

LOCALIZATION OF E-GOVERNANCE PROJECT

PostgreSQL 8.3.0 Documentation

Prepared By:
Project Team,
The ePlatform Model
DIT, MoIC, RGoB

Contents

1. Introduction	2
2. Creating and accessing a Database	2
3. Data types	3
4. Functions and operators	6
5. The SQL Language	10
• CREATE TABLE	
• INSERT	
• VIEW TABLE	
• UPDATE TABLE	
• DELETE TABLE	
• ALTER TABLE	
6. Views	11
7. Default Values	11
8. Constraints	12
9. Privileges	14
10. Tablespace	14
11. Schemas	15
12. Inheritance	16
13. Table partitioning	17
14. Queries	19
• SELECT	
• FROM Clause	
• JOINS between Tables	
• Table and Column Aliases	
• Subqueries	
• WHERE clause	
• GROUP BY and HAVING Clause	
• DISTICT	
• ORDER BY Clause	
15. Arrays	23
16. Transactions	24
17. Indexes	25
18. Full text search	26
19. Functions	28
20. PL/pgSQL- SQL procedural language	30
• OVER VIEW	
• BASIC STATEMENTS	
• CONTROL STRUCTURES (IF ... THEN, LOOP, WHILE, FOR)	
• CURSORS	
• TRIGGER PROCEDURE	

1. Introduction

What is PostgreSQL?

PostgreSQL is a relational database management system (RDBMS). That means it is a system for managing data stored in relations. (Tables)

PostgreSQL is an open-source descendant of this original Berkeley code. It supports a large part of the SQL standard and offers many modern features like complex queries, foreign keys, Triggers, Views, transactional integrity, multi-version concurrency control etc.

2. Creating and accessing a Database

Creating a Database

Syntax:

```
create database [database_name];
```

Example: *create database test;*

Accessing a Database

Once you have created a database, you can access it by running the PostgreSQL interactive terminal program, called *psql*, which allows you to interactively enter, edit, and execute SQL commands. You can also use the command prompt in windows as follows:

<Path where postgres is stored> psql <database_name> <user>

(by default 'postgres' user can be used)

```
C:\Program Files\PostgreSQL\8.3\bin>psql test postgres
Welcome to psql 8.3.5, the PostgreSQL interactive terminal.
```

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

```
Warning: Console code page (437) differs from Windows code page (1252)
8-bit characters might not work correctly. See psql reference
page "Notes for Windows users" for details.
```

```
test=#
```

This would mean you are a database superuser, which is most likely the case if you installed PostgreSQL yourself.

Deleting a Database

```
Drop database [database_name];
```

Example: *drop database test;*

Note: This command physically removes all the tables associated with that database and it cannot be undone.

To list all the databases use the command: `\l`

To list all the tables use: `\d`

To view the columns in that table use: `\d <table_name>`

3. Data Types

PostgreSQL supports the standard SQL data types `int`, `smallint`, `real`, `double precision`, `char(N)`, `varchar(N)`, `date`, `time`, `timestamp`, and `interval`, as well as other types of general utility and a rich set of geometric types.

PostgreSQL has a rich set of native data types available to users. Users can also add new types to PostgreSQL using the [CREATE TYPE](#) command.

Numeric types

Name	Storage Size	Description	Range
<code>smallint</code>	2 bytes	small-range integer	-32768 to +32767
<code>integer</code>	4 bytes	usual choice for integer	-2147483648 to +2147483647
<code>bigint</code>	8 bytes	large-range integer	-9223372036854775808 to 9223372036854775807
<code>decimal</code>	variable	user-specified precision, exact	no limit
<code>numeric</code>	variable	user-specified precision, exact	no limit
<code>real</code>	4 bytes	variable-precision, inexact	6 decimal digits precision
<code>double precision</code>	8 bytes	variable-precision, inexact	15 decimal digits precision
<code>serial</code>	4 bytes	autoincrementing integer	1 to 2147483647
<code>bigserial</code>	8 bytes	large autoincrementing integer	1 to 9223372036854775807

Monetary types

The `money` type stores a currency amount with a fixed fractional precision. Input is accepted in a variety of formats, including integer and floating-point literals, as well as “typical” currency formatting, such as '\$1,000.00'. Output is generally in the latter form but depends on the locale.

Name	Storage Size	Description	Range
<code>money</code>	8 bytes	currency	-92233720368547758.08 to

Name	Storage Size	Description	Range
		amount	+92233720368547758.07

Character types

Name	Description
character varying(<i>n</i>), varchar(<i>n</i>)	variable-length with limit
character(<i>n</i>), char(<i>n</i>)	fixed-length, blank padded
text	variable unlimited length

Date / Time types

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

Name	Storage Size	Description	Low Value	High Value	Resolution
timestamp [(<i>p</i>)] [without time zone]	8 bytes	both date and time	4713 BC	5874897 AD	1 microsecond / 14 digits
timestamp [(<i>p</i>)] with time zone	8 bytes	both date and time, with time zone	4713 BC	5874897 AD	1 microsecond / 14 digits
interval [(<i>p</i>)]	12 bytes	time intervals	-178000000 years	178000000 years	1 microsecond / 14 digits
date	4 bytes	dates only	4713 BC	5874897 AD	1 day
time [(<i>p</i>)] [without time zone]	8 bytes	times of day only	00:00:00	24:00:00	1 microsecond / 14 digits
time [(<i>p</i>)] with time zone	12 bytes	times of day only, with time zone	00:00:00+1459	24:00:00-1459	1 microsecond / 14 digits

Note: time, timestamp, and interval accept an optional precision value *p* which specifies the number of fractional digits retained in the seconds field. By default, there is no explicit bound on precision. The allowed range of *p* is from 0 to 6 for the timestamp and interval types.

Boolean types

Valid literal values for the “**True**” state are: ‘**TRUE**’, ‘**t**’, ‘**true**’, ‘**y**’, ‘**yes**’, ‘**1**’. For the “**false**” state, the values used are: ‘**FALSE**’, ‘**f**’, ‘**false**’, ‘**N**’, ‘**no**’, ‘**0**’.

Using the key words **TRUE** and **FALSE** is preferred (and SQL-compliant).

Enumerated types

Enumerated (**enum**) types are data types that are comprised of a static, predefined set of values with a specific order. They are equivalent to the enum types in a number of programming languages. An example of an enum type might be the days of the week, or a set of status values for a piece of data.

Enum types are created using the [CREATE TYPE](#) command.

Once created, the enum type can be used in table and function definitions much like any other type:

Example: Basic Enum Usage

```
test=# create type mood as ENUM ('sad', 'ok', 'happy');
CREATE TYPE
test=# create table GNH(name varchar(20), curr_mood mood);
CREATE TABLE
test=# insert into GNH values('Sonam', 'ok');
INSERT 0 1
test=# insert into GNH values('Yeshey', 'happy');
INSERT 0 1
test=# insert into GNH values('Tashi', 'sad');
INSERT 0 1
test=# select * from GNH;
 name | curr_mood
-----+-----
Sonam | ok
yeshey | happy
Tashi | sad
(3 rows)
test=#
```

Note: The ordering of the values in an enum type is the order in which the values were listed when the type was declared. All standard comparison operators and related aggregate functions are supported for enums. For example, 'ok' will have higher precedence than 'sad'.

4. Functions and Operators

Logical Operators

The usual logical operators are available: AND, OR, NOT.

Comparison Operators

Operator	Description
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to
=	equal
<> or !=	not equal

Note: In addition to the comparison operators, the special **BETWEEN** construct is available.

String functions and operators

Function	Return Type	Description	Example	Result
<i>string</i> <i>string</i>	text	String concatenation	'Post' 'greSQL'	PostgreSQL
char_length(<i>string</i>) or character_length(<i>string</i>)	int	Number of characters in string	char_length('jose')	4
lower(<i>string</i>)	text	Convert string to lower case	lower('TOM')	tom
overlay(<i>string</i> placing <i>string</i> from int [for int])	text	Replace substring	overlay('Txxxxas' placing 'hom' from 2 for 4)	Thomas
position(<i>substring</i> in <i>string</i>)	int	Location of specified substring	position('om' in 'Thomas')	3
substring(<i>string</i> [from int] [for int])	text	Extract substring	substring('Thomas' from 2 for 3)	hom
substring(<i>string</i> from <i>pattern</i> for <i>escape</i>)	text	Extract substring matching SQL regular expression.	substring('Thomas' from '%#"o_a#"_' for '#')	oma
trim([leading trailing both] [<i>characters</i>] from <i>string</i>)	text	Remove the longest string containing only the <i>characters</i> (a space by default) from the	trim(both 'x' from 'xTomxx')	Tom

Function	Return Type	Description	Example	Result
		start/end/both ends of the <i>string</i>		
upper(<i>string</i>)	text	Convert string to uppercase	upper('tom')	TOM
repeat(string text, number int)	text	Repeat string the specified number of times	repeat('Pg', 4)	PgPgPgPg
replace(string text, from text, to text)	text	Replace all occurrences in string of substring from with substring to	replace('abcdefabcdef', 'cd', 'XX')	abXXefabXXef
strpos(string, substring)	int	Location of specified substring	strpos('high', 'ig')	2
substr(string, from [, count])	text	Extract substring (same as substring(string from from for count))	substr('alphabet', 3, 2)	ph

Pattern Matching

Every *pattern* defines a set of strings. The **LIKE** expression returns **true** if the *string* is contained in the set of strings represented by *pattern*. As expected, the **NOT LIKE** expression returns **false** if LIKE returns true, and vice versa.

string **LIKE** *pattern* [ESCAPE *escape-character*]

string **NOT LIKE** *pattern* [ESCAPE *escape-character*]

Example:

```
'abc' LIKE 'abc' true
'abc' LIKE 'a%' true
'abc' LIKE '_b_' true
'abc' LIKE 'c' false
```

An underscore (_) in *pattern* stands for (matches) any single character; a percent sign (%) matches any string of zero or more characters. Likewise the various regular expression quantifiers are given in the table below.

Regular Expression Quantifiers

Quantifier	Matches
------------	---------

Quantifier	Matches
*	a sequence of 0 or more matches of the atom
+	a sequence of 1 or more matches of the atom
?	a sequence of 0 or 1 matches of the atom
{ <i>m</i> }	a sequence of exactly <i>m</i> matches of the atom
{ <i>m</i> ,}	a sequence of <i>m</i> or more matches of the atom
{ <i>m</i> , <i>n</i> }	a sequence of <i>m</i> through <i>n</i> (inclusive) matches of the atom; <i>m</i> cannot exceed <i>n</i>
*?	non-greedy version of *
+?	non-greedy version of +
??	non-greedy version of ?
{ <i>m</i> }?	non-greedy version of { <i>m</i> }
{ <i>m</i> ,}?	non-greedy version of { <i>m</i> ,}
{ <i>m</i> , <i>n</i> }?	non-greedy version of { <i>m</i> , <i>n</i> }
^	matches at the beginning of the string
\$	matches at the end of the string
(?=re)	positive lookahead matches at any point where a substring matching re begins (AREs only)
(?!re)	negative lookahead matches at any point where no substring matching re begins (AREs only)

Current Date/Time

PostgreSQL provides a number of functions that return values related to the current date and time. These SQL-standard functions all return values based on the start time of the current transaction:

CURRENT_DATE

CURRENT_TIME

CURRENT_TIMESTAMP

CURRENT_TIME(*precision*)

CURRENT_TIMESTAMP(*precision*)

LOCALTIME

LOCALTIMESTAMP

LOCALTIME(*precision*)

LOCALTIMESTAMP(*precision*)

CURRENT_TIME and CURRENT_TIMESTAMP deliver values with time zone; LOCALTIME and LOCALTIMESTAMP deliver values without time zone.

CURRENT_TIME, CURRENT_TIMESTAMP, LOCALTIME, and LOCALTIMESTAMP can optionally be given a precision parameter, which causes the result to be rounded to that many fractional digits in the seconds field. Without a precision parameter, the result is given to the full available precision.

Example:

```
test=# select current_time;
      timetz
```

```
-----
15:54:20.671+06
(1 row)
```

```
test=# select current_date;
      date
```

```
-----
2009-01-02
(1 row)
```

```
test=#
```

General-Purpose Aggregate Functions

Function	Argument Type	Return Type	Description
avg(expression)	smallint, int, bigint, real, double precision, numeric, or interval	numeric for any integer type argument, double precision for a floating-point argument, otherwise the same as the argument data type	the average (arithmetic mean) of all input values
count(*)		bigint	number of input rows
max(expression)	any array, numeric, string, or date/time type	same as argument type	maximum value of expression across all input values
min(expression)	any array, numeric, string, or date/time type	same as argument type	minimum value of expression across all input values
sum(expression)	smallint, int, bigint, real, double precision, numeric, or interval	bigint for smallint or int arguments, numeric for bigint arguments, double precision for floating-point arguments, otherwise the same as the argument data type	sum of expression across all input values

5. The SQL Language

CREATE TABLE

Syntax:

```
CREATE TABLE [table_name] (column1    datatype,  
                             column2    datatype,  
                             column n   datatype  
                             );
```

Example:

```
test=# CREATE TABLE weather (city varchar(30),  
test(# temp_lo int,  
test(# temp_hi int,  
test(# prcp real,  
test(# date date);  
CREATE TABLE  
test=#
```

Note: White space (i.e., spaces, tabs, and newlines) can be used freely in SQL commands. That means you can type the command aligned differently than above, or even all on one line. `psql` will recognize that the command is not terminated until the semicolon. Two dashes (“--”) introduce comments.

INSERT INTO TABLE

Syntax:

```
INSERT INTO [table_name] VALUES('value1', 'value2', ..... 'value n');
```

Example:

```
test=# INSERT INTO weather VALUES ('San Francisco', 46, 50, '1994-11-27');  
INSERT 0 1  
test=# select * from weather;  
  city  | temp_lo | temp_hi | date  
-----+-----+-----+-----  
San Francisco |    46 |    50 | 1994-11-27  
(1 row)
```

Note: You could also have used **COPY** to load large amounts of data from flat-text files. This is usually faster because the **COPY** command is optimized for this application while allowing less flexibility than **INSERT**. For example: **COPY weather FROM '/home/user/weather.txt'**; where the file name for the source file must be available to the backend server machine, not the client, since the backend server reads the file directly.

VIEW TABLE

An SQL **SELECT** statement is used to retrieve data from a table.

Syntax:

```
SELECT * FROM [table_name];
```

Example: `SELECT * FROM weather;`
Here `*` is a shorthand for “all columns”. So the same result would be had with:
`SELECT city, temp_lo, temp_hi, prcp, date FROM weather;`

UPDATE TABLE

Syntax:

```
UPDATE TABLE [table_name]
SET [expression]
WHERE condition;
```

DELETE TABLE

Syntax: `DROP TABLE [table_name];`

ALTER TABLE

Syntax:

```
ALTER TABLE [table_name]
ADD columnName      DataType;
```

```
ALTER TABLE [table_name]
DROP columnName;
```

6. VIEWS

Suppose the combined listing of weather records and city location is of particular interest to your application, but you do not want to type the query each time you need it. You can create a *view* over the query, which gives a name to the query that you can refer to like an ordinary table:

Example:

```
CREATE VIEW myview AS
SELECT city, temp_lo, temp_hi, prcp, date, location
FROM weather, cities
WHERE city = name;

SELECT * FROM myview;
```

Making liberal use of views is a key aspect of good SQL database design. Views allow you to encapsulate the details of the structure of your tables, which might change as your application evolves, behind consistent interfaces.

7. DEFAULT VALUES

A column can be assigned a default value. When a new row is created and no values are specified for some of the columns, those columns will be filled with their respective default values. If no default value is declared explicitly, the default value is the **null value**. This usually makes sense because a null value can be considered to represent unknown data.

```
Example: CREATE TABLE products (product_no integer,  
                                   name text,  
                                   price numeric DEFAULT 9.99  
                                   );
```

8. CONSTRAINTS

SQL allows you to define various constraints on columns and tables. Constraints give you as much control over the data in your tables as you wish. If a user attempts to store data in a column that would violate a constraint, an error is raised.

Check Constraints

It allows you to specify that the value in a certain column must satisfy a Boolean (truth-value) expression. For instance, to require positive product prices, you could use:

```
Example: CREATE TABLE products ( product_no integer,  
                                   name text,  
                                   price numeric CHECK (price > 0)  
                                   );
```

Not-Null Constraints

A not-null constraint simply specifies that a column must not accept a null value.

```
Example: CREATE TABLE products ( product_no integer NOT NULL,  
                                   name text NOT NULL,  
                                   price numeric  
                                   );
```

Unique Constraints

Unique constraints ensure that the data contained in a column or a group of columns is unique with respect to all the rows in the table.

```
Example: CREATE TABLE products ( product_no integer UNIQUE,  
                                   name text,  
                                   price numeric  
                                   );
```

Primary Keys

Technically, a primary key constraint is simply a combination of a unique constraint and a not-null constraint. It is used to uniquely identify each row in a table. It's a good practice to keep a primary key for each table.

```
Example: CREATE TABLE products ( product_no integer PRIMARY KEY,  
                                   name text,  
                                   price numeric  
                                   );
```

Foreign Keys

A foreign key constraint specifies that the values in a column (or a group of columns) must match the values appearing in some row of another table. We say this maintains the **referential integrity** between two related tables. A foreign key must reference columns that either are a primary key or form a unique constraint.

Example: Say you have the **product table** that we have used several times already. Let's also assume you have a table storing orders of those products. We want to ensure that the **orders table** only contains orders of products that actually exist. So we define a **foreign key constraint** in the orders table that references the products table:

Example:

```
CREATE TABLE orders ( order_id integer PRIMARY KEY,
                      product_no integer,
                      quantity integer
                      FOREIGN KEY product_no REFERENCES products (product_no)
                      );
```

Now it is impossible to create orders with product_no entries that do not appear in the products table.

A table can contain more than one foreign key constraint. This is used to implement **many-to-many relationships** between tables. In such tables restricting and cascading deletes are the two most common options. **RESTRICT** prevents deletion of a referenced row and **CASCADE** specifies that when a referenced row is deleted, row(s) referencing it should be automatically deleted as well.

Analogous to ON DELETE there is also ON UPDATE which is invoked when a referenced column is changed (updated). The possible actions are the same.

Example: Assuming products table and orders table are already in place with many-to-many relationship. The third table is prod_order table as shown below:

```
CREATE TABLE prod_order (
                      product_no integer REFERENCES products ON DELETE RESTRICT,
                      order_id integer REFERENCES orders ON DELETE CASCADE,
                      quantity integer,
                      PRIMARY KEY (product_no, order_id)
                      );
```

9. Privileges

When you create a database object, you become its owner. By default, only the owner of an object can do anything with the object. In order to allow other users to use it, **privileges** must be granted.

There are several different privileges: **SELECT, INSERT, UPDATE, DELETE, REFERENCES, TRIGGER, CREATE, CONNECT, TEMPORARY, EXECUTE, and USAGE**. The privileges applicable to a particular object vary depending on the object's type (table, function, etc). However, the right to modify or destroy an object is always the privilege of the owner only. To assign **privileges**, the **GRANT** command is used.

Syntax:

```
GRANT { { SELECT | INSERT | UPDATE | DELETE | REFERENCES | TRIGGER }  
[,...] | ALL [ PRIVILEGES ] }  
ON [ TABLE ] tablename [, ...]  
TO { [ GROUP ] rolename | PUBLIC } [, ...] [ WITH GRANT OPTION ]
```

For example, if Tashi is an existing user, and accounts is an existing table, the privilege to update the table can be granted with:

```
GRANT UPDATE ON accounts TO Tashi;
```

To **revoke** a **privilege**, use the fittingly named **REVOKE** command:

```
REVOKE ALL ON accounts FROM PUBLIC;
```

Writing **ALL** in place of a specific privilege grants all privileges that are relevant for the object type. The special “user” name **PUBLIC** can be used to grant a privilege to every user on the system. Also, “**group**” roles can be set up to help manage privileges when there are many users of a database.

10. Tablespace

Tablespaces in PostgreSQL allow database administrators to define locations in the file system where the files representing database objects can be stored. Once created, a tablespace can be referred to by name when creating database objects. By using tablespaces, you can control the disk layout of a PostgreSQL installation, like:

- if the partition or volume on which the cluster was initialized runs out of space and cannot be extended, a tablespace can be created on a different partition and used until the system can be reconfigured.
- tablespaces allow an administrator to use knowledge of the usage pattern of database objects to optimize performance.

For example, an index which is very heavily used can be placed on a very fast, highly available disk, such as an expensive solid state device. At the same time a table storing archived data which is rarely used or not performance critical could be stored on a less expensive, slower disk system.

```
Example: CREATE TABLESPACE fastspace LOCATION '/mnt/sda1/postgresql/data';
```

The location must be an existing, empty directory that is owned by the PostgreSQL system user. All objects subsequently created within the tablespace will be stored in files underneath this directory.

```
Example: To create a table in Tablespace space1.
```

```
CREATE TABLE foo(i int) TABLESPACE space1;
```

Two tablespaces are automatically created by **initdb**. The `pg_global` tablespace is used for shared system catalogs. The `pg_default` tablespace is the default tablespace of the `template1` and `template0` databases (and, therefore, will be the default tablespace for other databases as well, unless overridden by a `TABLESPACE` clause in `CREATE DATABASE`).

Once created, a tablespace can be used from any database, provided the requesting user has sufficient privilege. This means that a tablespace cannot be dropped until all objects in all databases using the tablespace have been removed.

To remove an empty tablespace, use the **DROP TABLESPACE** command.

11. Schemas

A database contains one or more named **schemas**, which in turn contain tables. Schemas also contain other kinds of named objects, including data types, functions, and operators. The same object name can be used in different schemas without conflict; for example, both `schema1` and `myschema` can contain tables named `mytable`. Unlike databases, schemas are not rigidly separated, a user can access objects in any of the schemas in the database he is connected to, if he has privileges to do so.

Schemas are used:

- To allow many users to use one database without interfering with each other.
- To organize database objects into logical groups to make them more manageable.
- Third-party applications can be put into separate schemas so they cannot collide with the names of other objects.

Creating a Schema

```
Example: CREATE SCHEMA myschema;
```

To create a table in the new schema, use:

```
CREATE TABLE myschema.mytable (...);
```

To access objects in a schema, write a **qualified name** consisting of the schema name and table name separated by a dot:

schema.table

To drop a schema if it's empty (all objects in it have been dropped), use:

```
DROP SCHEMA myschema;
```

To drop a schema including all contained objects, use:

```
DROP SCHEMA myschema CASCADE;
```

The Public Schema

In the previous sections we created tables without specifying any schema names. By default, such tables (and other objects) are automatically put into a schema named “**public**”. Every new database contains such a schema. Thus, the following are equivalent:

```
CREATE TABLE products (...);           and  
CREATE TABLE public.products (...);
```

Schemas and Privileges

By default, users cannot access any objects in schemas they do not own. To allow that, the owner of the schema needs to grant the **USAGE** privilege on the schema.

Note that by default, everyone has **CREATE** and **USAGE** privileges on the schema public. This allows all users that are able to connect to a given database to create objects in its public schema. If you do not want to allow that, you can revoke that privilege:

```
Example: REVOKE CREATE ON SCHEMA public FROM PUBLIC;
```

12. Inheritance

PostgreSQL implements table inheritance, which can be a useful tool for database designers.

```
Example: CREATE TABLE cities (name      text,  
                                population float,  
                                altitude  int  
                                );  
CREATE TABLE capitals (state      char(2)) INHERITS (cities);
```

In this case, the capitals table **inherits** all the columns of its parent table, cities. State capitals also have an extra column, state, that shows their state.

A **query** can reference either all rows of a table or all rows of a table plus all of its descendant tables.

All **check constraints** and **not-null constraints** on a parent table are automatically **inherited** by its children. Other types of constraints (**unique, primary key, and foreign key constraints**) are **not inherited**. Table access permissions are also not automatically inherited.

A table can inherit from more than one parent table, in which case it has the union of the columns defined by the parent tables. Any columns declared in the child table's definition are added to these. If the same column name appears in multiple parent tables, or in both the parent table and the child's definition, then these columns are **“merged”** so that there is only one such column in the child table. To be merged, columns must have the same data types, else an error is raised. A parent table cannot be dropped while any of its children remain. Neither can columns of child tables be dropped or altered if they are inherited from any parent tables. If you wish to remove a table and all of its descendants, one easy way is to drop the parent table with the **CASCADE** option.

Note: A serious limitation of the inheritance feature is that indexes (including unique constraints) and foreign key constraints only apply to single tables, not to their inheritance children. This is true on both the referencing and referenced sides of a foreign key constraint.

13. Table partitioning

Partitioning refers to splitting what is logically one large table into smaller physical pieces. Partitioning can provide several benefits especially for very large tables. The performance of various queries such as access, search, update, alter and delete can be improved dramatically by keeping the heavily accessed rows of the tables in one partition or in small numbers of partitions.

Currently, PostgreSQL supports **partitioning** via **table inheritance**. Each partition must be created as a child table of a single parent table. The parent table itself is normally **empty**, it exists just to represent the entire data set. The following forms of partitioning can be implemented in PostgreSQL:

Range Partitioning:

The table is partitioned into **“ranges”** defined by a key column or set of columns, with no overlap between the ranges of values assigned to different partitions. For example one might partition by date ranges, or by ranges of identifiers for particular business objects.

List Partitioning:

The table is partitioned by explicitly listing which key values appear in each partition.

Implementing Partitioning

To set up a partitioned table, do the following:

- Create the **“master”** table, from which all of the partitions will inherit. This table will contain no data. Do not define any check constraints on this table, unless you intend

them to be applied equally to all partitions. There is no point in defining any indexes or unique constraints on it, either.

Example: A database for a large ice cream company

```
CREATE TABLE measurement (city_id    int not null,
                           logdate    date not null,
                           peaktemp   int,
                           unitsales  int
                           );
```

- Create several “**child**” tables that each inherit from the master table. Normally, these tables will not add any columns to the set inherited from the master. We will refer to the child tables as partitions, though they are in every way normal PostgreSQL tables.

Example: We create one partition for each active month for ease of management.

```
CREATE TABLE measurement_y2008m01 ( ) INHERITS (measurement);
CREATE TABLE measurement_y2008m02 ( ) INHERITS (measurement);
-----
CREATE TABLE measurement_y2008m12 ( ) INHERITS (measurement);
```

- Add table constraints to the partition tables to define the allowed key values in each partition. Ensure that the constraints guarantee that there is no overlap between the key values permitted in different partitions.

Example:

```
CREATE TABLE measurement_y2008m02 (
CHECK( logdate >= DATE '2008-02-01' AND logdate < DATE '2008-03-01'))
INHERITS (measurement);
```

- For each partition, create an index on the key column(s), as well as any other indexes you might want. (The key index is not strictly necessary, but if you intend the key values to be unique then you should always create a unique or primary-key constraint for each partition.)

Example:

```
CREATE INDEX measurement_y2008m01_logdate ON measurement_y2006m01
(logdate);
CREATE INDEX measurement_y2008m02_logdate ON measurement_y2006m02
(logdate);
...
CREATE INDEX measurement_y2008m12_logdate ON measurement_y2007m12
(logdate);
```

- Optionally, define a trigger or rule to redirect data inserted into the master table to the appropriate partition.

- Ensure that the [constraint exclusion](#) configuration parameter is enabled in postgresql.conf. Without this, queries will not be optimized as desired.

Managing Partitions

The simplest option for removing old data is simply to drop the partition that is no longer necessary. This can very quickly delete millions of records because it doesn't have to individually delete every record.

```
DROP TABLE measurement_y2006m02;
```

Another option that is often preferable is to remove the partition from the partitioned table but retain access to it as a table in its own right:

```
ALTER TABLE measurement_y2006m02 NO INHERIT measurement;
```

Similarly we can add a new partition to handle new data.

14. Queries

The process of retrieving or the command to retrieve data from a database is called a **query**. In SQL the **SELECT** command is used to specify queries as you have already seen in several examples give above.

```
SELECT select_list FROM table_expression [sort_specification]
```

Table Expressions

A **table expression** computes a table. The table expression contains a **FROM** clause that is optionally followed by **WHERE**, **GROUP BY**, and **HAVING** clauses.

The FROM Clause

The **FROM Clause** derives a table from one or more other tables given in a comma-separated table reference list.

```
FROM table_reference [, table_reference [, ...]]
```

JOINS between Tables

A query that accesses multiple rows of the same or different tables at one time is called a **join query**.

JOIN Types: CROSS JOIN and Qualified JOIN (INNER JOIN, LEFT OUTER JOIN, RIGHT OUTER JOIN, FULL OUTER JOIN).

The words INNER and OUTER are optional in all forms. INNER is the default; LEFT, RIGHT, and FULL imply an outer join. The **ON** clause is the most general kind of join condition, it takes a Boolean value expression of the same kind as is used in a WHERE clause. A pair of rows from *T1* and *T2* match if the **ON** expression evaluates to true for them.

USING is a shorthand notation; it takes a comma-separated list of column names, which the joined tables must have in common, and forms a join condition specifying equality of each of these pairs of columns. Furthermore, the output of a **JOIN USING** has one column for each of the equated pairs of input columns, followed by all of the other columns from each table. Thus,

USING (a, b, c) is equivalent to **ON (t1.a = t2.a AND t1.b = t2.b AND t1.c = t2.c)** with the exception that if ON is used there will be two columns a, b, and c in the result, whereas with USING there will be only one of each.

Finally, **NATURAL** is a shorthand form of USING: it forms a USING list consisting of exactly those column names that appear in both input tables. As with USING, these columns appear only once in the output table.

CROSS JOIN:

T1 CROSS JOIN *T2*

The resulting table will contain a row consisting of all columns in *T1* followed by all columns in *T2*. If the tables have N and M rows respectively, the joined table will have N * M rows.

FROM *T1* CROSS JOIN *T2* is equivalent to FROM *T1*, *T2*. It is also equivalent to FROM *T1* INNER JOIN *T2*

Example: Say we have two tables *T1* and *T2* as follows:

Table *T1*

num | name

----+-----

1 | a

2 | b

3 | c

And table *T2* as

num | value

----+-----

1 | xxx

3 | yyy

5 | zzz

SELECT * FROM t1 CROSS JOIN t2;

num | name | num | value

----+-----+-----+-----

1 | a | 1 | xxx

1 | a | 3 | yyy

1 | a | 5 | zzz

2 | b | 1 | xxx

2 | b | 3 | yyy

2 | b | 5 | zzz

3 | c | 1 | xxx

3 | c | 3 | yyy

3 | c | 5 | zzz

(9 rows)

INNER JOIN

For each row R1 of T1, the joined table has a row for each row in T2 that satisfies the join condition with R1.

Example: `SELECT * FROM t1 INNER JOIN t2 ON t1.num = t2.num;`

```
num | name | num | value
```

```
-----+-----+-----
```

```
1 | a | 1 | xxx
```

```
3 | c | 3 | yyy
```

```
(2 rows)
```

LEFT OUTER JOIN

First, an inner join is performed. Then, for each row in T1 that does not satisfy the join condition with any row in T2, a joined row is added with null values in columns of T2. Thus, the joined table unconditionally has at least one row for each row in T1.

Example: `SELECT * FROM t1 LEFT JOIN t2 USING (num);`

```
num | name | value
```

```
-----+-----+-----
```

```
1 | a | xxx
```

```
2 | b |
```

```
3 | c | yyy
```

```
(3 rows)
```

RIGHT OUTER JOIN

First, an inner join is performed. Then, for each row in T2 that does not satisfy the join condition with any row in T1, a joined row is added with null values in columns of T1. This is the converse of a left join: the result table will unconditionally have a row for each row in T2.

Example: `SELECT * FROM t1 RIGHT JOIN t2 ON t1.num = t2.num;`

```
num | name | num | value
```

```
-----+-----+-----
```

```
1 | a | 1 | xxx
```

```
3 | c | 3 | yyy
```

```
 | | 5 | zzz
```

```
(3 rows)
```

FULL OUTER JOIN

First, an inner join is performed. Then, for each row in T1 that does not satisfy the join condition with any row in T2, a joined row is added with null values in columns of T2. Also, for each row of T2 that does not satisfy the join condition with any row in T1, a joined row with null values in the columns of T1 is added.

Example: `SELECT * FROM t1 FULL JOIN t2 ON t1.num = t2.num;`

```
num | name | num | value
```

```
-----+-----+-----
```

```
1 | a | 1 | xxx
```

```
2 | b | |
```

```
3 | c | 3 | yyy
```

```
 | | 5 | zzz
```

```
(4 rows)
```

Table and Column Aliases

A temporary name can be given to tables and complex table references to be used for references to the derived table in the rest of the query. This is called a *table alias*.

Syntax:

FROM *table_reference* AS *alias* or FROM *table_reference* *alias*

Example:

```
SELECT * FROM some_very_long_table_name AS a JOIN another_fairly_long_name AS b ON a.id = b.num;
```

Subqueries

A subquery is a query within a query.

Subquery Expressions

EXISTS

EXISTS (*subquery*)

Example:

```
SELECT col1 FROM tab1  
WHERE EXISTS(SELECT 1 FROM tab2 WHERE col2 = tab1.col2);
```

IN

expression **IN** (*subquery*)

NOT IN

expression **NOT IN** (*subquery*)

WHERE Clause

Syntax:

WHERE *search_condition*

where *search_condition* is any value expression that returns a value of type boolean.

The GROUP BY and HAVING Clauses

After passing the WHERE filter, the derived input table might be subject to grouping, using the GROUP BY clause, and elimination of group rows using the HAVING clause.

Syntax for GROUP BY:

```
SELECT select_list FROM ...[WHERE ...] GROUP BY grouping_column_reference [,  
grouping_column_reference]...
```

Syntax for GROUP BY HAVING:

```
SELECT select_list FROM ... [WHERE ...] GROUP BY ... HAVING boolean_expression;
```

DISTINCT

After the select list has been processed, the result table can optionally be subject to the elimination of duplicate rows. The DISTINCT key word is written directly after SELECT to specify this:

Syntax: SELECT **DISTINCT** *select_list* ...

ORDER BY Clause

After a query has produced an output table, it can be optionally sorted.

The **ORDER BY** clause specifies the sort order:

Syntax:

```
SELECT select_list
FROM table_expression
ORDER BY sort_expression1 [ASC | DESC] [NULLS { FIRST | LAST }]
[, sort_expression2 [ASC | DESC] [NULLS { FIRST | LAST }] ...]
```

ASC or DESC keyword is to set the sort direction to ascending or descending. ASC order is the default. The NULLS FIRST and NULLS LAST options can be used to determine whether nulls appear before or after non-null values in the sort ordering. By default, NULLS FIRST is the default for DESC order, and NULLS LAST otherwise.

Note: The ordering options are considered independently for each sort column. For example ORDER BY x, y DESC means ORDER BY x ASC, y DESC, which is not the same as ORDER BY x DESC, y DESC.

15. Arrays

PostgreSQL allows columns of a table to be defined as variable-length multidimensional **arrays**. Arrays of any built-in or user-defined base type, enum type, or composite type can be created. However, the current implementation does not enforce the array size limits — the behavior is the same as for arrays of unspecified length.

Example:

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

Array value Input

To write an array value as a literal constant, enclose the element values within curly braces and separate them by commas.

Example:

```
INSERT INTO sal_emp VALUES ('Bill', {10000, 11000, 12000, 13000}',
                              {{{"meeting", "lunch"}, {"training", "presentation"}}});
```

```
test=# select * from sal_emp;
```

```
name | pay_by_quarter | schedule
```

```
-----+-----
```

```
Bill | {10000,11000,12000,13000} | {{{meeting,lunch},{training,presentation}}}
```

```
(1 row)
```

Accessing Arrays

The array subscript numbers are written within square brackets. By default PostgreSQL uses the one-based numbering convention for arrays, that is, an array of n elements **starts** with **array[1]** and **ends** with **array[n]**.

Example:

```
SELECT pay_by_quarter[3] FROM sal_emp;
pay_by_quarter
-----
10000
(2 rows)
```

Modifying Arrays

An array value can be replaced completely:

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

or using the ARRAY expression syntax:

```
UPDATE sal_emp SET pay_by_quarter = ARRAY[25000,25000,27000,27000]
WHERE name = 'Carol';
```

16. Transactions

Transactions are a fundamental concept of all database systems. The essential point of a transaction is that it bundles multiple steps into a single, all-or-nothing operation. The intermediate states between the steps are not visible to other concurrent transactions, and if some failure occurs that prevents the transaction from completing, then none of the steps affect the database at all.

We also want a guarantee that once a transaction is completed and acknowledged by the database system, it has indeed been permanently recorded and won't be lost even if a crash ensues shortly thereafter.

Another important property of transactional databases is closely related to the notion of atomic updates: when multiple transactions are running concurrently, each one should not be able to see the incomplete changes made by others.

In PostgreSQL, a transaction is set up by surrounding the SQL commands of the transaction with BEGIN and COMMIT commands.

Example: *Our banking transaction would actually look like:*

```
BEGIN;
UPDATE accounts SET balance = balance - 100.00
WHERE name = 'Alice';
--- etc etc
COMMIT;
```

If, partway through the transaction, we decide we do not want to commit (perhaps we just noticed that Alice's balance went negative), we can issue the command ROLLBACK instead of COMMIT, and all our updates so far will be canceled.

It's possible to control the statements in a transaction in a more granular fashion through the use of *savepoints*. Savepoints allow you to selectively discard parts of the transaction, while committing the rest. After defining a savepoint with `SAVEPOINT`, you can if needed roll back to the savepoint with `ROLLBACK TO`. All the transaction's database changes between defining the savepoint and rolling back to it are discarded, but changes earlier than the savepoint are kept.

After rolling back to a savepoint, it continues to be defined, so you can roll back to it several times. Conversely, if you are sure you won't need to roll back to a particular savepoint again, it can be released, so the system can free some resources. Keep in mind that either releasing or rolling back to a savepoint will automatically release all savepoints that were defined after it.

All this is happening within the transaction block, so none of it is visible to other database sessions. When and if you commit the transaction block, the committed actions become visible as a unit to other sessions, while the rolled-back actions never become visible at all.

17. Indexes

Indexes are a common way to enhance database performance. An index allows the database server to find and retrieve specific rows much faster than it could do without an index. But indexes also add overhead to the database system as a whole, so they should be used sensibly.

Example:

```
CREATE INDEX test1_id_index ON test1 (id);
```

Note: The name `test1_id_index` can be chosen freely, but you should pick something that enables you to remember later what the index was for.

To remove an index, use the **DROP INDEX** command. Indexes can be added to and removed from tables at any time.

Once an index is created, no further intervention is required, the system will update the index when the table is modified, and it will use the index in queries when it thinks this would be more efficient than a sequential table scan.

Indexes can also benefit **UPDATE** and **DELETE** commands with search conditions. Indexes can moreover be used in join searches. Thus, an index defined on a column that is part of a join condition can significantly speed up queries with joins.

PostgreSQL provides several index types: **B-tree**, **Hash**, **GiST** and **GIN**. Each index type uses a different algorithm that is best suited to different types of queries. By default, the **CREATE INDEX** command will create a B-tree index, which fits the most common situations.

B-trees can handle equality and range queries on data that can be sorted into some ordering. In particular, the PostgreSQL query planner will consider using a B-tree index whenever an indexed column is involved in a comparison using one of these operators: `<`, `<=`, `=`, `>=`, `>`.

Constructs equivalent to combinations of these operators, such as **BETWEEN** and **IN**, can also be implemented with a B-tree index search. Also, an **IS NULL** condition on an index column can be used with a B-tree index.

Multicolumn Indexes

An index can be defined on more than one column of a table. Currently, only the B-tree and GiST index types support multicolumn indexes. Up to 32 columns can be specified. (This limit can be altered when building PostgreSQL; see the file `pg_config_manual.h`.)

Multicolumn indexes should be used sparingly. In most situations, an index on a single column is sufficient and saves space and time. Indexes with more than three columns are unlikely to be helpful unless the usage of the table is extremely stylized

A multicolumn B-tree index or GiST index can be used with query conditions that involve any subset of the index's columns.

18. Full Text Search

Full Text Search provides the capability to identify natural-language *documents* that satisfy a *query*, and optionally to sort them by relevance to the query. The most common type of search is to find all documents containing given *query terms* and return them in order of their *similarity* to the query. Notions of query and similarity are very flexible and depend on the specific application. The simplest search considers query as a set of words and similarity as the frequency of query words in the document.

Full text indexing allows documents to be *preprocessed* and an index saved for later rapid searching. Preprocessing includes:

- *Parsing documents into tokens(e.g. numbers, words, complex words, email addresses, so that they can be processed differently)*
- *Converting tokens into lexemes(A lexeme is a string, just like a token, but it has been normalized so that different forms of the same word are made alike)*
- *Storing preprocessed documents optimized for searching(each document can be represented as a sorted array of normalized lexemes)*

Full text searching in PostgreSQL is based on the match operator `@@`, which returns true if a **tsvector** (document) matches a **tsquery** (query). It doesn't matter which data type is written first:

Searching a Table

It is possible to do full text search with no index. A simple query to print the title of each row that contains the word friend in its body field is:

```
SELECT title
FROM pgweb
WHERE to_tsvector('english', body) @@ to_tsquery('english', 'friend');
```

This will also find related words such as friends and friendly, since all these are reduced to the same normalized lexeme.

The query above specifies that the english configuration is to be used to parse and normalize the strings. Alternatively we could omit the configuration parameters:

A more complex example is to select the ten most recent documents that contain create and table in the title or body:

```
SELECT title
FROM pgweb
WHERE to_tsvector(title || body) @@ to_tsquery('create & table')
ORDER BY last_mod_date DESC LIMIT 10;
```

Creating Indexes

We can create a GIN index to speed up text searches.

Example:

```
CREATE INDEX pgweb_idx ON pgweb USING gin(to_tsvector('english', body));
```

The two-argument version of `to_tsvector` was used in the index above, only a query reference that uses the 2-argument version of `to_tsvector` with the same configuration name will use that index. That is, `WHERE to_tsvector('english', body) @@ 'a & b'` can use the index, but `WHERE to_tsvector(body) @@ 'a & b'` cannot. This ensures that an index will be used only with the same configuration used to create the index entries.

Indexes can even concatenate columns:

Example:

```
CREATE INDEX pgweb_idx ON pgweb USING gin(to_tsvector('english', title || body));
```

Another approach is to create a separate `tsvector` column to hold the output of `to_tsvector`. This example is a concatenation of title and body, using `coalesce` to ensure that one field will still be indexed when the other is `NULL`.

Example:

```
ALTER TABLE pgweb ADD COLUMN textsearchable_index_col tsvector;
UPDATE pgweb SET textsearchable_index_col = to_tsvector('english', coalesce(title, '') ||
coalesce(body, ''));
```

Then we create a GIN index to speed up the search.

Example:

```
CREATE INDEX textsearch_idx ON pgweb USING gin(textsearchable_index_col);
Now we are ready to perform a fast full text search:
SELECT title
FROM pgweb
WHERE textsearchable_index_col @@ to_tsquery('create & table')
ORDER BY last_mod_date DESC LIMIT 10;
```

When using a separate column to store the `tsvector` representation, it is necessary to create a trigger to keep the `tsvector` column current anytime title or body changes.

19. Query Language (SQL) Functions

SQL functions execute an arbitrary list of SQL statements. Here's a simple example that removes rows with negative salaries from an emp table.

Example:

```
CREATE FUNCTION clean_emp() RETURNS void AS '
```

```
DELETE FROM emp
  WHERE salary < 0;
' LANGUAGE SQL;
```

```
SELECT clean_emp();
```

```
clean_emp
```

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

Note: It is usually most convenient to use dollar quoting for the string constant instead of the single quotes. It will be used in the following examples.

SQL Functions on Base Types

The simplest possible SQL function has no arguments and simply returns a base type, such as integer.

Example:

```
CREATE FUNCTION add_em(integer, integer) RETURNS integer AS $$
```

```
SELECT $1 + $2;
```

```
$$ LANGUAGE SQL;
```

```
SELECT add_em(1, 2) AS answer;
```

```
Answer
```

```
-----
3
```

SQL Functions on Composite Types

When writing functions with arguments of composite types, we must not only specify which argument we want (as we did above with \$1 and \$2) but also the desired attribute (field) of that argument.

Example: Suppose that emp is a table containing employee data, and therefore also the name of the composite type of each row of the table. Here is a function double_salary that computes what someone's salary would be if it were doubled:

```
CREATE TABLE emp (
```

```
  name    text,
```

```
  salary  numeric,
```

```
  age     integer,
```

```
  cubicle point
```

```
);
```

```
CREATE FUNCTION double_salary(emp) RETURNS numeric AS $$
SELECT $1.salary * 2 AS salary;
$$ LANGUAGE SQL;
```

```
SELECT name, double_salary(emp.*) AS dream
FROM emp
WHERE emp.cubicle ~= point '(2,1)';
name | dream
-----+-----
Bill | 8400
```

Notice the use of the syntax `$1.salary` to select one field of the argument row value. Also notice how the calling `SELECT` command uses `*` to select the entire current row of a table as a composite value.

We can drop a function using the **DROP FUNCTION** command.

```
DROP FUNCTION function_name;
```

Function Overloading

More than one function can be defined with the same SQL name, so long as the arguments they take are different. In other words, function names can be *overloaded*. When a query is executed, the server will determine which function to call from the data types and the number of the provided arguments.

Example:

```
CREATE FUNCTION test(int) RETURNS int
AS 'filename', 'test_1arg'
LANGUAGE C;
CREATE FUNCTION test(int, int) RETURNS int
AS 'filename', 'test_2arg'
LANGUAGE C;
```

20. PL/pgSQL- SQL procedural language

OVER VIEW

PL/pgSQL is a loadable procedural language for the PostgreSQL database system. The design goals of PL/pgSQL were to create a loadable procedural language that

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

Functions created with PL/pgSQL can be used anywhere that built-in functions could be used. SQL is the language PostgreSQL and most other relational databases use as query language. It's portable and easy to learn. But every SQL statement must be executed individually by the database server.

With PL/pgSQL you can group a block of computation and a series of queries *inside* the database server, thus having the power of a procedural language and the ease of use of SQL, but with considerable savings of client/server communication overhead.

- Extra round trips between client and server are eliminated
- Intermediate results that the client does not need do not have to be marshaled or transferred between server and client
- Multiple rounds of query parsing can be avoided

This can result in a considerable performance increase as compared to an application that does not use stored functions. Also, with PL/pgSQL you can use all the data types, operators and functions of SQL.

Functions written in PL/pgSQL can accept as arguments any scalar, array data type, composite type (row type) supported by the server, and they can return a result of any of these types.

Structure

PL/pgSQL is a block-structured language. The complete text of a function definition must be a *block*. A block is defined as:

```
[ <<label>> ]  
[ DECLARE  
  declarations ]  
BEGIN  
  Statements  
END [ label ];
```

Each declaration and each statement within a block is terminated by a semicolon. A block that appears within another block must have a semicolon after END, as shown above; however the final END that concludes a function body does not require a semicolon.

```
CREATE FUNCTION somefunc() RETURNS integer AS $$
<< outerblock >>
DECLARE
    quantity integer := 30;
BEGIN
    RAISE NOTICE 'Quantity here is %', quantity; -- Prints 30
    quantity := 50;
    ---- Create a subblock----
    DECLARE
        quantity integer := 80;
    BEGIN
        RAISE NOTICE 'Quantity here is %', quantity; -- Prints 80
        RAISE NOTICE 'Outer quantity here is %', outerblock.quantity; -- Prints 50
    END;

    RAISE NOTICE 'Quantity here is %', quantity; -- Prints 50
    RETURN quantity;
END;
$$ LANGUAGE plpgsql;
```

Note: PL/pgSQL's **BEGIN/END** are only for grouping; they do not start or end a transaction. Functions and trigger procedures are always executed within a transaction established by an outer query — they cannot start or commit that transaction, since there would be no context for them to execute in.

BASIC STATEMENTS

Assignment

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

variable := expression;

Examples:

```
tax := subtotal * 0.06;
my_record.user_id := 20;
```

Executing a Command With No Result

For any SQL command that does not return rows, for example INSERT without a RETURNING clause, you can execute the command within a PL/pgSQL function just by writing the command.

Executing a Query with a Single-Row Result

The result of a SQL command yielding a single row (possibly of multiple columns) can be assigned to a record variable, row-type variable, or list of scalar variables. This is done by writing the base SQL command and adding an INTO clause.

Example:

```
SELECT select_expressions INTO [STRICT] target FROM ...;  
INSERT ... RETURNING expressions INTO [STRICT] target;  
UPDATE ... RETURNING expressions INTO [STRICT] target;  
DELETE ... RETURNING expressions INTO [STRICT] target;
```

where **target** can be a record variable, a row variable, or a comma-separated list of simple variables and record/row fields. PL/pgSQL variables will be substituted into the rest of the query, and the plan is cached, just as described above for commands that do not return rows. This works for SELECT, INSERT/UPDATE/DELETE with RETURNING, and utility commands that return row-set results (such as EXPLAIN). Except for the INTO clause, the SQL command is the same as it would be written outside PL/pgSQL.

You can use an exception block if you wish to catch the error.

Example:

```
BEGIN  
SELECT * INTO STRICT myrec FROM emp WHERE empname = myname;  
EXCEPTION  
  WHEN NO_DATA_FOUND THEN  
    RAISE EXCEPTION 'employee % not found', myname;  
  WHEN TOO_MANY_ROWS THEN  
    RAISE EXCEPTION 'employee % not unique', myname;  
END;
```

Successful execution of a command with STRICT always sets FOUND to true.

Executing Dynamic Commands

Oftentimes you will want to generate dynamic commands inside your PL/pgSQL functions, that is, commands that will involve different tables or different data types each time they are executed, the **EXECUTE** statement is provided:

```
EXECUTE command-string [ INTO [STRICT] target ];
```

where *command-string* is an expression yielding a string (of type text) containing the command to be executed and *target* is a record variable, row variable, or a comma-separated list of simple variables and record/row fields.

The command string can be dynamically created within the function to perform actions on different tables and columns.

Obtaining the Result Status

There are several ways to determine the effect of a command. The first method is to use the GET DIAGNOSTICS command, which has the form:

```
GET DIAGNOSTICS variable = item [ , ... ];
```

Example:

```
GET DIAGNOSTICS integer_var = ROW_COUNT;
```

The second method to determine the effects of a command is to check the special variable named FOUND, which is of type boolean. FOUND starts out false within each PL/pgSQL function call. FOUND is a local variable within each PL/pgSQL function, any changes to it affect only the current function.

It is set by each of the following types of statements:

- A SELECT INTO statement sets FOUND true if a row is assigned, false if no row is returned.
- A PERFORM statement sets FOUND true if it produces (and discards) one or more rows, false if no row is produced.
- UPDATE, INSERT, and DELETE statements set FOUND true if at least one row is affected, false if no row is affected.
- A FETCH statement sets FOUND true if it returns a row, false if no row is returned.
- A MOVE statement sets FOUND true if it successfully repositions the cursor, false otherwise.
- A FOR statement sets FOUND true if it iterates one or more times, else false.

CONTROL STRUCTURES (IF ... THEN, LOOP, WHILE, FOR)

Control structures are probably the most useful (and important) part of PL/pgSQL. With PL/pgSQL's control structures, you can manipulate PostgreSQL data in a very flexible and powerful way.

Returning From a Function

There are two commands available that allow you to return data from a function: **RETURN** and **RETURN NEXT**

i) RETURN *expression*;

RETURN with an expression terminates the function and returns the value of *expression* to the caller.

If you declared the function with output parameters, write just RETURN with no expression. The current values of the output parameter variables will be returned.

ii) RETURN NEXT *expression*;

When a PL/pgSQL function is declared to return SETOF *sometype*, the individual items to return are specified by a sequence of **RETURN NEXT** or **RETURN QUERY** commands, and then a final RETURN command with no argument is used to indicate that the function has finished executing. It appends zero or more rows to the function's result set.

RETURN QUERY *query*;

RETURN QUERY appends the results of executing a query to the function's result set.

Example:

```
CREATE TABLE foo (fooid INT, foosubid INT, fooname TEXT);
INSERT INTO foo VALUES (1, 2, 'three');
INSERT INTO foo VALUES (4, 5, 'six');

CREATE OR REPLACE FUNCTION getAllFoo() RETURNS SETOF foo AS
$BODY$
DECLARE
    r foo%rowtype;
BEGIN
    FOR r IN SELECT * FROM foo
    WHERE fooid > 0
    LOOP
        -- can do some processing here
        RETURN NEXT r; -- return current row of SELECT
    END LOOP;
    RETURN;
END
$BODY$
LANGUAGE 'plpgsql';

SELECT * FROM getallfoo();
```

Note that functions using RETURN NEXT or RETURN QUERY must be called as a table source in a FROM clause.

Conditionals

IF statements execute commands based on certain conditions. PL/pgSQL has five forms of IF:

- IF ... THEN
- IF ... THEN ... ELSE
- IF ... THEN ... ELSE IF
- IF ... THEN ... ELSIF ... THEN ... ELSE
- IF ... THEN ... ELSEIF ... THEN ... ELSE

Example: IF ... THEN

```
IF v_user_id <> 0 THEN
    UPDATE users SET email = v_email WHERE user_id = v_user_id;
END IF;
```

Example: IF ... THEN ... ELSE

```
IF v_count > 0 THEN
    INSERT INTO users_count (count) VALUES (v_count);
    RETURN 't';
ELSE
    RETURN 'f';
END IF;
```

```

Example: IF ... THEN ... ELSE IF
IF demo_row.sex = 'm' THEN
    pretty_sex := 'man';
ELSE
    IF demo_row.sex = 'f' THEN
        pretty_sex := 'woman';
    END IF;
END IF;

```

```

Example: IF-THEN-ELSIF-ELSE
IF number = 0 THEN
    result := 'zero';
ELSIF number > 0 THEN
    result := 'positive';
ELSIF number < 0 THEN
    result := 'negative';
ELSE
    -- hmm, the only other possibility is that number is null
    result := 'NULL';
END IF;

```

Simple Loops

With the LOOP, EXIT, CONTINUE, WHILE, and FOR statements, you can arrange for your PL/pgSQL function to repeat a series of commands.

LOOP

```

[ <<label>> ]
LOOP
    Statements
END LOOP [ label ];

```

LOOP defines an unconditional loop that is repeated indefinitely until terminated by an **EXIT** or **RETURN** statement. The optional *label* can be used by EXIT and CONTINUE statements in nested loops to specify which loop the statement should be applied to.

Exit

```

EXIT [ label ] [ WHEN boolean-expression ];

```

EXIT can be used with all types of loops; it is not limited to use with unconditional loops. When used with a BEGIN block, EXIT passes control to the next statement after the end of the block.

CONTINUE

```

CONTINUE [ label ] [ WHEN boolean-expression ];

```

```

LOOP
    EXIT WHEN count > 100; -- some computations
    CONTINUE WHEN count < 50; -- some computations for count IN [50 .. 100]
END LOOP;

```

WHILE

```
[ <<label>> ]  
WHILE boolean-expression LOOP  
    Statements  
END LOOP [ label ];
```

FOR (integer variant)

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

Example:

```
FOR i IN 1..10 LOOP  
    -- i will take on the values 1,2,3,4,5,6,7,8,9,10 within the loop  
END LOOP;
```

```
FOR i IN REVERSE 10..1 LOOP  
    -- i will take on the values 10,9,8,7,6,5,4,3,2,1 within the loop  
END LOOP;
```

```
FOR i IN REVERSE 10..1 BY 2 LOOP  
    -- i will take on the values 10,8,6,4,2 within the loop  
END LOOP;
```

CURSORS

Rather than executing a whole query at once, it is possible to set up a **cursor** that encapsulates the query, and then read the query result a few rows at a time. One reason for doing this is to avoid memory overrun when the result contains a large number of rows. A more interesting usage is to return a reference to a cursor that a function has created, allowing the caller to read the rows. This provides an efficient way to return large row sets from functions.

Declaring Cursor Variables

All access to cursors in PL/pgSQL goes through **cursor variables**, which are always of the special data type **refcursor**. One way to create a cursor variable is just to declare it as a variable of type **refcursor**. Another way is to use the cursor declaration syntax, which in general is:

```
name [ [ NO ] SCROLL ] CURSOR [ ( arguments ) ] FOR query;
```

If **SCROLL** is specified, the cursor will be capable of scrolling backward; if **NO SCROLL** is specified, backward fetches will be rejected; if neither specification appears, it is query-dependent whether backward fetches will be allowed. *arguments*, if specified, is a comma-separated list of pairs *name datatype* that define names to be replaced by parameter values in the given query. The actual values to substitute for these names will be specified later, when the cursor is opened.

Examples:

```
DECLARE
```

```
  curs1 refcursor;
```

```
  curs2 CURSOR FOR SELECT * FROM tenk1;
```

```
  curs3 CURSOR (key integer) IS SELECT * FROM tenk1 WHERE unique1 = key;
```

Opening an unbound Cursor

```
OPEN unbound_cursor [ [ NO ] SCROLL ] FOR query;
```

OPEN FOR *query*

```
Example: OPEN curs1 FOR SELECT * FROM foo WHERE key = mykey;
```

OPEN FOR EXECUTE

```
Example: OPEN curs1 FOR EXECUTE 'SELECT * FROM ' || quote_ident($1);
```

Opening a Bound Cursor

```
OPEN bound_cursor [ ( argument_values ) ];
```

Examples:

```
OPEN curs2;
```

```
OPEN curs3(42);
```

Using Cursors

Once a cursor has been opened, it can be manipulated, these manipulations need not occur in the same function that opened the cursor to begin with.

FETCH

```
FETCH [ direction { FROM | IN } ] cursor INTO target;
```

FETCH retrieves the next row from the cursor into a target, which might be a row variable, a record variable, or a comma-separated list of simple variables, just like SELECT INTO. The *direction* clause can be any of the variants allowed in the SQL [FETCH](#) command except the ones that can fetch more than one row; namely, it can be NEXT, PRIOR, FIRST, LAST, ABSOLUTE *count*, RELATIVE *count*, FORWARD, or BACKWARD

Examples:

```
FETCH curs1 INTO rowvar;
```

```
FETCH curs2 INTO foo, bar, baz;
```

```
FETCH LAST FROM curs3 INTO x, y;
```

```
FETCH RELATIVE -2 FROM curs4 INTO x;
```

MOVE

```
MOVE [ direction { FROM | IN } ] cursor;
```

MOVE repositions a cursor without retrieving any data. MOVE works exactly like the FETCH command, except it only repositions the cursor and does not return the row moved to.

Examples:

```
MOVE curs1;  
MOVE LAST FROM curs3;  
MOVE RELATIVE -2 FROM curs4;
```

CLOSE

CLOSE cursor;

CLOSE closes the portal underlying an open cursor. This can be used to release resources earlier than end of transaction, or to free up the cursor variable to be opened again.

Returning Cursors

PL/pgSQL functions can return cursors to the caller. This is useful to return multiple rows or columns, especially with very large result sets. To do this, the function opens the cursor and returns the cursor name to the caller. The caller can then fetch rows from the cursor. The cursor can be closed by the caller, or it will be closed automatically when the transaction closes.

The following example shows one way to return multiple cursors from a single function:

```
CREATE FUNCTION myfunc(refcursor, refcursor) RETURNS SETOF refcursor AS $$  
BEGIN  
    OPEN $1 FOR SELECT * FROM table_1;  
    RETURN NEXT $1;  
    OPEN $2 FOR SELECT * FROM table_2;  
    RETURN NEXT $2;  
END;  
$$ LANGUAGE plpgsql;  
-- need to be in a transaction to use cursors.  
BEGIN;  
  
SELECT * FROM myfunc('a', 'b');  
FETCH ALL FROM a;  
FETCH ALL FROM b;  
COMMIT;
```

Errors and Messages

Use the RAISE statement to report messages and raise errors.

RAISE level 'format' [, expression [, ...]];

Possible levels are DEBUG, LOG, INFO, NOTICE, WARNING, and EXCEPTION. EXCEPTION raises an error (which normally aborts the current transaction); the other levels only generate messages of different priority levels.

In this example, the value of `v_job_id` will replace the `%` in the string:
`RAISE NOTICE 'Calling cs_create_job(%)', v_job_id;`
This example will abort the transaction with the given error message:
`RAISE EXCEPTION 'Nonexistent ID --> %', user_id;`

Triggers

Overview of Trigger Behavior

A trigger is a specification that the database should automatically execute a particular function whenever a certain type of operation is performed. Triggers can be defined to execute either before or after any **INSERT, UPDATE, or DELETE** operation, either once per modified row, or once per SQL statement. If a trigger event occurs, the trigger's function is called at the appropriate time to handle the event.

The trigger function must be defined before the trigger itself can be created. The trigger function must be declared as a function taking no arguments and returning type `trigger`.

If more than one trigger is defined for the same event on the same relation, the triggers will be fired in alphabetical order by trigger name.

TRIGGER PROCEDURE

PL/pgSQL can be used to define trigger procedures. A trigger procedure is created with the `CREATE FUNCTION` command, declaring it as a function with no arguments and a return type of `trigger`. A trigger function must return either `NULL` or a record/row value having exactly the structure of the table the trigger was fired for.

Example: A PL/pgSQL Trigger Procedure

This example trigger ensures that any time a row is inserted or updated in the table, the current user name and time are stamped into the row. And it checks that an employee's name is given and that the salary is a positive value.

```
CREATE TABLE emp (  
    empname text,  
    salary integer,  
    last_date timestamp,  
    last_user text  
);  
  
CREATE FUNCTION emp_stamp() RETURNS trigger AS $emp_stamp$  
BEGIN  
    -- Check that empname and salary are given  
    IF NEW.empname IS NULL THEN  
        RAISE EXCEPTION 'empname cannot be null';  
    END IF;  
    IF NEW.salary IS NULL THEN  
        RAISE EXCEPTION '% cannot have null salary', NEW.empname;  
    END IF;  
  
    -- Who works for us when she must pay for it?
```

```
IF NEW.salary < 0 THEN
    RAISE EXCEPTION '% cannot have a negative salary', NEW.empname;
END IF;

-- Remember who changed the payroll when
NEW.last_date := current_timestamp;
NEW.last_user := current_user;
RETURN NEW;
END;
$emp_stamp$ LANGUAGE plpgsql;

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