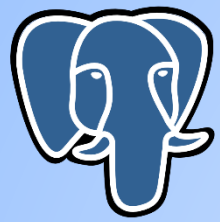
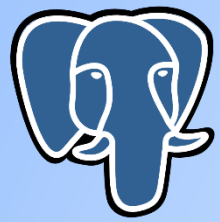


PostgreSQL

Module 7

Tables and Data

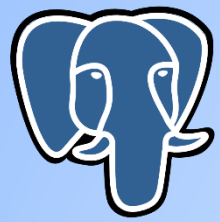


PostgreSQL System Catalog Views

- Catalogs are named collections of schemas
- When you create a table, PostgreSQL stores the definition of that table in the system catalog
- The system catalog is a collection of PostgreSQL tables
- You can issue SELECT statements against the system catalog tables just like any other table
- A cluster is an implementation-defined collection of catalogs
- **Cluster > Catalog > Schema > Table > Columns & Rows**
- So in Postgres we have this containment hierarchy:
  - A computer may have one cluster or multiple
  - A database server is a cluster
  - A cluster has catalogs (Catalog = Database)
  - Catalogs have schemas. (Schema = namespace of tables, and security boundary)
  - Schemas have tables
  - Tables have rows
  - Rows have values, defined by columns

Querying Metadata of PostgreSQL

- **Metadata**
  - Metadata is the information about the structure of your database including the definitions of entire database schema
  - It also includes definition of each and every object like – tables, views, indices, users, functions – within that schema
  - PostgreSQL database metadata from INFORMATION_SCHEMA views or by querying the system catalogs like pg_class, pg_user, pg_view, etc
- **System Catalog**
  - The place where a relational database management system stores schema metadata
  - Example - Information about tables and columns, and internal bookkeeping information
  - PostgreSQL's system catalogs are regular tables
  - For example, CREATE DATABASE inserts a row into the pg_database catalog — and actually creates the database on disk

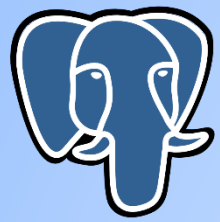


Querying Metadata of PostgreSQL

- **INFORMATION_SCHEMA**
  - It consists of a set of views that contain information about the objects defined in the current database
  - Is defined in the SQL standard and can therefore be expected to be portable and remain stable
- **Task: Define Tables to Test**

```
-- TABLE TEST2 with some CONSTRAINTs and an INDEX
CREATE TABLE TEST2 (ID INTEGER NOT NULL, FIELD1 INTEGER, FIELD2 CHAR(15),
FIELD3 VARCHAR(50), FIELD4 INTEGER, FIELD5 INTEGER, ID2 INTEGER NOT NULL,
PRIMARY KEY(ID2), CONSTRAINT TEST2_FIELD1ID_IDX UNIQUE (ID, FIELD1),
CONSTRAINT TEST4_FIELD4ID_IDX UNIQUE (FIELD4)
);

-- Create index for table TEST2
CREATE INDEX TEST2_FIELD5_IDX ON TEST2(FIELD5);
```



Querying Metadata of PostgreSQL

- **List Tables**
  - Using System Catalogs

```
SELECT relname FROM pg_class
WHERE relname not like ‘pg_%’;
```

  - Using INFORMATION_SCHEMA

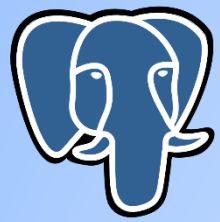
```
SELECT table_name FROM information_schema.tables
WHERE table_schema NOT IN
('pg_catalog', 'information_schema');
```

- **List Table Fields**
  - Using System Catalogs for table created with name test2

```
SELECT a.attname FROM pg_class c, pg_attribute a, pg_type t
WHERE c.relname = 'test2'
AND a.attnum > 0 AND a.attrelid = c.oid AND a.atttypid = t.oid;
```

  - Using INFORMATION_SCHEMA

```
SELECT column_name FROM information_schema.columns
WHERE table_name = 'test2';
```



Querying Metadata of PostgreSQL

- **List Views**
  - Using System Catalogs

```
SELECT viewname FROM pg_views
WHERE schemaname NOT IN ('pg_catalog', 'information_schema')
AND viewname !~ '^pg_';
```

  - Using INFORMATION_SCHEMA

```
SELECT table_name FROM information_schema.tables WHERE table_type = 'VIEW'
AND table_schema NOT IN ('pg_catalog', 'information_schema')
AND table_name !~ '^pg_';
```

- **List Indices**
  - Using System Catalogs for table created with name test2

```
SELECT c.relname AS index_name FROM pg_class AS a
JOIN pg_index AS b ON (a.oid = b.indrelid)
JOIN pg_class AS c ON (c.oid = b.indexrelid)
WHERE a.relname = 'test2';
```

Querying Metadata of PostgreSQL

- **List Users**
  - Using System Catalogs

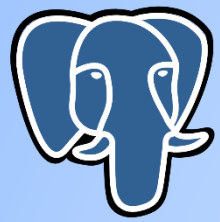
```
SELECT usename FROM pg_user;
```

- **Other Useful Queries**
  - **Query: To retrieve name of the current database**

```
SELECT * FROM information_schema.information_schema_catalog_name
```

  - **Query: To retrieve a list of tables and views defined in the current database**

```
SELECT table_catalog, table_schema, table_name, table_type
    FROM information_schema.tables ORDER BY table_name;
```



Querying Metadata of PostgreSQL

- **Query: To retrieve a list of user defined tables defined in the current database**

```
SELECT table_catalog, table_schema, table_name,
table_type
FROM information_schema.tables
WHERE table_type = 'BASE TABLE‘ AND table_schema
'pg_catalog‘ AND table_schema 'information_schema‘
ORDERBY table_name;
```

- **Query: To list the schemas owned by currently logged in user**

```
SELECT * FROM information_schema.schemata ORDER BY
schema_name;
```

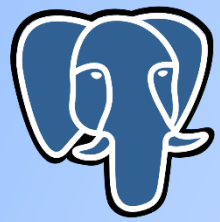
- **Query: To list all the views in the current database for which the currently logged in user has access permissions**

```
SELECT * FROM information_schema.views;
```

PostgreSQL

Table Partitioning

- Dividing the tables into smaller manageable parts is called partitioning
- PostgreSQL supports partitioning via table inheritance
- Basically, you have to create each partition as a child table of the master table
- PostgreSQL partitioning can be implemented in range partitioning or list partitioning
  - **Range partitioning** can be done for example by ID ranges (like 0-100 000, 100 001-200 000, 200 001-300 000…) or Date ranges (like 2009-11-01 – 2009-11-30, 2009-12-01 – 2009-12-31…).
  - **List partitioning** can be done for example by list of cities (like New York, Los Angeles, Chicago, Houston, Philadelphia…) or list of categories (like Programming, Home, Food…).



How to create partition?

- Five easy steps how to make a partitioned table in PostgreSQL
- 1. Create master table
- 2. Create child tables without overlapping table constraints
- 3. Create indexes
- 4. Create trigger function to inserting data to child tables
- 5. Enable constraint exclusion

How to create partition?

1. Create very simple master table. This table can contain simple data

- create table logs (id serial primary key, created_at timestamp without time zone default now(), content text);

2. **Create child tables, which inherits the master table and adds checks for dates, because we want ensure that we have only right data on each partition. We will partition table by date into four quarters of the year**

- create table logs_q1 (check (created_at >= date '2014-01-01' and created_at <= date '2014-03-31')) inherits (logs);
- create table logs_q2 (check (created_at >= date '2014-04-01' and created_at <= date '2014-06-30')) inherits (logs);
- create table logs_q3(check (created_at >= date '2014-07-01' and created_at <= date '2014-09-30')) inherits (logs);
- create table logs_q4
- (check (created_at >= date '2014-10-01' and created_at <= date '2014-12-31')) inherits (logs);

How to create partition?

- create table logs_q4(check (created_at >= date '2014-10-01' and created_at <= date '2014-12-31')) inherits (logs);

3.**Create indexes to child tables to speed up day field usage, because almost all queries (INSERTs, SELECTs and UPDATEs) on the date field.**

- create index logs_q1_created_at on logs_q1 using btree (created_at);
- create index logs_q2_created_at on logs_q2 using btree (created_at);
- create index logs_q3_created_at on logs_q3 using btree (created_at);
- create index logs_q4_created_at on logs_q4 using btree (created_at);

How to create partition?

4. Then we need insert trigger and of course trigger function to master table. Conditions must be exactly the same as what the child tables checks

- create or replace function on_logs_insert() returns trigger as $$

```
begin
if ( new.created_at >= date '2014-01-01' and new.created_at <= date '2014-03-31') then
        insert into logs_q1 values (new.*);
elsif ( new.created_at >= date '2014-04-01' and new.created_at <= date '2014-06-30')
then insert into logs_q2 values (new.*);
elsif ( new.created_at >= date '2014-07-01' and new.created_at <= date '2014-09-30')
then insert into logs_q3 values (new.*);
elsif ( new.created_at >= date '2014-10-01' and new.created_at <= date '2014-12-31')
theni insert into logs_q4 values (new.*);
else raise exception 'created_at date out of range';
end if;
return null;
end; $$ language plpgsql;
```

How to create partition?

Let's attach the trigger function defined above to logs table.

- create trigger logs_insert

  before insert on logs

  for each row execute procedure on_logs_insert();

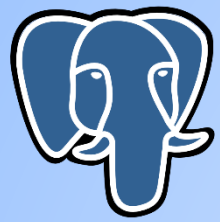
5. **Make sure that Constraint Exclusion is enabled. Constraint exclusion is driven by CHECK constraints. If constraint exclusion is disabled then query is not using check constraints and every query scans thru whole all child tables. So constraint exclusion is very important when using partitioned tables.**

   Let constraint exclusion on with following row on postgresql.conf:

   constraint_exclusion = on

   Set constraint exclusion on following command on psql or psqlrc

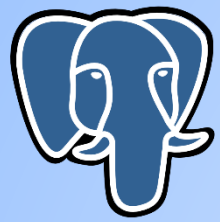
   SET constraint_exclusion = on;



How to create partition?

- **Finally, let's insert some data into logs table to see the partitioning in work.**
- insert into logs (created_at, content) values (date '2014-02-03', 'Content 1'), (date '2014-03-11', 'Content 2'), (date '2014-04-13', 'Content 3'), (date '2014-07-08', 'Content 4'),

  (date '2014-10-23', 'Content 5');
- **Now check the values,**
  - select * from logs;
  - select * from logs_q1;
  - select * from logs_q2;
  - select * from logs_q3;
  - select * from logs_q4;



Identifying and Removing Duplicates

- Relational databases work on the idea that items of data can be uniquely identified.
- Identify the duplicates using a query such as the following:
  - **CREATE UNLOGGED TABLE dup_cust AS SELECT * FROM cust WHERE customerid IN (SELECT customerid FROM cust GROUP BY customerid HAVING count(*) > 1);**
  - An UNLOGGED table can be created with less I/O because it does not write WAL. It is better than a temporary table, because it doesn't disappear if you disconnect and then reconnect.

1. First, we start a transaction:

BEGIN;

2. Lock the table in order to prevent any INSERT, UPDATE or DELETE operations:

LOCK TABLE new_cust IN SHARE ROW EXCLUSIVE MODE;

3. Now, we locate all duplicates, keeping track of the minimum ctid

CREATE TEMPORARY TABLE dups_cust AS SELECT customerid, min(ctid) AS min_ctid

FROM new_cust GROUP BY customerid HAVING count(*) > 1;

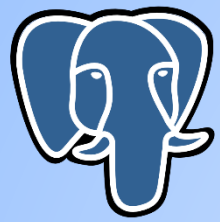
4. Then, we can delete each duplicate

DELETE FROM new_cust USING dups_cust WHERE new_cust.customerid = dups_cust.customerid AND new_cust.ctid != dups_cust.min_ctid;

5. We commit the transaction, which also releases the lock :

COMMIT;

6. Finally, we clean up - **VACUUM new_cust;**



Preventing Duplicate Rows

- Preventing duplicate rows is one of the most important aspects of data quality for any database
- To prevent duplicate rows, we need to create a unique index that the database server can use to enforce uniqueness of a particular set of columns.
  - Create a primary key constraint on the set of columns.
    - **ALTER TABLE newcust ADD PRIMARY KEY(customerid);**
  - Create a unique constraint on the set of columns.
    - **ALTER TABLE newcust ADD UNIQUE(customerid);**
- Note that PostgreSQL allows you to have multiple indexes with exactly the same definition. This is useful in some contexts, but can also be annoying if you accidentally create multiple indexes, as each index has its own cost in terms of writes.

Loading Data From a Spreadsheet

- Spreadsheets are the most obvious starting place for most data stores. Loading data from these sources is a frequent and important task for many DBAs
  - We can then load it into an existing PostgreSQL table, using the following psql command:
    - **postgres=# \COPY sample FROM sample.csv CSV HEADER**
    - **postgres=# SELECT * FROM sample;**

```
 key | value
-----+-------
   1 | c
   2 | d
```

  - Note that the file can include a full file path if the data is in a different directory