Accelerating FPC

Introduction

FPC is Foreign Plan Costing. It is about identifying the cost of an optimal plan outside its optimality region. Consider a Query Q which has 2 base relations, predicates p1 and p2 associated with the base relations having selectivity s1 and s2 respectively. We say that a plan P(s1, s2) is optimal for the query Q if cost of execution of the query through the plan p(s1, s2) is lesser than that of executing the same query through any other plan. Hence FPC is all about estimation the cost of non optimal plans at different selectivities of a query. Accelerating FPC is about optimizing the execution time of FPC.[1]

Query Optimization

Basically for a given user query there are a lot of possible ways to achieve the desired results. Hence comes the scope of query optimization. Before discussing the Query optimization process, one has to realize that different plans of execution render different execution costs (by costs we mean Execution time primarily and it also includes Amount of memory required, Latency etc. As other parameters). Now that we know different strategies take different time, Database Engine must make decision to execute the query with most optimal plan possible. Optimal plan is the plan that takes the least cost.

Query Optimization is Dynamic Programming

Most of the real world optimizers implement a strategy called Cost Based Optimization [2]. We subject each plan to a cost model which predicts the cost of execution of a query through all possible plans and then ultimately choose the one with the least cost. In this process one can easily identify the fact that most of the computations are repeated and hence deduce the fact that these repeated computations can be memorized and hence we can construct a DP for identifying the best plan.

Query optimization is NP hard

Query optimization happens to be a combinatorial difficult search problem. Even though the problem has a nice DP structure associated with it, the size of the DP Lattice is very large and in fact exponential in terms of the number of base relations in the query (relations in the from clause). Hence DP is an affordable solution only when the number of relations is manageable and when the base relations become large we will have to adopt different strategy to solve this problem. Optimizers today use a strategy called Genetic Query Optimization when the number of base relations in the query cross a certain threshold (postgres by default has the threshold set to 12 and it is user tuneable)

Query optimization is error contained

As we discussed earlier Query optimization follows cost based selection strategy. The costs are derived by the underlying cost model. Most of this cost model is based on the statistics present in the database system. These statistics are not reliable 100%, because most of the times the stats are out of date and moreover the stats are in general approximation of the data present in the database. Hence the stats come with an error margin associated with them, which in theory can completely deviate the result from the optimal and this is undesirable. Therefore optimizers are expected to merit the error in the cost models.

Query optimization is overkill

Most of toays' optimizers make too many fine grained choices based on the underlying cost model which is error prone as discussed earlier. Hence if the optimizers makes choices completely relying on the cost model it is bound to not get the optimal plan.
There is a rich literature that proves that query Optimization overkills the purpose[3]

Query optimization is underkill

Query optimization is an under kill as well. The reason being while selecting the plans, we only account for some factors like selectivities, I/O cost etc and neglect many other factors which might significantly contribute to the query cost in pathalogical cases. There is a very good study in this context called as parametric query optimizer [4]. This paper essentially talks about how the plan choices could vary if we account for the number of buffers that would be available for the query at run time which most of the optimizers donot account for. The idea in a nutshell is to identify the optimal plans across the entire bufferspace,that would probably be available at the run time, during compile time itself. Once we have the paramertrically (No. Buffers in run time) optimal set of plans, at run time we can check the number of buffers actually available for the query during run time and choose the plan appropriately.

Can Selectivity be a parameter in PQO

Given that optimizers do account for selectivites, is it a good idea to discuss about selectivity as an error prone parameter in this optimization exercise, like number of buffers, the unaccounted parameter. To answer this question we should be convinced that the errors in the cost model and the underlying stats totally deviate the query plan from the optimal. Moreover, this result is very well established in the picasso paper. Hence it makes absolute sense to discuss about selectivity as a parameter in the Parametric Query Optimization process.

Analysing Query Optimizers' performance wrt to selectivity Estimation errors

As we have discussed that the selectivities of the query are highly error prone, it is a good exercise to check how well the modern optimizers perform. To check this, we will have to analyze the impact of the fine grained choices made by the Query optimizer giving complete credit to the error prone cost model. Hence a conservative approach in this analysis process is to subject the vaious Parametrically optimal set of plans wrt selectivity to the optimizers cost model itself and check it across different selectivities of the query.

Need for FPC

FPC is Foreign Plan Costing ie. evaluating the cost of executing a query through an arbitrary plan. Plan is foreign to the location and hence the terminology. We are interested to find the cost of foreign plans across the entire query selectivity space. Note that we only need the cost of a foreign plan at a particular selectivity. The result set generated by the query is immaterial for the analysis.

Generating cost of foreign plans

To get the cost of a foreign plan, we should be able to force a plan outside its optimal region. In postgres this is possible by tuning the GUC(Grand Unified Configuration) variables.

GUC variables are primarily provided by postgres to control postgres execution and tune it to our desires. These GUC parameters can be set per user per database. When postgres optimizer does the plan selection process the GUC variables alter the code flow in such a way that postgres ultimately selects our plan and returns the cost of the plan. There is a feature called Explain Query which returns the plan chosen by postgres to execute the query. It only returns the plan and does not execute the query as such.

The list of parameters that can be tuned using GUC are available here [5]

Basically how the GUC parameters affect the code flow is as follows

Let's say we disabled seqential scan for a certain query through the command

enable_seqscan(false)

At run time postgres optimizer while costing the plan with sequential scan on a relation, it will check the state of sequential scan global variable and if it is disabled postgres will bump up the cost to infinity. Hence when the plan selection happens the sequential scan gets pruned away by virtue of its cost being infinity.

Other GUC parameters are also handled similarly

GUC context per relation per query

As discussed earlier postgres allows users to set the GUC variables per user per database. But we need more fine grained control over the GUC context because we will encounter situations like Sequential scan should be forced for relation1 but not for relation2 in the same query.

This is achieved through postgres hooks [6]. Hooks are basically function pointers exposed by postgres which any third party library can make use of to provide their custom implementation. Hence at runtime postgres will check if a hook is defined, and if it is defined, postgres transfers control to the third party implementation and relinquishes control later.

Hence we can override the hooks wherever appropriate and provide our implementations thereby forcing postgres to select the plan of our choice. In this process we can make use of hooks to tune the GUC variables appropriately so that code flow happens as we desire.

FPC Implementation

Basically for the FPC module to be invoked we will have to present the query as follows

explain Query **fpc** PathToPlan.XML

On every query invocation, we check for the keyword fpc and if fpc is present we load the fpc library which defines all the hooks and instantiates the hint state. Hint state is a structure that stores the information about the foreign plan ie. The plan that is to be forced. The information that is contained in the hint state is

- Scan to be forced on a base relation
- Joins to be performed among base relations, or among the join relations (relations that areare derived from others)
- Order in which the joins have to be performed.

Hence at run time when postgres has to make a choice about a strategy to be chosen for a particular base/join relation, we set the GUC parameters accordingly via hooks.

Eg. If sequential scan is to be forced on relation1 we set

enable_seqscan(true)

enable_indexscan(false)

enable_indexonlyscan(false)

enable_tidscan(false)

Now when the Optimizer explores the alternate strategies the cost of all other scans except that of the seq scan would have been bumped to infinity and hence seq sca will be forced.

FPC Implementation is slow

With FPC in hand we know the plan we want to force before hand and hence it is not an optimization problem anymore. The combinatorially search difficulty is hence eliminated. Hence it is a natural tendency to expect that the costs be returned many folds faster that the the time taken by the native optimizer. But it turned out that FPC was running slow and hence this project, Accelerating FPC.

Avoiding Index Fetches

I successfully integrated Perf, linux Performance analyser tool to get the execution profiles of the FPC module. The very begining analysis of the profile with small queries showed that fetching the index_paths on a relation consumed a lot of time. Hence investigating in this direction led to the fact that we are fetching indexes on relations where non index scans are to be forced. We fetch all the indexes first and later invoke the hook, which consults the hintstate to only find out that index scan is not the desired scan strategy for this relation and thereby sets the indexList for the relation to NULL. Hence changes are done to check if the index scan is allowed for the relation and only if it is allowed do we fetch the index list for the relation.

This did not show much improvements in the performance because for small query the time that was taken to execute the query itself was in the order of 10ms and this small change didnt contribute much.

Eliminating planFile parsing

On invocation of FPC we parse the plan.xml file and convert it to a string like

IndexScan(customer_address) IndexScan(customer) IndexScan(catalog_sales) IndexScan(date_dim) NestLoop(date_dim catalog_sales customer customer_address) NestLoop(date_dim catalog_sales customer) NestLoop(date_dim catalog_sales) Order((((date_dim catalog_sales)customer)customer_address))

We then load the hint state from this string. Hence changes are made to write the string to the xml file once parsing is done so that on successive invocation of the same plan we donot parse the file anymore and fetch the string dirtectly.

Profiling shows that time taken to parse the total file itself is close to 0ms and hence this change also didnt give any significant improvements.

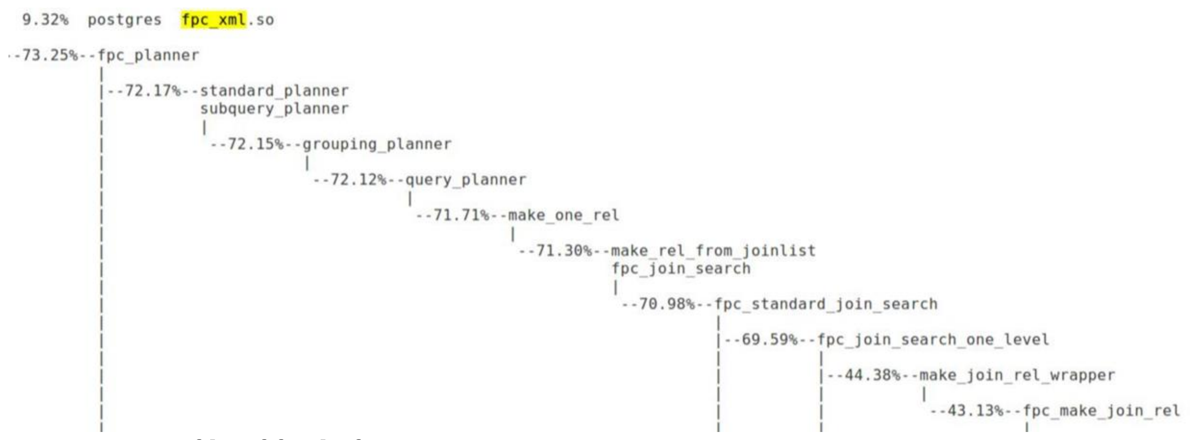
The Candidate for Optimization

Inpecting the profile for a query with 21 base relations revealed that more than 90% of the time is consumed by join_search. This is because we were creating join relations unnecessarily. Idelally we should only create 21 join relations because we know the plan beforehand. Hence changes are made to skip the creation of the join relations that are not required.

This change contributed to performance improvements significantly.

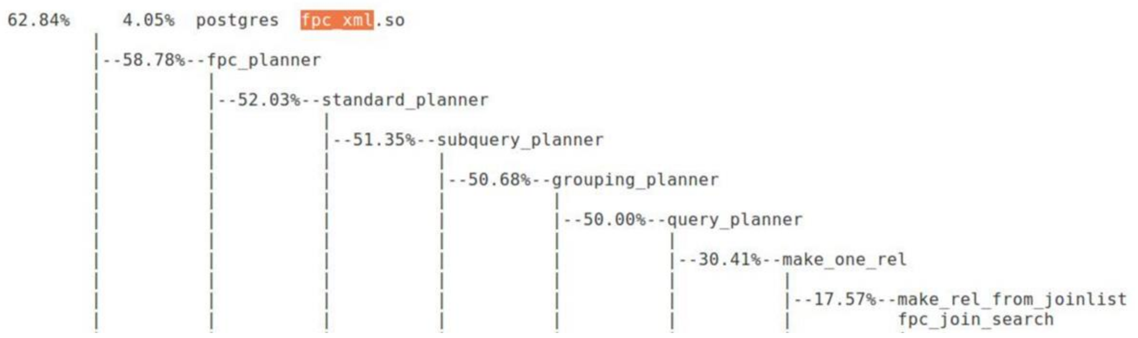
For this particular query with 21 relations the execution time came down from 1400 ms to 11 ms.

This is because the number of join relations that are created is significantly brought down from **126711** join relations to just **21** join relations.



```
  9.32%  postgres  fpc_xml.so
--73.25%--fpc_planner
          |
          |--72.17%--standard_planner
          |          subquery_planner
          |          |
          |           --72.15%--grouping_planner
          |                     |
          |                      --72.12%--query_planner
          |                                |
          |                                 --71.71%--make_one_rel
          |                                           |
          |                                            --71.30%--make_rel_from_joinlist
          |                                                      fpc_join_search
          |                                                      |
          |                                                       --70.98%--fpc_standard_join_search
          |                                                                 |
          |                                                                 |--69.59%--fpc_join_search_one_level
          |                                                                 |          |
          |                                                                 |          |--44.38%--make_join_rel_wrapper
          |                                                                 |          |          |
          |                                                                 |          |           --43.13%--fpc_make_join_rel
          |                                                                 |          |                     |
```

Figure I : Profile of fpc before optimization



```
62.84%     4.05%  postgres  fpc_xml.so
           |
           |--58.78%--fpc_planner
           |          |
           |          |--52.03%--standard_planner
           |          |          |
           |          |          |--51.35%--subquery_planner
           |          |          |          |
           |          |          |          |--50.68%--grouping_planner
           |          |          |          |          |
           |          |          |          |          |--50.00%--query_planner
           |          |          |          |          |          |
           |          |          |          |          |          |--30.41%--make_one_rel
           |          |          |          |          |          |          |
           |          |          |          |          |          |          |--17.57%--make_rel_from_joinlist
           |          |          |          |          |          |          |          fpc_join_search
```

Figure II : Profile of fpc after optimization

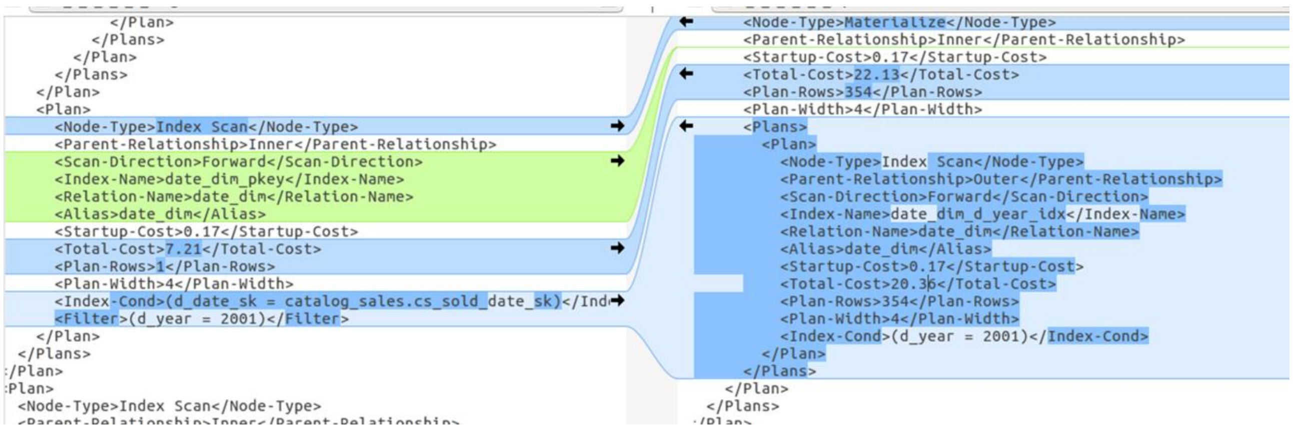
Results

The results are summarized in the table below

TPCDS Query No	7	15	18	19	26	27	91
No Plans	127	14	20	20	120	120	178
No Error Prone Selectivity (total no. Plans)	4(79375)	3(1750)	6(312500)	5(62500)	4(75000)	4(75000)	4(111250)
Cost difference > 0.1%	0	0	1000	0	0	0	0
Cost difference > 0.001%	0	0	14600	0	0	0	0
Original Execution time(ms)	23005675	45922	27923054	4065217	23510300	21887948	80314988
Optimal Execution time(ms)	21274111	39811	22467583	3673301	21250111	19316161	56532230
% decrease in Execution time	7.52	13.3	18.89	9.64	9.61	11.71	29.61
Plan Differences	0	0	8500	0	0	0	0

Table 1: Performance Analysis

There is a plan mismatch for the query No. 81 (highlighted in red in Table 1) because for some plans Postgres does materialization of join relations which we dont tune currently while forcing the plan. Hence this difference creates a significant difference in the cost. Apart from the GUC parameters we dont force, through out the analysis i could not find a plan where the scan and join conditions differ. On scruiting it was found that cost differences occured only for 2 plans where this materialization was happening against our expectations.



```
          </Plan>
        </Plans>
      </Plan>
    </Plans>
  </Plan>
  <Plan>
    <Node-Type>Index Scan</Node-Type>
    <Parent-Relationship>Inner</Parent-Relationship>
    <Scan-Direction>Forward</Scan-Direction>
    <Index-Name>date_dim_pkey</Index-Name>
    <Relation-Name>date_dim</Relation-Name>
    <Alias>date_dim</Alias>
    <Startup-Cost>0.17</Startup-Cost>
    <Total-Cost>7.21</Total-Cost>
    <Plan-Rows>1</Plan-Rows>
    <Plan-Width>4</Plan-Width>
    <Index-Cond>(d_date_sk = catalog_sales.cs_sold_date_sk)</Ind
    <Filter>(d_year = 2001)</Filter>
  </Plan>
</Plans>
:/Plan>
:Plan>
<Node-Type>Index Scan</Node-Type>
```

```
    <Node-Type>Materialize</Node-Type>
    <Parent-Relationship>Inner</Parent-Relationship>
    <Startup-Cost>0.17</Startup-Cost>
    <Total-Cost>22.13</Total-Cost>
    <Plan-Rows>354</Plan-Rows>
    <Plan-Width>4</Plan-Width>
    <Plans>
      <Plan>
        <Node-Type>Index Scan</Node-Type>
        <Parent-Relationship>Outer</Parent-Relationship>
        <Scan-Direction>Forward</Scan-Direction>
        <Index-Name>date_dim_d_year_idx</Index-Name>
        <Relation-Name>date_dim</Relation-Name>
        <Alias>date_dim</Alias>
        <Startup-Cost>0.17</Startup-Cost>
        <Total-Cost>20.36</Total-Cost>
        <Plan-Rows>354</Plan-Rows>
        <Plan-Width>4</Plan-Width>
        <Index-Cond>(d_year = 2001)</Index-Cond>
      </Plan>
    </Plans>
  </Plan>
</Plans>
```

Figure III: Plan Difference for Query 18

Conclusion

All the performance data is shown for TPCDS queries.We are able to see performance improvement because earlier execution time was a factor of $2^{(No.BaseRelations)}$but now it is only a factor of number Base Relations.

References

[1] http://dsl.cds.iisc.ac.in/publications/thesis/rajmohan.pdf

[2] Harish D. P Darera, and J. Haristsa, “on the production of Anorexic Plan Diagrams”, VLDB 2007

[3] Yannis E. Ioannidis, Raymond T Ng, Tymos K. Sellis, “Parametric Query Optimization”

[4] https://www.postgresql.org/docs/9.1/static/runtime-config-query.html

[5] https://wiki.postgresql.org/images/e/e3/Hooks_in_postgresql.pdf