PostgreSQL

Module 11

Regular Maintenance

Planning Maintenance

- Database maintenance is about making your database run smoothly.
- Decide a regular date on which to perform certain actions.
- Build a regular cycle of activity around the following tasks:
  - Observe long-term trends in system performance and keep track of the growth of database volumes.
  - Organize regular reviews of written plans, and test scripts. Check the tape rotation, confirm that you still have the password to the offsite backups, and so on.
  - To reduce bloat, as well as collecting optimizer statistics through ANALYZE. Also, regularly check index usage and drop unused indexes.
  - What happens is that a database server gets slower over a very long period. Nobody ever noticed any particular day when it got slow—it just got slower over time.

Controlling Automatic Database Maintenance

- VACUUM reclaims storage occupied by dead tuples
- Tuples that are deleted or obsoleted by an update are not physically removed from their table
- VACUUM can only be performed by a superuser
- VACUUM will skip over any tables that the calling user does not have permission to vacuum
- Adding or deleting a large number of rows, it might be a good idea to issue a VACUUM ANALYZE command for the affected table
- VACUUM [FULL] [FREEZE] [VERBOSE] ANALYZE [table [(column [, ...])]]

Controlling Automatic Database Maintenance

parameters	Explanation
VACUUM ANALYZE	It performs a VACUUM and then an ANALYZE for each selected table.
ANALYZE	It collects statistics about the contents of tables in the database, and stores the results in the pg_statistic system catalog)
COLUMN	The name of a specific column to analyze. Defaults to all columns. If a column list is specified, ANALYZE is implied.

Example : create a big table and insert the values like following procedure

```
postgres=# create table k1 as select * from pg_tables;
postgres=# insert into k1  select * from pg_tables;
postgres=# insert into k1  select * from pg_tables;
postgres=# insert into k1  select * from pg_tables;
```

Controlling Automatic Database Maintenance

```
postgres=# insert into k1 select * from k1;
postgres=# insert into k1 select * from k1;
postgres=# insert into k1 select * from k1;
postgres=# insert into k1 select * from k1;
postgres=# insert into k1 select * from k1;
postgres=# insert into k1 select * from k1;
postgres=# insert into k1 select * from k1;
postgres=# insert into k1 select * from k1;
postgres=# insert into k1 select * from k1;
postgres=# insert into k1 select * from k1;
postgres=# insert into k1 select * from k1;
```

- Check the k1 table if any dead tubles or fragmented is occure or not

```
postgres=# \d pg_stat_all_tables
```

Controlling Automatic Database Maintenance

```
postgres=# select
n_dead_tup ,last_vacuum,last_analyze,n_tup_upd,
n_tup_del,n_tup_hot_upd,relname ,seq_scan,idx_scan  from
pg_stat_all_tables where relname='k1';
```

- update k1 set tableowner='sup2';
- Also, try deleting some records
- And check again
- postgres=# select n_dead_tup ,last_vacuum,last_analyze,n_tup_upd, n_tup_del,n_tup_hot_upd,relname ,seq_scan,idx_scan from pg_stat_all_tables where relname='k1';

Controlling Automatic Database Maintenance

- Autovacuum is enabled by default in PostgreSQL 9.4, and mostly does a great job of maintaining your PostgreSQL database
- You must have both of the following parameters enabled in your postgresql.conf file:
  - **autovacuum = on**
  - **track_counts = on**
- Most of the preceding global parameters can also be set at the table level. For example, if you think that you don't want a table to be autovacuumed, then you can set:
  - ALTER TABLE big_table SET (autovacuum_enabled = off);

Dealing with bloating tables and indexes

- If the database has been maintained without vacuuming or if the data is badly structured, we might experience bloating tables and indexes
- The problem with bloating tables and indexes is that they occupy more storage space than required
- If there are lots of dead rows in a table, the bloat percentage is higher
- how to deal with it:
  - First, we are going to activate the pgstattuple module
    - postgres=# create schema stats;
    - postgres=# create extension pgstattuple with schema stats;
  - create a table and add some rows into it:
    - postgres=# CREATE TABLE num_test AS SELECT * FROM generate_series(1, 100000);

Dealing with bloating tables and indexes

- Use the pgstattuple function, provided by the pgstattuple extension, to examine row-level statistics for the num_test table:
  - postgres=# SELECT * FROM stats.pgstattuple('num_test’);
- delete some data from the num_test table:
  - Postgres=# DELETE FROM num_test WHERE generate_series % 2 = 0;
- reuse the pgstattuple module to examine the table bloat in the num_test table:
  - Postgres=# SELECT * FROM stats.pgstattuple('num_test’);
- vacuum the table in order to remove the table bloat:
  - Postgres=# VACUUM num_test;
- reexamine the row-level statistics for the num_test table:
  - Postgres=# SELECT * FROM stats.pgstattuple('num_test’);

Dealing with bloating tables and indexes

- The following query can help identify whether there are any bloating indexes for a particular table:
  - postgres=# SELECT relname, pg_table_size(oid) as index_size,

    100-(stats.pgstatindex(relname)).avg_leaf_density AS bloat_ratio

    FROM pg_class WHERE relname ~ 'casedemo' AND relkind = ‘i’;
- To overcome the problem of a bloating index, you need to rebuild indexes
- To identify the bloats of an index, we have to use another function called pgstatindex(), as follows:
  - **postgres=# SELECT * FROM pgstatindex('test_idx');**

Removing issues that cause bloat

- Bloat can be caused by long running queries or long running write transactions that execute alongside write-heavy workloads

```
postgres=# SELECT now() - case when backend_xid is not null then xact_start else query_start end as age, pid, backend_xid as xid, backend_xmin as xmin, state FROM pg_stat_activity ORDER BY 1 desc;
```

- You may want to consider setting the idle_in_transaction_session_timeoutparameter so that transactions in that mode will be cancelled

Adding A Constraint Without Checking Existing Rows

- A table constraint is a guarantee that must be satisfied by all the rows in the table
- Therefore, adding a constraint to a table is a two-phase procedure: first, the constraint is created, and then all the existing rows are checked
- How to enforce a constraint on future transactions *only*, without checking existing rows:
  - Enabling the constraint on newer rows of a large table that cannot remain unavailable for a long time.
  - Enforcing the constraint on newer rows, while keeping older rows that are known to violate the constraint.
- The constraint is marked as NOT VALID to make it clear that it does not exclude violations, unlike ordinary constraints.

Adding A Constraint Without Checking Existing Rows

- Example:
  - postgres=# CREATE TABLE ft(fk int PRIMARY KEY, fs text);
  - postgres=# CREATE TABLE pt(pk int, ps text);
  - postgres=# INSERT INTO ft(fk,fs) VALUES (1,'one'), (2,'two’);
  - postgres=# INSERT INTO pt(pk,ps) VALUES (1,'I'), (2,'II'), (3,'III’);
  - We have inserted inconsistent data on purpose so that any attempt to check existing rows will be revealed by an error message
  - Create an ordinary foreign key, we get an error:
    - postgres=# ALTER TABLE pt ADD CONSTRAINT pc FOREIGN KEY (pk) REFERENCES ft(fk);

Adding A Constraint Without Checking Existing Rows

- Example:
  - postgres=# ALTER TABLE pt ADD CONSTRAINT pc FOREIGN KEY (pk) REFERENCES ft(fk) NOT VALID;
  - postgres=# d pt
  - The violation is detected when we try to transform the NOT VALID constraint into a valid one:
    - postgres=# ALTER TABLE pt VALIDATE CONSTRAINT pc;
  - Validation becomes possible after removing the inconsistency:
    - postgres=# DELETE FROM pt WHERE pk = 3;
    - postgres=# ALTER TABLE pt VALIDATE CONSTRAINT pc;
    - postgres=# d pt