Diagnostic tools and query tuning examples in PostgreSQL

Peter Petrov,
Senior DBA,
June 17, 2021

postgrespro.com

Agenda (1)

- PostgreSQL workload monitoring tools.
- List of extensions for tracking resource-intensive queries.
- Detecting resource consuming queries by using the pg_profile module.
- Additional features provided by pgpro_stats and pgpro_pwr modules.
- Tuning a query with the GROUP BY clause.
- Data search optimization based on a list of values presented as a string.

Agenda (2)

- Usage of the LIMIT clause instead of the DISTINCT clause and window functions.
- Subqueries optimization.
- Statement optimization with filtering on a computed column.
- Query tuning with a complex calculated expression.
- Extended statistics usage for correcting rows estimates in a query plan.
- Extended statistics and IN operators.
- Excluding filtering conditions during query planning.

PostgreSQL workload monitoring tools (1)

Mamonsu is a monitoring agent for collecting PostgreSQL and system metrics and sending them to the Zabbix server:

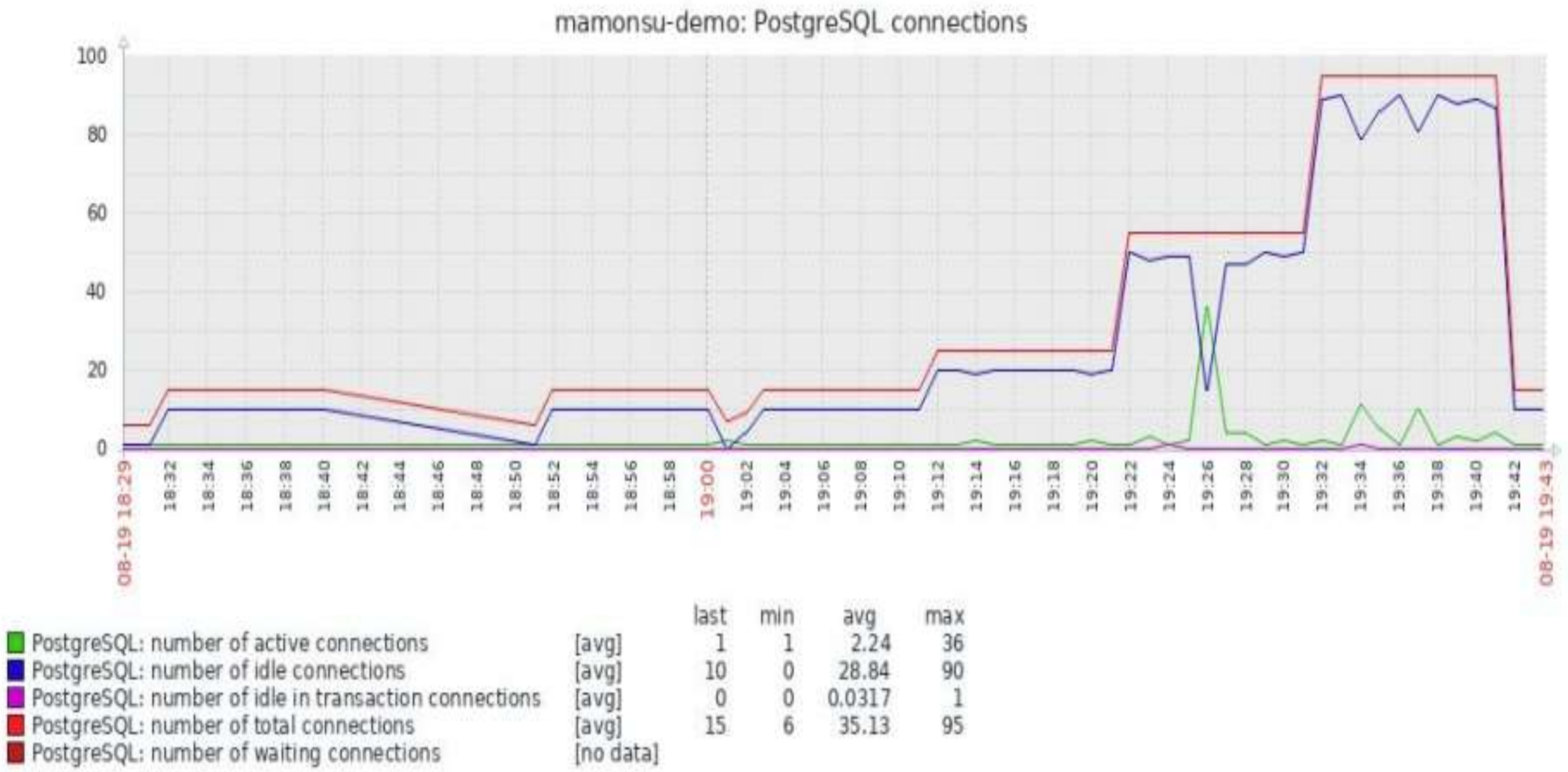
- Works with various operating systems / OSs
- 1 agent = 1 database instance
- Works with PostgreSQL version >= 9.5
- Provides various metrics related to PostgreSQL activity

PostgreSQL workload monitoring tools (2)

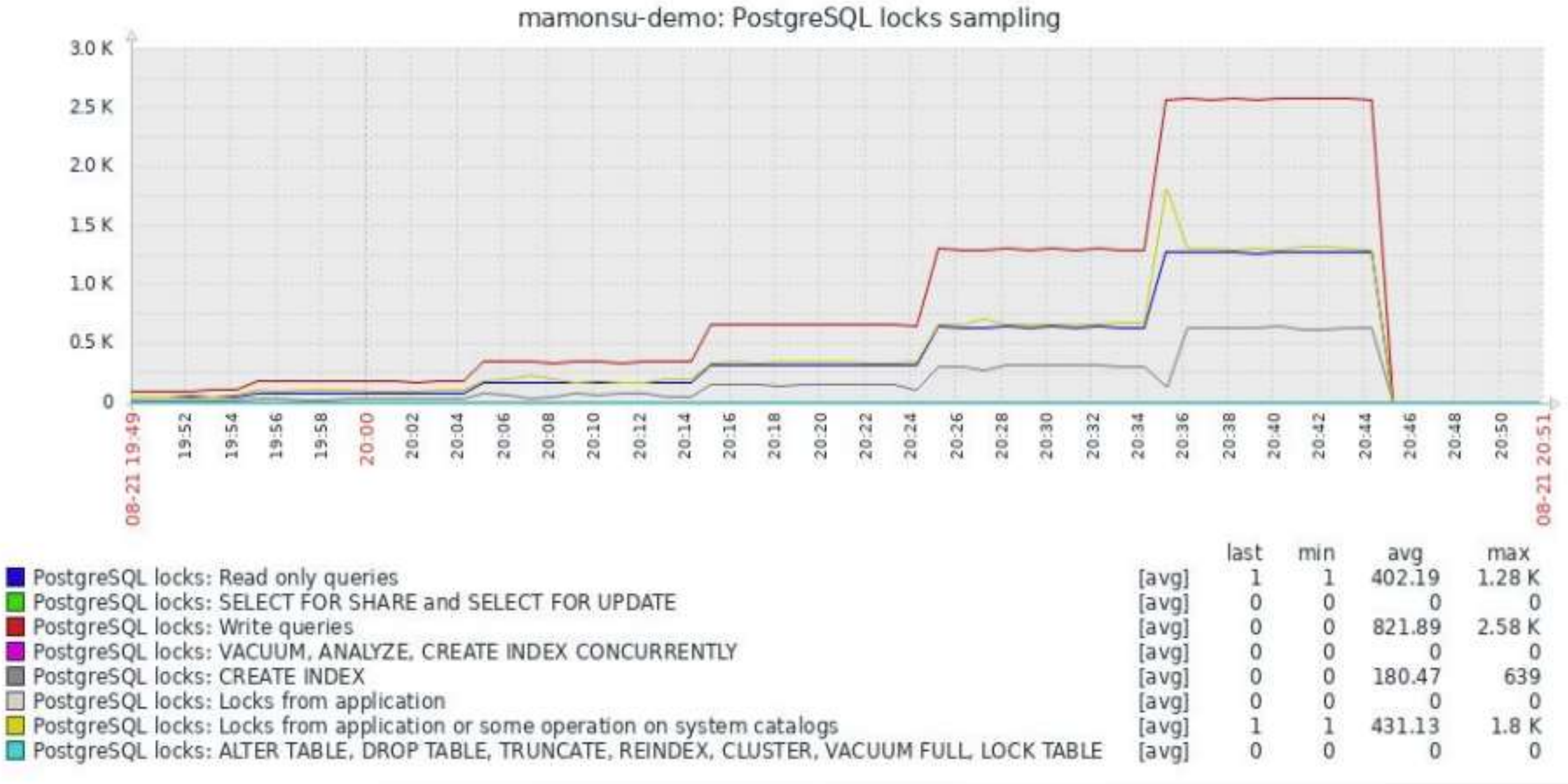
Zabbix Agent 2 is another tool for collecting various metrics which is available from Zabbix Server version 5.0:

- 1 agent can collect more than 95 metrics from multiple PostgreSQL instances.
- Available from Zabbix standard repository.
- Can work with PostgreSQL version 10 and higher.
- An opportunity to write custom plugins by using Golang.

PostgreSQL statistics connection



PostgreSQL locks sampling



List of extensions for tracking resource-intensive queries

pg_stat_statements for analyzing which queries have the longest execution time.

pg_stat_kcache for finding queries consuming the most CPU system and user time.

auto_explain for finding query plans and parameters for further tuning.

pg_wait_sampling for collecting the history of wait events and waits profiles.

plprofiler for creating performance profiles of PL/pgSQL functions and stored procedures.

Detecting resource consuming queries by using the pg_profile module

pg_profile can be used for creating historic workload repository containing various metrics such as:

- SQL Query statistics
- DML statistics
- Schema object statistics
- Vacuum-related statistics

Top SQL by execution time collected by the pg_profile module

Query ID	Database	Exec (s)	%Total	I/O time (s)		Rows	Execution times (ms)				Executions
				Read	Write		Mean	Min	Max	StdErr	
04ccad2749 [f287fe6aee8206ff]	data_db	568435.73	50.16	46357.14	0.00	56251	10105.344	5853.911	52907.023	3149.484	56251
16844de544 [99e0a6c6d0250ae7]	data_db	275047.09	24.27	0.46		56266	4888.336	4296.680	7265.277	458.831	56266
d2972b4cd4 [9e2fc21c4af82b3]	data_db	98871.86	8.73			55416	1784.175	1322.458	7172.350	585.637	55416
20d4a6bbe1 [ae253f2950531042]	data_db	60491.75	5.34			38757	1530.352	1329.667	4263.860	143.886	39528
680db037fd [3c0842cdf1cd90ec]	data_db	56346.51	4.97	0.18		11504	4897.993	4302.192	7420.182	461.443	11504
ae4d21e89c [4d144b46c513d3ba]	data_db	7312.75	0.65	673.43		796	9186.868	6154.602	17678.778	2227.804	796
ae4d21e89c [66ac855538c3d306]	data_db	7217.26	0.64	663.60		790	9135.771	6168.327	16583.076	2273.547	790
a722875b7d [15ec70c4c8f53ad1]	data_db	6026.51	0.53	0.02		1210	4980.590	4343.231	7167.028	514.164	1210

Top SQL by shared blocks fetched by the pg_profile module

Query ID	Database	blks fetched	%Total	Hits(%)	Elapsed(s)	Rows	Executions
04ccad2749 [f287fe6aee8206ff]	data_db	87023201012	39.99	85.16	568435.7	56251	56251
16844de544 [99e0a6c6d0250ae7]	data_db	40645632675	18.68	100.00	275047.1	56266	56266
d2972b4cd4 [9e2fc21c4af82b3]	data_db	40029581352	18.40	100.00	98871.9	55416	55416
20d4a6bbe1 [ae253f2950531042]	data_db	28552001009	13.12	100.00	60491.7	38757	39528
680db037fd [3c0842cdf1cd90ec]	data_db	8310018767	3.82	100.00	56346.5	11504	11504
8314e8a8a3 [76f909ceca8a56f9]	data_db	1737509525	0.80	75.72	3334.5	1123	1123
ae4d21e89c [4d144b46c513d3ba]	data_db	1231564389	0.57	84.13	7312.7	796	796
ae4d21e89c [66ac855538c3d306]	data_db	1222288398	0.56	84.23	7217.3	790	790

Top SQL by I/O waiting time collected by the pg_profile module

Query ID	Database	IO(s)	R(s)	W(s)	%Total	Reads			Writes			Elapsed(s)	Executions
						Shr	Loc	Tmp	Shr	Loc	Tmp		
04ccad2749 [f287fe6aee8206ff]	data_db	46357.139	46357.137	0.002	83.28	12911683448			24			568435.7	56251
f5500f7865 [1885229ba51e31d6]	data_db	1712.090	1712.013	0.077	3.08	84503980			5348			3338.5	1
8314e8a8a3 [76f909ceca8a56f9]	data_db	1665.391	1665.391		2.99	421918322						3334.5	1123
18b27b21fb [96449e23cb054092]	data_db	1264.005	1264.005		2.27	11953330						3158.0	181225
b448d1417b [265a69f6be6b326b]	data_db	1101.802	1101.802		1.98	6605417						1710.5	650058
8b90892d3f [76f909ceca8a56f9]	data_db	869.443	869.443		1.56	228886445						1278.4	473
ae4d21e89c [4d144b46c513d3ba]	data_db	673.434	673.434		1.21	195397547						7312.7	796
ae4d21e89c [66ac855538c3d306]	data_db	663.598	663.598		1.19	192704550						7217.3	790

Detecting resource-consuming UPDATE

Based on the information provided by the pg_profile module, one of the most resource-consuming queries has been found.

```
UPDATE contract.request_data
   SET status_code = $1
     , request_date = $2
     , response_date = $3
     , model_version = $4
     , contract_id = $12
 WHERE id = $13;
```

Why did it work so slow? We need the execution plan.

The execution plan for resource-consuming UPDATE statement

The execution plan and parameters have been received by using the auto_explain module. To find the required string, **Seq scan** access method was used which was the main reason of poor performance.

```
UPDATE ON request_data (cost=0.00..1824166.48 ROWS=91465
width=946)
-> Seq Scan ON request_data (cost=0.00..1824166.48
ROWS=91465 width=946)
     FILTER: ((id)::NUMERIC = '18310725'::NUMERIC)
```

On the application side, **BigDecimal** data type was used, the corresponding type in PostgreSQL is **numeric** which is not the same as **bigint**. After applying the changes, the query has begun to run for 20ms.

Additional features provided by pgpro_stats and pgpro_pwr modules

pgpro_stats is used as a combination of **pg_stat_statements**, **pg_stat_kcache** and **pg_wait_sampling** modules (only for Postgres Pro customers)

pgpro_pwr serves for gathering information from the **pgpro_stats** module (only for Postgres Pro customers)

These modules allow to get lock statistics and query execution plans and show them in separate sections of a **pgpro_pwr** report.

Wait statistics by database and top wait events

Wait statistics by database

Database	Wait event type	Waited (s)	%Total
db2	Client	207.30	90.49
db2	IO	21.76	9.50
db2	*	229.06	99.99
pg_profile	LWLock	0.03	0.01
pg_profile	*	0.03	0.01
Total		229.09	

Top wait events

Database	Wait event type	Wait event	Waited (s)	%Total
db2	Client	ClientWrite	207.30	90.49
db2	IO	BufFileWrite	15.30	6.68
db2	IO	BufFileRead	6.46	2.82
pg_profile	LWLock	BufferMapping	0.02	0.01
pg_profile	LWLock	LockManager	0.01	0.00

Wait event types

All wait event types					
Query ID	**Plan ID**	**Database**	**Waited (s)**	**%Total**	**Details**
15e052870b [d1937b74a857ee1b]	e4fa4a8c1185fed0	db2	105.11	45.88	Client: 105.11
fc9e858f09 [7a77ed62d2679a51]	3e3f4c16c408f623	db2	38.55	16.83	Client: 38.55
28397ca62a [480c7b3b4837b2c]	4c2069bda720277	db2	31.19	13.61	Client: 31.19
34464d840e [54d4e146036bdb4e]	111b80468c1a2489	db2	21.85	9.54	Client: 21.85
0babcd5300 [d281f4e1cf9ce40f]	e93e0d3c9a364d37	db2	20.62	9.00	IO: 20.62
bb70911186 [42126e815669f880]	93c68b8ab0931dee	pg_profile	0.03	0.01	LWLock: 0.03

IO wait event type					
Query ID	**Plan ID**	**Database**	**Waited (s)**	**%Total**	**Details**
0babcd5300 [d281f4e1cf9ce40f]	e93e0d3c9a364d37	db2	20.62	9.00	BufFileWrite: 14.58 BufFileRead: 6.04

LWLock wait event type					
Query ID	**Plan ID**	**Database**	**Waited (s)**	**%Total**	**Details**
bb70911186 [42126e815669f880]	93c68b8ab0931dee	pg_profile	0.03	0.01	BufferMapping: 0.02 LockManager: 0.01

Query execution plans

fc9e858f09	SELECT p.prod_id , p.category , p.title , p.actor , p.price , p.special , p.common_prod_id FROM products p WHERE p.title LIKE $1 AND p.price BETWEEN $2 AND $3 ORDER BY p.prod_id, p
3e3f4c16c408f623	Sort Output: prod_id, category, title, actor, price, special, common_prod_id Sort Key: p.prod_id, p.category DESC, p.title DESC, p.actor, p.price DESC, p.special, p.common_prod_id -> Bitmap Heap Scan on public.products p Output: prod_id, category, title, actor, price, special, common_prod_id Filter: ((p.title ~~ $1) AND (p.price >= $4) AND (p.price <= $5)) -> Bitmap Index Scan on prod_title_cat_prod_id Index Cond: ((p.title ~>=~ $6) AND (p.title ~<~ $7))
7c56f6b3ea	SELECT extname, extnamespace::regnamespace::name AS extnamespace, extversion FROM pg_catalog.pg_extension WHERE extname IN ($1,$2,$3)
18558adc1f4d44fa	Seq Scan on pg_catalog.pg_extension Output: extname, ((extnamespace)::regnamespace)::name, extversion Filter: (pg_extension.extname = ANY ($4))
f47e5c907c	select count($1) as pgpro_fxs from pg_catalog.pg_proc where proname IN ($2,$3,$4)
a0f5bbbbe9d25819	Aggregate Output: count($1) -> Index Only Scan using pg_proc_proname_args_nsp_index on pg_catalog.pg_proc Output: proname, proargtypes, pronamespace Index Cond: (pg_proc.proname = ANY ($5))

Tuning a query with a GROUP BY clause

In PostgreSQL, query execution time was 93 seconds, so it needed optimization.

To solve this problem, the query execution plan was required.

```
EXPLAIN (ANALYZE)
SELECT "d"."DOCUMENT_ID"
     , "gb"."A1"
     , "gb"."A2"
  FROM "dbo"."DOCUMENT" AS "d"
  LEFT JOIN (SELECT "dd"."DOCUMENT_ID"
                  , MIN("dd"."DATE_BEG") AS "A1"
                  , SUM("dd"."SUMMA") AS "A2"
               FROM "dbo"."DOCUMENT_DEBIT" "dd"
              WHERE "dd"."STORNO_STATE" = 1
              GROUP BY "dd"."DOCUMENT_ID"
            ) "gb"
     ON "gb"."DOCUMENT_ID" = "d"."DOCUMENT_ID"
  WHERE "d"."DEAL_ID" = 1259
  ORDER BY "d"."DATE_BEG"
```

The execution plan for the query with the GROUP BY clause

```
Sort (COST=3434984.74..3434985.16 ROWS=169 width=32) (actual TIME=92706.912..92706.923
ROWS=137 loops=1)
Sort KEY: ""Extent1"".""DATE_BEG"""
  Sort Method: quicksort Memory: 34kB
    -> Hash Right Join (cost=2546248.06..3434974.30 rows=169 width=32) (actual
time=57337.715..92706.636 rows=137 loops=1)
           Hash Cond: (""Extent20"".""DOCUMENT_ID"" = ""Extent1"".""DOCUMENT_ID"")
               -> HashAggregate (cost=2545629.74..3087954.23 rows=27437761 width=184)
(actual time=57336.820..90483.727 rows=27364695 loops=1)
                      Group Key: ""Extent20"".""DOCUMENT_ID"""
                      Planned Partitions: 256 Batches: 257 Memory Usage: 16401kB Disk
Usage: 1293840kB
                          -> Seq Scan ON ""DOCUMENT_DEBIT"" ""Extent20""
(COST=0.00..787898.18 ROWS=27437761 width=16) (actual TIME=0.014..38584.091
ROWS=27419629 loops=1)"
                                 Filter: (""STORNO_STATE"" = 1)" ROWS Removed BY
Filter: 117265"
               -> Hash (cost=616.21..616.21 rows=169 width=16) (actual
time=0.335..0.336 rows=137 loops=1)
                      Buckets: 1024 Batches: 1 Memory Usage: 15kB
                   -> Index Scan using ""IX_FK_DOCUMENT_DEAL"" on ""DOCUMENT""
""Extent1"" (cost=0.44..616.21 rows=169 width=16) (actual time=0.068..0.291 rows=137
loops=1)
                          Index Cond: (""DEAL_ID"" = 1259)
Planning Time: 0.502 ms
Execution Time: 92809.802 ms
```

A suggestion for the query optimization with the GROUP BY clause

After filtering on the field “DEAL_ID”, only 137 rows remained, so it was possible to reduce the main dataset and then calculate the aggregates for it. The execution time for this query has reduced to less than a second.

```
EXPLAIN (ANALYZE)
SELECT d."DOCUMENT_ID"
     , MIN(dd."DATE_BEG") AS "A1"
     , SUM(dd."SUMMA") AS "A2"
  FROM "dbo"."DOCUMENT" d
  LEFT "dbo"."DOCUMENT_DEBIT" dd
    ON dd."DOCUMENT_ID" = d."DOCUMENT_ID"
   AND dd."STORNO_STATE" = 1
 WHERE d."DEAL_ID" = 1259
 GROUP BY d."DOCUMENT_ID", d."DATE_BEG"
 ORDER BY d."DATE_BEG";
```

```
CREATE UNIQUE INDEX doc_deal_id_doc_id_date_beg_ux
    ON "dbo"."DOCUMENT"("DEAL_ID", "DOCUMENT_ID", "DATE_BEG");

CREATE INDEX doc_deb_doc_id_storno_state_ix
    ON "dbo"."DOCUMENT_DEBIT"("DOCUMENT_ID", "STORNO_STATE");

VACUUM (ANALYZE) "dbo"."DOCUMENT";

VACUUM (ANALYZE) "dbo"."DOCUMENT_DEBIT";
```

Data search optimization based on a list of values presented as a string

It is required to find records in which the "**status**" field matches at least one value from the list. In this case, it is presented as a string of values separated by commas.

```
EXPLAIN (ANALYZE)
WITH statuses AS (
SELECT v.status::BIGINT
  FROM regexp_split_to_table('10,30,20', ',') AS
v(STATUS)
)
SELECT id
  FROM req.lot l
 WHERE STATUS IN (SELECT STATUS FROM statuses s);
```

What will be the execution plan in this case?

The original query execution plan

At first, all rows from the lot table were extracted by using Seq Scan access method, then they were filtered by Hash Join method. The execution plan is presented below:

```
Hash JOIN (COST=17.00..29888.66 ROWS=140490 width=8) (actual TIME=41.228..649.965 ROWS=118
loops=1)
  Hash Cond: (lot.status = (v.status)::BIGINT)
    -> Seq Scan ON lot (COST=0.00..27184.79 ROWS=280979 width=16) (actual
TIME=0.058..397.005 ROWS=280979 loops=1)
      -> Hash (COST=14.50..14.50 ROWS=200 width=32) (actual TIME=0.109..0.111 ROWS=3
loops=1)
        Buckets: 1024 Batches: 1 Memory Usage: 9kB
        -> HashAggregate (COST=12.50..14.50 ROWS=200 width=32) (actual TIME=0.093..0.096
ROWS=3 loops=1)
             GROUP KEY: (v.status)::BIGINT
               -> FUNCTION Scan ON regexp_split_to_table v (COST=0.00..10.00 ROWS=1000
width=32) (actual TIME=0.082..0.084 ROWS=3 loops=1)
Planning TIME: 1.232 ms
Execution TIME: 650.235 ms (10 ROWS)
```

The execution plan for the query with the IN operator

```sql
EXPLAIN (ANALYZE)
SELECT l.id
  FROM req.lot l
 WHERE l.status IN (10, 20, 30);
```

The IN operator is equivalent to searching through a list of values. What will be the execution plan like in this case?

```
                                    QUERY PLAN
------------------------------------------------------------------------------
INDEX Scan USING ixf__lot__status__is_active ON lot l (COST=0.42..221.04
ROWS=112 width=8) (actual TIME=0.310..5.448 ROWS=118 loops=1)
    INDEX Cond: (status = ANY ('{10,20,30}'::BIGINT[]))
Planning TIME: 1.334 ms
Execution TIME: 5.554 ms (4 ROWS)
```

A query with regexp_split_to_array

We need a function, transforming a row into an array. For that PostgreSQL function **regexp_split_to_array** is required.

```
EXPLAIN (ANALYZE)
SELECT l.id
  FROM req.lot l
 WHERE l.status = ANY(regexp_split_to_array('10,30,20',
',')::BIGINT[]);
```

```
                              QUERY PLAN
----------------------------------------------------------------------
INDEX Scan USING ixf__lot__status__is_active ON lot l (COST=0.42..221.04
ROWS=112 width=8) (actual TIME=0.310..5.448 ROWS=118 loops=1)
    INDEX Cond: (status = ANY ('{10,20,30}'::BIGINT[]))
Planning TIME: 1.334 ms
Execution TIME: 5.554 ms (4 ROWS)
```

The original query execution time: 650.235 ms.

The current query execution time: 5.554 ms.

Usage of the LIMIT clause instead of the DISTINCT clause and window functions

In PostgreSQL query's execution time is 3.4 seconds, so optimization is required.

To solve this issue, we need to know the execution plan for this statement.

```
EXPLAIN (ANALYZE)
SELECT l.id
     , li_norm.*
  FROM req.lot l
  JOIN lateral (SELECT DISTINCT li.lot_id
                     , first_value(li.id) OVER (partition BY li.lot_id ORDER BY li.plan_price DESC) AS id
                  FROM req.lot_item li
                 WHERE li.is_active
                   AND li.lot_id = l.id
               ) li_norm
    ON li_norm.lot_id = l.id
 WHERE l.status = ANY(regexp_split_to_array('2', ',')::bigint[]);
```

The execution plan of the query with DISTINCT and window function

```
                                        QUERY PLAN
-----------------------------------------------------------------------------------
Nested Loop (cost=48.23..3639059.79 ROWS=6 width=24) (actual TIME=0.276..3345.057
ROWS=72128 loops=1)
  -> INDEX Scan USING pk__lot ON lot l (cost=0.42..45195.22 ROWS=74392 width=8) (actual
TIME=0.108..520.787 ROWS=74436 loops=1)
       FILTER: (STATUS = ANY ('{2}'::BIGINT[]))
       ROWS Removed BY FILTER: 206543
  -> Subquery Scan ON li_norm (cost=47.80..48.30 ROWS=1 width=16) (actual TIME=0.037..0.037
ROWS=1 loops=74436)
       FILTER: (l.id = li_norm.lot_id)
       -> HashAggregate (cost=47.80..48.02 ROWS=22 width=22) (actual TIME=0.036..0.036
ROWS=1 loops=74436)
          GROUP KEY: li.lot_id, first_value(li.id) OVER (?)
            -> WindowAgg (cost=47.25..47.69 ROWS=22 width=22) (actual TIME=0.029..0.033
ROWS=5 loops=74436)
              -> Sort (cost=47.25..47.31 ROWS=22 width=22) (actual TIME=0.026..0.027 ROWS=5
loops=74436)
                   Sort KEY: li.plan_price DESC
                   Sort Method: quicksort Memory: 25kB
                     -> INDEX Scan USING ixf__lot_item__lot_id__item_id__is_active ON
lot_item li (cost=0.38..46.21 ROWS=22 width=22) (actual TIME=0.007..0.021 ROWS=5 lo
ops=74436)
                          INDEX Cond: (lot_id = l.id)
Planning TIME: 0.960 ms
Execution TIME: 3355.723 ms
```

Replacing DISTINCT and window function with the LIMIT clause

For every lot object it is required to find out a corresponding row from the **lot_item** table with the maximum **plan_price**. Therefore, the query can be changed like this:

```
SELECT li.dic_direction_id
     , li.plan_year
     , li.item_id
  FROM req.lot_item li
 WHERE li.lot_id = l.id
   AND li.is_active
 ORDER BY li.plan_price DESC
 LIMIT 1
```

To find a row by using Index Only Scan, we need to create an index with the INCLUDE clause, where non-key fields will be stored. I.e, fields that are not used in filtering/sorting operations.

```
CREATE INDEX li_lot_id_plan_price_year_dic_direction_id_ix
    ON req.lot_item(lot_id, is_active, plan_price, id)
INCLUDE(plan_year, dic_direction_id, item_id);
```

The new query text after the DISTINCT and window function replacement

The new form of the query after DISTINCT and window function replacement is presented below:

```
EXPLAIN (ANALYZE)
SELECT l.id
     , li_norm.*
  FROM req.lot l
  JOIN LATERAL (SELECT li.dic_direction_id
                     , li.plan_year
                     , li.item_id
                  FROM req.lot_item li
                 WHERE li.lot_id = l.id
                   AND li.is_active
                 ORDER BY li.plan_price DESC
                 LIMIT 1
               ) li_norm
    ON (1 = 1)
 WHERE l.status = ANY(regexp_split_to_array('2', ',')::BIGINT[]);
```

The execution plan of the query with the LIMIT clause

```
                                        QUERY PLAN
-------------------------------------------------------------------------------
Nested LOOP (COST=707.39..67460.04 ROWS=74392 width=32) (actual TIME=9.644..474.265
ROWS=72128 loops=1)
  -> Bitmap Heap Scan ON lot l (COST=706.96..25918.87 ROWS=74392 width=8) (actual
TIME=9.577..80.787 ROWS=74436 loops=1)
       RECHECK Cond: (status = ANY ('{2}'::BIGINT[]))
       Heap Blocks: exact=16830
       -> Bitmap INDEX Scan ON ixf__lot__status__is_active (COST=0.00..688.36 ROWS=74392
width=0) (actual TIME=6.111..6.112 ROWS=74436 loops=1)
            INDEX Cond: (status = ANY ('{2}'::BIGINT[]))
  -> LIMIT (COST=0.43..0.54 ROWS=1 width=30) (actual TIME=0.005..0.005 ROWS=1 loops=74436)
    -> INDEX ONLY Scan BACKWARD USING li_lot_id_plan_price_year_dic_direction_id_ix ON
lot_item li (COST=0.43..2.87 ROWS=22 width=30) (actual TIME=0.004..0.004 ROWS=1 loo ps=74436)
         INDEX Cond: ((lot_id = l.id) AND (is_active = TRUE))
         Heap Fetches: 0
Planning TIME: 0.821 ms
Execution TIME: 479.703 ms
```

The original query execution time: 3355.723 ms.

The current query execution time: 479 ms

Subqueries optimization

It is required to get summary data for rows from the lot table. The original query version is presented below, its execution time was almost 4 minutes. The main reason was sequential scan on the **purchase_result** table while calculating values for the **pur_result** column.

```
EXPLAIN (ANALYZE)
SELECT l.id
     , (SELECT string_agg(doc_number, '; ')
          FROM buy.purchase_result
         WHERE lot_id = l.id
       ) AS pur_result
     , (SELECT COUNT(*)
          FROM buy.purchase_result pr
         WHERE pr.lot_id = l.id
           AND pr.is_active) AS pr_count
     , (SELECT string_agg(DISTINCT sup.name_full, ';')
          FROM buy.purchase_result pr
          JOIN req.supplier sup
            ON pr.supplier_id = sup.id
           AND sup.is_active
         WHERE pr.lot_id = l.id
           AND pr.is_active
       ) AS sup_info
  FROM req.lot l
 WHERE l.organization_id = 964;
```

The execution plan of the statement with multiple subqueries

```
                                               QUERY PLAN
-------------------------------------------------------------------------------------------
INDEX ONLY Scan USING lt_organization_id_ux ON lot l (cost=0.42..41848864.82 ROWS=7459 width=80) (actual
TIME=144.894..243809.902 ROWS=7495 loops=1)
  INDEX Cond: (organization_id = 964)
  Heap Fetches: 0
  SubPlan 1
    -> Aggregate (cost=5594.87..5594.88 ROWS=1 width=32) (actual TIME=32.387..32.388 ROWS=1 loops=7495)
      -> Seq Scan ON purchase_result (cost=0.00..5594.86 ROWS=2 width=6) (actual TIME=29.084..32.361
ROWS=0 loops=7495)
           FILTER: (lot_id = l.id) ROWS
           Removed BY FILTER: 147909
  SubPlan 2
    -> Aggregate (cost=2.41..2.42 ROWS=1 width=8) (actual TIME=0.044..0.044 ROWS=1 loops=7495)
      -> INDEX ONLY Scan USING ixf__purchase_result__lot_id ON purchase_result pr (cost=0.38..2.40 ROWS=2
width=0) (actual TIME=0.032..0.033 ROWS=0 loops=7495)
           INDEX Cond: (lot_id = l.id) Heap Fetches: 0
  SubPlan 3
    -> Aggregate (cost=13.19..13.20 ROWS=1 width=32) (actual TIME=0.064..0.064 ROWS=1 loops=7495)
      -> Nested Loop (cost=0.67..13.18 ROWS=1 width=97) (actual TIME=0.018..0.022 ROWS=0 loops=7495)
        -> INDEX Scan USING ixf__purchase_result__lot_id ON purchase_result pr_1 (cost=0.38..4.57 ROWS=2
width=8) (actual TIME=0.005..0.006 ROWS=0 loops=7495)
             INDEX Cond: (lot_id = l.id)
        -> INDEX Scan USING pk__supplier ON supplier sup (cost=0.29..4.31 ROWS=1 width=105) (actual
TIME=0.023..0.023 ROWS=1 loops=3419)
             INDEX Cond: (id = pr_1.supplier_id)
             FILTER: is_active
             ROWS Removed BY FILTER: 0
Planning TIME: 5.320 ms
Execution TIME: 243821.165 ms
```

Building one subquery using the LATERAL clause

To optimize this statement, it is required to write one query which will relate to the main dataset with the help of the LATERAL clause. We also need to build some additional indexes.

```
SELECT l.id
     , pr.doc_numbers
     , pr.pr_count
     , pr.sup_info
  FROM req.lot l
  LEFT JOIN LATERAL (SELECT string_agg(pr.doc_number, '; ') AS doc_numbers
                          , COUNT(*) FILTER(WHERE pr.is_active) AS pr_count
                          , string_agg(DISTINCT sup.name_full, ';') FILTER(WHERE pr.is_active) AS sup_info
                       FROM buy.purchase_result pr
                       LEFT JOIN req.supplier sup
                         ON sup.id = pr.supplier_id
                        AND sup.is_active
                      WHERE pr.lot_id = l.id
                    ) pr
    ON (1 = 1)
 WHERE l.organization_id = 964;

CREATE INDEX pr_lot_id_doc_number_ix
    ON buy.purchase_result(lot_id, is_active, supplier_id, doc_number);

CREATE UNIQUE INDEX sup_info_ux ON req.supplier(id, is_active, name_full);
```

The execution plan of the query with the LATERAL item

```
                                        QUERY PLAN
-----------------------------------------------------------------------------------------
Nested LOOP LEFT JOIN (COST=7.77..64162.05 ROWS=7461 width=80) (actual
TIME=1.479..135.591 ROWS=7495 loops=1)
  -> INDEX Scan USING lot_dic_cur_id_year_status_org_id_type_correct_last_version_ix ON
lot l (COST=0.42..9136.45 ROWS=7461 width=8) (actual TIME=1.416..21.601 ROWS=7495 LOOP
s=1)
       INDEX Cond: (organization_id = 964)
  -> AGGREGATE (COST=7.35..7.36 ROWS=1 width=72) (actual TIME=0.014..0.014 ROWS=1
loops=7495)
    -> Nested LOOP LEFT JOIN (COST=0.83..7.33 ROWS=2 width=104) (actual TIME=0.006..0.008
ROWS=0 loops=7495)
      -> INDEX ONLY Scan USING pr_lot_id_doc_number_ix ON purchase_result pr
(COST=0.42..2.46 ROWS=2 width=15) (actual TIME=0.004..0.004 ROWS=0 loops=7495)
           INDEX Cond: (lot_id = l.id)
           Heap Fetches: 0
      -> INDEX ONLY Scan USING sup_info_ux ON supplier sup (COST=0.41..2.43 ROWS=1
width=105) (actual TIME=0.005..0.005 ROWS=1 loops=3419)
           INDEX Cond: ((id = pr.supplier_id) AND (is_active = TRUE))
           Heap Fetches: 0
Planning TIME: 1.268 ms
Execution TIME: 136.788 ms
```

The original query execution time: 243821.165 ms

The current query execution time: 136.788 ms

Statement optimization with filtering on a computed column

It is required to filter rows by using the year value extracted from the **date_delivery_to** column

```
EXPLAIN (ANALYZE)
SELECT l.id
  FROM req.lot l
  LEFT JOIN (SELECT l.id
                  , EXTRACT(YEAR FROM l.date_delivery_to) delivery
                FROM req.lot l
            ) date_to
    ON date_to.id = l.id
 WHERE l.organization_id = 964
   AND date_to.delivery >= 2019;
```

What will be the execution plan in this case?

The execution plan of the query with filtering on a computed column

```
                                    QUERY PLAN
------------------------------------------------------------------------------------------
Nested LOOP (COST=0.42..45711.14 ROWS=2486 width=8) (actual TIME=4.558..200.813 ROWS=4081
loops=1)
  -> Seq Scan ON lot l (COST=0.00..27887.24 ROWS=7459 width=8) (actual TIME=1.260..157.953
ROWS=7495 loops=1)
       Filter: (organization_id = 964)
       ROWS Removed BY Filter: 273484
  -> INDEX Scan USING pk__lot ON lot l_1 (COST=0.42..2.39 ROWS=1 width=8) (actual
TIME=0.005..0.005 ROWS=1 loops=7495)
       INDEX Cond: (id = l.id)
       Filter: (DATE_PART('year'::TEXT, (date_delivery_to)::TIMESTAMP WITHOUT TIME ZONE) >=
'2019'::DOUBLE PRECISION)
       ROWS Removed BY Filter: 0
Planning TIME: 0.905 ms
Execution TIME: 201.428 ms
```

Is it possible to execute this statement without re-accessing the **lot** table?

Replacing filtering on a calculated column

If the year >= 2019, then date_delivery_to >= ‘2019-01-01’::date, which avoids re-accessing the lot table

```
EXPLAIN (ANALYZE)
SELECT l.id
  FROM req.lot l
 WHERE l.organization_id = 964
   AND l.date_delivery_to >= make_date(2019, 1, 1);
```

For improving query speed an additional index is required.

```
CREATE INDEX org_id_ddt_ix ON req.lot(organization_id,
date_delivery_to);
```

How will change the execution plan in this case?

The execution plan of the query after replacement filtering on a calculated column

```
                                        QUERY PLAN
----------------------------------------------------------------------------------------
Bitmap Heap Scan ON lot l (COST=39.97..4879.92 ROWS=3078 width=8) (actual
TIME=1.017..7.861 ROWS=4081 loops=1)
  RECHECK Cond: ((organization_id = 964) AND (date_delivery_to >= '2019-01-01'::DATE))
  Heap Blocks: exact=2325
    -> Bitmap INDEX Scan ON org_id_ddt_ix (COST=0.00..39.20 ROWS=3078 width=0) (actual
TIME=0.650..0.651 ROWS=4081 loops=1)
         INDEX Cond: ((organization_id = 964) AND (date_delivery_to >= '2019-01-
01'::DATE))
Planning TIME: 0.332 ms
Execution TIME: 8.129 ms
```

The original query execution time: 201.428 ms

The current query execution time: 8.129 ms

Query tuning with a calculated expression based on two columns from one table

In this case we need to find rows with a non-zero section, which is a calculated expression based on two columns from one table.

```
EXPLAIN (ANALYZE)
WITH ds AS (
SELECT l.id

      , CASE
          WHEN EXTRACT(YEAR FROM l.date_planned) = 2019 AND
               EXTRACT(YEAR FROM l.date_delivery_from) = 2019 THEN 1
          WHEN EXTRACT(YEAR FROM l.date_planned) = 2019 AND
               EXTRACT(YEAR FROM l.date_delivery_from) > 2019 THEN 21
          ELSE 0
        END AS razdel
  FROM req.lot l
 WHERE l.year < 2019
)
SELECT *
  FROM ds
 WHERE razdel != 0;
```

The execution plan of the query with the calculated expression based on two columns from one table

```
                                    QUERY PLAN
--------------------------------------------------------------------------------
Bitmap Heap Scan ON lot l (cost=1712.96..39577.92 ROWS=179325 width=12) (actual
TIME=143.985..523.901 ROWS=1445 loops=1)
  Recheck Cond: (YEAR < 2019)
  FILTER: (CASE WHEN ((date_part('year'::text, (date_planned)::TIMESTAMP WITHOUT TIME
zone) = '2019'::DOUBLE PRECISION) AND (date_part('year'::text, (date_delivery_from)::times
tamp WITHOUT TIME zone) = '2019'::DOUBLE PRECISION)) THEN 1 WHEN ((date_part('year'::text,
(date_planned)::TIMESTAMP WITHOUT TIME zone) = '2019'::DOUBLE PRECISION) AND
(date_part('year'::text, (date_delivery_from)::TIMESTAMP WITHOUT TIME zone) >
'2019'::DOUBLE PRECISION)) THEN 21 ELSE 0 END <> 0)
  ROWS Removed BY FILTER: 178781
  Heap Blocks: exact=20196
    -> Bitmap INDEX Scan ON ix__lot__year__is_last_version__is_active (cost=0.00..1668.12
ROWS=180227 width=0) (actual TIME=14.598..14.599 ROWS=180226 loops=1)
         INDEX Cond: (YEAR < 2019)
Planning TIME: 4.737 ms
Execution TIME: 524.258 ms
```

There is a huge difference between estimated and actual row counts (179325 and 1445).

Is it possible to replace this calculated expression?

Replacing the calculated column with two additional filter conditions

At any date from the segment 2019-01-01 and 2019-12-31 the year will be equal to 2019.

At any date >= 2019-01-01 the year >= 2019.

So, it is possible to replace the calculated expression with new filtration clauses.

```
EXPLAIN (ANALYZE)
SELECT l.id
  FROM req.lot l
 WHERE l.year < 2019
   AND l.date_planned BETWEEN make_date(2019, 1, 1) AND make_date(2019,
12, 31)
   AND l.date_delivery_from >= make_date(2019, 1, 1);
```

How will the estimated row counts change in this case?

The execution plan of the query after the calculated expression replacement

```
                                        QUERY PLAN
-----------------------------------------------------------------------------------
Bitmap Heap Scan ON lot l (cost=1670.43..29649.97 ROWS=9215 width=8) (actual
TIME=110.011..346.557 ROWS=1445 loops=1)
  Recheck Cond: (YEAR < 2019)
  FILTER: ((date_planned >= '2019-01-01'::DATE) AND (date_planned <= '2019-12-
31'::DATE) AND (date_delivery_from >= '2019-01-01'::DATE))
  ROWS Removed BY FILTER: 178781
    Heap Blocks: exact=20196
      -> Bitmap INDEX Scan ON ix__lot__year__is_last_version__is_active
(cost=0.00..1668.12 ROWS=180227 width=0) (actual TIME=16.352..16.353 ROWS=180226
loops=1)
           INDEX Cond: (YEAR < 2019)
Planning TIME: 1.963 ms
Execution TIME: 346.833 ms
```

It is clear, that estimated row count has dramatically reduced from 179325 to 9215, which means 19x faster.

What can be done to improve the estimates?

Extended statistics usage for correcting rows estimates in a query plan

Let’s use the extended statistics of the mcv type to determine how often the combination of the **year** and **date_planned** fields occurs. We also need to increase the columns statistics target to improve their frequencies accuracy.

```
CREATE STATISTICS lot_year_date_planned(mcv) ON YEAR, date_planned FROM
req.lot;

ALTER TABLE req.lot ALTER COLUMN YEAR SET STATISTICS 1250;

ALTER TABLE req.lot ALTER COLUMN date_planned SET STATISTICS 1250;

ANALYZE req.lot;
```

We also should build an additional index to speed up the query.

```
CREATE INDEX req_dp_ddf_year_ix ON req.lot(date_planned,
date_delivery_from, YEAR);
```

The execution plan of the query after the extended statistics gathering and the index building

```
                                  QUERY PLAN
---------------------------------------------------------------------------------
Bitmap Heap Scan ON lot l (cost=634.36..3059.41 ROWS=1397 width=8) (actual
TIME=5.102..7.942 ROWS=1445 loops=1)
  Recheck Cond: ((date_planned >= '2019-01-01'::DATE) AND (date_planned <= '2019-
12-31'::DATE) AND (date_delivery_from >= '2019-01-01'::DATE) AND (YEAR < 2019))
  Heap Blocks: exact=936
  -> Bitmap INDEX Scan ON req_dp_ddf_year_ix (cost=0.00..634.01 ROWS=1397 width=0)
(actual TIME=4.970..4.970 ROWS=1445 loops=1)
       INDEX Cond: ((date_planned >= '2019-01-01'::DATE) AND (date_planned <=
'2019-12-31'::DATE) AND (date_delivery_from >= '2019-01-01'::DATE) AND (YEAR <
2019))
Planning TIME: 3.181 ms
Execution TIME: 8.060 ms
```

The original query execution time: 523.901 ms.

The current query execution time: 7.942 ms.

It is the least difference between estimated and actual row counts.

Extended statistics and IN clauses

```
SELECT r.case_id
  FROM ci_case_char r
 WHERE r.char_type_cd = 'RETLTYPE'
   AND r.char_val IN ('0', '2', '8');
```

In PostgreSQL 12, the estimated row count was less than the actual number by more than 100 times. Extended statistics didn’t help in this case, so the IN clause was replaced on additional filter clauses united by OR operators.

```
SELECT r.case_id
  FROM ci_case_char r
 WHERE r.char_type_cd = 'RETLTYPE’
   AND (r.char_val = '0' OR r.char_val = '2' OR r.char_val = '8');
```

However, starting from PostgreSQL 13 this issue gets resolved by creating extended statistics of the mcv type.

Excluding filtering conditions during query planning

In PostgreSQL, it is possible to exclude filtering conditions at the stage of query planning. Let’s consider how the following construction will work based on the value of the **version_cond** parameter.

```
WITH params AS NOT MATERIALIZED (
SELECT :version_cond AS version_cond
)
SELECT l.id
  FROM req.lot l
  JOIN params p
    ON (1 = 1)
 WHERE l.year = 2019
   AND ((p.version_cond = 1 AND l.status = 50 AND l.type_correct = 0) OR
        (p.version_cond = 2 AND l.status = 50 AND l.is_last_version)
       );
```

The execution plan of the query in case of version_cond = 1

There is no need to filter rows by the **is_last_version** column, because it meets the **version_cond = 2** condition.

```
                                    QUERY PLAN
--------------------------------------------------------------------------------
Bitmap Heap Scan ON lot l (cost=212.32..15299.08 ROWS=14580 width=8) (actual
TIME=1.716..14.347 ROWS=18576 loops=1)
  Recheck Cond: ((YEAR = 2019) AND (type_correct = 0) AND (STATUS = 50))
  Heap Blocks: exact=3262
  -> Bitmap INDEX Scan ON year_type_cor_status_ix (cost=0.00..208.67 ROWS=14580
width=0) (actual TIME=1.243..1.244 ROWS=18576 loops=1)
       INDEX Cond: ((YEAR = 2019) AND (type_correct = 0) AND (STATUS = 50))
Planning TIME: 0.364 ms
Execution TIME: 15.251 ms
```

The execution plan of the query in case of version_cond = 2

There is no need to filter rows by the **type_correct** column, since it meets the version_cond = 1 condition.

```
                                          QUERY PLAN
-----------------------------------------------------------------------------
Bitmap Heap Scan ON lot l (cost=212.32..15262.63 ROWS=14580 width=8) (actual
TIME=3.067..36.650 ROWS=18576 loops=1)
  Recheck Cond: ((YEAR = 2019) AND (STATUS = 50))
  FILTER: is_last_version
  Heap Blocks: exact=3262
  -> Bitmap INDEX Scan ON year_type_cor_lv_ix (cost=0.00..208.67 ROWS=14580
width=0) (actual TIME=2.612..2.612 ROWS=18576 loops=1)
       INDEX Cond: ((YEAR = 2019) AND (is_last_version = TRUE) AND (STATUS = 50))
Planning TIME: 3.586 ms
Execution TIME: 37.574 ms
```

The execution plan of the query in case of version_cond = 3

If version_cond = 3, then an empty dataset will be returned, since 3 is not equal to 1 and 2. All of this happens during the query planning stage.

```
                                         QUERY PLAN
-----------------------------------------------------------------------------
RESULT  (cost=0.00..0.00 ROWS=0 width=0) (actual TIME=0.002..0.002 ROWS=0 loops=1)
  One-TIME FILTER: FALSE
Planning TIME: 0.429 ms
Execution TIME: 0.034 ms
```

In PostgreSQL, it is possible to exclude certain query conditions during query planning, which allows developer to write less dynamic SQL code.

Links

- pg_stat_statements module. https://www.postgresql.org/docs/13/pgstatstatements.html
- pg_stat_kcache module. https://github.com/powa-team/pg_stat_kcache
- pg_wait_sampling module. https://github.com/postgrespro/pg_wait_sampling
- auto_explain module. https://www.postgresql.org/docs/13/auto-explain.html
- pgpro_stats module. https://postgrespro.com/docs/enterprise/12/pgpro-stats
- pg_profile module. https://github.com/zubkov-andrei/pg_profile
- pgpro_pwr module. https://postgrespro.com/docs/enterprise/12/pgpro-pwr
- mamonsu. https://github.com/postgrespro/mamonsu
- zabbix agent 2 https://github.com/zabbix/zabbix/tree/master/src/go/cmd/zabbix_agent2

Postgres Professional

http://postgrespro.com/

p.petrov@postgrespro.com

info@postgrespro.com