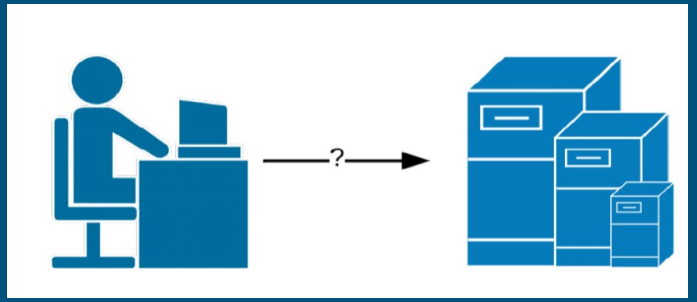
Databases 9

Lecture 9. Query plan. Query optimization. Part 1

You have written a request. What's next?

Connecting to a DBMS



1. The client process accesses the main server process
2. The main server process creates a new one when requesting a connection
3. When the connection is established, the client can send a request to the server
4. The request will be transmitted in plain text
5. There is no request processing at this stage

Request parsing

1. The parser checks the request for the correct syntax
2. The parser returns the result of the work:
    1. Parse tree, if the syntax is correct
    2. Error if the syntax is incorrect. The user gets an error
3. The converter accepts the parse tree as input and converts it into a query tree:
    1. An understanding is formed of which tables, functions and operators the query uses
    2. Accessing the database only within transactions

Request Tree

Special internal representation of SQL queries with its full analysis by key parameters:

- Type of command (SELECT, UPDATE, DELETE, INSERT)
- List of relationships used
- The target ratio in which the result will be recorded
- A list of fields (* is converted to a complete list of all fields)
- A list of restrictions (which are specified in WHERE)
- A list of joins
- Other parameters, for example, ORDER BY

The system of rules

What are you doing:

CREATE VIEW myview AS SELECT * FROM mytab;

What's really going on:

CREATE VIEW myview (same column list as mytab);

CREATE RULE "_RETURN" AS ON

SELECT TO myview DO INSTEAD

SELECT * FROM mytab;

Query rewriting system

1. The request tree is accepted as input
2. References to views are replaced with tables using a system of rules
3. The system returns an updated query tree

Scheduler / Optimizer

The task of the scheduler/optimizer is to build the best

execution plan:

- The same SQL query can be executed in different ways
- The result of their execution will be identical
- Different numbers of resources are spent on different methods

The result of the scheduler is a query plan

Request execution

1. The output takes the plan (tree) of the request
2. The handler recursively bypasses the request plan
3. When accessing each node of the plan, the following should be obtained 1 or more rows, or a message that the output of rows is completed

Request lifecycle:

1. A connection to the DBMS is being created. A request is sent to the DBMS.
2. The parser checks the correctness of the query syntax and creates a query tree.
3. The query rewriting system transforms the query.
4. The scheduler/optimizer creates a query plan.
5. The handler recursively traverses the plan and gets the rows.

How queries are executed

User sends SQL query
Database server:

- Parses the query lexically (lexical analyzer) – identifies tokens (keywords, string and numeric literals)
- Parses the request syntactically (parser) - makes sure that the resulting set of lexemes corresponds to the grammar of the language
  - Result: the request is represented in the memory of the serving process as an abstract syntax tree

How queries are executed

- Parses a query semantically (semantic analyzer) - determines whether there are tables and other objects in the database that the query refers to by name, and whether the user has permission to access these objects
  - Result: the parse tree is rebuilt, supplemented with links to specific database objects, indicating data types and other information
- Transforms (rewrites) a query - to restrict access at the row level, replace parse tree nodes with subtrees (for example, in the case of views or subqueries)
- Schedules the request. At this stage, a plan tree is built, the nodes of which contain not logical, but physical operations on data

Planner

The planner is a PostgreSQL component trying to work out the most efficient way to execute an SQL query.

The execution plan contains information about how the database server will organize the viewing of the tables involved in the request.

Query planner

The scheduler uses a cost optimizer. Essentially this means that the scheduler:

- will build several query execution plans
- will determine their conditional cost (depending on the amount of data and processor resources)
- will choose the most optimal plan (but this is not guaranteed!)

When choosing the most optimal plan, the planner will not use "strict" algorithms, but rather a set of heuristics, as well as available database statistics

Query planner

Why use it?

- See what's going on under the hood
- Evaluate whether the query execution plan is optimal
- Adjust the request, rewriting rules, statistics, etc.

Query planner: syntax

You can display the query plan using the EXPLAIN command

EXPLAIN [(parameter [, ...])] operator
EXPLAIN [ANALYZE] [VERBOSE] statement

Example:
EXPLAIN SELECT * FROM foo;

```
                       QUERY PLAN                    16
------------------------------------------------ -------
  Seq Scan on foo (cost=0.00..155.00 rows=10000 width=4)
(1 row)
```

The EXPLAIN operator

- outputs the execution plan generated by the PostgreSQL scheduler for the specified operator.
- shows how tables affected by the operator will be scanned — just sequentially, by index, etc.
- shows which join algorithm will be selected to combine rows read from tables
- shows the expected cost of executing the request
- MISSING in the SQL standard

The EXPLAIN operator

```
EXPLAIN [ ( option [, ...] ) ] statement
EXPLAIN [ ANALYZE ] [ VERBOSE ] statement

where option can be one of:

  ANALYZE [ boolean ]
  VERBOSE [ boolean ]
  COSTS [ boolean ]
  BUFFERS [ boolean ]
  TIMING [ boolean ]
  FORMAT { TEXT | XML | JSON | YAML }
```

ANALYZE operator

- collects statistical information about the contents of tables in the database and stores the results in the pg_statistic system directory
- analyzes all tables in the current database without parameters
- if the table name is passed in the parameters, processes only the specified table
- if a list of column names is passed in the parameters , statistics will be collected only for these columns
- MISSING in the SQL standard

Query plan: test table

```
CREATE TABLE foo (c1 integer, c2 text);

INSERT INTO foo
   SELECT
      i
      , md5(random()::text)
   FROM
      generate_series(1, 1000000) AS i;
```

Query plan: simple sampling

```
EXPLAIN SELECT * FROM foo;

----
QUERY PLAN
Seq Scan on foo  (cost=0.00..18334.00 rows=1000000 width=37)
```

There are several ways to read data from a table . In our case, EXPLAIN reports that Seq Scan is used — sequential, block by block, reading of table data.

Query plan: simple sampling

What is cost? This is not time, but a kind of spherical concept in a vacuum, designed to assess the cost of the operation.

- The first value 0 is the cost of getting the first row.
- The second is 18334 — the cost of getting all the lines.

rows — the approximate number of rows returned when performing the Seq Scan operation. Important! Now no rows are being subtracted from the table. At all. Therefore, the value is approximate.

width — the average size of a single string in bytes.

Where does the statistics come from?

```
SELECT relpages, reltuples
  FROM pg_class
 WHERE relname = 'foo';
----
relpages | reltuples
--------------------
8334     | 1000010
```

Statistics are updated only when you run VACUUM or ANALYZE

Query plan: simple sampling

```
EXPLAIN (ANALYZE) SELECT * FROM foo;

----
QUERY PLAN
Seq Scan on foo  (cost=0.00..18334.10 rows=1000010 width=37)
            (actual time=0.402..97.000 rows=1000010 loops=1)
Planning time: 0.042 ms
Execution time: 138.229 ms
```

Such a request will be executed in reality. If you do EXPLAIN (ANALYZE) for INSERT, DELETE or UPDATE, your data will change.

Don't forget about ROLLBACK

Indexes

Indexes in PostgreSQL are special database objects designed mainly to speed up data access.

These are auxiliary structures: any index can be deleted and restored from the information in the table.

Indexes

- Indexes are database objects designed primarily to speed up data access
- By using an index, the database server can find and retrieve the rows it needs much faster than without it. However, indexes are associated with an additional load on the DBMS as a whole, so they should be used carefully
- Postgres core has 6 built-in types of indexes:
  - Btree (default)
  - hash
  - gist
  - spgist
  - gin
  - brin

Indexes: syntax

The syntax is complex and branched:

```
CREATE [ UNIQUE ] INDEX [ CONCURRENTLY ] [ [ IF NOT EXISTS ] name ] ON [ ONLY ]
table_name [ USING method ]
    ( { column_name | ( expression ) } [ COLLATE collation_rule ] [ operator_class
[ ( op_class_parameter = value [, ... ]) ] ] [ ASC | DESC ] [ NULLS { FIRST | LAST } ] [, ...]
)
    [ INCLUDE ( column_name [, ...] ) ]
    [NULLS [NOT] DISTINCT]
    [ WITH ( storage_parameter [= value] [, ... ]) ]
    [ TABLESPACE table_space ]
    [WHERE predicate]
```


But for ease of understanding it will be enough for us:

```
CREATE INDEX name ON table USING type (column)
```

Indexes: b-tree

B-trees can handle equality conditions and range tests with data that can be sorted in some order. The query planner may use such an index when the indexed column is subject to a comparison with one of the following operators: <, <=, =, >=, >

The optimizer can use these indexes in queries with the LIKE and ~ pattern comparison operators if the pattern is defined by a constant and it is bound to the beginning of the string - for example, col LIKE 'foo%' or col ~ '^foo', but not col LIKE '%bar'. By default this only works with the C locale (i.e. ASCII characters)

Indexes: hash

- Hash indexes store a 32-bit hash code derived from the value of the indexed column, so hash indexes only work with simple equality conditions.
- The query planner can apply a hash index only if the column being indexed is subject to a comparison with the = operator

Indexes: gist (generalized search tree)

- It is not one specific index, but an "infrastructure" that allows many different indexing strategies to be implemented
- Allows you to work, for example, with geometric objects, texts for full-text search, arrays and other complex data structures
- Depending on the settings, it supports a large number of operators (for example, for two-dimensional geometric data types: <<, &<, &>, >> , <<|, &<|, etc.)

Indexes: spgist (space-partitioned generalized search tree)

- Roughly speaking, we can say that this is a special case of gist
- Supports indexing of data structured according to the “space sharing” principle. This means that spgist is effective for data that is naturally divided into disjoint subsets. spgist provides higher efficiency and smaller index size for certain types of queries and data structures compared to GIST

Indexes: gin (generalized inverted index)

- This is an index type corresponding to the inverted index data structure
- Effective for searching sets by their subsets (i.e. when the query is a subset of the desired set)

Indexes: brin (block range index)

- Designed to work with very large tables where the data is relatively ordered by the value of the indexed column
- Stores generalized information about the values located in physically sequentially located blocks of the table. Therefore, such indexes are most effective on columns whose values correlate well with the physical order of the table columns

Query plan: WHERE

```
EXPLAIN (ANALYZE) SELECT * FROM foo WHERE c1 > 500;

----
QUERY PLAN
Seq scan on foo  (cost=0.00..12500.71 rows=416671 width=37)
            (actual time=0.059..54.462 rows=333337 loops=3)
  Filter: (c1 > 500)
  Rows Removed by Filter: 510
Planning time: 0.175 ms
Execution time: 693.216 ms
```

Query plan: WHERE + INDEX (many rows)

```
CREATE INDEX ON foo(c1);
EXPLAIN (ANALYZE) SELECT * FROM foo WHERE c1 > 500;
----
QUERY PLAN
Seq Scan on foo  (cost=0.00..20834.12 rows=999522 width=37)
(actual time=0.010..139.253 rows=999500 loops=1)
  Filter: (c1 > 500)
  Rows Removed by Filter: 510
Planning time: 0.096 ms
Execution time: 180.288 ms
```

Query plan: WHERE + INDEX (few rows)

```
SET enable_seqscan TO on;
EXPLAIN (ANALYZE) SELECT * FROM foo WHERE c1 < 500;
----
QUERY PLAN
Index Scan using foo_c1_idx on foo
                        (cost=0.42..24.96 rows=488 width=37)
                 (actual time=0.011..0.162 rows=509 loops=1)
  Index Cond: (c1 < 500)
Planning time: 0.148 ms
Execution time: 0.201 ms
```

Query plan: subtotal

Now we know:

- Seq Scan — the whole table is read.
- Index Scan — an index is used for WHERE conditions, reads the table when selecting rows.
- Bitmap Index Scan — first Index Scan, then sampling control by table. Effective for a large number of rows.
- Index Only Scan is the fastest. Only the index is read.

Request plan: ORDER BY

```
EXPLAIN (ANALYZE) SELECT * FROM foo ORDER BY c1;


----
QUERY PLAN
Gather Merge (cost=63789.95..161019.97 rows=833342 width=37)
         (actual time=288.249..650.175 rows=1000010 loops=1)
        ->  Sort  (cost=62789.92..63831.60 rows=416671 width=37)
              (actual time=266.951..314.654 rows=333337 loops=3)
        Sort Key: c1
        Sort Method: external sort  Disk: 16888kB
        ->  Parallel Seq Scan on foo
                     (cost=0.00..12500.71 rows=416671 width=37)
                (actual time=0.059..54.462 rows=333337 loops=3)
Planning time: 0.175 ms
Execution time: 693.216 ms
```

Query Plan: LIMIT

```
EXPLAIN (ANALYZE, BUFFERS)
   SELECT * FROM foo WHERE c2 LIKE 'ab%' LIMIT 10;


----
QUERY PLAN
Limit  (cost=0.00..20.63 rows=10 width=37)
(actual time=0.186..0.577 rows=10 loops=1)
  Buffers: shared hit=19
  ->  Seq Scan on foo  (cost=0.00..20834.12 rows=10101 width=37)
                       (actual time=0.184..0.567 rows=10 loops=1)
        Filter: (c2 ~~ 'ab%'::text)
        Rows Removed by Filter: 2240
        Buffers: shared hit=19
Planning time: 0.886 ms
Execution time: 0.691 ms
```

Query Plan: JOIN

```
CREATE TABLE bar (c1 integer, c2 boolean);

INSERT INTO bar
  SELECT
    i
    , i % 2 = 1
  FROM
    generate_series(1, 500000) AS i;

ANALYZE bar;
```

```
EXPLAIN (ANALYZE)
   SELECT * FROM foo JOIN bar ON foo.c1 = bar.c1;
----
QUERY PLAN
Hash Join  (cost=15417.00..60081.14 rows=500000 width=42)
        (actual time=185.091..982.071 rows=500010 loops=1)
  Hash Cond: (foo.c1 = bar.c1)
  ->  Seq Scan on foo  (cost=0.00..18334.10 rows=1000010 width=37)
                  (actual time=0.053..224.915 rows=1000010 loops=1)
  ->  Hash  (cost=7213.00..7213.00 rows=500000 width=5)
     (actual time=182.530..182.530 rows=500000 loops=1)
        Buckets: 131072  Batches: 8  Memory Usage: 3282kB
        ->  Seq Scan on bar  (cost=0.00..7213.00 rows=500000 width=5)
                        (actual time=0.024..73.284 rows=500000 loops=1)
Planning time: 4.460 ms
Execution time: 1005.429 ms
```

Query plan: JOIN + INDEX

```
CREATE INDEX ON foo(c1);
CREATE INDEX ON bar(c1);
EXPLAIN (ANALYZE)
  SELECT * FROM foo JOIN bar ON foo.c1 = bar.c1;
----
QUERY PLAN
Merge Join  (cost=1.33..39748.36 rows=500000 width=42)
      (actual time=0.014..428.590 rows=500010 loops=1)
  Merge Cond: (foo.c1 = bar.c1)
  ->  Index Scan using foo_c1_idx on foo
                  (cost=0.42..34317.58 rows=1000010 width=37)
             (actual time=0.007..127.049 rows=500011 loops=1)
  ->  Index Scan using bar_c1_idx on bar
                    (cost=0.42..15212.42 rows=500000 width=5)
             (actual time=0.005..112.125 rows=500010 loops=1)
Planning time: 0.435 ms
Execution time: 450.289 ms
```

Query Plan: Join Methods

- Nested loop (pseudocode):
  for i in first_table:
  for j in second_table where second_table.i = i:
  check the conditions and form a string
- Hash join (pseudocode):
  build a hash table from first_table
  for j in second_table:
  if key_exists(hash(second_table.j)):
  check the conditions and form a string
- Merge join (pseudocode):
  merge two sorted first_table and second_table
  check the conditions and form a string

Query plan: Nested loop

Behind:

- Very cheap
- Very fast on small volumes
- Does not require a lot of memory
- Ideal for lightning fast queries
- The only one can connect not only by equality

Against:

- Doesn't work well for large amounts of data

Query plan: Hash join

Behind:

- No index needed
- Relatively fast
- Can be used for FULL OUTER JOIN

Against:

- Loves memory
- Equality join only
- Dislikes many values in join columns
- Long time to get the first line

Query plan: Merge join

Behind:

- Fast on large and small volumes
- Does not require a lot of memory
- Can OUTER JOIN
- Suitable for joining more than two tables

Against:

- Requires sorted data streams, which means or index, or sort
- Equality join only

The planner is not stupid

```
DROP INDEX foo_c1_idx;
DROP INDEX bar_c1_idx;

EXPLAIN (ANALYZE)
SELECT * FROM foo, bar WHERE foo.c1 = bar.c1;

EXPLAIN (ANALYZE)
SELECT * FROM foo CROSS JOIN bar WHERE foo.c1 = bar.c1;

EXPLAIN (ANALYZE)
SELECT * FROM foo INNER JOIN bar ON foo.c1 = bar.c1;
```

Links

https://www.dalibo.org/_media/understanding_explain.pdf

http://langtoday.com/?p=229

http://langtoday.com/?p=270

https://habr.com/ru/post/203320/