PostgreSQL Tools for Hunting Down and Fixing Non-Optimal Queries

Ekaterina Sokolova (aka Cate)
Developer at Postgres Professional

Increasing DBMS performance

- ✦ provide high-quality services
- ✦ increase response speed to be competitive
- ✦ optimize processes to avoid problems when loads increase
- ✦ to strive to become better

A Beginner's Guide to Detectives

- searching for suspects
- interrogation of suspects: is it really suboptimal or not
- neutralize the culprits

SUSPECTS:
Non-optimal queries

1. Queries with long time of execution

1. Queries with long time of execution

Let's save sql commands that take longer than a certain amount of time to execute:

Add
log_min_duration_statement = <time_in_ms>
to postgresql.conf

1. Queries with long time of execution

Let's save sql commands that take longer than a certain amount of time to execute:

Add

log_min_duration_statement = <time_in_ms>

to postgresql.conf

Reload configuration:

SELECT pg_reload_conf();

OR

restart server: **pg_ctl restart**

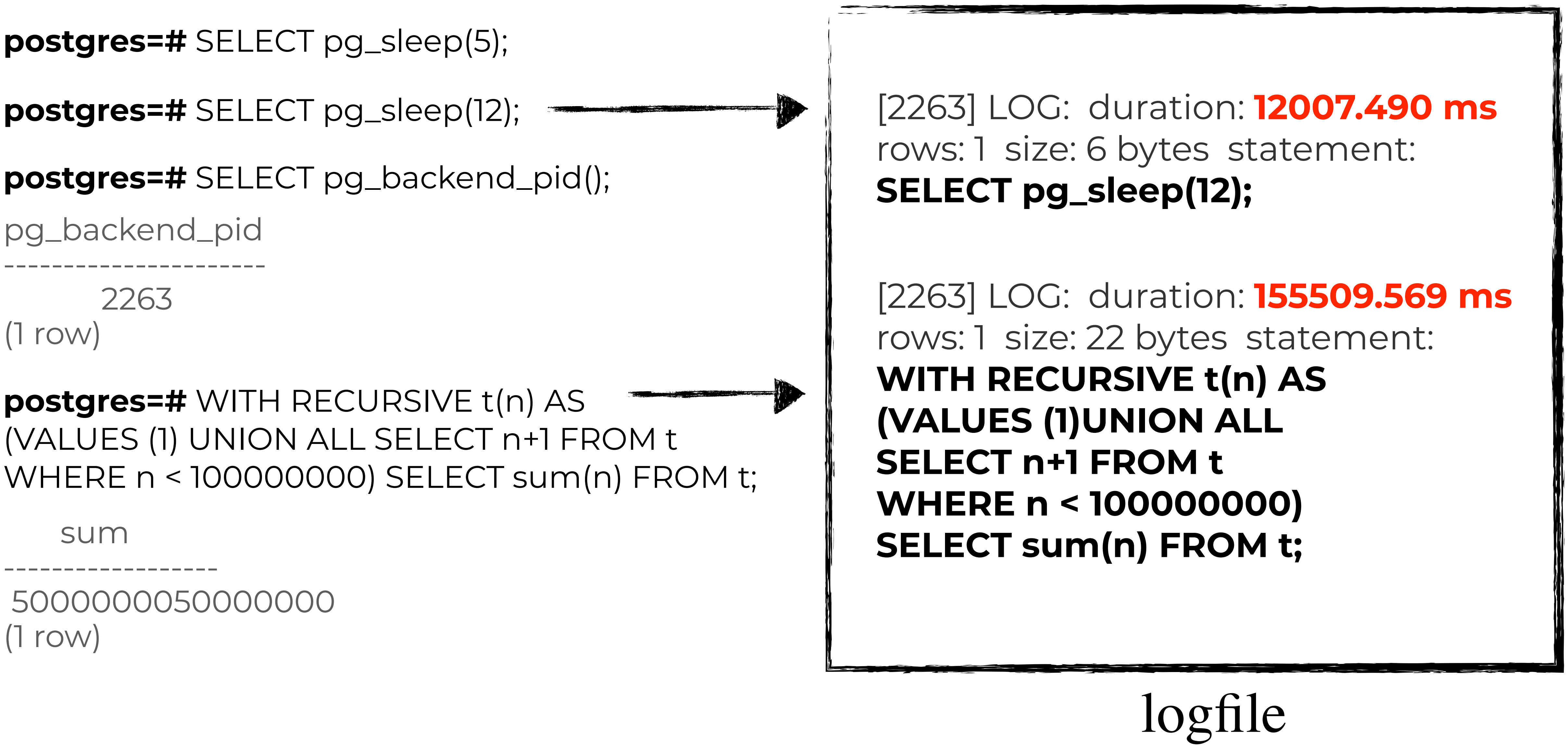
1. Queries with long time of execution

```
log_min_duration_statement = 10000
```

```
postgres=# SELECT pg_sleep(5);

postgres=# SELECT pg_sleep(12);

postgres=# SELECT pg_backend_pid();
pg_backend_pid
----------------------
          2263
(1 row)

postgres=# WITH RECURSIVE t(n) AS
(VALUES (1) UNION ALL SELECT n+1 FROM t
WHERE n < 100000000) SELECT sum(n) FROM t;

      sum
------------------
 5000000050000000
(1 row)
```

1. Queries with long time of execution

```
log_min_duration_statement = 10000
```



```
postgres=# SELECT pg_sleep(5);

postgres=# SELECT pg_sleep(12);

postgres=# SELECT pg_backend_pid();
pg_backend_pid
----------------------
        2263
(1 row)

postgres=# WITH RECURSIVE t(n) AS
(VALUES (1) UNION ALL SELECT n+1 FROM t
WHERE n < 100000000) SELECT sum(n) FROM t;

     sum
------------------
 5000000050000000
(1 row)
```

```
[2263] LOG:  duration: 12007.490 ms
rows: 1  size: 6 bytes  statement:
SELECT pg_sleep(12);

[2263] LOG:  duration: 155509.569 ms
rows: 1  size: 22 bytes  statement:
WITH RECURSIVE t(n) AS
(VALUES (1)UNION ALL
SELECT n+1 FROM t
WHERE n < 100000000)
SELECT sum(n) FROM t;
```

logfile

Not every long-running operation is bad and not every suboptimal action is long.

Spying on statements

pg_stat_statements

Module **pg_stat_statements** — tracking execution statistics of all SQL statements executed by a server.

1. Add to postgresql.conf

```
shared_preload_libraries = 'pg_stat_statements'
```

2. Into psql

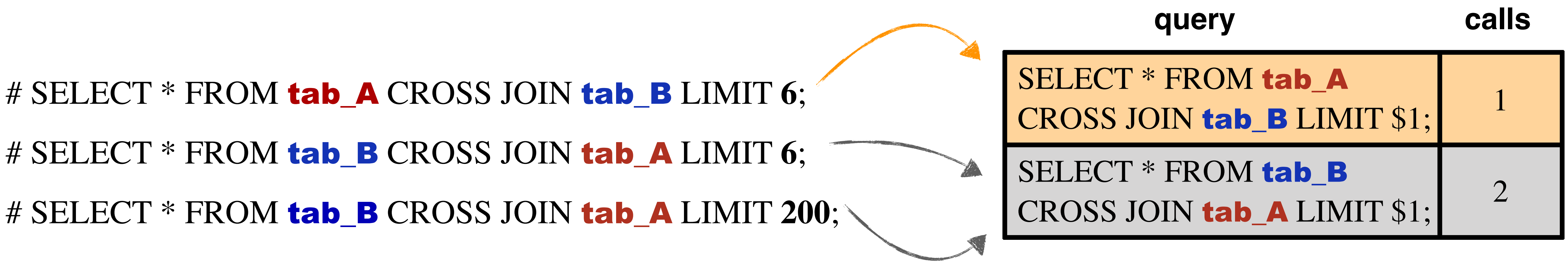
```
CREATE EXTENSION pg_stat_statements;
```

Spying on statements

pg_stat_statements

query, -- text of query

calls, -- number of times executed



```
# SELECT * FROM tab_A CROSS JOIN tab_B LIMIT 6;
# SELECT * FROM tab_B CROSS JOIN tab_A LIMIT 6;
# SELECT * FROM tab_B CROSS JOIN tab_A LIMIT 200;
```

query	calls
SELECT * FROM tab_A CROSS JOIN tab_B LIMIT $1;	1
SELECT * FROM tab_B CROSS JOIN tab_A LIMIT $1;	2

Spying on statements

pg_stat_statements

query,	-- text of query
calls,	-- number of times executed
total_exec_time,	-- total time spent in the statement
min_exec_time,	
max_exec_time,	
mean_exec_time,	
stddev_exec_time,	-- population standard deviation of time spent
rows	-- total number of rows retrieved or affected

Spying on statements

pg_stat_statements

query,	-- text of query
calls,	-- number of times executed
total_exec_time,	-- total time spent in the statement
min_exec_time,	
max_exec_time,	
mean_exec_time,	
stddev_exec_time,	-- population standard deviation of time spent
rows	-- total number of rows retrieved or affected

2. «Suspicious» queries

Spying on statements

pg_stat_statements

query,	-- text of query
calls,	-- number of times executed
total_exec_time,	-- total time spent in the statement
min_exec_time,	
max_exec_time,	
mean_exec_time,	
stddev_exec_time,	-- population standard deviation of time spent
rows	-- total number of rows retrieved or affected

```
SELECT query, calls,
total_exec_time, min_exec_time,
max_exec_time, mean_exec_time,
stddev_exec_time, rows
FROM pg_stat_statements
ORDER BY     max
            mean_exec_time  DESC;
```

SELECT query, calls,

total_exec_time, min_exec_time,

max_exec_time, mean_exec_time,

stddev_exec_time, rows

FROM pg_stat_statements

ORDER BY **max** **~~mean_exec_time~~** **DESC;**

total_exec_time

3. Frequent queries

query	SELECT $1
calls	4293
total_exec_time	20.360238999999982
min_exec_time	0.002057
max_exec_time	0.19523300000000002
mean_exec_time	0.004742659911483821
stddev_exec_time	0.008009316879670965
rows	4293

SELECT query, calls,

total_exec_time, min_exec_time,

max_exec_time, mean_exec_time,

stddev_exec_time, rows

FROM pg_stat_statements

ORDER BY ~~**max**~~ ~~**mean_exec_time**~~ **DESC;**

total_exec_time

3. Frequent queries

```
SELECT query, calls,
total_exec_time, min_exec_time,
max_exec_time, mean_exec_time,
stddev_exec_time, rows,
(100 * total_exec_time / sum(total_exec_time)
OVER ()) AS cpu_perc
FROM pg_stat_statements
ORDER BY   cpu_perc  DESC;
```

A Beginner's Guide to Detectives

✓ searching for suspects

✦ interrogation of suspects: is it really suboptimal or not and what we can do

✦ neutralize the culprits

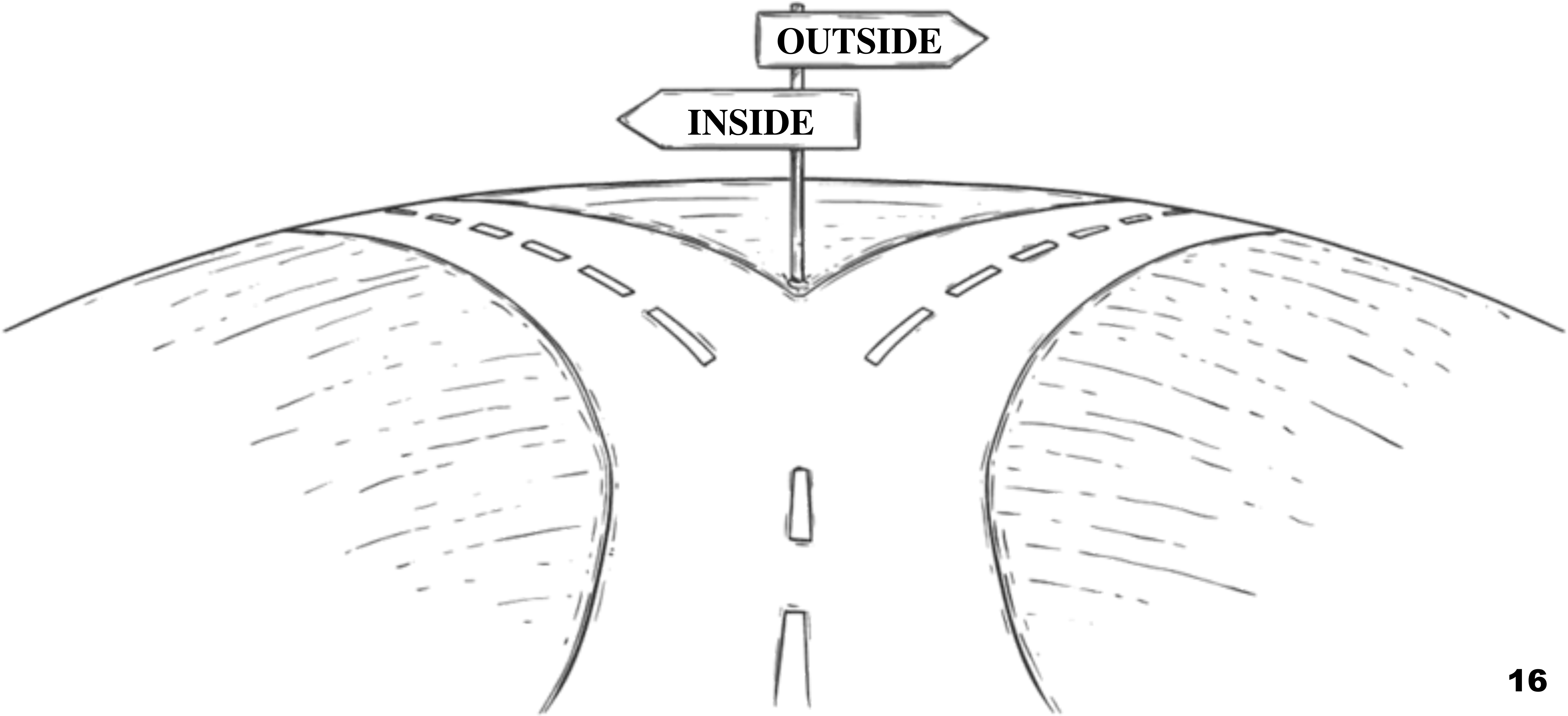
Get rid of non-optimality

Ability to manage queries

- ✦ change work routine

Unable to manage queries

- ✦ set up infrastructure

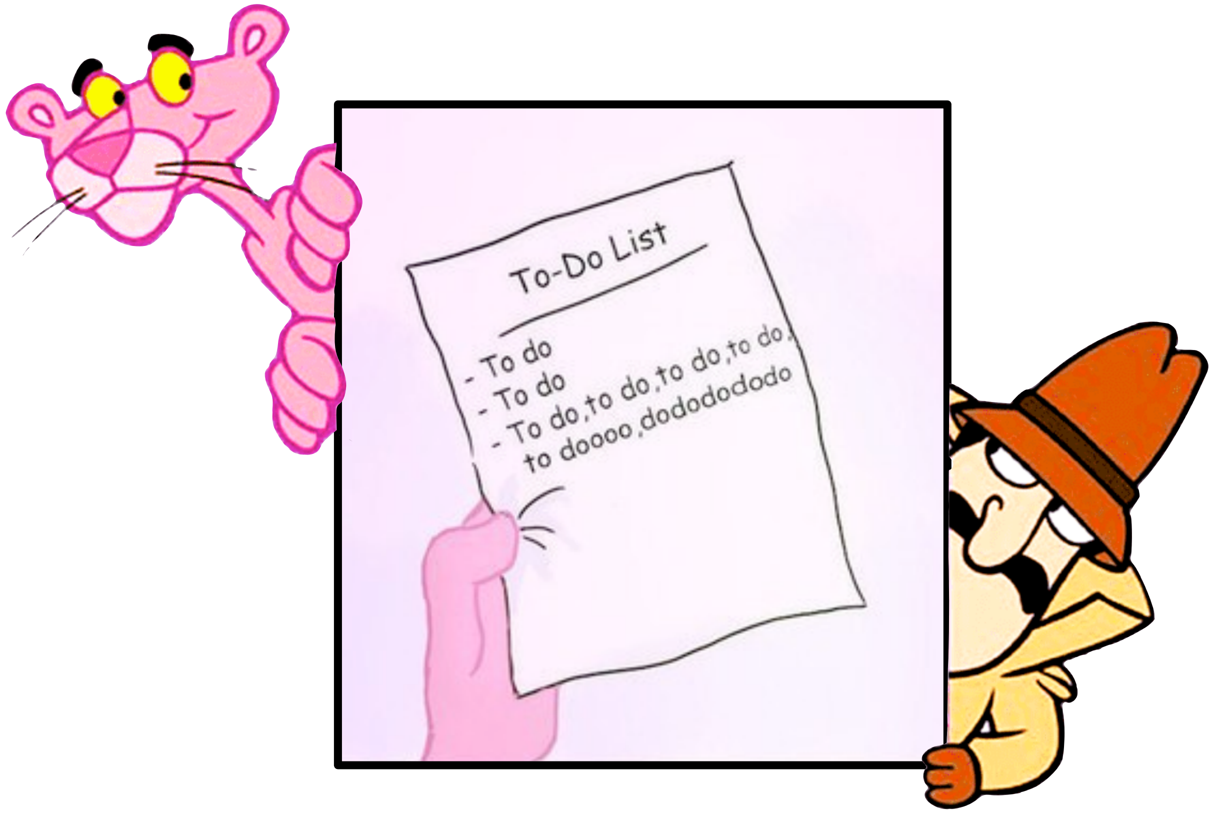


Checklist of questions

- ✦ Do we really need this query?

Checklist of questions

- Do we really need this query?
  - Get rid of unnecessary queries
  - Create MATERIALIZED VIEWs for a frequently repeated query on a rarely changing data set

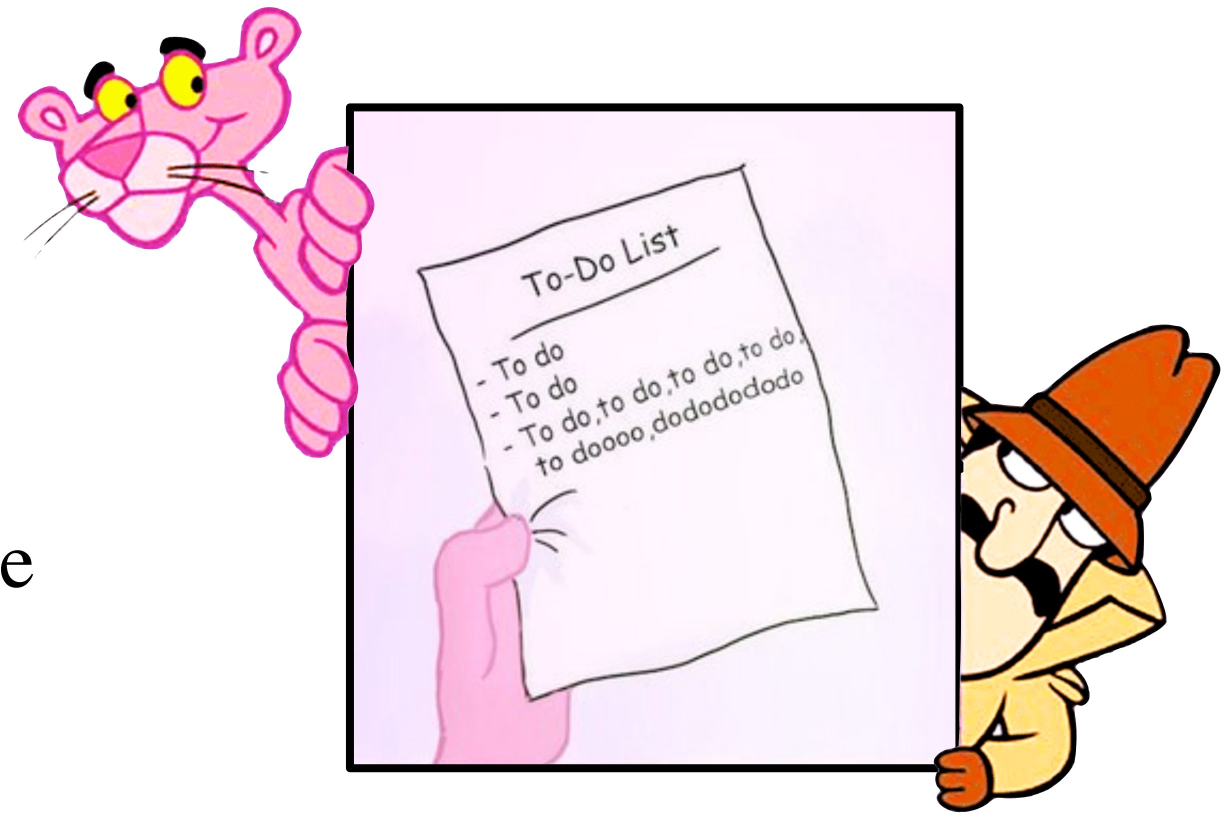


Checklist of questions

- Do we really need this query?
- Do we need it at a particular time?

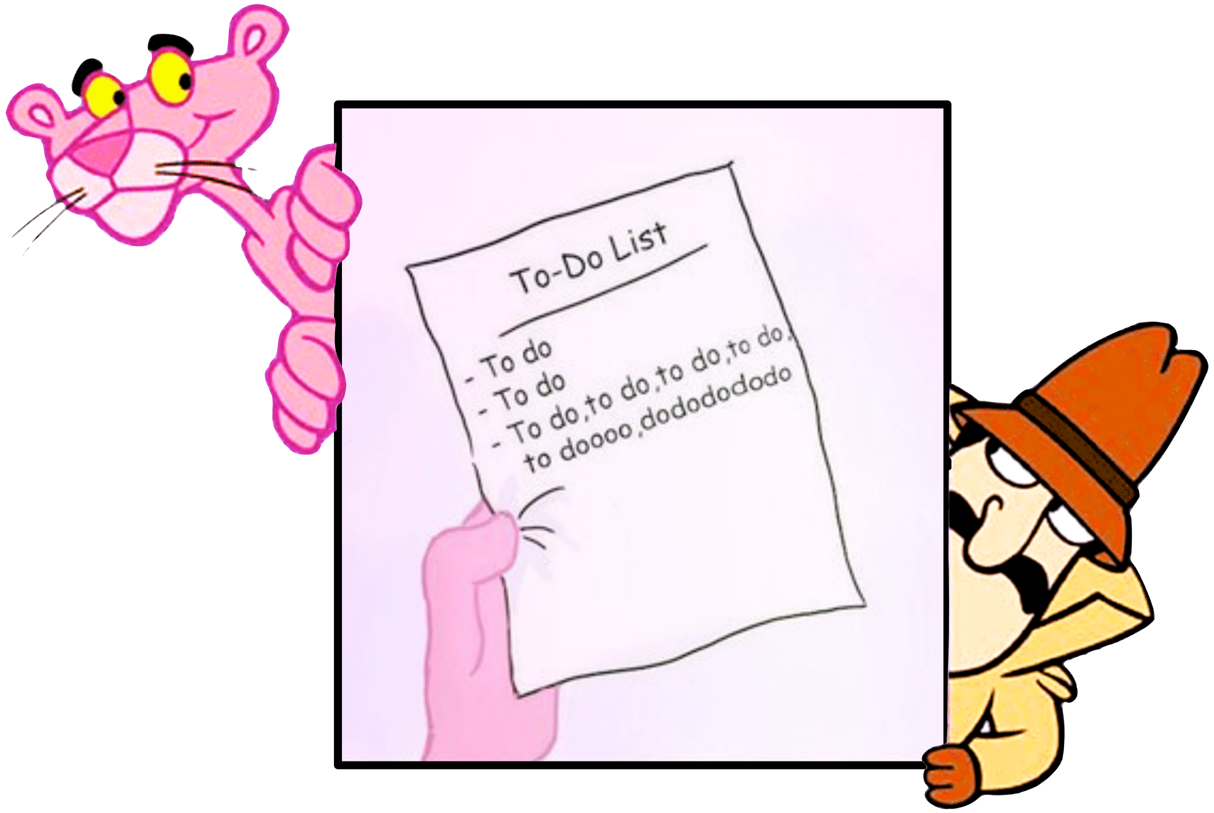
Checklist of questions

- Do we really need this query?
- Do we need it at a particular time?
  - Can it be postponed until a period of lower server load?
  - Can it be scheduled to run after more urgent queries on the same data to avoid blocking?



Checklist of questions

- Do we really need this query?
- Do we need it at a particular time?
- Was there anything blocking it?

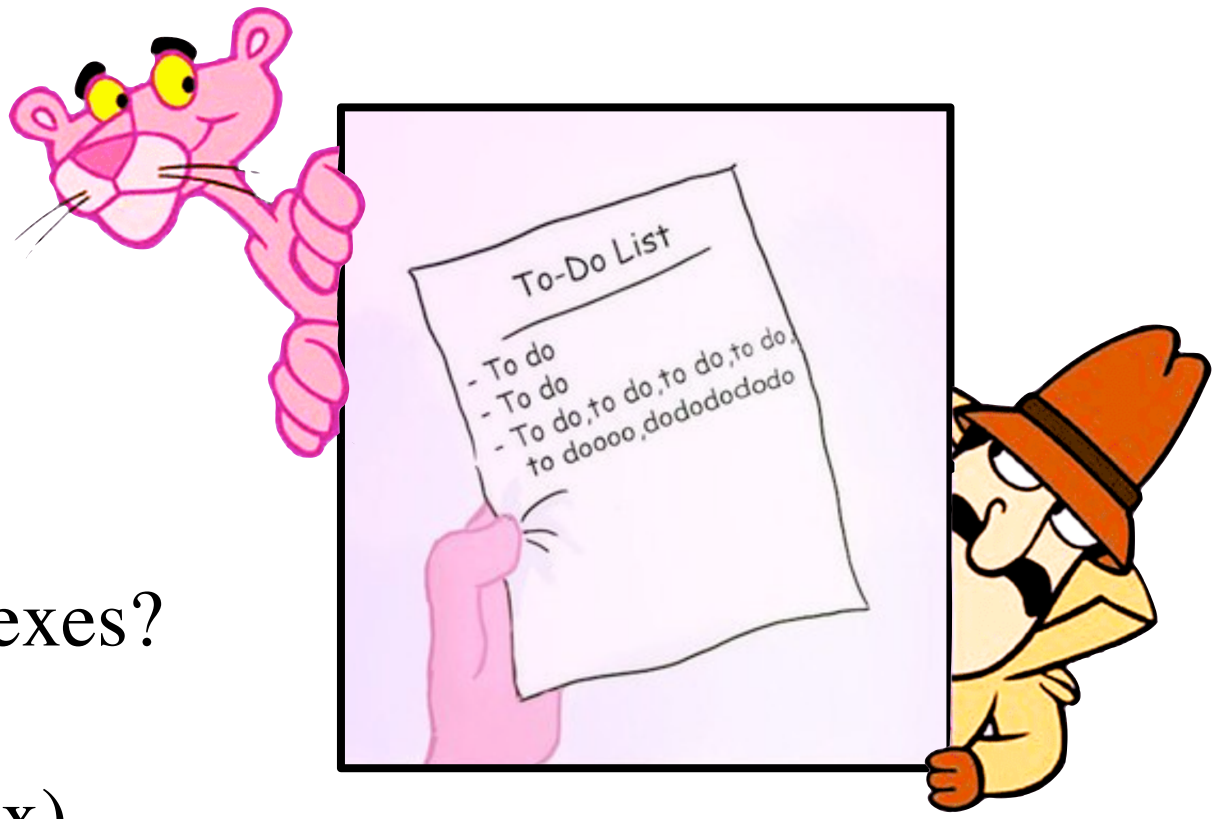


Checklist of questions

- Do we really need this query?
- Do we need it at a particular time?
- Was there anything blocking it?
- Is there type match everywhere?
    - Java type -> SQL type
    - Variable type 🤝 operator
    - Index for x, but select x^2

Checklist of questions

- Do we really need this query?
- Do we need it at a particular time?
- Was there anything blocking it?
- Is there type match everywhere?
- Are there extra indexes? Or lack of indexes?
    - remove unused indexes (pg_stat_user_indexes & pg_index)
    - Use The Index, Luke (website)
    - hypothetical indexes (hypopg)

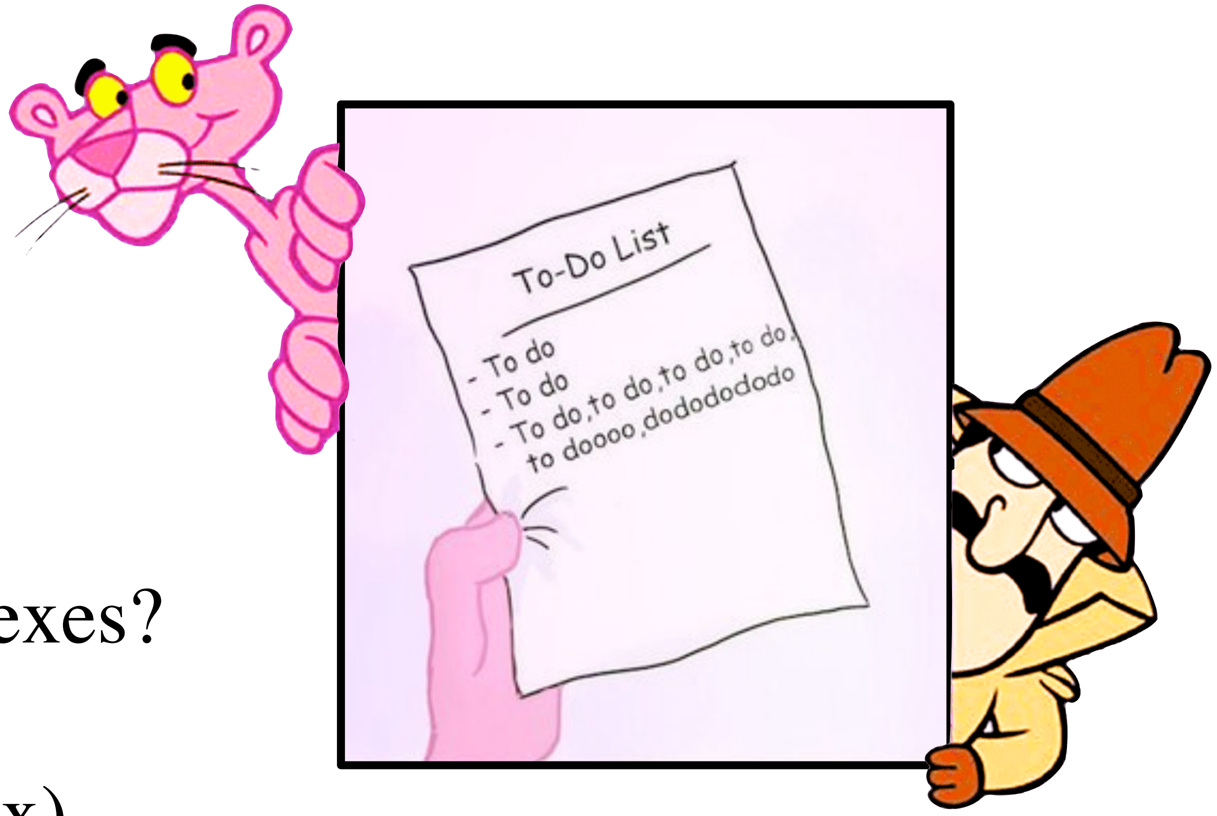


Indexes in PostgreSQL

Type of index	Performance	When to use
B-tree (default)	O(log(n))	Can be used for both equality and range queries
Hash	O(1)	Only works for equality comparisons
GiST (Generalized Search Tree)	O(log(n))	Can be used with geometric data types for equality and range comparisons
SP-GiST (Space-partitioned DiST)	O(log(n))	For insertion and queries Non-balanced, disk-based data structures
GIN (Generalized Inverted Indexes)	O(log(n))	Indexing data types that map multiple values to one row (i.e. arrays and full text search)
RUM	O(log(n))	Like GIN but it allows additional information to be stored in the index
BRIN (Block Range Index)	20x faster than B-tree	99%+ space savings. Table entries have to be ordered in the same format as the data on disk
Bloom	O(n)	Sufficiently "wide" tables, queries can use filtering by any of the fields, with false positives

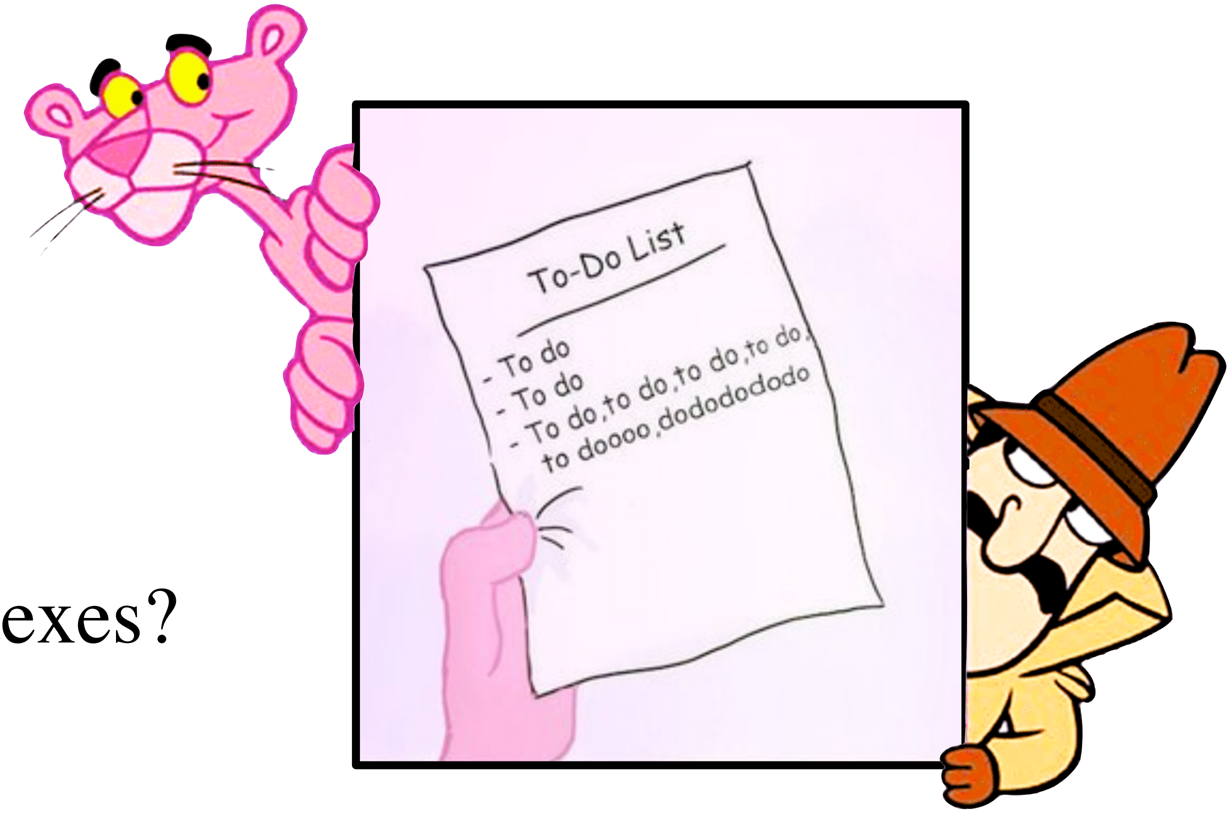
Checklist of questions

- Do we really need this query?
- Do we need it at a particular time?
- Was there anything blocking it?
- Is there type match everywhere?
- Are there extra indexes? Or lack of indexes?
  - remove unused indexes (pg_stat_user_indexes & pg_index)
  - Use The Index, Luke (website)
  - hypothetical indexes (hypopg)

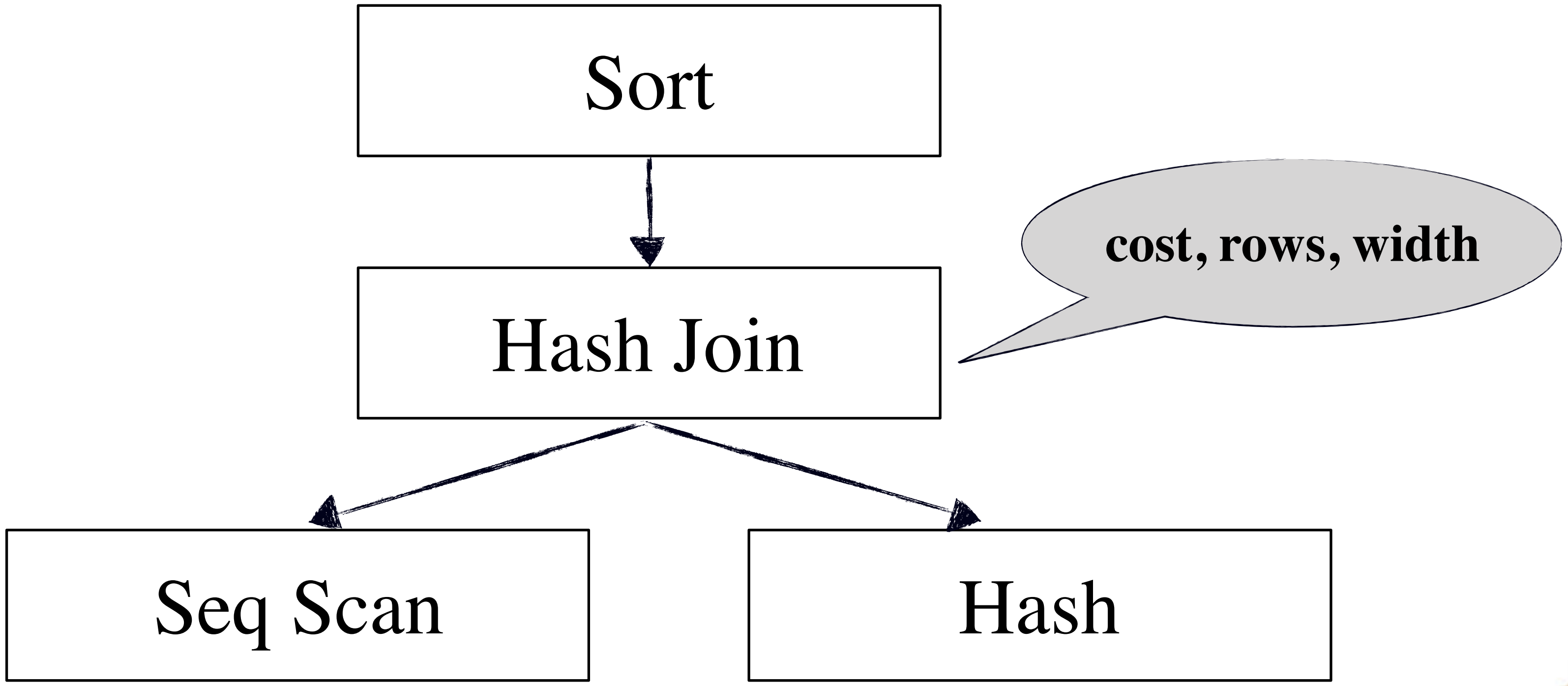


Checklist of questions

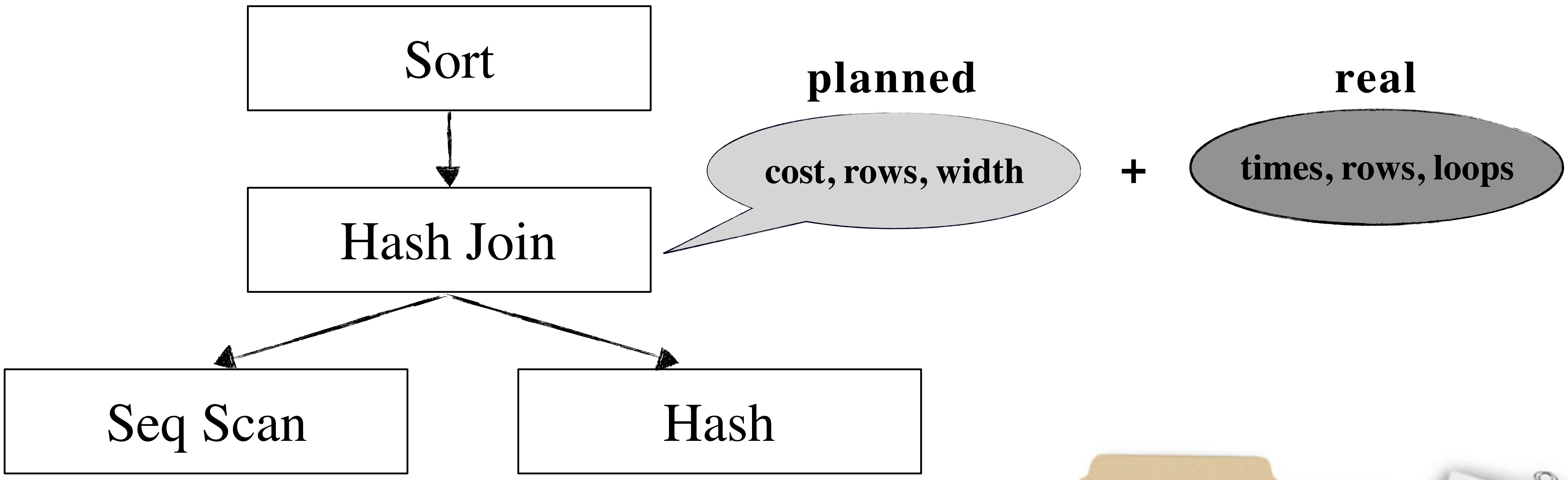
- ✦ Do we really need this query?
- ✦ Do we need it at a particular time?
- ✦ Was there anything blocking it?
- ✦ Is there type match everywhere?
- ✦ Are there extra indexes? Or lack of indexes?
- ✦ Was it executed optimally?
- ✦ Was there a logical error in the query?



EXPLAIN



EXPLAIN ANALYZE



auto_explain

Module provides a means for logging execution plans of slow statements automatically

auto_explain.**log_min_duration** (integer)

auto_explain.**log_analyze** (boolean)

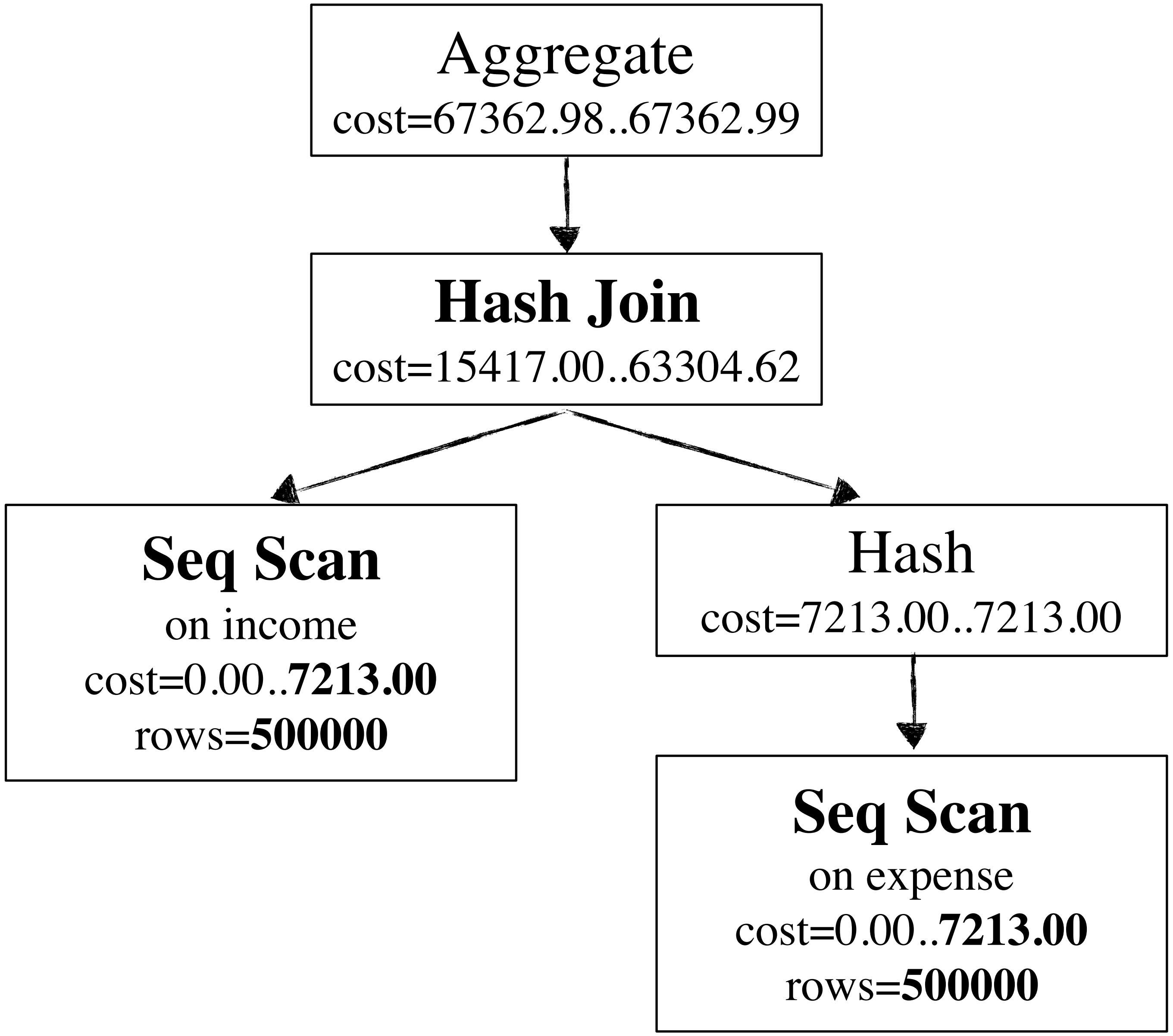
```
EXPLAIN SELECT max(income.value - expense.value)
FROM income FULL JOIN expense ON income.value = expense.value
WHERE expense.value < income.value;
```

```
Aggregate (cost=67362.98..67362.99 rows=1 width=4)
  ->  Hash Join (cost=15417.00..63304.62 rows=811670 width=8)
        Hash Cond: (income.value = expense.value)
        Join Filter: (expense.value < income.value)
        ->  Seq Scan on income (cost=0.00..7213.00 rows=500000 width=4)
        ->  Hash (cost=7213.00..7213.00 rows=500000 width=4)
              ->  Seq Scan on expense (cost=0.00..7213.00 rows=500000 width=4)
```

EXPLAIN ANALYZE SELECT max(income.value - expense.value)

FROM income FULL JOIN expense ON income.value = expense.value

WHERE expense.value < income.value;



Aggregate
cost=67362.98..67362.99

Hash Join
cost=15417.00..63304.62

Seq Scan
on income
cost=0.00..**7213.00**
rows=**500000**

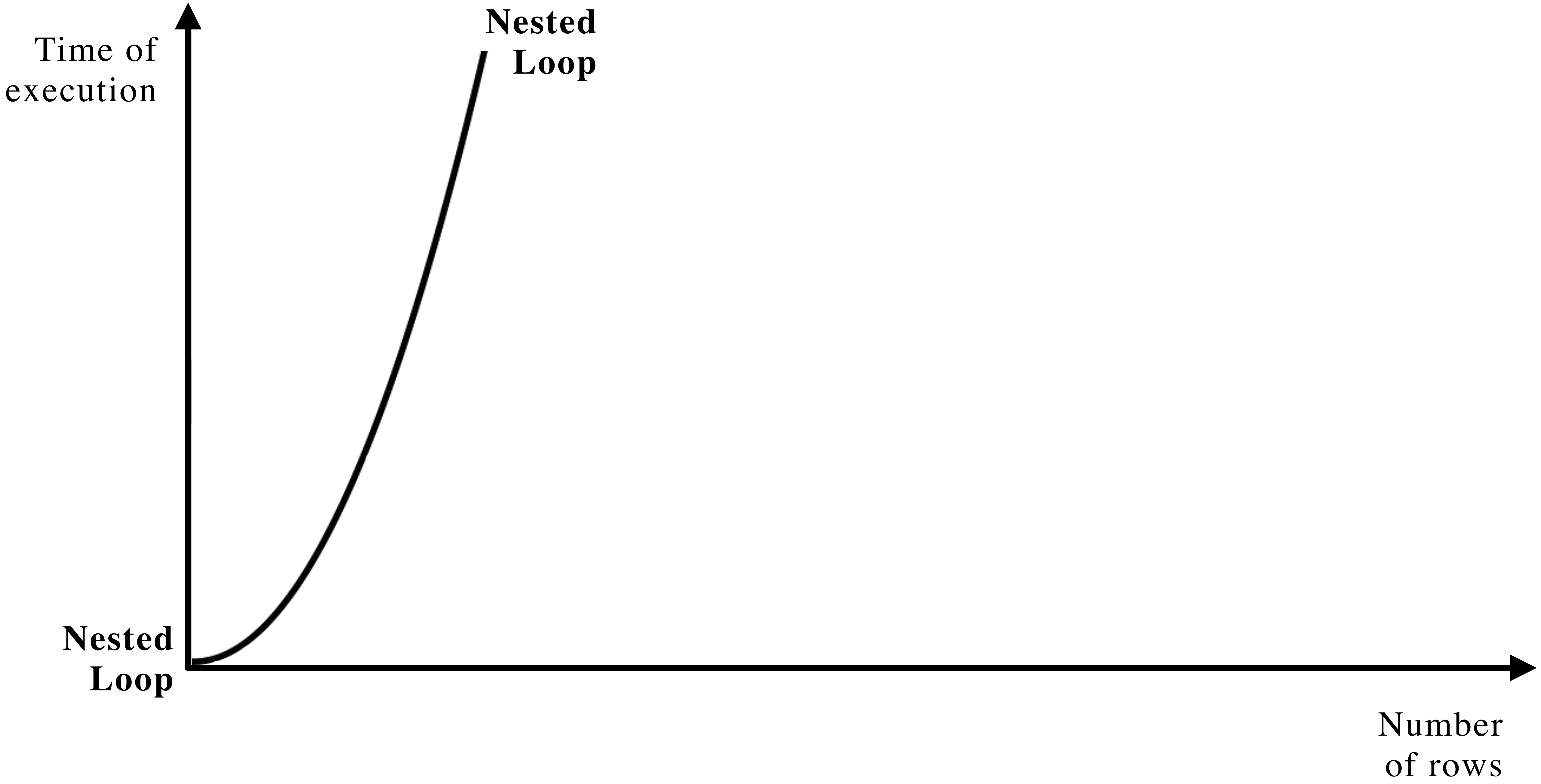
Hash
cost=7213.00..7213.00

Seq Scan
on expense
cost=0.00..**7213.00**
rows=**500000**

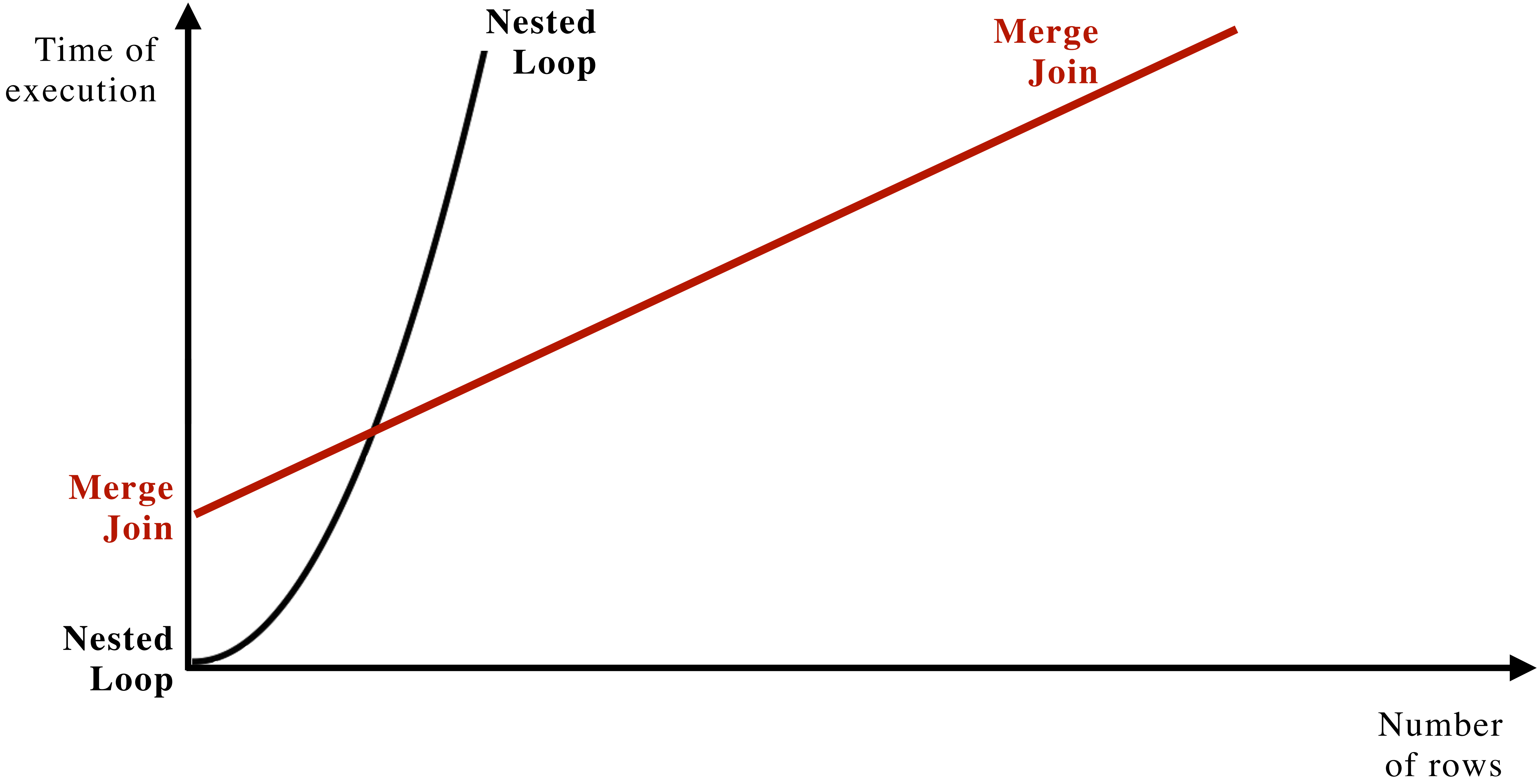
Planning Time: 0.137 ms

Execution Time: 2033.951 ms

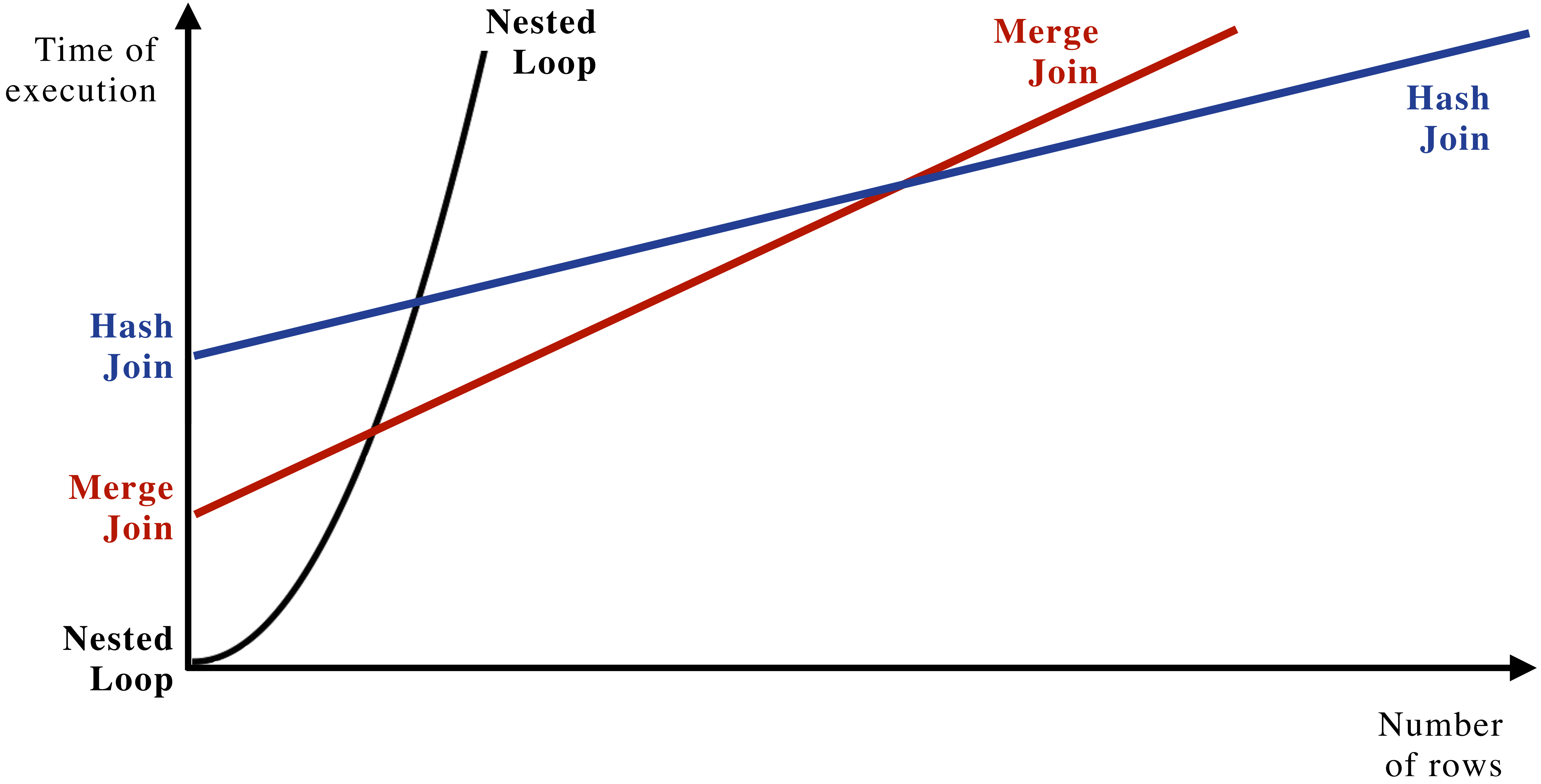
How was the plan chosen?



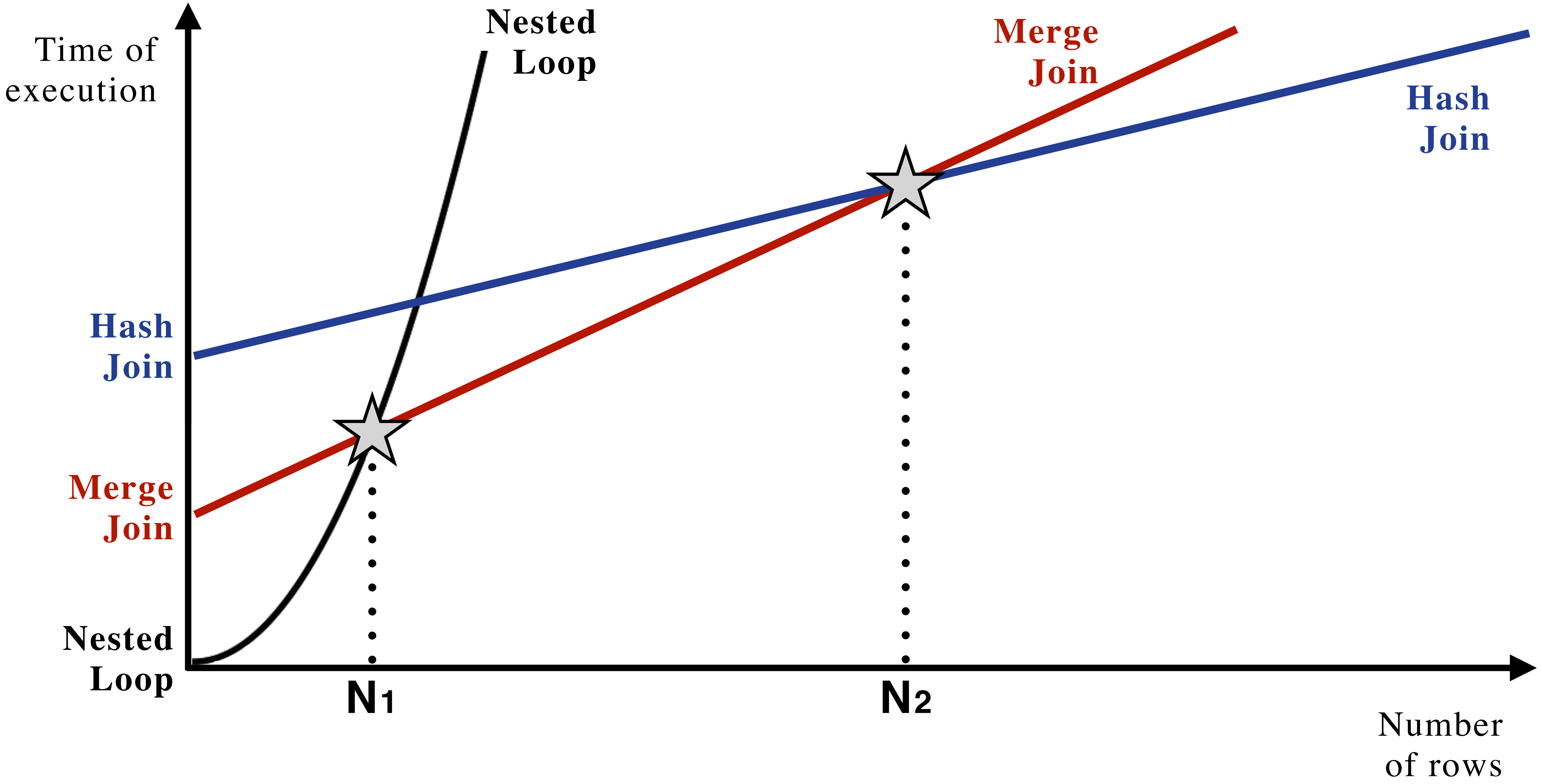
How was the plan chosen?



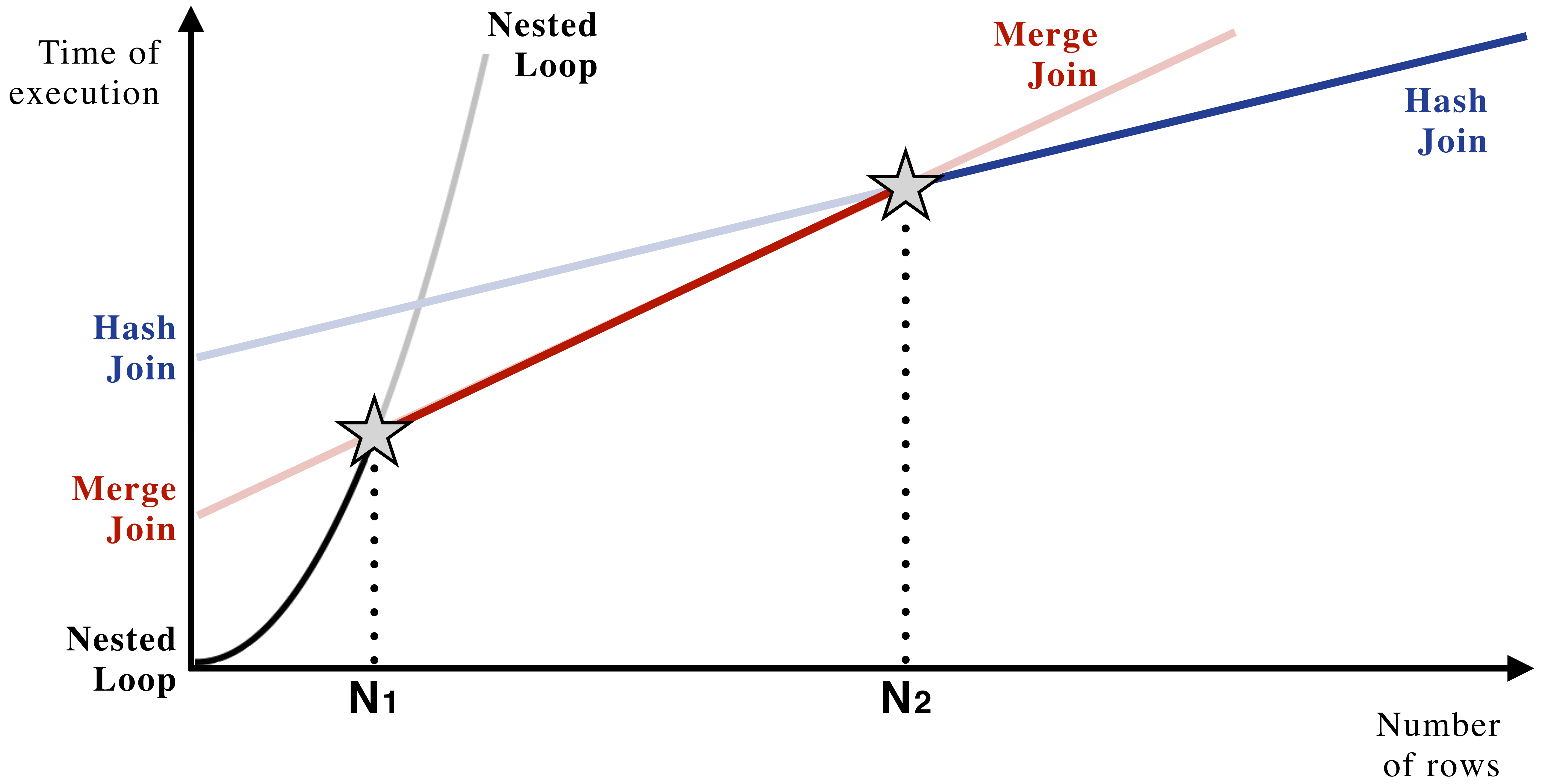
How was the plan chosen?



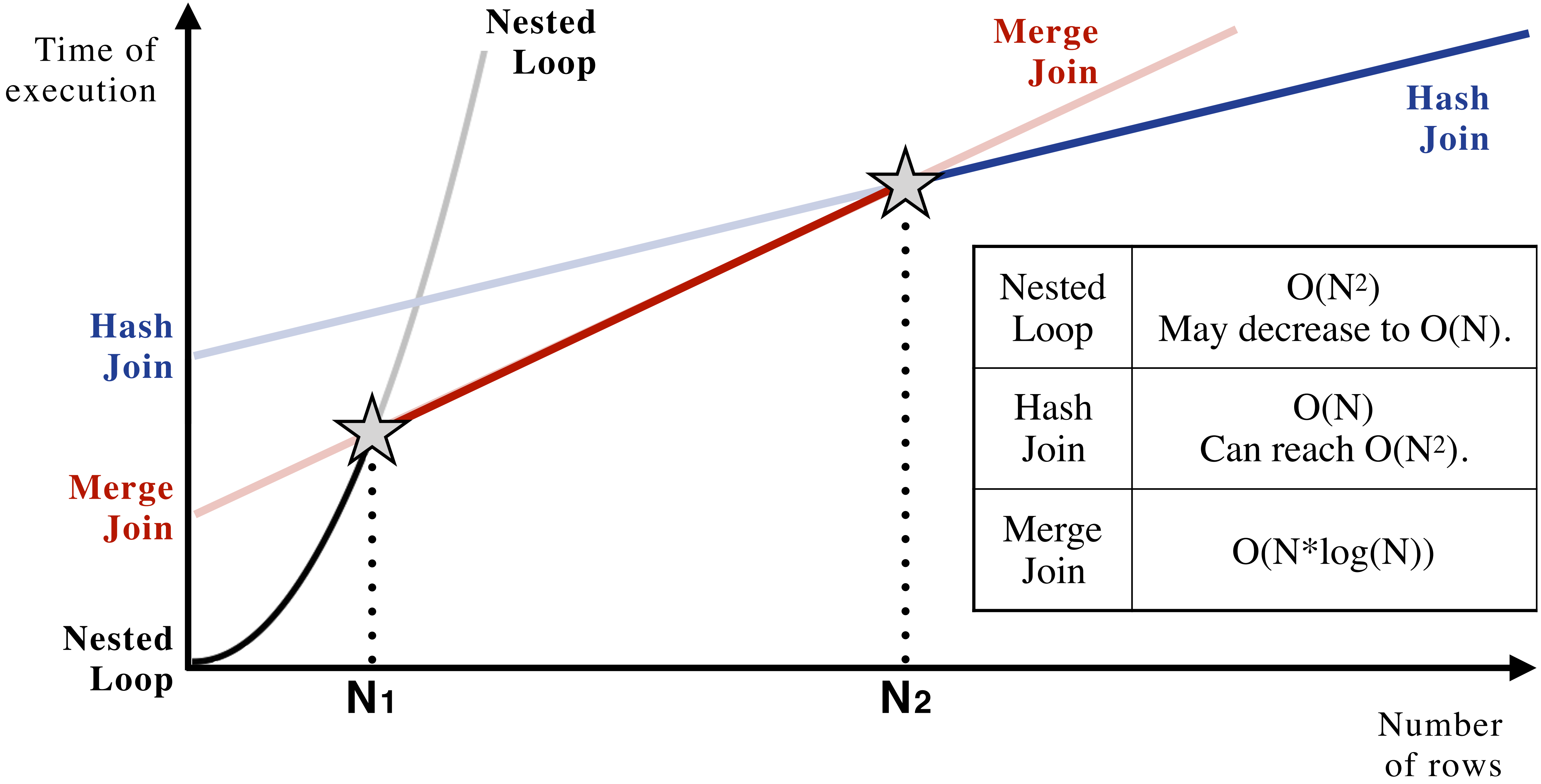
How was the plan chosen?



How was the plan chosen?

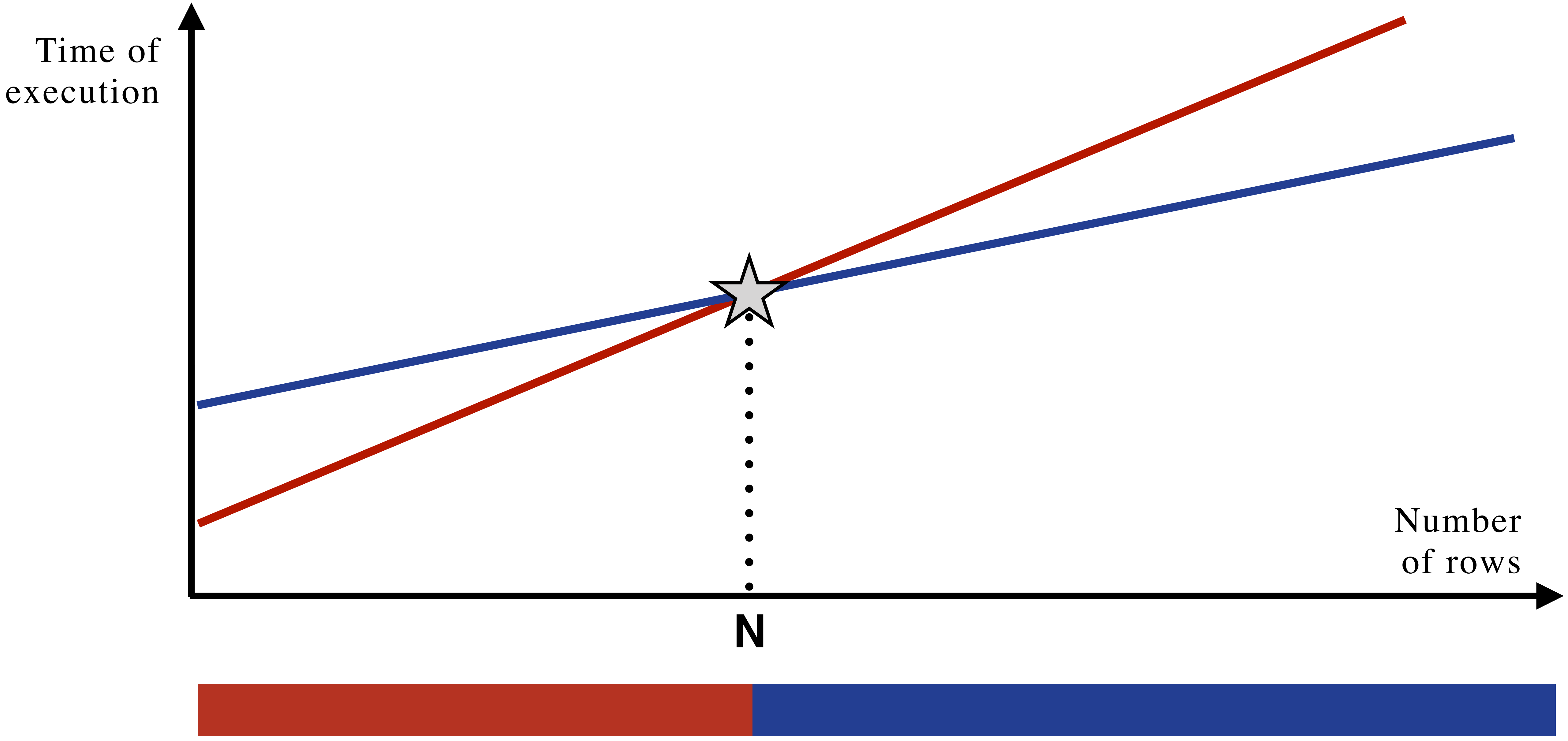


How was the plan chosen?

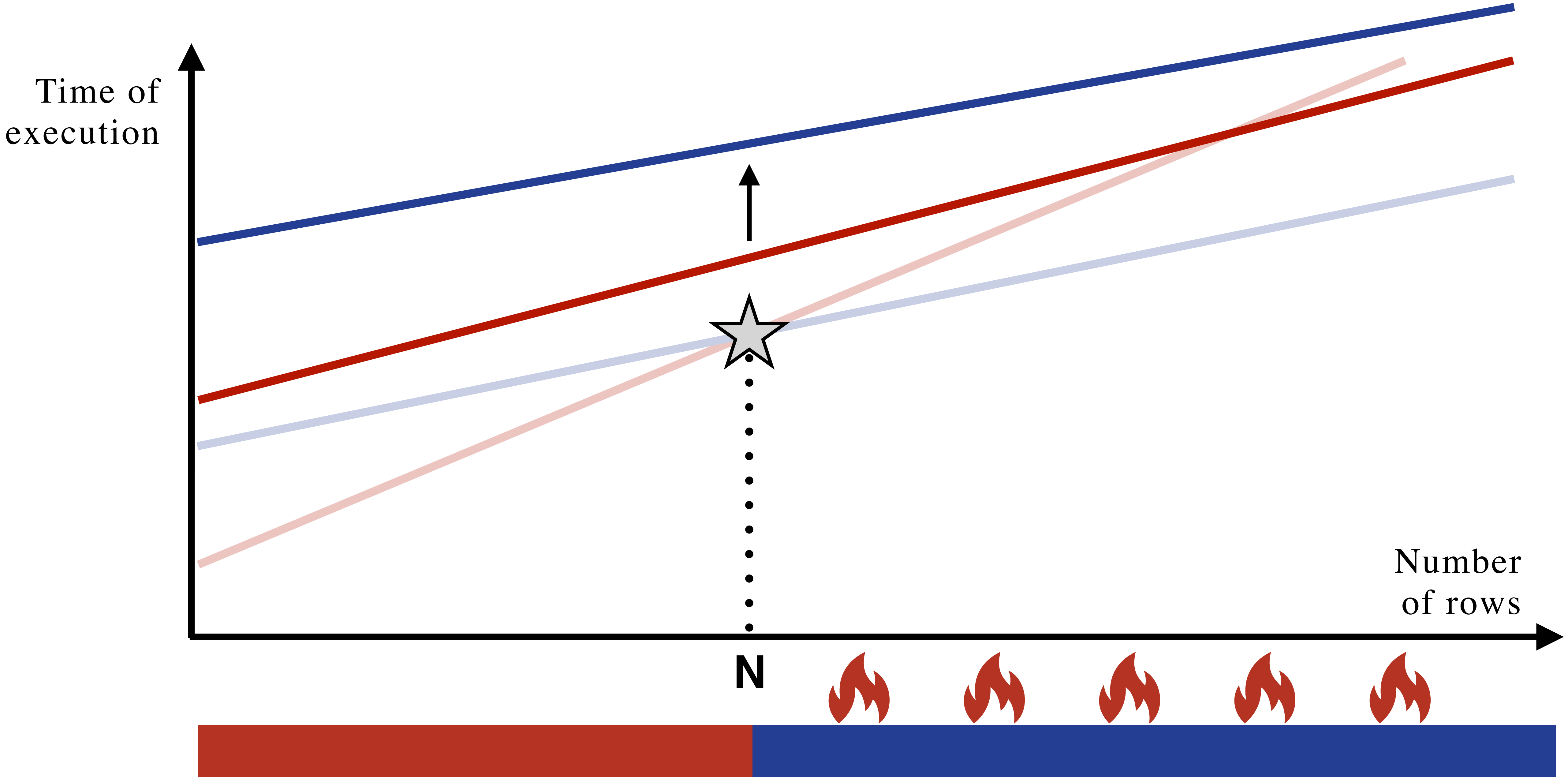


Nested Loop	$O(N^2)$ May decrease to $O(N)$.
Hash Join	$O(N)$ Can reach $O(N^2)$.
Merge Join	$O(N*\log(N))$

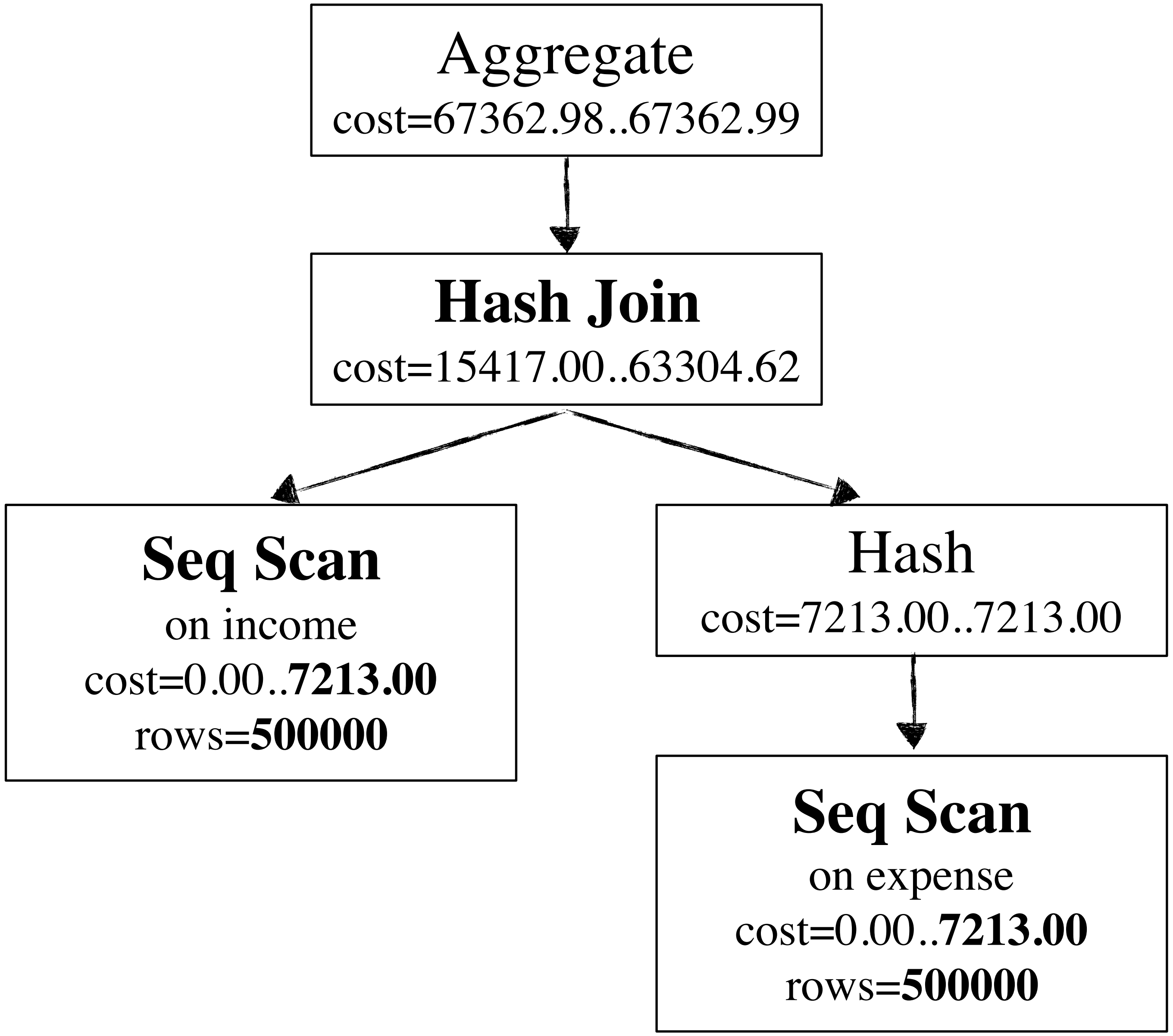
Expectation



Reality



EXPLAIN ANALYZE SELECT max(income.value - expense.value)
FROM income FULL JOIN expense ON income.value = expense.value
WHERE expense.value < income.value;



Aggregate
cost=67362.98..67362.99

Hash Join
cost=15417.00..63304.62

Seq Scan
on income
cost=0.00..**7213.00**
rows=**500000**

Hash
cost=7213.00..7213.00

Seq Scan
on expense
cost=0.00..**7213.00**
rows=**500000**

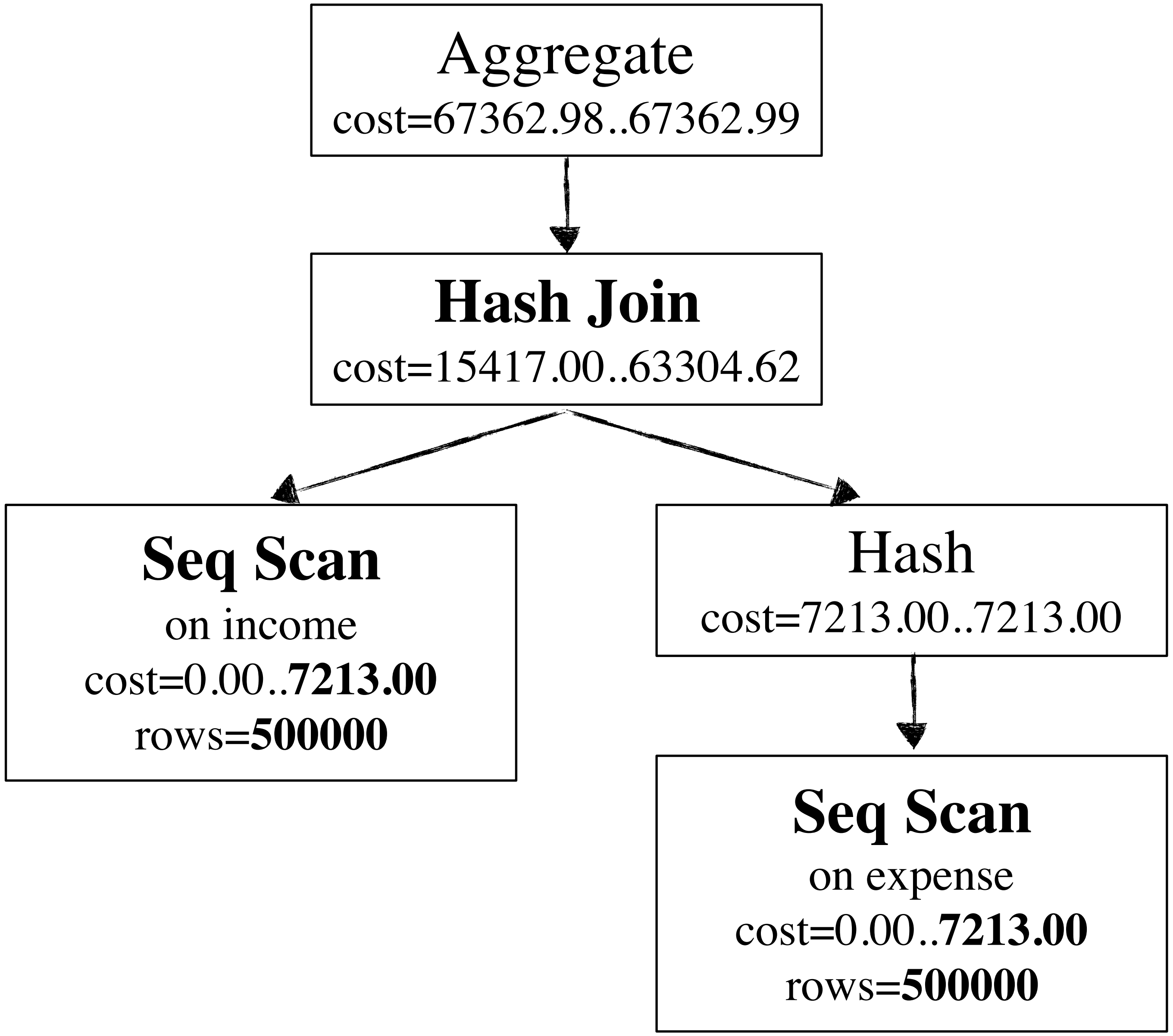
Planning Time: 0.137 ms

Execution Time: 2033.951 ms

EXPLAIN ANALYZE SELECT max(income.value - expense.value)
FROM income FULL JOIN expense ON income.value = expense.value
WHERE expense.value < income.value;

SET enable_seqscan TO OFF;

cost += 10.000.000.000



Aggregate
cost=67362.98..67362.99

Hash Join
cost=15417.00..63304.62

Seq Scan
on income
cost=0.00..**7213.00**
rows=**500000**

Hash
cost=7213.00..7213.00

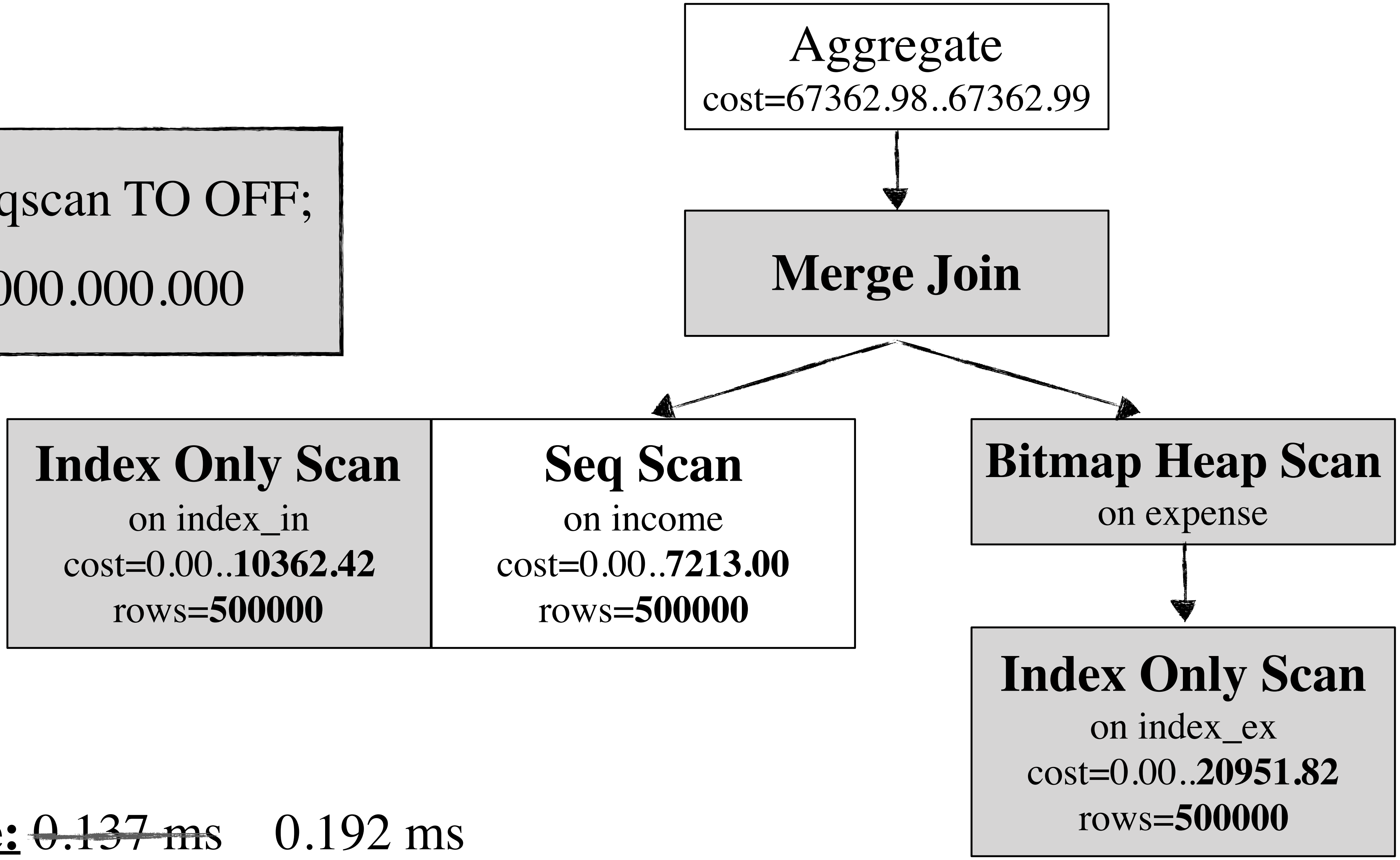
Seq Scan
on expense
cost=0.00..**7213.00**
rows=**500000**

Planning Time: 0.137 ms
Execution Time: 2033.951 ms

EXPLAIN ANALYZE SELECT max(income.value - expense.value)
FROM income FULL JOIN expense ON income.value = expense.value
WHERE expense.value < income.value;

SET enable_seqscan TO OFF;

cost += 10.000.000.000



Planning Time: ~~0.137 ms~~ 0.192 ms

Execution Time: ~~2033.951 ms~~ **784.764 ms**

```
EXPLAIN ANALYZE SELECT max(income.value - expense.value)
FROM income FULL JOIN expense ON income.value = expense.value
WHERE expense.value < income.value;
```

```
SET enable_seqscan TO OFF;

cost += 10.000.000.000
```

Let's find out who ruined the performance

pg_hint_plan

Module allows a user to control an execution plan.

It use hinting phrases mentioned in comments of a special form inside the SQL-query.

pgpro_multiplan

Module allows the user to save query execution plans, thereby avoiding repeated optimization of identical queries.

Module allows the user to save query execution plans, thereby avoiding repeated optimization of identical queries.

pgpro_multiplan

aqo

Postgres Pro Enterprise extension for cost-based query optimization. Using machine learning methods, aqo improves cardinality estimation, which can optimize execution plans and, consequently, speed up query execution.

aqe

Adaptive query execution enables reoptimizing a query, if during the execution some trigger indicates that it is non-optimal, so a more optimal plan should be looked for.

pgpro_multiplan

Module allows the user to save query execution plans, thereby avoiding repeated optimization of identical queries.

aqo

Postgres Pro Enterprise extension for cost-based query optimization. Using machine learning methods, aqo improves cardinality estimation, which can optimize execution plans and, consequently, speed up query execution.

✱ Alena Rybakina: Adaptive query optimization in PostgreSQL

aqe

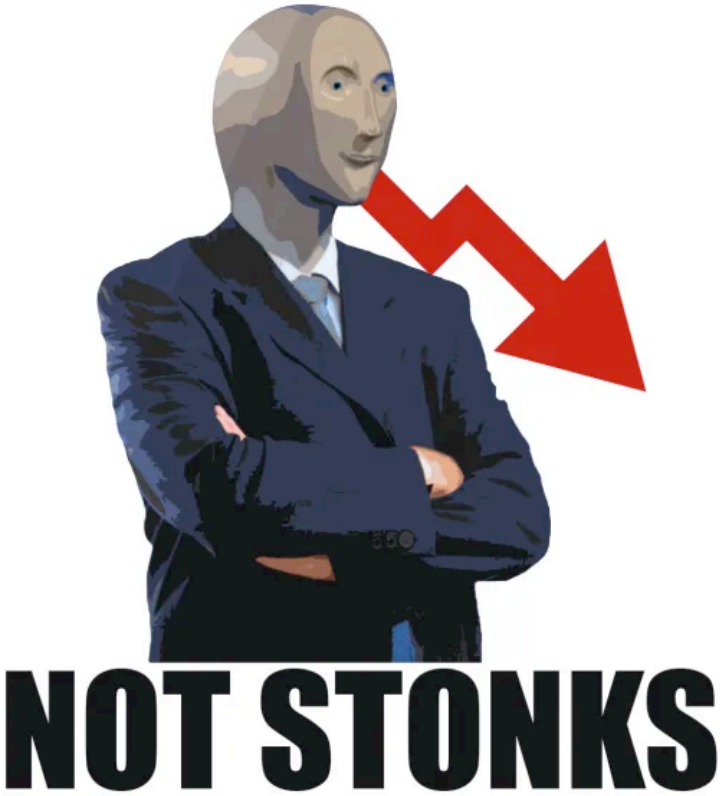
Adaptive query execution enables reoptimizing a query, if during the execution some trigger indicates that it is non-optimal, so a more optimal plan should be looked for.

A Beginner's Guide to Detectives

✓ searching for suspects

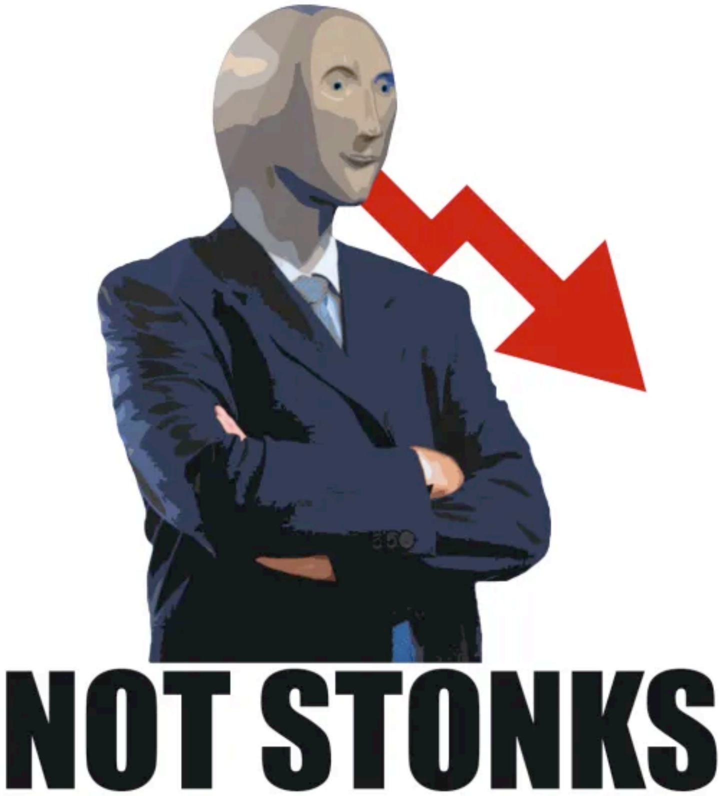
✓ interrogation of suspects: is it really suboptimal or not

✓ neutralize the culprits



HERE & NOW

- Is something running?
- Is something blocked?
- What is the progress of statement execution?



HERE & NOW

Is something running?

pg_stat_activity

```
SELECT pid, backend_type, state,
       query, wait_event_type,
       wait_event
FROM pg_stat_activity;
```

pid	3539
backend_type	autovacuum launcher
state	
query	
wait_event_type	Activity
wait_event	AutoVacuumMain
pid	3599
backend_type	**client backend**
state	**active**
query	select pid, backend_type, state, query, wait_event_type, wait_event from pg_stat_activity;
wait_event_type	
wait_event	

- Is something running? ✓
- Is something blocked?
- What is the progress of statement execution?

HERE
& NOW

Is something blocked?

1 **Tools built into the IDE**

2 **Special programs**

Is something blocked?

1 **Tools built into the IDE**

2 **Special programs**

3 **Well-written SQL-query**

```
SELECT * FROM pg_locks
LEFT JOIN pg_stat_activity
ON pg_locks.pid = pg_stat_activity.pid;
```

Is something blocked?

1 **Tools built into the IDE**

2 **Special programs**

3 **Well-written SQL-query**

```
SELECT * FROM pg_locks
LEFT JOIN pg_stat_activity
ON pg_locks.pid = pg_stat_activity.pid;
```

blocked pid	blocked user	blocking pid	blocking user	blocked statement	current statement in blocking process

pg_locks monitoring

```
SELECT blocked_locks.pid     AS blocked_pid,
         blocked_activity.usename  AS blocked_user,
         blocking_locks.pid     AS blocking_pid,
         blocking_activity.usename AS blocking_user,
         blocked_activity.query    AS blocked_statement,
         blocking_activity.query   AS current_statement_in_blocking_process
   FROM  pg_catalog.pg_locks         blocked_locks
    JOIN pg_catalog.pg_stat_activity blocked_activity
        ON blocked_activity.pid = blocked_locks.pid
    JOIN pg_catalog.pg_locks         blocking_locks
        ON blocking_locks.locktype = blocked_locks.locktype
        AND blocking_locks.DATABASE IS NOT DISTINCT FROM blocked_locks.DATABASE
        AND blocking_locks.relation IS NOT DISTINCT FROM blocked_locks.relation
        AND blocking_locks.page IS NOT DISTINCT FROM blocked_locks.page
        AND blocking_locks.tuple IS NOT DISTINCT FROM blocked_locks.tuple
        AND blocking_locks.virtualxid IS NOT DISTINCT FROM blocked_locks.virtualxid
        AND blocking_locks.transactionid IS NOT DISTINCT FROM blocked_locks.transactionid
        AND blocking_locks.classid IS NOT DISTINCT FROM blocked_locks.classid
        AND blocking_locks.objid IS NOT DISTINCT FROM blocked_locks.objid
        AND blocking_locks.objsubid IS NOT DISTINCT FROM blocked_locks.objsubid
        AND blocking_locks.pid != blocked_locks.pid
    JOIN pg_catalog.pg_stat_activity blocking_activity ON blocking_activity.pid = blocking_locks.pid
   WHERE NOT blocked_locks.GRANTED;
```

- Is something running? ✓
- Is something blocked? ✓
- What is the progress of statement execution?

HERE
& NOW

How is the system command doing?

pg_stat_progress_*

- ANALYZE
- CREATE INDEX
- VACUUM
- CLUSTER
- Base Backup
- COPY

— Pid and command text

— Ratio

— Phase

— Number of blocks/rows/bytes already processed

How is the system command doing?

pg_stat_progress_*

```
select * from pg_stat_progress_vacuum;
```

pid	5190	5190
datid	13263	13263
datname	postgres	postgres
relid	16384	16387
phase	vacuuming indexes	vacuuming indexes
heap_blks_total	12620	6638
heap_blks_scanned	12620	6638
heap_blks_vacuumed	0	0
index_vacuum_count	0	0
max_dead_tuples	3672420	1931658
num_dead_tuples	1499548	1500000

How is the user query doing?

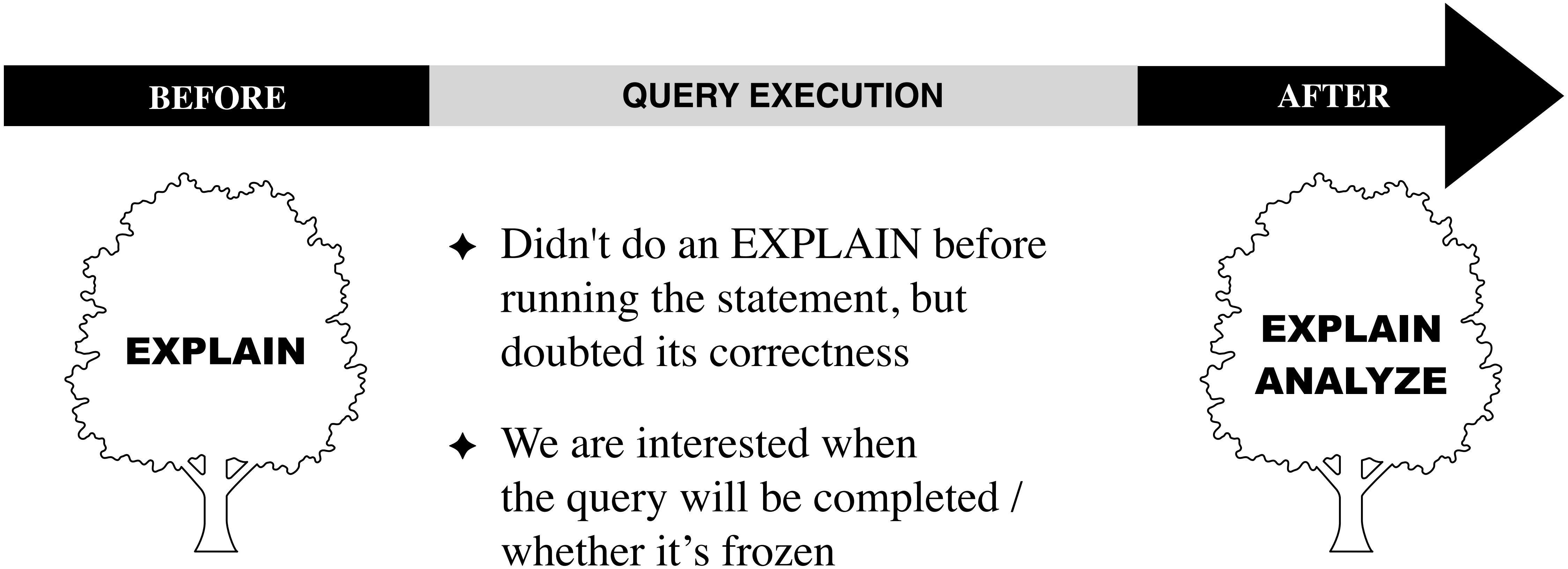
query is long-running

```
WITH RECURSIVE t(n) AS
   (VALUES (1)UNION ALL
        SELECT n+1 FROM t
        WHERE n < 100000000)
SELECT sum(n) FROM t;
```

```
SELECT my_table.*
FROM some_table,
        some_table AS my_table
GROUP BY my_table.c1;
```

… or wrongly written

How is the user query doing?



BEFORE

QUERY EXECUTION

AFTER

EXPLAIN

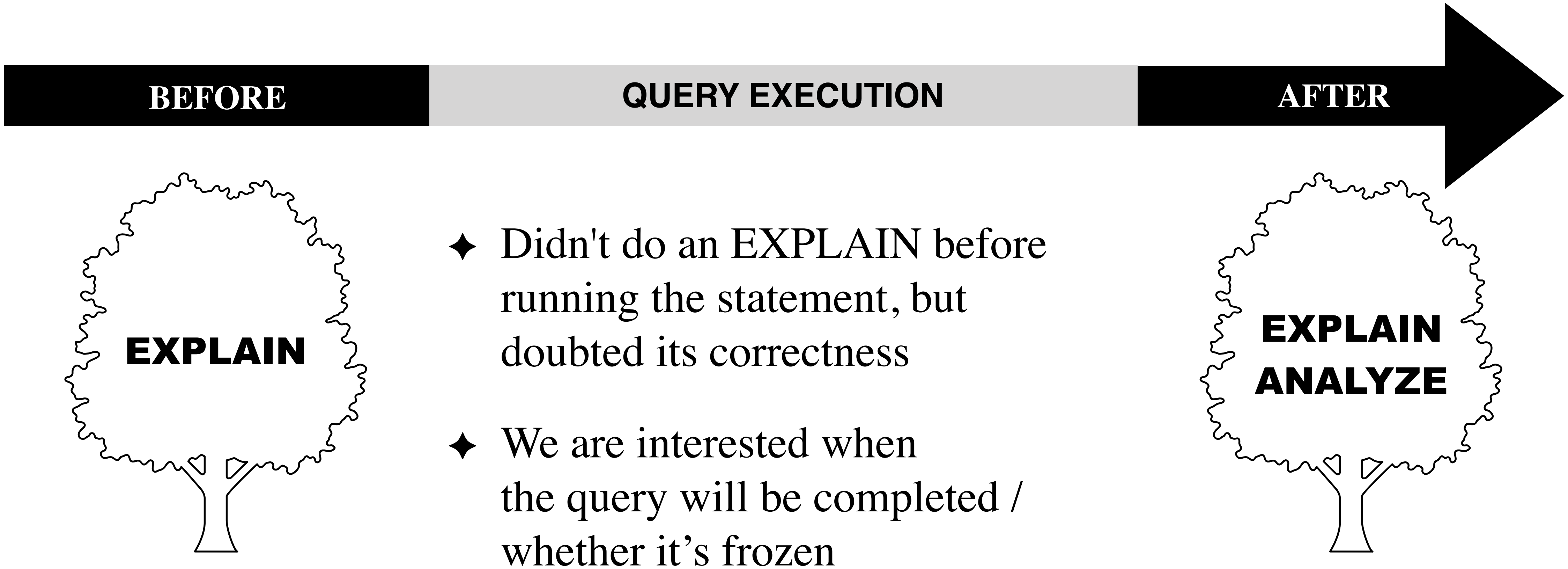
EXPLAIN ANALYZE

- ✦ Didn't do an EXPLAIN before running the statement, but doubted its correctness
- ✦ We are interested when the query will be completed / whether it’s frozen

How is the user query doing?

pg_query_state

runtime EXPLAIN ANALYZE



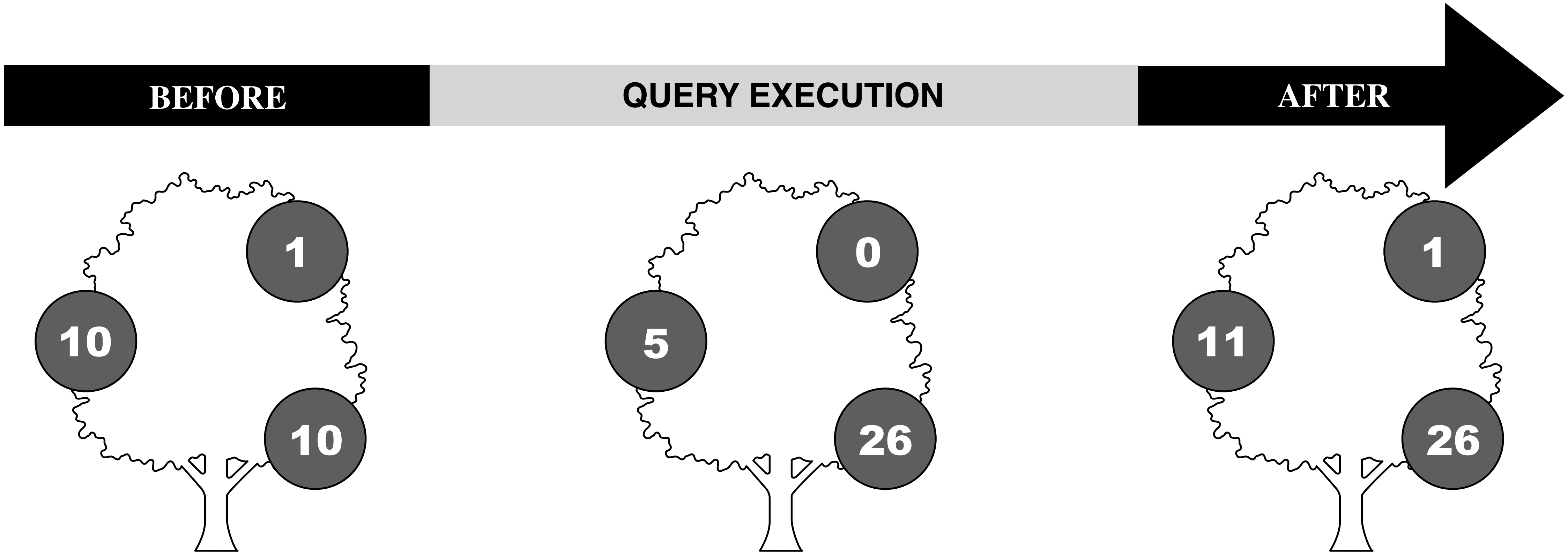
- ✦ Didn't do an EXPLAIN before running the statement, but doubted its correctness
- ✦ We are interested when the query will be completed / whether it’s frozen

✱ patch to PostgreSQL core is required

How is the user query doing?

pg_query_state

runtime EXPLAIN ANALYZE



BEFORE

QUERY EXECUTION

AFTER

EXPLAIN

EXPLAIN ANALYZE

How is the user query doing?

pg_query_state

runtime EXPLAIN ANALYZE

BEFORE

QUERY EXECUTION

AFTER

```
SELECT * FROM pg_query_state(4925);
```

pid	frame number	query_text	plan	leader pid
4925	0	insert into tab_a select generate_series(1,100000);	Insert on tab_a (**Current loop:** actual rows=0, loop number=1) -> ProjectSet (**Current loop: actual rows=35708**, loop number=1) -> Result (**Current loop:** actual rows=1, loop number=1)	

How is the user query doing?

pg_query_state runtime EXPLAIN ANALYZE

✦ Function argument: **pid** of the server process

```
# SELECT pg_backend_pid();
            4925
# run the query
```

```
# SELECT * FROM
        pg_query_state(4925);
```

How is the user query doing?

pg_query_state

runtime EXPLAIN ANALYZE

- Function argument: **pid** of the server process

```
# INSERT INTO tab_a SELECT
  generate_series(1,10000000);
```

```
# SELECT pid FROM pg_stat_activity
  WHERE query LIKE
                      'insert into tab_a%';

4925

# SELECT * FROM
       pg_query_state(4925);
```

SELECT n_join_foo_bar();

pid	frame number	query_text	plan	leader pid
4925	0	SELECT n_join_foo_bar	Result (Current loop: actual rows=0, loop number=1)	(null)
4925	1	SELECT (SELECT count(*) FROM foo JOIN bar ON foo.c1=bar.c1)	Result (Current loop: actual rows=0, loop number=1) InitPlan 1 (returns $0) -> Aggregate (Current loop: actual rows=0, loop number=1) -> Nested Loop (Current loop: actual rows=51, loop number=1) Join Filter: (foo.c1 = bar.c1) Rows Removed by Join Filter: 51636304 -> Seq Scan on bar (Current loop: actual rows=52, loop number=1) -> Materialize (actual rows=1000000 loops=51) (Current loop: actual rows=636355, loop number=52) -> Seq Scan on foo (Current loop: actual rows=1000000, loop number=1)	(null)

SET max_parallel_workers_per_gather = 2;

SELECT count(*) FROM foo JOIN bar ON foo.c1 = bar.c1;

pid	query_text	plan	leader pid

4925	SELECT count(*)
	FROM foo
	JOIN bar ON
	foo.c1=bar.c1
		Finalize Aggregate (Current loop: actual rows=0, loop number=1)
		-> Gather (Current loop: actual rows=0, loop number=1)
		Workers Planned: 2
		Workers Launched: 2
		-> Partial Aggregate (Current loop: actual rows=0, loop number=1)
		. . .
			(null)

4932	<parallel query>	Partial Aggregate (Current loop: actual rows=0, loop number=1) -> Nested Loop (Current loop: actual rows=10, loop number=1) Join Filter: (foo.c1 = bar.c1) Rows Removed by Join Filter: 4896779 -> Parallel Seq Scan on foo (Current loop: actual rows=10, loop number=1) -> Seq Scan on bar (actual rows=500000 loops=9) (Current loop: actual rows=396789, loop number=10)	4925
4933	<parallel query>	. . .	4925

What is the progress of statement execution?

pg_query_state

runtime EXPLAIN ANALYZE

```
# SELECT * FROM
  pg_progress_bar(4925);

 progress_bar
 ------------------
    0.6087927
(1 row)
```

SELECT * FROM pg_progress_bar_visual(4925, 5);

```
Progress = 0.043510
Progress = 0.168168
Progress = 0.292632
Progress = 0.407450
Progress = 0.530559
Progress = 0.645778
Progress = 0.735760
Progress = 0.871077
Progress = 0.995097
Progress = 1.000000
```

- **Is something running?** ✓
- **Is something blocked?** ✓
- **What is the progress of statement execution?** ✓

HERE
& NOW

Results of the investigation

Results of the investigation

- **Find the suspects:** candidate queries for improvement
- **Clear the crime scene:**
  get rid of redundant requests, move non-urgent ones

Results of the investigation

- **Find the suspects:** candidate queries for improvement
- **Clear the crime scene:**
  get rid of redundant requests, move non-urgent ones
- **Interrogation:** look at plans of heavy queries
- Check the conformity of types, create/delete indexes, configure parameters

Results of the investigation

- **Find the suspects:** candidate queries for improvement
- **Clear the crime scene:**
  get rid of redundant requests, move non-urgent ones
- **Interrogation:** look at plans of heavy queries
- Check the conformity of types, create/delete indexes, configure parameters
- **Take off the handcuffs:** checking blockages
- **Examination:** how far the query has been completed
- **Verdict:** continue execution or cancel

Thank you for your attention

Ekaterina Sokolova (aka Cate)
e.sokolova@postgrespro.ru