Understanding EXPLAIN

Table des matières

Understanding EXPLAIN....4
1 Slides license....5
2 Author....5
3 How can I understand EXPLAIN?....6
4 First, our objects....6
5 Let's try to read our table....7
6 Let's update our table statistics....8
7 What does ANALYZE do?....8
8 Width....9
9 Rows....10
10 Total cost....10
11 But what happens really?....11
12 And at the physical level?....12
13 Let's use the BUFFERS option....12
14 And now, with a warm up cache?....13
15 Let's try with a bigger cache....14
16 By the way....14
17 Now, let's try a WHERE clause....15
18 Cost - more complicated process....16
19 Number of rows - ditto....17
20 Would an index help here?....17
21 Why not use the index?....18
22 Let's try to force the use of index....19
23 Let's see with another query....19
24 Let's complicate the filter....20
25 Let's try to filter on c2 only....20
26 Let's add an index on c2....21
27 Let's add an index with an opclass....21
28 Let's try a covering index....22
29 All the major scan nodes....23
30 Let's try an ORDER BY clause....23
31 Are we sure it writes on the disk?....24
32 Let's bring more mem to the sort....25
33 An index can also help here....25
34 Let's get back to c2....26
35 And now, let's limit the rows....26
36 Let's have another table....27
37 ... and join them....27
38 It helps to have an index....28
39 Left join anyone?....29
40 Let's DROP an index....29
41 We delete some more row....30
42 And with an empty table?....31
43 Always used in a CROSS JOIN....31
44 All the join nodes....32

45 What about inherited tables?....................32
46 Let's SELECT on baz....................33
47 Let's try with an index....................34
48 Let's try an aggregate....................34
49 Let's try max()....................35
50 Max() is better with an index....................35
51 Works also with booleans in 9.2....................36
52 Let's GROUP BY....................36
53 Having a bigger work_mem helps....................37
54 Or an index....................37
55 All the aggregate nodes....................38
56 Tools....................38
57 pgAdmin....................39
58 explain.depesz.com....................39
59 explain.depesz.com screenshot....................40
60 pg_stat_plans....................40
61 pg_stat_plans example....................41
62 By the way....................41
63 Conclusion....................42
64 Question?....................42

Understanding EXPLAIN

1 Slides license

- Creative Common BY-NC-SA
- You are free
  - to Share
  - to Remix
- Under the following conditions
  - Attribution
  - Noncommercial
  - Share Alike

2 Author

- Guillaume Lelarge
- Work
  - CTO of Dalibo
  - email: guillaume.lelarge@dalibo.com
- Community
  - pgAdmin, the french documentation, pgconf.eu organization, vice-treasurer of PostgreSQL Europe, etc.
  - email: guillaume@lelarge.info
  - twitter: @g_lelarge

3 How can I understand EXPLAIN?

- EXPLAIN is a bit scary
- But you just need a few hints to understand it
- That's what this whole talk is about
  - understand it
  - use it
  - love it

EXPLAIN is a really nice command that gives you lots of informations but it's often easy to not know what to do with all this.

That's quite unfortunate because there isn't much to know in it.

That's what this whole talk is about: get a nice understanding of the informations given by this command, know how to use this information, and fix your queries so that they work faster.

4 First, our objects

- Create them

```
CREATE TABLE foo (c1 integer, c2 text);
INSERT INTO foo
  SELECT i, md5(random()::text)
  FROM generate_series(1, 1000000) AS i;
```

This two queries add a table, and fill it with random informations. This will be the root of our examples.

5 Let's try to read our table

```
EXPLAIN SELECT * FROM foo;

                QUERY PLAN
------------------------------------------
 Seq Scan on foo  (cost=0.00..18584.82
                  rows=1025082 width=36)
```

- Cost
  - cost to get the first row: 0.00
  - cost to get all rows: 18584.82
  - in “page cost” unit
- Rows
  - number of rows
- Width
  - average width of a row
  - in bytes

When you want to read a table, PostgreSQL has many ways to do it. The simpler one is to read it sequentially, block by block.

That's what this execution plan shows. You also have some statistical informations given by the planner.

The first cost number is the cost to get the first row. The second number is the cost to get all the rows from the sequential scan. Beware that this cost is not in time unit. This is a page cost unit. You usually compare these numbers to the sequential page cost (which is 1.0 by default).

The rows number is the number of rows the planner is expecting to get while doing the sequential scan.

The width number is the average width of one row of the resulting data set.

What we can say here is that the number of rows is a bit off the real number of rows. Not much though as the autovacuum probably fired while doing the INSERTs.

6 Let's update our table statistics

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

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

- Yay, good number of rows
- But how are the statistics calculated?

So we need to update the table's statistics, and for this, we use the ANALYZE command. As I only want statistics on the table foo, I specify the table, but it's not required.

Once the ANALYZE is done, if I execute the EXPLAIN command once again, statistical informations from the planner are fixed. The number of rows is good, the width is a bit higher, and the cost is a bit lower (the cost depends on the number of rows, not on the average row width).

This is all good, but how are all these informations calculated? Why ANALYZE is important here?

7 What does ANALYZE do?

- It reads some random rows of the table
  - 300*default_statistics_target
- It computes some statistics of the values in each column of the table
- Number of most common values, and histogram
  - depends on default_statistics_target
  - may be customized for each column
- And it stores the column statistics in the pg_statistic catalog
- Try the pg_stats view
  - easier to read
  - and lots of quite interesting informations

ANALYZE reads a part of each table in the database (unless you give a specific table, in which case it only reads a part of this table). The sample it reads is taken randomly. Its size depends on the default_statistics_target parameter value. It'll read 300 times this value. By default, default_statistics_target is 100 (since 8.4), so, by default, it will read 30k random rows in the table.

On this sample, it computes some statistical informations, such ash the percentage of NULL values, the average width of a row, the number of distinct values, etc. It also stores the most common values and their frequencies. The number of these values depends on the value of the default_statistics_target parameter. You can customize this number for each row with an ALTER TABLE statement.

Every statistical information is stored in the pg_statistic system catalog. It's quite hard to read, so there is a nicer view called pg_stats. We'll see a few of its columns during the rest of the talk.

We'll now see how the planner computes the statistical informations given by the EXPLAIN statement.

8 Width

```
SELECT sum(avg_width) AS width
FROM pg_stats
WHERE tablename='foo';

 width
-------
    37
```

- Average size (in bytes) of a row in the resulting data set
- The lesser, the better

The width information is the average width of the row in the resulting data set. As the ANALYZE statement gathers the average width of each column of each table, we can get this information by summing the value of each column of a table. This gives the SQL query in the slide.

9 Rows

```
SELECT reltuples FROM pg_class WHERE relname='foo';

 reltuples
-----------
     1e+06
```

- All relations metadata appear in the pg_class catalog
- Many interesting informations
  - reltuples
  - relpages
  - relallvisible

The number of rows is easier to get. The pg_class system catalog is a catalog of all relations (tables, indexes, sequences, etc) in the database. This catalog holds metadatas about the relations. It also holds some statistical informations, such as the estimated number of rows (column reltuples), the estimated number of file pages (relpages), and the number of pages containing only visibles rows for all the current transactions (relallvisible).

This time, we are interested in reltuples.

10 Total cost

- In a sequential scan, the executor needs:
  - to read all the blocks of relation foo
  - to check every row in each block to filter “unvisible” rows

```
SELECT relpages*current_setting('seq_page_cost')::float4
   + reltuples*current_setting('cpu_tuple_cost')::float4
       AS total_cost
FROM pg_class
WHERE relname='foo';

 total_cost
------------
      18334
```

The total cost is a bit more difficult to compute. When PostgreSQL does a sequential scan, it mostly has to do two things. First, it needs to read all the blocks of the table. For each block, it needs to find and check each row. So there is the cost of reading each blocks. The postgresql.conf file indicates the cost to read one sequential page with the seq_page_cost parameter. The cost to read all blocks of a table is the seq_page_cost parameter value times the number of blocks of the relation (relpages in pg_class). Then, to check every row, there is also a cost in the postgresql.conf file: cpu_tuple_cost. All it needs to do is multiply this cost with the number of rows (reltuples in pg_class). And then it adds these two costs, and it's done.

So all is fine and we know how the planner calculates the numbers for this query.

11 But what happens really?

- Let's use the ANALYZE option to find out

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

                 QUERY PLAN
----------------------------------------------
 Seq Scan on foo  (cost=0.00..18334.00
                   rows=1000000
                   width=37)
                  (actual time=0.016..96.074
                   rows=1000000
                   loops=1)
 Total runtime: 132.573 ms
```

- 3 new informations
  - real execution time in milliseconds
  - real number of lines
  - and number of loops
- Be careful, it really executes the query!

But maybe you want to know what really happens if you execute the query. It's usually better to get the actual execution time, and the actual number of rows.

We can do this by adding the ANALYZE OPTION to the EXPLAIN command. Notice the parentheses around the option. It's the new way to add options to the EXPLAIN command. New as in “since 9.0”.

The results show three more informations, coming from the actual execution of the query. We first get the actual execution time to get the first row, the actual execution time to get all rows, the actual number of rows, and the number of loops. We won't explain the loop right now. The execution times are in milliseconds, so the sequential scan here lasts 96ms, and gathers 1 million rows. The estimated number of rows was quite right.

Be careful when using the ANALYZE option with any DML command. An EXPLAIN ANALYZE UPDATE... will really update some rows. Same thing with DELETE, or INSERT. Be sure to enclose them in an explicit transaction, so that you can ROLLBACk the changes.

12 And at the physical level?

- First, let's drop all caches

```
  $ /etc/init.d/postgresql stop
  $ sync
  $ echo 3 > /proc/sys/vm/drop_caches
  $ /etc/init.d/postgresql start
```

Now let's see what happens at the physical level.

13 Let's use the BUFFERS option

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

                  QUERY PLAN
------------------------------------------------
 Seq Scan on foo  (cost=0.00..18334.00
                   rows=1000000 width=37)
                  (actual time=14.638..507.726
                   rows=1000000 loops=1)
   Buffers: shared read=8334
 Total runtime: 638.084 ms
```

- BUFFERS is a 9.0 feature
- New informations:
  - blocks read, and written
  - in the PostgreSQL cache (hit)
  - and outside (read)

We will use another option of the EXPLAIN command. This option, called BUFFERS, is available since the 9.0 release. It adds some new informations related to buffers: buffers read in the cache of PostgreSQL, buffers read by Linux, buffers written, buffers for temporary objects, etc.

On this example, “shared read” are the number of blocks read outside of PostgreSQL. It makes sense since we just launched PostgreSQL, and so we have an empty cache. PostgreSQL had to access 8334 blocks to read the whole table.

14 And now, with a warm up cache?

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

                  QUERY PLAN
----------------------------------------------
 Seq Scan on foo  (cost=0.00..18334.00
                   rows=1000000 width=37)
                  (actual time=0.046..115.908
                   rows=1000000 loops=1)
   Buffers: shared hit=32 read=8302
 Total runtime: 171.803 ms


SELECT current_setting('shared_buffers') AS shared_buffers,
       pg_size_pretty(pg_table_size('foo')) AS table_size;

 shared_buffers | table_size
----------------+------------
 32MB           | 65 MB
```

Let's execute the query a second time, now that the cache is warmed up. What we discover now is that we read 32 blocks from PostgreSQL cache, and 8302 outside of PostgreSQL cache. That's a lot of non-cached data.

One of the reason it does so is that our table is bigger than the cache. Our table size is 65MB, more than twice the PostgreSQL cache. But the real reason is that PostgreSQL uses a ring buffer optimization. It means a single SELECT can't use the whole cache for itself. The same thing happens with VACUUM. It tries to be nice with potential other sessions. With a bigger cache, we will have the same behaviour, but a bigger part of the table will be able to stay in the cache.

Notice anyway that the query is three times faster than with an empty cache. Having lots of Linux disk cache really helps.

15 Let's try with a bigger cache

- With shared_buffers at 320MB

```
EXPLAIN (ANALYZE,BUFFERS) SELECT * FROM foo;
               QUERY PLAN
-----------------------------------------------
 Seq Scan on foo  (cost=0.00..18334.00
                   rows=1000000 width=37)
                  (actual time=15.009..532.372
                   rows=1000000 loops=1)
   Buffers: shared read=8334
 Total runtime: 646.289 ms

EXPLAIN (ANALYZE,BUFFERS) SELECT * FROM foo;
               QUERY PLAN
-----------------------------------------------
 Seq Scan on foo  (cost=0.00..18334.00
                   rows=1000000 width=37)
                  (actual time=0.014..100.762
                   rows=1000000 loops=1)
   Buffers: shared hit=8334
 Total runtime: 147.999 ms
```

Let's put 320MB in PostgreSQL cache. We empty the Linux cache et restart PostgreSQL.

At the first query run, we have all the table read from disk, and the query takes 646ms to execute. At the second run, everything is in the PostgreSQL cache (we only have shared hits), and the query takes only 147ms.

16 By the way

- If you want to know more about the buffer cache
  - go see Bruce Momjian's talk
- “Inside PostgreSQL Shared Memory”
  - Tomorrow, 1:40pm to 2:30pm
  - Room Seine

17 Now, let's try a WHERE clause

```
EXPLAIN SELECT * FROM foo WHERE c1 > 500;

               QUERY PLAN
-----------------------------------------
 Seq Scan on foo  (cost=0.00..20834.00
                   rows=999579 width=37)
   Filter: (c1 > 500)
```

- New line: `Filter: (c1 > 500)`
- Cost is higher
  - less rows
  - but more work because of the filtering
- Number of rows is not correct, but good enough

This time, we use a WHERE clause. We don't have an index on this column (and actually, we don't have any index for this table), so the only way to answer the query is to do a sequential scan. While reading each block and checking each row's visibility, PostgreSQL will check that the c1 value is bigger than 500. If it isn't, it'll skip the row. That's the meaning of the new “Filter” line.

Even if we end up with less rows, we have more work to do because we need to check the value of the c1 column on each row of the table. So, the estimated total cost will be bigger than the query without the filter.

As you can see, the estimated number of rows is not exact, but it's quite good nonetheless.

18 Cost - more complicated process

- The executor
  - reads all the blocks of relation foo
  - filters the rows according to the WHERE clause
  - checks the remaining rows to make sure they are visible

```
SELECT
   relpages*current_setting('seq_page_cost')::float4
 + reltuples*current_setting('cpu_tuple_cost')::float4
 + reltuples*current_setting('cpu_operator_cost')::float4
   AS total_cost
FROM pg_class
WHERE relname='foo';

 total_cost
------------
      20834
```

It's a bit trickier to compute the cost. When PostgreSQL needs to execute this query, it needs to read all the blocks of the table. So we still have the reading cost with this formulae: number of blocks times cost of a sequential scan. It also still needs to check the visibility of each row, so the old formulae stands again: number of tuples times cpu_tuple_cost. The only difference is in the filter: we need to use the operator on the column for each row. We also have a parameter that gives the cost to use an operator (named cpu_operator_cost). We'll use this operator on each line, so we get this new formulae: number of tuples times cpu_operator_cost.

Add each specific cost, and we get the whole total cost of the query.

19 Number of rows - ditto

```
SELECT histogram_bounds
FROM pg_stats
WHERE tablename='foo' AND attname='c1';

                    histogram_bounds
-----------------------------------------------------
 {57,10590,19725,30449,39956,50167,59505,...,999931}


SELECT round(
(
 (10590.0-500)/(10590-57)
 +
 (current_setting('default_statistics_target')::int4 - 1)
)
* 10000.0
) AS rows;

  rows
---------
 999579
```

The estimated number of rows is even more complicated.

PostgreSQL uses an histogram of the values to get an estimate of the number of rows. This histogram is splitted in 100 parts containing the same number of rows. 100 is the default value of default_statistics_target. The table contains 1 million rows, so each part will have 10000 rows.

The first part has values between 57 and 10590. As we are looking for values after 500, it means we'll have only a percentage of this part, and all the other parts completely. The formulae on the slide reflects this, and gives the number of rows estimated by the planner.

20 Would an index help here?

```
CREATE INDEX ON foo(c1);
EXPLAIN SELECT * FROM foo WHERE c1 > 500;

               QUERY PLAN
-----------------------------------------
 Seq Scan on foo  (cost=0.00..20834.00
                  rows=999529 width=37)
   Filter: (c1 > 500)
```

- Nope...

Usually, everybody thinks an index would help to run a query faster if we have a WHERE clause.

In our case, that isn't true. Remember, this is a 1 million rows table, and we skip only 500 rows. It's much better to do the filter with the CPU than to use an index.

21 Why not use the index?

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

                   QUERY PLAN
-----------------------------------------------
 Seq Scan on foo  (cost=0.00..20834.00
                   rows=999493 width=37)
                  (actual time=0.125..147.613
                   rows=999500 loops=1)
   Filter: (c1 > 500)
   Rows Removed by Filter: 500
 Total runtime: 184.901 ms
```

- New line: `Rows Removed by Filter: 500`
  - Displayed only with ANALYZE option
  - 9.2 feature
- It needs to read 99.95% of the table!

If you don't have any knowledge of the table, the 9.2 release helps you a bit more. Let's try an `EXPLAIN ANALYZE` on the foo table in 9.2. Notice the new “Rows Removed by Filter” line. 500 rows are removed by the filter and we end up with 999500 rows in the resulting set. It needs to read 99.95% of the table! It won't ever use an index in that case.

22 Let's try to force the use of index

```
SET enable_seqscan TO off;
EXPLAIN (ANALYZE) SELECT * FROM foo WHERE c1 > 500;
SET enable_seqscan TO on;

                    QUERY PLAN
----------------------------------------------------
 Index Scan using foo_c1_idx on foo
   (cost=0.00..36801.25 rows=999480 width=37)
   (actual time=0.099..186.982 rows=999500 loops=1)
   Index Cond: (c1 > 500)
 Total runtime: 221.354 ms
(3 rows)
```

- Works, but slowly
- The planner was right
  - the index here has no value

To make sure it really costs more, let's try to force the planner to use an index scan. To do that, we have all sort of “enable_*” parameters. The one we need is the enable_seqscan parameter that we set to off. Beware that it won't disable completely sequential scan (if you don't have an index, there is no other way than a sequential scan), but it will make it really harder to use an sequential scan (by adding a really impressive cost to this specific kind of node).

So, when we set enable_seqscan to off, it will use an index. But you can see that it takes a bit more time to use the index (221ms) than it it to read the whole table and to filter it (184ms).

23 Let's see with another query

```
EXPLAIN SELECT * FROM foo WHERE c1 < 500;

              QUERY PLAN
-----------------------------------------
 Index Scan using foo_c1_idx on foo
   (cost=0.00..24.59 rows=471 width=37)
   Index Cond: (c1 < 500)
```

- New line: `Index Cond: (c1 < 500)`
- It still reads the table to get the visibility informations

Now, let's change the filter. This time, we have an index scan. We have that kind of node because we read a small number of rows (only 499 on 1 million).

24 Let's complicate the filter

```
EXPLAIN SELECT * FROM foo
        WHERE c1 < 500 AND c2 LIKE 'abcd%';

               QUERY PLAN
----------------------------------------
 Index Scan using foo_c1_idx on foo
   (cost=0.00..25.77 rows=71 width=37)
   Index Cond: (c1 < 500)
   Filter: (c2 ~~ 'abcd%'::text)
```

- “Index Cond”, and “Filter”

Now let's add another filter. What we end up with is an index scan on the c1 column, and a filter on the rows read in the table to get only the rows which have “abcd” at the beginning of the c2 column.

25 Let's try to filter on c2 only

```
EXPLAIN (ANALYZE)
SELECT * FROM foo WHERE c2 LIKE 'abcd%';

                  QUERY PLAN
-----------------------------------------------
 Seq Scan on foo  (cost=0.00..20834.00
                   rows=100 width=37)
                  (actual time=21.435..134.153
                   rows=15 loops=1)
   Filter: (c2 ~~ 'abcd%'::text)
   Rows Removed by Filter: 999985
 Total runtime: 134.198 ms
```

- 999985 rows removed by the filter
- Only 15 rows as a result
- So an index would definitely help here…

Here, we only filter on c2. We don't have index on c2, so PostgreSQL does a sequential scan. What we can notice here is that the sequential scan reads a million rows (999985+15), and only keeps 15 of them. An index on c2 would definitely help.

26 Let's add an index on c2

```
CREATE INDEX ON foo(c2);
EXPLAIN (ANALYZE) SELECT * FROM foo
WHERE c2 LIKE 'abcd%';

                  QUERY PLAN
------------------------------------------------
 Seq Scan on foo  (cost=0.00..20834.00
                   rows=100 width=37)
                  (actual time=22.189..143.467
                   rows=15 loops=1)
   Filter: (c2 ~~ 'abcd%'::text)
   Rows Removed by Filter: 999985
 Total runtime: 143.512 ms
```

- It didn't work!
- UTF8 encoding...
  - we need to use an operator class

So let's add an index on c2. Once we execute the EXPLAIN, we can see that we still have a sequential scan. The index doesn't seem interesting to answer the query. That is weird.

Actually, the problem is that the index doesn't fit. My c2 column stores values in UTF8 encoding. When you use a encoding other than C, you need to specifiy the operator class (varchar_pattern_ops, text_pattern_ops, etc.) while creating the index.

27 Let's add an index with an opclass

```
CREATE INDEX ON foo(c2 text_pattern_ops);
EXPLAIN SELECT * FROM foo WHERE c2 LIKE 'abcd%';

                   QUERY PLAN
--------------------------------------------------
 Bitmap Heap Scan on foo  (cost=7.29..57.91
                          rows=100 width=37)
   Filter: (c2 ~~ 'abcd%'::text)
   ->  Bitmap Index Scan on foo_c2_idx1
         (cost=0.00..7.26 rows=13 width=0)
         Index Cond: ((c2 ~>=~ 'abcd'::text)
                     AND (c2 ~<~ 'abce'::text))
```

- New kind of nodes: Bitmap Index Scan

Once there is an index with the good operator class, the planner can use the index to make the query run faster.

Notice the new kind of scan node. Bitmap Index Scan is another way to use an index. And there is another one in 9.2.

28 Let's try a covering index

```
EXPLAIN SELECT c1 FROM foo WHERE c1 < 500;

                 QUERY PLAN
---------------------------------------------
 Index Only Scan using foo_c1_idx on foo
   (cost=0.00..26.40 rows=517 width=4)
   Index Cond: (c1 < 500)
```

- 9.2 feature
- Does NOT read the table for the visibility bits

In some queries, all the redundant informations stored in the index are the only informations needed to answer the query. The query is this slide is a perfect example. In the foo_c1_idx index, we have the value of the c1 column, so it can be used to do the filter and to give the value in the resulting set. We usually use the term “covering index” in such example.

The “Index Only Scan” node is used when a covering index is available. It's quicker than the usual “Index Scan” because it doesn't need to read the table to know if a row is visible or not.

It's a 9.2 feature.

29 All the major scan nodes

- Sequential Scan
  - read all the table in sequential order
- Index Scan
  - read the index to filter on the WHERE clause
  - and the table to filter the “invisible” rows
- Bitmap Index Scan
  - the same
  - but read fully the index, and then the table
  - much quicker for a bigger number of rows
- Index Only Scan
  - for covering index

So we have seen all the major scan nodes: sequential scan, index scan, bitmap index scan, index only scan. They all are interesting and useful to answer different kinds of queries.

There are other scan nods, such as the function scan or the values scan, but we won't have the time to get into such details during this talk.

30 Let's try an ORDER BY clause

```
DROP INDEX foo_c1_idx;
EXPLAIN (ANALYZE) SELECT * FROM foo ORDER BY c1;

                       QUERY PLAN
-------------------------------------------------------
 Sort
   (cost=172682.84..175182.84 rows=1000000 width=37)
   (actual time=612.793..729.135 rows=1000000 loops=1)
   Sort Key: c1
   Sort Method: external sort  Disk: 45952kB
   ->  Seq Scan on foo  (cost=0.00..18334.00
                         rows=1000000 width=37)
                        (actual time=0.017..111.068
                         rows=1000000 loops=1)
 Total runtime: 772.685 ms
```

There are many ways to answer to an “ORDER BY” (or to do a sort for that matter).

The simpler one (or the one that doesn't imply other objects than the table) is a sort done by the CPU when executing the query. This takes CPU time and memory. In this example, it needs so much memory that PostgreSQL does a sort on disk. It uses 45MB. As it writes many data on the disk, it takes a lot of time. The time used to do the sequential scan (111ms) is far less than the time to do the sorting (618ms, result of 729ms-111ms).

It's also the first time that we have a two-nodes plan. The first node to be executed is the sequential scan, and the last one is the sort. The first nodes to be executed are the one on the farest right, and the last ones are on the left.

Notice also the two more informations that we have: sort key, and sort method.

31 Are we sure it writes on the disk?

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

                        QUERY PLAN
-------------------------------------------------------
 Sort
   (cost=172682.84..175182.84 rows=1000000 width=37)
   (actual time=632.796..756.958 rows=1000000 loops=1)
   Sort Key: c1
   Sort Method: external sort  Disk: 45952kB
   Buffers: shared hit=3910 read=4424,
            temp read=5744 written=5744
   ->  Seq Scan on foo  (cost=0.00..18334.00
                         rows=1000000 width=37)
                        (actual time=0.134..110.617
                         rows=1000000 loops=1)
         Buffers: shared hit=3910 read=4424
 Total runtime: 811.308 ms
```

To make sure PostgreSQL really writes to disk the temporary data for the sort, we can use the BUFFERS option of EXPLAIN. The interesting line here is “temp read=5744 written=5744”. PostgreSQL wrote 5744 of temporary data, and read it. 5744 blocks of 8KB are the 45MB we were expecting.

32 Let's bring more mem to the sort

```
SET work_mem TO '200MB';
EXPLAIN (ANALYZE) SELECT * FROM foo ORDER BY c1;

                      QUERY PLAN
-------------------------------------------------------
 Sort
   (cost=117991.84..120491.84 rows=1000000 width=37)
   (actual time=552.401..598.984 rows=1000000 loops=1)
   Sort Key: c1
   Sort Method: quicksort  Memory: 102702kB
   ->  Seq Scan on foo  (cost=0.00..18334.00
                         rows=1000000 width=37)
                        (actual time=0.014..102.907
                         rows=1000000 loops=1)
 Total runtime: 637.525 ms
```

PostgreSQL can also do the sort in memory, but we need to give it enough memory to use. You can do this through the work_mem parameter. By default, it's at 1MB. Here, we set it to 200MB, and rexecute the query. PostgreSQL switches to a quicksort in-memory algorithm and that's quicker (and it's even way quicker in 9.2, thanks to Peter Geoghegan and its use of an optimized quicksort).

33 An index can also help here

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

                      QUERY PLAN
------------------------------------------------------
 Index Scan using foo_c1_idx on foo
   (cost=0.00..34318.35 rows=1000000 width=37)
   (actual time=0.030..179.810 rows=1000000 loops=1)
 Total runtime: 214.442 ms
```

An index contains data sorted, so another way to do an “ORDERY BY” is to read an index. In this example, we see that PostgreSQL prefers reading the index than sort the data in memory. And the result is even quicker: three times faster.

34 Let's get back to c2

```
DROP INDEX foo_c2_idx1;
EXPLAIN (ANALYZE,BUFFERS)
  SELECT * FROM foo WHERE c2 LIKE 'ab%';

                 QUERY PLAN
-------------------------------------------------
 Seq Scan on foo
 (cost=0.00..20834.00 rows=100 width=37)
 (actual time=0.066..134.149 rows=3978 loops=1)
   Filter: (c2 ~~ 'ab%'::text)
   Rows Removed by Filter: 996022
   Buffers: shared hit=8334
 Total runtime: 134.367 ms
```

Let's drop the index on the c2 column, and use EXPLAIN ANALYZE on a new query.

On this example, we want all rows which have 'ab' in the beginning of the c2 value. As there is no index, PostgreSQL does a sequential scan and gets rid of 996002 rows.

35 And now, let's limit the rows

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

                     QUERY PLAN
---------------------------------------------------
 Limit
 (cost=0.00..2083.40 rows=10 width=37)
 (actual time=0.065..0.626 rows=10 loops=1)
   Buffers: shared hit=19
   ->  Seq Scan on foo
       (cost=0.00..20834.00 rows=100 width=37)
       (actual time=0.063..0.623 rows=10 loops=1)
         Filter: (c2 ~~ 'ab%'::text)
         Rows Removed by Filter: 2174
         Buffers: shared hit=19
 Total runtime: 0.652 ms
```

With a LIMIT clause, we get the same sequential scan but we can notice than it only gets rid of 2174 rows. When there is a Limit node, the scan is warned that it does not need to get all the rows, but only the first X rows (in this example, the first 10 rows). And so, PostgreSQL had to read 2184 rows before having the ten rows that have the c2 column value beginning with 'ab'.

It also works with an index scan.

36 Let's have another table

- Create it

```
CREATE TABLE bar (c1 integer, c2 boolean);
INSERT INTO bar
  SELECT i, i%2=1
  FROM generate_series(1, 500000) AS i;
```

- Analyse it

```
ANALYZE bar;
```

We need to create another table to test joins. We also insert some random rows in it to populate the table. And finally, we run ANALYZE on it in order to get good statistics.

37 ... and join them

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

                      QUERY PLAN
----------------------------------------------------------
 Hash Join  (cost=15417.00..63831.00 rows=500000 width=42)
            (actual time=133.820..766.437 rows=500000 ...)
   Hash Cond: (foo.c1 = bar.c1)
   ->  Seq Scan on foo
       (cost=0.00..18334.00 rows=1000000 width=37)
       (actual time=0.009..107.492 rows=1000000 loops=1)
   ->  Hash  (cost=7213.00..7213.00 rows=500000 width=5)
      (actual time=133.166..133.166 rows=500000 loops=1)
         Buckets: 4096  Batches: 32  Memory Usage: 576kB
         ->  Seq Scan on bar
             (cost=0.00..7213.00 rows=500000 width=5)
             (actual time=0.011..49.898 rows=500000 loops=1)
 Total runtime: 782.830 ms
```

This is a simple join. There are a few ways to do these kind of joins and one of them is to do a HashJoin. The idea behind the hash join is to hash a table, and then hash each row of the other table and compare each hash with the hashed table. With such a join, you only can have an equality operator for the join.

On this example, PostgreSQL does a sequential scan of table bar, and computes the hash for each of its rows. Then, it does a sequential scan of foo, and for each row, computes the hash of the row and compares it to the bar hashed table. If it matches, the row will be put in the resulting set. If it doesn't match, the row is skipped.

It works well with enough memory to keep the hashed table in memory.

38 It helps to have an index

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

                        QUERY PLAN
---------------------------------------------------------
 Merge Join  (cost=0.82..39993.03 rows=500000 width=42)
       (actual time=0.039..393.172 rows=500000 loops=1)
   Merge Cond: (foo.c1 = bar.c1)
   ->  Index Scan using foo_c1_idx on foo
       (cost=0.00..34318.35 rows=1000000 width=37)
       (actual time=0.018..103.613 rows=500001 loops=1)
   ->  Index Scan using bar_c1_idx on bar
       (cost=0.00..15212.80 rows=500000 width=5)
       (actual time=0.013..101.863 rows=500000 loops=1)
 Total runtime: 411.022 ms
```

If you add an index on c1 for table bar, then PostgreSQL will see two indexes. This is a way to get sorted data, and a good way to join two tables is to merge two sorted data sets. With this new index, PostgreSQL can quickly read the data sorted for both tables, and so it prefers to do a merge join.

39 Left join anyone?

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

                        QUERY PLAN
---------------------------------------------------------
 Merge Left Join
 (cost=0.82..58281.15 rows=1000000 width=42)
 (actual time=0.036..540.328 rows=1000000 loops=1)
   Merge Cond: (foo.c1 = bar.c1)
   ->  Index Scan using foo_c1_idx on foo
       (cost=0.00..34318.35 rows=1000000 width=37)
       (actual time=0.016..191.645 rows=1000000 loops=1)
   ->  Index Scan using bar_c1_idx on bar
       (cost=0.00..15212.80 rows=500000 width=5)
       (actual time=0.012..93.921 rows=500000 loops=1)
 Total runtime: 573.966 ms
(5 rows)
```

This example replaces the simple JOIN with a LEFT JOIN. The result is a Merge Left Join node instead of a Merge Join node.

40 Let's DROP an index

```
DELETE FROM bar WHERE c1>500; DROP INDEX bar_c1_idx;
ANALYZE bar;
EXPLAIN (ANALYZE)
SELECT * FROM foo JOIN bar ON foo.c1=bar.c1;

                             QUERY PLAN
----------------------------------------------------------
 Merge Join  (cost=2240.53..2265.26 rows=500 width=42)
         (actual time=40.433..40.740 rows=500 loops=1)
   Merge Cond: (foo.c1 = bar.c1)
   ->  Index Scan using foo_c1_idx on foo
       (cost=0.00..34318.35 rows=1000000 width=37)
       (actual time=0.016..0.130 rows=501 loops=1)
   ->  Sort  (cost=2240.41..2241.66 rows=500 width=5)
        (actual time=40.405..40.432 rows=500 loops=1)
         Sort Key: bar.c1
         Sort Method: quicksort  Memory: 48kB
         ->  Seq Scan on bar
             (cost=0.00..2218.00 rows=500 width=5)
             (actual time=0.020..40.297 rows=500 loops=1)
 Total runtime: 40.809 ms
```

To prove that the Merge Join node needs sorted data sets, we drop the index on bar, and delete most of the values of bar. There is not a lof of rows in bar, so the sorting will be quick. In this example, PostgreSQL choose to do a sequential scan on bar, then sort the data with a quicksort in-memory. With the index scan on the foo table, it gets everything it needs to do a quick Merge Join.

41 We delete some more row

```
DELETE FROM foo WHERE c1>1000;
ANALYZE;
EXPLAIN (ANALYZE)
SELECT * FROM foo JOIN bar ON foo.c1=bar.c1;

                              QUERY PLAN
----------------------------------------------------------
 Nested Loop  (cost=10.79..7413.13 rows=500 width=42)
          (actual time=0.034..3.287 rows=500 loops=1)
   ->  Seq Scan on bar  (cost=0.00..8.00 rows=500 width=5)
              (actual time=0.006..0.131 rows=500 loops=1)
   ->  Bitmap Heap Scan on foo
       (cost=10.79..14.80 rows=1 width=37)
       (actual time=0.005..0.005 rows=1 loops=500)
         Recheck Cond: (c1 = bar.c1)
         ->  Bitmap Index Scan on foo_c1_idx
             (cost=0.00..10.79 rows=1 width=0)
             (actual time=0.003..0.003 rows=1 loops=500)
               Index Cond: (c1 = bar.c1)
 Total runtime: 3.381 ms
```

With more rows deleted, PostgreSQL will switch to a Nested Loop.

42 And with an empty table?

```
TRUNCATE bar;
ANALYZE;
EXPLAIN (ANALYZE)
SELECT * FROM foo JOIN bar ON foo.c1>bar.c1;

                        QUERY PLAN
---------------------------------------------------
 Nested Loop
 (cost=0.00..21405802.11 rows=776666667 width=42)
 (actual time=0.004..0.004 rows=0 loops=1)
   ->  Seq Scan on bar
       (cost=0.00..33.30 rows=2330 width=5)
       (actual time=0.002..0.002 rows=0 loops=1)
   ->  Index Scan using foo_c1_idx on foo
       (cost=0.00..5853.70 rows=333333 width=37)
       (never executed)
         Index Cond: (c1 > bar.c1)
 Total runtime: 0.049 ms
```

With no rows, PostgreSQL still thinks it has some rows, but very little rows. So it chooses a Nested Loop. The interesting thing this time is that the index scan is actually never executed because there is no row in the bar table.

43 Always used in a CROSS JOIN

```
EXPLAIN SELECT * FROM foo CROSS JOIN bar ;
                  QUERY PLAN
-------------------------------------------------------
 Nested Loop
 (cost=0.00..1641020027.00 rows=100000000000 width=42)
   ->  Seq Scan on foo
       (cost=0.00..18334.00 rows=1000000 width=37)
   ->  Materialize
       (cost=0.00..2334.00 rows=100000 width=5)
         ->  Seq Scan on bar
             (cost=0.00..1443.00 rows=100000 width=5)
```

Nested Loop is also the kind of node used for CROSS JOIN. CROSS JOIN can't be used anywhere else.

44 All the join nodes

- Nested Loop
  - for small tables
  - quick to start, slow to finish
- Merge Join
  - sort, then merge
  - slow start with no index
  - quickest on big data sets
- Hash Join
  - only equality join
  - really quick with enough memory
  - but slow start

Now, we've seen all the join nodes. PostgreSQL doesn't have anymore join nodes. The three available are enough to do each kind of joins.

45 What about inherited tables?

- Our table:

```
CREATE TABLE baz(c1 timestamp, c2 text);
```

- The first inherited table, and its data:

```
CREATE TABLE baz_2012(
  CHECK(c1 BETWEEN '2012-01-01'
               AND '2012-12-31')
  ) INHERITS(baz);

INSERT INTO baz_2012
  SELECT now()-i*interval '1 day', 'line '||i
  FROM generate_series(1, 200) AS i;
```

- And two more inherited tables
  - baz_2011, baz_2010

Inherited tables are mostly used with partitionning, but can also be used in some corner cases.

Here, we create the master table, and three inherited tables. We also populate them with 200 rows each.

46 Let's SELECT on baz

```
EXPLAIN (ANALYZE) SELECT * FROM baz WHERE c1
BETWEEN '2012-06-08' AND '2012-06-10';

                        QUERY PLAN
---------------------------------------------------------
 Result  (cost=0.00..21.00 rows=11 width=18)
   ->  Append  (cost=0.00..21.00 rows=11 width=18)
         ->  Seq Scan on baz
             (cost=0.00..0.00 rows=1 width=40)
               Filter: ((c1 >= '2012-06-08...)
                    AND (c1 <= '2012-06-10...))
         ->  Seq Scan on baz_2012 baz
             (cost=0.00..21.00 rows=10 width=16)
               Filter: ((c1 >= '2012-06-08...)
                    AND (c1 <= '2012-06-10...))
```

- Only one partition scanned
  - and the master table

With a SELECT on some days of 2012, we see that PostgreSQL only scans the master table and the table related to 2012. It doesn't scan the other tables. The constraints available on each table garantuees PostgreSQL that only bar_2012 contains data on 2012.

This will work only if the constraint_exclusion parameter is enabled. So don't disable it.

47 Let's try with an index

```
CREATE INDEX ON baz_2012(c1);
INSERT INTO baz_2012 <a few times>
EXPLAIN (ANALYZE) SELECT * FROM baz WHERE c1='2012-06-10';

                       QUERY PLAN
-----------------------------------------------------------
 Result  (cost=0.00..8.27 rows=2 width=28)
         (actual time=0.014..0.014 rows=0 loops=1)
   ->  Append  (cost=0.00..8.27 rows=2 width=28)
               (actual time=0.013..0.013 rows=0 loops=1)
         ->  Seq Scan on baz
             (cost=0.00..0.00 rows=1 width=40)
             (actual time=0.002..0.002 rows=0 loops=1)
               Filter: (c1 = '2012-06-10...)
         ->  Index Scan using baz_2012_c1_idx
                           on baz_2012 baz
             (cost=0.00..8.27 rows=1 width=16)
             (actual time=0.011..0.011 rows=0 loops=1)
               Index Cond: (c1 = '2012-06-10...)
 Total runtime: 0.052 ms
```

This optimization also works with an index. It can use some specific index on each table.

48 Let's try an aggregate

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

                  QUERY PLAN
---------------------------------------------------
 Aggregate
 (cost=20834.00..20834.01 rows=1 width=0)
   ->  Seq Scan on foo
       (cost=0.00..18334.00 rows=1000000 width=0)
```

This is the simpler query you can have with an aggregate, and probably the most used.

There's only one way to do this: a sequential scan to count all the rows. And that's what PostgreSQL does with the “Aggregate” node.

49 Let's try max()

```
DROP INDEX foo_c2_idx;
EXPLAIN (ANALYZE) SELECT max(c2) FROM foo;
                    QUERY PLAN
---------------------------------------------------------
 Aggregate
 (cost=20834.00..20834.01 rows=1 width=33)
 (actual time=257.813..257.814 rows=1 loops=1)
   ->  Seq Scan on foo
       (cost=0.00..18334.00 rows=1000000 width=33)
       (actual time=0.011..89.625 rows=1000000 loops=1)
 Total runtime: 257.856 ms
```

- Kinda ugly

max() is another really used aggregate function. There are two ways to get the maximum value of a column.

The simpler (and slower) one is to read all the rows, and keep in mind the maximum value of the visible rows.

50 Max() is better with an index

```
CREATE INDEX ON foo(c2);
EXPLAIN (ANALYZE) SELECT max(c2) FROM foo;

                      QUERY PLAN
-----------------------------------------------------------
 Result
 (cost=0.08..0.09 rows=1 width=0)
 (actual time=0.183..0.184 rows=1 loops=1)
   InitPlan 1 (returns $0)
     ->  Limit  (cost=0.00..0.08 rows=1 width=33)
                (actual time=0.177..0.178 rows=1 loops=1)
           ->  Index Only Scan Backward using foo_c2_idx
                                         on foo
               (cost=0.00..79677.29 rows=1000000 width=33)
               (actual time=0.176..0.176 rows=1 loops=1)
                 Index Cond: (c2 IS NOT NULL)
                 Heap Fetches: 1
 Total runtime: 0.223 ms
```

- Really better

The other way is to use an index. As an index has the values already sorted, you only need to get to the bigger value. It usually implies an Index Scan (backwards if you want the bigger value) and limit the reading of the index to the first value.

In 9.2, it can also use an “Index Only Scan”.

51 Works also with booleans in 9.2

```
CREATE INDEX ON foo(c2);
EXPLAIN SELECT bool_and(c2) FROM bar;

                       QUERY PLAN
----------------------------------------------------------
 Result  (cost=0.04..0.05 rows=1 width=0)
   InitPlan 1 (returns $0)
     ->  Limit  (cost=0.00..0.04 rows=1 width=1)
           ->  Index Only Scan using bar_c2_idx on bar
               (cost=0.00..21447.34 rows=500000 width=1)
                 Index Cond: (c2 IS NOT NULL)
```

This kind of micro-optimization also works with booleans with the 9.2 release. Never seen it in actual production case, but it works :)

52 Let's GROUP BY

```
DROP INDEX foo_c2_idx;
EXPLAIN (ANALYZE)
  SELECT c2, count(*) FROM foo GROUP BY c2;

                        QUERY PLAN
----------------------------------------------------------
 GroupAggregate
 (cost=172682.84..190161.00 rows=997816 width=33)
 (actual time=4444.907..5674.155 rows=999763 loops=1)
   ->  Sort
         (cost=172682.84..175182.84 rows=1000000 width=33)
         (actual time=4444.893..5368.587 rows=1000000
                                       loops=1)
           Sort Key: c2
           Sort Method: external merge  Disk: 42080kB
           ->  Seq Scan on foo
               (cost=0.00..18334.00 rows=1000000 width=33)
               (actual time=0.014..147.936 rows=1000000
                                         loops=1)
 Total runtime: 5715.402 ms
```

Doing a count(*) for each distinct value of a column can be done in two ways. The simpler way is to sort the data, and group them according to the values.

In this example, the planner does not have any index to do the sorting, so it gets back to a sequential scan, and then a sort. The sort is too big to stay in memory, so it uses an external sort. After the sort, it simply needs to group all non distinct values, and computes the count. What really takes time here is the sort (5.3 seconds for a 5.6 seconds query). Having a bigger work_mem could help, having an index would definitely help.

53 Having a bigger work_mem helps

```
SET work_mem TO '200MB';
EXPLAIN (ANALYZE)
  SELECT c2, count(*) FROM foo GROUP BY c2;

                        QUERY PLAN
---------------------------------------------------------
 HashAggregate
 (cost=23334.00..33312.16 rows=997816 width=33)
 (actual time=581.653..976.936 rows=999763 loops=1)
   ->  Seq Scan on foo
       (cost=0.00..18334.00 rows=1000000 width=33)
       (actual time=0.021..96.977 rows=1000000 loops=1)
 Total runtime: 1019.810 ms
```

With a bigger work_mem, PostgreSQL switched to an HashAggregate because the hashing can stay in memory.

54 Or an index

```
RESET work_mem;
CREATE INDEX ON foo(c2);a
EXPLAIN (ANALYZE)
  SELECT c2, count(*) FROM foo GROUP BY c2;

                       QUERY PLAN
-----------------------------------------------------------
 GroupAggregate
 (cost=0.00..92155.45 rows=997816 width=33)
 (actual time=0.108..1439.023 rows=999763 loops=1)
   ->  Index Only Scan using foo_c2_idx on foo
       (cost=0.00..77177.29 rows=1000000 width=33)
       (actual time=0.095..1043.090 rows=1000000 loops=1)
         Heap Fetches: 1000000
 Total runtime: 1475.775 ms
```

An index doesn't help hashing. So, with an index and a small work_mem, we're back to the

GroupAggregate. But this time, it uses an index scan instead of a sequential scan followed by a sort.

55 All the aggregate nodes

- Aggregate
- GroupAggregate
- HashAggregate

These are all the aggregate nodes available for the planner.

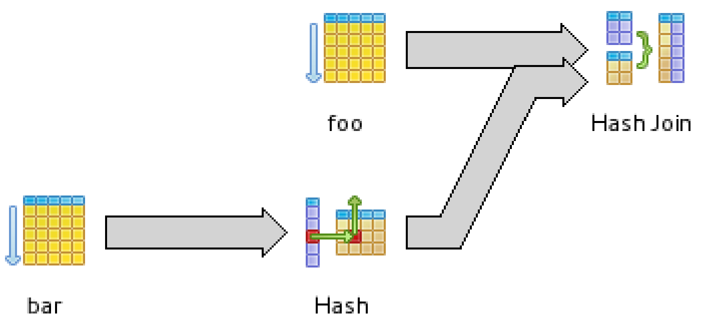
56 Tools

- Not many tools
  - but some useful ones
- pgAdmin
- explain.depesz.com
- pg_stat_plans

Unfortunately, we don't have many tools to help us with execution plans. We have pgAdmin with its graphical explain plan, explain.depesz.com with its colorful tables, and pg_stat_plans.

57 pgAdmin

- Nice query tool



- Graphical EXPLAIN
- Better explains the first node to start, and how they are connected
- Bigger arrows means lots of data

pgAdmin is a really good admin tool. Its query tool has many features, one of them being a graphical explain. It's probably the only admin tool with such a feature.

It really helps to understand which node is the first to be executed, and how they are all connected. The size of the arrows has a meaning: the more data you got, the bigger they are.

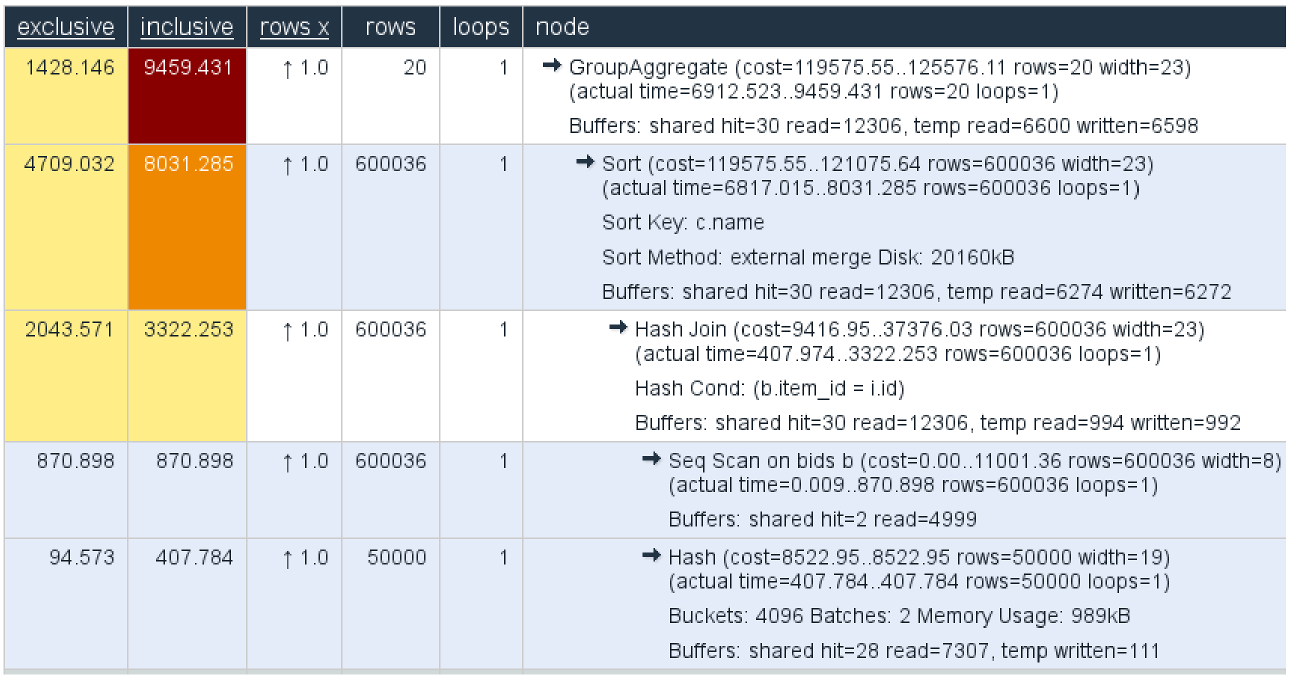
58 explain.depesz.com

- Even better than pgAdmin's graphical EXPLAIN
- You can install it locally
  - One perl module
  - One mojolicious webserver
- Gives you the exclusive time to each node
  - Probably the best idea for this tool
- Colors to quickly spot the issue in the plan

explain.depesz.com is a great website where you can post the explain plan you have from your EXPLAIN statement. The most interesting informations is the exclusive time for each node. You can do it manually but it's better when it's done for you. Moreover, there are highlighting colors that help you to find the weak spots in your explain plan.

If you don't want to post your explain plans to avoid disclosing any important informations, you can also install it on one of your servers. You have one Perl module to install, and a mojolicius webserver.

59 explain.depesz.com screenshot



exclusive	inclusive	rows x	rows	loops	node
1428.146	9459.431	↑ 1.0	20	1	→ GroupAggregate (cost=119575.55..125576.11 rows=20 width=23) (actual time=6912.523..9459.431 rows=20 loops=1) Buffers: shared hit=30 read=12306, temp read=6600 written=6598
4709.032	8031.285	↑ 1.0	600036	1	→ Sort (cost=119575.55..121075.64 rows=600036 width=23) (actual time=6817.015..8031.285 rows=600036 loops=1) Sort Key: c.name Sort Method: external merge Disk: 20160kB Buffers: shared hit=30 read=12306, temp read=6274 written=6272
2043.571	3322.253	↑ 1.0	600036	1	→ Hash Join (cost=9416.95..37376.03 rows=600036 width=23) (actual time=407.974..3322.253 rows=600036 loops=1) Hash Cond: (b.item_id = i.id) Buffers: shared hit=30 read=12306, temp read=994 written=992
870.898	870.898	↑ 1.0	600036	1	→ Seq Scan on bids b (cost=0.00..11001.36 rows=600036 width=8) (actual time=0.009..870.898 rows=600036 loops=1) Buffers: shared hit=2 read=4999
94.573	407.784	↑ 1.0	50000	1	→ Hash (cost=8522.95..8522.95 rows=50000 width=19) (actual time=407.784..407.784 rows=50000 loops=1) Buckets: 4096 Batches: 2 Memory Usage: 989kB Buffers: shared hit=28 read=7307, temp written=111

60 pg_stat_plans

- New extension from Peter Geoghegan
- Way better than the usual pg_stat_statements
  - You can now have the query plan

pg_stat_plans is a quite new extension, as it was released last week. It's written by Peter Geoghegan, from 2ndQuadrant.

This extension goes a bit farther than pg_stat_statements because it can give you the explain plan of your last executed queries.

61 pg_stat_plans example

```
SELECT pg_stat_plans_explain(planid, userid, dbid),
  planid, last_startup_cost, last_total_cost
FROM pg_stat_plans
WHERE planid = 3002758564;
-[ RECORD 1 ]---------+---------------------------------
pg_stat_plans_explain | GroupAggregate
                        (cost=0.00..92155.45 rows=997816
                         width=33)
                      |   ->  Index Only Scan
                              using foo_c2_idx
                              on foo
                              (cost=0.00..77177.29
                               rows=1000000
                               width=33)
planid                | 3002758564
last_startup_cost     | 0
last_total_cost       | 92155.450671524
```

On this example, we only grabbed the plan, its startup cost, and its total cost. You can have more informations like had_our_search_path and query_valid.

You should read the documentation on github: https://github.com/2ndQuadrant/pg_stat_plans

62 By the way

- If you want to know more about query logging and this new extension
  - go see Greg Smith's and Peter Geoghegan's talk
- “Beyond Query Logging”
  - Friday, 9:30am to 10:20am
  - Room Seine

63 Conclusion

- PostgreSQL planner is really great
- The EXPLAIN gives you a lot of clues to understand
  - what the executor is doing
  - why it is doing it
- With some better understanding, you may be able to fix some of your queries

64 Question?

- If I don't have time to take any question:
  - I'm here till the end of the event
  - guillaume@lelarge.info
- But if I have, go ahead :)