Digital Career Institute

Databases - Basic Performance

Digital Career Institute

Database Workload

Database Workload

The workload on the database depends on two main factors:

The **more operations** required per unit of time the higher the workload will be.

The **more complex** the operations requested the higher the workload will be.

The Complexity of SQL Queries

Digital Career Institute

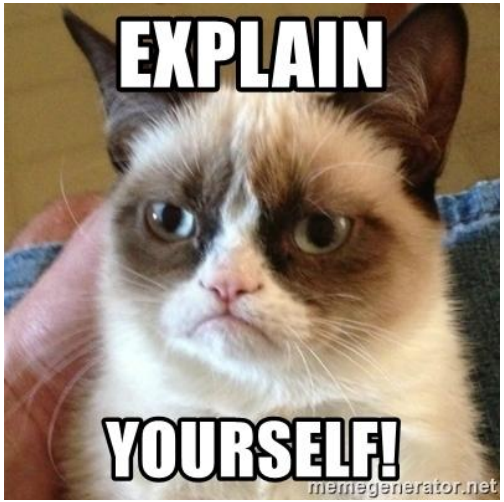
The PostgreSQL console provides a `\timing` command to measure the time spent on each operation.

```
personal=# \timing
Timing is on
personal=# SELECT * FROM people;
 id | first_name | last_name | social_sec
----+------------+-----------+------------
(0 rows)

Time: 46.376 ms
```

The Complexity of SQL Queries

The complexity of SQL queries can be explained using SQL itself.




```
EXPLAIN <SQL_Query>;
```

EXPLAIN

Digital Career Institute

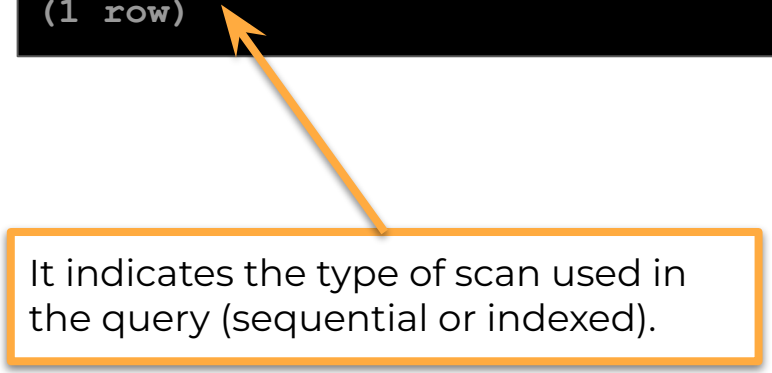
```
EXPLAIN <SQL_Query>;
```

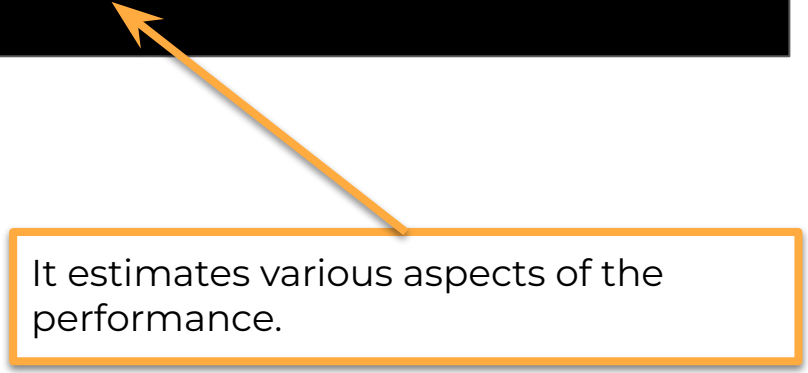
```
personal=# EXPLAIN SELECT * FROM people;
                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on people  (cost=0.00..11.90 rows=190 width=398)
(1 row)
```

EXPLAIN does not execute the query, it just analyzes it.
This may be very useful with malformed queries that take too long to execute.

Reading EXPLAIN Output

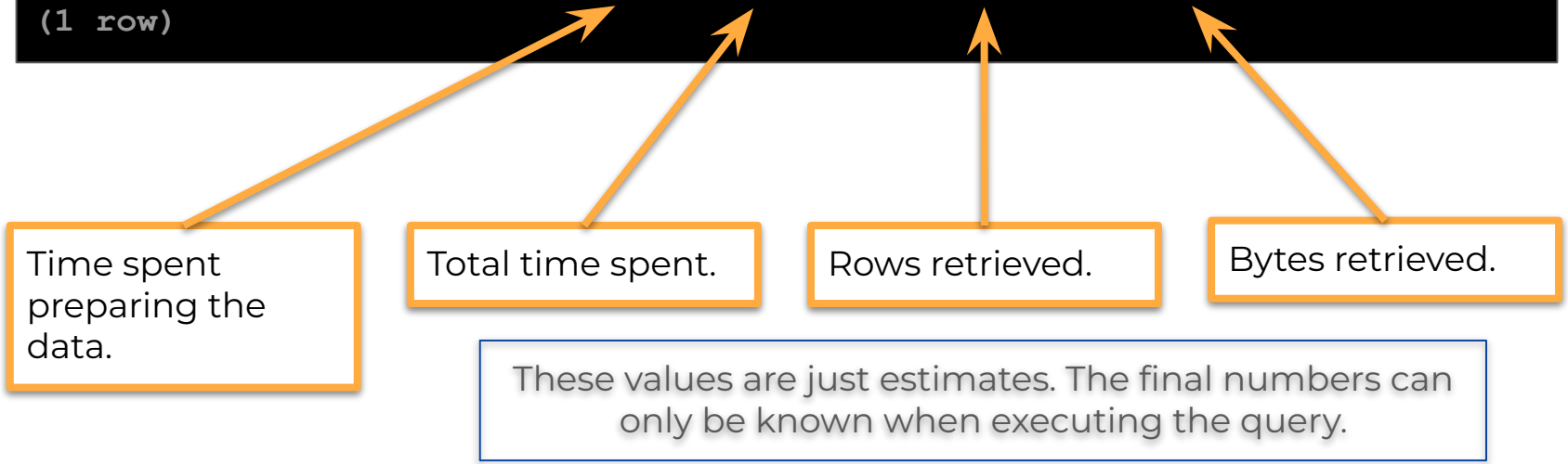
```
personal=# EXPLAIN SELECT * FROM people;
                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on people  (cost=0.00..11.90 rows=190 width=398)
(1 row)
```





Reading EXPLAIN Output

```
personal=# EXPLAIN SELECT * FROM people;
                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on people  (cost=0.00..11.90 rows=190 width=398)
(1 row)
```



Getting the Final Numbers

```
personal=# EXPLAIN ANALYZE SELECT * FROM people;
                           QUERY PLAN
-----------------------------------------------------------------
 Seq Scan on people  (cost=0.00..11.90 rows=190 width=398)
                     (actual time=0.001..0.001 rows=0 loops=1)
 Planning time: 0.064 ms
 Execution time: 0.022 ms
(3 rows)
```

`EXPLAIN ANALYZE` will actually execute the query and provide the real time spent, as well as the final rows returned.

The **`loops`** value indicates the number of times the table (**`people`**) has been looped through. The values in **`actual time`** are for each loop, so the total time is this time multiplied by the number of loops.

Additional Information

```
personal=# EXPLAIN VERBOSE SELECT * FROM people;
                            QUERY PLAN
-------------------------------------------------------------------
 Seq Scan on public.people  (cost=0.00..11.90 rows=190 width=398)
   Output: id, first_name, last_name, social_sec
(2 rows)
```

`EXPLAIN VERBOSE` will add information on the schema used and the columns in the output.

`EXPLAIN ANALYZE VERBOSE` can also be used to combine all information.

Using Explain

```
personal=# EXPLAIN SELECT * FROM people WHERE first_name = 'Maria';
                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on people  (cost=0.00..12.38 rows=1 width=398)
   Filter: ((first_name)::text = 'Maria'::text)
(2 rows)
```

If there is a **`WHERE`** clause but it uses a field without an index, the **`EXPLAIN`** output shows a sequential scan will be done.

The estimated total cost is slightly higher than using a query without a **`WHERE`** clause.

Using Explain

```
personal=# EXPLAIN SELECT * FROM people WHERE id = 1;
                                QUERY PLAN
-------------------------------------------------------------------------
 Index Scan using people_pkey on people  (cost=0.14..8.16 rows=1 width=398)
   Index Cond: (id = 1)
(2 rows)
```

If there is a **`WHERE`** clause and it uses a field with an index, the **`EXPLAIN`** output will change.

The estimated total cost is 30% lower than searching a non-indexed column.

Using Explain

```
personal=# EXPLAIN SELECT * FROM people WHERE last_name = 'Smith';
                                QUERY PLAN
-------------------------------------------------------------------------
 Index Scan using people_last_name_idx on people  (cost=0.14..8.16 rows=1
width=398)
   Index Cond: (id = 1)
(2 rows)
```

A primary index and a secondary index show no performance difference.

Using Explain

```
map=# EXPLAIN SELECT * FROM city, country
map-# WHERE country.id = city.country_id AND code = 'DE';
                                QUERY PLAN
-------------------------------------------------------------------------
 Hash Join  (cost=8.18..22.04 rows=1 width=401)
   Hash Cond: (city.country_id = country.id)
   ->  Seq Scan on city  (cost=0.00..12.80 rows=280 width=259)
   ->  Hash  (cost=8.17..8.17 rows=1 width=142)
         ->  Index Scan using country_code_key on country
                                          (cost=0.15..8.17 rows=1 width=142)
               Index Cond: (code = 'ES'::bpchar)
(6 rows)
```

The first thing PostgreSQL will execute is the indexed filter condition in the country table. Then it will scan the city table sequentially, to combine those records with the cities.

We learned ...

- That the database workload has to do with the amount of transactions requested per minute.
- That is also has to do with the complexity of the SQL transactions requested.
- That we can analyze the complexity of a particular SQL statement with the **`EXPLAIN`** command.
- How to read the output of the **`EXPLAIN`** command and how to get estimates and actual times.

Query Optimization

Populating Tables

Digital Career Institute

1. Use the `COPY` command instead of `INSERT`.
2. Use a single `INSERT` of multiple rows instead of multiple `INSERT` commands.
3. If you use multiple `INSERT` commands, use a single transaction.
4. If possible, remove all indexes from the table before populating.
5. If possible, remove all foreign key constraints.

If the database is in production and users are working on it, removing indexes and foreign key constraints may lead to poor performance and inconsistency.

Optimizing SQL Queries

Digital Career Institute

Some of the most important issues to be optimized have to do with:

1. **Table size**. The larger the table the longer it will take to perform a sequential scan, even if it is only to retrieve one row.
2. **Joins**. If the `JOIN` statements return a lot of rows, the query may be slow.
3. **Aggregations**: Combining multiple rows to produce a result requires more computation than retrieving those rows.

Optimizing SQL Queries

Digital Career Institute

Some tips:

1. **Be picky**. Select only the information you need, both in terms of rows (`WHERE`) and columns (`SELECT`). Use indexed columns to filter data whenever possible.
2. **Sort the joins**. Filter the first table before joining it to the rest. Make sure the initial table size is small, so later joins remain relatively small.
3. **Indexes**. Use indexes to do the first initial filter on the main table. Do not define indexes on columns not used often for searching.
4. **Aggregations**: If possible, avoid using aggregations on common queries with `JOIN` clauses.

Optimizing SQL Queries

Digital Career Institute

Some more tips:

1. Do not use `JOIN` clauses on entire tables if they have many rows.
2. Use proper column types and limits (INT, BIGINT,...).
3. Run `EXPLAIN` on the most common or critical queries to identify issues. Especially when they use custom types of `JOIN` clauses (LEFT, RIGHT,...).
4. Use `JOIN` clauses instead of combining tables with the `FROM` clause. Check the order of the joins and filters used.
5. Do not combine the `IN` operator with subqueries returning big table sizes.
6. Use `VACUUM` often. It reclaims storage and makes the query planner more efficient.

IN Subquery Example

Digital Career Institute

```
SELECT * FROM view1 WHERE
id IN (1,2,3,4,5,6,7,8,9,10)
ORDER BY field1;
```

9ms

```
SELECT * FROM view1 WHERE
id IN (
  SELECT id FROM table ORDER BY field2 LIMIT 10
) ORDER BY field1;
```

25s

IN Subquery Example

Digital Career Institute

```
SELECT * FROM view1 WHERE
id IN (1,2,3,4,5,6,7,8,9,10)
ORDER BY field1;
```

9ms

```
SELECT * FROM view1 WHERE
id = ANY(ARRAY(
  SELECT id FROM table ORDER BY field2 LIMIT 10
)) ORDER BY field1;
```

1.2ms

Application Optimization

Digital Career Institute

Not only the queries must be optimized, but also the amount of queries requested must be efficient.

Applications must make an efficient use of each connection and reduce the number of queries required to obtain the same information.

A common malpractice is to perform **N+1 queries** to obtain the same information we could retrieve with a single query.

N+1 Queries

Digital Career Institute

```
>>> cursor.execute("SELECT * FROM city")
>>> for city in cursor.fetchall():
...     cursor.execute(f"SELECT name FROM country WHERE id = {city[4]}")
...     country = cursor.fetchone()
...     print(f"City name: {city[1]}. Country name: {country[0]}")
...
City name: Berlin. Country name: Germany
City name: Marseille. Country name: France
```

If the table city has 100 rows, the above code will execute 101 queries. That is N+1 queries, N being the number of rows returned in the first query.

Each query is very simple and efficient, but adds an overhead in terms of network latency and database query planning that may be critical.

N+1 Queries

Digital Career Institute

```
>>> cursor.execute("SELECT city.name, country.name "
...                "FROM city WHERE city.country_id = country.id")
>>> for city in cursor.fetchall():
...     print(f"City name: {city[0]. Country name: {city[1]}")
...
City name: Berlin. Country name: Germany
City name: Marseille. Country name: France
```

The same information can be obtained using a single query that combines the appropriate tables.

N+1 Queries

Digital Career Institute

The N+1 performance problem happens more often than expected, especially in complex queries, where it is easier for the developer to write multiple simple queries than writing a more complex (and efficient) one.

This is why it is important to understand the proper use of joins, indexes and checking the database logs to detect these kind of behaviors, as well as using `EXPLAIN` to better understand what happens in each query.

We learned ...

- What are the best ways to populate a table with multiple rows.
- That we must be very cautious when using **`JOIN`** statements.
- That we must try to reduce the total amount of data as soon as possible in our statements.
- That we should use **`EXPLAIN`** to spot the bottlenecks.
- That combining the **`IN`** operator with subqueries may increase the times exponentially.
- That the applications also need to optimize the number of queries requested.

Documentation

Links

Indexes:

- https://www.postgresql.org/docs/current/indexes.html
- https://use-the-index-luke.com/sql/table-of-contents
- https://dataschool.com/sql-optimization/how-indexing-works/

SQL query performance:

- https://www.postgresql.org/docs/current/performance-tips.html
- https://www.postgresql.org/docs/current/sql-explain.html
- https://www.postgresqltutorial.com/postgresql-explain/
- https://www.postgresql.org/docs/current/explicit-joins.html
- https://stackoverflow.com/questions/14987321/postgresql-in-operator-with-subquery-poor-performance/15007154
- https://medium.com/@gwynngroupinc/sql-query-performance-2482bb04aa69
- https://stackoverflow.com/questions/97197/what-is-the-n1-selects-problem-in-orm-object-relational-mapping

Contact Details
DCI Digital Career Institute gGmbH

Digital Career Institute