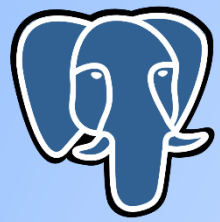
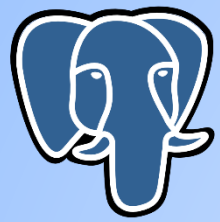


PostgreSQL

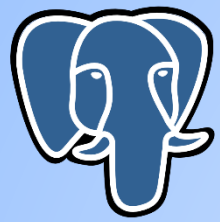
Module 12

Performance and Concurrency



Module Overview

- Understanding Why Queries Slow Down
- Finding out what makes SQL slow
- Checking for memory and I/O
- Reducing the number of rows returned
- Speeding Up Queries Without Rewriting Them
- Parallel Query
- Why a query is not using an index
- Checking for missing indexes
- Forcing a query to use an index
- Finding unused index
- Carefully removing unwanted indexes



Understanding Why Queries Slow Down

- In production environments with large databases and high concurrent access, it might happen that queries that used to run in tens of milliseconds suddenly take several seconds.
- Analyze your database using this code:
  - **db_01=# analyse;**
  - The ANALYZE command updates statistics about data size and data distribution in all tables.
- Looking at **pg_stat_statements** can help you analyze the impact of a particular query in terms of caching, by inspecting two fields: shared_blks_hit and shared_blks_read.
- You must ensure that the performance improvement is not due to caching of the pages required by the requested query. Make sure that you repeat your query several times before classifying it as slow.

Finding Out What Makes SQL Slow

- A SQL statement can be slow for a lot of reasons. Here, we will give a short list of these, with at least one way of recognizing each reason.
- Run the query with EXPLAIN ANALYZE to see how much data is processed in order to complete the query, as follows:
  - **mydb=# EXPLAIN ANALYZE SELECT count(*) FROM t;**
- See how many rows are processed and how many blocks of data are accessed by comparing the output of the following query before and after the query is run on an idle system.
  - **SELECT s.relid, s.schemaname, s.relname, seq_scan, seq_tup_read, idx_scan, idx_tup_fetch, heap_blks_read, heap_blks_hit, idx_blks_read, idx_blks_hit, toast_blks_read, toast_blks_hit**

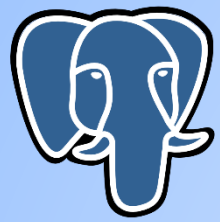
    FROM pg_stat_user_tables s

    JOIN pg_statio_user_tables sio ON s.relid = sio.relid

    WHERE s.schemaname = 'public' AND s.relname = 't'

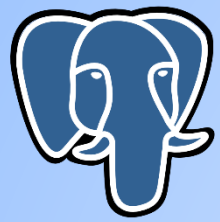
Finding Out What Makes SQL Slow

- Read through just three rows using an index on the serial id column
  - **mydb=# CREATE TABLE events(id SERIAL);**
  - **mydb=# INSERT INTO events SELECT generate_series(1,1000000);**
  - **mydb=# EXPLAIN ANALYZE SELECT * FROM events ORDER BY id DESC LIMIT 3;**
  - **CREATE INDEX events_id_ndx ON events(id);**
  - **EXPLAIN ANALYZE SELECT * FROM events ORDER BY id DESC LIMIT 3;**
- This produces a 10,000-times difference in query runtime, even when all of the data is in the memory



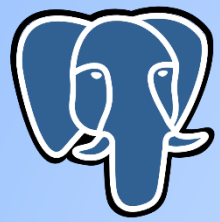
Checking for memory and I/O

- You can inspect memory and I/O
- If track_io_timing on, PostgreSQL will collect information about disk wait and present it to you
- PostgreSQL exposes disk wait in various ways. One way to inspect things is to take a look at pg_stat_database:
  - **test=# \d pg_stat_database**
  - there are two fields towards the end: blk_read_time and blk_write_time - tell you about the amount of time PostgreSQL has spent on waiting for the OS to respond
  - In many cases, high blk_read_time and blk_write_time happen when temp_files and temp_bytes show high numbers



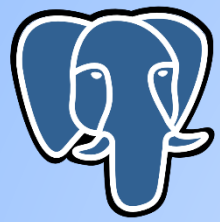
Reducing The Number Of Rows Returned

- Although often the problem is producing many rows in the first place, it is made worse by returning all the unnecessary rows to the client.
- If you need the next 20 documents, don't just query with a limit of 40 and throw away the first 20. Instead, use OFFSET 20 LIMIT 20 to return the next 20 documents
- Create table generate_table as select generate_series(1,50);
- Select * from generate_table offset 20 limit 20;



Speeding Up Queries Without Rewriting Them

- As a first step, you can start providing better information to the optimizer.
- If EXPLAIN ANALYZE reveals that the estimates in the database differ a lot from the metrics returned by the actual query execution, you need to instruct PostgreSQL to collect more fine-grained statistics.
- The current default statistics target can be shown using this command:
  - **SHOW default_statistics_target;**
- If you set the default_statistics_target parameter for a database, then it takes effect the next time anyone connects to the database. So, you should either reconnect or set it for your current session by issuing SET default_statistics_target = 300 before ANALYZE.

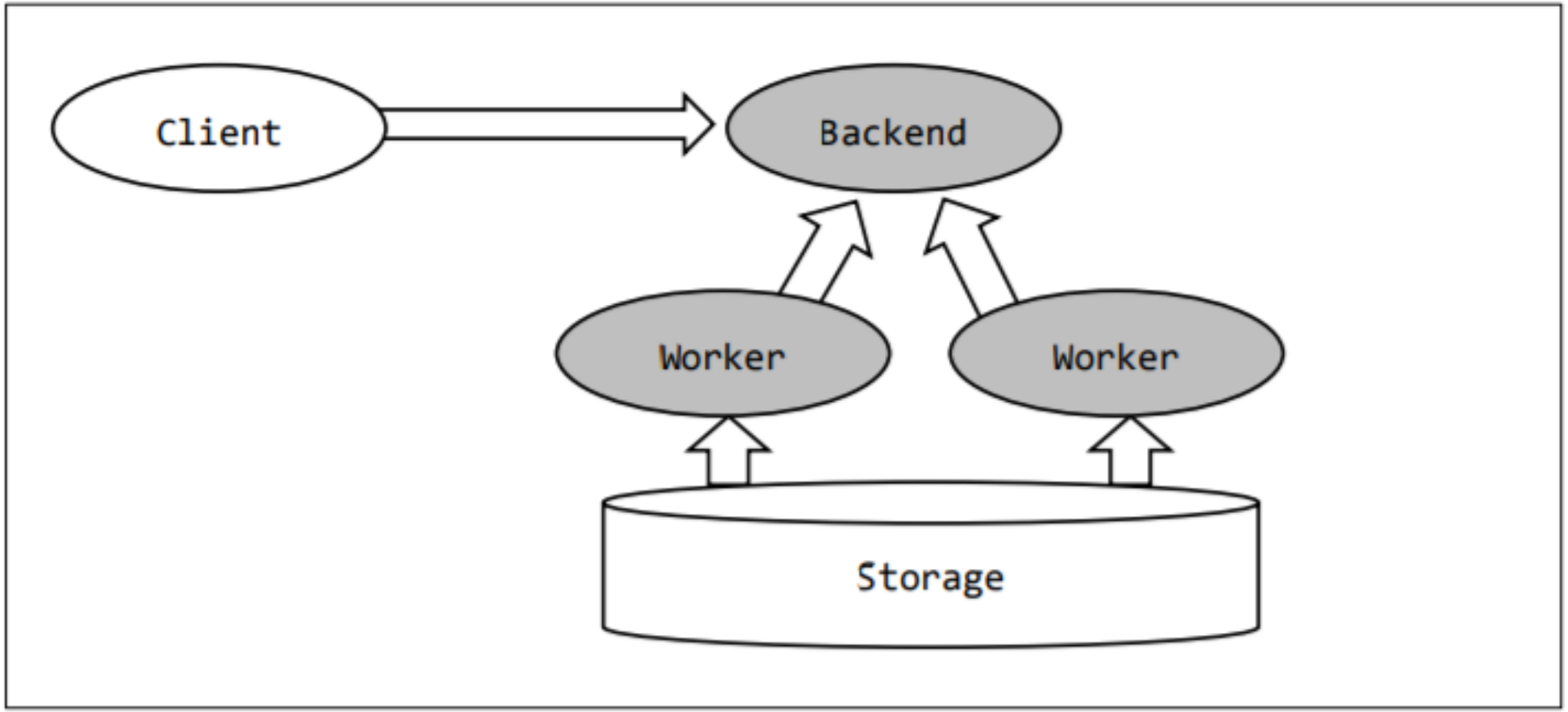


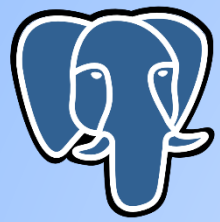
Parallel Query

- All of the SQL statements ware executed only by the back-end process that accepts the connection
- In PostgreSQL 9.6 it is possible to perform parallel processing by multiple worker processes
- Parallel processing can be executed only for Seq Scan, Join and Aggregate
- The degree of parallelism depends on the size of the table
- Processes executing parallel processing use the mechanism of the Background Worker
  - It is determined by the parameter max_parallel_workers_per_gather or max_worker_processes

Parallel Query

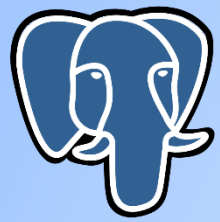
Figure 17 Parallel Seq Scan / Parallel Aggregate





Why A Query Is Not Using An Index

- There can be several reasons for this, but most often, the reason is that the optimizer believes that, based on the available distribution statistics, it is cheaper and faster to use a query plan that does not use an index.
- Force index usage and compare plan costs with an index and without, like this:
  - **demo=# CREATE TABLE itable(id int PRIMARY KEY);**
  - **demo=# INSERT INTO itable SELECT generate_series(1,10000);**
  - **demo=# ANALYZE;**
  - **demo=# EXPLAIN ANALYZE SELECT count(*) FROM itable WHERE id > 500;**
  - **Demo=# set enable_seqscan to false;**
  - **Demo=# explain analyze SELECT count(*) FROM itable WHERE id > 500;**



Checking for missing indexes

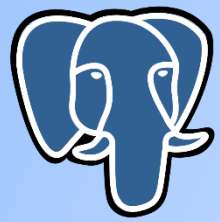
- Missing indexes are super important to achieve super bad performance
- It always makes sense to spend some time on just checking whether all desired indexes are there
  - SELECT schemaname, relname, seq_scan,

    seq_tup_read, idx_scan,

    seq_tup_read / seq_scan AS avg

    FROM pg_stat_user_tables WHERE seq_scan > 0

    ORDER BY seq_tup_read DESC LIMIT 20;



Forcing A Query To Use An Index

- As the PostgreSQL optimizer does not take into account the parallel load caused by other backends, it may make sense to lie to PostgreSQL about some statistics in order to make it use indexes.
- Here, you tell PostgreSQL that it is really very expensive to do sequential scans. It still performs seqscan (instead of failing) if it is the only way to do the query:
  - **mydb=# CREATE TABLE table_with_no_index(id int);**
  - **mydb=# SET enable_seqscan TO false;**
  - **SET**
  - **mydb=# EXPLAIN SELECT * FROM table_with_no_index WHERE id > 10;**

Routine reindexing

- In some scenarios, it is worth rebuilding indexes periodically with the REINDEX command
- With the help of the REINDEX command, index pages that have become completely empty are reclaimed for reuse
- Indexes can be rebuilt at various levels
  - You can recreate the index at the individual index level
    - REINDEX INDEX customer_pkey;
  - you can recreate the indexes at the table level
    - REINDEX TABLE CUSTOMER;
  - You can recreate the indexes at the database level, whereby you recreate all of the indexes within the current database
    - REINDEX DATABASE hrdb;
- The REINDEX command is used to rebuild an index using the data stored in the index's table, thereby replacing the old copy of the index

Routine reindexing

- REINDEX is used in the following situations:
  - REINDEX is to be used when an index becomes corrupted and does not contain any valid data as REINDEX provides a recovery method
  - REINDEX is to be used when an index becomes bloated, that is, when it contains many empty pages
  - REINDEXreduces the space consumption of the index by writing a new version of the index without dead pages
  - REINDEX locks out write operations on the index's parent table but does not block read operations on the table
- There is an option through which the REINDEX command can rebuild an index without locking out write operations on the index's parent table
  - you can use the CREATE INDEX CONCURRENTLY command
    - which will build the index without taking any locks that prevent concurrent inserts, updates, and deletes on the table

Routine reindexing

- So, instead of rebuilding the index, you have to perform the following three steps:
  - First, create an index identical to the one you wish to rebuild using the CREATE INDEX CONCURRENTLY option
  - Next, drop the old index
  - The final step is to rename the new index to the same name as the one that the old index had
- Example :
  - CREATE INDEX CONCURRENTLY card_index ON creditcard (cardno);
  - BEGIN;
  - DROP INDEX credit_card_idx;
  - ALTER INDEX card_index RENAME TO credit_card_idx;
  - COMMIT;

Maintaining indexes

- Autovacuum does not detect bloated indexes, nor does it do anything to rebuild indexes
- PostgreSQL supports commands that will rebuild indexes for you
- the REINDEX command in a convenient way from the operating system:
  - $ reindexdb
- If you want to reindex all databases, then use the following:
  - $ reindexdb –a
- REINDEXputs a full-table lock (AccessExclusiveLock) on the table while it runs
- Try these steps instead:
  - First, let's create a test table with two indexes-a primary key and an additional index-as follows:

Maintaining indexes

- DROP TABLE IF EXISTS test;
- CREATE TABLE test

  (id INTEGER PRIMARY KEY

  ,category TEXT

  , value TEXT);

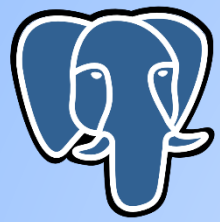
  CREATE INDEX ON test (category);
- let's look at the internal identifier of the tables, oid, and the current file number (relfilenodes), as follows:
  - SELECT oid, relname, relfilenode

    FROM pg_class

    WHERE oid in (SELECT indexrelid

    FROM pg_index

    WHERE indrelid = 'test'::regclass);

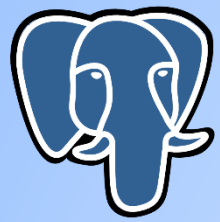


Maintaining indexes

- PostgreSQL supports a command known as CREATE INDEX CONCURRENTLY, which builds an index without taking a full table lock
- PostgreSQL also supports the ability to have two indexes, with different names, that have exactly the same definition
- Let's see that
  - CREATE INDEX CONCURRENTLY new_index ON test (category);
  - BEGIN;
  - DROP INDEX test_category_idx;
  - ALTER INDEX new_index RENAME TO test_category_idx;
  - COMMIT;
- check our internal identifiers again
  - SELECT oid, relname, relfilenode FROM pg_class WHERE oid in (SELECT indexrelid FROM pg_index WHERE indrelid = 'test'::regclass);

Finding Unused Indexes

- Selecting the correct set of indexes for a workload is known to be a hard problem.
- PostgreSQL keeps track of each access against an index. We can view that information and use it to see whether an index is unused, as follows:
  - **postgres=# SELECT schemaname, relname, indexrelname, idx_scan FROM pg_stat_user_indexes ORDER BY idx_scan;**
- The PostgreSQL statistics accumulate various pieces of useful information. These statistics can be reset to zero using an administrator function.
- Also, some indexes that show usage might be showing usage that was historical, and there is no further usage. Or it might be the case that some queries use an index where they could just as easily and almost as cheaply use an alternative index

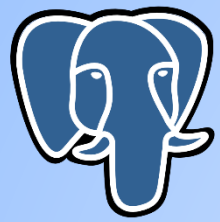


Carefully removing unwanted indexes

- The thinking is that it takes a long time to build an index, and a short time to drop it.
- A procedure that allows us to deactivate an index without actually dropping it:
  - **CREATE OR REPLACE FUNCTION trial_undrop_index(iname TEXT) RETURNS VOID LANGUAGE SQL AS $$ UPDATE pg_index SET indisvalid = true WHERE indexrelid = $1::regclass; $$;**
- Once the index is built and the "valid" flag is set, then if we set the flag to invalid, the index will still be maintained. It's just that it will not be used by queries.
- This allows us to turn off the index quickly, though with the option to turn it on again if we realize that we actually do need the index after all.

Parameters related to performance

Parameter name	Description	Note
autovacuum_work_mem	Memory size of automatic Vacuum	9.4 or later
effective_cache_size	The effective capacity of the disk cache that is available to a single query	
effective_io_concurrency	The number of concurrent disk I / O operation	
maintenance_work_mem	The memory for the maintenance work such as VACUUM, CREATE INDEX, etc.	
shared_buffers	Shared buffer size	
temp_buffers	Memory size for temporary table	
wal_buffers	Shared memory where WAL information is stored	
work_mem	Temporary memory for hash, and sort	
max_wal_size	WAL write amount which becomes a generation opportunity of checkpoint	9.5 or later



Conclusion

- Performance and concurrency are two problems that are often tightly coupled—when concurrency grows, performance usually degrades, in some cases a lot. If you take care of performance problems, you can achieve better concurrency.
- Find slow queries and also how to find queries that make other queries slow.
- Performance tuning, unfortunately, is still not an exact science.