command identifiers are 8 bit objects; hence, it is an error to have more than 256 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** (POSTQUEL) 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-intypes** (postquel) and b. system types (postquel). For a list of user defined operators consult your system administrator or run a query on the 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 expression (not in Version 2.1)
class expression (not in Version 2.1)

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 command. 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 POSTQUEL function DEPT (char16) returning (dept) as
  retrieve (dept.all) where dept.dname = $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

Aggregate expressions are not supported in Version 2.1.

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:

```
sum {e.salary from e in emp where e.age > 30}
```

or

```
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
```

Set Expressions

Set expressions are not supported in Version 2.1.

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 2.1.

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 2.1 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:

and
and not
or
or not

A clause is an a_expr that evaluates to a Boolean over a set of instances. *Not* followed by a qualification is a legal qualification.

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. Additionally, one can 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

"MMM DD [HH:MM:SS] YYYY"

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 1940'', ''Mar 3 23:59:59 1941'']

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 2.1:

aggregates and aggregate expressions
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).

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 (either explicitly or by default) are assigned default values. The expression for each field must be of the correct data type. There is no automatic coercion of expressions.

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

The "*" indicates a transitive closure and POSTGRES will run the command until it produces no further effect.

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(postquel), retrieve(commands), definetype(commands).

BUGS

The code to support "*" is very buggy.

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 2.1.

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 property.

Transactions are supported by page level locks which are escalated to the relation level if excessive page level locks are set.

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** domname [**using** operator]

DESCRIPTION

This command instructs POSTGRES to keep the class specified by *classname* approximately sorted on *domname* using the specified operator to determine the sort order. The operator must be a binary operator and both operands must be of type *domname* 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 *domname* 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 2.1.

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 (0, 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 (0, 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 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(postquel), create(postquel), libpq(commands).

BUGS

Files used as arguments to the copy command must reside on the database server.

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 **copyto**.

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.

b Copy has virtually no error checking, and a malformed input file will likely cause the backend to crash.

NAME

create — create a new class

SYNOPSIS

create classname (attributename = type { , attributename = type}) [key (attributename [[using] operator] { , attributename [[using] operator])] [inherits (classname { , classname}] [archive_mode]

DESCRIPTION

Create will enter a new class into the current data base. The class will be "owned" by the user issueing 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 other create statements. See **introduction** (commands) for a further discussion of this point.

The class is created as a heap with no initial data. A class can have no more than 1600 domains, but this limit may be configured lower at some sites. A class cannot have the same name as a system catalog class.

The archive specification for each class can 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

For details of the optimization, see [STON87]. Once the archive status is set, there is no way to change it. See the **purge** (commands) command for details on specifying how much history is kept.

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
```

SEE ALSO

destroy(commands)

BUGS

Key and archive_mode are not implemented in Version 2.1.

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 rules 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["January 17, 1990"]
```

SEE ALSO

merge(commands).

NAME

define c function — define a new C function

SYNOPSIS

define c function function_name (file = "filename" , returntype = <typename> [, **iscachable**]) **arg** (type–1 { , type–n })

DESCRIPTION

Via this command, the implementor of a C function can register it to 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 a *filename* which indicates the FULL PATH to the object code in .o format for the function. (POSTGRES will not compile a function automatically - it must be compiled before it is used in a define c function command.) This code will be dynamically loaded when necessary for execution. Repeated execution of a function will cause negligible additional overhead, as the function will remain in a main memory cache.

The presense of the *iscachable* flag indicates that the function can be precomputed. Under a variety of circumstances, POSTGRES **caches** the result of a function for improved performance. Most functions can be evaluated earlier than requested; however, some functions (such as "time-of-day") cannot. Thus, the *iscachable* flag is used to indicate which option is appropriate for the function being defined. If the flag is not specified, POSTGRES defaults to never precomputing the function.

Functions can be either called in the POSTGRES address space or a process will be forked for the function and a remote procedure call executed. The choice of **trusted** or **untrusted** operation is controlled by the DBA of the data base in question who can set the "trusted" flag in the appropriate system catalog.

When a function is executed, POSTGRES automatically performs type-checking of the parameters and signals an error if there is a type mismatch.

C functions are currently available in two variations. If a function is defined whose arguments and return types are all **base** types, then this is a **normal** function. Normal functions can be used in the query language POSTQUEL to perform computations and also can be associated with POSTQUEL operators using **define operator** (commands).

For example, The following command defines a function, overpaid.

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

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

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

On the other hand, the first argument to a function can also be of type set or of type **classname,** representing one or more instances of a particular class. In this case an **inheritable** function is defined. An inheritable function essentially specifies a new attribute for the associated class, whose data type is the return type of the function and whose attribute name is the name of the function. Inheritable functions can be referenced using either the attribute notation or the function notation in POSTQUEL as explained in the following example.

Consider an inheritable function overpaid-2, defined as follows:

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

The following queries are now accepted:

```
retrieve (EMP.name) where overpaid_2(EMP)

retrieve (EMP.name) where EMP.overpaid_2

retrieve (EMP.name) where EMP.overpaid_2()
```

In this case, in the body of the overpaid_2 function, the fields in the EMP record must be extracted using a function call getattr(name) as explained in the companion POSTGRES tutorial.

Alternately, the following two commands do essentially the same thing.

```
addattr (overpaid = bool) to EMP

define rule example
on retrieve to EMP.overpaid
then do instead retrieve (overpaid = overpaid_2(current.salary, current.age))
```

SEE ALSO

information(unix), remove function(commands). Tutorial: Creating and Using a C function.

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 c function.

BUGS

Untrusted operation is not implemented in Version 2.1.

C functions cannot return composite types.

The notation X.f is not supported for an inheritable function, f.

Inheritable C functions are restricted to have a single tuple argument in Version 2.1.

There are numerous bugs in the 2.1 dynamic loader. If you have problems with loading a C function, please consult the release notes. Also, the dynamic loader for Ultrix has exceedingly bad performance.

NAME

define postquel function — define a new POSTQUEL function

SYNOPSIS

define postquel function function_name (type–1 { , type–n }) **returns** class-name **is** postquel-query | { list-of-postquel-queries }

DESCRIPTION

The user can define a POSTQUEL function which consists of a single postquel query or a brace-enclosed list of postquel queries. Via this command, the implementor of a **POSTQUEL function** can register it to POSTGRES. Subsequently, this user is treated as the owner of the function.

When defining a POSTQUEL function, the input data types, *type-1*, *type-2*, ..., *type-n* must be specified, along with the queries that constitute the function. The return type of a POSTQUEL function is one of the composite types indicated in the **introduction** (commands) section of the manual. If not specified, then *set* is the default return type.

POSTQUEL functions are automatically cachable as long as no POSTQUEL command in the function in turn contains an uncachable function. Lastly, functions coded in POSTQUEL are automatically trusted, since a POSTQUEL function cannot escape from the POSTGRES run-time system.

POSTQUEL functions are currently available in the same two variations allowed for C functions. If a POSTQUEL function is defined whose arguments are all **base** types, then this is a **normal** function. Normal functions can be used in the query language POSTQUEL to perform computations. The parameters to a normal function are specified in the body of the function using the markers, $1, ..., $n, as noted in the example function, TP1, defined as follows:

```
define postquel function TP1(int4, float8) is
        replace BANK (balance = balance - $2)
        where BANK.accountno = $1
```

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

```
retrieve (x = TP1( 17,100.0))
```

On the other hand, the first argument to a POSTQUEL function can also be a class name, in which case an **inheritable** function is defined. Inheritable functions can be referenced using either the column notation or the function notation in POSTQUEL as explained in the example below. Moreover, the "cascaded dot" notation available to reference composite columns is available for inheritable functions. Lastly, an inheritable function takes a value for each instance in the class, *classname,* which is the first argument to the function. Hence, in the body of the function, any references to classname.attribute are automatically references to the corresponding data element in the current instance of classname. Other parameters are denoted positionally as in normal functions.

To illustrate inheritable functions we use the GROUPS class as follows:

```
create GROUPS(name = char16, age = int4)
```

An example inheritable function would be:

```
define postquel function composition (GROUPS),
```

```
retrieve (EMP.all)
where EMP.age > GROUPS.age
```

Composition has a value for each instance of GROUPS which is the query:

```
retrieve (EMP.all)
where EMP.age > current.age
```

Hence, the following command would define a new group "elders" whose members were over 50.

```
append to groups (name = "elders", age = 50)
```

Both of the following commands obtain the names of all employees who are "elders".

```
retrieve (GROUPS.composition.name)
where GROUPS.name = "elders"

retrieve (composition(GROUPS).name)
where GROUPS.name = "elders"
```

Alternately, the user can manually achieve the same definition of the composition function by the following two commands:

```
addattr (composition = EMP) to GROUPS

define rule example
on retrieve to GROUPS.composition
then do instead retrieve (EMP.all) where EMP.age > current.age
```

BUGS

POSTQUEL functions currently can neither accept arguments nor return results of a base type.

Inheritable functions are currently restricted to a single argument of type classname.

Inheritable POSTQUEL functions are not currently propagated thru the inheritance hierarchy.

Inheritable functions cannot currently use functional notation and must instead use the column notation. For example, if a function foo takes the class EMP as an argument

```
retrieve ( EMP.foo.all )
```

works, but

```
retrieve ( foo(EMP).all )
```

does not.

POSTQUEL functions cannot currently take a list of queries as the function body.

In Version 2.1, POSTQUEL functions are not cachable, no matter what their composition may be.

In Version 2.1, POSTQUEL functions exhibit relational-cross-product rather than outer-join semantics. This means, for instance, that given this schema :

```
create emp ( name = char16, salary = int4, dept = char16 )

define postquel function hobbies ( emp ) returns hobbies is
retrieve ( hobbies.all ) where hobbies.empname = emp.name
```

```
append emp ( name = "goh", salary = 1000, dept = "toy" )
append emp ( name = "hong", salary = 1000, dept = "not-toy" )
append emp ( name = "cimarron", salary = 1500, dept = "toy" )
append emp ( name = "ron", salary = 29000, dept = "sys-admin" )

create hobbies ( activity = char16, empname = char16 )

append hobbies ( activity = "kayaking", empname = "goh" )
append hobbies ( activity = "basketball", empname = "hong" )
append hobbies ( activity = "basketball", empname = "ron" )
```

the query

```
retrieve ( emp.hobbies.activity , emp.name )
```

returns

```
activity                          name
--------------------------------------
kayakinggoh
basketball                        hong
basketball                        ron
--------------------------------------
```

rather than

```
activity                          name
--------------------------------------
kayakinggoh
basketball                        hong
basketball                        ron
NULL    cimarron
--------------------------------------
```

NAME

define aggregate — define a new aggregate

SYNOPSIS

define aggregate agg-name [**as**] (state-transition-function, final-calculation-function)

DESCRIPTION

An aggregate consists of two functions, a *state transition* function, T:

T(internal-state, next-data_item) ---> next-internal-state

and a **final calculation** function, C:

C(internal-state) ---> aggregate-value

These functions are required to have the following three properties:

(1) The output of state-transition-function and the input of final-calculation-function must be the same type, S.

(2) The output of final-calculation-function can be of arbitrary type.

(3) The input to state-transition-function must include as its first argument a value of type S. The other arguments must match the data types of the object being aggregated.

EXAMPLE

The *average* aggregate could consist of a state transition function which uses as its state the sum computed so far 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 function must also be able to initialize correctly when passed a null current state. The final calculation function divides the sum by the count to produce the final answer.

```
/* Define an aggregate for average */

define aggregate avg as (add-new-value-function, divide-by-total-function)
```

BUGS

Define aggregate is not implemented in Version 2.1.

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 }) [**with** (parameter-list)]

DESCRIPTION

This command constructs an index called *index-name*. If the **archive** keyword is absent, the *classname* class is indexed. In contrast, 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 classes. A class is used to indicate the collection of functions and operators which the access method should use to manipulate the index.

Predefined type classes are:

int2_ops float4_ops
int4_ops float8_ops
area_ops oid_ops

All are defined for the normal comparison operators (<, <=, =, >, >=).

New classes can be added dynamically by making an insertion in the pg_opclass class in the system catalogs. Operators can be associated with a class by making insertions in the pg_amop class in the system catalogs.

The *parameter-list* specifies access method specific performance parameters such as the fill-factor to be used when loading the pages of the index or the minimum and maximum number of pages to allocate.

Version 2.1 of POSTGRES comes with a standard B-tree access method, a linear hashing access method and a R-tree access method. In addition, users are encouraged to write their own. To define a new access method, the functions indicated in the pg_am relation must be written. Unfortunately there is no documentation to support an access method writer; hence he should look carefully at the source code for the B-tree access method included with POSTGRES.

EXAMPLE

Create a btree index on the emp class using the age attribute.

```
define index emp-index on emp using btree (age int4_ops)
```

BUGS

Archives are not supported in Version 2.1.

There should be an access method designers guide.

The *parameter-list* is not supported in Version 2.1.

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 C 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 C 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 C function(commands)

BUGS

Operator names cannot be composed of alphabetic characters in Version 2.1. Operator precedence and associativity are not implemented in Version 2.1.

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

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. In version 2.1, 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.**

EXAMPLES

/* Make Sam get the same salary adjustment as Joe */

```
define rule example_1 is
on replace to EMP.salary where current.name = ''Joe''
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 = ''Joe''
replace EMP (salary = current.salary) where EMP.name = ''Bill''
```

/* Deny Joe access to the salary of employees in the shoe department */

```
define rule example_3 is
on retrieve to EMP.salary where current.dept = ''shoe'' and user() = ''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.dname = ''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(postquel).

BUGS

Exceptions are not implemented in Version 2.1.

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, and it can only be a replace, append, retrieve or delete command.

The rewrite system currently processes only a subset of the rule set. Specifically, it can only accept rules of the form:

```
on retrieve ...
then do [instead] retrieve ...

or
on retrieve ....
then do replace current ...
```

NAME

define type — define a new base data type

SYNOPSIS

define type typename (internallength = (number | variable),
[externallength = (number | variable),]
input = function, output = function
[, element = typename]
[, 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 C 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**.

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.

EXAMPLE

```
/* This command creates the box data type and then uses the type in a
   relation definition */

   define type box (internallength = 8,
          input = my_procedure_1, output = my_procedure_2)

   create MYBOXES (id = integer, description = box)
```

SEE ALSO

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

NAME

define view — construct a virtual class

SYNTAX

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 to POSTGRES. This view is not physically materialized; instead the rule system is used to support view processing as in [STON90]. Specifically, a 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 rule example1
  on delete to toyemp
  then do instead delete emp where emp.OID = current.OID
```

SEE ALSO

postquel(commands), create(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.

The "*" indicates a transitive closure and POSTGRES will run the command until it produces no further effect.

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).

BUGS

The code to support "*" is very buggy.

NAME

destroy — destroy existing classes

SYNOPSIS

destroy classname1 { , 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

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.

EXAMPLE

```
/* 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

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 about one minute.

On DECstations, you must use the "-G O" option when compiling object files to be loaded.

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 will not work until Version 3.

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 unavailable in Version 2.1.

NAME

purge — discard historical data

SYNOPSIS

purge classname [**before** abstime]

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 *after* 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 "January 1, 1989"
```

Retain only the current data in EMP

```
purge EMP
```

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).

BUGS

Remove aggregate is not implemented in Version 2.1.

NAME

remove function — remove a user defined function

SYNOPSIS

remove function functionname

DESCRIPTION

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

EXAMPLE

```
/* The following command will remove the square root function */
        remove function sqrt
```

SEE ALSO

define C function(commands).

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

```
/* The following 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 the 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 rule rule_name

DESCRIPTION

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

EXAMPLE

```
/* This example drops the rule example_1 */

  remove 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 be supported also.

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.

The * indicates a transitive closure and POSTGRES will run the command until it produces no further effect.

EXAMPLE

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

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

BUGS

The code to support ''*'' is very buggy.

NAME

retrieve — retrieve instances from a class

SYNTAX

retrieve[***]** [(**into** classname [archive_mode] | **portal** 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 a portal 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 named portals, retrieve passes data to an application without conversion to external format. 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.

The **sort** clause allows a user to specify that he wishes the instances sorted according to the corresponding operator. This operator must be a binary one returning a boolean. Multiple sort fields are allowed and are applied from left to right.

The "*" indicates a transitive closure, and POSTGRES will run the command until it produces no effect.

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 temp (e.all)
  from e in emp
  where e.sal > avg {emp.salary}
```

```
/* 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)
```

SEE ALSO

postquel(postquel), create(commands).

BUGS

"Retrieve into" does not delete duplicates in Version 2.1.

"Archive_mode" is not implemented in Version 2.1.

The code to support "*" is very buggy.

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 returned through an IPC channel.

This version of the documentation is based on the C library. A similar package exists for Common Lisp.

CONTROL 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.
- **PGOPTION** contains optional arguements to the POSTGRES back-end.

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
                                        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 */

char    *PQinitstr = NULL;      /* Initialization string passed to back-end */

int     PQtracep = 0 ;          /* 1 to print out front-end debugging messages */
```

QUERY

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

PQsetdb — Make the specified database the current database.

PQsetdb (dbname)
char *dbname;

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.

PQreset ()

Resets communication in case of errors.

PQfinish — Close communication ports with the back-end.

PQfinish ()

Terminates communications and frees up the memory taken up by the libpq buffer. PQexec — Submit a query to POSTGRES.

PQexec (query) char * query;

This function returns a status indicator or an error message.

PORTAL

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 empty string, 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 retrievel 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 O, then only return the number of asynchronous portals.

PQnames — Return all portal names.

```
void PQnames  ( 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;
```

PQrulep — Return 1 if an asynchronous portal.

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

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;
```

All values are returned as string. It is the programmer's responsibility to convert them to the correct type.

FUNCTIONS

The *copy* command in P 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(postquel).

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 backend.

This function waits until the backend has finished processing the copy. It should either be issued when the last string has been sent to the backend using PQputline() or when the last string has been received from the backend using 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.

TRACING

PQtrace — Enable tracing.

void PQtrace ()

PQuntrace — Disable tracing.

void PQuntrace ()

BUGS

Only 3 portals can be open at a time.

IPC glitches between the front-end and the back-end may cause a query to hang. When this happens try killing the query with a keyboard interrupt (^C). If this does not work, you may have to kill the process.

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

SAMPLE

```
/*
/*
 * 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");

   /* 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 ("0 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");

        /* Try out some other functions. */

        /* Print out the number of portals. */
        printf ("0umber 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 ();
    }
```

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; this feature will reappear in Version 2.02 in an improved way. 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.

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/{dbdb,local}.bki — template script

DESCRIPTION

Backend Interface (BKI) template script files are used to describe the construction of databases. The backend interface is a stripped-down version of postgres intended for setting up the first database, and other administrative tasks. It is *not* intended for use by humans.

This stripped-down backend reads special ".bki" files. "XXX.bki" represents any arbitrary file name. *Createdb* uses this type of file to direct the construction of the system catalogs. (In addition, the POSTGRES super-user may run scripts directly by running **backend** with commands that follow in the next section.) **Backend** interprets the sequence of commands and macro definitions found in template files in the manner similar to what is described below. In particular, this description will be easier to understand if the example in .../files/global1.bki.

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>
    with (name_1 collection_1 { , name_2 collection_2 , ... })

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

createdb(unix), template(files), .../src/support/backend.c

NAME

.../files/dayfile1 — POSTGRES login message

DESCRIPTION

The contents of the dayfile reflect user information of general system interest, and is more or less analogous to /etc/motd in UNIX. The file has no set format; its contents are simply returned as a string when the function dayfile() is called. Moreover the dayfile is not mandatory, and its absence will not generate errors of any sort; the same is true when the dayfile is present but not readable.

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.

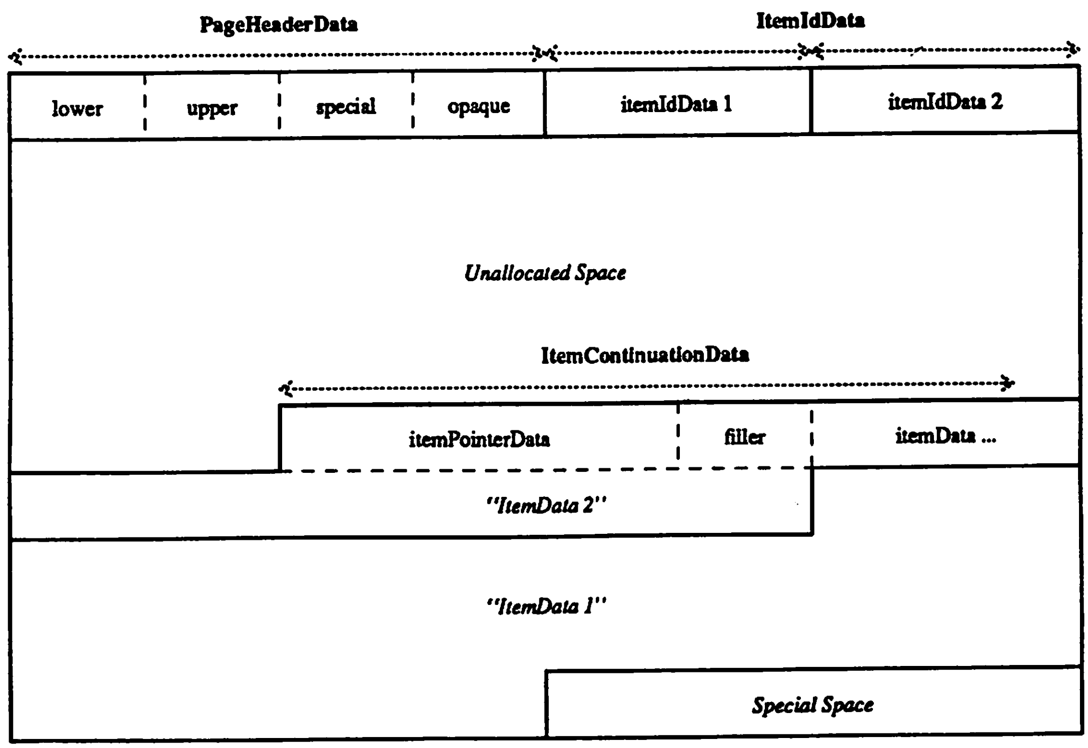

Diagram 1: Sample Page Layout

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 "createdb postgres" for the first time. 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 an AMI template script, but 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 in Version 2.1.

SEE ALSO

bki(files), createdb(unix)

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.