On the generation of TOP-K query execution plans

A Project Report
Submitted in partial fulfilment of the
requirements for the Degree of
Master of Engineering
in
Computer Science and Engineering

by

Hiranya Sonowal

Computer Science and Automation
Indian Institute of Science
Bangalore – 560 012

July 2011

TO

my parents

and

the masses of IISc

Acknowledgements

I would like to thank my advisor Prof. Jayant R. Haritsa for his encouragement and valuable guidance throughout my project work. I would like to sincerely acknowledge his invaluable support in all forms throughout my stay in IISc.

I would like thank all the members of DSL and CSA for their help. I would also like to thank all my friends of IISc from various departments(extending from aerospace to biochemistry) who have made my stay in IISc really memorable. Lastly I thank my parents for their constant support throughout my career.

Abstract

Query optimizers return the best query-evaluation plan from among the strategies available for processing a given query. Currently available query optimizers return only the best plan. An exact plan rank list can be obtained by carrying the K best available sub plans from one level to another of a Dynamic programming(DP lattice) lattice. This technique is known as TOP-K. Basic TOP-K without any optimization takes a lot of time to generate the K best plans. We present here some optimizations which when applied can bring down this time drastically.

One of the applications of plan rank list is in the PlanFill algorithm. PlanFill algorithm can generate accurate plan diagrams if exact plan rank list is used. Thus TOP-K should be used. The main objective of the PlanFill algorithm is to generate plan diagrams in very less time. But our research discloses that if we use TOP-K, efficiency decreases very much. In this paper we show that the inefficiency is due to the small cost difference between TOP-1 and TOP-2. TOP-2 is very much close to TOP-1. Thus the main objective of PlanFill is not achieved and TOP-K cannot be used in PlanFill.

An approximate plan rank list can be obtained from root of the DP lattice. This is called as ROOT-K. ROOT-K, although it delivers an approximate plan rank list can be used in PlanFill. As ROOT-K is very much relaxed in terms of cost as compared to TOP-K, so it is efficient as compared to TOP-K in PlanFill algorithm. On first observation it seems that although ROOT-K is much more efficient in terms of time, it will have less accuracy or in other sense more error prone in terms of plan diagram generation. In contrast to that, our experiments reveal that ROOT-K is actually, to some extent accurate and often efficient in plan diagram generation. In this paper, we reveal the *reason behind ROOT-K's accuracy and efficiency.*

To the best of our knowledge, this is the first thesis researching on the analysis of TOP-K plans. So, as of now there are very limited applications of TOP-K. Although very limited in terms of applications, we still try to highlight some other prospective applications of TOP-K, through this thesis.

Contents

List of Figures

Chapter 1

Introduction

Given an SQL query associated with multiple relations, the query optimizer searches for the best query execution plan. It does it by finding the best join order for successively larger subsets of relations. At each step, sub-plans having neither least cost nor some interesting order are pruned. After termination of search process, the least cost plan from this search tree is chosen. Generating the Plan Rank List at first glance from this search tree seems to be obvious and trivial - sort all the plans available at the root according to their cost.

Unfortunately the plan rank list obtained in such a way is not the actual plan rank list. Some plans may get pruned early in the search process while being compared to the best plan and so may not appear in the final search tree.

In order to get an actual plan rank list we should not throw away the plans at each level. Instead we carry those K-best plans at each level to the next level in the dynamic programming lattice. Through this process of plan enumeration we will explicitly get the actual list of ranked plans in the root node. We call this way of producing a plan list as TOP-K.

1.1 Challenges

As we carry on the K-best plans from one level to the next level of the dynamic programming lattice, the number of computations required increases exponentially. Generating the actual plan rank list in this fashion takes more time and memory. Hence the challenge here is to bring down the plane enumeration time and memory consumed. Although we cannot bring down the amount of memory consumed, we can still bring down the enumeration time to a great extent by performing some optimizations.

1.2 Contribution

In this thesis, we reflect various optimizations performed on the TOP-K procedure, so that the plan enumeration time goes down. We also tried to find some applications of the plan rank list.

We implemented the TOP-K procedure in POSTGRES database engine. We applied the various optimizations required and found that the run time can be actually brought down significantly.

An application of plan rank list is in the PlanFill algorithm [2]. The PlanFill algorithm is a technique to efficiently approximate plan diagrams. Although PlanFill Algorithm was an application of the TOP-K, we discovered that TOP-K is not at all suitable for the algorithm. In other words, TOP-K was a failure for the PlanFill Algorithm. We also discovered the facts behind the failures of TOP-K.

Chapter 2

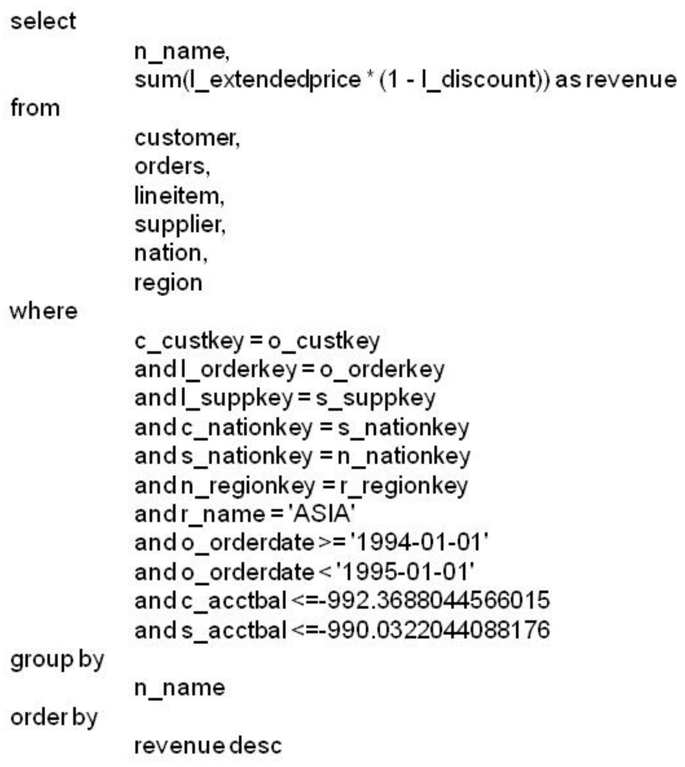
Generating the K best execution plans

2.1 The TOP-K procedure

Modern query optimizers are based on the dynamic programming (DP) based search described in [6], in order to obtain the cost-optimal plan for a given query. Given a query associated with multiple relations, the query optimizer searches for the best query execution plan by finding the best join strategy for successively larger subsets of relations. Further, the join-graph is evaluated to reduce the search space to only meaningful join orders. At each step, sub-plans having neither least cost nor some cheapest interesting order are pruned. After termination of the search process, the least cost plan from the root of this search tree is chosen.

Figure 2.1 shows an SQL query. A dynamic programming lattice is shown in Figure 2.2. Figure 2.3 is best query execution plan for the Query shown in Figure 2.1.

Generating the second-best plan at first glance from the search tree seems to be obvious and trivial - sort all the plans available at the root according to their cost and choose the second-best from this list. Unfortunately, this will not work as some plans may get pruned early in the search process while being compared to the best plan and may not appear in the final root node of the search tree. As an example, consider Figure



```
select
            n_name,
            sum(l_extendedprice * (1 - l_discount)) as revenue
from
            customer,
            orders,
            lineitem,
            supplier,
            nation,
            region
where
            c_custkey = o_custkey
            and l_orderkey = o_orderkey
            and l_suppkey = s_suppkey
            and c_nationkey = s_nationkey
            and s_nationkey = n_nationkey
            and n_regionkey = r_regionkey
            and r_name = 'ASIA'
            and o_orderdate >= '1994-01-01'
            and o_orderdate < '1995-01-01'
            and c_acctbal <=-992.3688044566015
            and s_acctbal <=-990.0322044088176
group by
            n_name
order by
            revenue desc
```

Figure 2.1: SQL Query

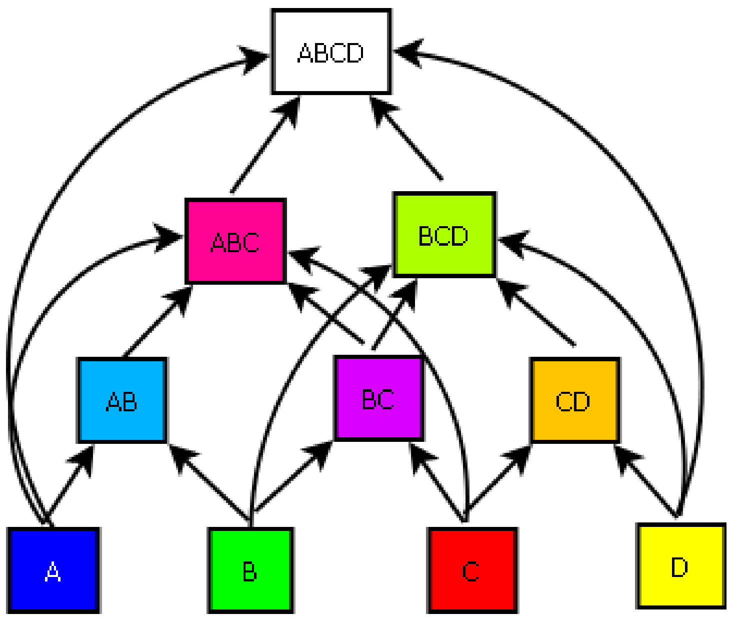


Figure 2.2: Dynamic Programming Lattice

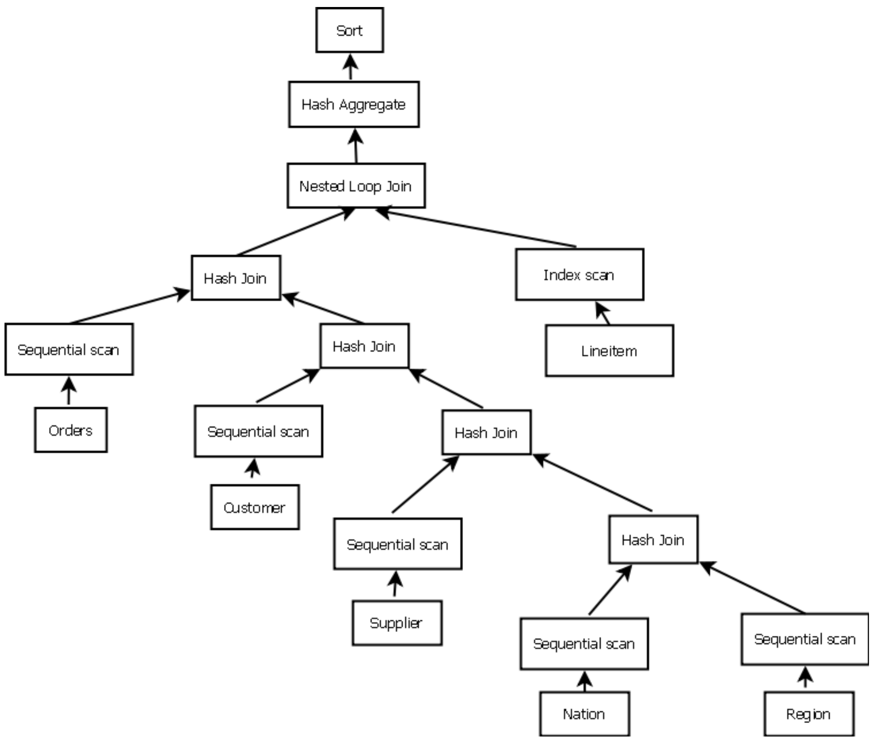


Figure 2.3: Query Execution Plan

2.4 which shows a sample DP-Lattice. The path structure of best plan (solid-red) and its 2nd sibling (dotted-blue), which reached the root (i.e. ABCDE of the DP lattice) are shown here. The cost of respective node is displayed within brackets. A possible candidate of second-best plan e.g. the sub-plan shown in Figure 2.5 got pruned earlier. As can be seen from this figure, the node ADE pruned the join order A$\bowtie$ DE as compared to AD $\bowtie$ E (best plan) for being costlier at this sub-plan level.

Thus we observe that if we don't carry on best plans from one level of the DP lattice to another then we will not get the actual plan rank list. So, we expand the candidate set of sub-plans at each node of the DP lattice. The expanded set contains the second-best strategy for generating the sub-plan represented by that node in addition to the optimizer's default choice. Both these sub-plans are propagated to the higher level of the DP lattice. As shown in Figure 2.6, the dotted lines depict the path of the second-best sub-plans at the respective node e.g. now both the paths for generating the node ADE

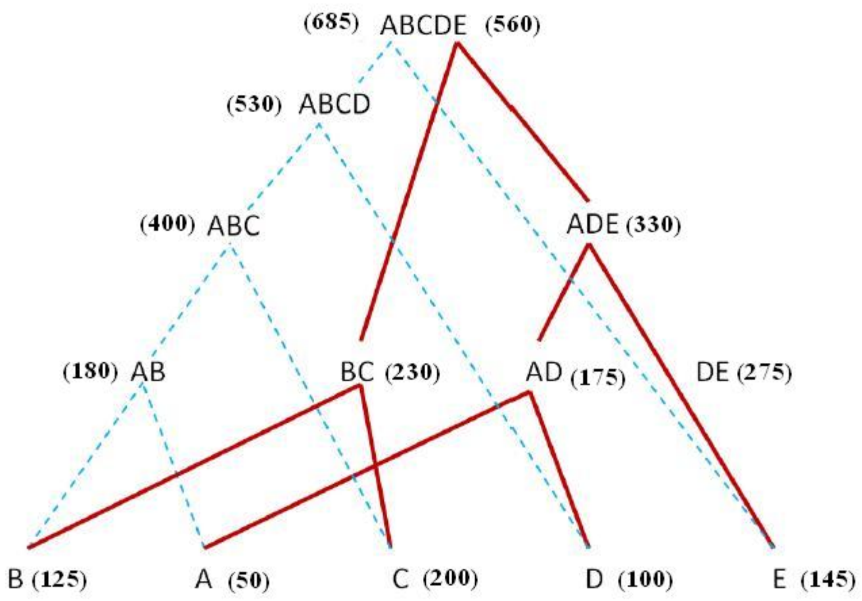


Figure 2.4: Path Structure of best and second best plans at root in a DP lattice

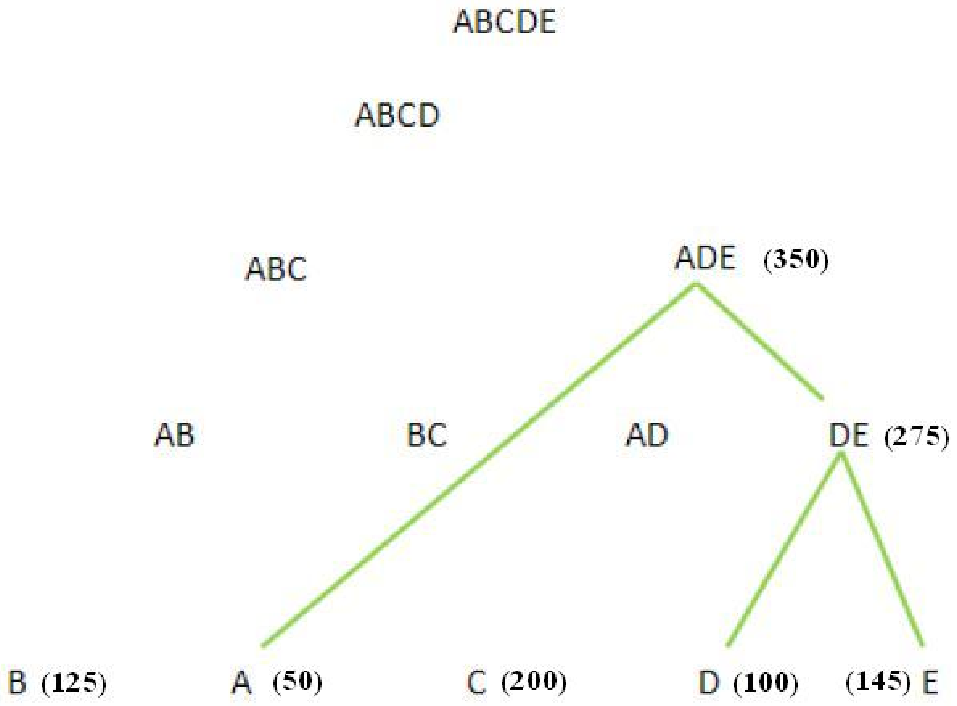


Figure 2.5: Pruned subplan candidate to 2nd best plan

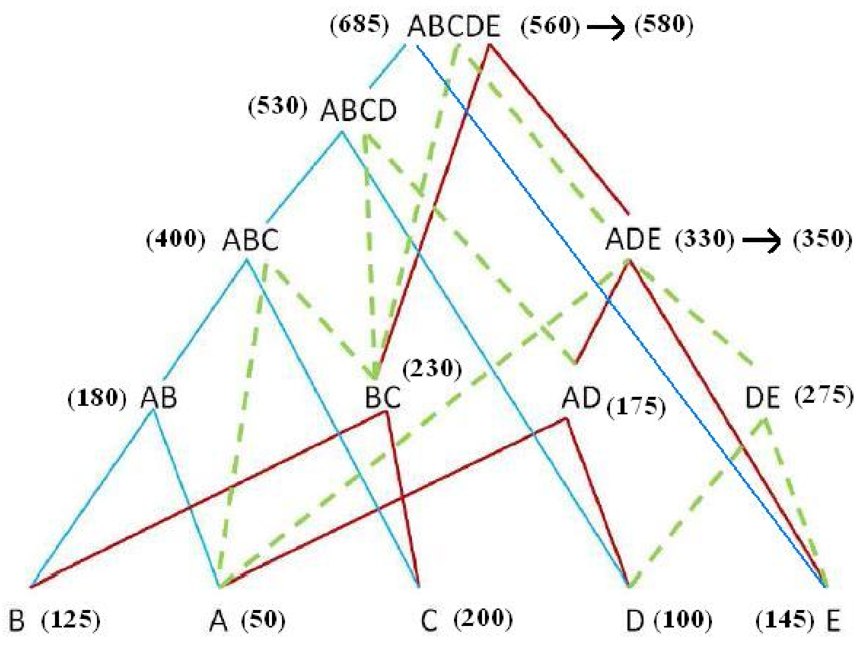


Figure 2.6: TOP-K algorithm

i.e. AD ⋈ E and A ⋈ DE are retained. In this expansion process, the true second-best plan will be explicitly found at the root node. The respective cost of nodes are displayed within brackets and second best cost with an arrow before it. One issue with this approach is that it would now require four times the optimization overhead at each node for enumerating all possible combinations among the pairs of sub-plans arriving from lower level nodes. For example, at node ABCDE, it would require to individually compute the best combination strategy for (AD ⋈ E) ⋈ BC, for (AD ⋈ E) ⋈ CB, for (A ⋈ DE) ⋈ BC, and for (A ⋈ DE) ⋈ CB, considering CB is the 2nd best way of generating (B ⋈ C). However, we can optimize on the above by realizing that the best combination strategy will be the same for all four options i.e. the best join strategy identified for (AD ⋈E) ⋈ BC and (A ⋈ DE) ⋈ BC will be same. This is because the strategies are evaluated in terms of cardinalities and number of distinct values present in BC and ADE, which is independent of the underlying sub-plans generating them. Therefore, only one optimization, say (AD ⋈ E) ⋈ BC, needs to be carried out, and the results can be reused

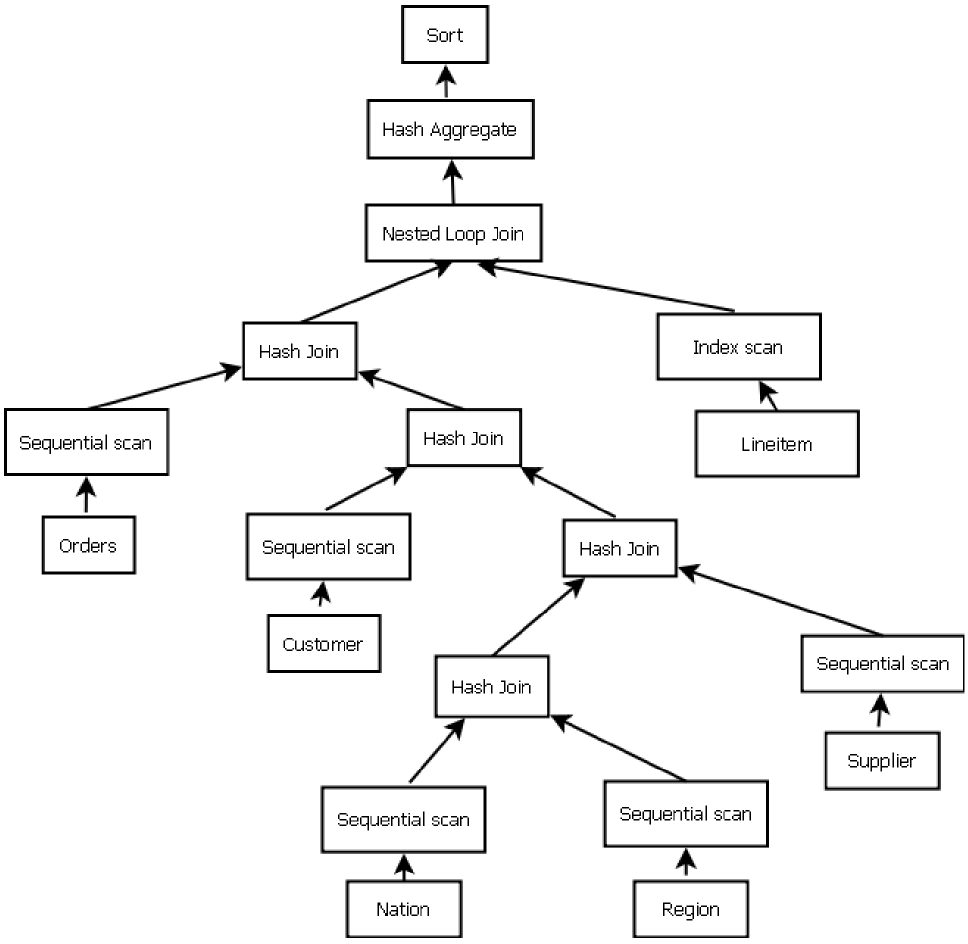


Figure 2.7: TOP-2

for the other three pairs. This method of inheriting cost of best-pair cannot be applied directly to sub-plans with different interesting orders. In case, we encounter sub-plans having different interesting orders, at least one from each different interesting orders have to be costed explicitly. But practically for TPC-H templates, we have observed that during DP procedure the number of sub-plans having interesting orders is much lesser compared to the number of sub-plans without any interesting order.

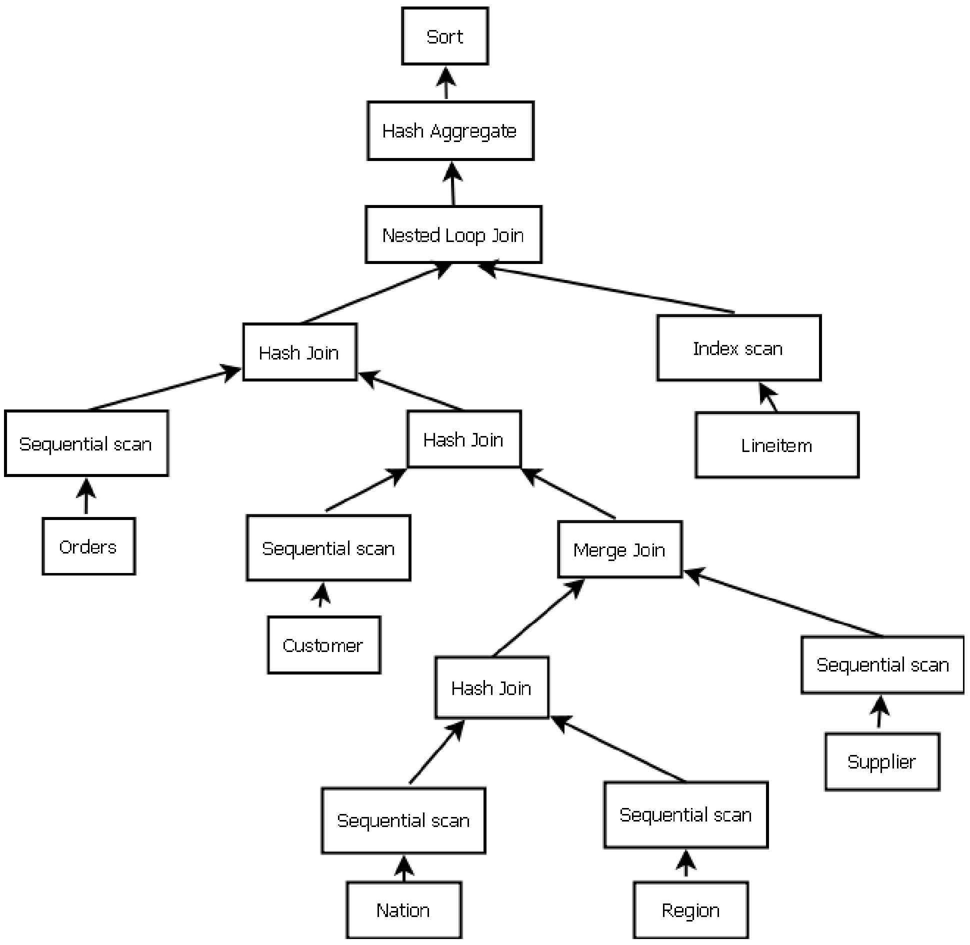


Figure 2.8: TOP-3

Chapter 3

Optimizations performed

As plain plan enumeration in TOP-K takes a lot of time, hence in order to improve performance, we derived some optimizations which can substantially bring down the plan enumeration timings. The various optimizations performed were as follows:

3.1 1st optimization: Cost inheritance

As already mentioned in the previous chapter, at node ABCDE, we would individually compute the best combination strategy for (AD $\bowtie$ E) $\bowtie$ BC, for (AD $\bowtie$ E) $\bowtie$ CB, for (A $\bowtie$ DE) $\bowtie$ BC, and for (A $\bowtie$ DE) $\bowtie$ CB, considering CB is the 2nd best way of generating (B $\bowtie$ C). However, we can optimize on the above by realizing that the best combination strategy will be the same for all four options i.e. the best join strategy and join order identified for (AD $\bowtie$ E) $\bowtie$ BC and (A $\bowtie$ DE) $\bowtie$ BC will be same. This is because the strategies are evaluated in terms of cardinalities and number of distinct values present in BC and ADE, which is independent of the underlying sub-plan generating them. Therefore, only one optimization, say (AD $\bowtie$ E) $\bowtie$ BC, needs to be carried out, and the results reused for the other three pairings. This is reflected in Figure 3.1 where we optimize (i) only and inherit the the cost for (ii), (iii), (iv).

But for a word of caution, this method of inheriting cost of best-pair cannot be applied directly to sub-plans with interesting orders. If there are interesting paths then

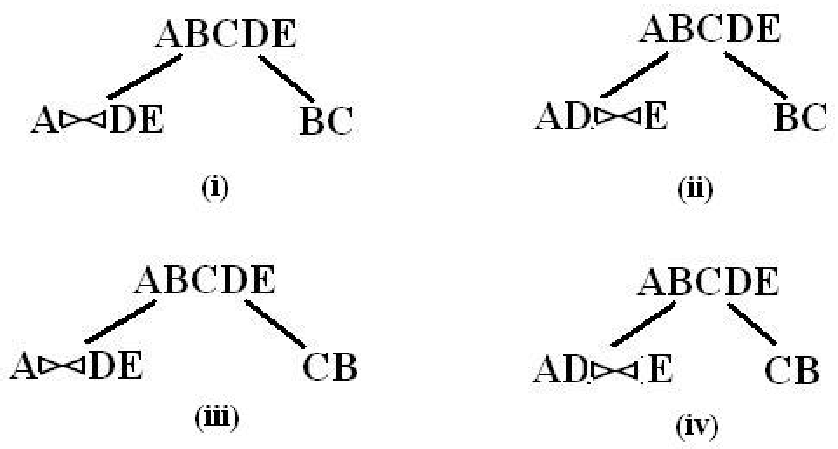


Figure 3.1: Cost Inheritance

we need to optimize the heads of left and right side relations and then inherit the other cost.

3.2 2nd optimization: Hash join once optimization-inheritance

Although a path may be interesting, but in hash join it is of no use. So in hash join whatever may be the path (normal or interesting) it is always treated as normal. So the hash join needs to be optimized only once and rest can inherit it.

3.3 3rd optimization: Selective Combination

Another optimization used here was that if the first optimization is directly rejected then we can reject all the others without even inheriting the cost as they are obvious

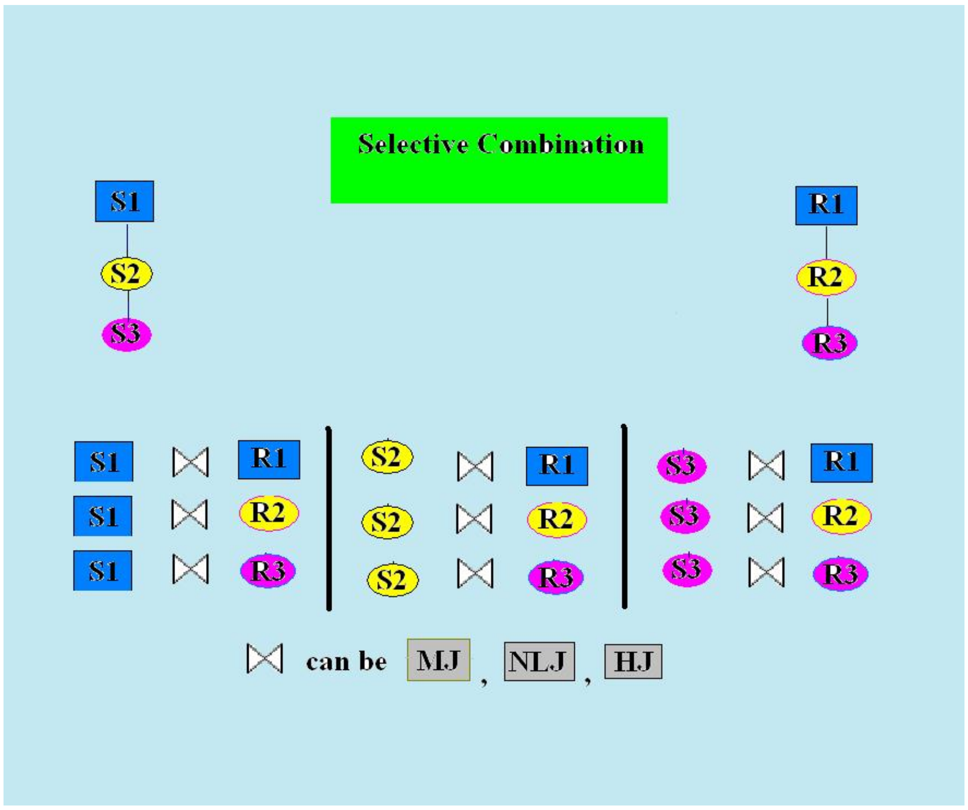


Figure 3.2: Selective Combination

to be rejected. If the best boring plan from left-hand relation and right-hand relation is rejected, then all the wagons can be rejected. But these will not work if the 1st one is accepted and second one is rejected. We cannot straight forward reject the remaining plans. Suppose(as shown in Figure 3.2) there are left relation has trains S1,S2,S3 and right relation has trains R1,R2,R3. The rectangles can be considered as the engines and eclipses as wagons. Consider all are boring. Now if S1-Merge Join-R1 is rejected for being costly(suppose S1-Hash Join-R1 had a lower cost) we can straight forward reject other costing without even inheriting the costs. But if S1-Merge Join-R1 is not rejected but accepted and S1-Merge Join-R2 is rejected, we can throw away S1-Merge Join-R3 optimization but we cannot throw away all optimizations. Although S1-Merge Join-R2 was rejected, it might be the case S2-Merge Join-R1 might be less costly then S1-Merge Join-R2. So we have to inherit and see whether it should be rejected or retained. Such selective inheritance and rejectance also reduces a large number of unwanted inheritances and enumerations.

Here comes the importance of order of join algorithms. PostgreSQL tries first Merge Join then Nested Loop Join and then Hash Join for enumeration. Now if we get all the best subplans at Merge Join itself then we can do away with the enumerations for Nested Loop Join and Hash Join. So, getting the correct order of plan enumeration also matters. This enumeration is currently hard coded in PostgreSQL. But after extensive experiments we found that the order Hash Join, Merge Join and Nested Loop Join proves to be most fruitfull.

Another better approach to deal this problem in future is to make it self learning. For example if in the current step we find that the Merge Join contained the best plans then in next level enumeration we can start with Merge Join itself.

3.4 4th optimization : Selective inheritance in Interesting paths

In case of merge joins, interesting paths do become necessary. But sometimes the interesting path does not contain the sorted path which is actually necessary for the merge join. So in such a case the interesting path simply acts like a normal path and hence its cost can be inherited.

3.5 5th optimization : POSTGRES specific optimizations

It was predicted that if we have cost inheritance as the only optimization then the time taken for plan enumeration for finding K-best plans will be almost same as the plan enumeration time for finding only the single best plan. But it was observed that cost inheritance reduces the plan enumeration time but not to a great extent. The reason behind this was that the costing function in postgres did not take up much time. In postgres, functions like add_path took up much of the time for data structure maintenance. So it was observed that we should not only use cost inheritance but also restrict the number of times the function add_path was called.

The general procedure present in postgres was to find the cost of a plan enumeration and then call the add_path function to find whether it will be accepted or not. So have to called the add_path functions almost everytime.

In order to do away with this, we changed the code flow of the in postgres. The basic procedure we took up, was to find the cost of various plan enumerations through cost inheritance in a batch. Then we sorted all the cost and then called the add_path function. This way of flow of code restricted the calls to add_path function, as a result of which we were able to bring down the plan enumeration time to a greater extent.

It is to be noted here that although selective combination brings down the plan enumeration time, this 5th optimization further brings down the plan enumeration time.

Chapter 4

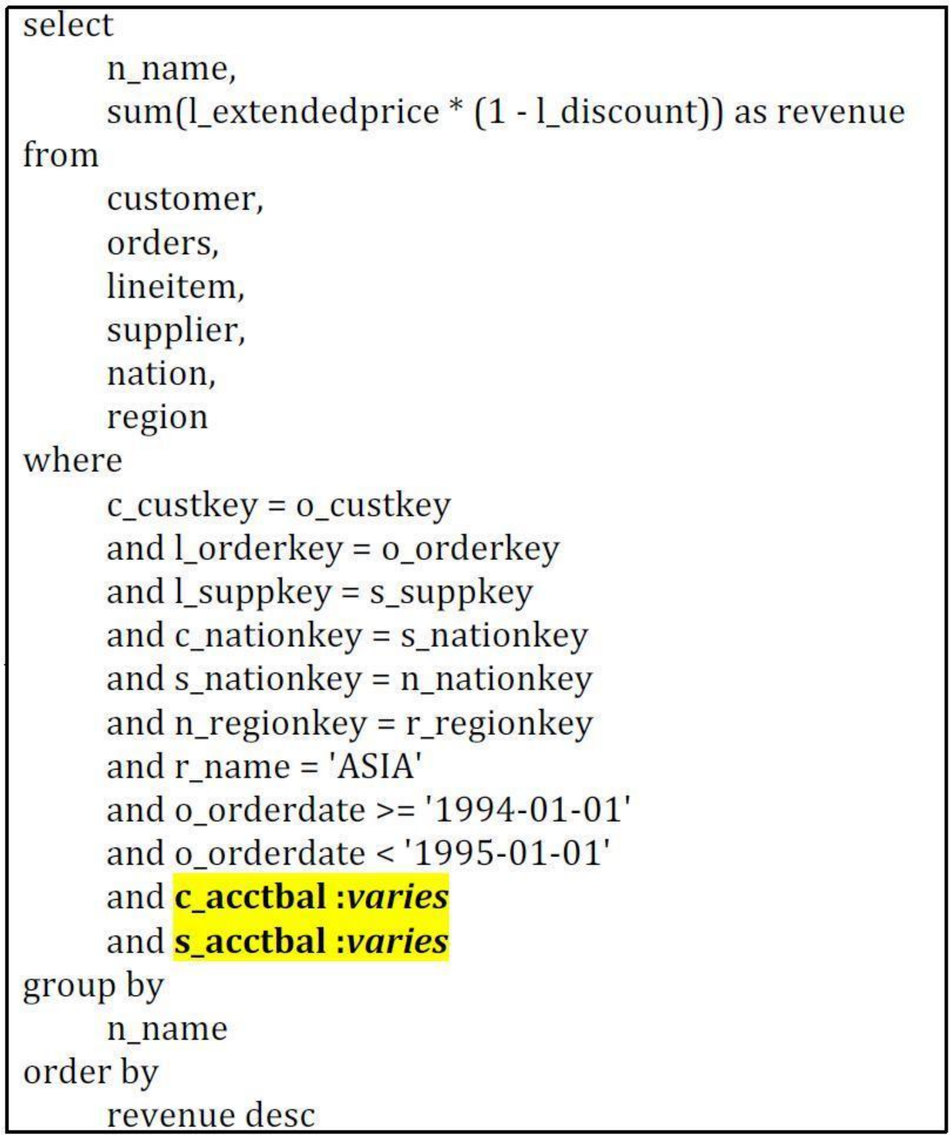
Application of TOP-K plan rank list

Before introducing the PlanFill Algorithm, we introduce the general plan diagram generation procedure.

4.1 Plan Diagram Generation

For a given d-dimensional query template and a plot resolution of r, the Picasso tool [18] generates r^d queries that are either uniformly or exponentially (user's choice) distributed over the selectivity space. Then, for each of these query points, based on the associated selectivity values, a query with the appropriate constants instantiated is submitted to the query optimizer to be explained that is, to have its optimal plan computed. After the plans corresponding to all the points are obtained, a different color is associated with each unique plan, and all query points are colored with their associated plan colors. Then, the rest of the diagram is colored by painting the region around each point with the color corresponding to its plan. The diagram so obtained is know as plan diagram.

For example, consider QT5, the parameterized 2D query template shown in Figure 4.1 , based on Query 5 of the TPC-H benchmark . Here, selectivity variations on the CUSTOMER and SUPPLIER relations are specified through the c acctbal :varies and s acctbal :varies predicates, respectively. The associated plan diagram for QT5 is shown in Figure 4.2, produced with the Picasso optimizer visualization tool [18] on PostgreSQL



```
select
        n_name,
        sum(l_extendedprice * (1 - l_discount)) as revenue
from
        customer,
        orders,
        lineitem,
        supplier,
        nation,
        region
where
        c_custkey = o_custkey
        and l_orderkey = o_orderkey
        and l_suppkey = s_suppkey
        and c_nationkey = s_nationkey
        and s_nationkey = n_nationkey
        and n_regionkey = r_regionkey
        and r_name = 'ASIA'
        and o_orderdate >= '1994-01-01'
        and o_orderdate < '1995-01-01'
        and c_acctbal :varies
        and s_acctbal :varies
group by
        n_name
order by
        revenue desc
```

Figure 4.1: Query Template QT5

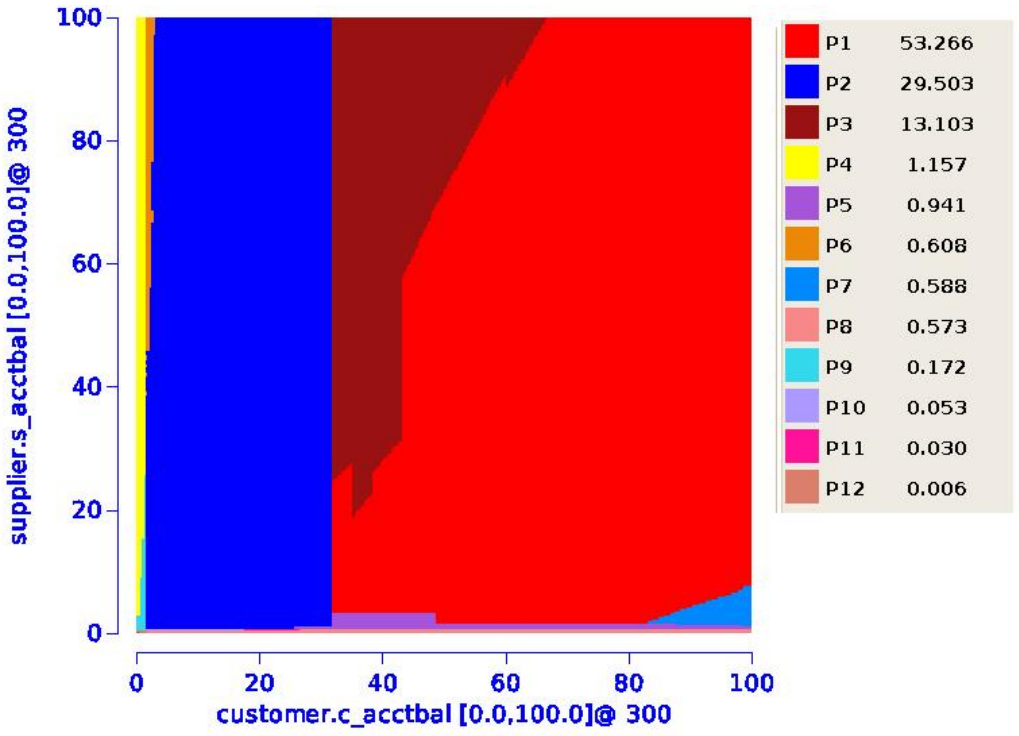


Figure 4.2: Plan Diagram for QT5

engine.

This approach of producing plan diagram is absolutely acceptable for low dimensions and resolutions. However, it becomes very much expensive for higher dimensions and fine-grained resolutions due to the exponential growth in computational overheads. For example,a 2D plan diagram with a resolution of 1000 on each selectivity dimension requires invoking the optimizer 10^6 times. Even if we take an estimate of about half-second per optimization, the total time required to produce the picture is close to a week!. Therefore, although plan diagrams have proved to be extremely useful, their high-dimension and fine-resolution versions pose serious computational challenges. So the need for Approximate production of plan diagrams became very much obvious. The PlanFIll algorithm is one of them.

The PlanFill algorithm uses the Plan Rank List(PRL) and Foreign Plan Costing features(FPC). Specifcally, it assumes that on each invocation, the optimizer returns both best and second-best plans (i.e. PRL with k = 2). PlanFill achieves speedup by reducing total number of optimizations, carried out to produce the plan diagram. Also, this technique is the most intrusive in nature and significantly modify the normal course of DP. We now take a detor to present the strategies incorporated for (a) generating second-best plan (PRL) and (b) costing any plan at any point (FPC), in the optimizer kernel.

4.1.1 Foreign Plan Costing

A plan tree is costed by optimizer in a bottom-up procedure. A leaf node, which corresponds to a base relation is costed according to the estimated number of rows (cardinality), that can participate in the query from that relation. Any intermediate node is costed de-pending on the estimated number of rows of its children (or child in case of a unary node). Therefore, the cost of a plan tree is primarily governed by the cardinality estimates of the base relations.

Each query in the selectivity space is uniquely identified by the associated base relation selectivities. Therefore, given a plan tree for a particular query location q_1, if we simply change the cardinality estimates of base relations (leaf nodes), we will obtain a plan for a different query q_2 in the same selectivity space. Note that, this plan may not be optimal at the new query location but if we carry out the costing procedure on this modified plan, we will eventually get the cost of that plan at this new location. This is supported by the fact that, primary inputs of plan-costing are the base relation cardinality estimates.Therefore our strategy for costing a plan P at a foreign query location q_f is as follows:

1. Get the base relation selectivities associated with the query q_f .

2. In the plan tree of P, visit the appropriate leaf nodes and inject the constants

corresponding to the new selectivities into those nodes, by modifying the associated restriction clauses.

3. Cost this modified plan tree through the usual costing process of optimizer.

The PlanFill algorithm for a 2D query template is shown in Figure 4.3. The algorithm starts with optimizing the query point q(x_{min}; y_{min}) corresponding to the bottom-left query point in the plan diagram. Let p_1 be the optimizer-estimated best plan at q, with cost c_1(q), and let p_2 be the second best plan, with cost c_2(q). We then assign the plan p_1 to all points $q^{'}$ in the first quadrant relative to q as the origin, which obey the constraint that $c_1(q^{'}) \leq c_2$(q). After this step is complete, we then move to the next unassigned point in row-major order relative to q, and repeat the process, which continues until no unassigned points remain.

This algorithm is predicated on the Plan Cost Monotonicity (PCM) assumption, that the cost of a plan is monotonically non-decreasing with the increasing selectivity, throughout the selectivity space, which is true in practice for most query templates on all industrial-strength query optimizers.

The following theorem proves that the PlanFill algorithm will exactly produce the true plan diagram P without any approximation whatsoever. That is, by definition, there are no plan-identity and plan-location errors.

THEOREM: *The plan assigned by PlanFill to any point in the approximate plan diagram A is exactly the same as that assigned in P.*

Proof: Let $P_0 \subseteq$ P be the set of points which were optimized. Consider a point $q^{'} \in$ P$\backslash P_0$ with a plan p_1. Let q $\in P_0$ be the point that was optimized when $q^{'}$ was assigned the plan p_1. Let p_2 be the second best plan at q.

```
PlanFill (QueryTemplate $QT$)
    1. Let A be an empty plan diagram.
    2. Set $q = (x_{min}, y_{min})$
    3. while ($q \neq null$)
        (a) Optimize query template $QT$ at point $q$.
        (b) Let $p_1$ and $p_2$ be the optimal and second-best plan at $q$,
            respectively.
        (c) for all unassigned points $q'$ in the first quadrant of $q$
                if ($c_1(q') \leq c_2(q)$), assign plan $p_1$ to $q'$
        (d) Set $q$ = next unassigned query point in A
    4. Return A
    5. End Algorithm
```

Figure 4.3: PlanFill Algorithm

For the sake of contradiction, let p_k (k$\neq$1), be the optimal plan at $q^{'}$. We know that for a cost-based optimizer, $c_k(q^{'}) < c_1(q^{'})$. This implies that $c_k(q^{'}) < c_2$(q) (due to the algorithm). Using the PCM property, we have c_k(q)$\leq$ $c_k(q^{'})\Rightarrow$ c_1(q) $\leq$ c_k(q) $<$ c_2(q). This means that p_2 is not the second best plan at q, a contradiction. ■

Figure 4.3 shows the planfill algorithm. Figure 4.4 depicts how an unoptimised point can be assigned a plan through PlanFill algorithm. Figure 4.5 shows how full region can be assigned plans by optimizing a single point(provided we are lucky). Figure 4.6 reflects the validity of the PlanFill algorithm as already proved.

Denoting the exact plan diagram as P and the approximation as A, there are two categories of errors that arise in the process for approximating plan diagram:

Plan Identity Error (PIE): It refers to the possibility of the approximation missing out on a subset of the plans present in the exact plan diagram. It is computed as the percentage of plans lost in A relative to P.

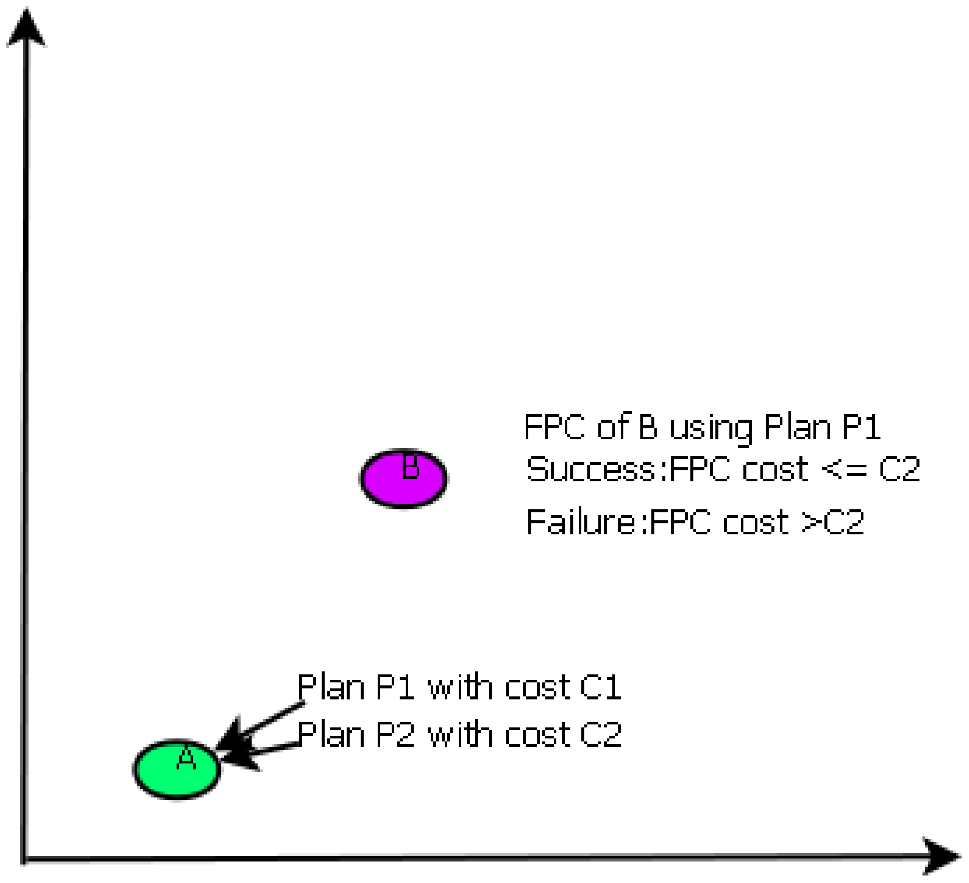


Figure 4.4: PlanFill illustration

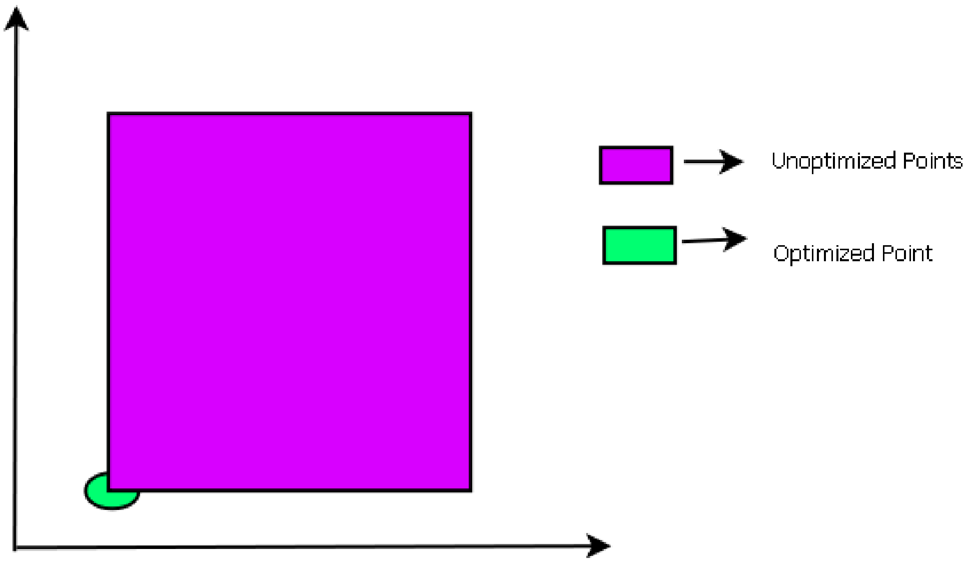


Figure 4.5: PlanFill Algorithm success

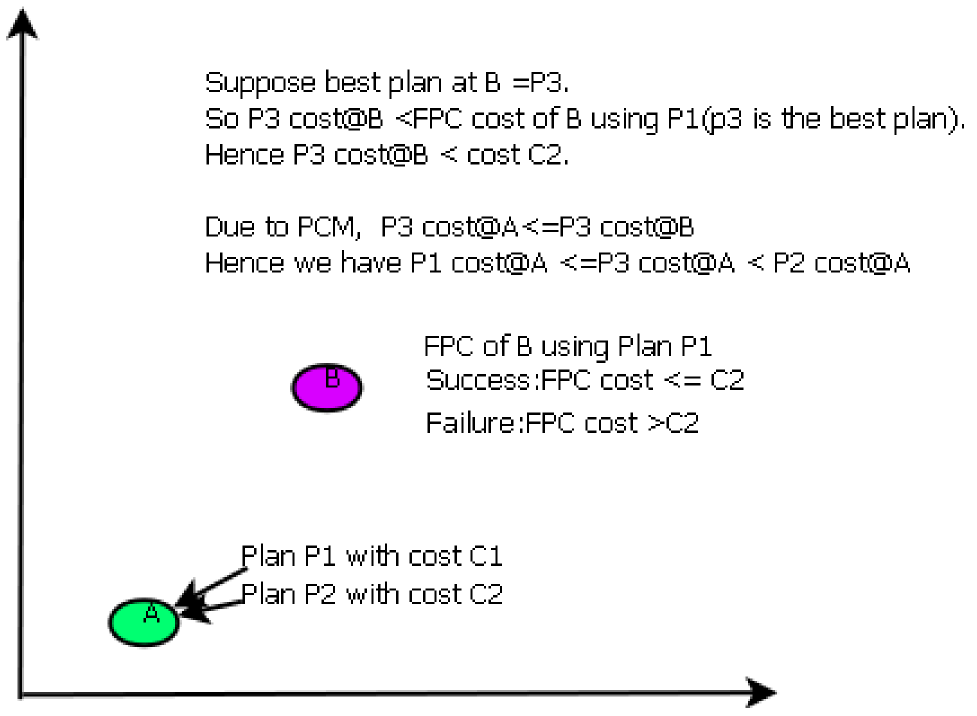

Suppose best plan at B =P3.
So P3 cost@B <FPC cost of B using P1(p3 is the best plan).
Hence P3 cost@B < cost C2.
Due to PCM, P3 cost@A<=P3 cost@B
Hence we have P1 cost@A <=P3 cost@A < P2 cost@A
FPC of B using Plan P1
Success:FPC cost <= C2
Failure:FPC cost >C2
B
Plan P1 with cost C1
Plan P2 with cost C2
A

Figure 4.6: Validity of PlanFill Algorithm

Plan Location Error (PLE): It refers to the possibility of incorrectly assigning plans to query points in the approximate plan diagram. It is computed as the percentage of incorrectly (relative to P) assigned points in A.

Chapter 5

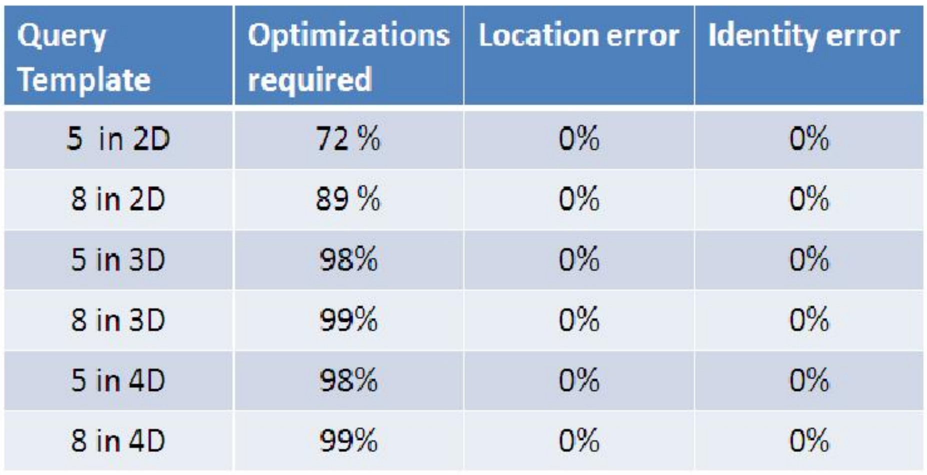
Experiments for PlanFill algorithm

The testbed used in our experiments is the Picasso Query Optimizer visualization tool [20], executing on a Sun Ultra 20 workstation equipped with an Opteron Dual Core 2.5GHz processor, 4 GB of main memory and 500 GB of hard disk, running the UBUNTU 10.04 operating system. The experiments were conducted over plan diagrams produced from a variety of multi-dimensional TPC-H based query templates. In our discussion, we use QTx to refer to a query template based on Query x of the TPC-H benchmark [17], The TPC-H database was of size 1GB. The database Query Optimizer used for the experiments was Postgres [16].

5.1 PlanFill using TOP-K

As an application of plan rank list, the PlanFill algorithm was used to produce plan diagrams. Using TOP-2 although we can produce the exact plan diagram, we did not get much efficiency. The number of optimizations required to be done while using TOP-2 was some times even greater than 90%. This grave failure of TOP-2 in terms of efficiency can be attributed to the very low cost difference between TOP-1 and TOP-2.

Although it was predicted that we can draw accurate plan diagrams using TOP-K, but due to the failure of TOP-K in terms of performance we cannot use TOP-K. Hence we try to use a plan rank list obtained through a method called ROOT-K. The method



Query Template	Optimizations required	Location error	Identity error
5 in 2D	72 %	0%	0%
8 in 2D	89 %	0%	0%
5 in 3D	98%	0%	0%
8 in 3D	99%	0%	0%
5 in 4D	98%	0%	0%
8 in 4D	99%	0%	0%

Figure 5.1: PlanFill Results using TOP-K

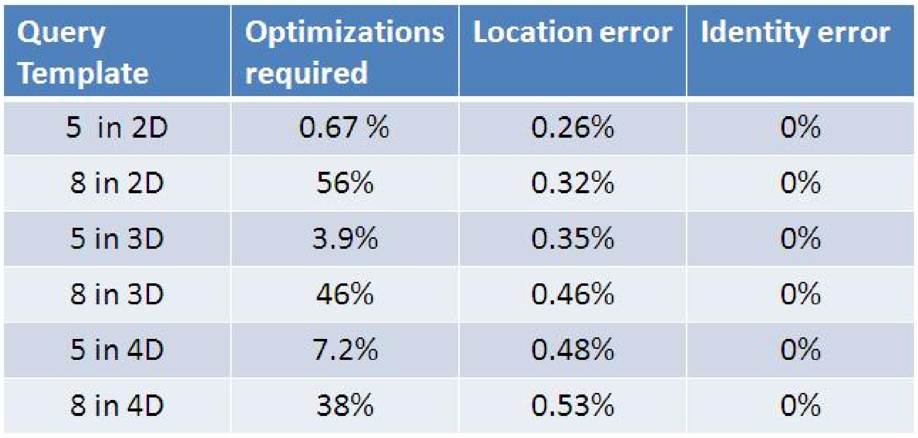
of obtaining the plan rank list from the root node of a simple DP lattice(In the simple DP lattice we do not carry k-best plans from one level to another) is known as ROOT-K. The plan rank list obtained through ROOT-K is not the actual plan rank list. In other words it is a relaxed form of plan rank list as compared to TOP-K in terms of cost.

5.2 PlanFill using ROOT-K

So PlanFill was implemented again using ROOT-2. The results this time were very much satisfactory in most of the cases. The success behind ROOT-2 in being efficient in terms of plan diagram generation can be attributed to the cost difference between ROOT-1 and ROOT-2. As already mentioned the cost gap between ROOT-1 and ROOT-2 is larger than TOP-2. So ROOT-2 became successful.

For example, at a certain point in the query space the TOP-1(TOP-1 and ROOT-1 are same) cost was 57577.37, TOP-2 cost was 57577.39 and ROOT-2 cost was 57753.65. Thus we observe that percentage wise ROOT-2 cost is not much different from TOP-2 but actually TOP-2 cost is almost same as TOP-1 and the gap between ROOT-2 and TOP-1 is very much huge compared to the gap between TOP-2 and TOP-1.

Figure 5.3 shows the frequency distribution graph of ROOT-2/TOP-2 cost for Query



Query Template	Optimizations required	Location error	Identity error
5 in 2D	0.67 %	0.26%	0%
8 in 2D	56%	0.32%	0%
5 in 3D	3.9%	0.35%	0%
8 in 3D	46%	0.46%	0%
5 in 4D	7.2%	0.48%	0%
8 in 4D	38%	0.53%	0%

Figure 5.2: PlanFill Results using ROOT-K

Template 5. On the x-axis lies the ROOT-2 by TOP-2 ratio and on the y-axis lies the frequency of points.

Figure 5.4 shows the points where optimization took place in the selectivity space for QT5(using TOP-2). This figure can be actually thought of as the black and white negative of the Figure 4.2. The x-axis and y-axis for this figure are same as that of Figure 4.2.

Figure 5.5 shows the query points in selectivity space where ROOT-2 by TOP-2 ratio is extremely small. The x-axis and y-axis for this figure are same as that of Figure 4.2.

Figure 6.1 and Figure 5.2 shows the number of optimizations required and errors present for the QT5 and QT8 in various resolutions and dimensions. In 2D the resoltion used was 1000×1000, for 3D the resolution used was 100×100×100 and for 4D the resolution used was 30×30×30×30.

Two errors that we take into account are the plan location error and the plan identity error. As Root-2 was very much relaxed in terms of cost, hence it was expected that the plan location error and plan identity error will be very much high. But surprisingly the error was not much. Actually in most of the cases the Plan identity error was zero and the plan location error was less than 1%. This was very stunning.

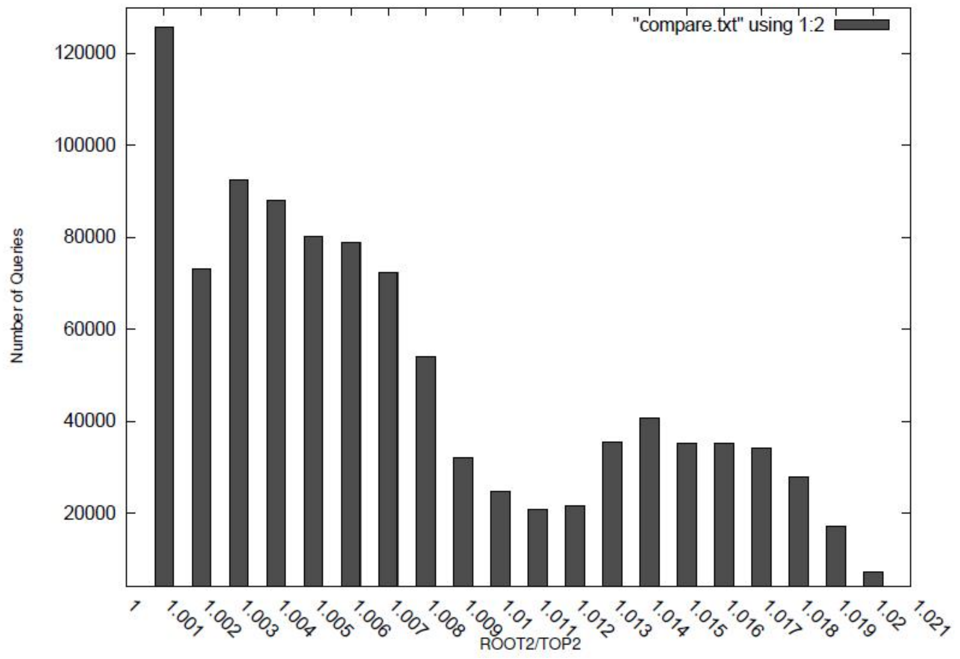


Figure 5.3: ROOT2 by TOP2 cost Ratio

5.3 Reasons behind success of ROOT-K and failure of TOP-K

Although we found that ROOT-2 was far away from ROOT-1 as compared to TOP-1, we also found that this was not so in the whole selectivity space. There were regions in the selectivity space where the TOP-2 approached ROOT-2. On investigating these regions, it was observed that these regions were actually the plan transition boundaries(region where best plan changes). This was the turning point behind the accuracy of ROOT-2. As near the plan boundaries TOP-2 approached ROOT-2, planfill in these regions behaved as if we are using TOP-2 instead of ROOT-2. As we already know that TOP-2 gives accuracy, hence ROOT-2 also gave accuracy.

Thus ROOT-2 became successful in PlanFill.

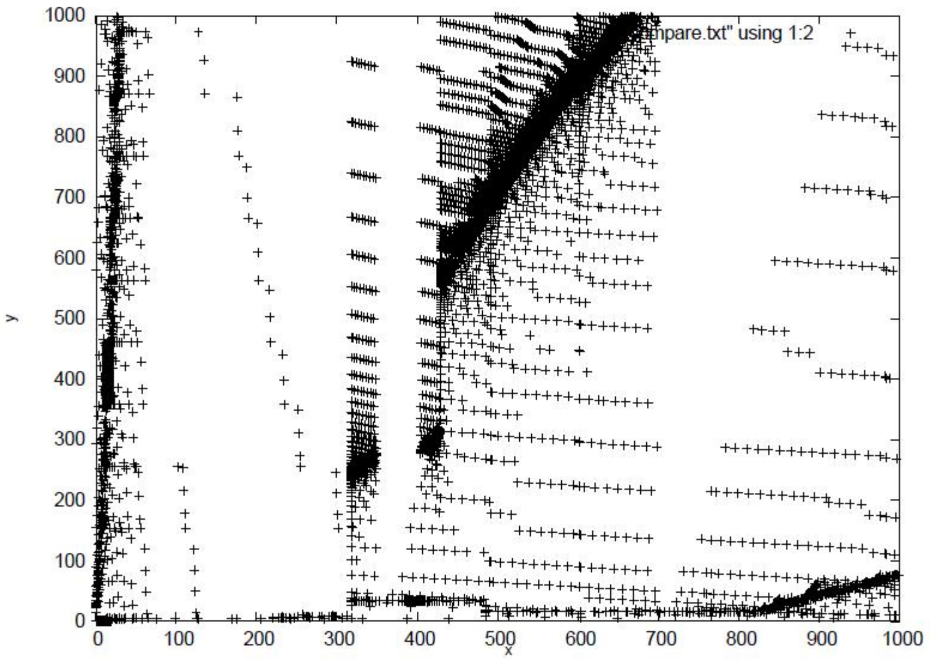


Figure 5.4: Query Points where optimizaiton took place

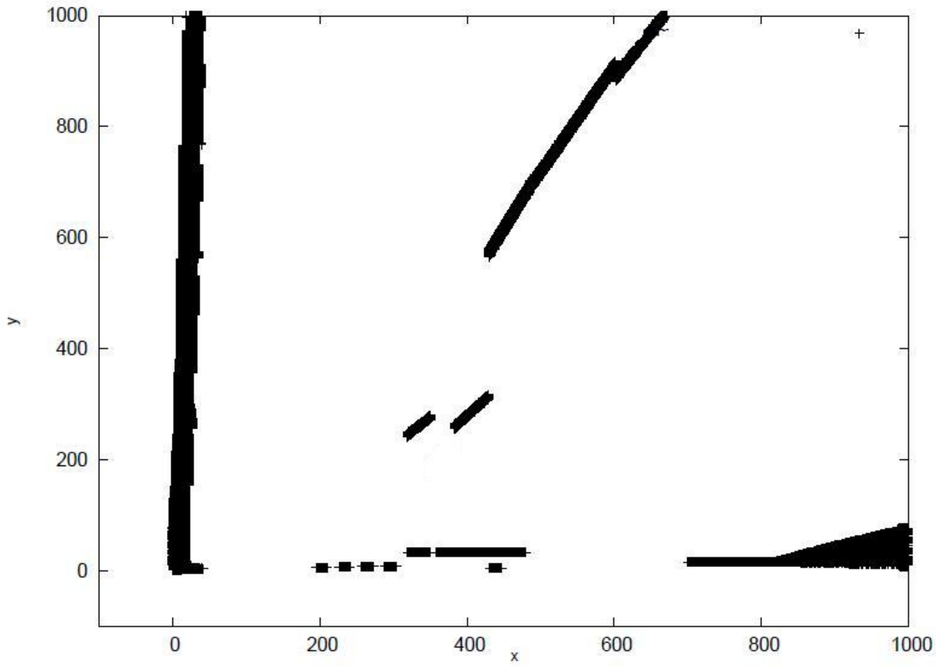


Figure 5.5: Query Points where TOP2 approaches ROOT2 in the selectivity space

Chapter 6

Experiments on TOP-K and ROOT-K

In this chapter we show the results of various optimizations performed on TOP-K.

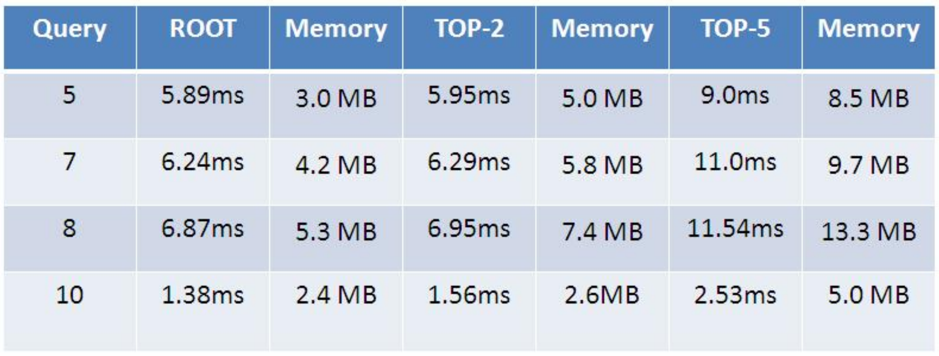
6.1 Pre-Optimization results

Without applying any optimization the time taken for generating the top-k plans is very much as expected. The last row of the table shown in Figure 6.2 (Query 5) shows the time taken for generating top-k execution plans without using any optimization.

6.2 Post-Optimization results

After applying the various optimizations mentioned above we could substantially bring down the plan enumeration time. e.g. The time required for finding 2^{nd} best plan is almost same as that of the best plan. Figure 6.1 shows the observed statistics.

Figure 6.2(Query 5) shows the time taken by different enumerations. The default enumeration sequence in Postgres is merge join - nested loop join - hash join. There can be different sequence enumerations. Out of the various enumerations we show the best one(hash join - merge join - nested loop join) along with two other enumeration sequence.



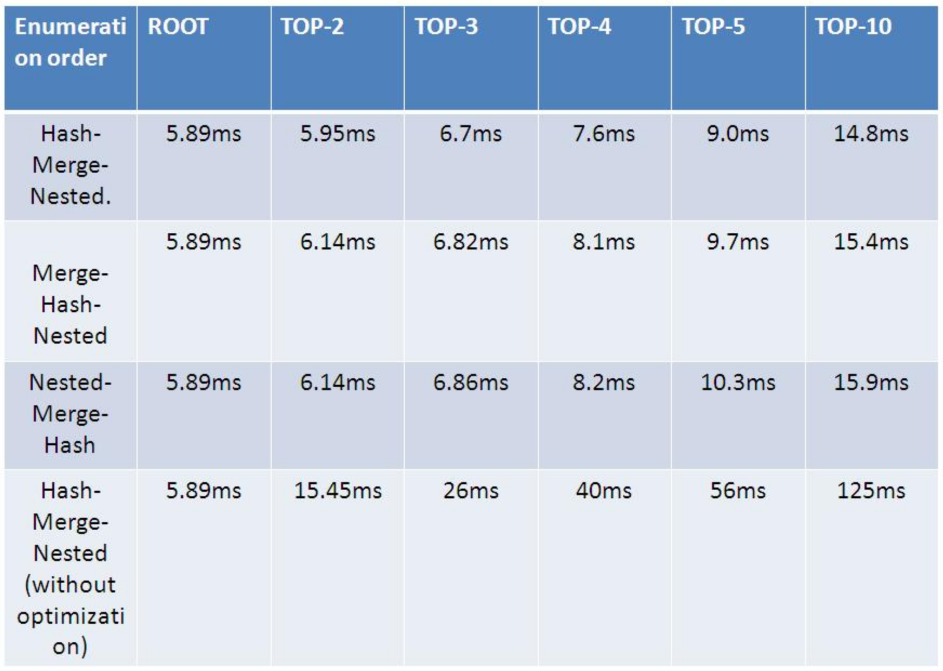
Query	ROOT	Memory	TOP-2	Memory	TOP-5	Memory
5	5.89ms	3.0 MB	5.95ms	5.0 MB	9.0ms	8.5 MB
7	6.24ms	4.2 MB	6.29ms	5.8 MB	11.0ms	9.7 MB
8	6.87ms	5.3 MB	6.95ms	7.4 MB	11.54ms	13.3 MB
10	1.38ms	2.4 MB	1.56ms	2.6MB	2.53ms	5.0 MB

Figure 6.1: Enumeration time taken and memory consumed by TOP-K

It can be observed that if we know beforehand the best enumeration sequence(GOD's sequence) then we can bring down the enumeration time to an extent.

6.3 Comparison between ROOT-K and TOP-K

Experiments were also performed to find the characteristics between ROOT-K and TOP-K. The cost of ROOT-K and TOP-K of various query templates were compared across the extreme corners in the selectivity space plus the midpoint of the selectivity space. It was observed that ROOT-K is often steeper than TOP-K. But if we relax the cost to around 20% then ROOT-10 becomes almost equivalent to TOP-10. So if there is a relaxation of cost of around 20% we can use ROOT-10 instead of TOP-10. Figure 6.3 to Figure 6.6 (performed at 1000×1000 resolution) shows the various comparisons. On the x axis best K plans are present and the y-axis contains the ratio between the cost of ROOT-K and TOP-K. Considering the selectivity space as a rectangle, A represents the lower left corner, B represents the lower right corner, C represents the upper left corner, D represents the upper right corner and E represents the mid point of the selectivity space. The Threshold is considered as 20%.



Enumeration order	ROOT	TOP-2	TOP-3	TOP-4	TOP-5	TOP-10
Hash-Merge-Nested.	5.89ms	5.95ms	6.7ms	7.6ms	9.0ms	14.8ms
Merge-Hash-Nested	5.89ms	6.14ms	6.82ms	8.1ms	9.7ms	15.4ms
Nested-Merge-Hash	5.89ms	6.14ms	6.86ms	8.2ms	10.3ms	15.9ms
Hash-Merge-Nested (without optimization)	5.89ms	15.45ms	26ms	40ms	56ms	125ms

Figure 6.2: Enumeration time taken and memory consumed by TOP-K using different plan enumeration sequences

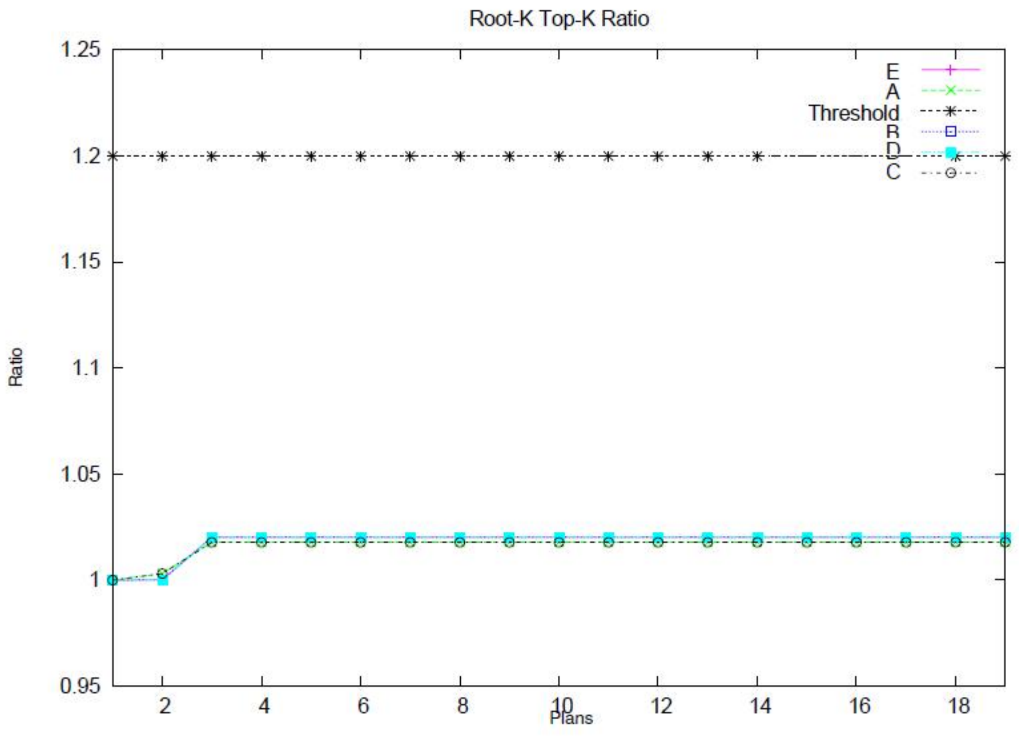


Figure 6.3: QT5 ROOT-K TOP-K ratio at 5 different points

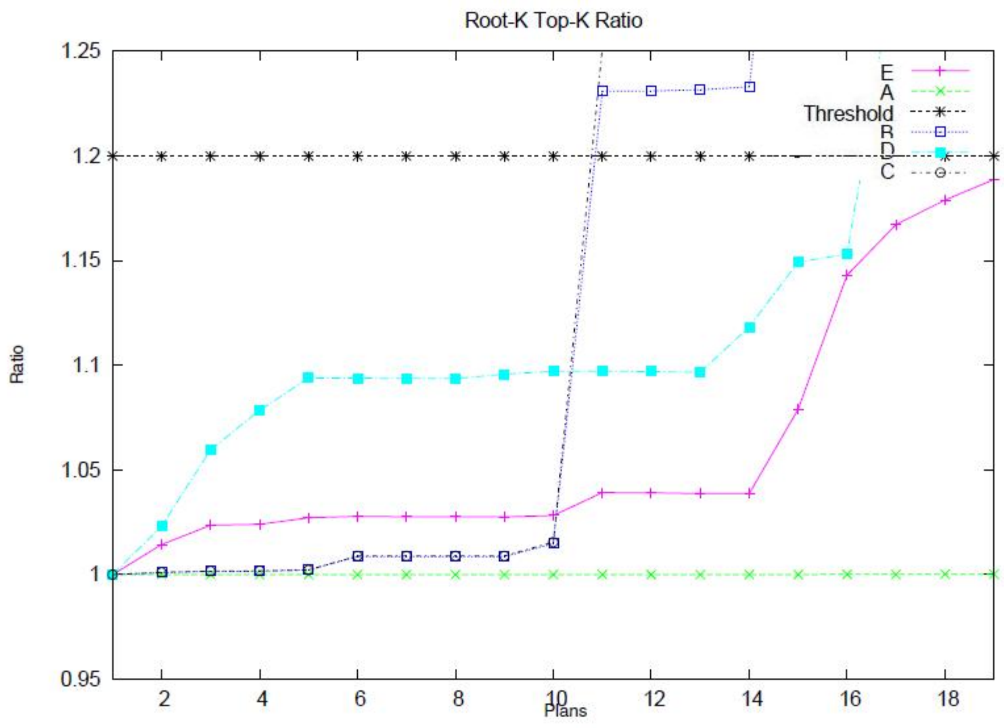


Figure 6.4: QT7 ROOT-K TOP-K ratio at 5 different points

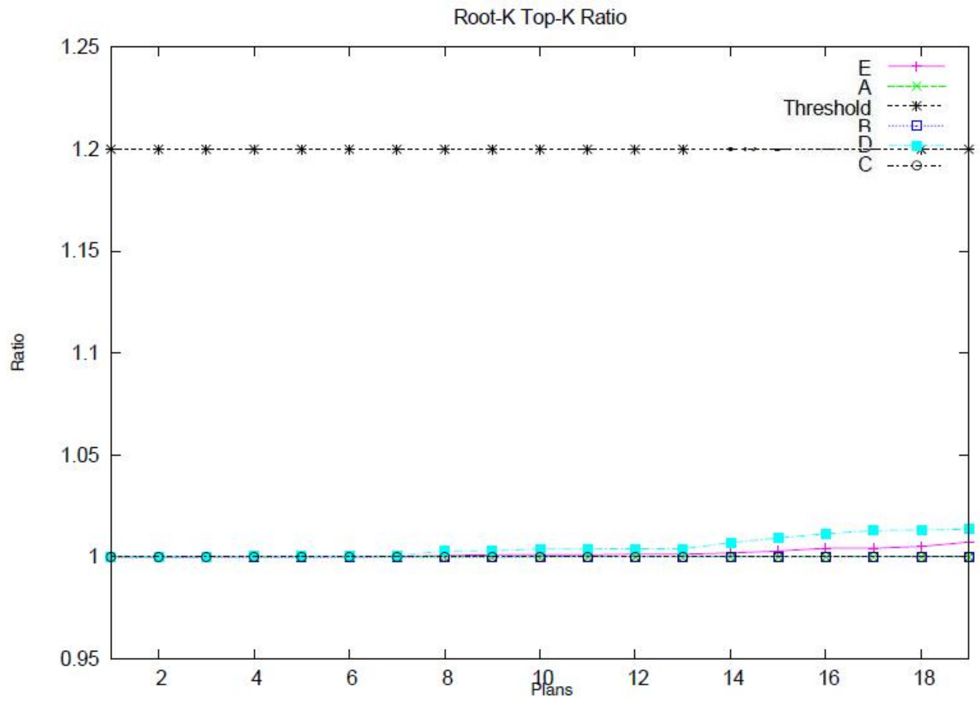


Figure 6.5: QT8 ROOT-K TOP-K ratio at 5 different points

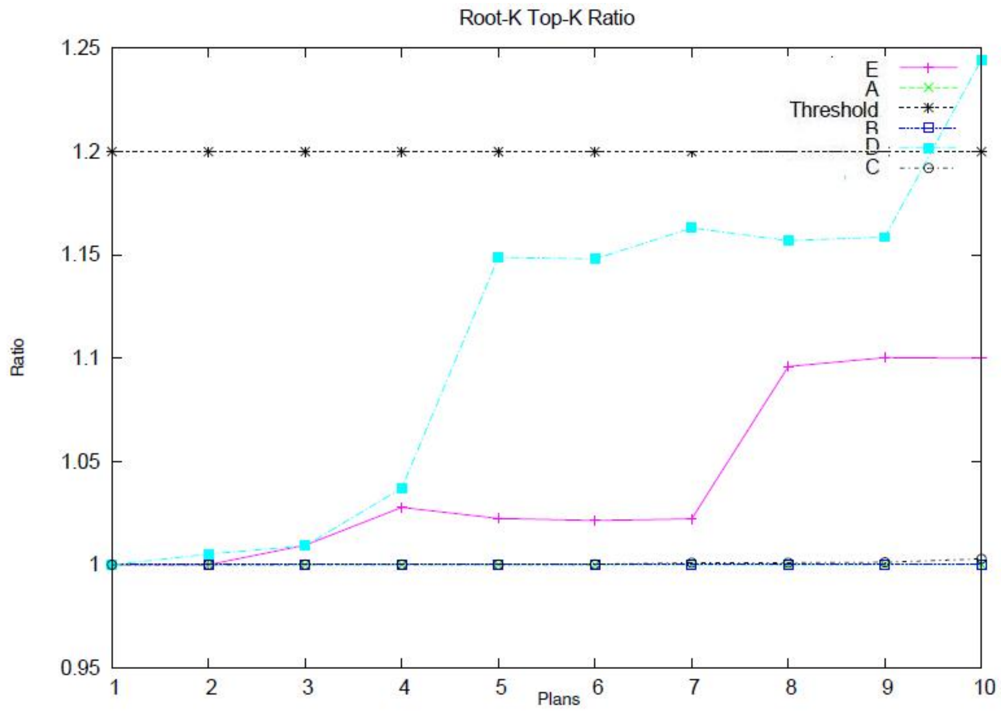


Figure 6.6: QT10 ROOT-K TOP-K ratio at 5 different points

Chapter 7

Probable applications of TOP-K

The basic reason why TOP-K was inefficient in terms of plan diagram generation was because of TOP-2's closeness to TOP-1 in terms of cost. From the plan rank list that we obtain through TOP-K, it is observed that almost all the K plans are close to each other in terms of cost. In other words all these K plans have almost near to similar cost. Although they are almost similar in their cost, still they are physically not same. So these physical differences can come into use in some cases. The various other uses of TOP-K which we researched are as follows:

7.1 Peak power reduction:

Database engines, a key component of many enterprize information systems, have been found to be major power consumers during their data processing activities, especially while executing complex queries. Peak power deals with server design, capacity planning, and prevention of overheating surges. It has been observed that the best plan(in terms of cost) returned by the query optimizer may not be the best plan in terms of Peak power reduction. But there might exist some other plan which might not be as cheap as the best plan(in terms of cost) but still have its cost within some limit of the best cost and also at the same time consumes less peak power as compared to the original best plan.

So its better to search for such plans.

From latest research it has been observed that we can model a query execution plan in terms of segmented pipelines. The peak power is observed in one of these pipelines. If we can replace this pipeline with another plan which does not have this pipeline and also which consumes less peak power then we can substantially lower our power consumptions.

From extensive observations of plans obtained from TOP-K it is observed that successive plans vary from each other at one of the join algorithms or join orders. So from the TOP-K plan rank list if we get a plan where the peak power consuming pipeline is replaced by another pipeline then we can replace this present plan. Also another added advantage is the cost. The plans in TOP-K are not much different in terms of cost. So such a replacement of plan will be considered as a healthy replacement of plan(near to optimal cost and also less power consuming).

7.2 Query re-optimizations:

Traditional query optimizers depend on the accuracy of estimated statistics for choosing an optimal plan. But there are many reasons due to which the gathered statistics may not be accurate or up to date. So such a design often leads to choosing of suboptimal plan for complex choices. Also estimates for intermediate subexpressions grow exponentially in the presence of skewed data distributions. Re-optimization comes handy to cope up with such mistakes. Current optimizers first use a traditional optimizer to pickup a plan in compile time and then during runtime react to estimation errors and resulting suboptimalities detected in the plan during query optimization. The basic idea is to collect statistics at key points during execution of the query and then use these statistics to optimize the execution of the remainder query by changing the execution plan of the remainder of the query. This process is also known as mid-query re-optimization[10]. Figure 7.1 shows a mid-query re-optimisation.

An important aspect of such a re-optimization is the collection of statistics at key

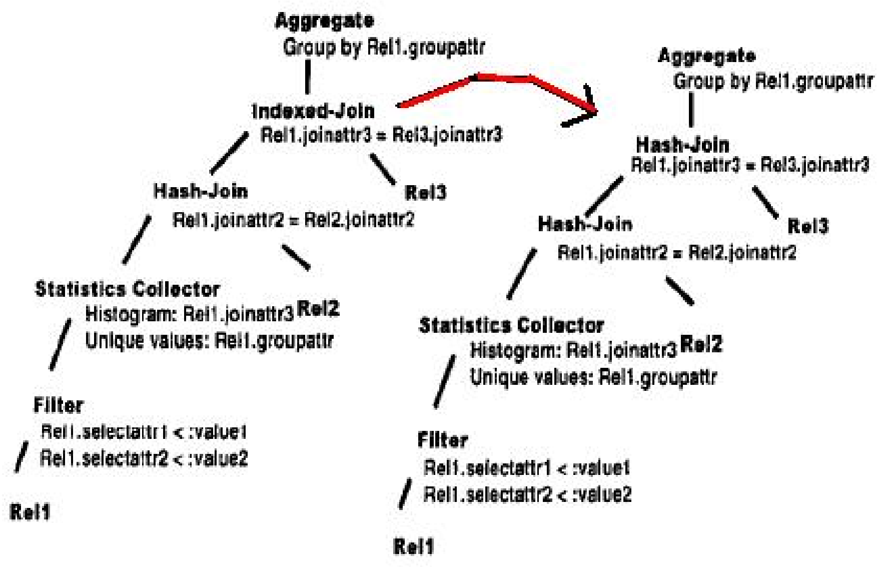

Aggregate
Group by Rel1.groupattr
Indexed-Join
Rel1.joinattr3 = Rel3.joinattr3
Hash-Join
Rel1.joinattr2 = Rel2.joinattr2
Rel3
Statistics Collector
Histogram: Rel1.joinattr3
Unique values: Rel1.groupattr
Rel2
Filter
Rel1.selectattr1 < :value1
Rel1.selectattr2 < :value2
Rel1
Aggregate
Group by Rel1.groupattr
Hash-Join
Rel1.joinattr3 = Rel3.joinattr3
Hash-Join
Rel1.joinattr2 = Rel2.joinattr2
Rel3
Statistics Collector
Histogram: Rel1.joinattr3
Unique values: Rel1.groupattr
Rel2
Filter
Rel1.selectattr1 < :value1
Rel1.selectattr2 < :value2
Rel1

Figure 7.1: Example for mid-query re-optimization

points. These requires materialization of intermediate results. If pipeline execution is present at such key points then materializing the intermediate results would result in addition of cost. So the best plan may not remain the best plan at all due to addition of materializing cost. At such points we can take help of TOP-K. We can search from the TOP-K plan rank list for a plan where there are no pipelines at the key statistics collection points. If we succed in getting such a plan then we have materiliazing points at the key statistics collection points and also not much increase in overall cost(cost of plans from TOP-K are close to each other). Thus we have cleverly sided ourselves from the intermediate results' materializing cost in the actual for the moment-optimal plan.

Another type of re-optimization, also considering robust plans[12] [13] is known as proactive re-optimization. Here in such type of mid query re-optimization we not only consider the best plan but also the best robust plan. A significant step of such re-optimisation is the generation of switchable plans. A switchable plan [13] is a set of plans S where:

(i) All plans in S have a different join operator as the root operator.

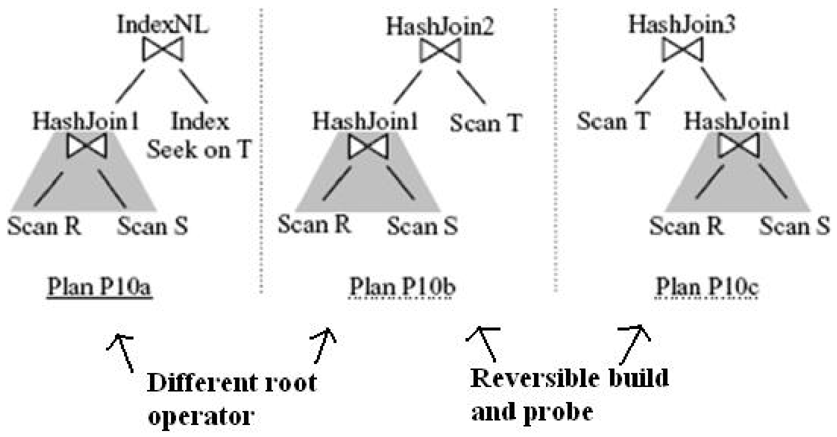


Figure 7.2: Switchable plans

(ii) All plans in S have the same subplan for deep subtree input to the root operator.

(iii) All plans in S ahve the same base table, but not necessarily the same acces path, as the other input to the root operator.

Figure 7.2 shows an example of switchable plans. The above points mentioned generally are reflected in the plans obtained from TOP-K. So TOP-K have a chance to supply us with switchable plans.

Chapter 8

Conclusion and Future Work

TOP-K needs additional enumeration of plans as compared to TOP-K. But due to various optimizations performed on the DP lattice, we can substantially bring down the time required to get the K best plans.

One of the application of Plan rank list is the PlanFill algorithm. ROOT-K and TOP-K are two different ways of getting the plan rank list. TOP-2 gives the actual plan rank list. So PlanFill gives the accurate plan diagram if we use TOP-2. But as TOP-2 is too close to the best plan hence we donot get much efficiency using TOP-2. As efficiency is the main concern lot for the PlanFill algorithm, hence we cannot use TOP-K in the Plan FIll algorithm.

ROOT-K on the other hand is an approximate of the plan rank list. It is observed that the ROOT-K list cost is much relaxed as compared to the TOP-K list. So efficiency is definitely much better than that of TOP-K.But as wrong plans might get assigned due to the cost relaxation hence it was thought that accuracy will drop. But from our research it got revealed that actually accuracy does not drop and is still greater than 99%. ROOT-2 is an efficient approximation TOP-2 in terms of Plan diagram generation through planfill. ROOT-2 is found to be always accurate in terms and also often efficient.

Future work:

It is found that ROOT-2 is often efficient and always accurate in terms of plan diagram generation. It was also observed that if ROOT2 too is close to best plan cost then even ROOT2's efficiency goes down. In such a case PlanFill algorithm fails. In future we can look into predicting the success of Plan fill algorithm even before running it.

It was also observed that TOP-K has got no use presently. TOP-K still can be obtained with low overheads in terms of memory and time. So in future we can further look into uses of TOP-K.

Certain modifications in PlanFill can also be done to fasten the plan diagram generation. We have observed that ROOT-2/TOP-2 ratio approaches 1 as we go near a plan boundary. If we get these plan boundaries in some easier way then we can paint the plan diagram in an easer way. Just as a painter paints the outline first and then paints the picture, similarly if we get the plan boundaries then we can easily paint the picture in less time. Rough plan boundaries can also be obtained through a low resolution picture also. Thus once running exact plan generation in low resolution we can get a somewhat plan boundary. Inside the boundary we can use some technique to find if cost ratio approaches to 1 thus finding other finer plan boundaries.

We would also like to research some other important applications of TOP-K and implement them inside the query optimizer.

Bibliography

[1] Harish D, "SIGHT and SEER: Efficient Production and Reduction of Query Optimizer Plan Diagrams", Master's Thesis, Indian Inst. of Science, June 2008. *http://dsl.serc.iisc.ernet.in/publications/ thesis/harish.pdf*

[2] S. Bhaumik, "Efficient Generation of Query Optimizer Diagrams", Master's Thesis, Indian Inst. of Science, June 2009. *http://dsl.serc.iisc.ernet.in/publications/ thesis/sourjya.pdf*

[3] A. Dey, "Efficiently Approximating Query Optimizer Diagrams ", Master's Thesis, Indian Inst. of Science, June 2009. *http://dsl.serc.iisc.ernet.in/publications/ thesis/atreyee.pdf*

[4] A. Dutt, "Efficient Identification of Robust Plans and Efficient Generation of Plan Diagrams ", Master's Thesis, Indian Inst. of Science, June 2010. *http://dsl.serc.iisc.ernet.in/publications/ thesis/anshuman.pdf*

[5] N. Reddy and J. Haritsa, "Analyzing Plan Diagrams of Database Query Optimizers", *Proc. of 31st Intl. Conf. on Very Large Data Bases (VLDB)*, August 2005.

[6] P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie and T. Price, "Access Path Selection in a Relational Database System", *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, June 1979.

[7] G. Graefe and K. Ward, "Dynamic Query evalution Plans", *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*,May 1989.

[8] G. Graefe and R. Cole, "Optimization of Dynamic Query Evalution plans", *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, 1994.

[9] A. Dey, S. Bhaumik, Harish D. and J. Haritsa, "Efficiently Approximating Query Optimizer Plan Diagrams", *Proc. of 34th Intl. Conf. on Very Large Data Bases (VLDB), Auckland, New Zealand*, August 2008

[10] N. Kabra, D.J. DeWitt, "Efficient Mid-Query Re-Optimization of suboptimal query execution plans", *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, June 1998.

[11] Ron Avanur, Joseph M. Hellerstein, "Eddies: Continuosly Adaptive Query Processing", *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*,June 2000.

[12] V. Markl et al., "Robust Query Processing through Progressive Optimization", *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, 2004.

[13] Shivnath Babu et al., "Proactive Re-optimization", *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, 2005.

[14] Harish D., P. Darera and J. Haritsa, "Identifying Robust Plans through Plan Diagram Reduction ", *Proc. of 34th Intl. Conf. on Very Large Data Bases (VLDB), Auckland, New Zealand*, August 2008

[15] M. Abhirama, S. Bhaumik, A. Dey, H. Shrimal, J. Haritsa, "On the Stability of Plan Costs and the Costs of Plan Stability", *Proc. of 34th Intl. Conf. on Very Large Data Bases (VLDB), Singapore*, September 2010

[16] *http://www.postgresql.org/docs/8.3/static/release-8-3-6.html*

[17] *http://www.tpc.org/tpch*

[18] *http://dsl.serc.iisc.ernet.in/projects/PICASSO/picasso.html*