PostgreSQL

PostgreSQL

Module 9
Administration

Adding An External Module To Postgresql

- PostgreSQL is designed to be highly extensible
- Database extensions can be loaded simply and easily
  - CREATE EXTENSION myext;
  - ALTER EXTENSION myext UPDATE;
- There are many kinds of additional module offerings, such as the following:
  - Additional functions
  - Additional data types
  - Additional operators
  - Additional indexes.
- The procedure that makes a module usable is actually a two-step process.
  - First, you install the module's files on your system so that they become available to the database server
  - Next, you connect to the database (or databases) where you want to use the module, and create the required objects.

There's more…

- There are several ways to make additional modules available for your database server, as follows:
  - Installing from a manually downloaded package
  - Installing from source code
- The procedure that makes a module usable is actually a two-step process. First, you install the module's files on your system so that they become available to the database server. Next, you connect to the database (or databases) where you want to use the module, and create the required objects
- PostgreSQL functions and objects can reference code in these libraries, allowing extensions to be bound tightly to the running server process

Using An Installed Module

- How to enable an installed module so that it can be used in a particular database.
- Extension support is a great step towards an automated package management system for PostgreSQL.
- Using the extension infrastructure
  - **CREATE EXTENSION myextname;**
  - This will automatically create all the required objects inside the current database.
  - For security reasons, you need to do so as a database superuser. For instance, if you want to install the dblink extension, type this:
  - **CREATE EXTENSION dblink;**

Managing Installed Extensions

- we list all available extensions:
  - **postgres-# select * FROM pg_available_extensions**
  - **postgres-# ORDER BY name;**

```
name              | dblink
default_version   | 1.0
installed_version | 1.0
comment           | connect to other PostgreSQL databases from within a database
```

  - we can list all the objects in the dblink extension, as follow:
    - **postgres=# \dx+ dblink**
    - **Postgres=# \dx+ db***
- The pg_available_extensions system view shows one row for each extension control file in the SHAREDIR/extension directory .The pg_extension catalog table records only the extensions that have actually been created.

Listing Extensions In The Database

- A series of patches implements extensions support for dump and restore for PostgreSQL.
- In PostgreSQL 9.1 and later versions, there is a catalog table recording the list of installed extensions, so this recipe is quite simple. Issue the following command:
  - **postgres=> SELECT * FROM pg_extension;**

```
-[ RECORD 1 ]--+---------
extname        | plpgsql
extowner       | 10
extnamespace   | 11
extrelocatable | f
extversion     | 1.0
extconfig      |
extcondition   |
```

  - To get the same list with fewer technical details, you can use the meta-command **\dx**, like when listing databases
  - The pg_available_extensions system view shows one row for each extension control file in the SHAREDIR/extension directory .The pg_extension catalog table records only the extensions that have actually been created

Extensions- hstore

- Implements hstore data type for storing key-value pairs in a single value
  - Postgres=# create extension hstore;
  - Postgres=# create table pref(email text primary key, timezone text, properties hstore);
  - postgres=# insert into pref values ('abc@xyz.com','Asia/Pasific','skills => "PostgreSQL, Android, Python"');
  - postgres =# select * from pref;
  - Postgres=# select properties -> ‘skills’ from pref;
  - Postgres=# select properties -> ‘skills’ from pref where properties-> ‘skills’=‘PostgreSQL’;

Encrypting Sensitive Data

- Encrypt data using the pgcrypto contrib package.
- Make sure that pgcrypto is installed on your database host.
- For one way encryption, the crypt function packaged in pgcrypto provides an added level of security above the md5 way.
  - **CREATE TABLE testusers(username varchar(100) PRIMARY KEY, cryptpwd text, md5pwd text);**
  - **INSERT INTO testusers(username, cryptpwd, md5pwd) VALUES ('robby', crypt('test', gen_salt('md5')), md5('test')), ('artoo', crypt('test',gen_salt('md5')), md5('test'));**

There's more…

- **SELECT username, cryptpwd, md5pwd FROM testusers;**
- Observe that both users have chosen the same password **test**. The md5 version is the same for both, but the crypted password is different although they are the same password.

Extensions- chkpass

- Postgres=# create extension chkpass;
- Postgres=# create table t11(id int, username varchar, password chkpass);
- postgres=# insert into t11 values (1,'q1','testpass');
- postgres=# insert into t11 values (1,'q2','password');
- postgres=# insert into t11 values (2,'q3','test');
- Postgres=# select * from t11;
- Postgres=# select * From t11 where password = 'password';

Extensions- citext

- Postgres=# create extension citext;
- Postgres=# create table citext_test (id int, text varchar);
- Postgres=# insert into citext_test values (1,'Simmi');
- Postgres=# create table citext_test1 (id int, text citext);
- Postgres=# insert into citext_test values (1,'Simmi');
- Postgres=# Select * from citext_test where text = 'simmi';
- Postgres=# Select * from citext_test1 where text = 'simmi';

Extensions- Dblink

- Postgres=# **create database demo;**
- Postgres=# **\c demo;**
- demo=# **create extension dblink;**
- demo =# **SELECT dblink_connect(‘dbname=postgres user=postgres password=root');**
- demo =# **CREATE TABLE tblA (id serial, detail text);**
- Demo=# **\c postgres**
- Postgres=# **create table tblB (id serial, detail text);**
- postgres=#**INSERT INTO tblB(detail) VALUES (‘abc’), (‘pqr’);**
- postgres =# **\c demo**
- demo=# **SELECT * FROM dblink('dbname=postgres user=postgres password=root', 'select * from tblB')** AS t1(id integer, detail text);
- demo=# **INSERT INTO tblA SELECT id, detail**

  FROM dblink('dbname=demo', 'SELECT id, detail FROM tblB) AS t(id integer, detail text);

Accessing Objects In Other Postgresql Databases

- Sometimes, you may want to access data in other PostgreSQL databases.
- The **PostgreSQL Foreign Data Wrapper** is a specific contrib extension that uses the Foreign Data Wrapper infrastructure to connect to remote PostgreSQL servers.
- FDW provides a means to access data outside a PostgreSQL cluster. The data may be on a filesystem, on a MySQL server, or another PostgreSQL cluster. We will take a look at two examples: one accessing a file and another accessing a PostgreSQL. The steps usually involve:
- 1. Installing the extension
- 2. Creating a server object. This provides details of the server (such as IP and port or path)
- 3. Creating user mapping. This involves providing the authentication information to connect to other databases. This step may or may not be necessary depending on the type of server object.
- 4. Creating the foreign table. This creates a table that maps to another table or file on the server object.

FDW for files

1. Install the extension as follows:

➨ CREATE EXTENSION file_fdw;

2. Create the server object with the following command:

➨ CREATE SERVER file_server FOREIGN DATA WRAPPER file_fdw;

3. Create the foreign table. The syntax is very similar to the standard CREATE TABLE syntax. We have the keyword FOREIGN and need to mention the server object it points to:

➨ CREATE FOREIGN TABLE file_to_tbl(id integer, name varchar(40)) SERVER file_server OPTIONS (delimiter ',', filename '/home/postgres/names.csv', format 'csv');

Postgres_FDW

- Let's create a new user for this:
  - CREATE USER myuser PASSWORD 'myuser‘ CREATEDB;
- Connect to demo (an existing database) as postgres and create the table that will be accessed using FDW:
  - CREATE TABLE prod_table (id serial, name varchar(50));
  - INSERT INTO prod_table (name) VALUES ('Scott'), ('Thomson');
  - GRANT SELECT on prod_table to myuser;
- Log in as myuser to PostgreSQL and create the database where the FDW will be created with the following command:
  - CREATE DATABASE report;
- Log in as postgres to the new database (report) and install the extension:
  - CREATE EXTENSION postgres_fdw ;

Postgres_FDW

- Create the server object:
  - CREATE SERVER f_prod FOREIGN DATA WRAPPER postgres_fdw OPTIONS (dbname ‘demo', host ‘localhost’, port '5432');
  - ALTER SERVER f_prod OWNER TO myuser;
- Then, create the user mapping:
  - CREATE USER MAPPING FOR myuser SERVER f_prod OPTIONS (user 'myuser', password 'myuser');
- Next, log in as myuser1 to report and create the foreign table:
  - CREATE FOREIGN TABLE prod_table_fdw (id integer, name varchar(100)) SERVER f_prod options(table_name ‘prod_table’);
  - SELECT * FROM prod_table_fdw;

Writing A Script

- Database administration often involves applying a coordinated set of changes to the database
- One of PostgreSQL's great strengths is the transaction system, wherein almost all actions can be executed inside a transaction
- Transactions definitely apply to **Data Definition Language** (**DDL**), which refers to the set of SQL commands used to define, modify, and delete database objects
  - Script into a transaction, like the following:
    - **BEGIN;**
    - **command 1;**
    - **command 2;**
    - **command 3;**
    - **COMMIT;**

Writing A Script

- To run single transaction
  - bash $ psql -1 -f myscript.sql
  - bash $ psql --single-transaction -f myscript.sql
- Keyword DROP have an option IF EXISTS
  - Example: DROP VIEW IF EXISTS cust_view;
- Keyword CREATE, mostly have the optional suffix OR REPLACE
  - Example: CREATE OR REPLACE VIEW cust_view AS SELECT * FROM cust;
- To change the output definition of a function or a view, then using OR REPLACE is not sufficient. In that case, you must use DROP and recreate

Writing A Psql Script

- The default mode for the psql script tool is to continue processing when it finds an error. This sounds dumb, but it exists for historical compatibility only.
  - **$ $EDITOR test.sql**

    mistake1;

    mistake2;

    mistake3;
  - Alternatively, we can edit the test.sql file with the initial line shown here:
    - **$ $EDITOR test.sql**

    \set ON_ERROR_STOP

    mistake1;

    mistake2;

    mistake3;

Performing Actions On Many Tables

```
create schema test;
create table test.a (col1 INTEGER);
create table test.b (col1 INTEGER);
create table test.c (col1 INTEGER);
```

Script that lists the tables we want to perform tasks against, something like following:

```
SELECT relname FROM pg_class c JOIN pg_namespace n ON
c.relnamespace = n.oid WHERE n.nspname = 'test';
```

We then use the preceding SQL to generate the text for an SQL script, substituting the schema name and tablename into the SQL text. We then output to a script file named multi.sql as follows:

```
postgres=# \t on
postgres=# \o multi.sql
```

Performing Actions On Many Tables

```
SELECT 'ALTER TABLE '|| n.nspname || '.' || c.relname || ' ADD COLUMN
last_update_timestamp TIMESTAMP WITH TIME ZONE;' FROM pg_class c JOIN
pg_namespace n ON c.relnamespace = n.oid WHERE n.nspname = 'test';
```

Once we've generated the script, we can just check if it all looks correct

```
postgres=# \! cat multi.sql
```

and then run the script and watch the results (success!).

```
postgres=# \i multi.sql
```

Changing The Data Type Of A Column

- Changing column data types is not an everyday task, but when we need to do it, we must know all the details so that we can perform the conversion on a production system without any errors.
- Command to change the data type of a column :
  - **postgres=# CREATE TABLE birthday (name TEXT, dob INTEGER);**
  - **postgres=# ALTER TABLE birthday**
  - **postgres-# ALTER COLUMN dob SET DATA TYPE text;**
- Explicit conversion with the USING clause, as follows:
  - **postgres=# ALTER TABLE birthday ALTER COLUMN dob SET DATA TYPE integer USING dob::integer;**
- The USING clause can also be used to handle complex expressions involving other columns. This could be used for data transformations, which might be useful for DBAs in some circumstances, such as migrating to a new database design on a production database server.

Changing The Data Type Of A Column

- Now let's try moving to a DATE type:
  - postgres=# ALTER TABLE birthday ALTER COLUMN dob SET DATA TYPE date USING date(to_date(dob::text, 'YYMMDD') - (case when dob/10000 < 15 then interval '0' else interval '100 years' end);
- With PostgreSQL you can also set or drop default expressions, whether or not the NOT NULL constraints are applied.
  - ALTER TABLE foo ALTER COLUMN col SET NOT NULL;
  - ALTER TABLE foo ALTER COLUMN col DROP NOT NULL;

Adding/removing schemas

- create a new schema as same name as an existing user
  - Create user scarlett;
  - CREATE SCHEMA AUTHORIZATION scarlett;
- The DROP SCHEMA command won't work unless the schema is empty
  - So,
    - **DROP SCHEMA IF EXISTS newschema CASCADE;**
- Move one table from its current schema to a new schema
  - ALTER TABLE cust SET SCHEMA anotherschema;
  - all the indexes, triggers, and rules defined on those tables will also be moved to the new schema

Moving objects between tablespaces

- Drop tablespace, when it is empty
- Also, tablespaces can contain both permanent and temporary objects
- Temporary objects are created according to the setting of the **temp_tablespaces** parameter
- Identify the tablespace of each user object
  - SELECT spcname,relname,CASE WHEN relpersistence = 't' THEN 'temp ' ELSE '' END ||

```
  CASE
     WHEN relkind = 'r' THEN  'table'
     WHEN relkind = 'v' THEN  'view'
     WHEN relkind = 'S' THEN  'sequence'
     WHEN relkind = 'c' THEN  'type'
  ELSE 'index' END as objtype
FROM pg_class c join pg_tablespace ts ON (CASE WHEN c.reltablespace = 0 THEN
(SELECT dattablespace FROM pg_database WHERE datname = current_database())
   ELSE c.reltablespace END) = ts.oid
WHERE relname NOT LIKE 'pg_toast%' AND
relnamespace NOT IN (SELECT oid FROM pg_namespace
WHERE nspname IN ('pg_catalog', 'information_schema'));
```

Moving objects between tablespaces

- ALTER TABLE mytable SET TABLESPACE new_tablespace;
- Indexes can exist in separate tablespaces
  - ALTER INDEX mytable_val_idx SET TABLESPACE new_tablespace;
- If you want to move a table and its indexes all
  - BEGIN;
  - ALTER TABLE mytable SET TABLESPACE new_tablespace;
  - ALTER INDEX mytable_val1_idx SET TABLESPACE new_tablespace;
  - ALTER INDEX mytable_val2_idx SET TABLESPACE new_tablespace;
  - COMMIT;
- To ensure that objects are created in the right place next time you create them
  - SET default_tablespace = 'new_tablespace';
  - ALTER DATABASE mydb SET default_tablespace = 'new_tablespace';
- Do not run the following command by mistake
  - ALTER DATABASE mydb SET TABLESPACE new_tablespace;

Transactions

- A collection of data manipulation execution steps that are treated as a single unit of work
  - Use to group multiple statements
  - Use when multiple clients are accessing data from the same table concurrently
- All or none of the steps succeed
  - Execute if all steps are good
  - Cancel if steps have error or are incomplete
- ACID compliant

Transaction SQL Control Statements

- **START TRANSACTION** (or **BEGIN**): Explicitly begins a new transaction
- **SAVEPOINT**: Assigns a location in the process of a transaction for future reference
- **COMMIT**: Makes the changes from the current transaction permanent
- **ROLLBACK**: Cancels the changes from the current transaction
- **ROLLBACK TO SAVEPOINT**: Cancels the changes executed after the savepoint
- **SET AUTOCOMMIT**: Disables or enables the default autocommit mode for the current connection

SQL Control Statements Flow: Example

AUTOCOMMIT Mode

- Determines how and when new transactions are started
- AUTOCOMMIT mode is enabled by default:
  - Implicitly commits each statement as a transaction
  - \set AUTOCOMMIT off
  - When AUTOCOMMIT is disabled, transactions span multiple statements by default.
  - You can end a transaction with COMMIT or ROLLBACK. Solutions

Exception Handling

- *To err is coder!*
- Implementing RAISE statements takes care of error handling in an efficient manner with the options of NOTICE, DEBUG, and EXCEPTION
- It sends information to be displayed and logs it in PostgreSQL logs, i.e, in /var/lib/pgsql/9.6/data/pg_log
- Syntax:

```
RAISE NOTICE|DEBUG|EXCEPTION "your message string";
```

- Example:

```
IF ( a is not an int ) then
Return a;
Else
Raise NOTICE 'a is integer, enter string';
```

ACID

- Atomic
  - All statements execute successfully or are canceled as a unit
- Consistent
  - A database that is in a consistent state when a transaction begins is left in a consistent state by the transaction
- Isolated
  - One transaction does not affect another
- Durable
  - All changes made by transactions that complete successfully are recorded properly in the database
  - Changes are not lost

ACID

Period	T1	T2
P1	Starts summing	
P2	Adds balance of ID 1 to SUM	Starts
P3	Continues summing	Debits account ID 1
P4	…	Credits account ID 3
P5	…	Commits
P6	Adds balance of IDs…,all,…rest to SUM	
P7	Returns SUM	

Transaction Isolation Problems

Three common problems:

- “Dirty” read
  - When a transaction reads the changes made by another uncommitted transaction
- Non-repeatable read
  - When changes from another committed transaction cause a prior read operation to be non-repeatable
- Phantom read (or phantom row)
  - A row that appears but was not previously visible within the same transaction

Isolation Levels

Four isolation levels:

- READ UNCOMMITTED
  - Allows a transaction to see uncommitted changes made by other transactions
- READ COMMITTED
  - Allows a transaction to see committed changes made by other transactions
- REPEATABLE READ
  - Ensures consistent SELECT output for each transaction
- SERIALIZABLE
  - Completely isolates the effects of a transaction from others
- Default level for PostgreSQL
  - READ COMMITTED

PostgreSQL Isolation Levels

Isolation level	Dirty reads	Non-repeatable reads	Phantoms
Read uncommitted	Allowed, but not in PG	Possible	Possible
Read committed	Not possible	Possible	Possible
Repeatable read	Not possible	Not possible	Allowed, but not in PG
Serializable	Not possible	Not possible	Not possible

Read committed

- This is the default isolation level
- In this level, a query sees a snapshot of the data with all changes that were committed before the statement began executing
- Postgres=# show default_transaction_isolation;
- Postgres=# select txid_current();
- Postgres=# select txid_current();
- Postgres=# select txid_current();

Concurrent Updates

- Transaction A does:
  - UPDATE foo SET id = 4 WHERE id = 1

  and before it commits, transaction B comes along and wants to do the same thing on the same row
- B clearly must wait to see if A commits or not
  - UPDATE foo SET id = 2 WHERE id = 1
- If A aborts then B can go ahead, using the pre-existing value of x. But if A commits, what then?
  - Using the old value of x will yield a clearly unacceptable result: x ends up by putting id value to 4 not 2 after both transactions commit
  - But if B is allowed to increment the new value of x, then B is reading data committed since it began execution. This violates the basic principle of transaction isolation

Repeatable read

- In this level, a query sees data as it existed at the beginning of the transaction
- If we execute the same SELECT multiple times within a transaction, the results will be consistent
- Postgres=# begin;
- Postgres=# SHOW transaction_isolation;
- Postgres=# SET transaction isolation level repeatable read;
- Postgres=# SHOW transaction_isolation;
- Postgres=# commit;
- Postgres=# SHOW transaction_isolation;
- Postgres=# INSERT INTO account (first_name,last_name, account_bal) values ('Jane', 'Adam', 1000);

Serializable level

- This is the strictest level available
- To guarantee true serializability, PostgreSQL uses predicate locking, which means that it keeps locks that allow it to determine when a write would have had an impact on the result of a previous read from a concurrent transaction, had it run first. In PostgreSQL, these locks do not cause any blocking and therefore cannot play any part in causing a deadlock

- Postgres=# SELECT first_name, account_bal FROM account
- postgres=# SET default_transaction_isolation = 'serializable';

PostgreSQL and MVCC

- PostgreSQL uses MVCC to provide different views of the database to different sessions, based on the isolation level setting
- The key rule is that readers should not block writers and writers should not block readers
- PostgreSQL uses the concept of transaction ID to achieve MVCC
  - Transaction ID being retrieved using the txid_current() function

PostgreSQL

P_0

T_1 Commit

T_2 Rollback

T_3 Commit

T_7 Commit

T_8 Commit

T_9 Rollback

T_{10} Commit

P_1

T_{11} Commit

T_{12} Commit

T_{13} Rollback

Multi-version Concurrency Control

- A PostgreSQL application sees the following behavior of concurrent transactions:
  - Each transaction sees a snapshot (database version) as of its start time no matter what other transactions are doing while it runs
  - Readers do not block writers, writers do not block readers.
  - Writers only block each other when updating the same row

PostgreSQL and MVCC

- PostgreSQL uses a few system columns to manage transactions
- The two most relevant columns to generate snapshots of data are:
  - xmin: This column holds the identity of the transaction, which created the specific version of the row.
  - xmax: This column holds the identity of the transaction, which deleted the specific row
  - Example:
    - Postgres=# BEGIN;
    - Postgres=# CREATE TABLE employees(id integer, first_name varchar, last_name varchar);
    - Postgres=# INSERT INTO emp(first_name, last_name) VALUES ('SCOTT','TIGER');
    - Postgres=# SELECT xmin,xmax, * FROM emp; Commit;

PostgreSQL and MVCC

- Example:
  - Postgres=# BEGIN;
  - Postgres=# SELECT txid_current();
  - Postgres=# UPDATE employees SET last_name = 'TGR';
  - Postgres=# SELECT xmin,xmax, * FROM emp;
  - In session 2
  - Postgres=# SELECT xmin,xmax, * FROM emp;
  - Go to session 1
  - Postgres=# COMMIT;

PostgreSQL and MVCC

- Example of the transaction ID 10
  - Postgres=# DELETE FROM employee;
  - Postgres=# BEGIN;
  - Postgres=# SELECT txid_current();
  - Postgres=# SELECT xmin,xmax, * FROM emp;
  - In second session, insert a record
    - Postgres=# INSERT INTO emp(first_name, last_name) VALUES ('SCOTT','TIGER');
  - In first session:
    - SELECT xmin, xmax, * FROM emp;