Subquery Plan Reuse based Query Optimization

Meduri Venkata Vamsikrishna, Kian-Lee Tan

National University of Singapore
School of Computing
Singapore
{vamsi,tankl}@comp.nus.edu.sg

Abstract

In this paper, we revisit the problem of query optimization in relational DBMS. We propose a scheme to reduce the search space of Dynamic Programming based on reuse of query plans among similar subqueries. The method generates the cover set of similar subgraphs present in the query graph and allows their corresponding subqueries to share query plans among themselves in the search space. Numerous variants to this scheme have been developed for enhanced memory efficiency. Our implementation and experimental study in PostgreSQL show that one of the schemes is better suited to improve the performance of (Iterative) Dynamic Programming.

1 Introduction

Dynamic Programming(DP) generates an optimal plan for a query. However, for queries with large number of tables and/or clauses (predicates), it is infeasible for Dynamic Programming to optimize them as the query optimizer easily runs out of memory in such cases. Even for queries that are seemingly simple (with few relations and predicates), the search space can be large.

As an example, let us consider two versions of a 4-table star query.

```
Q1:  SELECT COUNT(*) FROM emp, sal, dept,
mngr WHERE emp.sal_id = sal.sal_id AND
emp.dept_id = dept.dept_id AND emp.mngr_id
= mngr.mngr_id;

Q2:  SELECT COUNT(*) FROM emp, sal, dept,
mngr WHERE emp.emp_id = sal.emp_id AND
emp.emp_id = dept.emp_id AND emp.emp_id =
mngr.emp_id;
```

In query Q1, the hub table *emp* uses a unique (separate) column to join with each of its neighboring tables *sal*, *dept* and *mngr*. For instance, *emp* and *sal* join on the column *sal_id* whereas *emp* and *dept* join on a separate column *dept_id*. In other words, the primary key of a table is never multi-referenced (assuming that all the predicates follow Primary key foreign key relationships). Whereas in query Q2, *emp* joins with *sal*, *dept* and *mngr* on the same column *emp_id*. We can see from Table 1 that the number of query plans (as generated by PostgreSQL 8.3.7 optimizer) at each DP lattice level is higher in the case of query Q2 compared to Q1. This happens because the optimizer applies the transitive property using *emp_id* and infers new relationships among the tables. *emp* and *sal* join on column *emp_id*, *sal* and *dept* also join on *emp_id*, so transitively an inference is made that *emp* and *dept* join on *emp_id*, thereby inferring predicates which lead to more subplans.

Table 1: 4-predicate star queries with different join columns run on PostgreSQL optimizer

Query	DP Lattice:
Q1	LEVEL 2: NUM OF QUERY PLANS = 3 LEVEL 3: NUM OF QUERY PLANS = 6 LEVEL 4: NUM OF QUERY PLANS = 3
Q2	LEVEL 2: NUM OF QUERY PLANS = 6 LEVEL 3: NUM OF QUERY PLANS = 12 LEVEL 4: NUM OF QUERY PLANS = 7

To illustrate the inference of predicates, in Figure 1, the sub query plans for level "2" in the DP lattice are enumerated for Q2 and the predicates which have been inferred from transitive property are depicted in dotted lines. It should be noted that this kind of inference happens at further lattice levels as well for Q2. In the TPC-H schema, the column "NATION_KEY" belonging to the table NATION is referenced by tables SUPPLIER and CUSTOMER. Similarly in the schema of TPC-E, the primary key S_SYMB is referenced by the tables LAST_TRADE, TRADE_REQUEST and TRADE. Query Q5 in TPC-H benchmark is being listed below. We can see in italicized predicates, how "s_nationkey" is being multi-referenced. Hence, even a query with a modest number of relations can get complex because

17^{th} *International Conference on Management of Data COMAD 2011, Bangalore, India, December 19–21, 2011*

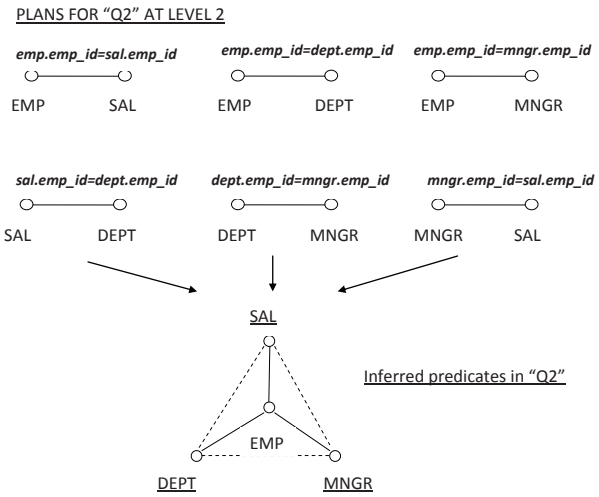


Figure 1: Inferred predicates for a star query based on multi-referenced join column.

of induced density from inferred predicates.

```
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 = '[REGION]' and o_orderdate >= date '[DATE]' and
o_orderdate < date '[DATE]' + interval '1' year group by
n_name order by revenue desc;
```

In this paper, we revisit the problem of optimizing queries. Instead of fully generating the exponential search space, we aim at generating a part of the search space and reusing it for the remaining fraction, thus bringing about computational and memory savings, and getting a high quality query plan close to optimality. Our principle idea is to reduce the size of the set of sub plans $Plans_i$ for each level "i" in the DP lattice through sub plan reuse. This needs the detection of similar sub queries which in turn requires the identification of similar sub graphs in the query graph (query graph is a way of representing the query as a graph with relations being nodes and predicates being the edges between them). Hence, the problem has been converted to a graph problem where we need to discover sub graph isomorphism internally, i.e. within a large graph. The collection of sets of similar subgraphs from all levels in the DP lattice is termed as the cover set of similar subgraphs. Once the cover set of subgraphs is generated, construction of query plans for each level in the DP lattice begins and because of exhaustive re-use of sub query plans among the similar subqueries identified by similar subgraphs present in the cover set, memory savings can be achieved. These memory savings enable our scheme to push query optimization to the next level in the DP lattice. It should be noted that the generation of the cover set of subgraphs is memory intensive and computationally expensive. Hence we optimize the cover set generation.

Figure 2 gives a pictorial representation of our scheme after the identification of similar subqueries. Similar subquery sets are fed to the DP lattice at each level. In the figure, during plan generation for level 3, the optimizer identifies from the similar subquery set that (1,2,3) is similar to (4,5,6) and hence the least cost plan of (1,2,3) is reused for (4,5,6). The plan for (4,5,6) is still constructed but in a light weight manner by imitating the join order, join methods and indexing decisions at join node and scan node respectively, thus bringing upon computation savings by avoiding the conventional method of plan generation. Memory savings are brought about since the plans for the various join orders of (4,5,6) are not being generated. So our scheme benefits from a mixture of CPU and memory savings.

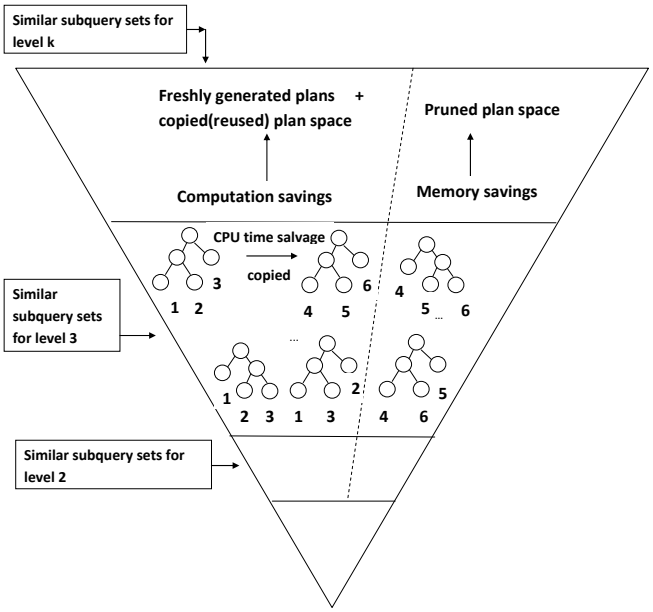


Figure 2: Search space generation in Dynamic programming lattice through sub-query plan reuse.

We propose a memory-efficient scheme for generating similar subqueries and the query plans at each level in the DP lattice. In cases where DP runs out of memory before generating query plans at a particular level, our scheme can perform better because of memory savings. However the savings are significant in the case of Iterative Dynamic Programming(IDP). In our experiments, we will show the memory savings on sparsely and densely connected queries with our scheme embedded in IDP. Intuitively it can be inferred that dense queries benefit from our scheme because the more dense the queries are, the more the predicates are and hence the sub query reuse is also high enough to push IDP to the next level in the DP lattice. But as mentioned earlier, density is prominent in many seemingly sparse queries (like Q5 in TPC-H) because of multi-referenced columns and transitive in-

ference of predicates. This gives more scope for our scheme to perform better in such cases as well. From the previous example, the sub query plans of Query Q2 at each lattice level are higher than those of Q1 as seen in Table 1, thus getting more scope for plan reuse from our optimization scheme.

2 Related Work

There are several works which aim at enumerating only a fraction of the exponential search space thereby saving time and memory expended in optimizing the queries. [8] attempts to reduce the search space by identifying connected subgraphs of a query graph and generating query plans only for the sub queries corresponding to those connected subgraphs. PostgreSQL optimizer implements this feature by default. [1] proposes a rank based pruning of join candidates to fit DP based query optimization for memory constrained hand held devices. Retaining only the best ranked join candidates based on plan cost and selectivity at each level in the DP lattice and employing left deep trees alone for plan generation (by eliminating bushy trees) lead to compromise in plan quality. [4] also employs pruning of join candidates to extend DP to higher levels for star-chain queries but the extent of pruning is lesser than in [1]. [4] identifies hub relations (relations with highest degree) in the join graph (same as query graph) that are difficult to optimize and applies a skyline function based on features (rows, cost,selectivity) to prune away certain combinations that fail to provide least cost. The problem with this approach is that certain join candidates which may give an optimal join order are getting pruned. On the other hand, [14], [6] and [9]challenge the conventional ideas of avoiding cartesian products and implementing left deep trees to reduce search space claiming that the inclusion of cross products and bushy trees is essential for optimality. [2] extends the greedy algorithm by optimizing the manner in which two sub query plans are merged based on various heuristics like least selectivity and least intermediate result size. Variants of this scheme including cases like bushy vs linear trees, inner-index based joins, group by, semi joins and anti joins were implemented. Though the idea is to enhance optimality within a greedy algorithm, the inherent suboptimality because of extensive pruning of various possible join candidates affects the plan quality. Nevertheless, the scheme scales up to complex queries of 50 relations. In contrast to this, the extent of pruning induced by Iterative Dynamic Programming (IDP) is very minimal. Our scheme keeps the search space of IDP in tact while reaping benefits out of subplan reuse. Hence, our work is not aimed at scale up in the number of relations, but addresses complexity induced by join predicates among queries.

Our work aims at modifying the Standard best row variant of Iterative Dynamic Programming (IDP)([5]). (IDP) performs DP iteratively by breaking to greedy optimization method at regular intervals as defined by a parameter "k", before starting the next iteration of DP. In general, the higher the value of "k", the better the plan quality will be, since IDP gets to run in an ideal DP fashion for a longer time before breaking to a greedy point. But for each query, depending on its complexity, there will be a maximum reachable value of "k" beyond which IDP will run out of memory. So we aim to extend "k" to achieve better IDP plan.

[17] uses the notion of similar subqueries for complex query optimization. It identifies largest similar substructures within the query graph and reuses query plans among themselves. These plans are prematurely executed and the similar subgraphs in the query graph are replaced by the result tables from each subquery execution. Query optimization is continued on the resulting query graph again using a randomized algorithm. The problem with this scheme is that the plan is pre-maturely being executed without knowing whether it is an optimal join order and it is being replaced by the result node. This sub optimal replacement poses a serious hindrance to optimality. Also the representative subquery plan generated by the randomized algorithm is not guaranteed to be optimal in the first place.

Since we search for similar subgraphs in the query graph, we have also done extensive literature survey on isomorphic graph detection. Partitioned Pattern count(PPC) trees ([15]) and divide-and-conquer based split search algorithm in feature trees ([10]) are examples of similar subtree detection done in a top-down way. [17] adopts a bottom-up way of sub query identification in a greedy manner and it aims at identifying only the largest similar subgraphs within a query graph. [7], [11] and [3] find largest common subgraph from a set of graphs. [16] and [12] find similar subgraphs in a set of graphs S given a query graph Q.

In this work, our aim is not to find the largest common subgraph or the similar subgraphs to a given query graph. Instead, within a given graph, we find similar subgraphs of all sizes. So we adopt a bottom-up exhaustive similar subgraph generation and it has to be incrementally done. Because this process is expensive, we prune certain subgraphs and trade it with the opportunity of plan reuse.

3 Sub query plan Reuse based Dynamic Programming (SRDP)

Our approach involves two steps:

- Generation of the cover set of similar subgraphs from the query graph.
- Re-use of query plans for similar subqueries represented by the similar subgraphs.

The major traits of this method that differentiate it from the existing works [17] and [4] are:

1. It doesn't generate the largest similar subgraphs alone, rather it searches for all-sized common subgraphs within the query graph to aggressively re-use plans during the generation of plans at each level in DP.

2. It avoids pruning of join candidates.

Algorithm 1 : Subquery plan Reuse based Dynamic Programming

Require: $Query$(Selectivity and relation size error bounds are pre-set)
Ensure: $Plans$
1: $QueryGraph$ = makeQueryGraph($Query$)
2: $CoverSet$ = buildCoverSet($QueryGraph$)
3: **for** lev=2 to max in the DP lattice **do**
4: $Plans[lev]$ = newBuildPlanRel($Plans$,$CoverSet$)
5: **end for**
6: **return** $Plans$

The basic algorithm has been listed as Algorithm 1. Line 2 of that algorithm corresponds to cover set generation which will be described in Section 3.1. Using the cover set of similar subgraphs for plan generation which is stated in lines 3 to 5 will be described in Section 3.2. It should be noted that when there is no more opportunity to reuse plans beyond a particular lattice level because of lack of similar subqueries, $newBuildPlanRel()$ uses conventional DP method of plan generation. The cover set generation and plan reuse as presented by the basic algorithm are not necessarily memory efficient as such. Our optimized approach is stated in Algorithm 2.

3.1 Generating cover set of similar subgraphs

We define a pair of similar subgraphs $\{S, S'\}$ as a pair of subgraphs having the same graph structure and *similar* features, i.e, each vertex, v, in S should have a corresponding vertex, v', in S' such that differences between table sizes and selectivities of the containing edges lie within the corresponding error bounds. It should be noted that this need not be the only way similarity is defined. Another possible approach may ignore equivalence in structure or selectivities. It may consider equivalence in intermediate result sizes generated by two subqueries as the sole heuristic in defining similarity. But in our paper, we use the former definition.
Our idea is to generate "sets" of similar subgraphs and not just "pairs", so that the query plan generated for one representative subquery corresponding to the subgraph can be re-used by all other subqueries indicated by the remaining subgraphs in the similar set. The cover set of subgraphs can be expressed as $\sum_{lev=2}^{n} Sets_{lev}$ where $Sets_{lev} = \sum_{i=1}^{total} Subgraphset_i$. Here "total" indicates the total number of similar subgraph sets at level "lev". $Subgraphset_i$ indicates the i^{th} similar subgraph set. The summation or total collection of all such subgraph sets at level "lev" is represented by $Sets_{lev}$. The total collection of all such subgraph sets over all levels gives the cover set of subgraphs.

As mentioned in Algorithm 1, after constructing the query graph (using makeQueryGraph()) from the join predicates participating in the query, the cover set of similar subgraphs is built using buildCoverSet(). This means, from lev=2 to lev=$levelsNeeded$, sets of similar subgraphs are identified at each level which are aggregately termed as "cover set". For the query

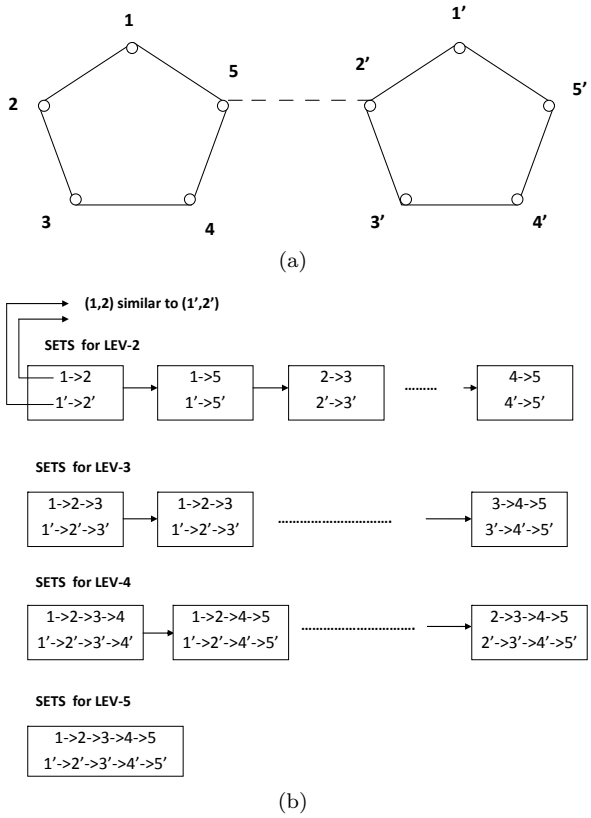


Figure 3: Sample query graph and its cover set of subgraphs

graph in Figure 3(a), [17] identifies the two pentagons as largest similar queries and executes a single plan for both of them for reduced query complexity, thereby fixing join order prematurely. [4] identifies 2' and 5 as hubs having highest degree and prunes away join candidates arriving from edges(predicates) incident on those hubs using a skyline function. But our algorithm generates an optimal plan neither using pruning nor fixing join order. From the structural information of the graph like table sizes and index information, we find that 1 is similar to 1', 2 is similar to 2' ,..., 5 is similar to 5'. These pairs are fed into a seed list. The members of a seed list are grown to find similar subgraphs of size 2 (or 2 vertices). Sim-

ilar subgraphs should have their table size differences and also selectivity differences between corresponding edges lying within acceptable error bounds. For example (1,2) and (1',2') are 2-sized similar subgraphs because their selectivities differ within the selectivity error bound. Such similar subgraphs are put into the same set. (1,5) and(1',5') are similar to each other but are unrelated to (1,2) or (1',2'). So they go into a new similar subgraph set at the same level. (Please refer Figure 3(b)).At level 2, if the plan generator wants to create a plan for (1',2') it will re-use the plan generated for (1,2), of course by replacing the base relations with (1',2'). This extends to further lattice levels.
Generation of cover set involves two stages:

1. Formation and growth of seed list to form 2-sized subgraph sets.
2. Growth of "lev" sized similar subgraph sets to obtain "lev+1" sized sets.

Stage 2 is run iteratively till we can no longer find similar subgraph sets.

3.1.1 Construction of seed List

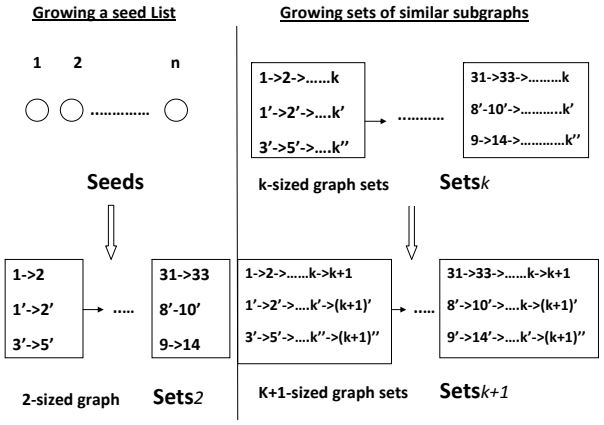
Seed list construction involves partitioning the base relations participating in the query into various groups based upon their table size differences and indexing information. Given two relations R_i and R_j, if $\frac{|relSize(R_i)-relSize(R_j)|}{max(relSize(R_i),relSize(R_j))} < relErrorBound$ where $relErrorBound$ is the acceptable fractional difference in table sizes, and if R_i and R_j are similar with respect to indexes, R_i and R_j fall into the same group in the seed list. If R_i is indexed, R_j also should have an index and vice-versa, else both of them should not have any indexes. But if both are indexed, it is not necessary that both the relations should have the same index, they are allowed to have different kind of indexes built upon them (to avoid similarity definition being too restrictive). Eventually, a group in the seed list can contain any number of relations satisfying the similarity requirement. Seed list for the query graph in Figure 3(a) is shown below in Table 2.

Table 2: SeedList

GroupId	Seeds
0	1,1'
1	2, 2'
2	3, 3'
3	4, 4'
4	5, 5'

3.1.2 Growth of seed list and subgraphs

Growing the seed list implies the formation of sets of similar subgraphs for level 2 in the DP lattice. Each set holds graph entries that are similar to each other, and each graph is represented as a linked list of relation-ids of base tables. A sample of similar subgraph sets can be seen in Figure 4(a). Just like $growSeedList()$ forms sets of subgraphs for level 2 from a set of seeds, $growSubGraph()$ grows sets of subgraphs at an arbitrary level "k" to form level (k+1) sets of subgraphs. Both of them adopt the same exhaustive style of algorithms. Growth of list and growth of sets of similar subgraphs are illustrated in Figures 4(a). To grow an



(a)

Figure 4: Growth of a seed versus growth of a subgraph.

arbitrary seed $Seed_i$, we need to fetch the neighbors of $Seed_i$ from the query graph. If $neighbours(Seed_i)$ denotes the set of neighbors, each entry in this set has to be extracted and paired with $Seed_i$ to form a 2-sized graph or a 2-vertex graph. If we can find other 2-vertex graphs similar to this graph, all of them together can form a similar subgraph set.

Let ($Seed_i$, Rel_i) be the candidate graph for which similar subgraphs have to be found. There are two ways to accomplish this:

- Check other neighbors of $Seed_i$ barring Rel_i. Combine each of them with $Seed_i$ to form a new subgraph and verify its similarity with the candidate graph.
- Grow another seed, $Seed_j$, from the same group as $Seed_i$ in the seed list. Compare the grown graph with candidate graph for similarity.

The idea behind this method of similar subgraph identification is that when we use the 1^{st} way, we are covering all possible subgraphs that contain the same seed. When we use the 2^{nd} way, we are covering all possible subgraphs that contain other seeds which are are feature wise (table size is a feature) similar to the candidate seed. Such similar seeds can be found in the seed list by referring to the candidate seed's group. This signifies the importance of seed list construction.

This method can be illustrated with an example shown in Figure 5.

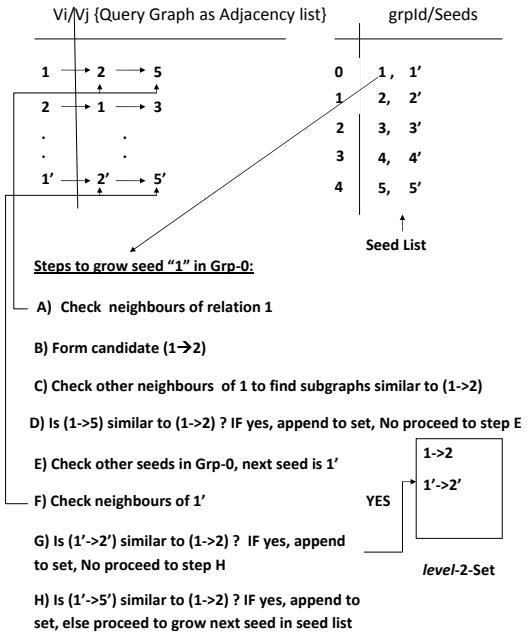
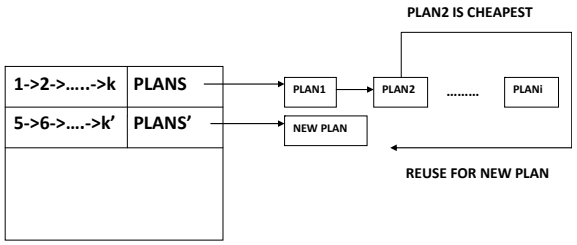


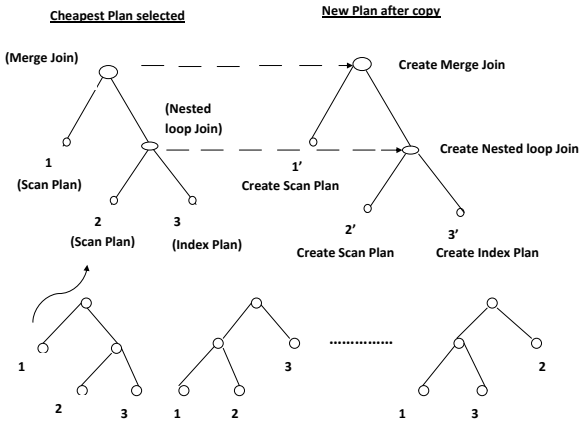
Figure 5: Example to illustrate growth of a seed in the seed list.

Two subgraphs are similar if the participating nodes are similar with respect to index presence and have their table sizes differing within $relErrorBound$ and selectivity difference between the predicates is within $selErrorBound$ i.e, $\frac{|sel(R_i)-sel(R_j)|}{max(sel(R_i),sel(R_j))} < selErrorBound$. Growth of similar subgraph sets at a particular level k to produce level (k+1) sets also uses a similar algorithm as $growSeedList()$. The only difference is that a list of level-k sets are being grown instead of seeds. Just like seeds within a seed group are similar to each other, subgraphs within the same set are similar to each other. To grow a seed S_i, its neighbour N_i is fetched from the query graph (adjacency list). Whereas, to grow a k-sized graph, say $Graph_k$, consisting of nodes S_i to S_k, neighbours of each node, namely, N_i to N_k arrive from the query graph. $Graph_k$ combined with N_1 may be similar to $Graph_k$ + N_2 thus producing a (k+1)-sized similar subgraph pair. Similarly, if $Graph_k$ is similar to $Graph'_k$, adding a vertex N_i to both of them could produce a new similar subgraph pair.

In the case of $growSeedList$, we have a seed list with each row corresponding to a seed group. But in this scenario, we have a list of sets with each set corresponding to a similar subgraph group of k-sized graphs. A seed group is also a similar subgraph group but of graph size 1. To grow a seed, we fetch all its neighbours to construct level 2 sized subgraphs. But to grow a subgraph, we need to fetch neighbours of all the vertices (base relations) participating in the subgraph. This is because subgraph growth can happen via any of the constituent vertices. Similar subgraphs for a candidate subgraph of level (k+1) are identified



(a) Plan reuse within the same similar subgraph set



(b) Cheapest $Plan$ reuse for $newPlan$

Figure 6:

as follows:

- Try growing the level k subgraph with any other neighbour of any of the constituent nodes, S_1 to S_k. Compare the new (k+1)-sized subgraph with the candidate subgraph.
- Grow any other level k subgraph belonging to the same similar subgraph set as the k-sized subgraph from which the candidate subgraph of size (k+1) has been grown.

3.2 Plan generation using similar sub queries

Plan generation is done once the cover set of similar subgraphs is available. Suppose there are "n" similar subgraphs S_1, S_2,..., S_n in a set at an arbitrary lattice level, and one of them, say S_i had its set of plans (for various join orders) generated through the traditional DP approach. When we need to generate a plan for any other member among the remaining (n-1) subgraphs in the set, we can just access the cheapest among the set of plans for S_i and reuse it for the new subgraph. Figure 6(b) explains plan reuse by example. The copied (newly constructed) plan imitates the join order, join methods and indexing decisions exactly like the original plan. While building a plan node at each level (be it root or intermediate node), the optimizer checks the type of join used for the original plan

at that level and reuses the same kind of join. For base relations, the algorithm checks the kind of scan plan or index plan built on $Plan$'s base relations and reuses the same type of plan for $newPlan$'s base relations. It should be noted that if there is no existing index on the $newPlan$'s base relation, a sequential scan is done. For example, in Figure 6(b), base relation 3' chooses to create an index plan because base relation 3 has an index plan. But the index on 3 may be clustered and the index on 3' may be non-clustered. We create an index plan for the non-clustered index present on 3', but no fresh indices are created. Our original idea was to reuse the plan per-se and not by reconstruction. This could have happened by replacing the leaf nodes of $Plan$ by those of $newPlan$ dynamically at run-time, which requires storing multiple leaf sets and not having the memory overhead of re-constructing root node and intermediate nodes. But because of the limitations imposed by PostgreSQL during implementation, we stick to reconstruction. Since plan reuse is done by reconstruction, interesting orders such as sortedness can still be retained. If an "ORDER BY" or a "GROUP BY" clause is present upon an intermediate node in the $newPlan$, but not in $Plan$, it is still possible to include the sorting decision in $newPlan$. Like [2] which pushes the clauses downwards on the tree, we can check if the current intermediate node covers just enough leaf nodes (base relations) for the clause to be applied and do so accordingly. However it must be observed that the queries used in our experimental setup consider conjunction of equi-join predicates and not "group by" or "order by" clauses.

A set of m relations typically needs O(m!) join orders during plan generation. These are logical plans which specify a relation x should be joined with a relation y before joining the result node to z. But after applying various join methods, the number of possible physical plans shoots up. The memory savings for our scheme come from not generating multiple logical and physical plans for various join orders for a join candidate in the context of plan reuse. But we should note that no particular combination of relations is being denied plan construction by our scheme.

3.3 Memory efficient algorithms

Algorithm 1 assumes that the entire cover set can fit into the main memory before passing it over to plan generation. But in complex queries, as the number of relations and predicates increases (especially when DP can no longer handle), holding the entire cover set in memory is not possible. Even the generation of the cover set takes longer time. So we adopt a more memory-efficient approach.

3.3.1 Improving Cover set generation

At a given lattice level lev, similar subgraph sets are formed by growing subgraph sets from $lev - 1$ using $growSubgraph()$. We introduce a new function $growSelectedSubGraph()$ to selectively grow $lev - 1$ sets. Essentially, the more entries a similar subgraph set has, the higher is the opportunity of query plan reuse. But in the case of complete (or very dense) query graphs of large sizes, the number of entries in a similar subgraph set will be extremely huge. But there will be a few more sets at the same lattice level with relatively fewer subgraph entries.

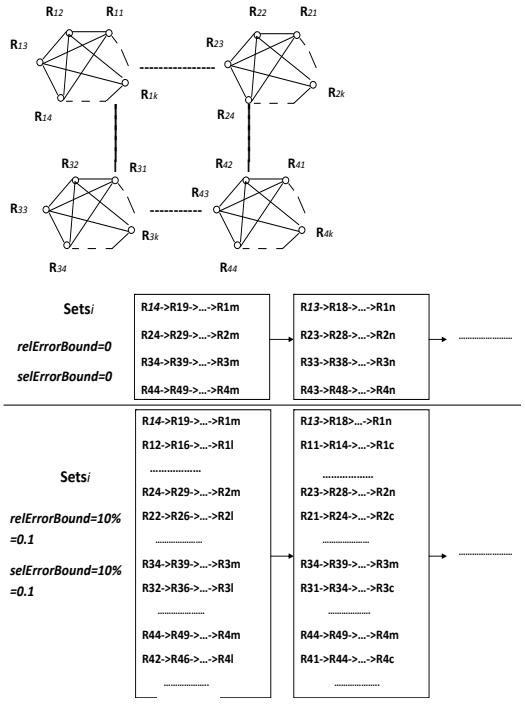


Figure 7: Increase in population of a subgraph set with error bound relaxation

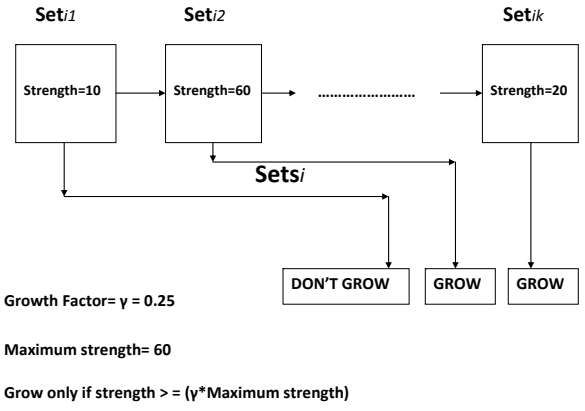


Figure 8: Growth of selected subgraph sets

In Figure 7, there are 4 largest *common* subgraphs in the query graph. Each of them is a dense subgraph with an arbitrary number of vertices assumed to be high. Ideally if the $relErrorBound$ is 0 and

selErrorBound is 0, which means that the relation sizes should be exactly similar and selectivities of predicates must be same between similar subgraphs, we can find only four entries in each set holding similar subgraphs. But if the selectivity and relation size error bounds are relaxed slightly, more similar subgraphs can be found and this eventually leads to more entries of subgraphs in each set. This explains the time taken in generating these sets on a very dense graph. There may be sets which hold *fewer* subgraphs. If we do not generate such sets, we may lose the plan reuse opportunity for the subqueries corresponding to those subgraphs. But it is worthwhile given the amount of time and space we save by not generating them. Please note that pruning subgraph sets doesn't mean plan pruning, query plans will still be constructed for those join candidates using conventional DP. It should also be noted that for sparse query graphs or averagely dense query graphs, this pruning is not even required since performance will be insensitive to subgraph pruning in those cases. This pruning (or avoiding generation) of "less dense" similar subgraph sets is done by fixing a parameter "j". Among all the sets at a given lattice level, the set with the largest number of subgraphs is examined and this count is stored in $maxStrength$. A threshold is defined as $threshold = \frac{maxStrength}{j}$. For different values of "j", $threshold$ is set to different values. So, the prune factor is "j" and the growth factor can be defined as $\gamma = \frac{1}{j}$. Any set with its strength beneath the threshold value is not grown as shown in Figure 8.

3.3.2 Improving Plan generation

We discussed how to improve memory efficiency of cover set generation by making the cover set smaller and its generation faster. But it is still required to hold the entire cover set in main memory. This becomes a bottleneck to plan generation, because while building the subquery plans, they start competing with the cover set for memory. So we try to enhance the available space for subquery plans by avoiding the construction of cover set at one go. Rather we interleave cover set generation and plan generation. Growth of subgraphs happens only when needed, and the subgraphs are deleted when they are no longer required. If we want to generate plans at an arbitrary lattice level "lev", we need to construct similar subgraph sets for level "lev" from the subgraph sets corresponding to level "$lev-1$" and immediately delete the "$lev-1$" subgraph sets. This means, at any point of time, main memory has to hold subgraph sets only from at least one level and not more than two levels. Lines 11 to 14 in Algorithm 2 illustrate dynamic subgraph construction and deletion on the fly. Line 15 portrays the construction of subquery plans at a particular level by looking up to subgraph sets corresponding to that level.

3.4 Subquery plan Reuse based Iterative Dynamic Programming (SRIDP)

In the standard best row variant of IDP parameterized by "k", dynamic programming is applied on a set of "n" tables till 1-way join plans, 2-way join plans, so on upto k-way joins are built. For (k+1)-way join, a greedy lookahead approach termed "ballooning" is used to select the least cost plan among the k-way join plans, and use that plan as a building block along with the 1-way join plans of relations not participating in the k-way join plan that was chosen just now. Iteratively, dynamic programming is applied on this new set of 1-way join plans till the value "k" is reached. To illustrate with an example, suppose k=2. {A,B,C,D} is a set of relations. 1-way and 2-way join plans of this set are generated using DP algorithm. But when 3-way join has to be computed, using ballooning, we choose the plan for {A,C} if it has the least cost among the join candidates at that lattice level. Now we have the chosen plan for {A,C} which is called as new plan {T} and 1-way plans for {B} and {D}. On these three plans {T}, plan{B}, plan{D}, the next iteration of IDP is applied. This goes on till the plan for entire set of relations is generated. Applying ballooning and formulation of one-way plans for the next iteration are illustrated in lines 18 to 20 in Algorithm 2.

3.4.1 Why embed our scheme in IDP?

Using our DP-based approach for complex queries, we can push query optimization to a few more lattice levels as compared to pure DP, but for certain queries we may not be able to reach completion since we completely avoid pruning join candidates. For example given a complex query with 20 relations, DP may run out of memory at level 10 in the DP lattice. Our scheme may run out of memory at level 15, still the savings may not count since the query could not be run to completion even with our scheme. So we need a platform to demonstrate our savings clearly. Iterative Dynamic programming (IDP) is one such algorithm which can make use of our scheme effectively. Theoretically, for a query of a typical complexity, IDP can always find a "k" which can enable it to run to completion and return a query plan. In the above mentioned example of a complex query with 20 relations, if we set "k" to any value higher than 10, conventional IDP cannot run to completion but with our scheme embedded in IDP, the maximum possible value of "k" can be stretched to 15. The advantage is that, because of extending "k", greedy selection is being postponed to a latter point in the DP lattice and a better plan is obtained. The plan quality of subquery reuse based IDP(k=15) will be higher than IDP(k=10). This will be shown in the experiments section. The bottom line of this approach is that any amount of memory savings achieved in the "push" created in DP lattice can be transformed to real benefits by integrating our scheme

with IDP. The detailed algorithm of our IDP based approach is listed in Algorithm 2.

Algorithm 2 : Memory efficient Sub query plan reuse based IDP : SRIDP

Require: $Query$, k, $pruneFactor$ (Selectivity and relation size error bounds are pre-set)
Ensure: $queryplan$

```
1: numRels = numOfRels(Query)
2: numOfIterations = ⌈numRels/k⌉
3: QueryGraph = makeQueryGraph(Query)
4: seedList = makeSeedList(QueryGraph)
5: for iteration = 1 do
6:    for lev = 1 to k do
7:       if lev=2 then
8:          Sets_2 = growSeedList(seedList)
9:          Plans[2] = newBuildPlanRel(Plans,Sets_2)
10:      else
11:         if Sets_{lev-1} can be extended to get new subgraph sets then
12:            Sets_lev = growSelectedSubGraph(Sets_{lev-1}, pruneFactor)
13:            delete(Sets_{lev-1})
14:         end if
15:         Plans[lev] = newBuildPlanRel(Plans,Sets_lev)
16:      end if
17:   end for
18:   Plans[lev] = applyBallooning(Plans[lev])
19:   RepeatRels = relationsIn(Plans[lev])
20:   Plans[1]=Plans[1] − 1-wayPlans(RepeatRels) + Plans[lev]
21: end for
22: for iteration = 2 to numOfIterations do
23:   Plans[lev] = traditionalIDP(k,Plans[1],Query)
24: end for
25: return Plans[lev]
```

4 Performance Study

The experiments were run on a PC with Intel(R) Xeon(R) 2.33GHz CPU and 3 GB RAM. All the algorithms were implemented in PostgreSQL 8.3.7. Our experimental database consists of 80 tables. The relations are randomly populated and the table sizes vary from 1000 to 8,000,000 tuples. Our experiments measure the plan quality (which is essentially related to plan cost) and optimization time over various parameter settings. The parameters are number of relations, query density, similarity measures for subqueries (percentage relaxations over similarity in relation size and selectivity) and prune factor on cover set generation(fraction of similar subgraph sets at each level that will be retained in main memory).

The default experimental settings are listed in Table 3.

Table 3: Default Parameter Settings

#Rels	Density	Similarity	prune factor
13	2	30,30	30

All our queries are synthetic and by default, contain 13 relations. We have various density levels with which queries are generated, namely, 1,2,4 with 1 being the most dense and the default being 2. Queries are randomly generated by fixing a lower and upper bound on the number of allowable predicates for a particular density level. While making sure we generate a connected graph (without any disjoint sets) we assign each node a random degree between the lower bound and the allowed maximum degree at that level. For example, at density level "d" the maximum allowed degree of a node is defined as $\#Relations/d$.

Similarity relaxation is in percentage. 30,30 denotes the relaxation in table size and selectivity difference among subgraphs to be deemed similar. That means two or more subgraphs are considered similar to each other if their table size and selectivity differences are within 30%. Prune factor is listed as 30, which means that similar subgraph sets of strength less than 1/30 th fraction of the highest populated subgraph set should be pruned off. This default fraction is indeed very low because we do not wish to lose the opportunity of subquery plan reuse. This value of prune factor can be considered equivalent to "no pruning of similar subgraph sets".

On a micro level, construction of a query plan for a join candidate using traditional DP takes 27 microsec for 2 relations to 110 microsec for 10 relations. But using SRIDP (which is how we abbreviate our scheme - Subquery plan Reuse based Iterative Dynamic Programming), the time expended in a light weight plan construction by reuse remains constant at 2 micro sec for any number of relations. Because for large number of relations, traditional plan generation needs to consider combinations from all the lower levels before constructing the final plan but plan reuse needs to copy from the cheapest plan of the similar subquery straight away without making any cost estimation for subplans. So the effort put for reuse remains the same. If scan of subgraph sets is done and if there is no match for the given subquery or if there is no other candidate in the subgraph set providing reuse, construction of plan has to be done afresh. Even in that case, the overhead incurred in scan of sets is 1 micro sec. If at a particular level in the DP lattice, there are no more similar subgraphs, even that overhead of subgraph set scan will disappear.

But there is always some extra time incurred in the generation of cover set of similar subgraphs which is controlled by the prune factor (γ). So our aim is to stay as close as possible to conventional IDP in query optimization time but to get a better plan. This happens when our subquery plan reuse based IDP can push the value of "k", where "k" determines the level in the DP lattice where a shift to greedy plan selection happens (as mentioned in the previous section).

4.1 Varying the number of relations

In this experiment, we vary the number of relations from 12 to 20 and study its effect on plan quality and running time. Figure 9(a) portrays how our scheme, SRIDP, pushes the value of "k" beyond what IDP is capable of. It can be seen that the value of "k" has consistently improved using our scheme SRIDP as compared to IDP over the varying number of relations thus retaining optimality for longer number of iterations before making a greedy choice. The consistent stretch in "k" across varying relations can be attributed to the static similarity parameters. Hence the fraction of similar subqueries that are being reused are the same across all the queries. It is important to understand that "k" is an indicator of the memory savings our scheme obtained because of plan reuse but not a measurement of plan quality. It must be noted that Skyline DP ([4]) proposed purely on the basis of pruning to reduce search space hasn't finished optimization and ran out of memory for all the queries shown in the figure. The explanation is deferred to the discussion in Section 4.5.

Figure 9(c) and Figure 10(a)show whether the increase in "k" using our scheme has translated to improved plan quality. Even though plan cost is an estimate made by the database on the expected running time, it need not be necessarily accurate. But the execution time plotted in Figure 10(a) shows that there is a definite improvement in plan quality.

Figure 9(b) shows the optimization time in seconds. SRIDP takes longer than IDP because cover set generation needs time. This sacrifice is worthwhile given the enhancement in plan quality. In one of the following experiments we show how generating only a fraction of the cover set by pruning off a few similar subgraph sets (not plans) can lead to enhanced time performance without affecting plan quality.

Figures 10(a) and 10(b) show the plan execution time and total running time (optimization time + plan execution time) respectively for a set of medium dense (density level 2) queries. These readings emphasize that the gain in execution time is worthwhile the optimization time overhead, thus making SRIDP win in overall query running time in most of the cases.

4.2 Varying density

Figure 10(c) shows the variance in plan cost with the difference in density level.

It can be observed that SRIDP consistently performs better than IDP with respect to plan quality.

4.3 Varying similarity parameters

The parameters to adjust similarity among subqueries are allowed percentage difference in table size and selectivity. For a query of default settings, similarity relaxation was varied and the effect it had on plan cost was studied. Figure 11(a) shows the changes in plan cost with respect to variance in similarity parameters.

We expected that as relaxation increases, the plan cost becomes higher and higher thereby worsening plan quality. However we cannot always ensure that plan cost will monotonically increase with similarity relaxation. This is because, we cannot be sure of the number of copied (similarly reconstructed) plans participate in the final plan. Also we cannot be sure that copied plans are always worse, they might actually be optimal enough.

4.4 Varying similar subgraph sets held in memory

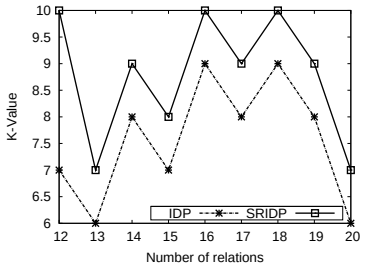
Generating the entire cover set of similar subgraph sets is time consuming. So we conducted a few experiments varying the subgraph set prune factor. This leads to reduced number of similar subgraph sets that are generated and thereby lessens memory consumption. We measured plan cost and optimization time. We observed that plan quality is least affected by the prune factor. However when considerable subgraph sets are pruned, the opportunity for subquery plan reuse decreases and hence the sub query plans need to be freshly generated. So at a very high fraction of prune factor, SRIDP runs out of memory. Else there is no considerable effect on plan quality, but optimization time reduces as the fraction of pruned sets grows higher.

Figure 11(b) plots plan cost against prune factor while Figure 11(c) depicts optimization time versus prune factor.

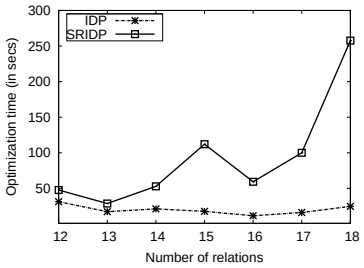
In Figure 11(b), we can observe that prune factor may change but the plan cost of SRIDP remains constant. The plan generated by IDP has also been plotted for cost comparison. Whereas in Figure 11(c), we can observe that when prune factor (denominator of fraction) is lower, the fraction of pruned subgraphs becomes higher and hence optimization time drops. When prune factor is 5 and higher, there was no effect on optimization time but at 3 and 2, the drop is seen. Anything beneath that causes SRIDP to run out of memory at that "k" level.

4.5 Discussion

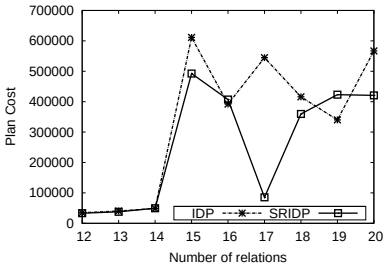
Our proposed heuristic of reusing subquery plans experimentally shows that it helps in stretching IDP to higher lattice levels. But going up by a lattice level always need not give an optimal plan. Still, our experiments done on random queries show that in most cases, our scheme performs better. However it has to be noted that the merit of the plans being reused at each lattice level will influence the overall plan quality. We thought of similarity relaxation as such a parameter that will determine the merit of plan reuse, because as relaxation becomes higher, it is natural that more plans will be reused and the optimality of overall plan



(a) K-value Vs #relations

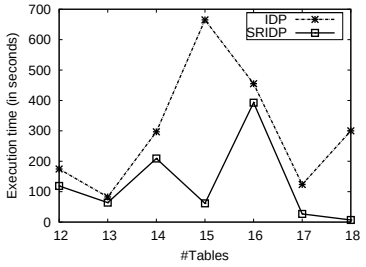


(b) Optimization time Vs #relations

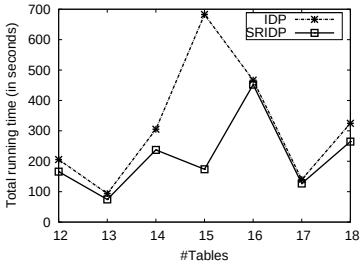


(c) Plan cost versus number of relations for medium density

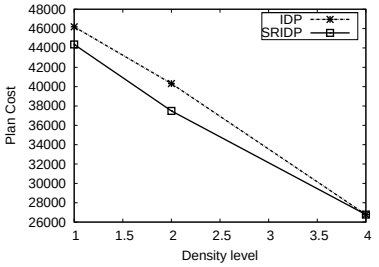
Figure 9:



(a) Query Execution time Vs #relations



(b) Total Query Running time(optimization + execution) Vs #relations



(c) Plan cost versus density level

Figure 10:

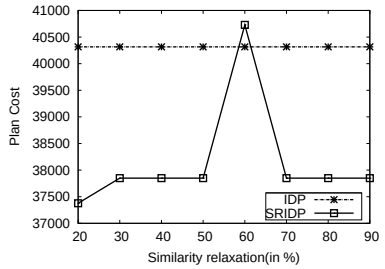
will be on a decline. But counter-intuitively, the sensitivity of plan quality to similarity relaxation did not follow a monotonic pattern. This could be because, our scheme may choose to reuse "u" plans at a lattice level. Out of them, "r" choices of reuse may be optimal and "w" reuses may be suboptimal (wrong choices) where $u = r + w$. But it highly matters how many of these plans will actually participate in the composition of the final plan. Even if $r < w$, it may turn out that the right choices may contribute to excellent plan quality. An example to this is Figure 11(a) where there was a sudden peak in plan quality at 60% and not at the other values. This is because plan reuse had a deteriorative effect on overall quality at that particular value owing to more w plans participating in the final plan. But in majority of the cases, the overall plan quality of SRIDP was better than that of IDP because more r choices contribute to the final plan making our scheme robust to relaxation parameters. The performance of our scheme was even better for high density queries, please refer to [13] for experiments on density level 1.

The insensitivity of plan quality to prune factor also takes the same explanation. An increase or decrease in the number of plans that are being reused doesn't necessarily affect plan quality, because what matters is the number of reused plans that participate in the final plan and their individual optimality.

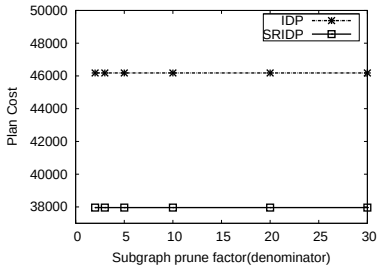
Another significant observation was skyline pruning proposed by [4], which performs well for star queries runs out of memory when run on medium to high density queries as a part of comparison with our scheme. This is because skyline pruning relies on the presence of hubs and pruning join candidates corresponding to the predicates involving the hub relation. At higher lattice levels, it aggregates relations to form a composite hub. In a star chain graph, pruning based on hub identification helps a lot but we tested skyline pruning on the same star chain queries when an attribute is multi-referenced. Then, the graph is not a star graph anymore because as explained earlier, transitive predicates are inferred and the star graph changes into a random graph where the presence of a single hub is no longer possible. In such cases, skyline pruning will not perform as well as it used to. In our experiments, skyline pruning ran out of memory at the same lattice level as IDP, when compared with SRIDP. This was because, the extent of pruning of join candidates employed by skyline pruning did not help it reach completion. It should be noted that the code for skyline pruning was obtained from the main author of [4] for the purpose of comparison and we thank him for passing us the code.

5 Conclusion

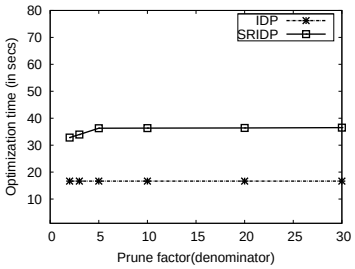
In our work, we proposed and implemented a memory efficient approach, SRIDP, to generate high quality



(a) Plan cost Vs table size and selectivity relaxation in %



(b) Plan cost versus prune factor



(c) Optimization time versus prune factor

Figure 11:

plans for complex queries using an IDP based query optimizer. The basic idea is to reuse the plans of similar sub queries among themselves and to bring about memory savings from avoiding plan generation for various join orders of a particular candidate in the DP lattice. We do not prune any join candidates in the DP lattice. Our results report a consistent increase in the value of "k" and better execution time using our scheme SRIDP as compared to IDP and better plan of higher quality in most of the cases. Our experiments studied the performance variance over a variety of parameters.

References

[1] I. T. Bowman and G. N. Paulley. Join enumeration in a memory-constrained environment. In *ICDE*, pages 645–654, 2000.

[2] N. Bruno, C. Galindo-Legaria, and M. Joshi. Polynomial heuristics for query optimization. *ICDE*, pages 589–600, 2010.

[3] B. Cuissart and J.-J. Hébrard. A direct algorithm to find a largest common connected induced subgraph of two graphs. In *GbRPR*, pages 162–171, 2005.

[4] G. C. Das and J. R. Haritsa. Robust heuristics for scalable optimization of complex sql queries. In *ICDE*, pages 1281–1283, 2007.

[5] D. Kossmann and K. Stocker. Iterative dynamic programming: A new class of query optimization algorithms. *ACM Trans. on Database Systems*, 25:2000, 1998.

[6] R. S. G. Lanzelotte, P. Valduriez, and M. Zaït. On the effectiveness of optimization search strategies for parallel execution spaces. In *VLDB*, pages 493–504, 1993.

[7] J. J. McGregor. Backtrack search algorithms and the maximal common subgraph problem. *Softw., Pract. Exper.*, 12(1):23–34, 1982.

[8] G. Moerkotte and T. Neumann. Dynamic programming strikes back. In *SIGMOD*, pages 539–552, 2008.

[9] K. Ono and G. M. Lohman. Measuring the complexity of join enumeration in query optimization. In D. McLeod, R. Sacks-Davis, and H.-J. Schek, editors, *VLDB*, pages 314–325. Morgan Kaufmann, 1990.

[10] M. Rarey and J. S. Dixon. Feature trees: A new molecular similarity measure based on tree matching. *Journal of Computer-Aided Molecular Design*, 12(5):471–490, 1998.

[11] A. Tatsuya. A polynomial time algorithm for finding a largest common subgraph of almost trees of bounded degree. *IEICE transactions on fundamentals of electronics, communications and computer sciences*, 76(9):1488–1493, 1993-09-25.

[12] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1):31–42, 1976.

[13] M. V. vamsikrishna. Exhaustive reuse of subquery plans to stretch iterative dynamic programming for complex query optimization. *M.Sc thesis*, 2011.

[14] B. Vance and D. Maier. Rapid bushy join-order optimization with cartesian products. In *SIGMOD*, pages 35–46, 1996.

[15] P. Viswanath, M. N. Murty, and S. Bhatnagar. Fusion of multiple approximate nearest neighbor classifiers for fast and efficient classification. *Information Fusion*, 5(4):239–250, 2004.

[16] X. Yan, P. S. Yu, and J. Han. Substructure similarity search in graph databases. In *SIGMOD*, pages 766–777, 2005.

[17] Q. Zhu, Y. Tao, and C. Zuzarte. Optimizing complex queries based on similarities of subqueries. *Knowl. Inf. Syst.*, 8(3):350–373, 2005.