citusdata

Deliver **Real-Time Big Data** to Your End Users Using **PostgreSQL**

CitusDB is a hybrid transactional and analytics database built on the latest version of PostgreSQL which:

- Delivers real-time big data insights to your end users
- Reduces the cost and complexity of your database infrastructure
- Leverages your existing PostgreSQL knowledge and applications
- Provides the flexibility to use either structured or unstructured data
- Elastically scales as your database grows
- Is always built on the latest version of PostgreSQL

We also offer two free, open source extensions for PostgreSQL:

- **pg_shard** allows you to scale out PostgreSQL across a cluster of commodity servers to handle big data operational workloads
- **cstore_fdw** can help you reduce PostgreSQL data storage costs significantly by applying columnar storage techniques to compress your database an average of 4-6x

Download our solutions now

www.citusdata.com

Get More Refcardz! Visit Refcardz.com

CONTENTS

Essential PostgreSQL

By Leo Hsu and Regina Obe

ABOUT POSTGRESQL

PostgreSQL is an open-source object-relational database with many enterprise-level features. It runs on numerous platforms: Linux, Unix, Windows, and Mac OS X. It is simple and quick to install, fast, and it sports advanced features such as: streaming replication, spatial support via PostGIS, windowing functions, table partitioning, and full-text search. In addition to its enterprise features, it has the added benefit of supporting numerous languages for authoring stored functions. It has an extensible procedural language architecture to introduce new languages. It also has an extensible type and index architecture for introducing new data types, operators, and indexes for these custom types, and support for querying external data sources such as CSV, web services, and other PostgreSQL services via its Foreign Data Wrapper (SQL/MED) support.

Targeted at novices and professionals alike, this Refcard will help you quickly navigate some of PostgreSQL's most popular features as well as its hidden gems. It will cover topics such as configuration, administration, backup, language support, and advanced SQL features. There will be a special focus on new features in PostgreSQL 9.3 and 9.4.

CONFIGURATION

PostgreSQL uses three main configuration files to control overall operations. You can find these files in the initialized data cluster (the folder specified during the initialization process using initdb -d).

HOT TIP

All these can be edited with a text editor. They can be edited via PgAmin III if you install the adminpack extension in master postgres db.
To do so: `CREATE EXTENSION ADMINPACK;`

FILE	PURPOSE
postgresql.conf	Controls the listening port, IP, and default query planner settings, memory settings, path settings, and logging settings. Can be queried via pg_settings database view.
pg_hba.conf	Controls the authentication models used by PostgreSQL and can be set per user, per database, per IP range, or a combination of all.
pg_indent.conf	Controls mapping of an OS user to a PostgreSQL user.

POSTGRESQL.CONF

The following settings are all located in the postgresql.conf file. Remember that these are default settings; many of these you can choose to override for each session, for each database, or for each user/role.

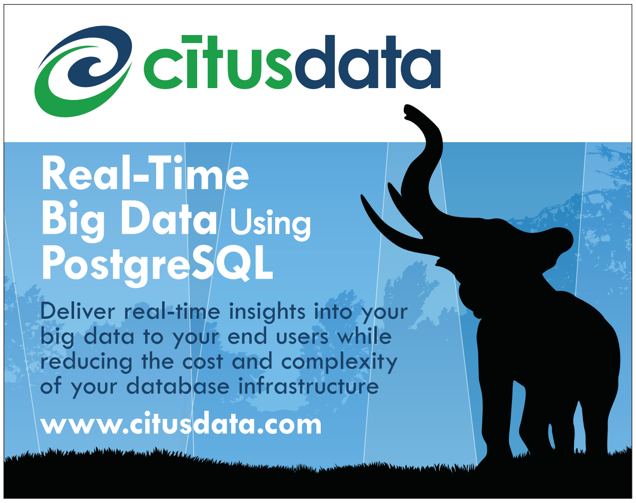
OPTION	DESCRIPTION
listen_ addresses	Use '*' to listen on all IPs of the server, 'localhost' to listen on just local, or a comma separated list of IPs to listen on. Requires service restart if changed and can only be set globally.
port	Defaults to 5432, but can be changed to allow multiple postgresql daemon clusters/ versions to coexist using same IP but different ports.
search_path	List of default schemas that don't need schema qualification. First schema is where non-schema qualified objects are created.
constraint_ exclusion	Options: on, off, or partial. Partial was introduced in 8.4 and is the new default. Allows planner to skip over tables if constraint ensures query conditions cannot be satisfied by the table. Mostly used for table partitioning via table inheritance.
shared_buffers	Controls how much memory is allocated to PostgreSQL and shared across all processes. Requires service restart and can only be set globally.

In PostgreSQL 9.4, a new SQL construction `ALTER SYSTEM` was introduced that allows you to set these settings at the system level without editing the postgresql.conf. For many, you still need to do a service restart and for others at least a:

```
SELECT pg_reload_conf();
```

PG_HBA.CONF

PostgreSQL supports many authentication schemes to control access to the database. The pg_hba.conf file dictates which schemes are used based on the rules found in this file. You can mix and match various authentication schemes at the



same time. The rules are applied sequentially such that the first match fitting a connection is the one that is used. This is important to remember because if you have a more restrictive rule above a less restrictive, then the more restrictive is the one that trumps.

The most commonly used authentication schemes are trust (which allows connections without a password) and md5 (which authenticates with md5 encrypted passwords). Others include: reject, crypt, password (this is plain text), krb5, ident (authenticate simply by identity of user in OS), pam, and ldap.

The example pg_hba.conf entries below allow all local connections to connect to all databases without a password and all remote connections to authenticate via md5.

#TYPE	DATABASE	USER	CIDR-ADDRESS	METHOD
HOST	ALL	ALL	127.0.0.1/32	TRUST
HOST	ALL	ALL	0.0.0.0/0	MD5

DATA TYPES

PostgreSQL has numerous built-in types. In addition, you can define custom types. Furthermore, all tables are considered to be types in their own right, and can therefore be used within another table's column. Below are the common built-in types:

DATE/TIME TYPES

TYPE	DESCRIPTION
date	The date is a datatype to represent dates with no time. Default representation is ISO 8601 e.g. 'YYYY-MM-DD'. Use datestyle configuration setting to control defaults.
timestamp	This includes both date and time and is timezone-neutral. *Example:* '2009-07-01 23:00'
timestamp with time zone	*Example:* '2009-07-01 23:00:00-04'
time	Time without date. *Example:* '23:14:20'
time with time zone	*Example:* '23:14:20-04'
interval	A unit of time used to add and subtract from a timestamp. *Example:* `SELECT TIMESTAMP 2009-07-01 23:14:20' + INTERVAL '4 months 2 days 10 hours 9 seconds'`
daterange, tsrange, tstzrange	New in PostgreSQL 9.2; defines a specific time range. Example is a date > 2012-07-01 and <= 2013-08-31 `SELECT '(2012-07-01, 2013-08-31]'::daterange;`
Constituents of datetime, use date_part function to extract	Century, day, decade, dow (starts Sunday), doy, epoch, hour, isodow (day of week starts on Monday), minute, month, quarter, week, year.

NUMERIC TYPES

TYPE	DESCRIPTION
int, int8	4 byte and 8 byte integers.
serial, serial4, serial8	Sequential integers; this can be used during table creation to specify auto-numbered fields.
numeric(s, p)	Decimal numbers; s is scale and p is precision.
double precision	Floating point numbers.
numrange, int4range	Introduced in 9.2 for defining number ranges. An integer >= 1 and < 10. `SELECT ‘[1,10)’::int4range;`
percentile_cont, percentile_disc	Continuous and discrete percentile aggregate. Can take a numeric value (between 0 and 1) for percentile rank or can take an array of numeric values between 0 and 1.

STRING TYPES

TYPE	DESCRIPTION
varchar(n) (a.k.a. character varying)	Max of n characters, no trailing spaces.
char(n)	Padded to n characters.
text	Unlimited text.

OTHER TYPES

TYPE	DESCRIPTION

array	Arrays in PostgreSQL are typed and you can create an array of any type. To define a column as an array of a specific type, follow with brackets. Example: `varchar(30)[]`. You can also autogenerate arrays in SQL statements with constructs such as:
	`SELECT ARRAY['john','jane'];`
	`SELECT ARRAY(SELECT emp_name FROM employees);`
	`SELECT array_agg(e.emp_name) FROM employees;`

enum	Enumerators: `CREATE TYPE cloth_colors AS ENUM ('red','blue','green');` When used in a table, you define the column as the name of the enum. Sorting is always in the order the items appear in the enum definition.
boolean	True/false.
bytea	Byte array used for storing binary objects, such as files.
lo	Large object. Stored in a separate system table with object ID reference to the large object. Useful for importing files from file system and storing and exporting back to file system.
JSON	JavaScript Object Notation (JSON) was introduced in PostgreSQL 9.2 and includes built-in validation. JSON stored as plain text. No direct index support. PostgreSQL 9.3 enhanced JSON functionality by providing more functions and operators that work with JSON. PostgreSQL 9.4 enhanced further by providing even more functions and operators.

TYPE	DESCRIPTION
jsonb	Binary form of JSON—introduced in PostgreSQL 9.4. Can be indexed using GIN indexes and supports intersects and containment operators in addition to all the functions and operators JSON supports. Performance is much faster than the JSON type. No duplicate keys per object are allowed; sort of keys per object are not maintained.

COMMON GLOBAL VARIABLES

TYPE	DESCRIPTION
CURRENT_TIMESTAMP, now()	Returns current date and time with timezone.
CURRENT_DATE	Returns current date without the time.
CURRENT_TIME	Returns current time without the date.

COMMONLY USED FUNCTIONS

DATE/TIME FUNCTIONS AND OPERATORS

TYPE	DESCRIPTION
age(timestamp1, timestamp2)	Returns an interval spanned between timestamp1 and timestamp2.
age(timestamp)	Difference from current time.

date_part(text, timestamp), date_part(text, interval)	`date_part('day', timestamp '2009-07-04 11:05:45') => 4`
	`date_part('hour', interval '560 minutes') => 9`

date_trunc(text, timestamp | timestamptz | date)	`date_trunc('hour', '2014-01-15 10:30 PM'::timestamp) => 2014-01-15 22:00:00`
operators +, -, / (for intervals only)	You can add (or subtract) intervals to datetimes. You can perform addition and subtraction between two datetimes. You can divide intervals into smaller intervals.
generate_series(timestamp, timestamp, [interval]) [8.4]	Generate rows of timestamps.

STRING FUNCTIONS AND OPERATORS

TYPE	DESCRIPTION
|| (string || string, string || number)	String concatenation.
left, right, substring	Returns left x elements, right x elements, or substring from position x for y number of elements.
length	Number of characters in string.

lpad, rpad	Left and right pad.
	`lpad('A', 5, 'X') => 'XXXXA'`
	`rpad('A', 5, 'X') => 'AXXXX'`

lower, upper, initcap	Lower, upper, proper case.
md5	Calculates the MD5 hash.

quote_ident	Quotes keywords and expressions not suitable for identity when unquoted.
	`quote_ident('in') => "in"`
	`quote_ident('big') => big`

quote_literal	Escapes both single and double quotes.
quote_nullable	Similar to `quote_literal` but doesn't quote NULL.
replace	`replace('1234abcv', '1234', 'joe') => joeabcv`
split_part	Takes a delimited string and returns the nth item. `split_part('abc|def', '|', 2) =>def`
string_agg	SQL aggregate function that aggregates a set of values into a string.
strpos(text, subtext)	Returns numeric position of subtext within text.
trim, btrim, ltrim, rtrim	Trim spaces in string.

ARRAY FUNCTIONS AND OPERATORS

TYPE	DESCRIPTION
||	Array concatenation. `ARRAY[1,2,3] || ARRAY[3,4,5] => {1,2,3,3,4,5}`
unnest	Converts an array to rows. `SELECT anum FROM unnest(ARRAY[1,2,3])`
array_agg	SQL aggregate function that aggregates a set of values into an array.

array_upper(anyarray, dimension)
array_lower(anyarray, dimension)
	Returns upper/lower bound of the requested array dimension.
	`SELECT array_upper(ARRAY[ARRAY['a'], ARRAY['b']], 1);`
	`outputs: 2`

array_to_string(anyarray, delimiter_text)	Converts an array to a text delimited by the delimiter. `array_to_string(ARRAY[12,34], '|') => '12|34'`

RANGE FUNCTIONS AND OPERATORS

TYPE	DESCRIPTION

lower(anyrange), upper(anyrange)	Lower bound and upper bound value of a range:
	`SELECT lower(a), upper(a)`
	`FROM (SELECT '[1,10]'::int4range AS a) AS f;`
	`outputs:`
	`lower | upper`
	`-------+-------`
	`    1 |     11`

@>	Contains range or element.
	`SELECT a @> 1 AS ce,`
	`  a @> '[2,3]'::int4range AS cr`
	`FROM (SELECT '[1,10]'::int4range AS a) AS f;`

&&	Have elements in common.

TYPE	DESCRIPTION

+	Union of 2 ranges.
	`SELECT '[2014-7-20,`
	`2014-10-20]'::daterange + '[2014-6-20,`
	`2014-7-22]'::daterange;`
	Output:
	`[2014-06-20,2014-10-21)`

*	Intersection.
	`SELECT '[2014-7-20,`
	`2014-10-20]'::daterange * '[2014-6-20,`
	`2014-7-22]'::daterange;`
	Output:
	`[2014-07-20,2014-07-23)`

-	Difference.
	`SELECT '[2014-7-20,`
	`2014-10-20]'::daterange - '[2014-6-20,`
	`2014-7-22]'::daterange;`
	Output:
	`[2014-07-20,2014-10-21)`

JSON/JSONB FUNCTIONS AND OPERATORS

TYPE	DESCRIPTION

->>	Extract an element of JSON/jsonb as text.
	`SELECT prod->>'price' As price`
	`FROM (`
	`SELECT '{"id": 1, "name": "milk",`
	`  "price": 2.50}'::json as prod) As f;`
	`outputs:2.50`

->	Extract an element of JSON/jsonb as JSON/jsonb (useful for doing more operations on a complex subelement).

#>>	Extract a nested element of JSON/jsonb as text.
	`SELECT prod#>>'{nutrition,vitamin`
	`d}'::text[] AS vd`
	`FROM (`
	`SELECT '{"id": 1,"name": "milk",`
	`  "price": 2.50, "nutrition": {"vitamin`
	`d": "30%"}}'::json AS prod) AS f;`
	Outputs: 30%

#>	Extract a nested element of JSON/jsonb as JSON/jsonb. Useful for doing more operations such as working with arrays within json.

WINDOW FUNCTIONS

TYPE	DESCRIPTION
row_number	Number of current row from its current partition.
rank, percent_rank, dense_rank	Ranking based on order in current partition (dense_rank is without gaps; percent_rank is relative rank).
lead, lag	Nth value relative to current, -nth value relative to current (n defaults to 1) in current partition.
first_value, last_value, nth_value	Absolute first/last/nth value in a partition based on order regardless of current position.

OTHER FUNCTIONS

TYPE	DESCRIPTION

generate_series(int1, int2, [step])
generate_series(timestamp1, timestamp2, [interval])
	Returns rows consisting of numbers from int1 to int2 with [step] as gaps. Step is optional and defaults to 1.

min, max, sum, avg, count	Common aggregates.
percentile_dist, percentile_cont [9.4]	Useful for computing medians.

DATABASE OBJECTS

Here is a listing of what you will find in a PostgreSQL server or database. An * means the object lives at the server level, not the database level.

OBJECT	DESCRIPTION
Databases*	PostgreSQL supports more than one database per service/daemon.
Tablespaces*	Logical representation of physical locations where tables are stored. You can store different tables in different tablespaces, and control data storage based on database and user/group role.
Languages	These are the procedural languages installed in the database.
Casts	PostgreSQL has the unique feature of having an extensible cast system. It has built-in casts, but allows you to define your own and override default casts. Casts allow you to define explicit behavior when casting from one object to another, and allow you to define autocast behavior.
Schemas	These are logical groupings of objects. One can think of them as mini-databases within a larger database. An object always resides in a schema.
Tables, Views	Views are virtual tables that encapsulate an SQL `SELECT` statement. In PostgreSQL, tables can inherit from other tables and data can be altered against views. PostgreSQL 9.1+ introduced Foreign Tables, which are references to data from a Foreign source via a foreign data wrapper (FDW). PostgreSQL 9.3 introduced materialized views, which are views that contain the cached data. These need to be refreshed to update the view cache.
Rules	Rules are similar to triggers, except they can only be written in SQL, and they rewrite a statement rather than actually updating directly. Views are actually implemented as `SELECT` rules (and can have `DO INSTEAD` inserts/update rules to make them updateable).
Functions, Triggers, and Aggregates	These can be written in any enabled language in the database, live in schemas. PostgreSQL allows you to define your own custom aggregate functions. Triggers are special classes of functions that have OLD and NEW variables available that hold a pointer to the OLD and NEW data. Triggers are bound to table. New in

OBJECT	DESCRIPTION
Functions, Triggers, and Aggregates *(cont.)*	PostgreSQL 9.3 are event triggers which are bound to events such as creation of table or deletion of table.
Operators, Operator Classes, Operator Families	Live in schemas. Many are predefined, but more can be added and allow you to define things such as +, =, etc. for custom data types.
Sequences	Autocreated when defining columns as serial. In PostgreSQL, sequences are objects in their own right and can be shared across many tables.
Types	Live in schemas. Don't forget that you have the flexibility to create your own custom data types in PostgreSQL.
Foreign Data Wrappers, Servers and User Mappings	Foreign Data Wrappers are remote data source drivers that allow you to access data in a non-PostgreSQL or remote PostgreSQL table. PostgreSQL 9.1 introduced these. 9.2 improved on general performance, and 9.3 introduced a new FDW called postgresfdw for connecting to other PostgreSQL servers, and also enhanced the API to support Foreign table updates.
Extensions [9.1+]	Packaging of functions, tables, and other objects for easy deployment in a database. These are installed using `CREATE EXTENSION`. `CREATE EXTENSION hstore;`

TOOLS

PostgreSQL comes bundled with several tools useful for administration and query writing.

TOOL	DESCRIPTION
psql	Command-line client packaged with PostgreSQL. Good for automating SQL jobs, copying data, outputing simple HTML reports.
createdb, dropdb	For creating and dropping a database from the OS shell.
pgAdminIII	Popular graphical user interface packaged with PostgreSQL.
pg_restore	Command-line tool for restoring compressed or .tar backups.
pg_dump	Command-line tool for doing backups. Great for automated backups.
pg_dumpall	Command-line tool for dumping all databases into a single backup.
pgAgent	A daemon/service that can be downloaded from http://www.pgadmin.org/download/pgagent.php. Used for scheduling SQL jobs and batch shell jobs. Jobs can be added easily and monitored using the PgAdmin III job interface.
pg_basebackup	Used for doing filesystem hot backup of db data cluster.
pg_upgrade	Used for updating in place from one major version of PostgreSQL to another.

PSQL COMMON TASKS

PSQL is a command-line tool that allows you to run ad-hoc queries, scripts, and other useful database management routines.

PSQL runs in both a non-interactive mode (straight from the OS shell prompt) and an interactive mode (PSQL terminal prompt). In both modes, the following arguments apply:

ARGUMENT	DESCRIPTION
-d	Database. Defaults to the user (via system identification if no user is specified).
-h	Server host. Defaults to localhost if not specified.
-p	Port. Defaults to 5432 if not specified.
-U	Username you are trying to log in with. Defaults to system user name.

PSQL NON-INTERACTIVE MODE

Getting help

```
$ psql -help
```

Execute an SQL script stored in a file

```
$ psql -h localhost -U postgres -p 5432 -f /path/to/
pgdumpall.sql
```

Output data in html format

```
$ psql -h someserver -p 5432 -U postgres -d dzone -H -c
"SELECT * FROM pg_tips" -o tips.html
```

Execute a single statement against a db

```
$ psql -U postgres -p 5432 -d dzone -c "CREATE TABLE
test(some_id serial PRIMARY KEY, some_text text);"
```

Execute an SQL batch script against a database and send output to file

```
$ psql -h localhost -U someuser -d dzone -f /path/to/
scriptfile.sql -o /path/to/outputfile.txt
```

PSQL INTERACTIVE MODE

To initiate interactive PSQL, type:

```
psql -U username -p 5432 -h localhost -d dzone
```

Once you are in the PSQL terminal there are a myriad of tasks you can perform. Below are some of the common ones.

COMMAND	TASK
\q	Quit
:q	Cancel out of more screen
\?	Help on psql commands
\h some_command	Help on SQL commands
\connect postgres	Switch database
\l	List all databases
\dtv p*	List tables and views that start with p.
\du	List user/group roles and their group memberships and server level permissions.
\d sometable	List columns, data types, and constraints for a table.
\i somefile	Execute SQL script stored in a file.
\o somefile	Output contents to file.
Use up and down arrows	Retrieve prior commands.

COMMAND	TASK
\timing	Toggle query timing on and off; when on, query output includes timing information.
\copy	Copy from client computer to server and from server to client computer. Example: The following command string copies data to local client computer in CSV format with header. `\copy (SELECT * FROM sometable) TO 'C:/sometable.csv' WITH HEADER CSV FORCE QUOTE`
\copy ... from program	Allows you to copy output from an external program such as ls, dir, wget, curl. New in 9.3.

ADMIN TASKS

BACKUP AND RESTORE

Below are common backup and restore statements.

Create a compressed backup

```
pg_dump -h someserver -p 5432 -U someuser -F -c -b -v -f
"/somepath/somedb.backup" somedb
```

Create a compressed backup of select tables

```
pg_dump -h localhost -p 5432 -U someuser -F -c -b -f
"C:/somedb.backup" -t "someschema.table1" -t "someschema.
table2" -v somedb
```

Create a compressed backup excluding a particular schema

```
pg_dump -h localhost -p 5432 -U someuser -F -c -b -f
"C:/somedb.backup" -N someschema -v somedb
```

Restore a compressed backup

```
pg_restore -h localhost -d db_to_restore_to -U someuser
/path/to/somedb.backup
```

Restore select schemas from backup

```
pg_restore -h localhost -d db_to_restore_to -U someuser
-n someschema1 -n someschema2 /path/to/somedb.backup
```

Output a table of contents from backup file

```
pg_restore -l -f "C:/toc.txt" "C:/somedb.backup"
```

Restore only items in the table of contents

```
pg_restore -h localhost -d db_to_restore -U someuser -L
"C:/toc.txt" "C:/somedb.backup"
```

OTHER

Change globally work mem (9.4+)

Requires reload and some require restart.

```
ALTER SYSTEM SET work_mem TO '20MB';
SELECT pg_reload_conf();
```

> **HOT TIP**
>
> pg_dumpall currently only dumps to plain text sql. pg_dumpall backups must be restored with psql. For space savings and flexibility, use pg_dump. With pg_dump compressed and tar backups, you can selectively restore objects. You cannot selectively restore with plain text backups.

Below are common switches used with pg_dump [D], pg_restore [R], pg_dumpall [A]. These tools are packaged with PostgreSQL and are in the bin folder. They are also packaged with pgAdmin III and are in the PgAdmin III/version/ folder.

SWITCH	TOOL	DESCRIPTION
-b, --blobs	D	Include large objects in dump.
-d, --dbname=NAME	R	Specify name of database to restore to.
-F, --format=c|t|p|d	D R	Specify backup file format (c = compressed, t = tar, p = plain text, d = directory). Plain-text backups must be restored with psql. Directory new in [9.2].
-c, --clean	D R A	Clean (drop) schema prior to create (for pg_dumpall drop database prior to create).
-g, --globals-only	A	Dump only global objects (roles, schemas, tablespaces), no databases.
-j, --jobs=NUM [8.4],[9.2]	D R	Use this multiple parallel jobs to restore. This is especially useful for large backups and speeds them up significantly in many cases. 8.4 introduced parallel restore (pg_restore). 9.2 introduced (in pg_dump) parallel backup (needs to have format directory based).
-l, --list	R	Print summarized TOC of the archive.
-L, --use-list=filename	R	Use TOC from this file for selcting/ordering output.
-n, --schema=NAME	D R	Dump/restore only select objects in schema(s).
-N, --exclude-schema=SCHEMA	D R	Exclude from dump/restore named schema.
-r, --roles-only	A	Dump only roles, no database or tablespace.
-t, --table=NAME	D	Backup only named table(s) along with associated indexes, constraints, and rules.
-T, --exclude-table=NAME	D	Exclude named table(s) from backup.
-v --verbose	D R A	Controls verbosity.
--exclude-table-data=TABLE [9.2]	D	Exclude dumping table data for specific table.
-s -section=pre-data|post-data|data [9.2]	D R	Dump or restore select parts. Pre-data just backs up or restores structures; post-data restores primary keys, foreign keys, and constraints. Data just restores data.
--if-exists [9.4]	D	Use IF EXISTS when dropping.

USER RIGHTS MANAGEMENT

These are SQL commands you can use to control rights. They can be run in the PSQL interactive, loading an SQL file, or via PgAdmin.

Create a new role with login rights that can create objects	`CREATE ROLE somerole LOGIN NOSUPERUSER INHERIT CREATEDB NOCREATEROLE;`

Create a group role with no login rights and members inherit rights of role	CREATE ROLE somerole NOSUPERUSER INHERIT NOCREATEDB NOCREATEROLE;
Add a role to another role	GRANT somerole TO someotherrole;

Give rights to a role	Example uses:
	GRANT SELECT, UPDATE ON TABLE sometable TO somerole;
	GRANT ALL ON TABLE sometable TO somerole;
	GRANT EXECUTE ON FUNCTION somefunction TO somerole;
	-- Grant execute to all users
	GRANT EXECUTE ON FUNCTION somefunction TO public;

Revoke rights	REVOKE ALL ON TABLE sometable FROM somerole;
Give insert/update rights to select columns	GRANT INSERT, UPDATE (somecolumn) ON sometable TO somerole;
Grant rights to all future tables in a schema	ALTER DEFAULT PRIVILEGES IN SCHEMA someschema GRANT ALL ON TABLES TO somerole;
Grant rights to all existing tables in a schema	GRANT ALL ON ALL TABLES IN SCHEMA someschema TO somerole;

DATA DEFINITION (DDL)

Many of the examples we have below use named schemas. If you leave out the schema, objects created will be in the first schema defined in the search_path and dropped by searching the search path sequentially for the named object.

Create a new database	CREATE DATABASE postgresql_dzone;
Install extension in a database	CREATE EXTENSION hstore;
Create a schema	CREATE SCHEMA someschema;
Changing database schema search path	Sets the default schema to someschema. ALTER DATABASE postgresql_dzone SET search_path = someschema, public;

Dropping objects with no dependencies	A drop without a CASCADE clause will not drop an object if there are objects that depend on it, such as views, functions, and tables.
	For drop database you should be connected to a database other than the one you're dropping.
	DROP DATABASE postgresql_dzone;
	DROP VIEW someview;
	ALTER TABLE sometable DROP COLUMN somecolumn;
	DROP FUNCTION somefunction;

Dropping object and all dependencies. *(Use with caution.)*	DROP SCHEMA someschema CASCADE;

Create a table	CREATE TABLE test_scores(student varchar(100),
	score integer, test_date date DEFAULT CURRENT_DATE,
	CONSTRAINT pk_test_scores PRIMARY KEY (student, test_date));

Create a child table	CREATE TABLE somechildtable (CONSTRAINT pk_somepk PRIMARY KEY (id)) INHERITS (someparenttable);
Create a check constraint	ALTER TABLE sometable ADD CONSTRAINT somecheckcontraint CHECK (id > 0);
Create or alter a view	CREATE OR REPLACE VIEW someview AS SELECT * FROM sometable; [Prior to version 8.4, adding new columns to a view requires dropping and recreating].
Create a materialized view	CREATE MATERIALIZED VIEW someview AS SELECT * FROM sometable;
Refresh materialized view	REFRESH MATERIALIZED VIEW someview;
Refresh materialized view without read blocking [9.4]	REFRESH MATERIALIZED VIEW CONCURRENTLY someview;

Create a view (doesn't allow insert if data would not be visible in view) [9.4]	CREATE OR REPLACE VIEW someview AS SELECT * FROM sometable
	WHERE active = true WITH CHECK OPTION;

Add a column to a table	ALTER TABLE sometable ADD COLUMN somecolumn timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP;
Add a functional index to a table	CREATE INDEX idx_someindex ON sometable USING btree (upper(somecolumn));
Create a new type	CREATE TYPE sometype AS (somecolumn integer, someothercolumn integer[]);

Create a trigger	CREATE OR REPLACE FUNCTION sometrigger()
	RETURNS trigger AS
	$$
	BEGIN
	IF OLD.somecolumn <> NEW.somecolumn OR
	(OLD.somecolumn IS NULL AND
	NEW.somecolumn IS NOT NULL) THEN
	NEW.sometimestamp := CURRENT_TIMESTAMP;
	END IF;
	RETURN NEW;
	END;
	$$
	LANGUAGE 'plpgsql' VOLATILE;

Add trigger to table	CREATE TRIGGER sometrigger BEFORE UPDATE ON sometable FOR EACH ROW
	EXECUTE PROCEDURE sometriggerupdate();

Suppress redundant updates	A built-in trigger that prevents updates that would not change any data.
	CREATE TRIGGER trig_01_suppress_redundant BEFORE UPDATE ON sometable FOR EACH ROW
	EXECUTE PROCEDURE suppress_redundant_updates_trigger();

HOT TIP

A table can have multiple triggers, and each trigger for a particular event on a table is run in alphabetical order of the named trigger. So if order is important, name your triggers such that they are sorted in the order you need them to run.

QUERY AND UPDATE (DML)

These are examples that show case some of PostgreSQL popular or unique query features.

ADDING AND UPDATING DATA

Insert statement with multirows	INSERT INTO test_scores(student,score,test_date)
	VALUES ('robe', 95, '2014-01-15'),
	('lhsu', 99, '2014-01-15'),
	('robe', 98, '2014-07-15'),
	('lhsu', 92, '2014-07-15'),
	('lhsu', 97,'2014-08-15');

Insert statement from SELECT, only load items not already in table	INSERT INTO tableA(id,price)
	SELECT invnew.id,invnew.price
	FROM tableB AS invnew LEFT JOIN tableA
	AS invold ON (invnew.id = invold.id)
	WHERE invold.price IS NULL;

Cross update, only update items for a particular store where price has changed	UPDATE tableA
	 SET price = invnew.price
	FROM tableB AS invnew
	WHERE invnew.id = tableA.id
	AND NOT (invnew.price = tableA.price);

Insert from a tab-delimited file no header	COPY products FROM "/tmp/productslist.txt" WITH DELIMITER '\t' NULL AS 'NULL';
Insert from a comma-delimited file with header row	--these copy from the server's file system COPY products FROM "/tmp/productslist.csv" WITH CSV HEADER NULL AS 'NULL';
Copy data to comma-delimited file and include header	--this outputs to the server's file system COPY (SELECT * FROM products WHERE product_rating = 'A') TO '/tmp/productsalist.csv' WITH CSV HEADER NULL AS 'NULL';

RETRIEVING DATA

View running queries	SELECT * FROM pg_stat_activity;

Select the first record of each distinct set of data	--this example selects the store and product
	--where the given store has the lowest price
	--for the product. This uses PostgreSQL
	--DISTINCT ON and an order by to resort
	--results by product_name.
	SELECT r.product_id, r.product_name, r.product_price
	FROM (SELECT DISTINCT ON(p.product_id) p.product_id, p.product_name, s.store_name, i.product_price
	FROM products AS p INNER JOIN inventory AS i
	     ON p.product_id = i.product_id
	    INNER JOIN store AS s ON i.store_id = s.store_id
	ORDER BY p.product_id, i.product_price) AS r;

Get last date's score for each student. Returns only one record per student	SELECT DISTINCT ON(student) student, score, test_date
	FROM test_scores
	ORDER BY student, test_date DESC;

Use window function to number records and get running average	SELECT row_number() OVER(wt) AS rn, student, test_date,
	 (AVG(score) OVER(wt))::numeric(8,2) AS avg_run
	FROM test_scores
	WINDOW wt AS (PARTITION BY student ORDER BY test_date);
	rn | student | test_date | avg_run
	----+---------+------------+---------
	  1 | lhsu | 2014-01-15 | 99.00
	  2 | lhsu | 2014-07-15 | 95.50
	  3 | lhsu | 2014-08-15 | 96.00
	  1 | robe | 2014-01-15 | 95.00
	  2 | robe | 2014-07-15 | 96.50

Get median values [9.4]	SELECT student, percentile_cont(0.5) WITHIN GROUP (ORDER BY score) AS m_continuous, percentile_disc(0.5) WITHIN GROUP (ORDER BY score) AS m_discrete FROM test_scores GROUP BY student; student | m_continuous | m_discrete --------+--------------+----------- lhsu | 97 | 97 robe | 96.5 | 95

Filtered aggregates [9.4] use instead of CASE WHEN (or subselect) (especially useful for aggregates like array_agg which may return nulls with CASE WHEN)	SELECT date_trunc('quarter',test_date)::date AS qtr_start,
	 array_agg(score) FILTER (WHERE student = 'lhsu') AS lhsu,
	 array_agg(score) FILTER (WHERE student = 'robe') AS robe
	FROM test_scores
	GROUP BY date_trunc('quarter',test_date);
	qtr_start | lhsu | robe
	------------+---------+------
	 2014-01-01 | {99} | {95}
	 2014-07-01 | {92,97} | {98}

Ordered aggregates, list scores in order of test date, one row for each student. Cast to make a string.	SELECT student, string_agg(score::text, ',' ORDER BY test_date DESC) AS scores FROM test_scores GROUP BY student; student | scores ---------+---------- lhsu | 97,92,99 robe | 98,95

Non-Recursive CTE with 2 CTE expressions. Note a CTE expression has only one WITH, each subexpression is separated by a , and the final query follows.
Example returns the lowest priced car in each fuel_grade, limiting to just Japan, USA, German
	WITH c AS
	(SELECT country_code, conv_us
	FROM country
	WHERE country IN('Japan', 'USA','Germany')
	),
	prices AS
	(SELECT p.car, p.fuel_grade, price*c.conv_us AS us_price
	FROM cars AS p
	        INNER JOIN c
	        ON p.country_code = c.country_code
	WHERE p.category = 'Cars'
	)
	SELECT DISTINCT ON(fuel_grade)
	 prices.car, us_price
	FROM prices
	ORDER BY fuel_grade, us_price;

Recursive CTE * inventory, gives full name which includes parent tree name e.g. Paper->Color->Red->20 lbs	WITH RECURSIVE tree AS
	(SELECT id, item, parentid,
	  CAST(item AS text) AS fullname
	FROM products
	WHERE parentid IS NULL
	UNION ALL
	SELECT p.id,p.item, p.parentid,
	  CAST(t.fullname || '->'
	   || p.item AS text) AS fullname
	FROM products AS p
	  INNER JOIN tree AS t
	  ON (p.parentid = t.id)
	)
	SELECT id, fullname
	FROM tree
	ORDER BY fullname;

PROCEDURAL LANGUAGES

PostgreSQL stands out from other databases with its extensive and extendable support for different languages to write database-stored functions. It allows you to call out to libraries native to that language. We will list the key language as well as some esoteric ones. The languages with an * are preinstalled with PostgreSQL and can be enabled. Some require further installs in addition to the language handler.

You can create set returning functions, simple scalar functions, triggers, and aggregate functions with most of these languages. This allows for languages that are highly optimized for a particular task to work directly with data without having to always copy it out to process as you normally would need with a simple database storage device. Language handlers can be of two flavors: trusted and untrusted. An untrusted language can access the filesystem directly.

From PostgreSQL 9.1+, languages not enabled by default in database or not built-in are installed using :

```
CREATE EXTENSION …;
CREATE EXTENSION 'plpythonu';
CREATE OR REPLACE somename(arg1 arg1type)
  RETURNS result_argtype AS
$$
    body goes here
$$
LANGUAGE 'somelang';
```

LANGUAGE	DESCRIPTION	REQ

**sql*
(trusted)**
	Enabled in all databases. Allows you to write simple functions and set returning functions in just SQL. The function internals are visible to the planner, so in many cases it performs better than other functions since the planner can strategize how to navigate based on the bigger query. It is simple and fast, but limited in functionality.
	`CREATE OR REPLACE FUNCTION prod_state(prev numeric, e1 numeric, e2 numeric).`
	`    RETURNS numeric AS`
	`$$`
	`    SELECT COALESCE($1,0) + COALESCE($2*$3,0);`
	`$$`
	`LANGUAGE 'sql' IMMUTABLE;`
		none

c*	Built in and always enabled. Often used to extend PostgreSQL (e.g. postgis, pgsphere, tablefunc) or, for example, to introduce new windowing functions (introduced in PostgreSQL 8.4). Functions are referenced from a .so or .dll file.
	`CREATE OR REPLACE FUNCTION st_summary(geometry)`
	`  RETURNS text AS`
	`'$libdir/postgis-2.1', 'LWGEOM_summary'`
	`  LANGUAGE 'c' IMMUTABLE STRICT;`
		none

**plpgsql*
(trusted)**
	Not always enabled, but packaged so it can be installed.
	`CREATE FUNCTION cp_upd(p_key integer, p_value varchar)`
	`RETURNS void AS`
	`$$`
	`BEGIN`
	`IF EXISTS(SELECT test_id FROM testtable WHERE test_id = p_key) THEN`
	`   UPDATE testtable`
	`     SET test_stuff = p_value`
	`   WHERE test_id = p_key;`
	`ELSE`
	`  INSERT INTO testtable (test_id,`
	`   test_stuff)`
	`  VALUES(p_key, p_value);`
	`END IF;`
	`     RETURN;`
	`END;`
	`$$`
	`LANGUAGE 'plpgsql' VOLATILE;`
		none

plv8 (trusted)	Good for manipulating JSON objects, reusing existing Javascript libraries, numeric processing. Comes packaged with 3 language bindings: Plv8 (aka PL/Javascript), plls (LiveScript), plcoffee (CoffeeScript).
	To install:
	`CREATE EXTENSION plv8;`
	`CREATE EXTENSION plls;`
	`CREATE EXTENSION plcoffee;`
		Google v8 engine

EXAMPLE FUNCTIONS

This next table demonstrates some examples of writing functions in various languages. For all functions you write, you can use the `CREATE` or `REPLACE FUNCTION` construction to overwrite existing functions that take same arguments. We use `CREATE FUNCTION` here.

LANGUAGE	EXAMPLE

**plperl (trusted),
plperlu (untrusted)**
	`CREATE FUNCTION use_quote(TEXT)`
	`RETURNS text AS $$`
	`    my $text_to_quote = shift;`
	`    my $qfunc = $_SHARED{myquote};`
	`    return &$qfunc($text_to_quote);`
	`$$ LANGUAGE plperl;`

**plpythonu,
plpython2u,
plpython3u (untrusted)**
	`CREATE FUNCTION fnfileexists(IN fname text) RETURNS boolean AS`
	`$$`
	` import os`
	` return os.path.exists(fname)`
	`$$`
	`LANGUAGE 'plpythonu' STRICT;`

plr	Good for doing advanced stats and plotting using the R statistical language.
	`CREATE FUNCTION  r_quantile(float8[])`
	`RETURNS float8[] AS`
	`$$`
	`quantile(arg1, probs = seq(0, 1, 0.25), names = FALSE)`
	`$$ LANGUAGE 'plr' IMMUTABLE STRICT;`

plv8	Allows you to write functions in JavaScript.
	`CREATE FUNCTION`
	` fib(n int) RETURNS int AS $$`
	`  function fib(n) {`
	`    return n<2 ? n : fib(n-1) + fib(n-2)`
	`  }`
	`  return fib(n)`
	`$$ LANGUAGE plv8 IMMUTABLE STRICT;`

COMMON PROCEDURAL TASKS

Create a table trigger and use in table

```
CREATE OR REPLACE FUNCTION mytable_ft_trigger()
  RETURNS trigger AS $$
BEGIN
         NEW.tsv :=
         setweight(to_tsvector('pg_catalog.english',
         coalesce(new.field1,'')), 'A') ||
         setweight(to_tsvector('pg_catalog.english',
         coalesce(NEW.field2,'')), 'B');
         return NEW;
END
$$ LANGUAGE plpgsql;
CREATE TRIGGER mytable_trigiu
 BEFORE INSERT OR UPDATE OF field1,field2
ON mytable
  FOR EACH ROW EXECUTE PROCEDURE mytable_ft_trigger()
```

Return sets and use out of params

```
CREATE OR REPLACE FUNCTION
  fn_sqltestmulti(param_subject varchar,
    OUT test_id integer,
    OUT test_stuff text)
    RETURNS SETOF record
   AS
$$
    SELECT test_id, test_stuff
       FROM testtable
         WHERE test_stuff LIKE $1;
$$
  LANGUAGE 'sql' STABLE;
--example
SELECT * FROM fn_sqltestmulti('%stuff%');
```

Return sets and use of table construct

```
CREATE OR REPLACE FUNCTION
  fn_sqltestmulti(param_subject varchar)
    RETURNS TABLE(test_id integer, test_stuff text)
   AS
$$
    SELECT test_id, test_stuff
       FROM testtable
         WHERE test_stuff LIKE $1;
$$
  LANGUAGE 'sql' STABLE;
```

EXTENSIONS

Extensions extend the capabilities of PostgreSQL by providing additional data types, functions, index types, and more. After installing an extension, you need to run the following command to enable it:

```
CREATE EXTENSION extension_name;
```

NOTABLE EXTENSIONS

EXTENSION	DESCRIPTION	LINK
PostGIS	Adds support for geographic objects allowing location queries to be run using SQL.	http://postgis.net/
pg_shard	Shards and replicates tables for horizontal scaling and high availability.	https://github.com/citusdata/pg_shard
pg_stat_statements	Tracks execution statistics of all SQL statements.	http://www.postgresql.org/docs/current/static/pgstatstatements.html
cstore_fdw	Columnar store for PostgreSQL.	https://github.com/citusdata/cstore_fdw
postgresql-hll	Distinct value counting with tunable precision.	https://github.com/aggregateknowledge/postgresql-hll
pgcrypto	Cryptographic functions.	http://www.postgresql.org/docs/current/static/pgcrypto.html
dblink	Connections to other PostgreSQL databases from a database session.	http://www.postgresql.org/docs/current/static/dblink.html

HOT TIP

For a full list of extensions shipped with PostgreSQL see: http://www.postgresql.org/docs/current/static/contrib.html

To search for third party extensions see: http://pgxn.org/

ABOUT THE AUTHORS

The wife and husband team of **Leo Hsu** and **Regina Obe** founded Paragon Corporation in 1997, which specializes in database technology and works with numerous organizations to design, develop, and maintain database and web applications. They have become active participants in the on-going development of PostGIS, a spatial extension of PostgreSQL. Regina is a member of the PostGIS core development team and Project Steering Committee. They maintain two sites: http://www.postgresonline.com -- provides tips and tricks for using PostgreSQL and http://www.bostongis.com - provides tips and tricks for using PostGIS and other open source and open GIS tools.

RECOMMENDED BOOK

"Thinking of migrating to PostgreSQL? This clear, fast-paced introduction helps you understand and use this open source database system. Not only will you learn about the enterprise class features in versions 9.2, 9.3, and 9.4, you'll also discover that PostgeSQL is more than a database system—it's also an impressive application platform."

BUY NOW

CREDITS:

Editor: **G. Ryan Spain** | Designer: **Yassee Mohebbi** | Production: **Chris Smith** | Sponsor Relations: **Brandon Rosser** | Marketing: **Chelsea Bosworth**

DZone

DZone communities deliver over 6 million pages each month to more than 3.3 million software developers, architects and decision makers. DZone offers something for everyone, including news, tutorials, cheat sheets, research guides, feature articles, source code and more.

"DZone is a developer's dream," says PC Magazine.

DZONE, INC.
150 PRESTON EXECUTIVE DR.
CARY, NC 27513

888.678.0399
919.678.0300

REFCARDZ FEEDBACK WELCOME
refcardz@dzone.com

SPONSORSHIP OPPORTUNITIES
sales@dzone.com

ISBN-13: 978-1-936502-77-6
ISBN-10: 1-936502-77-1
50795
9 781936 502776

VERSION 1.0 **$7.95**