Query Performance Tuning

02 June 2015

14:05

Sooner or later, we’ll all face a query that takes just a bit longer to execute than what we have patience for. The best and easiest fix to a sluggish query is to perfect the underlying SQL, followed by adding indexes, and updating planner statistics.

To guide you in these pursuits, Postgres comes with a built-in explainer that informs you how the query planner is going to execute your SQL.

Armed with your knack for writing flawless SQL, your instinct to sniff out useful indexes, and the insight of the explainer, you should have no trouble getting your queries to run as fast as what your hardware budget will allow.

**EXPLAIN and EXPLAIN ANALYZE**

The easiest tool for targeting query performance problems is using the EXPLAIN and EXPLAIN ANALYZE commands.

These have been around ever since the early years of PostgreSQL.

Since then it has matured into a full-blown tool capable of reporting highly detailed information about the query execution. Along the way, it added to its number of output formats.

In PostgreSQL 9.0+, you can even dump the output to XML or JSON.

Perhaps the most exciting enhancement for the common user came when pgAdmin introduced graphical EXPLAIN several years back.

With a hard and long stare, you can identify where the bottlenecks are in your query, which tables are missing indexes, and whether the path of execution took an unexpected turn.

EXPLAIN will give you just an idea of how the planner intends to execute the query without running it.

<http://www.postgresql.org/docs/current/static/sql-explain.html>

The most critical part of the display is the estimated statement execution cost, which is the planner's guess at how long it will take to run the statement (measured in cost units that are arbitrary, but conventionally mean disk page fetches).

Actually two numbers are shown: the start-up cost before the first row can be returned, and the total cost to return all the rows. For most queries the total cost is what matters,

EXPLAIN ANALYZE will actually execute the query and give you comparative analysis of expected versus actual.

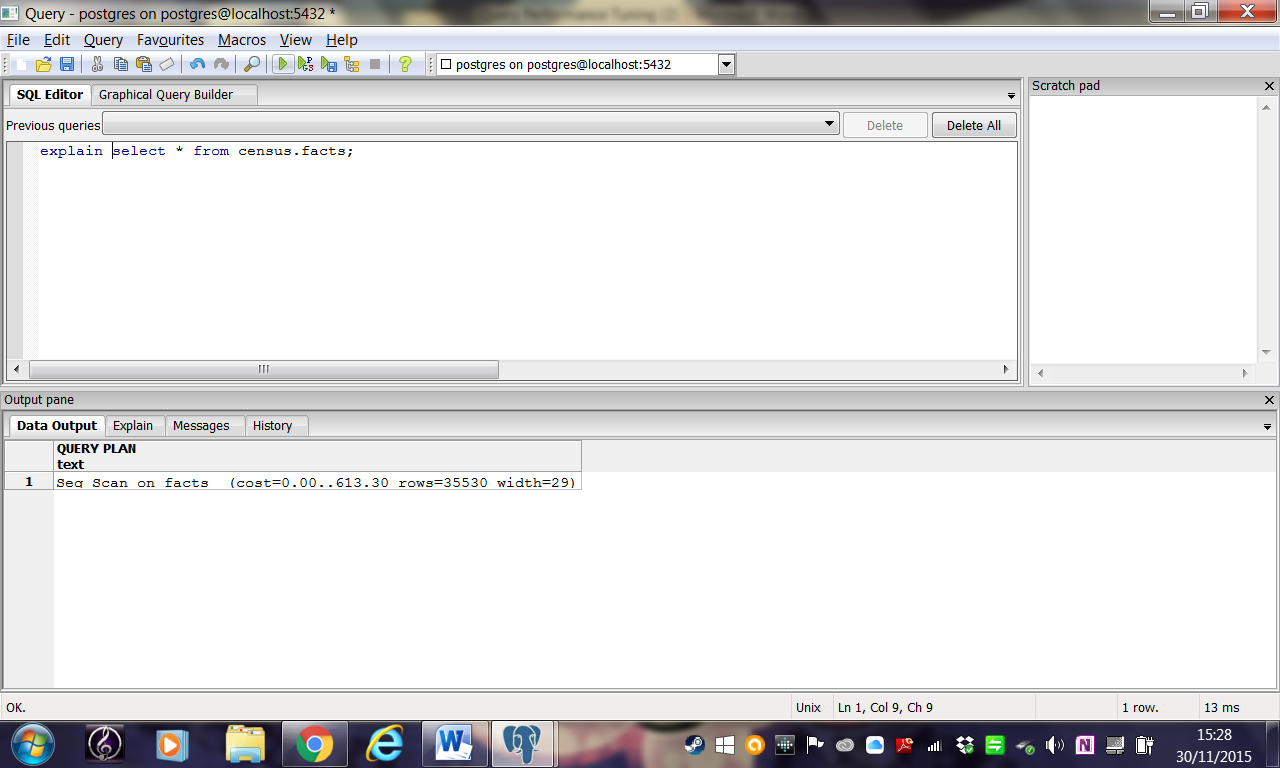
Then actual run time statistics are added to the display, including the total elapsed time expended within each plan node (in milliseconds) and the total number of rows it actually returned.

This is useful for seeing whether the planner's estimates are close to reality.

For the non-graphical version of EXPLAIN, simply preface your SQL with the EXPLAIN or EXPLAIN ANALYZE.

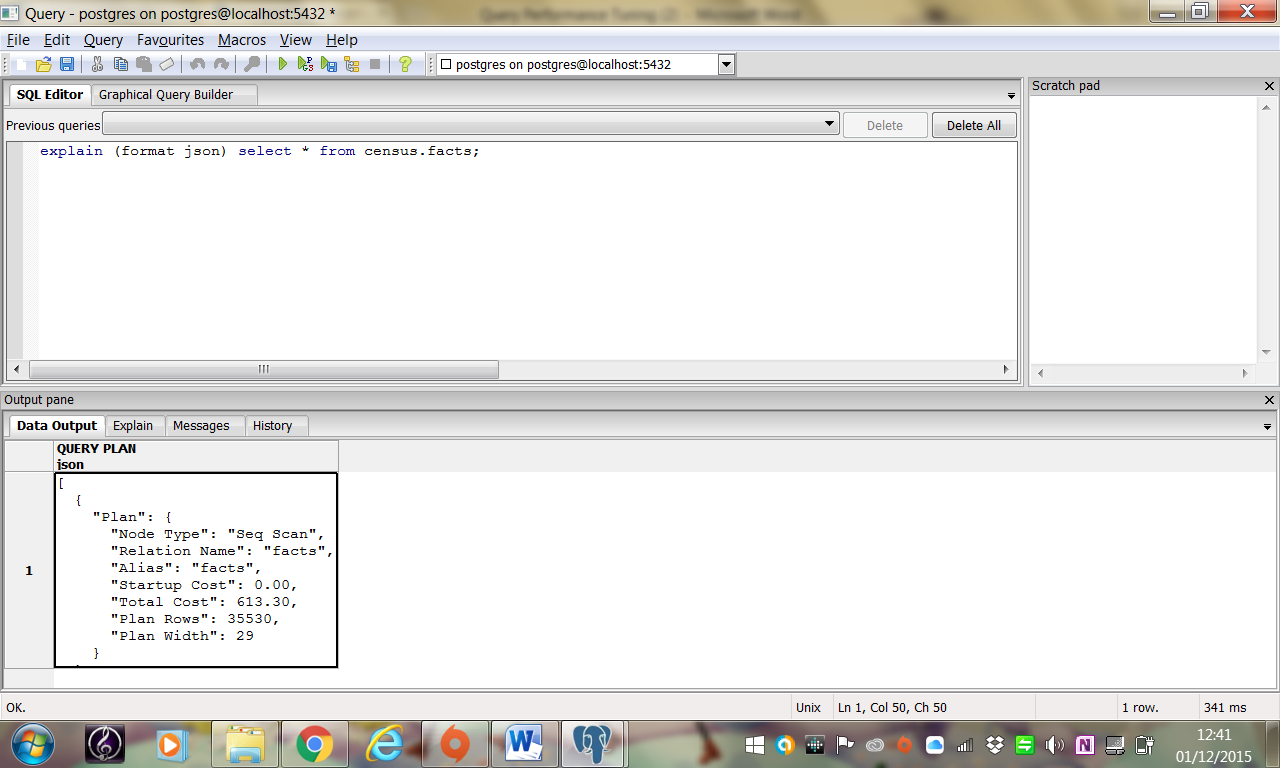
Example:

Load the census database we used in the last lecture again

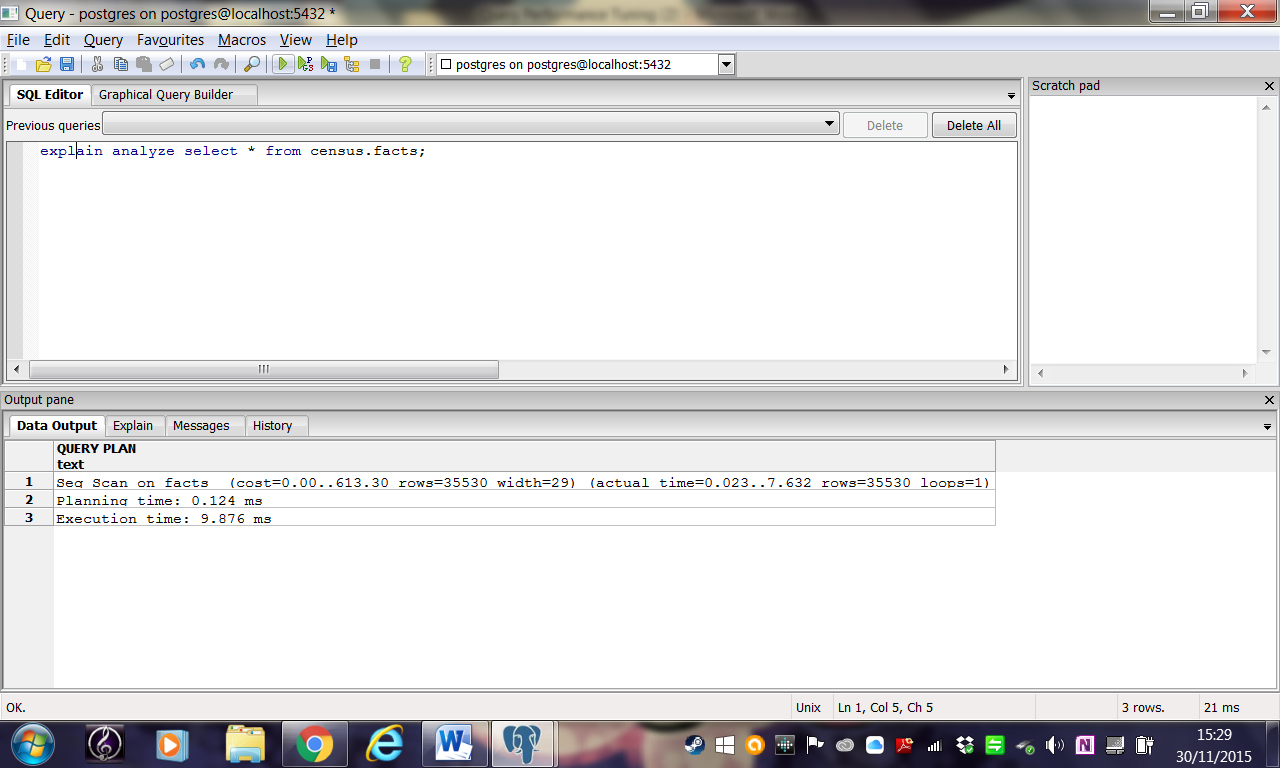


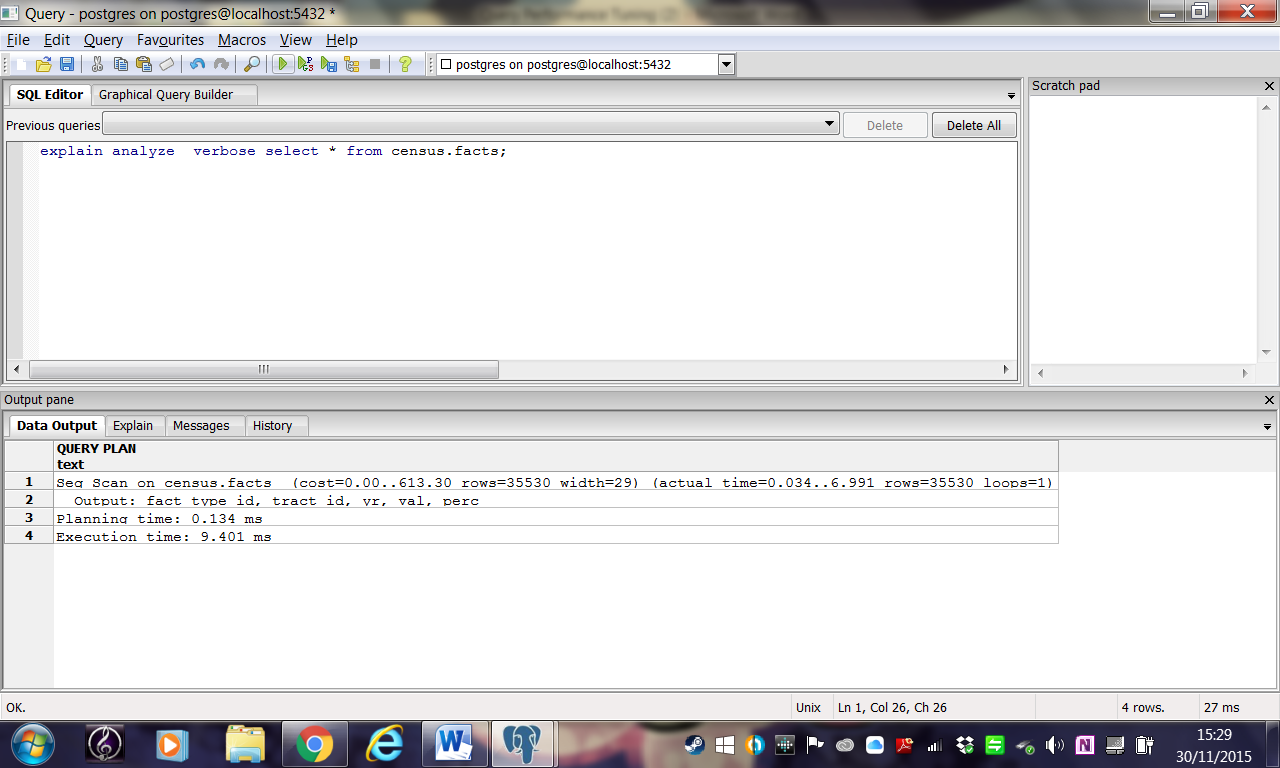
OR

explain (format json) select \* from census.facts;



Note width is width of each row in bytes.

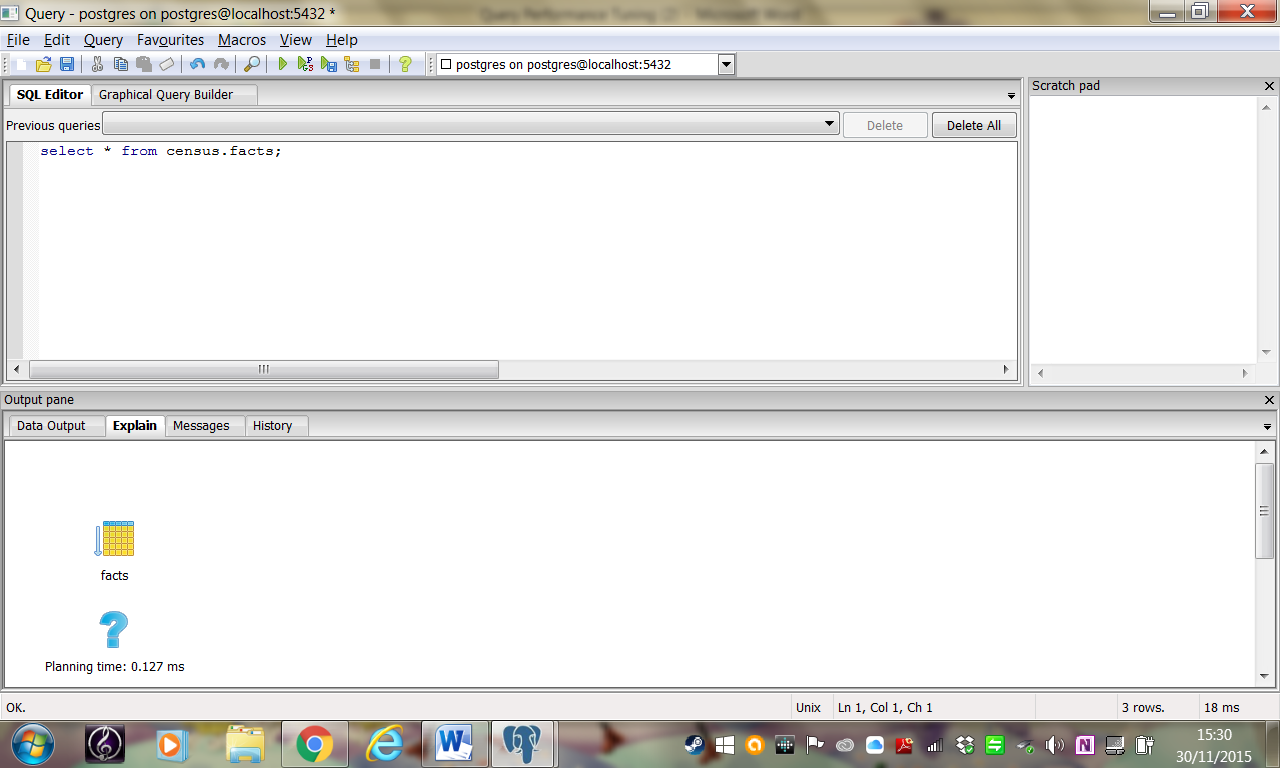




VERBOSE is an optional modifier that will give you details down to the columnar level.

You launch graphical EXPLAIN via pgAdmin.

Compose the query as usual, but instead of executing it, choose EXPLAIN or EXPLAIN ANALYZE from the drop down menu.



It goes without saying that to use graphical explain, you’ll need more than a command prompt.

 Let’s try an example, we’ll first use the EXPLAIN ANALYZE command.

**Example Explain analyse**

ANALYZE collects statistics about the contents of tables in the database, and stores the results in the [pg\_statistic](http://www.postgresql.org/docs/current/static/catalog-pg-statistic.html) system catalog.

Subsequently, the query planner uses these statistics to help determine the most efficient execution plans for queries.

With no parameter, ANALYZE examines every table in the current database.

With a parameter, ANALYZE examines only that table.

**Example: A query to select all rental accommodation in the census:**

SELECT fact\_subcats[4], category, yr

FROM census.lu\_fact\_types JOIN census.facts

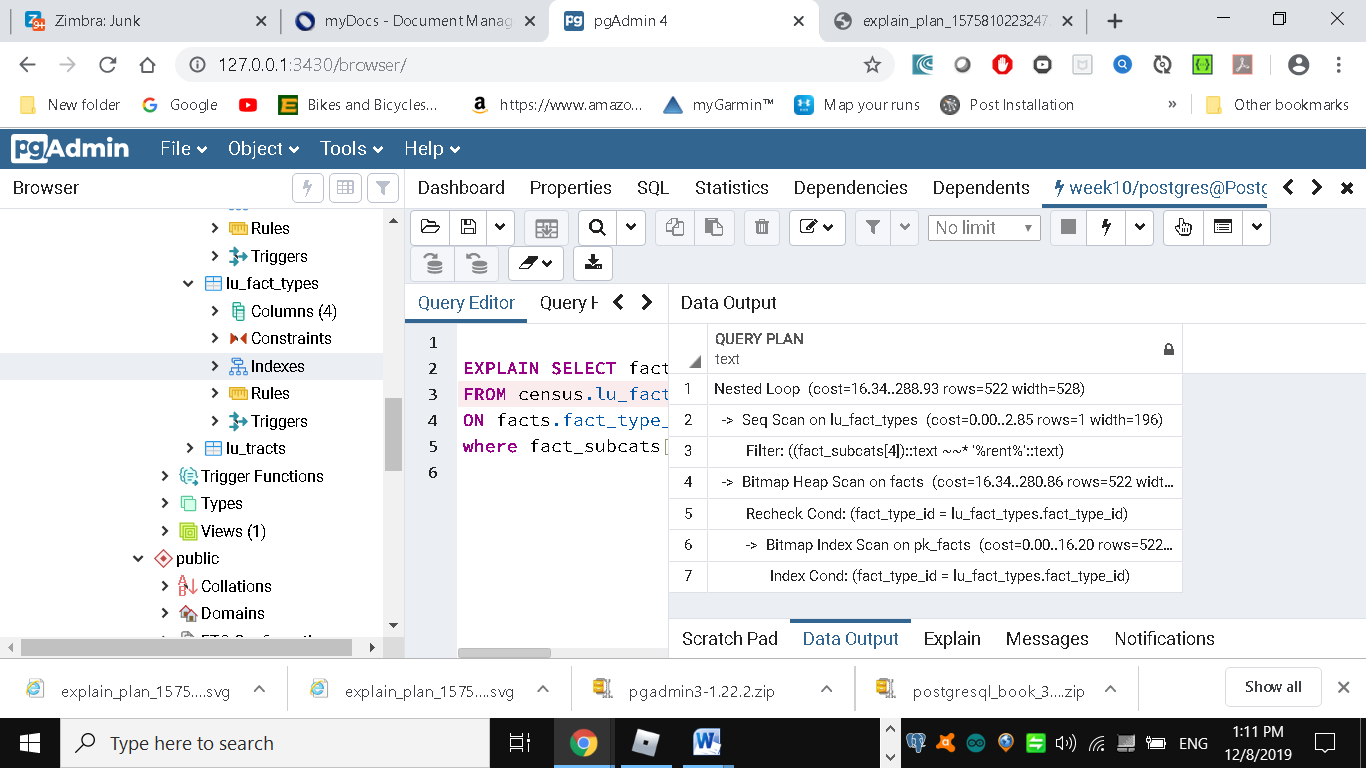
ON facts.fact\_type\_id = lu\_fact\_types.fact\_type\_id

where fact\_subcats[4] ILIKE '%rent%';

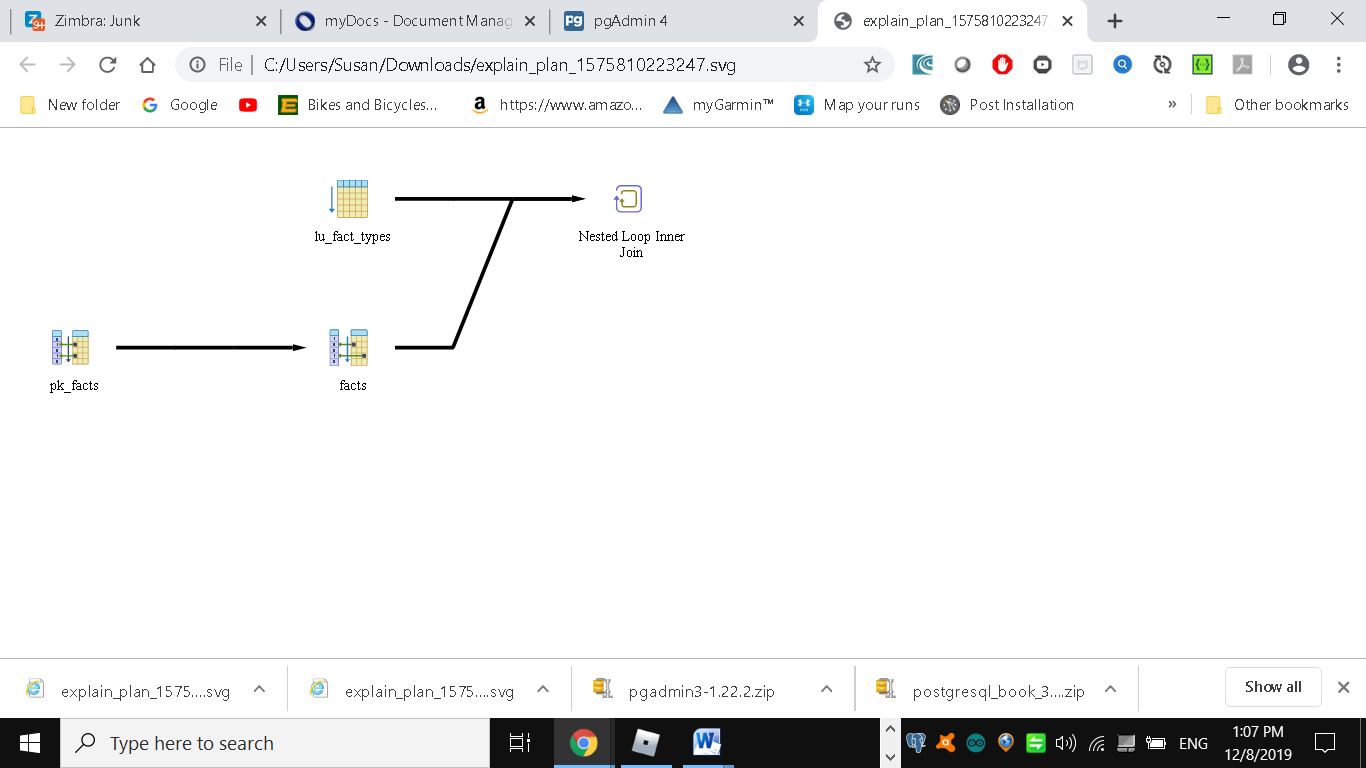
The output is shown below

**Example EXPLAIN ANALYZE output**

|  |
| --- |
| Nested Loop (cost=16.34..288.93 rows=522 width=528) (actual time=0.538..12.280 rows=4080 loops=1) |
| -> Seq Scan on lu\_fact\_types (cost=0.00..2.85 rows=1 width=196) (actual time=0.223..1.502 rows=20 loops=1) |
| Filter: ((fact\_subcats[4])::text ~~\* '%rent%'::text) |
| Rows Removed by Filter: 48 |
| -> Bitmap Heap Scan on facts (cost=16.34..280.86 rows=522 width=8) (actual time=0.112..0.220 rows=204 loops=20) |
| Recheck Cond: (fact\_type\_id = lu\_fact\_types.fact\_type\_id) |
| Heap Blocks: exact=51 |
| -> Bitmap Index Scan on pk\_facts (cost=0.00..16.20 rows=522 width=0) (actual time=0.086..0.086 rows=204 loops=20) |
| Index Cond: (fact\_type\_id = lu\_fact\_types.fact\_type\_id) |
| Planning Time: 3.847 ms |
| Execution Time: 13.335 ms |



If reading the output is giving you a headache, here’s the graphical EXPLAIN:



 Before leaving the section on EXPLAIN, we must pay homage to a new [online](http://explain.depesz.com/) EXPLAIN tool created by Hubert “depesz” Lubaczewski (<https://explain.depesz.com/>)

Using his site, you can copy and paste the text output of your EXPLAIN, and it will show you a beautifully formatted stats report as shown below

In the HTML tab, a nicely reformatted color-coded table of the plan will be displayed, with problem areas highlighted in vibrant colors, as shown below

Although the HTML table below provides much the same information as our plain-text plan, the color coding and breakout of numbers makes it easier to see that our actual values are far off from the estimated numbers. This suggests that our planner stats are probably not up to date.

**Online EXPLAIN stats**

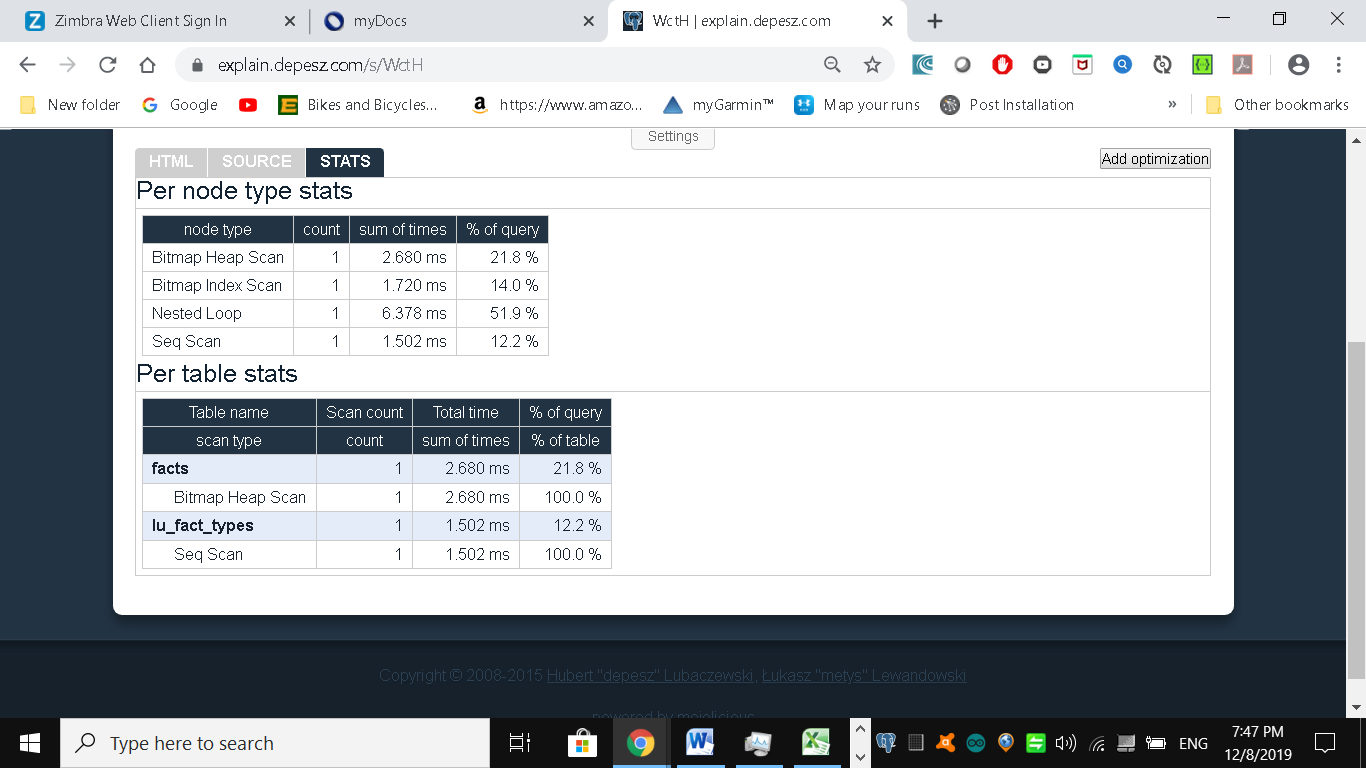
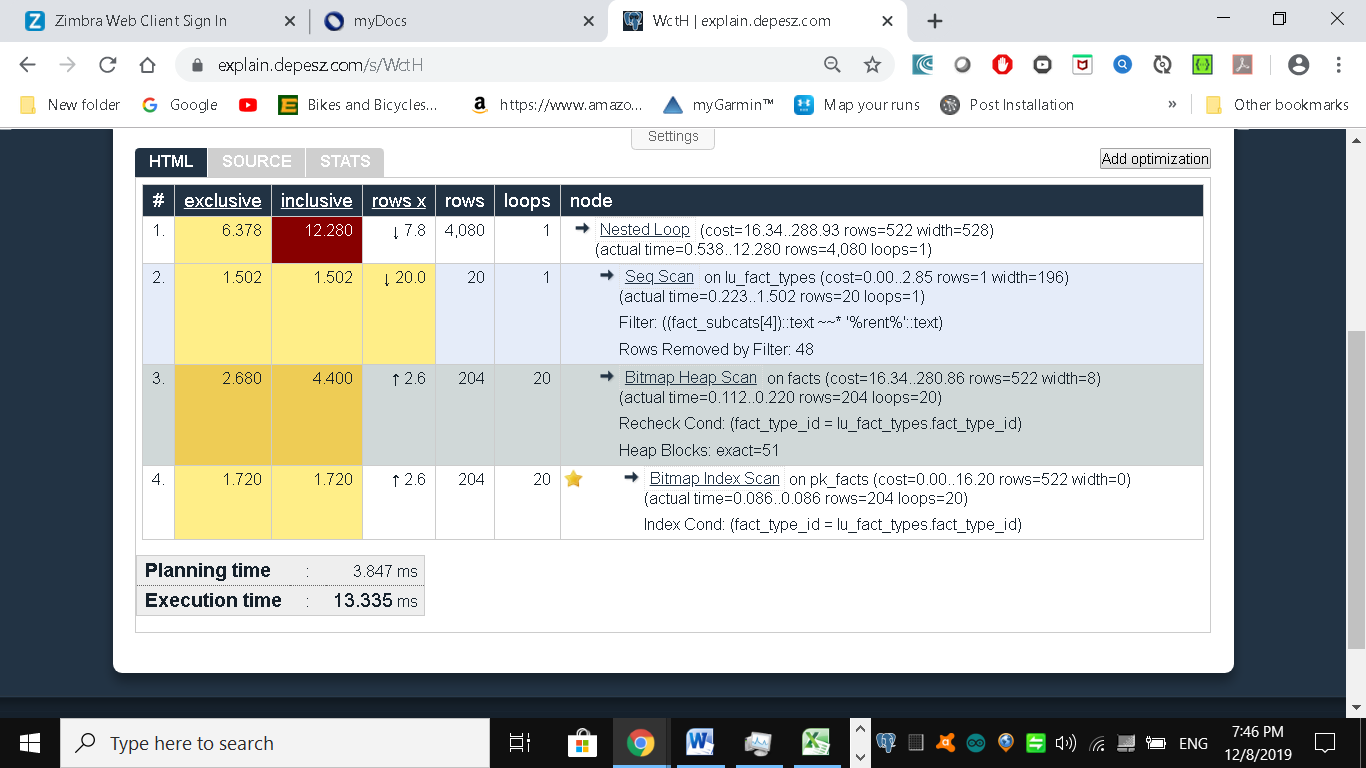


Figure Online EXPLAIN HTML table



Writing Better Queries

The best and easiest way to improve query performance is to start with well-written queries.

Four out of five queries we encounter are not written as efficiently as they could be. There appears to be two primary causes for all this bad querying.

First, we see people reuse SQL patterns without thinking.

For example, if they successfully write a query using a left join, they will continue to use left join when incorporating more tables instead of considering the sometimes more appropriate inner join.

Unlike other programming languages, the SQL language does not lend itself well to blind reuse.

Second, people don’t tend to keep up with the latest developments in their dialect of SQL.

If a PostgreSQL user is still writing SQL as if he still had an early version, he’d be oblivious to all the syntax-saving (and mind-saving) addendums that have come along.

Writing efficient SQL takes practice. There’s no such thing as a wrong query as long as you get the expected result, but there is such a thing as a slow query.

In this section, we’ll go over some of the common mistakes we see people make. Although this is about PostgreSQL, our constructive recommendations are applicable to other relational databases as well.

Overusing Subqueries in SELECT

A classic newbie mistake is to think about a query in independent pieces and then trying to gather them up all in one final SELECT.

Unlike conventional programming, SQL doesn’t take kindly to the idea of blackboxing where you can write a bunch of subqueries independently and then assemble them together mindlessly to get the final result.

You have to give your query the holistic treatment. How you piece together data from different views and tables is every bit as important as how you go about retrieving the data in the first place.

Example Overusing subqueries

SELECT tract\_id

,(SELECT COUNT(\*) FROM census.facts As F

WHERE F.tract\_id = T.tract\_id) As num\_facts

,(SELECT COUNT(\*) FROM census.lu\_fact\_types As Y

WHERE Y.fact\_type\_id IN (SELECT fact\_type\_id

FROM census.facts F WHERE F.tract\_id = T.tract\_id)) As num\_fact\_types

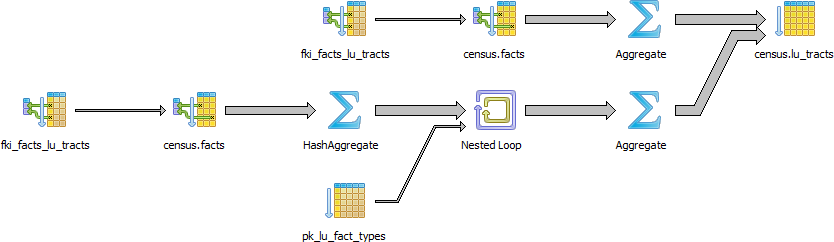
FROM census.lu\_tracts As T;

EXPLAIN ANALYZE gives the following:

|  |
| --- |
| Seq Scan on lu\_tracts t (cost=0.00..245267.10 rows=204 width=28) (actual time=8.636..1915.133 rows=204 loops=1) |
| SubPlan 1 |
| -> Aggregate (cost=702.18..702.19 rows=1 width=8) (actual time=8.364..8.364 rows=1 loops=204) |
| -> Seq Scan on facts f (cost=0.00..702.13 rows=24 width=0) (actual time=0.272..8.309 rows=68 loops=204) |
| Filter: ((tract\_id)::text = (t.tract\_id)::text) |
| Rows Removed by Filter: 35462 |
| SubPlan 2 |
| -> Aggregate (cost=500.06..500.07 rows=1 width=8) (actual time=1.002..1.002 rows=1 loops=204) |
| -> Nested Loop Semi Join (cost=0.29..500.00 rows=24 width=0) (actual time=0.060..0.928 rows=68 loops=204) |
| -> Seq Scan on lu\_fact\_types y (cost=0.00..2.68 rows=68 width=4) (actual time=0.008..0.026 rows=68 loops=204) |
| -> Index Only Scan using pk\_facts on facts f\_1 (cost=0.29..7.31 rows=1 width=4) (actual time=0.012..0.012 rows=1 loops=13872) |
| Index Cond: ((fact\_type\_id = y.fact\_type\_id) AND (tract\_id = (t.tract\_id)::text)) |
| Heap Fetches: 13872 |
| Planning Time: 0.892 ms |
| Execution Time: 1915.392 ms |

The graphical EXPLAIN plan for this query is shown below

Figure Graphical EXPLAIN plan of long-winded subselects (pgAdmin 3)



Or in latest version of PostgreSQL (pgAdmin 4)

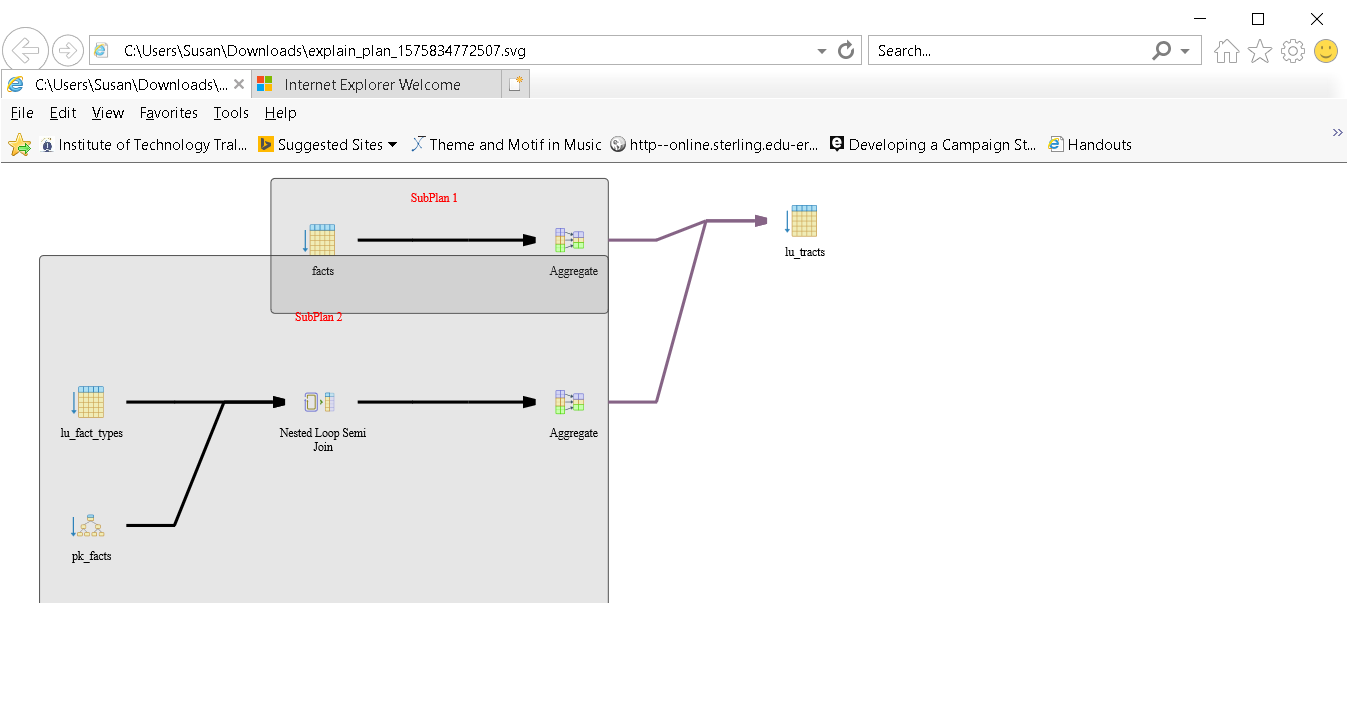
We’ll also show you the output from using the online EXPLAIN at <http://explain.depesz.com>.

Figure Online EXPLAIN of overusing subqueries

Machine generated alternative text: HTML 
exclusive 
10.709 
63.554 
153.712 
47.296 
59.120 
314.814 
155.190 
141.888 
44.340 
301.512 
inclusive 
1292135 
264.562 
201 _008 
47 _296 
1016.864 
957 _ 744 
341 _418 
186.228 
44 _ 340 
301 _512 
rows x 
0 
t 1.0 
t 1.0 
t 1.0 
t 1.0 
t 1.0 
680 
t 1.0 
t 1.0 
t 1.0 
rows 
1478 
68 
68 
68 
68 
68 
68 
loops 
1478 
1478 
1478 
1478 
1478 
1478 
1478 
1478 
100504 
node 
Seq Scan on lu tracts t 
(costzO.OO..615535.37 rows-1478 width-12) (actual tin-u 
SubPlan (forSeq Scan) 
Aggregate (costz207 _86_.207.87 rows-zl widthz0) 
Bitmap Heap Scan on facts f 
(costz4.79.207 rowsz68 widthz0) (actual time 
Recheck Cond: ((tract (ttract id)::text) 
Bitmap Index Scan on fki facts lu tracts 
(costz0.00._4.78 rowsz68 widthz0) (actual tim 
Index Cond: ((tract (ttract 
Aggregate rows-zl widthz0) 
Nested Loop 
(costz207.86..208.39 rowsz68 width-O) (actual ti 
HashAggregate 
(costz207.86_207.87 rowszl widthz4) (actua 
Bitmap Heap Scan on facts f 
(costz4.79._20789 rowsz68 widthz4) (acti 
Recheck Cond: ((tract (ttract 
Bitmap Index Scan on fki facts Iu trat 
rowsz68 widthz0) (ac 
Index Condi ((tract (ttract i/ 
Index Scan using pk lu fact_types on lu fact 
(costz0.00._0.50 rows: 1 (actual time 
Index Cond: (fact_type_id ffact_type_id) 

The query can be more efficiently written as shown below (note there are tools to help you rewrite queries)

This version of the query is not only shorter, but faster than the prior one. If you have even more rows or weaker hardware, the difference would be even more pronounced.

SELECT T.tract\_id, COUNT(f.fact\_type\_id) As num\_facts, COUNT(DISTINCT fact\_type\_id) As

num\_fact\_types

FROM census.lu\_tracts As T LEFT JOIN census.facts As F ON T.tract\_id = F.tract\_id

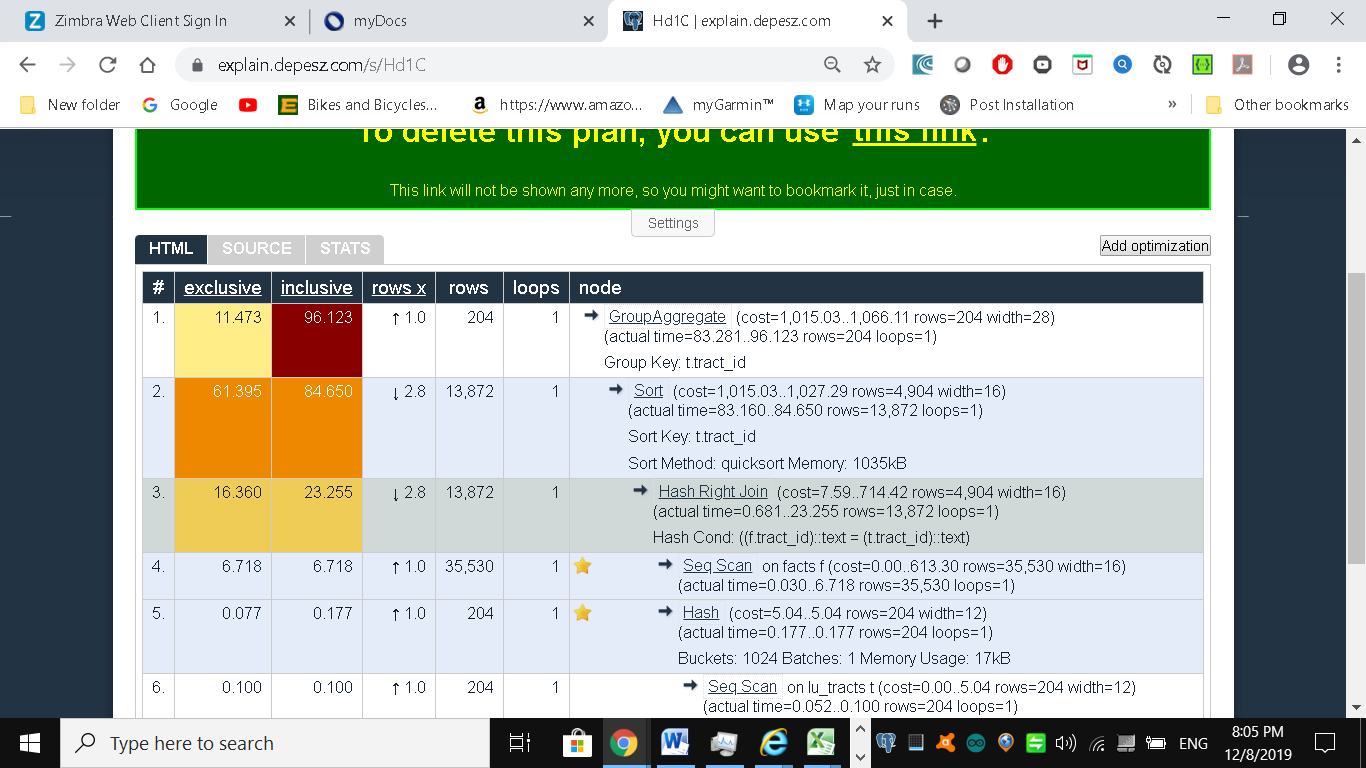
GROUP BY T.tract\_id;

 Explain Analyze gives the following:

|  |
| --- |
| GroupAggregate (cost=1015.03..1066.11 rows=204 width=28) (actual time=83.281..96.123 rows=204 loops=1) |
| Group Key: t.tract\_id |
| -> Sort (cost=1015.03..1027.29 rows=4904 width=16) (actual time=83.160..84.650 rows=13872 loops=1) |
| Sort Key: t.tract\_id |
| Sort Method: quicksort Memory: 1035kB |
| -> Hash Right Join (cost=7.59..714.42 rows=4904 width=16) (actual time=0.681..23.255 rows=13872 loops=1) |
| Hash Cond: ((f.tract\_id)::text = (t.tract\_id)::text) |
| -> Seq Scan on facts f (cost=0.00..613.30 rows=35530 width=16) (actual time=0.030..6.718 rows=35530 loops=1) |
| -> Hash (cost=5.04..5.04 rows=204 width=12) (actual time=0.177..0.177 rows=204 loops=1) |
| Buckets: 1024 Batches: 1 Memory Usage: 17kB |
| -> Seq Scan on lu\_tracts t (cost=0.00..5.04 rows=204 width=12) (actual time=0.052..0.100 rows=204 loops=1) |
| Planning Time: 0.940 ms |
| Execution Time: 96.776 ms |

Figure: Graphical explain plan of re-written subqueries

Machine generated alternative text: 25 55 dno 」 
0 「 5 」 
U24 nl 」 
U24 ' 

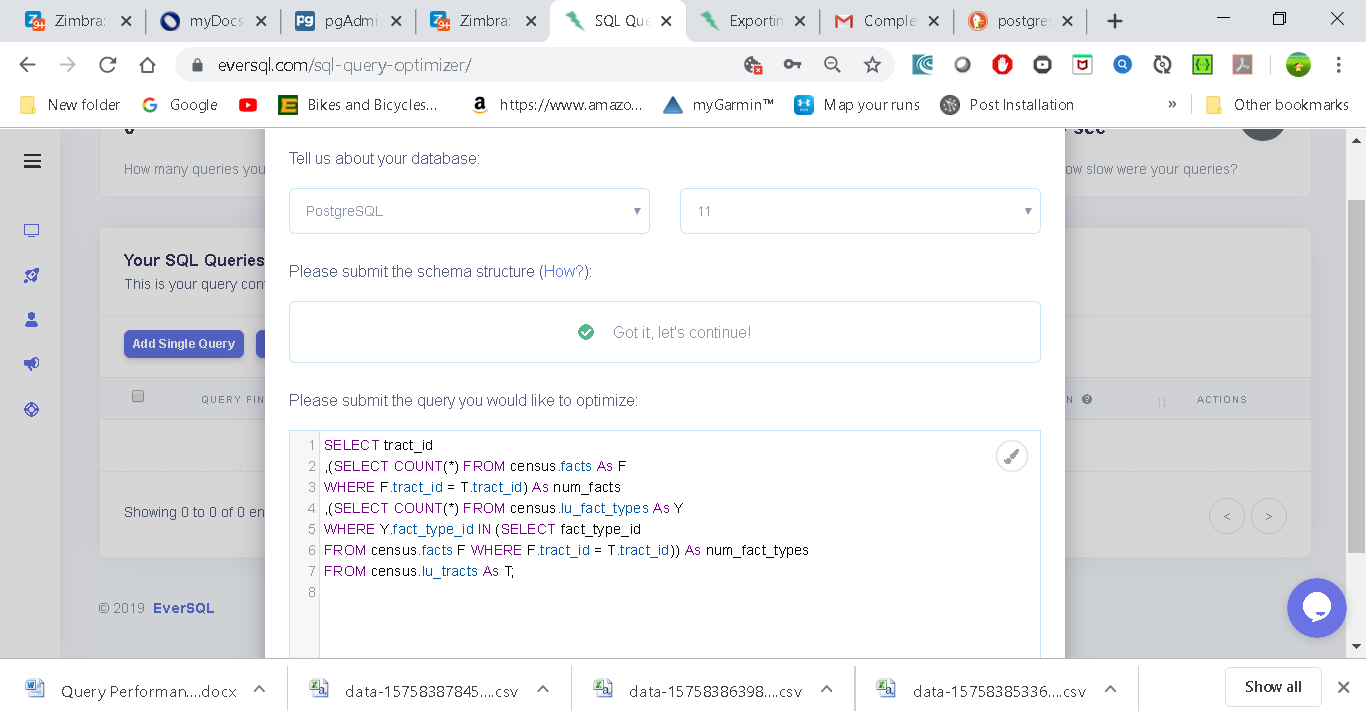


Keep in mind that we’re not asking you to avoid subqueries. We’re simply asking you to use them judiciously. When you do use them, be sure to pay extra attention on how you combine them into the main query.

Finally, remember that a subquery should try to work with the the main query, not independent of it.

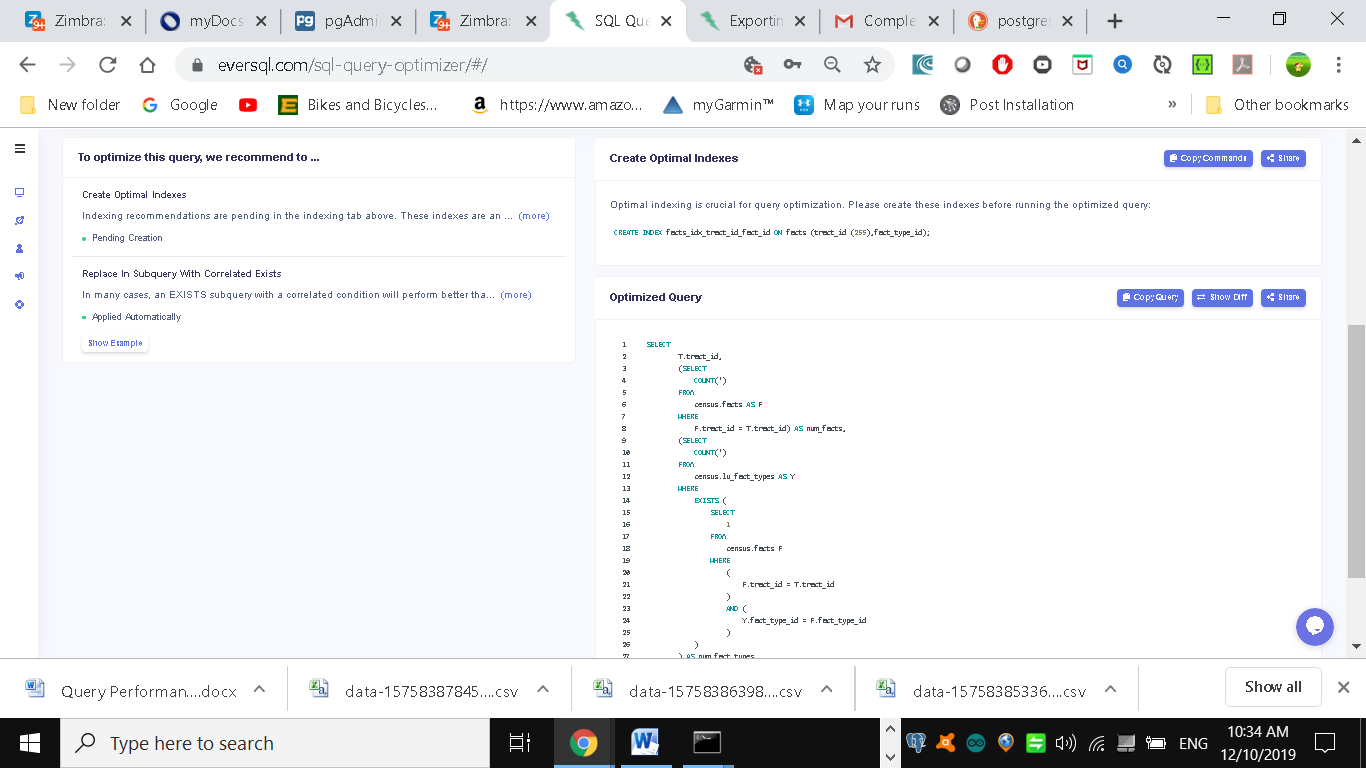
We are not all experts at writing optimal SQL – this takes years of practice. There are lots of tools to automatically optimize SQL queries. One such tool is available online at <https://www.eversql.com>

Try the SQL Optimizer at <https://www.eversql.com> to see how it rewrites the first query above (the one with too many sub sections) You will need to register with them and then upload the schema for the census database (pg\_dump -s databasename > schema.sql) (on the X drive as census.sql), along with the SQL statement.

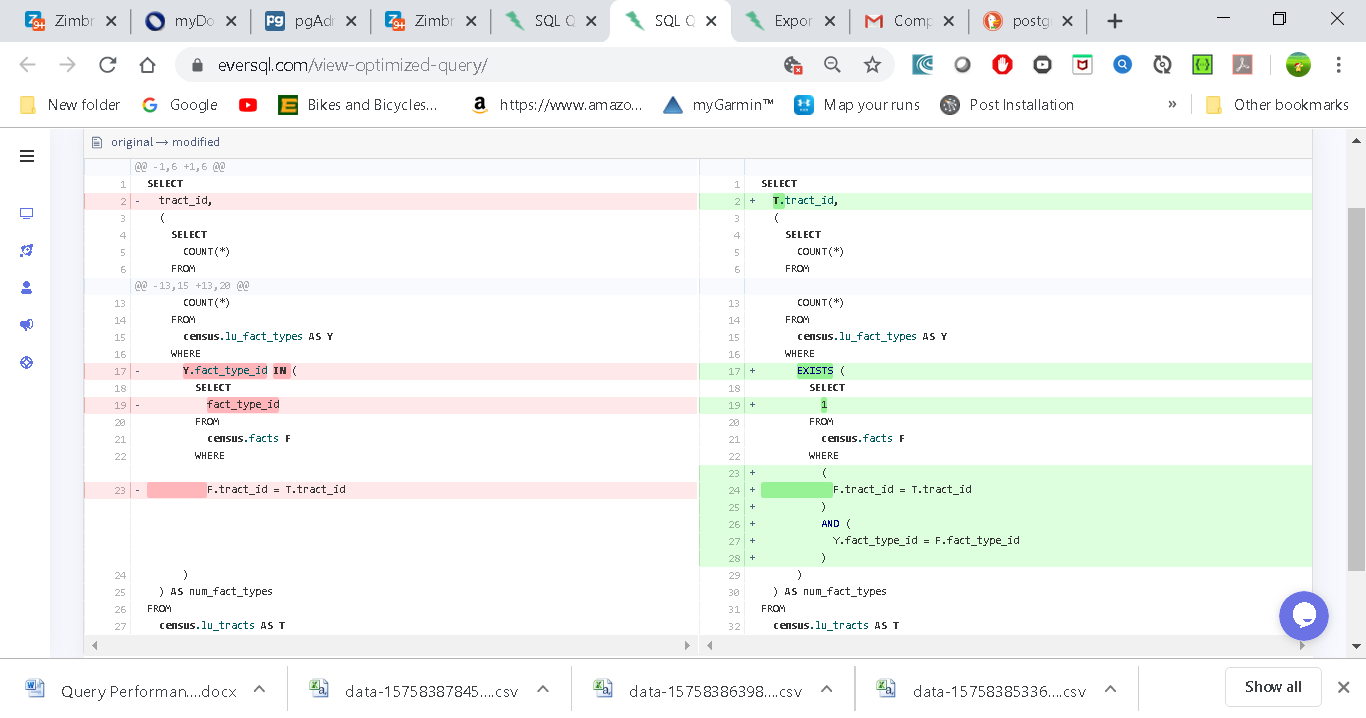


The results:

They recommend creating an index and replacing the sub-queries:



Side by side comparison of queries:



If you wish you could try out the results and see if the execution times increase:

CREATE INDEX facts\_idx\_tract\_id\_fact\_id ON facts (tract\_id (255),fact\_type\_id);

SELECT

T.tract\_id,

(SELECT

COUNT(\*)

FROM

census.facts AS F

WHERE

F.tract\_id = T.tract\_id) AS num\_facts,

(SELECT

COUNT(\*)

FROM

census.lu\_fact\_types AS Y

WHERE

EXISTS (

SELECT

1

FROM

census.facts F

WHERE

(

F.tract\_id = T.tract\_id

)

AND (

Y.fact\_type\_id = F.fact\_type\_id

)

)

) AS num\_fact\_types

FROM

census.lu\_tracts AS T

**Avoid SELECT \***

SELECT \* is wasteful. It’s akin to printing out a 1,000-page document when you only need ten pages.

Besides the obvious downside of adding to network traffic, there are two other drawbacks that you might not think of.

First, PostgreSQL stores large blob and text objects using TOAST (The Oversized-Attribute Storage Technique). TOAST maintains side tables for PostgreSQL to store this extra data.

The larger the data the more internally divided up it is. So retrieving a large field means that TOAST must assemble the data across different rows across different tables.

Imagine the extra processing should your table contain text data the size of War and Peace and you perform an unnecessary SELECT \*.

Second, when you define views, you often will include more columns than you’ll need.

PostgreSQL is smart enough that you can have all the columns you want in your view definition and even include complex calculations or joins without incurring penalty as long as you don’t ask for them.

You could easily end up pulling every column out of all joined tables inside the view.

To drive home our point, let’s wrap our census in a view and use the slow subselect example we proposed:

CREATE OR REPLACE VIEW vw\_stats AS

SELECT tract\_id

,(SELECT COUNT(\*) FROM census.facts As F WHERE F.tract\_id = T.tract\_id) As num\_facts

,(SELECT COUNT(\*) FROM census.lu\_fact\_types As Y

WHERE Y.fact\_type\_id

IN (SELECT fact\_type\_id FROM census.facts F

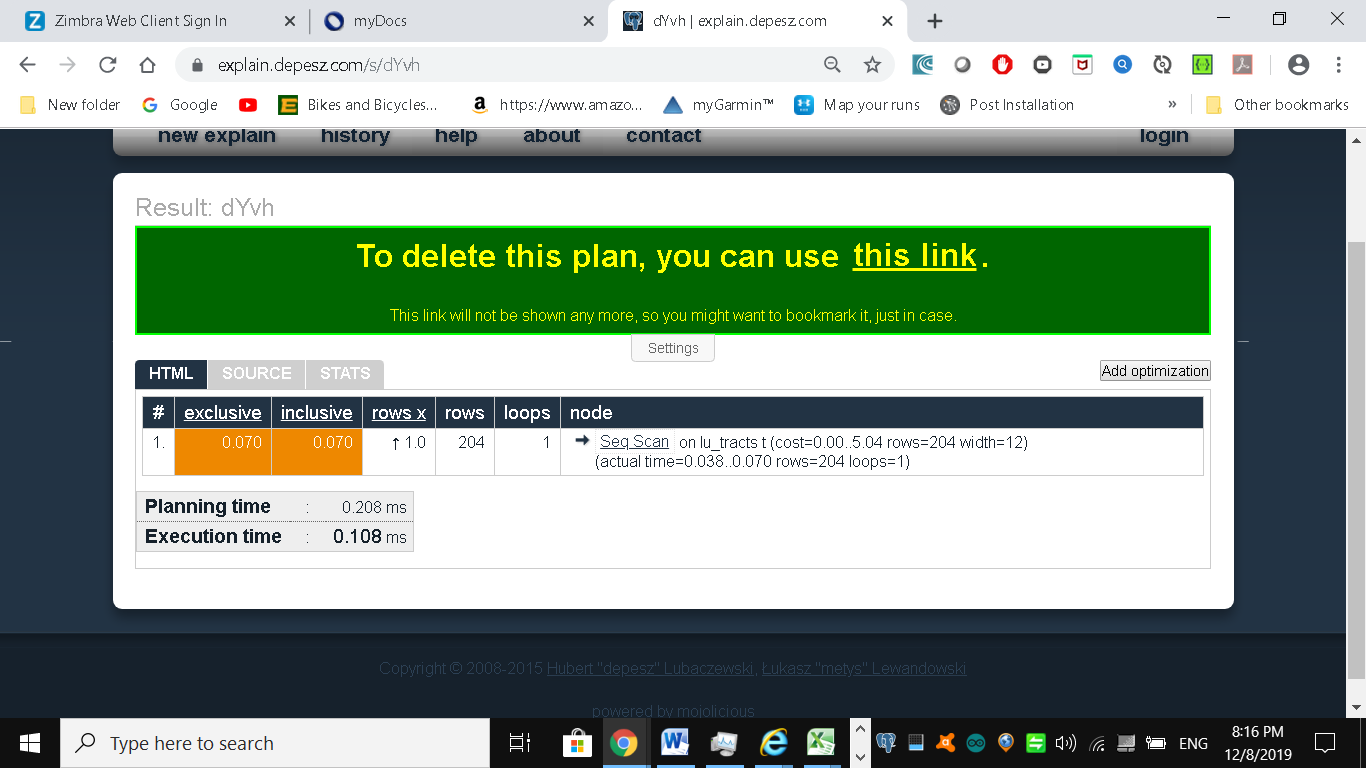
WHERE F.tract\_id = T.tract\_id)) As num\_fact\_types

FROM census.lu\_tracts As T;

Now if we query our view with this query:

EXPLAIN ANALYZE SELECT tract\_id FROM vw\_stats;

|  |
| --- |
| Seq Scan on lu\_tracts t (cost=0.00..5.04 rows=204 width=12) (actual time=0.038..0.070 rows=204 loops=1) |
| Planning Time: 0.208 ms |
| Execution Time: 0.108 ms |

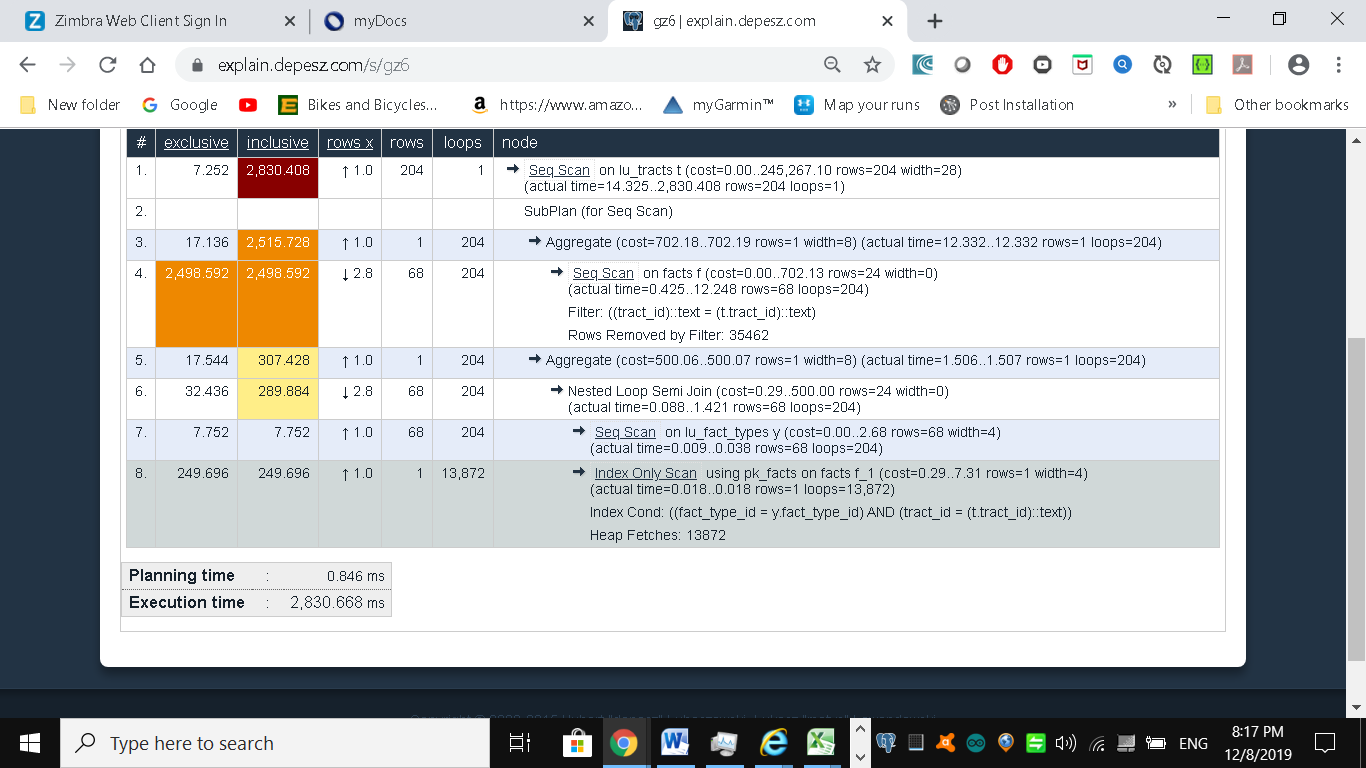


Execution time is about 11ms on our server. If you looked at the plan, you may be startled to find that it never even touches the facts table because it’s smart enough to know it doesn’t need to.

If we used the following:

EXPLAIN ANALYZE SELECT \* FROM vw\_stats;

|  |
| --- |
| Seq Scan on lu\_tracts t (cost=0.00..245267.10 rows=204 width=28) (actual time=14.325..2830.408 rows=204 loops=1) |
| SubPlan 1 |
| -> Aggregate (cost=702.18..702.19 rows=1 width=8) (actual time=12.332..12.332 rows=1 loops=204) |
| -> Seq Scan on facts f (cost=0.00..702.13 rows=24 width=0) (actual time=0.425..12.248 rows=68 loops=204) |
| Filter: ((tract\_id)::text = (t.tract\_id)::text) |
| Rows Removed by Filter: 35462 |
| SubPlan 2 |
| -> Aggregate (cost=500.06..500.07 rows=1 width=8) (actual time=1.506..1.507 rows=1 loops=204) |
| -> Nested Loop Semi Join (cost=0.29..500.00 rows=24 width=0) (actual time=0.088..1.421 rows=68 loops=204) |
| -> Seq Scan on lu\_fact\_types y (cost=0.00..2.68 rows=68 width=4) (actual time=0.009..0.038 rows=68 loops=204) |
| -> Index Only Scan using pk\_facts on facts f\_1 (cost=0.29..7.31 rows=1 width=4) (actual time=0.018..0.018 rows=1 loops=13872) |
| Index Cond: ((fact\_type\_id = y.fact\_type\_id) AND (tract\_id = (t.tract\_id)::text)) |
| Heap Fetches: 13872 |
| Planning Time: 0.846 ms |
| Execution Time: 2830.668 ms |



Our execution time skyrockets to 2830ms, and the plan is just as we had earlier

Though we’re looking at milliseconds still, imagine tables with tens of millions of rows and hundreds of columns.

Those milliseconds could transcribe into overtime at the office waiting for a query to finish.

**Make Good Use of CASE**

We’re always surprised how frequently people forget about using the ANSI-SQL CASE expression. In many aggregate situations, a CASE can obviate the need for inefficient subqueries.

 The CASE statement is SQL’s way of handling if/then logic. The CASEstatement is followed by at least one pair of WHEN and THEN statements—

Every CASE statement must end with the END statement. The ELSEstatement is optional, and provides a way to capture values not specified in the WHEN/THEN statements. CASE is easiest to understand in the context of an example:

**SELECT** player\_name,

**year**,

**CASE** **WHEN** **year** = 'SR' **THEN** 'yes'

**ELSE** NULL **END** **AS** is\_a\_senior

**FROM** benn.college\_football\_players

Or again

**SELECT** player\_name,

weight,

**CASE** **WHEN** weight > 250 **THEN** 'over 250'

**WHEN** weight > 200 **THEN** '201-250'

**WHEN** weight > 175 **THEN** '176-200'

**ELSE** '175 or under' **END** **AS** weight\_group

**FROM** benn.college\_football\_players

We’ll demonstrate with two equivalent queries and their corresponding plans.

Example Using subqueries instead of CASE

SELECT T.tract\_id, COUNT(\*) As tot, type\_1.tot AS type\_1

FROM census.lu\_tracts AS T

LEFT JOIN

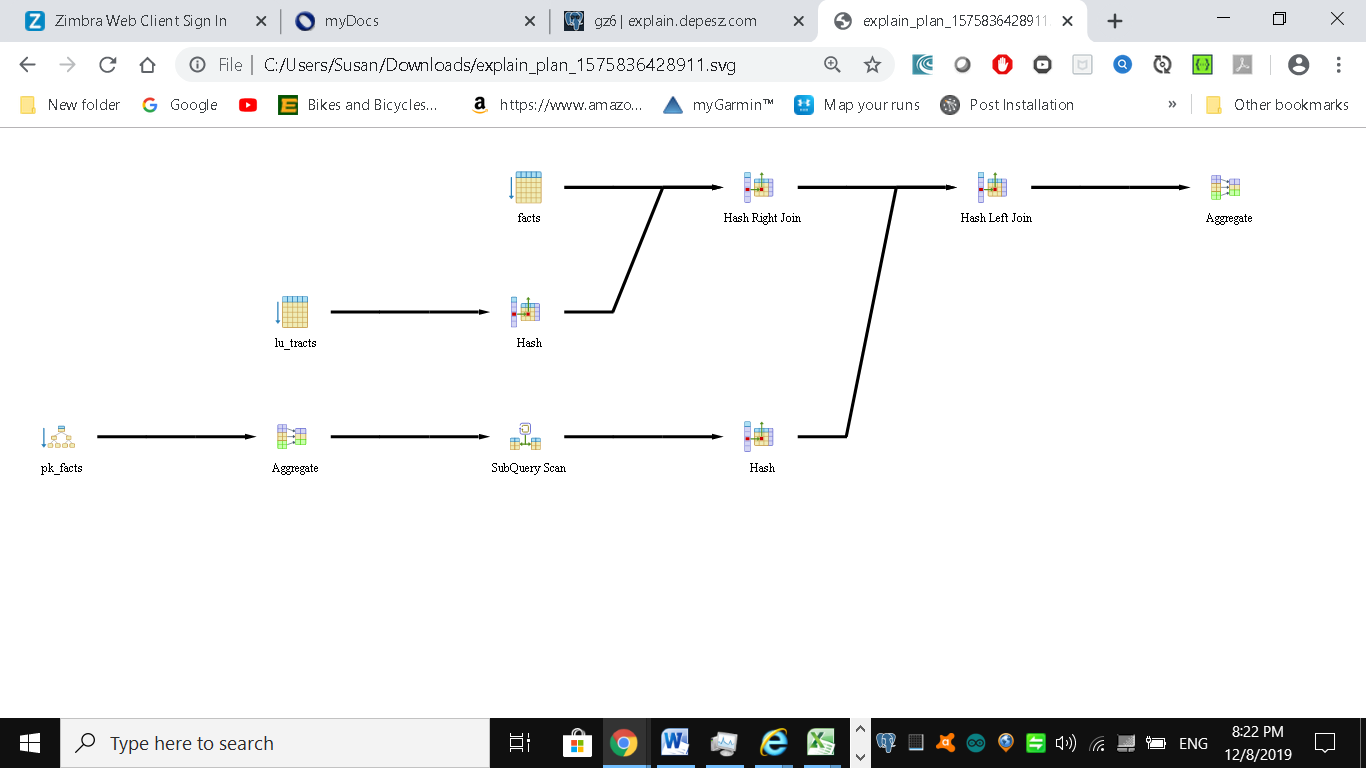
(SELECT tract\_id, COUNT(\*) As tot

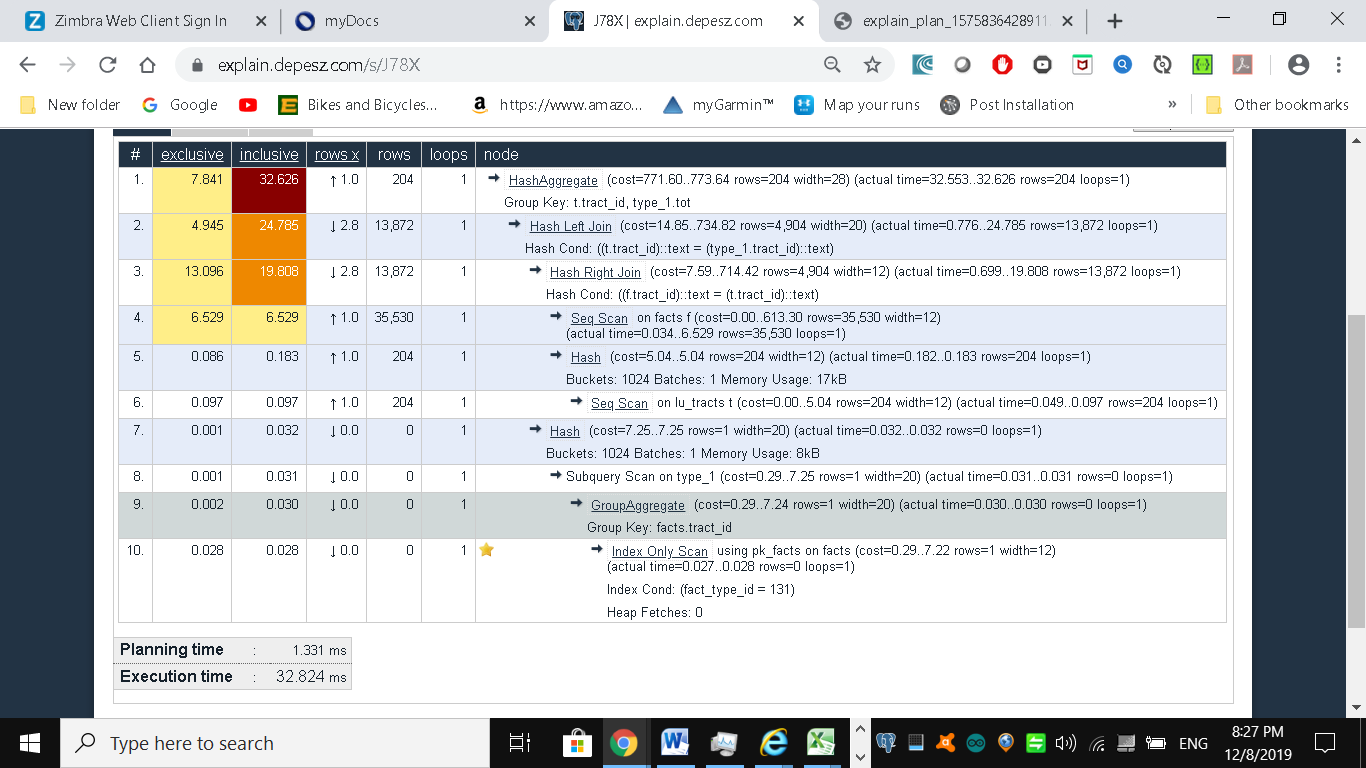
FROM census.facts WHERE fact\_type\_id = 131

GROUP BY tract\_id) As type\_1 ON T.tract\_id = type\_1.tract\_id

LEFT JOIN census.facts AS F ON T.tract\_id = F.tract\_id

GROUP BY T.tract\_id, type\_1.tot;





We now rewrite the query using CASE. You’ll find the revised query is usually faster (though not here) and much easier to read

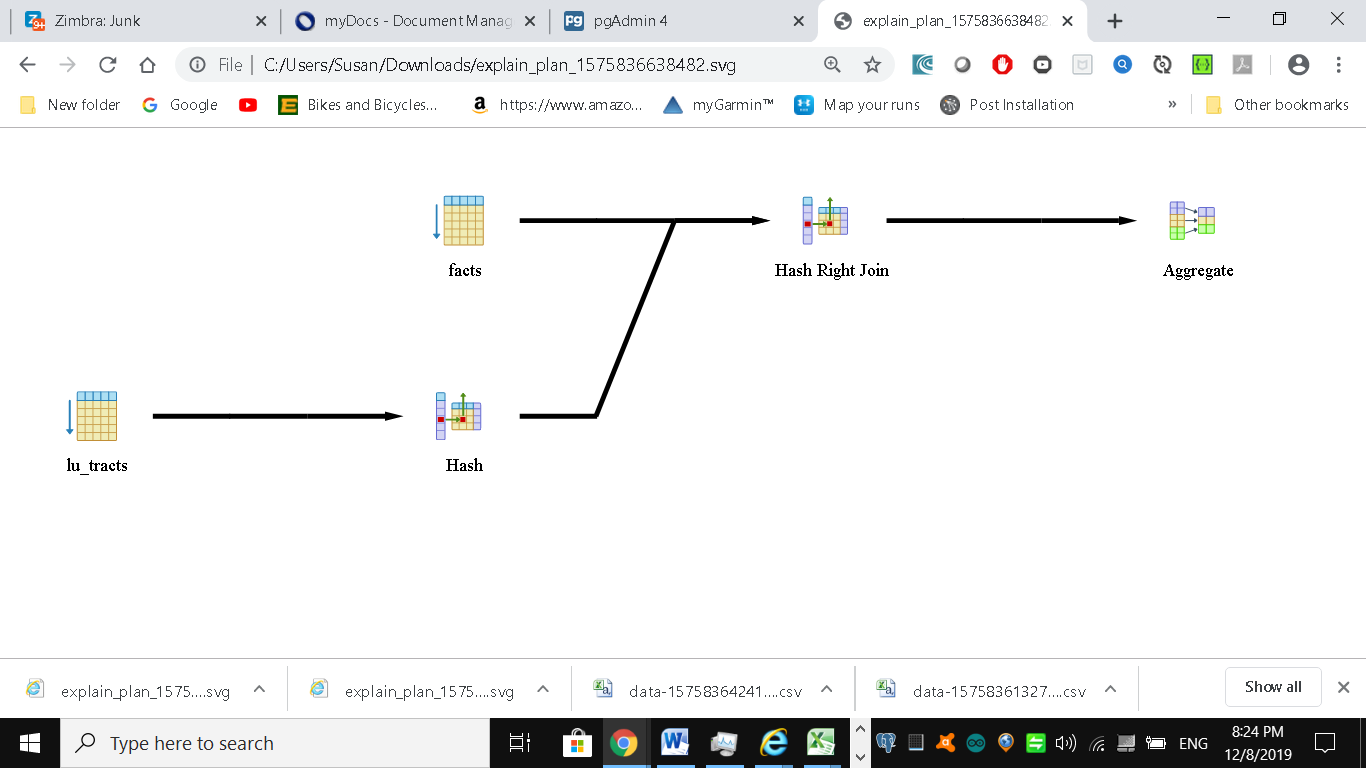
SELECT T.tract\_id, COUNT(\*) As tot

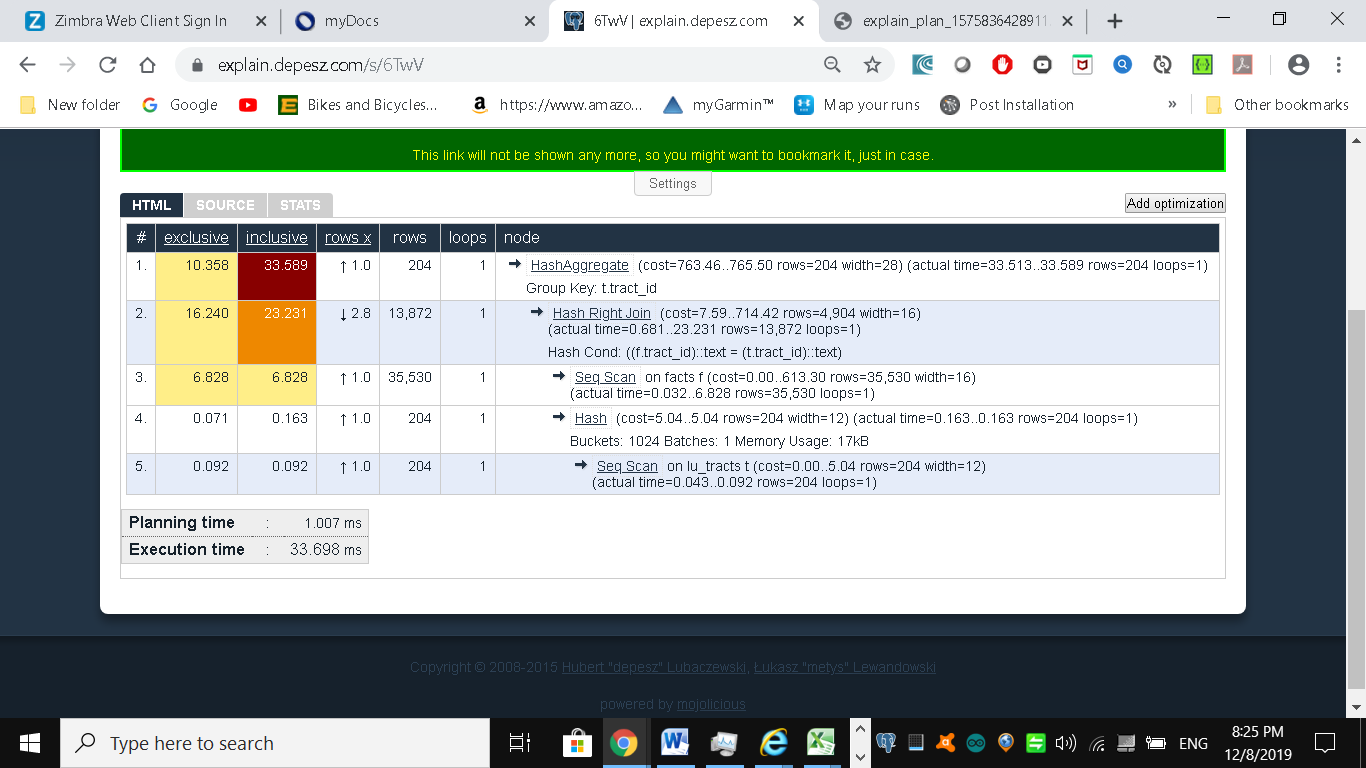
, COUNT(CASE WHEN f.fact\_type\_id = 131 THEN 1 ELSE NULL END) AS type\_1

FROM census.lu\_tracts AS T

LEFT JOIN census.facts AS F ON T.tract\_id = F.tract\_id

GROUP BY T.tract\_id;





Even though our rewritten query still doesn’t use the fact\_type index, it’s still generally faster than using subqueries because the planner scans the facts table only once.

Although not always the case, a shorter plan is generally not only easier to comprehend, but also performs better than a longer one.

Note that eversql.com recommends creating an index and avoiding the use of sub-queries as the optimizer cannot optimize then well. It recommends the following:

Optimal indexing is crucial for query optimization. Please create these indexes before running the optimized query:

CREATE INDEX facts\_idx\_tract\_id ON facts (tract\_id (255));

CREATE INDEX facts\_idx\_fact\_id\_tract\_id ON facts (fact\_type\_id,tract\_id (255));

SELECT

T.tract\_id,

COUNT(\*) AS tot,

type\_1.tot AS type\_1

FROM

census.lu\_tracts AS T

LEFT JOIN

(

SELECT

facts.tract\_id,

COUNT(\*) AS tot

FROM

census.facts

WHERE

facts.fact\_type\_id = 131

GROUP BY

facts.tract\_id

) AS type\_1

ON T.tract\_id = type\_1.tract\_id

LEFT JOIN

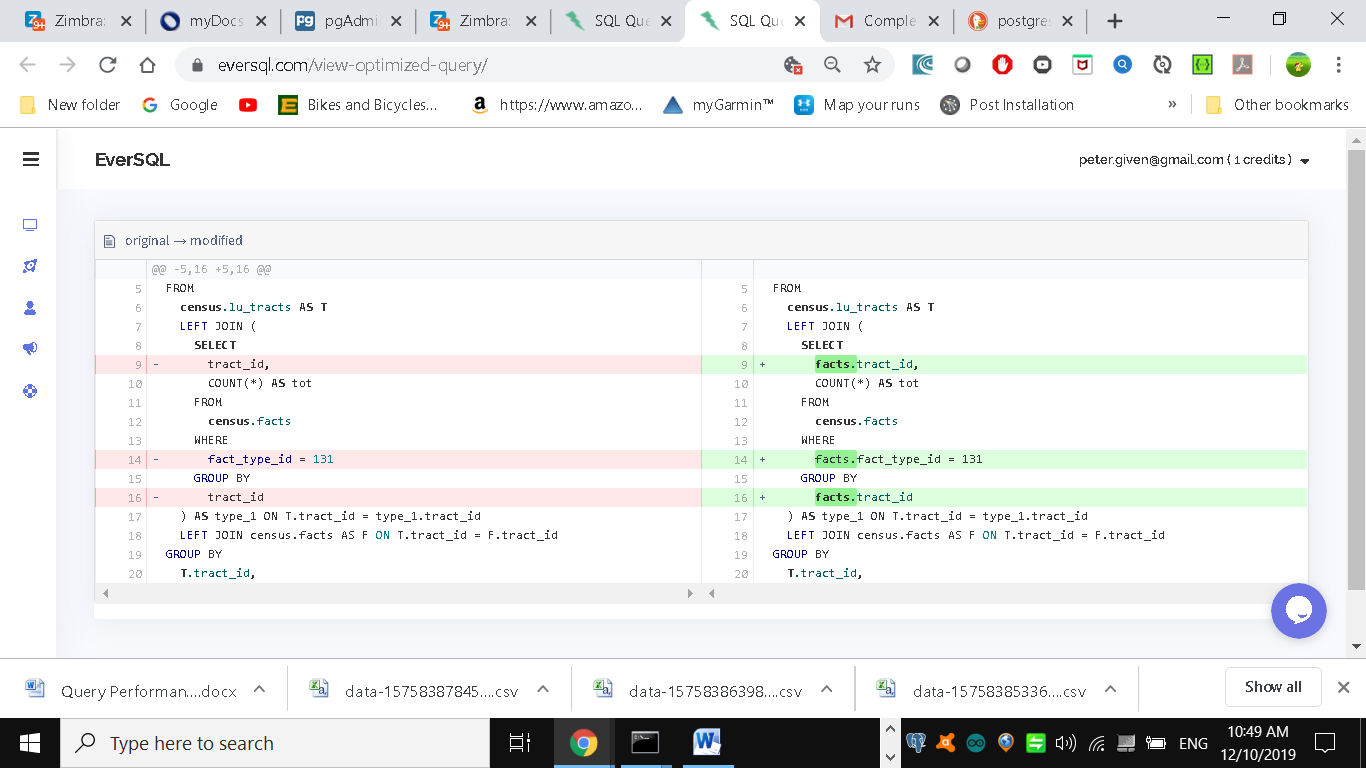
census.facts AS F

ON T.tract\_id = F.tract\_id

GROUP BY

T.tract\_id,

type\_1.tot



Guiding the Query Planner

The planner’s behaviour is driven by several cost settings, strategy settings, and its general perception of the distribution of data.

Based on distribution of data, the costs it ascribes to scanning indexes, and the indexes you have in place, it may choose to use one strategy over another.

In this section we’ll go over various approaches for optimizing the planner’s behaviour.

Strategy Settings

Although PostgreSQL query planner doesn’t provide the option to accept index hints like some other databases, when running a query you can disable various strategy settings on a per query or permanent basis to dissuade the planner from going down an unproductive path.

All planner optimizing settings are documented in the section Planner Method [Configuration](http://www.postgresql.org/docs/current/static/runtime-config-query.html).

By default, all strategy settings are enabled, giving the planner flexibility to maximize the choice of plans.

You can disable various strategies if you have some prior knowledge of the data.

Keep in mind that disabling doesn’t necessarily mean that the planner will be barred from using the strategy.

You’re only making a polite request to the planner to avoid it.

Two of our favourite method settings to disable are the enable\_nestloop and enable\_seqscan.

The reason is that these two strategies tend to be the slowest and should be relegated to be used only as a last resort.

Although you can disable them, the planner may still use them when it has no other viable alternative.

When you do see them being used, it’s a good idea to double-check that the planner is using them out of necessity, not out of ignorance.

One quick way to check is to actually disable them.

How Useful Is Your Index?

When the planner decides to perform a sequential scan, it plans to loop through all the rows of a table.

It will opt for this route if it finds no index that could satisfy a query condition, or it concludes that using an index is more costly than scanning the table.

If you disable the sequential scan strategy, and the planner still insists on using it, then it means that the planner thinks whatever indexes you have in place won’t be helpful for the particular query or you are missing indexes altogether.

A common mistake people make is they write queries and either don’t put indexes in their tables or put in indexes that can’t be used by their queries.

An easy way to check if your indexes are used is to query the pg\_stat\_user\_indexes and pg\_stat\_user\_tables views.

Let’s start off with a query against the lu\_fact\_types table. We’ll add a GIN index on the array column. GIN indexes are one of the few indexes you can use with arrays.

CREATE INDEX idx\_lu\_fact\_types ON census.lu\_fact\_types USING gin (fact\_subcats);

To test our index, we’ll execute a query to find all rows with subcats containing “White alone” or “Asian alone”.

We explicitly enabled sequential scan even though it’s the default setting, just to be sure.

The accompanying EXPLAIN output is shown below (Example Allow choice of Query index utilization)

set enable\_seqscan = true;

EXPLAIN ANALYZE

SELECT \*

FROM census.lu\_fact\_types

WHERE fact\_subcats && '{Renter occupied, For rent}';

|  |
| --- |
| Seq Scan on lu\_fact\_types (cost=0.00..2.85 rows=2 width=200) (actual time=0.052..0.116 rows=2 loops=1) |
| Filter: (fact\_subcats && '{"Renter occupied","For rent"}'::character varying[]) |
| Rows Removed by Filter: 66 |
| Planning Time: 0.281 ms |
| Execution Time: 0.144 ms |

Observe that when enable\_seqscan is enabled, our index is not being used and the planner has chosen to do a sequential scan.

This could be because our table is so small or because the index we have is no good for this query. If we repeat the query but turn off sequential scan beforehand

Example Coerce query index utilization

set enable\_seqscan = false;

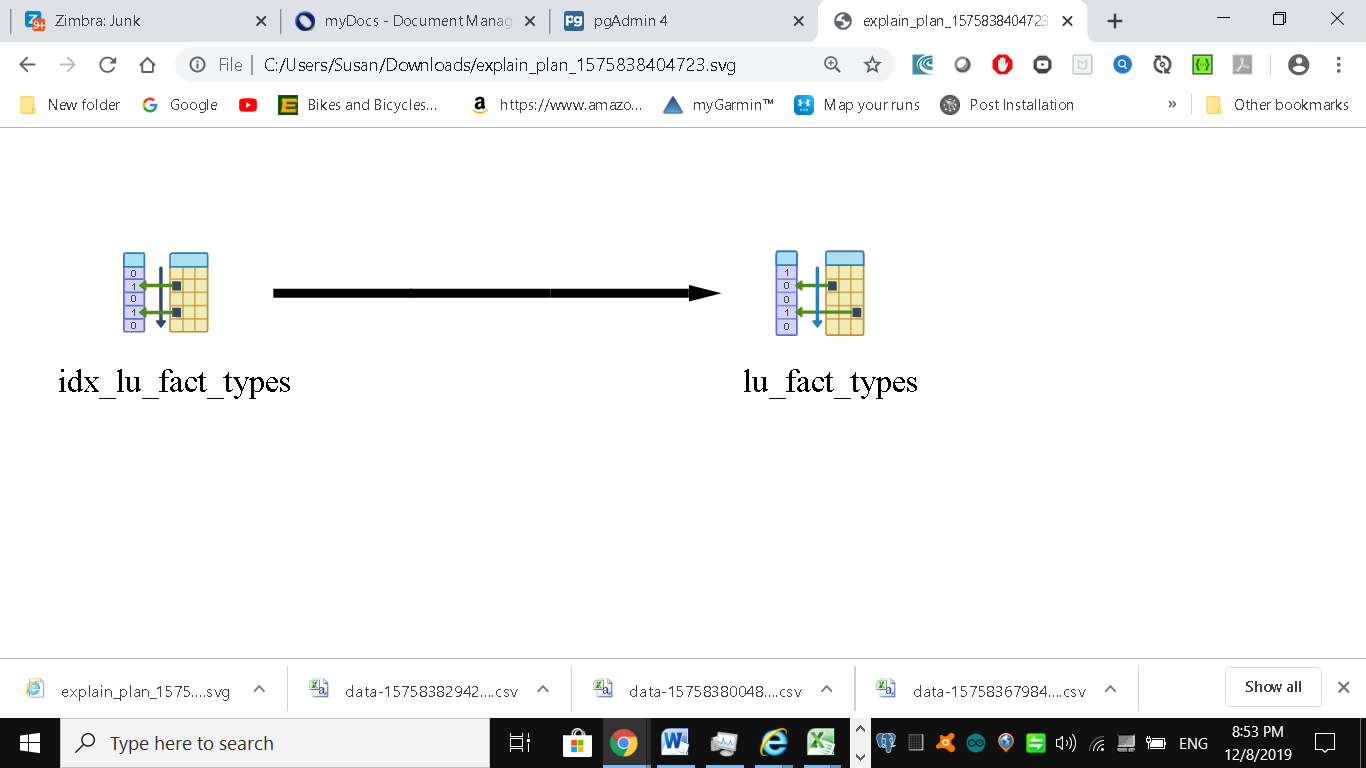
EXPLAIN ANALYZE

SELECT \*

FROM census.lu\_fact\_types

WHERE fact\_subcats && '{Renter occupied, For rent}';

|  |
| --- |
| Bitmap Heap Scan on lu\_fact\_types (cost=12.02..14.04 rows=2 width=200) (actual time=0.079..0.081 rows=2 loops=1) |
| Recheck Cond: (fact\_subcats && '{"Renter occupied","For rent"}'::character varying[]) |
| Heap Blocks: exact=1 |
| -> Bitmap Index Scan on idx\_lu\_fact\_types (cost=0.00..12.01 rows=2 width=0) (actual time=0.063..0.063 rows=2 loops=1) |
| Index Cond: (fact\_subcats && '{"Renter occupied","For rent"}'::character varying[]) |
| Planning Time: 0.463 ms |
| Execution Time: 0.201 ms |



We can see that we have succeeded in forcing the planner to use the index.

In contrast to the above, if we were to write a query of the form:

SELECT \* FROM census.lu\_fact\_types WHERE 'Renter occupied' = ANY(fact\_subcats)

|  |
| --- |
| Seq Scan on lu\_fact\_types (cost=10000000000.00..10000000003.53 rows=1 width=200) (actual time=0.100..0.219 rows=1 loops=1) |
| Filter: ('Renter occupied'::text = ANY ((fact\_subcats)::text[])) |
| Rows Removed by Filter: 67 |
| Planning Time: 0.417 ms |
| Execution Time: 0.264 ms |

We would discover that regardless of what we set enable\_seqscan to, the planner will always do a sequential scan because the index we have in place can’t service this query.

So in short, create useful indexes. Write your queries to take advantage of them. And experiment, experiment, experiment!

Table Stats

Despite what you might think or hope, the query planner is not a magician. Its decisions follow prescribed logic.

The rules that the planner follows depends heavily on the current state of the data. The planner can’t possibly scan all the tables and rows prior to formulating its plan. That would be self-defeating.

Instead, it relies on aggregated statistics about the data. To get a sense of what the planner uses, we’ll query the pg\_stats table

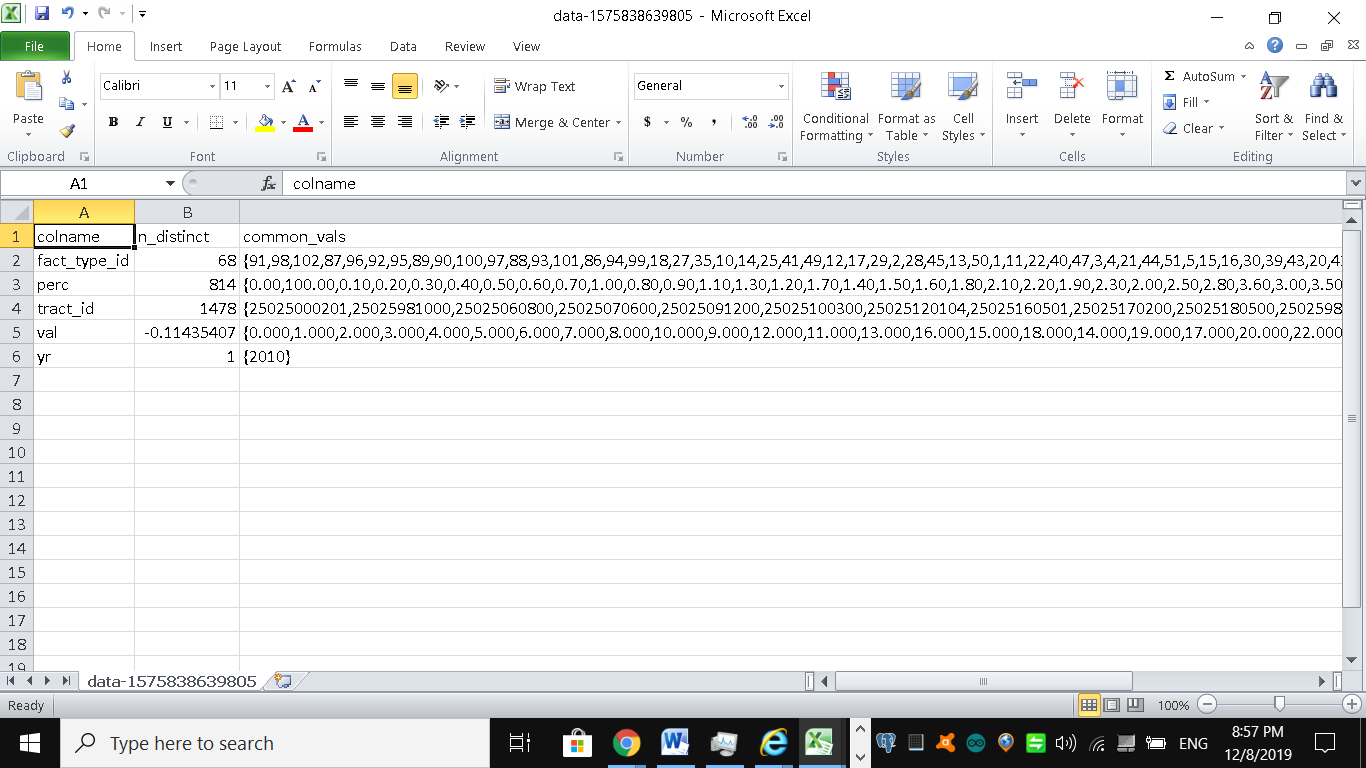
SELECT attname As colname, n\_distinct, most\_common\_vals AS common\_vals,

most\_common\_freqs As dist\_freq

FROM pg\_stats

WHERE tablename = 'facts'

ORDER BY schemaname, tablename, attname;



By using pg\_stats, the planner gains a sense of how actual values are dispersed within a given column and plan accordingly. The pg\_stats table is constantly updated as a background process.

After a large data load, or a major deletion, you should manually update the stats by executing a VACUUM ANALYZE. VACUUM permanently removes deleted rows from tables; ANALYZE updates the stats.

Having accurate and current stats is crucial for the planner to make the right decision. If stats differ greatly from reality, planner will often produce poor plans, the most detrimental of these being unnecessary sequential table scans.

Generally, only about 20 percent of the entire table is sampled to produce stats. This percentage could be even lower for really large tables.

You can control the number of rows sampled on a column-by-column basis by setting the STATISTICS value.

ALTER TABLE census.facts ALTER COLUMN fact\_type\_id SET STATISTICS 1000;

For columns that participate often in joins and are used heavily in WHERE clauses, you should consider increasing sampled rows.

 Caching

If you execute a complex query that takes a while to run, you’ll often notice the second time you run the query that it’s faster, sometimes much, much faster.

A good part of the reason for that is due to caching. If the same query executes in sequence and there has been no changes to the underlying data, you should get back the same result.

As long as there’s space in on-board memory to cache the data, the planner doesn’t need to re-plan or re-retrieve.

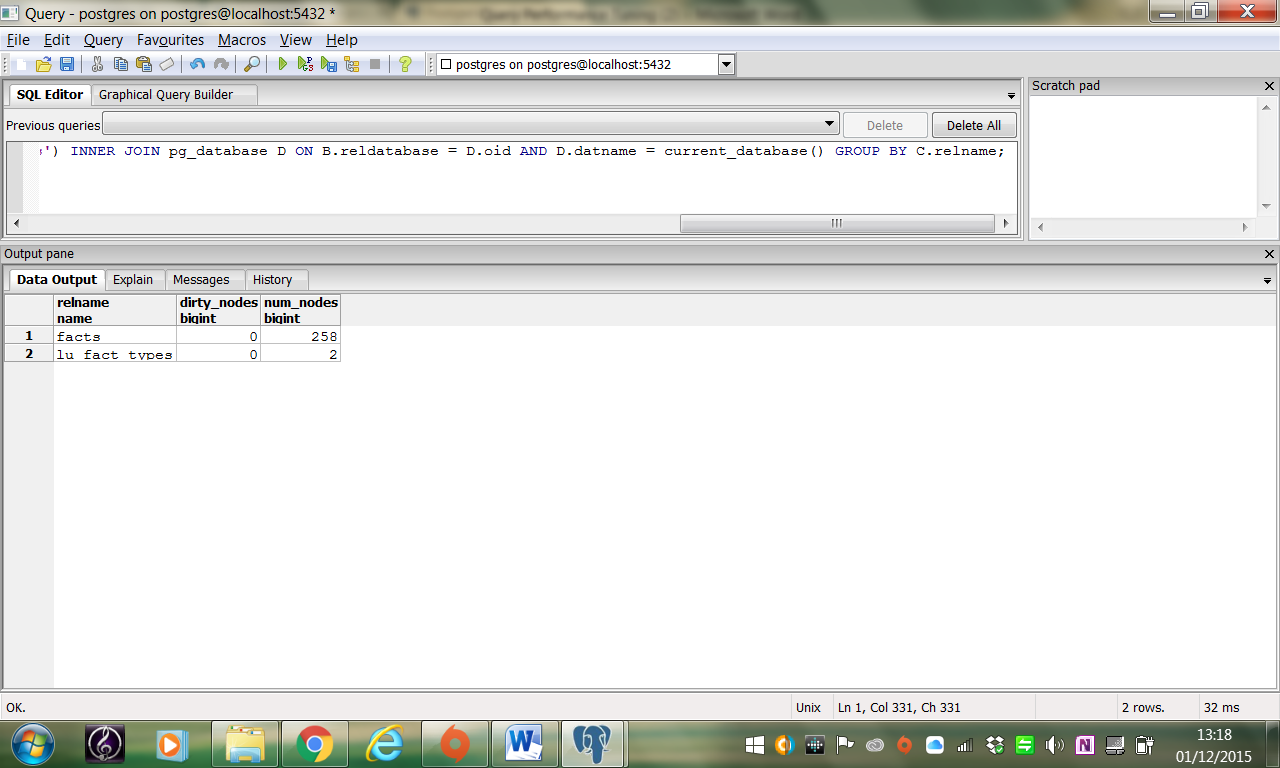
How do you check what’s in the current cache? If you are running PostgreSQL 9.1+, you can install the pg\_buffercache extension with the command:

CREATE EXTENSION pg\_buffercache;

You can then run a query against the pg\_buffercache table as shown below

Example: Are my table rows in buffer cache?

SELECT C.relname, COUNT(CASE WHEN B.isdirty THEN 1 ELSE NULL END) As dirty\_nodes, COUNT(\*) As num\_nodes FROM pg\_class AS C INNER JOIN pg\_buffercache B ON C.relfilenode = B.relfilenode AND C.relname IN('facts','lu\_fact\_types') INNER JOIN pg\_database D ON B.reldatabase = D.oid AND D.datname = current\_database() GROUP BY C.relname;



Example returned buffered records of facts and lu\_fact\_types. Of course, to actually see buffered rows, you need to run a query. Try the one below:

SELECT T.fact\_subcats[1], COUNT(\*) As num\_fact

FROM census.facts As F

INNER JOIN census.lu\_fact\_types AS T ON F.fact\_type\_id = T.fact\_type\_id

GROUP BY T.fact\_subcats[1];

The second time you run the query, you should notice at least a 10% performance speed increase and you should see the following cached in the buffer:

relname | dirty\_nodes | num\_nodes

--------------+-------------+-----------

facts | 0 | 736

lu\_fact\_types | 0 | 3

The more on-board memory you have dedicated to cache, the more room you’ll have to cache data.

You can set the amount of dedicated memory by changing shared\_buffers. Don’t increase shared\_buffers too high since at a certain point you’ll get diminishing returns from having to scan a bloated cache.

Using common table expressions and immutable functions also lead to more caching.

Nowadays, there’s no shortage of on-board memory. In version 9.2 of PostgreSQL, you can take advantage of this fact by pre-caching commonly used tables.

pg\_prewarm will allow you to rev up your PostgreSQL so that the first user to hit the database can experience the same performance boost offered by caching as later users.

A good article that describes this feature is [Caching](http://raghavt.blogspot.com/2012/04/caching-in-postgresql.html) in PostgreSQL.

See: <http://docs.oracle.com/cd/B19306_01/server.102/b14211/sql_1016.htm#i28528>

Exercise: Download and install SQL Optimizer for Oracle and use it to optimize a sql statement (e.g. the query used in the subqueries section)

Here are its features:

* **Optimize SQL** – Analyzes and improves original SQL statements automatically by using artificial intelligence to rewrite the syntax, and apply the Oracle optimization hints. Generates semantically equivalent and syntactically correct SQL statements that you can test run to determine the best ones for your environment.
* **Optimize indexes** – Enhances an application’s performance by analyzing its SQL workload (from Oracle Automatic Workload Repository, Foglight Performance Analysis or source code), and identifying indexing changes that will improve it.
* **Batch optimize SQL** – Automatically optimizes PL/SQL for performance by detecting issues (through Oracle System Global Area (SGA), Foglight Performance Analysis or source code) and providing rewrites that execute faster.
* **Scan SQL** - Extracts inefficient SQL statements automatically from your source code, plus reviews execution plans from these statements and categorizes them by complexity. Optimizes the problematic statements with Optimize SQL or Batch Optimize SQL.
* **Inspect SGA**– Captures, analyzes and classifies both running and executed SQL statements from Oracle SGA.  Send problematic statements to Optimize SQL or Batch Optimize SQL for optimization.
  + All Oracle processes use the SGA to hold information. The SGA is used to store incoming data (the data buffers as defined by the db\_cache\_sizeparameter), and internal control information that is needed by the database. You control the amount of memory to be allocated to the SGA by setting some of the Oracle initialization parameters?
* **Analyze impact** – Reviews the effect that database modifications may have on the SQL by identifying execution plan changes that could have occurred within a set of SQL statements. Collects SQL from a variety of sources, such as Oracle Automatic Workload Repository, Foglight Performance Analysis or source code.
* **Manage plans**– Manages both Oracle Stored Outlines and SQL Plan Baselines, and allows you to refine execution plans without changing the source code. Stored Baselines are optimization hints that help maintain an execution plan for a SQL statement.

See: <https://www.geekytidbits.com/performance-tuning-postgres/>