Contents

Introduction

A given SQL query (and hence, a query tree) can be executed in a wide variety of different ways, each of which will produce the same set of results. There are three core joining algorithms (Nested Loops, Hash Join and Merge Join), two aggregation algorithms (Hashing and Sorting), two scan algorithms (Index Scan and Sequential Scan). Hence for a non trivial query, there are many alternative plans that can be generated.

The task of the *planner/optimizer* is to create an optimal execution plan. It determines the best way to evaluate the query. This requires:

- Determining the set of possible plans.
- Choosing the "best" plan from this set.

The planner chooses between plans based on their estimated cost.

PostgreSQL planner's search procedure works with data structures called *paths*, which are simply cut-down representations of plans containing only as much information as the planner needs to make its decisions. After the cheapest path is determined, a full-fledged *plan tree* is built to pass to the executor. This represents the desired execution plan in sufficient detail for the executor to run it.

Optimizer Decisions

Given a query, the Optimizer has to make the following decisions.

- Scan Method – Sequential Scan, Index Scan, Bitmap Index Scan.
- Join Method – Nested Loop, Hash Join, Merge Join.
- Join Order.

PostgreSQL generates optimal query plans for most queries. However, in some situations, examining each possible way in which a query can be executed would take an excessive amount of time and memory space. In particular, this occurs when executing queries involving large numbers of join operations. Besides, there are times, especially with more complex

applications, when certain queries benefit from user intervention and some level of hand tuning. Plan forcing comes to the rescue.

Foreign Plan Costing (FPC)

Foreign Plan Costing is the process of forcing a plan to be chosen by the Query Optimizer to process a query.

When can FPC be beneficial?

- When the data distribution is heavily skewed, cardinality estimation (how many rows the query optimizer expects each operator to process) can be wildly incorrect, resulting in poor quality query plans and degraded performance.
- Performance of some queries deteriorates because of suboptimal execution plans generated from stale statistics.
- More flexibility for Database Administrators who wish to tweak Optimizer plan choices.

Requirements

- Define a way to specify the xml plan to be forced along with the query.
- Build XML parser
- Force scan types for the different relations in the query.
- Force join types and join orders among the different relations in the query.

Goal

Our goal is to implement Foreign Plan Costing in PostgreSQL and evaluate its performance.

Design and Implementation

Input Processing

To define a way to specify the XML plan along with the input, we have added the keyword "fpc".

Syntax : <*query*> ***fpc*** < *xml_file_path*>

query - Actual Query
fpc - keyword denoting that the user wants to use Plan Forcing.
<xml_file_path> - Fully qualified name of the XML file.

(Note: This XML file should contain valid plans for the query.)

Generating Valid XML plans

The most common and the reliable way to force a plan are to capture a plan that is produced by the Optimizer and then force it on the same query. Users can make use of the 'EXPLAIN' command available in PostgreSQL for this purpose, to generate valid plans to be given as the input to FPC.

Example:

```
EXPLAIN (FORMAT XML)  select * from supplier, partsupp, customer where s_suppkey
= ps_suppkey;
```

produces the plan used by the optimizer in XML format.

XML Parsing

PostgreSQL doesn't provide a DTD/XML Schema defined for the XML tags used in the query outputs. Thus we needed to walk through PostgreSQL code to find out all the possible tags. Once the tags were finalized, a dedicated XML parser needed to be built inside PostgreSQL. We used the Libxml2 to parse the XML documents. Libxml2 is the XML C parser [2]. It provides various library functions that help in parsing the XML documents.

Tweaking Planner Configuration

Planner Method Configuration parameters can be used to force the optimizer to choose a different plan. These GUC parameters can be set just for the duration of a function call or for a transaction or session alone using "set" command with appropriate GucAction defined. We employ this approach to effectively eliminate all plans other than the one given in XML file.

The following Configuration parameters are used:

enable_seqscan (boolean)

- Enables or disables the query planner's use of sequential scan plan types. The default is on.

enable_bitmapscan (boolean)

- Enables or disables the query planner's use of bitmap-scan plan types. The default is on.

enable_indexscan (boolean)

- Enables or disables the query planner's use of index-scan plan types. The default is on.

enable_indexonlyscan (boolean)

- Enables or disables the query planner's use of index-only-scan plan types. The default is on.

enable_mergejoin (boolean)

- Enables or disables the query planner's use of merge-join plan types. The default is on.

enable_nestloop (boolean)

- Enables or disables the query planner's use of nested-loop join plans. The default is on.

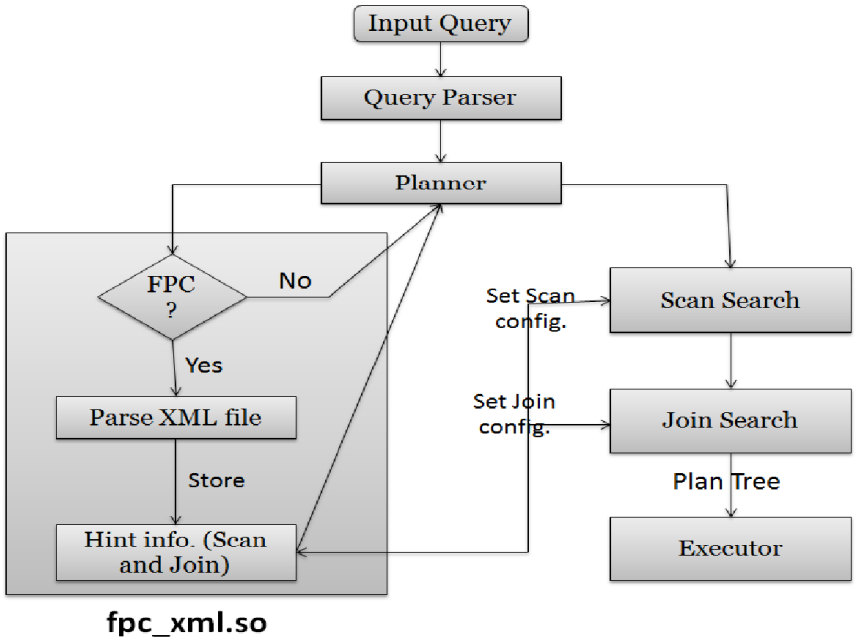
enable_hashjoin (boolean)

- Enables or disables the query planner's use of hash-join plan types. The default is on.

(Note: It is impossible to suppress sequential scans or nested loop joins entirely)

System Diagram

The system diagram shown below depicts the outline of plan forcing mechanism. FPC is implemented in *fpc_xml.so* using hook methods of PostgresSQL(explained below). When a query is issued, we check whether it is an fpc query and if so, we get the xml file name so that it can be sent to fpc_xml library during planning phase. If it is a normal query(without plan forcing) it follows usual path of going to planner then after all paths have been built, cheapest path is chosen and plan node is built which is then given to executor.



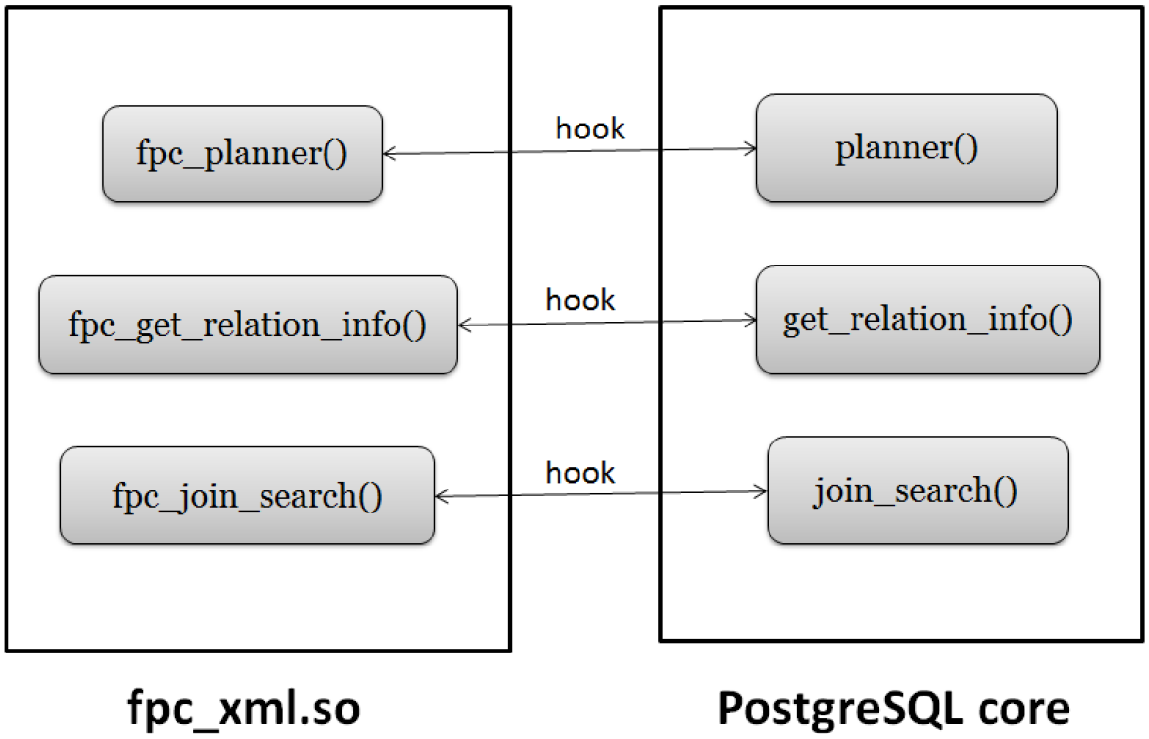
The scenario is different when we issue a fpc query. During planning phase, planner hook method is called. If it's a fpc query and 'fpc_enable' guc parameter is enabled then we parse the xml plan and get the scan hints, join hints and order of joins from the xml plan and store it in convenient data structures. Then during scan search and join search by the optimizer, we have hooked the methods which get relation catalog information and performs join search to fpc_xml library. We Set GUC planner configuration parameters temporarily, forcing the planner to ignore other plans and choose the forced plan for further processing. Once plan node is chosen, we reset GUC parameters to default value.

Hook Methods

Plan forcing is implemented in fpc_xml.c and it is bundled into a shared library called "fpc_xml.so". This Fpc_xml.so can be modified and built independent of postgreSQL. We have defined a custom GUC variable to enable fpc. Fpc_xml.so library can be loaded dynamically from postgreSQL on demand and used for plan forcing.

We have defined the hooks for the following methods in fpc_xml.c

Method	**filename**
planner()	/optimizer/plan/planner.c
get_relation_info()	/optimizer/util/plancat.c
standard_join_search()	/optimizer/path/allpaths.c



Hooks consist of global function pointers initially set to NULL. When PostgreSQL wants to use a hook it checks the global function pointer and executes it if it is set. A hook function is available in a shared library. At load time, PostgreSQL calls the _PG_init() function of the shared library. This function needs to set the pointer and usually saves the previous one. At unload time,

PostgreSQL calls the _PG_fini() function of the shared library. This function needs to unset the pointer and usually restores the previous one.

planner() hook is called when the planner phase starts. Here we parse xml plan and store information obtained from the XML plan. Then the default planner operations are executed. When get_relation_info() hook is invoked, we set scan configuration parameters and when join_search() hook is called, we set join configuration parameters here. Relations are joined according to the order used in XML formatted plan. GUC configuration parameters are set for a short period during planning phase and are reverted back to default values after that.

With this approach we were able to force scan methods, join methods and order of joins effectively without making modifications to PostgreSQL Core distribution planner code.

Limitations

The following are the limitations of the implemented FPC functionality.

- We can force index scan only if there are indices available. Otherwise, sequential scan will be used
- Only query plans that can be found by the typical search strategy of the query optimizer can be forced.
- FPC can only be used with select queries.

Source Files Modified

PostgreSQL 9.3.2 Core Distribution:

Modified file **:** *postgres.c* *in src/backend/tcop*

fpc_xml.so

Newly added file **:** *fpc_xml.c, Makefile*

Additional files : core_postgres.c, make_join_rel.c

(adapted from postgreSQL 9.3.2 core distribution)

These files is compiled into the shared library fpc_xml.so which is dynamically loaded into the PostgreSQL core. The generated .so file is added to the /contrib path of the PostgreSQL code. In the postgresql.conf file, under shared_preloaded_libraries, the created .so file is added.

Use cases

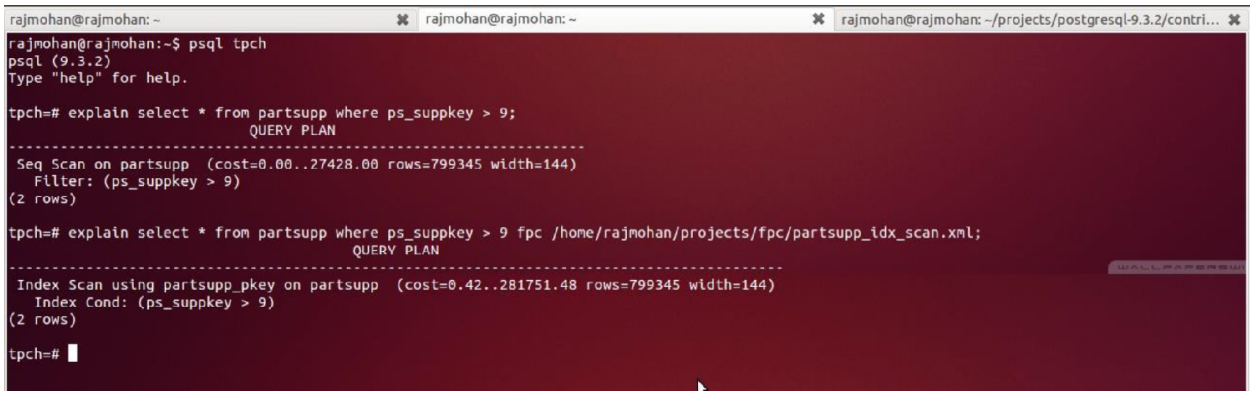
Various queries with Plan Forcing enabled were run on the TPC-H database. The following screenshots show the various results obtained. In each screenshot, first the query without "fpc" enabled is run to show the plan that the Optimizer chooses. Then the query is run with "fpc" enabled and the output shows the forced plans. The XMLs given as inputs are obtained from PostgreSQL EXPLAIN output.

Note: Indexes were generated at different stages to produce different use cases, so there might be some variations in the explain output of the query.

Plan Forcing – Scans

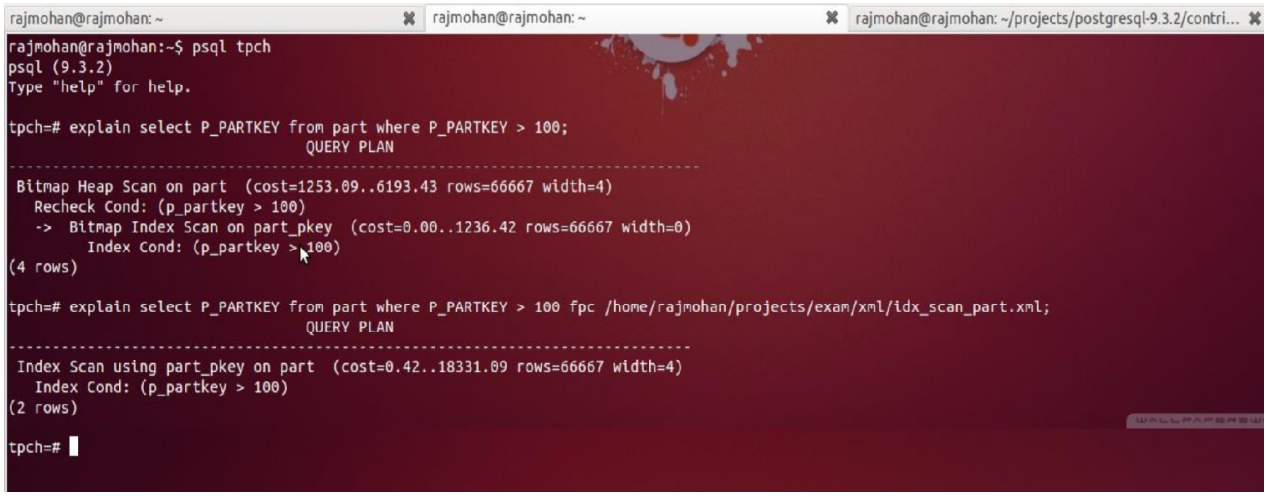
- Sequential Scan forced on a query that the Optimizer executes with Index Scan

- Index Scan forced on a query that the Optimizer executes with Sequential Scan



- Sequential Scan forced on Bitmap Index Scan

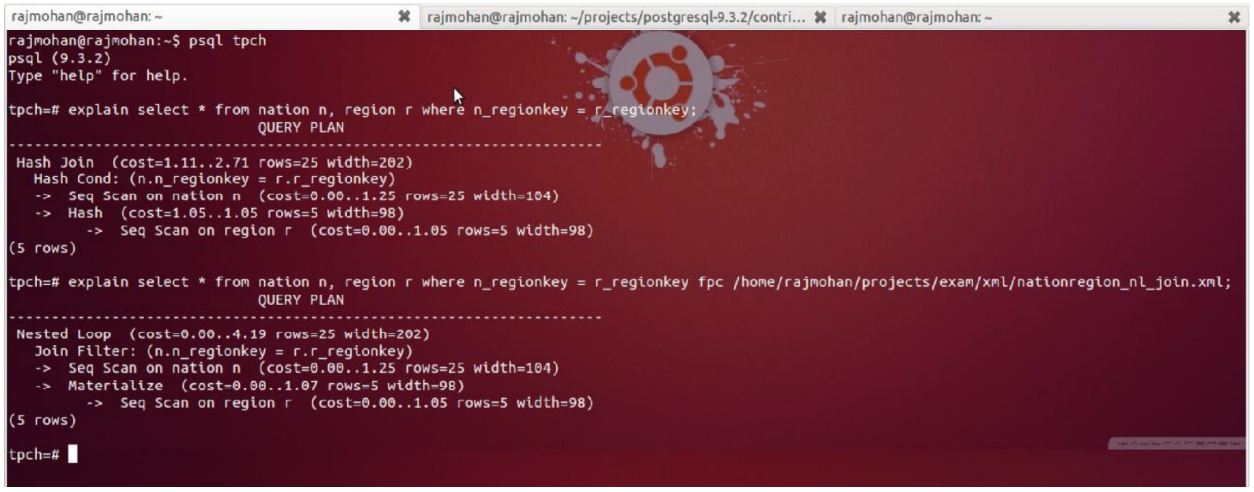
- Index Scan forced on Bitmap Index Scan



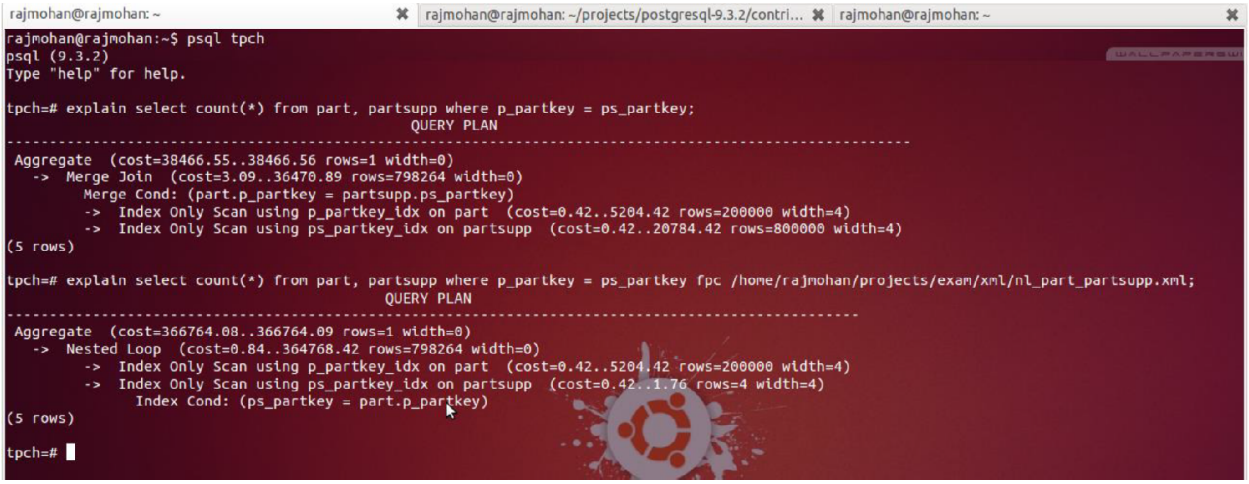
- Index Only Scan forced on Bitmap Index Scan

Plan Forcing – Joins

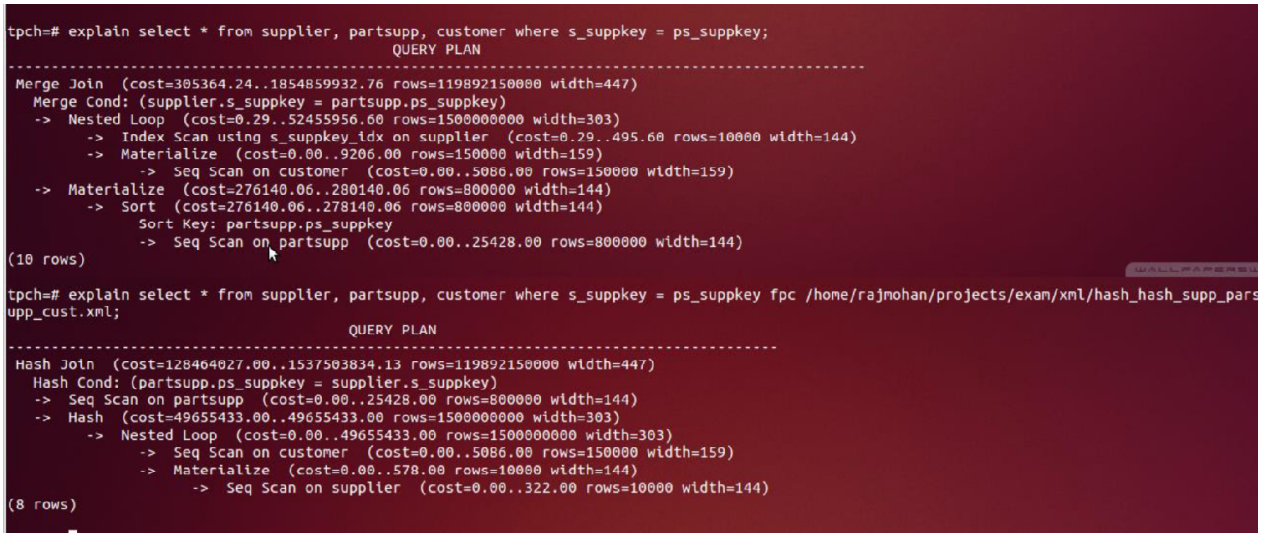
- Nested Loop Join forced on Hash Join



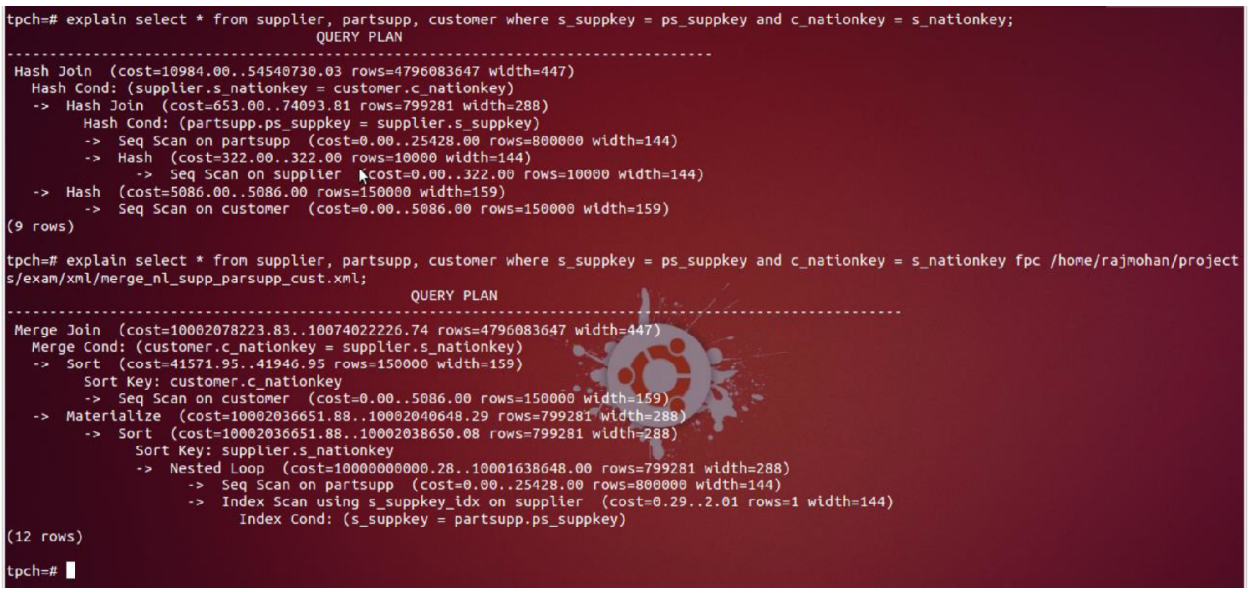
- Nested Loop forced on Merge Join



- Hash Join forced on Merge Join

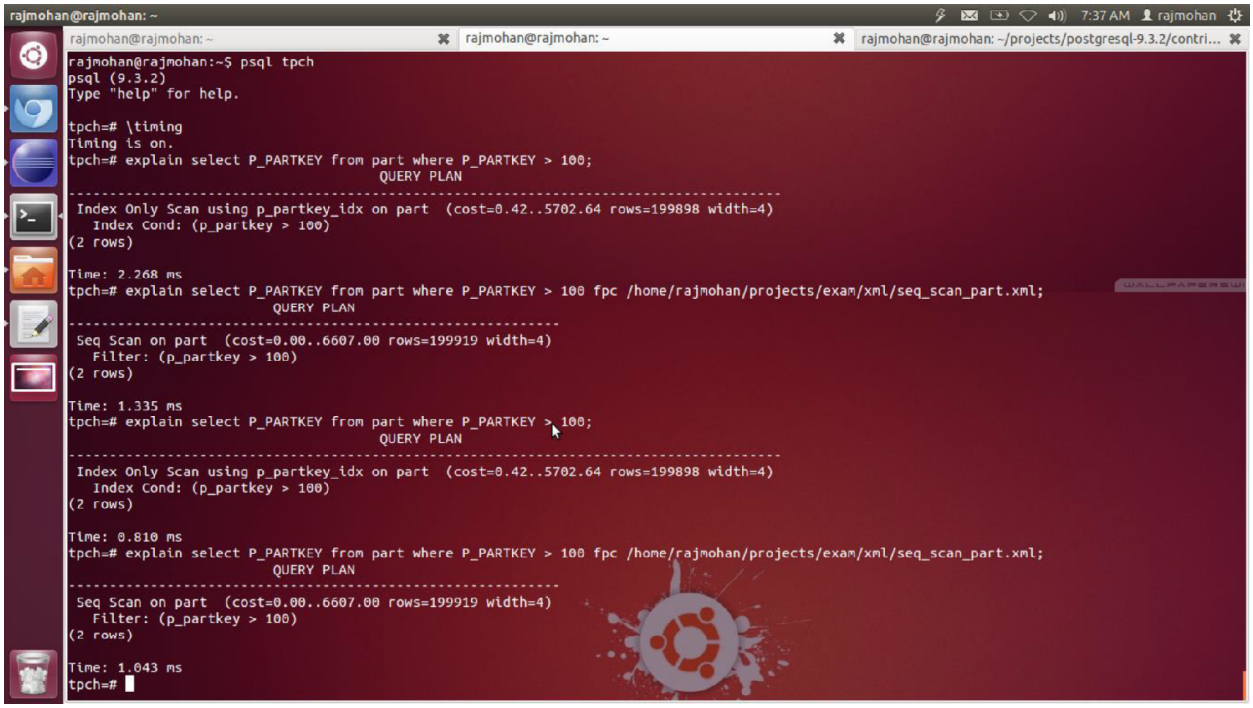


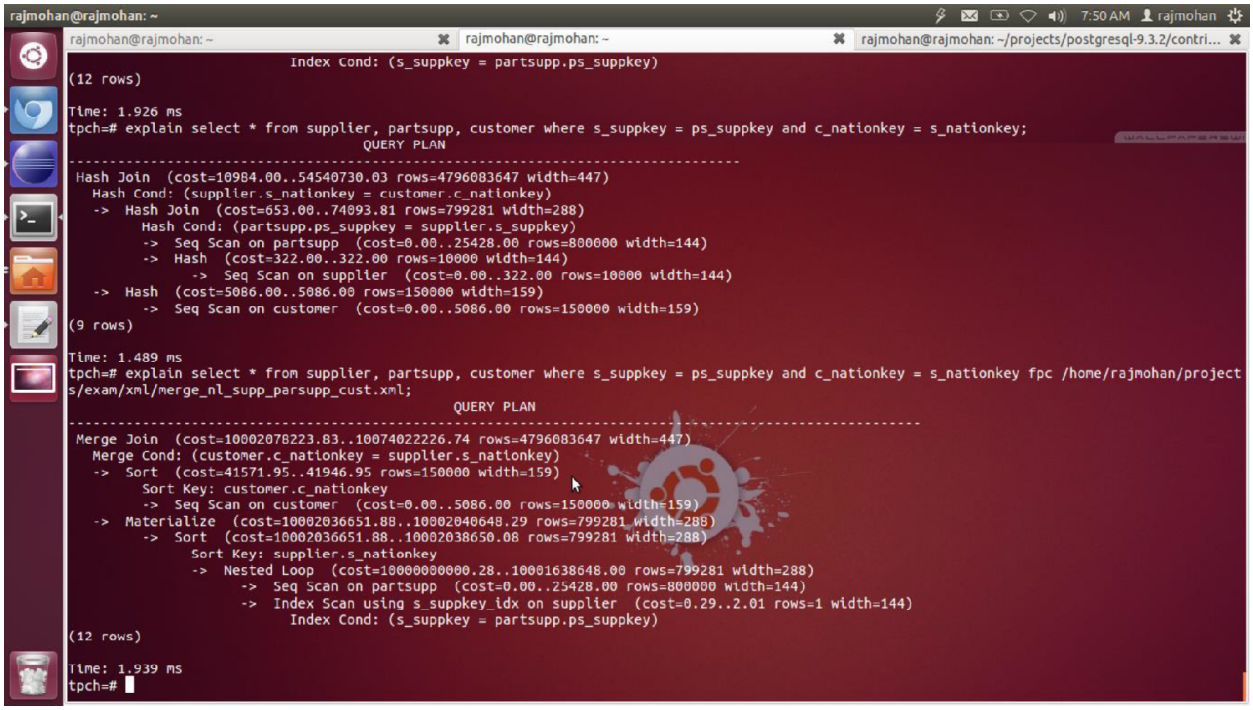
- (Merge Join, Nested Loop) forced on (Hash Join, Hash Join) and Index Scan forced on Sequential Scan



Timing Analysis

The following use cases are some examples of the timing comparisons between the existing Optimizer planning and the forced plan. Query timings are almost the same when using and not using the FPC. Generally, FPC takes a little more time than the normal execution. We believe the reason behind the similar timing between the FPC and the standard Optimizer search could be attributed to XML parsing.


rajmohan@rajmohan: ~
rajmohan@rajmohan:~$ psql tpch
psql (9.3.2)
Type "help" for help.
tpch=# \timing
Timing is on.
tpch=# explain select P_PARTKEY from part where P_PARTKEY > 100;
QUERY PLAN
Index Only Scan using p_partkey_idx on part (cost=0.42..5702.64 rows=199898 width=4)
Index Cond: (p_partkey > 100)
(2 rows)
Time: 2.268 ms
tpch=# explain select P_PARTKEY from part where P_PARTKEY > 100 fpc /home/rajmohan/projects/exam/xml/seq_scan_part.xml;
QUERY PLAN
Seq Scan on part (cost=0.00..6607.00 rows=199919 width=4)
Filter: (p_partkey > 100)
(2 rows)
Time: 1.335 ms
tpch=# explain select P_PARTKEY from part where P_PARTKEY > 100;
QUERY PLAN
Index Only Scan using p_partkey_idx on part (cost=0.42..5702.64 rows=199898 width=4)
Index Cond: (p_partkey > 100)
(2 rows)
Time: 0.810 ms
tpch=# explain select P_PARTKEY from part where P_PARTKEY > 100 fpc /home/rajmohan/projects/exam/xml/seq_scan_part.xml;
QUERY PLAN
Seq Scan on part (cost=0.00..6607.00 rows=199919 width=4)
Filter: (p_partkey > 100)
(2 rows)
Time: 1.043 ms
tpch=#


rajmohan@rajmohan: ~
Index Cond: (s_suppkey = partsupp.ps_suppkey)
(12 rows)
Time: 1.926 ms
tpch=# explain select * from supplier, partsupp, customer where s_suppkey = ps_suppkey and c_nationkey = s_nationkey;
QUERY PLAN
Hash Join (cost=10984.00..54540730.03 rows=4796083647 width=447)
Hash Cond: (supplier.s_nationkey = customer.c_nationkey)
-> Hash Join (cost=653.00..74093.81 rows=799281 width=288)
Hash Cond: (partsupp.ps_suppkey = supplier.s_suppkey)
-> Seq Scan on partsupp (cost=0.00..25428.00 rows=800000 width=144)
-> Hash (cost=322.00..322.00 rows=10000 width=144)
-> Seq Scan on supplier (cost=0.00..322.00 rows=10000 width=144)
-> Hash (cost=5086.00..5086.00 rows=150000 width=159)
-> Seq Scan on customer (cost=0.00..5086.00 rows=150000 width=159)
(9 rows)
Time: 1.489 ms
tpch=# explain select * from supplier, partsupp, customer where s_suppkey = ps_suppkey and c_nationkey = s_nationkey fpc /home/rajmohan/project
s/exam/xml/merge_nl_supp_parsupp_cust.xml;
QUERY PLAN
Merge Join (cost=10002078223.83..10074022226.74 rows=4796083647 width=447)
Merge Cond: (customer.c_nationkey = supplier.s_nationkey)
-> Sort (cost=41571.95..41946.95 rows=150000 width=159)
Sort Key: customer.c_nationkey
-> Seq Scan on customer (cost=0.00..5086.00 rows=150000 width=159)
-> Materialize (cost=10002036651.88..10002040648.29 rows=799281 width=288)
-> Sort (cost=10002036651.88..10002038650.08 rows=799281 width=288)
Sort Key: supplier.s_nationkey
-> Nested Loop (cost=10000000000.28..10001638648.00 rows=799281 width=288)
-> Seq Scan on partsupp (cost=0.00..25428.00 rows=800000 width=144)
-> Index Scan using s_suppkey_idx on supplier (cost=0.29..2.01 rows=1 width=144)
Index Cond: (s_suppkey = partsupp.ps_suppkey)
(12 rows)
Time: 1.939 ms
tpch=#

References

[1] http://www.postgresql.org/docs/9.3/interactive/index.html

[2] http://xmlsoft.org/

[3] http://www.postgresql.org/

[4] https://wiki.postgresql.org/wiki/Main_Page