Exhaustive Reuse of Subquery Plans to stretch Iterative Dynamic Programming for Complex Query optimization

Meduri Venkata Vamsikrishna

HT071146N

A THESIS SUBMITTED

FOR THE DEGREE OF MASTER OF SCIENCE

SCHOOL OF COMPUTING

NATIONAL UNIVERSITY OF SINGAPORE

2011

ACKNOWLEDGEMENT

I would like to express my deep and sincere gratitude to my supervisor, Prof. Tan Kian Lee. I am grateful for his invaluable support. His wide knowledge and his conscientious attitude of working set me a good example. His understanding and guidance have provided a good basis of my thesis. I would like to thank Su Zhan and Cao Yu. I really appreciate the help they gave me during the work. Their enthusiasm in research has encouraged me a lot.

Finally, I would like to thank my parents for their endless love and support.

CONTENTS

LIST OF FIGURES

SUMMARY

Query optimization using Dynamic Programming is known to produce the best quality plans thus enabling the execution of queries in optimal time. But Dynamic programming cannot be used for complex queries because of its inherent exponential nature owing to the explosive search space. Hence greedy and randomized methods of query optimization come into play. Since these algorithms cannot give optimal plans, the focus has always been on reducing the compromise in plan quality and still handle larger queries.

This thesis studies the various approaches that were adopted to address this problem. One of the earliest approaches was Iterative Dynamic Programming. Based on the study of the previous work, we proposed 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 and one of them has been found better suited to improve the performance of Iterative Dynamic Programming.

CHAPTER 1

INTRODUCTION

1.1 Motivation

Dynamic Programming(DP) generates the most optimal plan for a query. Complex queries typically have large number of tables and clauses (predicates) and make it infeasible for Dynamic Programming to optimize them as the query optimizer easily runs out of memory in such cases. In this work we intend to find similar subqueries of all sizes whose plans can be reused. The search space of Dynamic programming(DP) for a query with "n" relations can be expressed as a lattice with a combinatorial set of sub plans, $\forall_{i=0}^{n-1} \ |Plans_i| \geq {}^nC_i$ for each level i, where $Plans_i$ indicates the set of sub plans at level i. It can be seen that the count of sub plans is lower bounded by a combinatorial value because each combination of relations generates a different plan for each of the different join orders and different combinations of join methods that can be applied for each join operator in a query plan. However it should be noted that if the count of sub query plans

drops beneath the intended combinatorial value, it means that the plans involving cartesian products among relations have been eliminated.

Genetic, randomized and greedy heuristics have been proposed to enumerate only a fraction of this search space and generate query plan. But this cannot give an optimal plan of high quality. On the other hand, following the search space enumeration of DP is infeasible as the number of tables and clauses in the query go higher. Hence it is essential to strike a balance between scalability and optimality. Our scheme is aimed at generating the search space efficiently and to bring about an optimal plan.

Here is our problem statement.

"Optimization of complex queries to obtain high quality plans using Dynamic Programming based algorithms"

1.1.1 Our Approach

To optimize complex queries using DP-based algorithms, search space enumeration becomes a bottleneck. 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 $Plans_i$ for all levels in the DP lattice, i.e. for all values of i ranging from 0 to $n-1$, 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 are found. These memory savings enable our scheme to push query optimization to the next level in the DP lattice. Figure 1.1 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.

1.2 Contribution

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). Iterative dynamic programming (IDP) is a variant of Dynamic Programming which

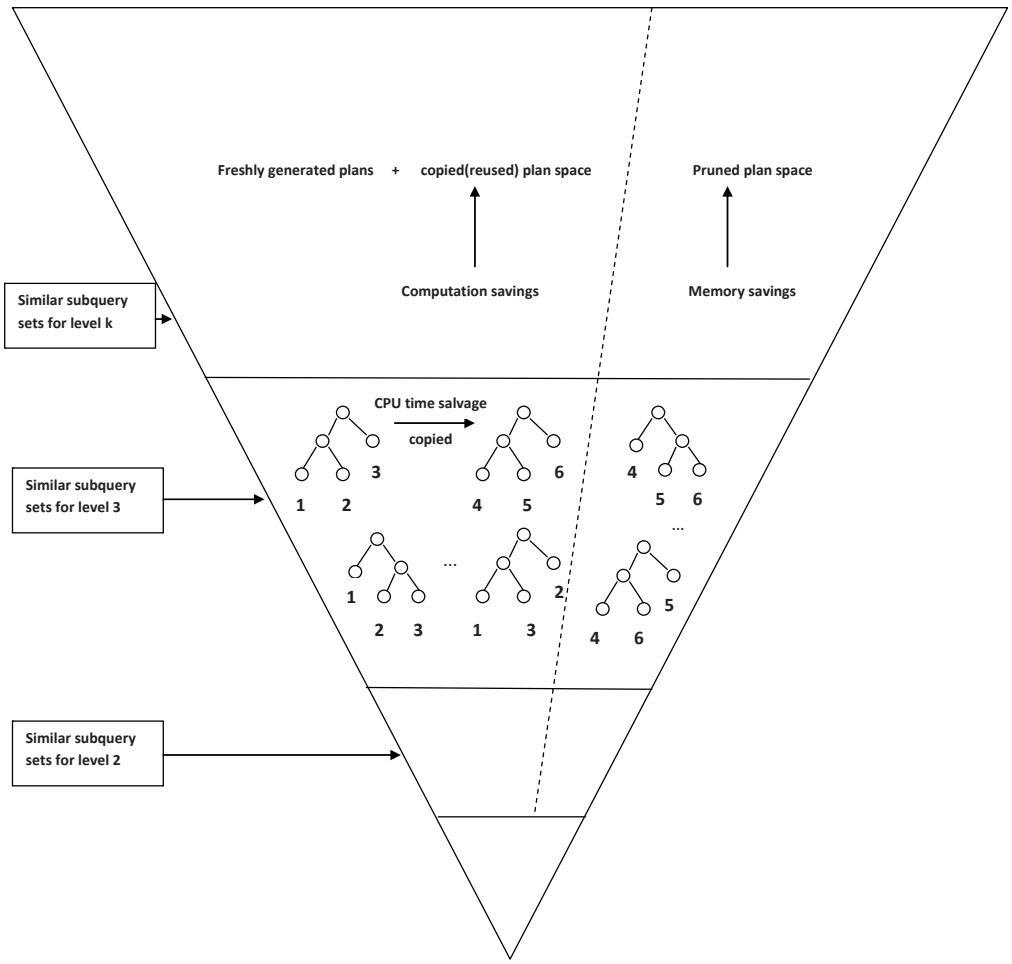

Figure 1.1: Search space generation in Dynamic programming lattice through subquery plan reuse.

breaks to greedy optimization method at regular intervals as defined by a parameter "k", before starting the next iteration of DP. The higher the value of "k", the better will be the plan quality 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 an optimal value of "k" for which IDP can run to completion and return a query plan. If the value of "k" is higher than the optimal, IDP will run out of memory. In the experiments, we are going to demonstrate memory savings on sparsely and densely connected queries with our scheme embedded in IDP. As a result, we are going to show cases where our IDP based approach can run

to completion for a higher value of "k" thus giving a good quality plan. Intuitively it can be inferred that dense queries are at advantage in using 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.

In real world, OLAP queries are dense and TPC-H benchmark is known to contain OLAP queries. Also, we have one more interesting observation to show how commonplace dense queries are in real applications and benchmarks. Let us consider two versions of a 4-table star query.

Table 1.1: 4-predicate star queries with different join columns run on PostgreSQL optimizer

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

A star query is essentially a sparse query with "n" relations and "n-1" edges where the hub relation at the center is connected to the remaining relations by predicates. In query Q1, in Table 1.2 the hub table *emp* never gets to join with any two tables on the same join column, 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 the remaining tables on the same column *emp_id*. We can see that the number of query plans (as generated by PostgreSQL 8.3.7 optimizer) at each level is higher in the case of query Q2. This happens because the optimizer applies the transitive property and infers new

relationships among the tables. For example, in Figure 1.2, the sub query plans for level "2" in the DP lattice are enumerated for Q1 and Q2 and the predicates which have been inferred from transitive property in the case of Q2 are depicted in dotted lines.

This holds true for further levels in the DP lattice too with search space differing significantly depending on the homogeneity or heterogeneity of the join columns used in the predicates. This gives more scope for our scheme to perform better, because even in sparse queries, multiple references of a column are commonplace leading to inferred edges and enhanced density of the query graph.

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.

```
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;
```

Table 4.2 has an example of a dense query for our scheme gives a better query plan.

Table 1.2: Plan Cost parameters for randomly connected query graph with multi referenced columns in predicates

Query (Number of tables)	**Number of edges**	**Scheme**	**Memory(in MB)**	**Time(in secs)**	**Plan Cost**
12	36	DP	out of memory	N/A	N/A
”	”	IDP(k=8)	out of memory	N/A	N/A
”	”	IDP(k=7)	945.26	23.93	$3.67x10^4$
”	”	Subplan reuse IDP(k=10)(0.4,0.4)	1475.69	37.74	$2.38x10^4$
”	”	Skyline DP	out of memory	N/A	N/A

While optimizing the query mentioned in Table 4.2 DP runs out of memory before generating query plans at level 8 in the DP lattice. IDP needs a ”k” value lesser than 8 to run because if DP is running out of memory at level 8, even IDP does, unless it breaks to greedy plan selection at a lattice level earlier than 8. Whereas, our algorithm, Subquery plan reuse embedded in IDP, can sustain a ”k” value of 8, because of the stretching we achieve due to plan reuse. That means we are breaking latter to a greedy point than IDP which leads to enhancement of plan quality with our scheme.

1.3 Organization of our Thesis

The rest of the thesis is organized as follows:

- Chapter 2 describes the existing approaches to reduce the search space of plan enumeration in Dynamic programming, randomized methods of query optimization and the detection of subgraph isomorphism.

- Chapter 3 presents our approach. It discusses our proposed solution: generation of the cover set of similar subgraphs and also the core aspect, reuse of sub-query plans among similar subqueries obtained from the cover set for

query plan generation. The scheme is implemented in both Dynamic programming and Iterative Dynamic Programming. We describe our naive approach and a memory conscious one for improving cover set generation and plan generation.

- Chapter 4 presents our experimental results demonstrating the CPU time speed up and memory savings on various queries.

- Chapter 5 marks the conclusion of our thesis providing a summary of our work and future directions.

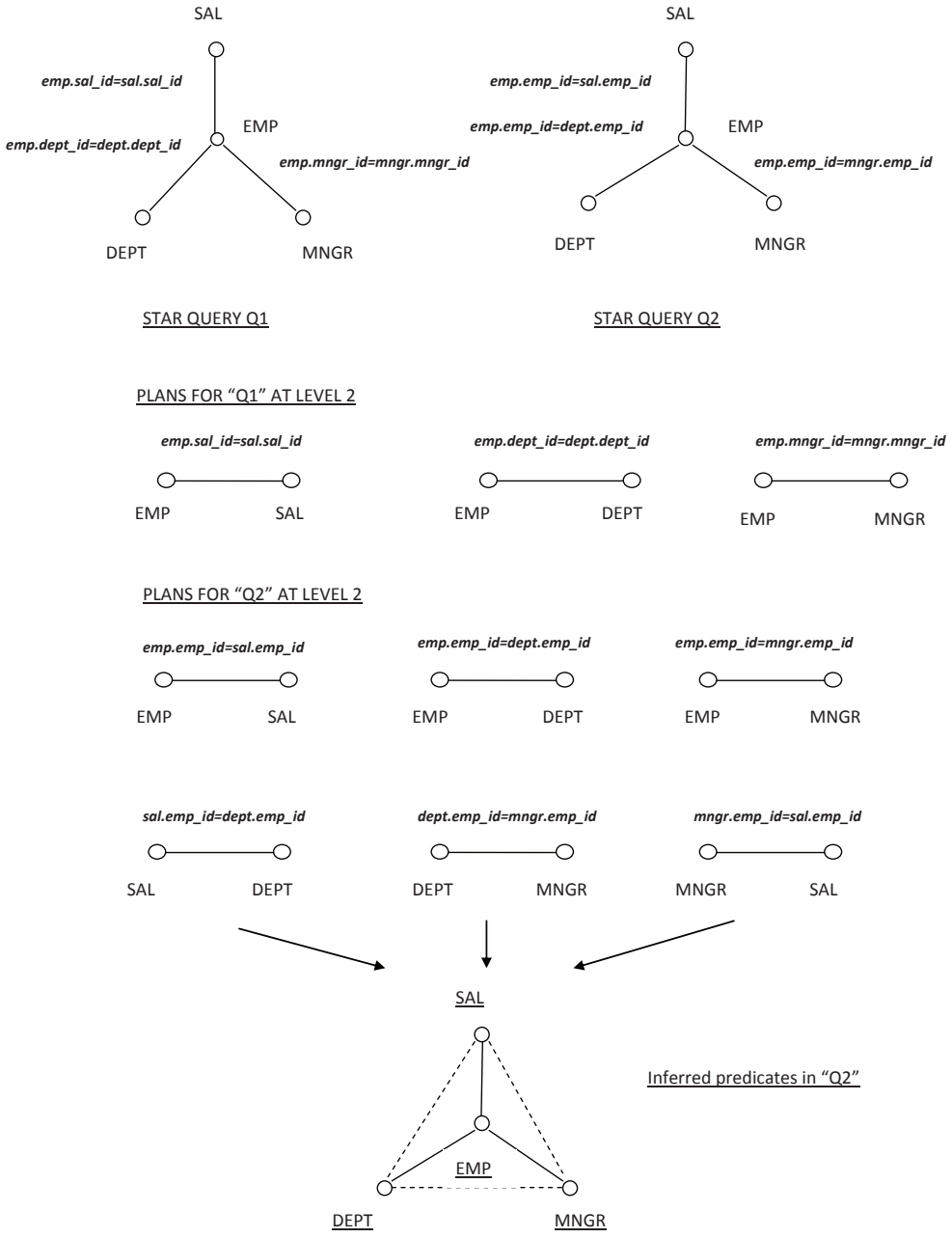

Figure 1.2: Varying densities for a star query based on join column.

CHAPTER 2

RELATED WORK

Our related work mainly comprises three main sections which are closely related to our work. They are Iterative Dynamic programming, Randomized query optimization using largest similar subqueries and Pruning based reduction of Dynamic programming search space. [17] proposes Iterative Dynamic programming. [40] handles optimization of complex queries using randomized algorithms but it cannot guarantee an optimal plan. [5] aims at improving memory efficiency of Dynamic Programming for queries with large number of tables but it greedily employs pruning of join candidates generated at each level in DP. Other essential related work includes various approaches to reduce DP search space, randomized query optimization algorithms and identification of subgraph isomorphism.

2.1 Iterative Dynamic programming

Our work aims at modifying the Standard best row variant of IDP. The standard best row variant of IDP tries to reduce the search space of DP by running DP for a while, and then adopting a greedy method of plan selection before resuming DP for the next iteration. In essence, DP is being run iteratively to retain optimality in query plan, and the greedy method's intervention is aimed at cutting down the search space and extending DP for complex queries. The algorithm for IDP is being presented in Algorithm 1.

A query with "n" relations has to go through exhaustive plan generation when DP is applied. For instance at level lev, ${}^nC_{lev}$ combinations of relations are generated assuming cross products are not eliminated. For each of these combinations, a plan is constructed for each of the various join orders. In the case of IDP, it tries to limit this exponential plan generation iteratively. A parameter "k" is fixed by assigning a value between 2 and rel where rel is the total number of relations in the DP lattice. Plan generation follows DP way of plan generation from lattice levels 2 to k. That means the number of combinations generated at levels 2,3,..,k-1 are ${}^nC_2$, ${}^nC_3$,..,${}^nC_{k-1}$ respectively. But at level "k", out of the ${}^nC_k$ combinations, only one combination is greedily picked up for which the plan cost is the least. Plans for the remaining ${}^nC_k - 1$ combinations are pruned because their cost is higher than the cheapest. Also, all the plans from level 2 to $k-1$ are discarded before starting the next iteration. The cheapest plan that has been picked up will be used 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. Then DP resumes with iteration number 2 on levels 1 to k which translates to k+1,k+2,..,2k-1 before greedily applying a cost based pruning at level "2k" followed by iteration number 3 of DP. This goes on till level rel is reached.

Algorithm 1 : Standard best row Variant of IDP

Require: $Query$
Require: k
Ensure: $queryplan$
1: $numRels = numOfRels(Query)$
2: $numOfIterations = numRels/k$
3: **for** $iteration = 0$ to $numOfIterations - 1$ **do**
4: **for** $lev = 1$ to k **do**
5: $Plans[lev]$= ApplyDP($Plans[]$)
6: **end for**
7: $Plans[lev]$ = makeGreedySelection($Plans[lev]$)
8: $participatingRels$ = relationsIn($Plans[lev]$)
9: $Plans[1] = Plans[1]$ - 1-wayPlansFor($participatingRels$) + $Plans[lev]$
10: **end for**
11: **return** $Plans[lev]$

Algorithm 1 describes the standard best row variant of IDP. k indicates the number of levels in the DP lattice for which plan generation can follow conventional DP style before breaking to greedy stage. Since the number of levels in the DP lattice is equal to the number of relations in the query, it is retrieved in line 1 and the number of iterations is obtained in line 2. *ApplyDP* indicates the regular DP plan generation and is run for an iteration on the set of plans available till then (shown in lines 4 to 6). *makeGreedySelection* chooses the cheapest plan and prunes the remaining plans belonging to array index *lev* (denoting the last index in that specific iteration) from the plan array. It also prunes all the plans for DP lattice levels 2 to $k - 1$ before starting the next iteration afresh. In line number 8, the set of relations participating in the k-way greedy plan are retrieved and in line number 9, the 1-way plans for those relations are deleted from lattice level 1. Instead of those "k" plans, the k-way greedy plan that was chosen will be appended to the set of plans in lattice level 1. Then the next iteration of IDP will be run from level 1 to k. Readers should note that this translates to the generation of plans for higher lattice levels.

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 dynamic programming algorithm. But when 3-way join has to be computed, by greedy approach, we choose the plan for {A,C} if it has the least cost among the join candidates at that lattice level. All other 2-way join plans are deleted like those for {A,B}, {B,C} etc.. One-way plans for {A} and {C} are also deleted. 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. Till we reach 2-way join plan on these plans DP is applied and greedy phase arrives before staring the next iteration. This goes on till the plan for entire set of relations is generated.

Since the parameter k is the deciding factor of plan quality, the higher the "k", the better is the plan. So our aim is to extend k to improve IDP for better plan.

2.2 Exploiting largest similar sub queries for Randomized query optimization

[40] uses the notion of similar subqueries for complex query optimization. It represents the query as a query graph and looks for largest similar substructures within the query graph. Exact common subgraphs may be difficult to find but the notion of similarity allows relaxation of various parameters defining commonality, thus providing more subgraphs that can be termed similar. The method then generates a plan for the representative query using randomized algorithms like AB or II and re-uses the plan for the remaining subqueries indicated by similar subgraphs. These plans are then executed and the similar subgraphs in the query graph are replaced by the result tables from each subquery execution. This is repeated for all other sets of similar subgraphs identified in the query graph. Query optimization is

continued on the resulting query graph again using a randomized algorithm. There are two issues with this method:

- The representative subquery plan generated by the randomized algorithm is not guaranteed to be optimal.
- 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 being a serious hindrance to optimality.

This can be illustrated by an example. If a query has 20 nodes(relations) and if the subgraph formed by nodes "1 to 5" is similar to that formed by "15 to 20", and also if these are the largest similar subgraphs found in the query graph, a plan will be generated for the representative subgraph "1 to 5" and the same plan will be re-used for "15 to 20". The respective plans are immediately executed and replaced by their result relations in the query graph. Now optimization resumes on the modified query graph with 12 nodes. This implies that the method is baselessly assuming that nodes "1 to 5" should be joined first and so about "15 to 20". But that may not be the optimal join order the query ideally requires. DP might have wanted "3 to 7" to be joined first.

But to use DP for complex queries, a more memory efficient algorithm is required.

2.3 Pruning based reduction of Dynamic Programming search space

[5] employs pruning to extend DP to higher levels. Their method is tailored to star-chain queries. They identify hub relations (relations with highest degree) in

the join graph (same as query graph) that are difficult to optimize and apply 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 are getting pruned. Let us take the same example query of 20 relations stated above. Suppose in the query graph, it is found that relation "5" and "15" have a degree of 4, which is the highest degree among all the 20 relations, pruning is applied on the edges of 5 and 15. Suppose 5 is connected to (6,7,8,9) and 15 is connected to (16,17,18,19), after applying the skyline function, only the least cost edges will remain out of 4 in each case. If (5,6) is the least cost edge, (5,7), (5,8) and(5,9) are pruned. Similarly let us say (15,16) alone is retained. In the following iterations of Dynamic Programming, if a plan for (5,7,8) has to be generated, the join order (5,7) followed by 8 will never arise because it has already been deleted. But possibly it would have been the most optimal join order for this combination.

Our work mainly focusses on retaining the quality of the optimal plan as generated by DP, yet to be able to extend it to complex queries with large number of tables avoiding pruning completely and also without fixing join order.

2.4 Top Down query optimization

Top down query optimization proposes memoization as against dynamic programming (bottom up). Just like Dynamic Programming stores all possible sub solutions before finding a new solution, in a top down approach, optimal sub-expressions of a query which can yield better plans are stored and this is the definition of memoization. In a DP lattice, the top most expression is a collection of all relations. A top down enumeration algorithm will keep searching for optimal sub expressions of the

higher level's expression at subsequent lower levels in the DP lattice. In [31], the algorithm estimates the lower and upper bounds of top-down query optimization. The paper states that in Cascades optimizer (adopting top down approach), the scheme looks for logically equivalent subexpressions within a group at a particular level in the DP lattice and avoids generation of plans for all those expressions whose estimated cost is higher. We should note that our scheme is different from this one because in this scheme, logically equivalent subexpressions refer to different join orders of the same set of relations. They do not search for similar subexpressions across different sets of relations as we do.

In [6], the authors propose a top down join enumeration algorithm that is different from top down transformational join enumeration. Their algorithm searches for minimal cutsets that can split a query graph into two connected components at each level in the DP lattice. They prove that top down search incurs no extra cost by adopting it instead of the traditional bottom up enumeration of DP lattice. The paper studies flexible memo table construction where plan reuse is plausible across different queries (inter query plan sharing only for exactly same tables) as against our approach which shares plans within the same query (intra query plan sharing for combination of relations termed similar, not necessarily same sets of tables).

2.5 Approaches to enumerate only a fraction of the exponential search space

2.5.1 Avoiding Cartesian Products

In [22] the authors attempt to reduce the search space by formulating connected subgraphs of a query graph in such a way that query plans need to be constructed

only for the sub queries corresponding to those connected subgraphs. DPSub and DPSubAlt algorithms ([21]) are variants of DP which look for connectivity in an enumerated subgraph (in the query graph) and its complement subgraph. If either one of them is not connected, it would contribute to a cartesian product and is hence pruned. The authors use Breadth first search to test the connectedness of the subgraph and its complement. Also, expressing the subgraph as a bitmap helps fast generation of subgraphs. #csg denotes the number of connected subgraphs and #cmp denotes the number of complementary graphs that are non overlapping with the given subgraph. So #ccp can be defined as the number of csg-cmp-pairs which is equivalent to the number of non overlapping relation pairs which will contribute to the sub query plan search space.

As a supplementary to our algorithm, we too proposed an algorithm in which we can test for the connectedness of every combination of relations using Depth first search and avoid plan generation if they cannot give a connected subgraph, but we later realized that PostgreSQL optimizer, by default, does that checking and chooses only connected components for plan generation.

2.5.2 Rank based pruning

In [3], the authors propose a deterministic join enumeration algorithm to perform DP-based query optimization implemented in Sybase SQL for memory constrained environment as found in hand held devices. But their algorithm is not anywhere close to optimal plan given by DP for the simple reason that it is a very greedy way of growing plans by estimation of plan costs and selectivity at each level, that too by retaining only one best. For example if the join order for "k" relations has been obtained till now, while enumerating "k+1"th relation, the candidate relations are ranked based on cardinality, out degree and whether an equi-join edge

(corresponding to predicate) exists between the new relation and one of the "k" tables, eventually the table with the best rank is added to generate the (k+1)-table plan. This is done to reduce the search space drastically using "branch and bound" technique (branching among many relations and bounding to the one with least cost) and also left deep join trees are employed.

2.5.3 Including cartesian products for Optimality

In [36], Vance and Maier propose an interesting phenomenon of including the combinations of relations contributing to cross products into the DP lattice. They challenge the conventional ideas of eliminating cross products and developing left deep trees which were perceived to be efficient in reducing the search space of DP. Their claim is that a cross product could also be optimal. They also avoid singleton relations used for left deep trees and consider bushy plans instead. A subset of relations involving a cross product is split into two sets of non-singleton relations. The pair of relation sets which incurs least cost is termed as the best split to compute cartesian product.

2.5.4 Multi-query optimization: reuse during processing

Since exhaustive plan space is expensive to search, identifying common subexpressions of a relational algebraic expression formed out of a query (intra query optimization) and also across multiple queries (inter query optimization).But both these methods are aimed at reusing the results of these common sub expressions than to reuse the plans themselves. The result reuse is during query processing as against the plan reuse in our work during query optimization.

2.5.5 Parameterized search space enumeration

In [18], the authors conclude that bushy plans combined with randomized query optimization is the best solution when DP search space becomes intractable. [24] addresses query optimization in Starburst optimizer and allows enumeration of cartesian products and composite inner joins which are nothing but bushy joins where the inner relation doesn't have to be a base relation unlike left deep trees. This is done to obtain high quality plans but at the same time, parameterized search space enumeration is introduced to keep a bound on the number of cartesian products and bushy trees. Starburst's optimizer can also detect inferred predicates like the one in PostgreSQL.

[8] proposes a random, uniformly distributed selection of query plans from the exponential search space and then applies a cost model to evaluate the best plans among those selected plans. This is proposed as an alternative to transformation based randomized schemes like iterative improvement and simulated annealing.

2.6 Genetic approach to query optimization

In [12] the authors focus on both left deep and bushy trees representing query plans by treating them as chromosomes and generating join output by joining the best plans. Search space reduction is done by choosing the local best.

2.7 Randomized algorithms for query optimization

There are several randomized algorithms that were proposed as an alternative approach for search space enumeration. Instead of considering the exponential search

space to get the best plan, supposedly low cost plans are obtained by considering a set of seed plans and moving to better plans by moves applied on the seeds and on the newer plans obtained. A move is defined as a single transformation which may involve flipping the left and right children of an operator in the plan tree, or changing the join method of the operator. For example in iterative improvement(II) algorithm, the plan space is referred to as strategic space contains states(strategies) which are nothing but plans. II always moves from a state to another state only if the newer state has a lesser plan cost. So the aim is to move towards a local minimum on which the transformation is applied again. This repeatedly happens till the least cost local minimum is found. In simulated annealing, instead of just moving to local minimum, a plan can also be transformed to get a higher cost plan but with a certain probability. This probability is reduced as time progresses (to put a check on the number of random uphill moves and to encourage more downhill moves) till it reaches zero when we reach with a least cost state among all the states that have been visited. But uphill moves are allowed in first place to avoid the algorithm from getting stuck in a local minimum. 2PO(2 phase optimization) is a combination of both the algorithms, where II is run in the first phase and from the output state, we run the second phase of SA by feeding it as the input state for the new phase with a low probability for uphill moves which will soon reach zero eventually. In [13], the authors study the cost functions evaluated for the strategy spaces of the three algorithms (II, SA, 2PO) and conclude that query optimization on bushy trees is easier than left deep trees which is contradictory to popular belief, because most of the previous work states that left deep trees are easier to describe the search space with, than bushy trees.

Tabu search [23] is also a randomized algorithm which prevents repetition of states in the move set, i.e, as moves are applied on states in the strategy space,

there is a danger of an older state getting repeated. This algorithm keeps track of the most recent states visited in a tabu list and makes sure that none of them occur as a result of a new move.

In KBZ algorithm, the minimum spanning tree is found from the query graph and with each node as the root, the join graph is linearized, followed by detection of the appropriate join sequence (among those linearized trees) with least cost as the optimal query plan. AB algorithm modifies KBZ algorithm to include cyclic queries, various join methods are applied on each of the joins, and also swapping relations to find interesting orders is also adopted. These are done to remove the constraints in KBZ algorithm and also to finish the search space enumeration and generate plan in polynomial time.

2.8 Ordering of relations and operators in the plan tree

[32] focusses on combining heuristics with randomized algorithms for query optimization. The heuristics involve pushing selections down the join tree, applying projections as early as possible and enumerating combinations involving cross products from the search space as late as possible. The augmentation heuristic and local optimization focus on ordering the relations to be joined in the order of increasing intermediate result sizes and ordering relations into clusters respectively.

2.9 Inclusion of new joins to query optimizer

In [9] the authors focus on how to include one-sided outer join and full outer join into the conventional optimizer using associative properties, reordering mechanisms,

simplifications of outer join into a regular join and finally enumerate the join orderings properly to be able to construct a plan tree.

Similarly [7] also talks about fixing the execution order when a "group by" operator is present in the join tree and when to evaluate it. It also discusses the transformation of sub queries by representing the given query in various ways in relational algebra and introducing additional sub expressions into the algebraic expression, if necessary, in order to remove sub queries.

In [25], the authors aim at constructing extended eligibility lists to handle outer join and anti join through reordering using associative and commutative operations.

2.10 Detection of Subgraph isomorphism

Our work is aimed at detecting similar subgraphs of all sizes within a given query graph. We reviewed the various approaches of finding graph isomorphism before deciding on our approach.

2.10.1 Top down approach to detect subgraph isomorphism

Identification of similar subgraphs are usually done bottom-up. In the case of top-down optimization, the given graph is split into two subgraphs using cutset identification. If we want to find similar subgraphs between two given query graphs (G_1,G_2), a cutset can split G into two connected subgraphs G_{11} and G_{12}. Another cutset can split G_2 into two connected subgraphs $G_{21}$and G_{22}. If similar subgraphs are found among the newly obtained subgraphs $\sum_{i=1}^{2}\sum_{j=1}^{2} G_{ij}$, the cutsets have been chosen correctly to detect inter-query similar subgraphs. The same applies to intra-query similar subgraph identification. The cutset has to be constructed in such a way that the connected subgraphs obtained are similar to each other, and

this procedure can continue recursively to find similar subgraphs of smaller sizes. Partitioned Pattern count(PPC) trees ([37]) and divide-and-conquer based split search algorithm in feature trees ([26]) are examples of similar subtree detection done in a top-down way. Mining closed frequent common sub graphs and inferring all other subgraphs from them instead of enumerating all common subgraphs can be a useful alternative.

This is not applicable to bottom-up DP lattice construction, where if subgraph reuse is targeted, similar subgraphs should also be identified in a bottom-up manner.

2.10.2 Bottom up approach to detect subgraph isomorphism

[40] adopts a bottom-up way of sub query identification but it aims at identifying only the largest similar subgraphs within a query graph, where it is not necessary to exhaustively look for all-sized similar subgraphs. So their algorithm is greedy in some sense, similar subgraphs are expanded till no more nodes can be added, but all the small-sized similar subgraphs are discarded. So the motive at every substep during expansion is to make sure the node being added has many unselected nodes (meaning nodes that are not participating in any similar subgraphs) adjacent to it satisfying the similarity requirement. This is because, that gives a scope for the expanded subgraph to be as large as possible, unlike our approach which searches for exhaustive collection of similar subgraphs irrespective of their sizes.

2.10.3 Finding maximum common subgraph

There are several works on finding maximum common subgraphs between two given graphs. Commonality is defined by the specific application (eg., Biology, Chem-

istry) depending on the attributes (eg., molecular features) and the accepted value of error bound between the attribute values. McGregor's similarity approach ([19]) adopts a back tracking algorithm while adding feasible pairs to enumerate all the common subgraphs before choosing the maximum sized pair. Durand-Pasari algorithm forms an association graph (in which similar vertex pairs from the original graph form vertices themselves and similar edge pairs from the original graph form edges themselves) from the given graph and reduces the maximum common subgraph detection problem in the original graph pair to a maximum clique detection problem in the association graph. Since each vertex in an association graph represents a pair of compatible vertices and each edge denotes a pair of compatible edges, the maximum clique in this graph will denote densely connected pairs, i.e, most compatible vertex pairs which will reflect to maximum common subgraph in the original graph. [1] proposes sorting of the subgraph pairs obtained on the basis of similarity scores obtained on the basis of degree of nodes and their neighbors, node and edge attribute similarity. Only common subgraphs of large sizes with scores above a threshold are retained and remaining are discarded.

[27] gives a thorough survey of the various approaches towards the detection of subgraph isomorphism. [34] is also aimed at finding the largest common subgraph from a set of graphs. It is a dynamic programming based technique where subproblem solutions are cached rather than recomputing. But since the original problem is NP-complete the algorithm provides a solution of polynomial time complexity to find connected common subgraph only when the participating graphs can be classified as 'almost trees with bounded degree'. This means the graphs are biconnected components having a number of edges within a constant difference from the number of vertices. Similarly there are genetic-based, greedy and randomized approaches to reduce the search space while detecting the maximum

common subgraphs. Screening methods introduce a lower bound to define similarity among graph substructures, thus allowing approximate solutions instead of exact similarity requirement which is rigorous. [40] also adopts screening by relaxing "commonality" to "similarity" which reflects in our approach too.

[4] aims at finding the largest common induced subgraph of two graphs. An induced subgraph, I of a graph G means that I is a subgraph of G such that all the edges in G which have both their end vertices in I are present in I. The algorithm constructs a labelled tree of connected subgraphs common to both the input graphs and returns the largest common subgraph.

Given a query graph Q and a set of graphs S, [39] finds all graphs in S which contain subgraphs similar to Q within a specific relaxation ratio. The algorithm reduces the number of direct structural comparisons of the graphs by introducing a feature based comparison which will prune a lot of non-matching graphs from S. A feature-graph matrix is constructed with features as rows and graphs in S as columns. The number of times a feature appears in a graph (number of embeddings of a given feature in a given graph)is the entry. The number of embeddings of each of the features in the query graph is also computed. If the difference of feature values between the query graph Q and a member in S is within the upper bound of relaxation, the graph member is further eligible for substructure similarity. Else, the graph member is pruned and not considered for structural comparison. Thus, the search space of graphs to be compared from S for a given query graph is reduced.

[35] attempts to check if for a given graph G, isomorphic subgraphs can be found in G'. The enumeration algorithm expresses the graphs as adjacency matrices and tries to find 1:1 correspondence between the matrix of G and a sub-matrix of G' and in this process, some of the 1's from the matrix of G' are removed to reduce

the search space comparisons.

2.10.4 Slightly lateral areas using subgraph detection

[2] aims at finding locally maximum dense graph clusters. Vital nodes which participate in more than one of these locally dense clusters are the ones which create overlap among those clusters. The idea of this algorithm is to find such communities (clusters) which are highly dense and overlapping. This can give us interesting information about a social network, for example, the communities in which an individual may actively participate, or related communities that share a lot of followers etc..

Image mining using inexact maximal common subgraph of multiple ARGs describes images as attributed relational graphs(ARGs) and discovers most common patterns in those images by finding the maximum common subgraph of the ARGs. The algorithm uses backtrack depth first search algorithm to detect maximum common subgraph.

From a slightly lateral topic, we studied a work in logic,([10]) where the problem is to find more than one Boolean formula which can define a subset of rows in the 0-1 data set, which is termed as re-description mining used in real world to detect genomes among people that are similar. To find such formulae which are syntactically different but logically same, the solution is supposed to enumerate the search space of Boolean formulae. So the algorithm ends up pruning the exponential search space of Boolean queries using greedy algorithms to prune away formulae based on Jaccard similarity and p-value which are measures of similarity and interestingness(significance) respectively. It also mines closed itemsets and sorts them by p-value and retains the best ones.

CHAPTER 3

SUB QUERY PLAN REUSE BASED ALGORITHMS : SRDP AND SRIDP

In this chapter, we describe our proposed approach in detail. In section 3.1, we present the basic plan generation algorithm and illustrate it by an example. Our scheme involves two essential steps: Cover set generation and plan construction by reuse. Sections 3.2 and 3.3 describe the construction of a query graph and the cover set of similar subgraphs respectively. The similar subqueries identified from the cover set are exploited for plan reuse in section 3.4. For enhanced memory efficiency, we try to make both the steps memory sensitive. Section 3.5 holds the algorithms for improving cover set generation and plan construction. Finally we present our scheme embedded in IDP algorithm in section 3.6.

3.1 Sub query Plan reuse based Dynamic programming (SRDP)

Our method is a modification of traditional Dynamic Programming for Query Optimization to make it more memory efficient. Initially, we are going to illustrate our approach with respect to Dynamic Programming with an example. In the later

sections, we will explain how our method works in the case of Iterative Dynamic Programming. 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 [40] and [5] 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 completely.

Algorithm 2 : Plan generation with subgraph reuse

Require: *Query*(Selectivity and row error bounds are pre-set)
Ensure: *plan* in the case of "explain query", *result* if query is executed
1: $QueryGraph$ = makeQueryGraph($Query$)
2: $CoverSet$ = buildCoverSet($QueryGraph$)
3: **for** lev=2 to $levelsNeeded$ in the DP lattice **do**
4: $Plans[lev]$ = newBuildPlanRel($Plans$,$CoverSet$)
5: **end for**
6: **return** $Plans$

As mentioned in Algorithm 2, 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". They are passed to the plan generation phase. So the plan generator looks for possibilities of plan reuse using the cover set. Suppose the plan generator has to construct plans for candidates at level 5 in the DP

lattice, it gets plans for all possible candidates from levels 1 to 4 and also a cover set of similar subgraphs. Before constructing a plan for a particular candidate (subquery), it checks if that candidate's query graph is present in the similar subgraph sets corresponding to level 5. If the candidate subquery is present in a set S_i, the plan generator verifies if any of the other candidates present in S_i had their plans generated. If yes, the plan is re-used and fresh plan generation is avoided. By reuse, it is meant that conventional method of plan generation is not followed. But a simpler plan is still constructed exactly similar to the existing plan giving memory and time savings. This is because the base relations differ from one candidate's plan to another, thus demanding the construction of a new plan. Memory savings are obtained because, usually for a particular join candidate, multiple plans are generated and stored before identifying the cheapest. But in the case of plan reuse, only the cheapest plan is reused. That implies, for the new candidate, only minimum number of plans are constructed thus saving memory.

If at a particular level, similar subgraphs are no longer there, plan generation is done the usual DP way.

Generation of cover set and plan reuse are elaborated in the following sections.

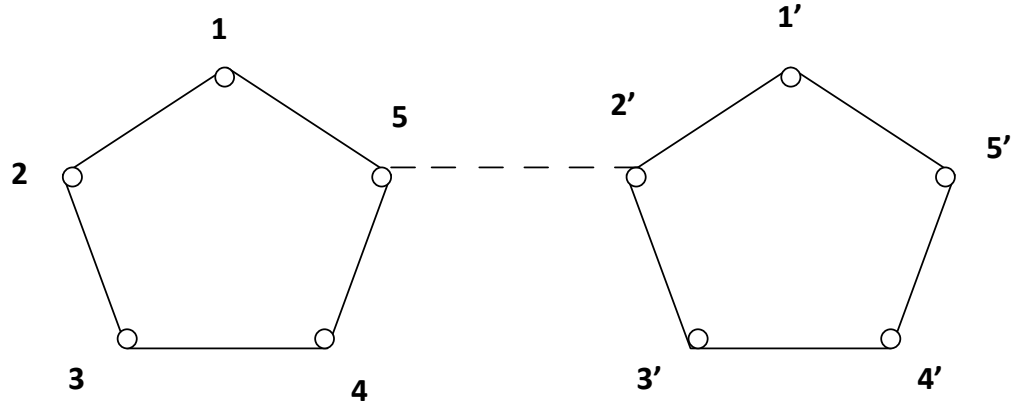

Figure 3.1: A Sample query Graph with similar subgraphs.

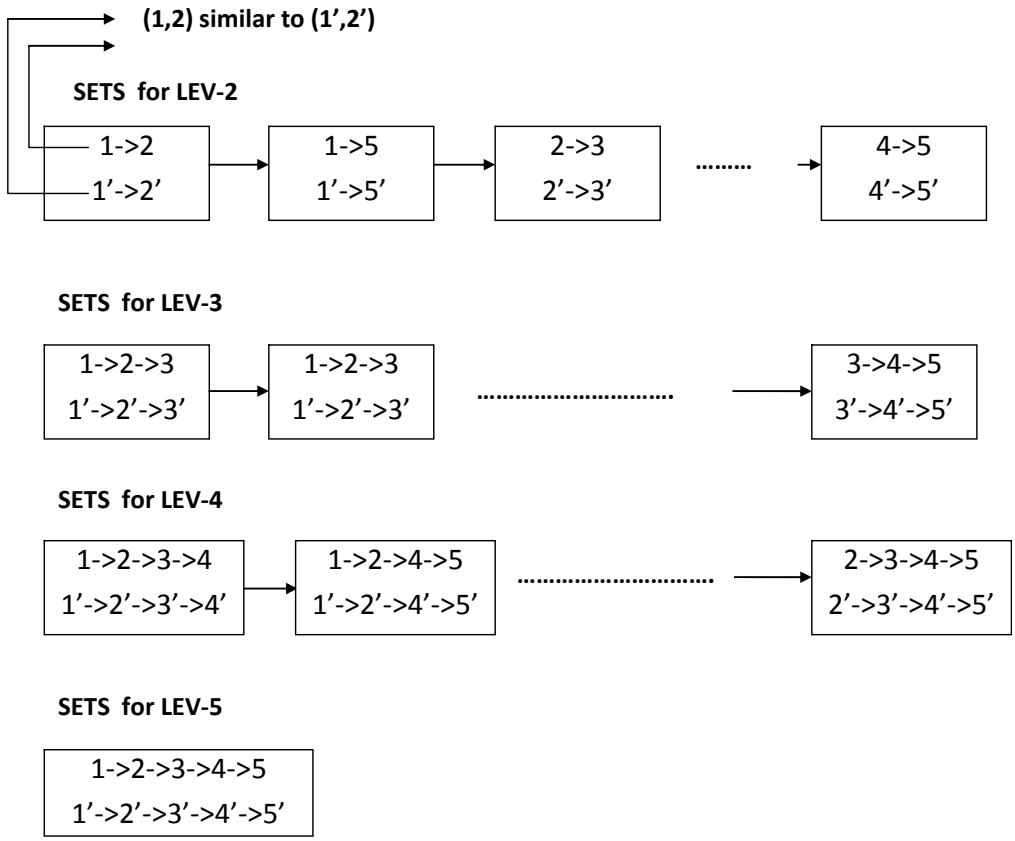

Figure 3.2: CoverSet of subgraphs for the Sample QueryGraph.

For the given Figure 3.1, a query graph is shown with two pentagons being identified as largest similar subgraphs.

Let us examine how two of the most relevant works construct plans for this query graph.

1. According to [40], these two pentagons are identified and the query plan which is generated for one pentagon is immediately used for the other. The plans are immediately executed. The two pentagons are replaced by two result relations and the new query graph would be two nodes connected to each other. Further optimization is done on this new query graph which has a single edge and two vertices using some greedy approach. This means the algorithm has somehow fixed that 1,2,3,4,5 should be joined first and same

about 1',2',3',4',5' and executed those plans.

2. In the case of pruning algorithm according to [5], hubs will be identified. In this case, hubs are 2' and 5. It will then prune away a few edges which it thinks are costlier in terms of table sizes and selectivity and continues optimization on the new pruned query graph.

But according to our algorithm, an optimal plan has to be generated with lesser cost without pruning and not fixing join order. From 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' such that the table size differences are within acceptable error bounds. These pairs are referred to as seed lists. The members of a seed list are grown to find similar subgraphs of size 2 (or 2 vertices). Similar 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 selectivities of (1,2) and (1',2') 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 3.2).

These similar subgraphs can lead to reuse of plans and thus give savings. But if largest sub query graphs alone are re-used as in [40], the savings are mild in the case of Dynamic Programming. Thus cover set of similar subgraphs are generated as shown in Figure 3.2. It should be noted that each similar subgraph set in this example consists only of 2 entries. This is because in Figure 3.1, we just have 2 subgraphs. If the number of similar subgraphs are "k", we have to make "k" entries into each set.

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').

Subgraph sets from level 2 are grown to generate subgraph sets for level 3. At level 3, (1,2,3)'s plan is reused for (1',2',3'). For(1,2,3) DP plan generator would have originally constructed many plans with various join orders like (2,3) joined first followed by 1, (1,2) joined first followed by 3 and so on. (Please refer Figure 3.3). All these plans would have incurred memory overhead and optimization time overhead. The same expenditure would have been done for (1',2',3') too which is avoided by our algorithm. Suppose (2,3) followed by 1 is the cheapest plan out of all the plans following various join orders for (1,2,3) the same is reused for (1',2',3').

Suppose DP has to build a plan for (1',2',3'), it will generate all possible join orders and build all possible plans and finally sets the cheapest join order and stores that plan as the cheapest plan. In further iterations of DP at higher levels, this cheapest plan is used whenever the need to use the plan for (1',2',3') arises. Because of plan reuse, we are actually avoiding all those steps and directly generating the plan with ideal join order for (1',2',3').

As mentioned earlier, reuse means reconstruction of plan in the same way. Suppose if merge join is used at root node for $Plan$, the same join method will be used for $Plan'$ also. Likewise, if a leaf node in $Plan$ has an index plan constructed on its base table, an index plan should be constructed for the corresponding leaf node in $Plan'$ provided an index has been built for this relation. Hence it is vital to check if the relations are similar not only with respect to table sizes but also indexing information. Plan reuse is clearly illustrated in Figure 3.3

Similar subgraph growth and plan reuses are done at level 4 and 5 too. As there are no more subgraph sets from level 6 to 10, pure Dynamic Programming is used according to the basic algorithm. But nowhere in this entire procedure have

we prematurely executed a copied plan unlike [40]. The plan copying is done only because the sub queries are similar, but the copied plan is not executed immediately. Finally, choosing the optimal plan at a particular level and execution is done by DP in its usual way. Likewise, we are not getting rid of certain join candidates to speed up our process unlike [5]. Thus without fixing join order or pruning, savings are brought up.

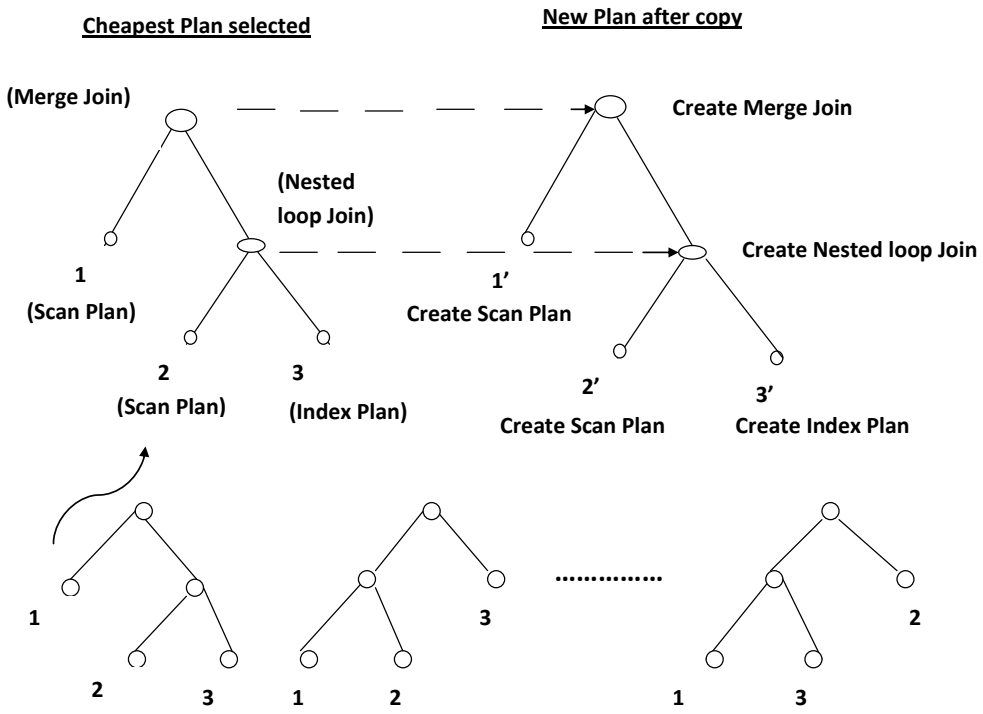

Figure 3.3: Cheapest *Plan* reuse for *Plan*'.

3.2 Building query graph

A query graph is a set of vertices "V" and edges "E" such that each vertex denotes a relation "R" present in the query and every edge incident on it represents the

predicate in which the vertex participates. A cover set is defined as the set of all possible subsets of the given set. Similarly the cover set of similar subgraphs should consist of all possible sets of similar subgraphs of various sizes in the given query graph.

Algorithm 3 : makeQueryGraph

Require: *Query*
Ensure: *QueryGraph*
1: initialize $adjacencyList = \emptyset$
2: $predSet$ = extractJoinPredicates($Query$)
3: **while** $predSet$ has more predicates **do**
4: $predicate$ = popFromPredSet($predSet$)
5: makeEntryAdjacencyList(left($predicate$),right($predicate$))
6: **end while**
7: $QueryGraph = adjacencyList$
8: **return** $QueryGraph$

Algorithm 3 explains how join predicates are extracted from the query and corresponding edges are added to the query graph. Entry of the *right* table's relation id is appended to the $left$ table's row in the adjacency list (representing query graph) and vice versa. Each node in the "QueryGraph's" adjacency list is a structure holding selectivity and relation size information in addition to the relationId. makeEntryAdjacencyList() makes these entries.

3.3 Generating cover set of similar subgraphs

Given a query $\hat{Q}$, with a set of relations R=$\{R_1, \ldots, R_n\}$ and a set of predicates P=$\{p_1, \ldots, p_k\}$, let "Q" be the query graph for $\hat{Q}$. Let V=$\{v_1, \ldots, v_n\}$ be the set of vertices and E=$\{e_1, \ldots, e_r\}$ be the set of edges in Q with an edge e_i corresponding to a predicate p_j where $p_j \epsilon P$. Each vertex in Q represents a table instance and each edge corresponds to a set of predicates, since two table instances can participate in multiple join predicates.

A pair of common subgraphs $\{S, S'\}$ is defined as a pair of isomorphic subgraphs having the same graph structure and features, i.e, each vertex in S should have a corresponding vertex in S' with same table size and each edge in S should have a corresponding edge in S' with same selectivity. But such isomorphic graphs are difficult to find in all cases. Thus, if we can relax row difference and selectivity difference, similar subgraphs can be found in all queries.

A pair of similar subgraphs $\{S, S'\}$ is defined 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.

The 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. Unlike [40], we do not want to generate the largest similar subgraphs. Rather we want to generate all sized-similar subgraphs. This is because, if we want to push DP to higher levels, at each level of plan generation, we need savings. This is possible if there are sets of similar subgraphs at each level so that plan re-use can be done at each step progressively.

So 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. Generation of cover set involves two stages:

1. Formation and growth of seed list to form $lev2$ 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.

Algorithm 4 : buildCoverSet

Require: $QueryGraph$
Require: $relErrorBound$
Require: $selErrorBound$
Ensure: $coverSet$
1: initialize $coverSet = \emptyset$
2: $seedList$ = makeSeedList($QueryGraph$)
3: lev=2
4: $Sets_2$ = growSeedList($seedList$)
5: **while** $Sets_{lev}$ can be extended to get new subgraph sets **do**
6: $Sets_{lev+1}$ = growSubGraph($Sets_{lev}$)
7: lev++
8: **end while**
9: **return** $\sum_{i=2}^{lev} Sets_i$

3.3.1 Construction of seed List

Seed list construction involves partitioning all the base relations participating in the query into various groups based upon their table size differences and indexing information. 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. Seed list for the query graph in Figure 3.1 is shown below in Table 3.1.

As per algorithm 5, each row in the $seedList$ corresponds to a group of similar seeds. So for each relation R_i, a flag $selected[i]$ is initialized to 0. The algorithm checks if R_i can match with any of the existing groups (or rows) in the $seedList$.

Table 3.1: SeedList

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

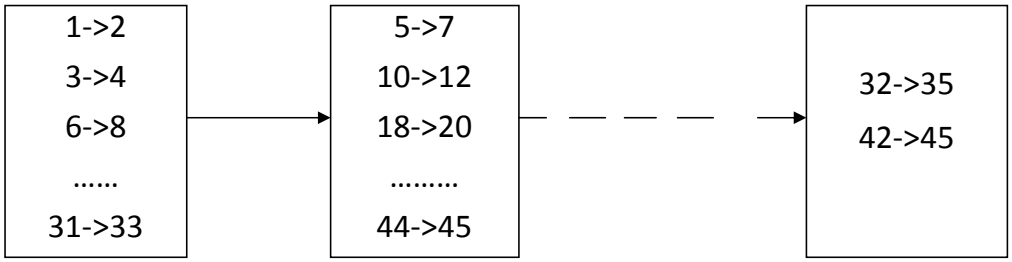

Figure 3.4: Sets of similar subgraphs for level 2.

If there is a match, R_i is appended to the row and its flag is set to 1. If there are no matches, upon looking at the flag value of 0, a new row (group) is added to the *seedList* with R_i as its first member.

3.3.2 Growth of seed list and subgraphs

Growing the seed list implies the formation of sets of similar subgraphs for level 2. Each set holds graph entries that are similar to each other, and each graph is represented as a linked list of relation-id s of base tables.

An example list of level 2 similar subgraph sets can be seen in Figure 3.4.

Each set in Figure 3.4 is shown to contain many graphs. Internally it is not just a set of graphs, but a set of structures with each structure holding a graph. The structure holds additional information like the *bitmapset* of relation id's in the graph and also a join-relation holding a pointer to the set of result plans constructed for that graph. The *bitmapset* facilitates comparison of graphs thereby avoiding scanning the graphs to save processing time. The join-relation is mainly used in

Algorithm 5 : makeSeedList

Require: $QueryGraph$
Require: $relErrorBound$
Ensure: $seedList$
1: initialize $seedList = \emptyset$, $numGroups = 0$
2: $numOfRelations = numOfRows(QueryGraph)$
3: **for** $i = 0$ To $numOfRelations$ **do**
4: initialize a flag array $selected[i] = 0$
5: **for** $groupId = 0$ To $numGroups$ **do**
6: $compareSeed = getFirstSeed(seedList[groupId])$
7: get $relSize(R_i)$ from $QueryGraph$
8: get $relSize(compareSeed)$ from $QueryGraph$
9: $R_j = compareSeed$
10: **if** $\frac{|relSize(R_i) - relSize(R_j)|}{max(relSize(R_i), relSize(R_j))} < relErrorBound$ **then**
11: **if** $bothIndexed(R_i, R_j)$ Or $bothNotIndexed(R_i, R_j)$ **then**
12: append R_i to $seedList[groupId]$
13: $selected[i] = 1$
14: break
15: **end if**
16: **end if**
17: **end for**
18: **if** $selected[i] == 0$ **then**
19: $numGroups++$
20: $FirstSeed(seedList[groupId]) = R_i$
21: **end if**
22: **end for**
23: **return** $seedList$

plan copying and plan reuse which will be described in the later sections.

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 style of algorithms. Growth of list and growth of sets of similar subgraphs are illustrated in Figure 3.5.

Given a seed list and a query graph, $growSeedList$ explores possibilities of forming level 2 sets of subgraphs from each seed belonging to every group present in the seed list.

To grow an arbitrary seed $Seed_i$, we need to fetch the neighbours of $Seed_i$ from the query graph. If $neighbours(Seed_i)$ denotes the set of neighbours, each entry

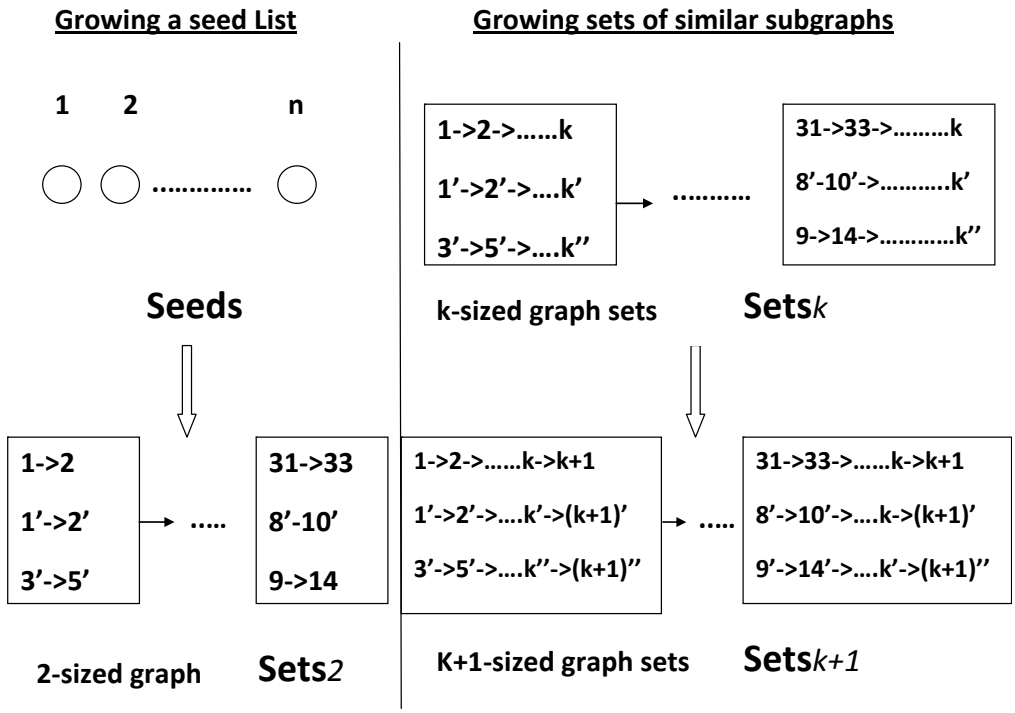

Figure 3.5: Growth of seeds versus growth of subgraphs.

in this list 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 neighbours 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 $Seed_i$. 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 the other seeds. Other seeds are feature wise (table size is a feature) similar to the candidate seed. So we are covering all possible ways in which similar subgraphs can be identified since a subgraph not containing the same seed or a similar seed can never be similar to the candidate graph. This signifies the importance of seed list construction.

This method can be illustrated with an example shown in Figure 3.6. Algorithm 6 explains the above steps in a detailed manner. Lines 4 to 9 explain the generation of a candidate graph using the neighbour list of the seed. Lines 10 to 21 explain the generation of similar subgraphs using the same seed and its remaining neighbours. Lines 22 to 27 explain the generation of similar subgraphs from other seeds in the same seed group. The algorithm also explains how to avoid repetition of sets. The first candidate graph in the set has to be thoroughly checked for duplication in all the existing sets of similar subgraphs at level 2 (in line 7). The remaining graph entries should do a duplication check only in the last (or the latest) set before getting appended to it (lines 13 and 26). This is valid because if the top entry of the set is new, the remaining similar candidates are bound to be fresh as well.

Two edges 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. The only difference is that a list of level k sets are being grown instead of seeds. 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. Figure 3.7 illustrates the difference between growing a seed and growth of a graph. 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 consisting of nodes S_i to S_k, neighbours of each node, namely, N_i to N_k arrive from the query graph. So in this case, there are many more candidate subgraphs for level(k+1) formed from the k-sized subgraph and any of the neighbours.

Similar subgraphs for a candidate subgraph of level (k+1) are identified 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.

Algorithm for subgraph growth is being listed in Algorithm 7 and it is almost similar to $growSeedList()$ except that the input is $Sets_k$ instead of $seedList$ and the output is $Sets_{k+1}$ instead of $Sets_2$. Also in line 4, $growSeedList$ extracts neighbours of the seed. Whereas, in this case, we need to extract neighbours of all the vertices participating in the subgraph to be grown.

3.4 Plan generation using similar sub queries

The basic algorithm has already been listed in Section 3.1 as Algorithm 2. Lines 1 and 2 of that algorithm have been described till now in sections 3.2 and 3.3. The crux of our approach is using the cover set of similar subgraphs for plan generation which is stated in lines 3 to 5. As mentioned earlier in Section 3.3.2 using Figure 3.4, each set of similar subgraphs holds graphs encapsulated in well-defined structures. By accessing those structures, we can retrieve not only the base relation ids of tables participating in the graph but also pointers to the subquery plans. A join-relation holds the pointers initially pointing to $NULL$ as long as the plans for that subgraph haven't been generated. But once the plans are generated, pointer entries are made. Suppose there are "n" similar subgraphs S_1, S_2,..., S_n in a set, and one of them, say S_i had its set of plans generated through the traditional Dynamic Programming approach. It is not just one plan but a *set* of plans, because each plan follows a different join order of the constituent relations and all these plans are held in memory. 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. Pointer to the plan for the new subgraph is stored in its structure.

In Figure 3.8, a new plan has to be constructed for the join relation set (5,6,..k'). So our algorithm scans all the subgraph sets at level "i" (as mentioned in Figure 3.5 in Section 3.3, the collection of sets at level "i" is termed $Sets_i$). If it does not find the entry of (5,6,..k') in any of the sets, it will build the set of plans for the relations using DP. But in this case the entry is found in one of the sets. Then we check if any other subgraphs in that set had their plans built already. If not, we have to build the set of plans for (5,6,..k') using DP and subsequently make an entry of the pointer to the plan list in the set. But in this case, the relation set is

in the same similar subgraph set as (1,2,..k). So from the list of plans generated for (1,2,..k), the cheapest plan is extracted and reused for construction of a new plan for (5,6,..k'). As against "i" plans constructed for (1,2,..k) only one plan is constructed for (5,6,..k') bringing memory and computation savings. In Chapter 4, we are going to show that the cost incurred in scanning the sets of subgraphs combined with the cost of plan reuse as proposed by our algorithm is very less compared to the cost incurred in fresh construction of the plan set done by DP.

Algorithm 8 explains the plan reuse. Reuse of a plan happens from $Plan$ to $newPlan$ using a recursive function. The function takes the root node of $Plan$ and the subgraphs corresponding to $Plan$ and $newPlan$ as inputs and gives out the root node of the $newPlan$ as output. While building a plan node at each level (be it root or intermediate node), the function checks the type of join used for the original plan at that level and reuses the same kind of join. The left and right child nodes for the current node of $newPlan$ are built by recursive calls of this function on left and right child nodes of the current node from $Plan$. 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. To identify the corresponding base relation in $newPlan$ for a base relation from $Plan$, the subgraphs $Subgraph$ and $Subgraph'$ are used for lookup. Suppose an index scan is built on a base relation R_a, the recursive function will identify the corresponding base relation, R'_a from the subgraph for $newPlan$ and then build an index scan on R'_a.

Example for Plan re-use has already been illustrated in Figure 3.3.

3.5 Memory efficient algorithms

Algorithm 2 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 the number of relations crosses 30 and the query graph is dense), holding the entire cover set in memory is not possible. Even the generation of the cover set takes longer time. So a more memory-efficient approach has been adopted.

3.5.1 Improving Cover set generation

To save time and memory spent on the generation of cover set, $growSubgraph()$ chooses to selectively grow sets containing large number of subgraphs. Essentially, the more subgraph sets we have, the higher the query plan reuse is among the similar subgraphs. But in the case of complete (or very dense) query graphs of large sizes, the number of relations is high and the number of edges between the vertices in a subgraph is as high or close to $|numOfVertices|^2$ where $|numOfVertices|$ indicates the number of vertices in a subgraph. In this scenario, the number of similar subgraph sets will be extremely huge. Out of them, there will be some sets having many subgraph entries and few other sets may have less number of entries.

In Figure 3.9, 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 lose the plan reuse opportunity among the subqueries corresponding to those subgraphs. But it is worthwhile given the amount of time we save by not generating them, and the price we pay is that, in plan generation phase we cannot reuse plans and we need to freshly generate plans for candidates belonging to those pruned subgraph sets. This procedure will not affect the optimality of the plan because pruning is done during common subgraph generation but not to the DP candidates themselves. This pruning (or avoiding generation) of certain similar subgraph sets is done by fixing a parameter $allowedStrength$. If the number of subgraphs within a set is less than $allowedStrength$, that set is not grown. It should be noted that for sparse query graphs or averagely dense query graphs, this pruning is not even required.

Fixing $allowedStrength$ happens in the following manner. Among all the sets, the set with the largest number of subgraphs is examined and this count is stored in $largestSetStrength$.
$allowedStrength = \frac{largestSetStrength}{j}$. For different values of "j", $allowedStrength$ is set to different values. So, the prune factor can be defined as $\gamma = \frac{1}{j}$. When γ=1, there is absolute pruning and no growth of sets at that level. When γ=0.5, the criterion for a subgraph set to grow is that it should contain at least as many similar subgraphs as half of the strength of the largest subgraph set at that level. For example in Figure 3.10, the set with strength of 60 is identified to be the largest and prune factor=0.25. So all sets which have at least as much as $\gamma * 60 = 15$ are grown, remaining sets are pruned.

3.5.2 Improving Plan generation

In section 3.5.1, 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. But cover set is essential for looking up to similar subgraphs and cannot be deleted. 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 as shown in Algorithm 9. First level 2 similar subgraphs sets are built and stored in memory by growing the seed list of similar base relations. Immediately for level 2 in the DP lattice, subquery plans are constructed. Then level 3 similar subgraph sets are generated from level 2 similar subgraph sets using *growSelectedSubGraph* and level 2 subgraph sets are deleted as they are no longer required. Because now we need to construct plans for level 3, and for that to happen, we need to look up to subgraph sets corresponding to level 3 only.

Growth of subgraphs happens only on need, and the subgraphs are deleted when they are no longer required.

If we want to generate plans at an arbitrary 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. Refer lines 7 to 9 in Algorithm 9 to understand dynamic subgraph construction and deletion on the fly. Line 11 portrays the construction of subquery plans at a particular level by looking up to subgraph sets corresponding to that level.

3.6 Embedding our scheme in Iterative Dynamic Programming (SRIDP)

We observed that with our scheme in Dynamic Programming, the memory savings come from reduction of the number of various alternatives arising from the different join orders of a combination of relations. A set of m relations typically needs O(m!) join orders. 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. For instance, we have a candidate relation set {1,2,3,4} for which a query plan needs to be built. The optimizer generates various plans for different join orders using different join methods before selecting the cheapest plan. Even after setting the cheapest plan with an ideal join order, PostgreSQL's version of DP still holds the plans for the remaining join orders in memory. If our scheme identifies that relation set {5,6,7,8} is similar to {1,2,3,4}, we build a plan for 5,6,7,8 similar to the cheapest plan of {1,2,3,4} and thereby avoid generating plans for the remaining join orders of {5,6,7,8}. This gives memory savings in our scheme. But we should note that no particular combination of relations is being denied plan construction by our scheme. If the number of join candidates at a level "r" is ${}^{n}C_{r}$, our scheme still constructs plans for all the ${}^{n}C_{r}$ candidates because we do not trade savings with optimality. Because of not pruning the number of join candidates despite plan reuse, there is a chance that for certain queries, our scheme can push the Dynamic Programming method of query optimization to a few more levels in the DP lattice and stop. For example given a complex query with 30 relations, DP may run out of memory at level 15 in the DP lattice. Our scheme may run out of memory at level 25, 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 30 relations, if we set "k" to any value higher than 15, IDP cannot run to completion. Because if k=15, IDP needs to use traditional DP method of query optimization from level 2 to level 14 in the DP lattice. Only at level 15, it can greedily choose, typically out of ${}^{30}C_{15}$ join candidates, only one join candidate whose plan cost is cheaper than the remaining ${}^{30}C_{15} - 1$ join candidates (and append the plan to 1-way plans at lattice level 1) before resuming 2nd round of DP on levels 16 to 30 in the DP lattice. (It should be understood that lattice levels 16 to 30 are portrayed as levels 1 to 15 in iteration number 2 of IDP.) So if "k" is set to a value higher than 15, IDP is forced to run DP for levels 2 to k-1 which will run out of memory at level 15. Hence, IDP as such cannot run for a "k" higher than 15.

Whereas, with our scheme embedded in IDP, the maximum possible value of "k" can be stretched to 25. That means for levels 2 to 24, sub query reuse based DP can run without memory issues and at level 25, greedy method of plan selection can be applied. 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=25) will be higher than IDP(k=15). 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 10

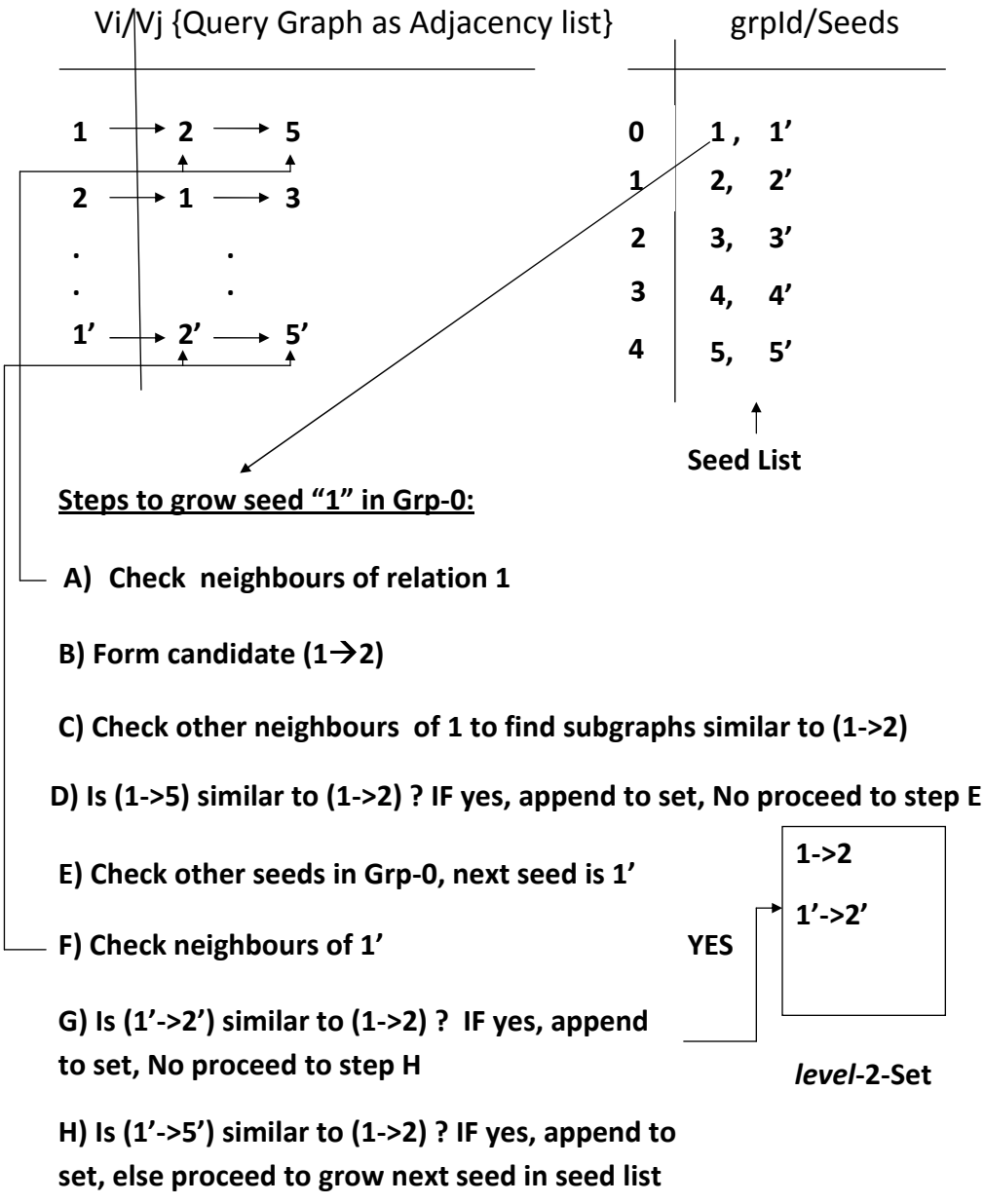

Figure 3.6: Example to illustrate growth of a seed in the seed list.

Algorithm 6 : growSeedList

Require: *seedList*
Require: *QueryGraph*
Require: *rowErrorBound*
Require: *selErrorBound*
Ensure: $Sets_2$

```
1: initialize Sets_2 = ∅, latestSet = ∅
2: for each row in seedList indicated by seedGroup do
3:    for each seed in the group seedGroup do
4:       neighbours(seed)=ExtractNeighbours(QueryGraph,seed)
5:       for each neighbour in neighbours(seed) do
6:          candidateGraph = makeNewGraph(seed, neighbour)
7:          if candidateGraph already exists in any of the sets in Sets_2 then
8:             continue
9:          end if
10:         while there are more neighbours in neighbours(seed) do
11:            extract another neighbour' from neighbours(seed)
12:            candidateGraph' = makeNewGraph(seed, neighbour')
13:            if similar(candidateGraph, candidateGraph') within row and selectivity
               error bounds and candidateGraph' not a duplicate then
14:               if FirstMember(latestSet)!=candidateGraph then
15:                  latestSet = makeSet(candidateGraph,candidategraph')
16:               else
17:                  append candidateGraph' to latestSet
18:               end if
19:               append latestSet to Sets_2
20:            end if
21:         end while
22:         while there are more seeds in seedGroup do
23:            extract another seed' from seedGroup
24:            grow seed' using queryGraph to form CandidateGraph'
25:            candidateGraph' = makeNewGraph(seed', neighbour(seed'))
26:            add candidateGraph' to latestSet if it is similar to candidateGraph and
               not a duplicate entry
27:         end while
28:      end for
29:   end for
30: end for
31: return Sets_2
```

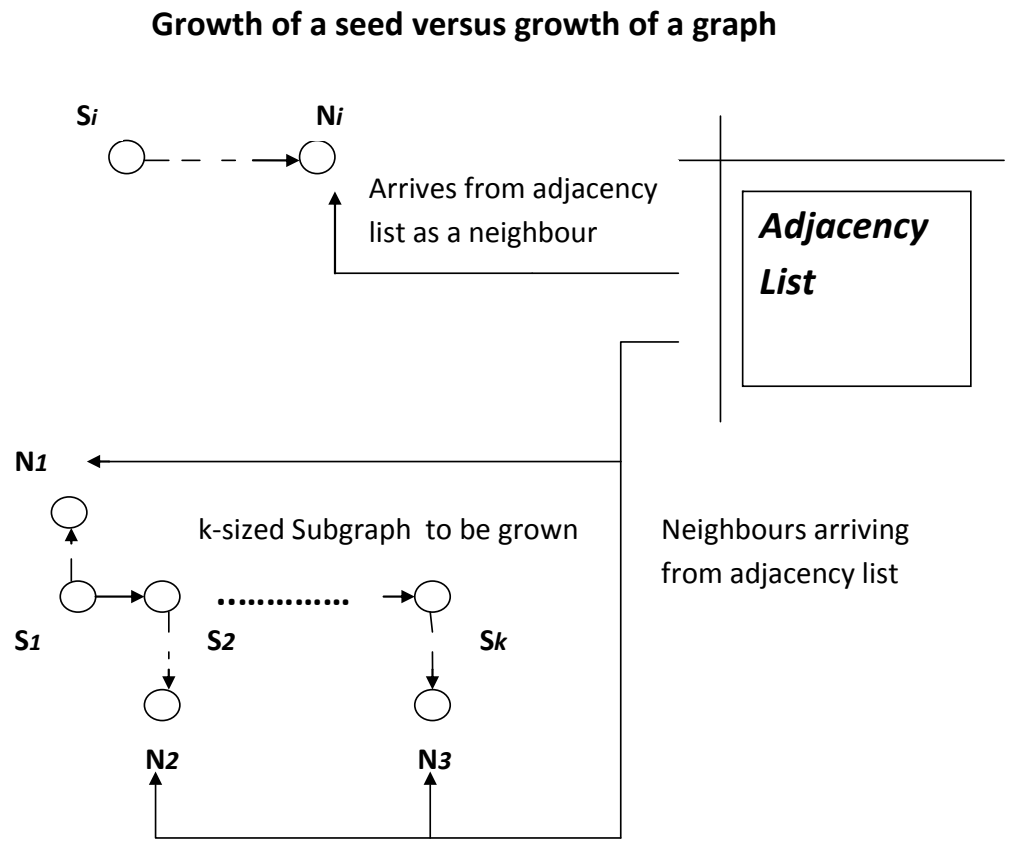

Figure 3.7: Growth of a seed versus growth of a subgraph.

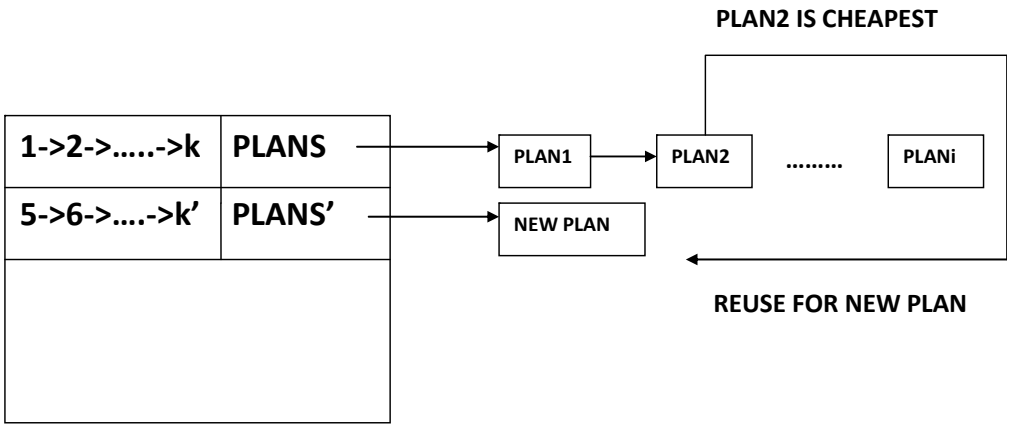

Figure 3.8: Plan reuse within the same similar subgraph set.

Algorithm 7 : growSubGraph

Require: $Sets_k$
Require: $QueryGraph$
Require: $rowErrorBound$
Require: $selErrorBound$
Ensure: $Sets_{k+1}$

initialize $Sets_{k+1} = \emptyset$, $latestSet = \emptyset$
for each set in $Sets_k$ indicated by $setGroup$ **do**
  for each $subGraph$ in the group $setGroup$ **do**
    $neighbours(subGraph)$=ExtractNeighbours($QueryGraph$, $\sum_{i=1}^{k} vertex_i(subGraph)$)
    for each $neighbour$ in $neighbours(subGraph)$ **do**
      $candidateGraph$ = makeNewGraph($subGraph$, $neighbour$)
      if $candidateGraph$ already exists in any of the sets in $Sets_{k+1}$ **then**
        continue
      end if
      while there are more neighbours in $neighbours(subGraph)$ **do**
        extract another $neighbour'$ from $neighbours(subGraph)$
        $candidateGraph'$ = makeNewGraph($subGraph$, $neighbour'$)
        if similar($candidateGraph$, $candidateGraph'$) within row and selectivity error bounds **then**
          if FirstMember($latestSet$)!=$candidateGraph$ **then**
            $latestSet$ = makeSet($candidateGraph$,$candidategraph'$)
          else
            append $candidateGraph'$ to $latestSet$
          end if
          append $latestSet$ to $Sets_{k+1}$
        end if
      end while
      while there are more subgraphs in $setGroup$ **do**
        extract another $subGraph'$ from $setGroup$
        grow $subGraph'$ using queryGraph to form $CandidateGraph'$
        $candidateGraph'$ = makeNewGraph($subGraph'$, $neighbour(subGraph')$)
        add $candidateGraph'$ to $latestSet$ if it is similar to $candidateGraph$ and not a duplicate entry
      end while
    end for
  end for
end for
return $Sets_{k+1}$

Algorithm 8 : Recursive function for Plan reuse

Require: *node*: root node of *Plan*, *Subgraph*, *Subgraph′*
Ensure: *newNode*: root node of *newPlan*

if *node* is a joinPlanNode **then**
  joinType = joinType(*node*)
  leftChild = PlanReuse(leftChild(*node*), *Subgraph*, *Subgraph′*)
  rightChild = PlanReuse(rightChild(*node*), *Subgraph*, *Subgraph′*)
  newNode = buildJoinPlan(*joinType*, *leftChild*, *rightChild*)
else
  if *node* is a ScanPlan **then**
    oldBaseRel = BaseRel(*node*)
    newBaseRel = FindBaseRel(*Subgraph*, *Subgraph′*, *oldBaseRel*)
    newNode = buildScanPlan(*newBaseRel*)
  end if
  if *node* is an IndexPlan **then**
    oldBaseRel = BaseRel(*node*)
    newBaseRel = FindBaseRel(*Subgraph*, *Subgraph′*, *oldBaseRel*)
    newNode = buildIndexPlan(*newBaseRel*)
  end if
end if
return *newNode*

Algorithm 9 : Memory efficient Plan generation with subgraph reuse

Require: *Query*(Selectivity and row error bounds are pre-set)
Require: *pruneFactor*
Ensure: *plan* in the case of "explain query", *result* if query is executed

QueryGraph = makeQueryGraph(*Query*)
seedList = makeSeedList(*QueryGraph*)
lev=2
$Sets_2$ = growSeedList(*seedList*)
Plans[2] = newBuildPlanRel(*Plans*,$Sets_2$)
for *lev*=3 to *levelsNeeded* **do**
  if $Sets_{lev-1}$ can be extended to get new subgraph sets **then**
    $Sets_{lev}$ = growSelectedSubGraph($Sets_{lev-1}$, *pruneFactor*)
    delete($Sets_{lev-1}$)
  end if
  Plans[*lev*] = newBuildPlanRel(*Plans*,$Sets_{lev}$)
end for
return *Plans*

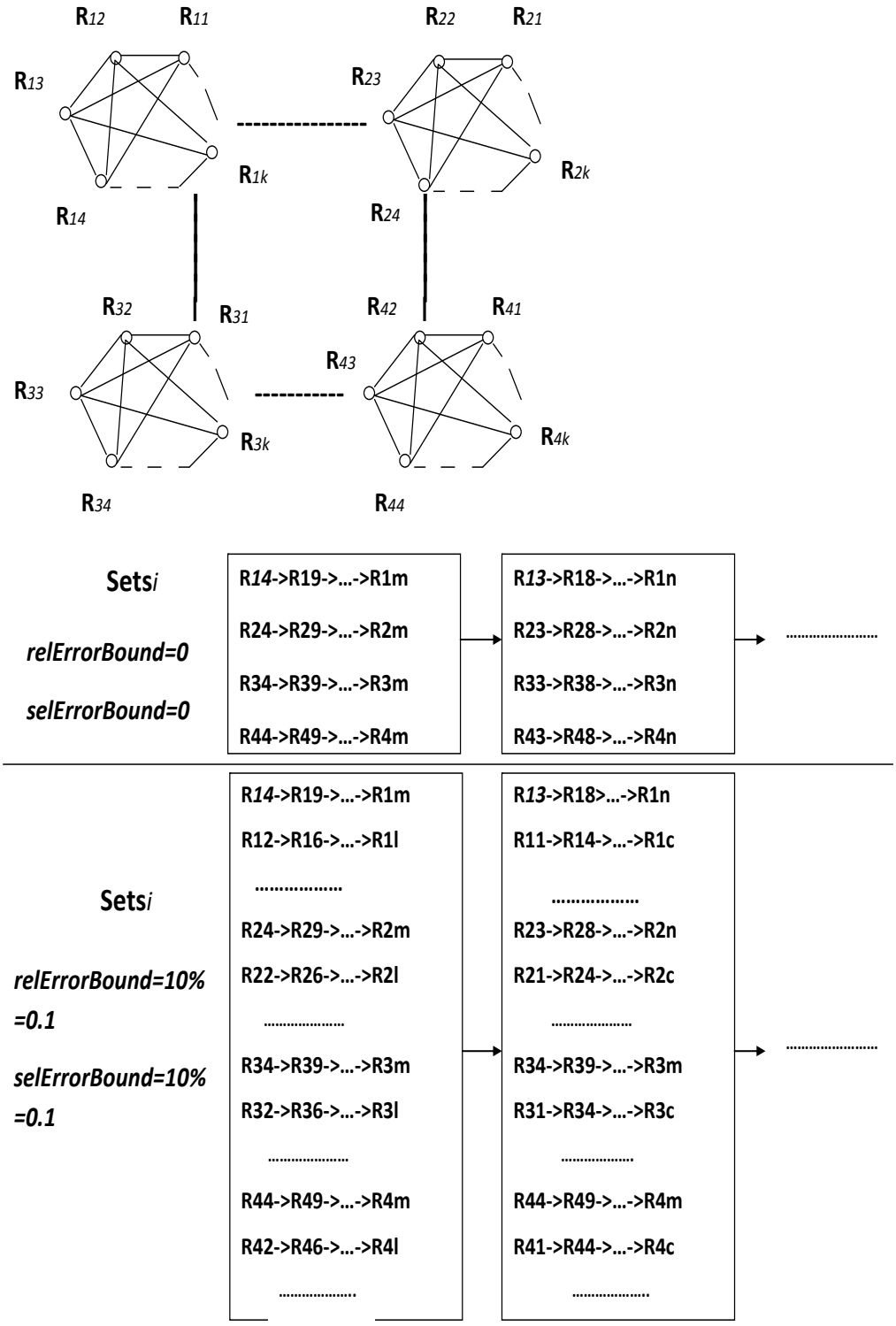

Figure 3.9: Increase in population of a subgraph set with error bound relaxation.

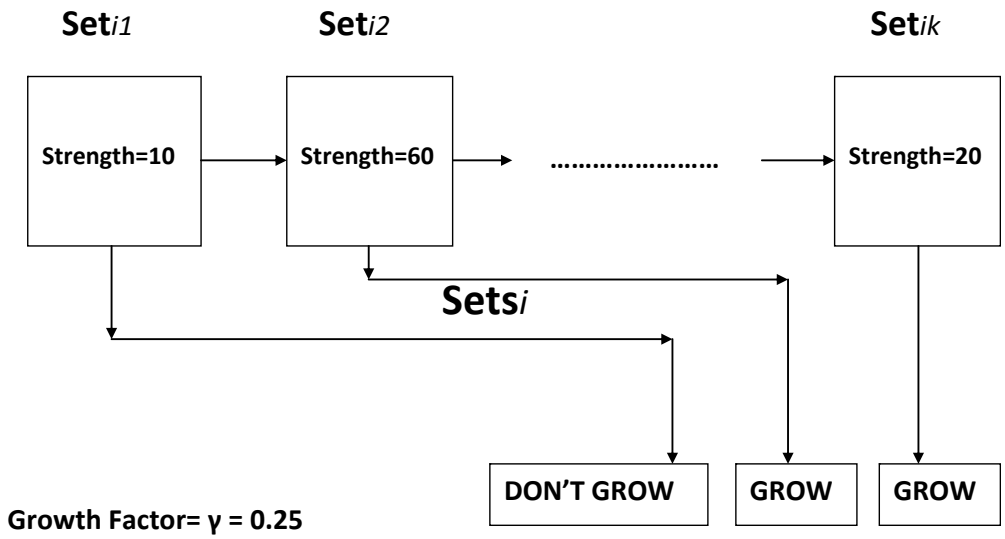

Figure 3.10: Growth of selected subgraph sets.

Algorithm 10 : Memory efficient Sub query plan reuse based IDP : SRIDP

Require: $Query$
Require: k
Ensure: $queryplan$
1: $numRels = numOfRels(Query)$
2: $numOfIterations = numRels/k$
3: $QueryGraph$ = makeQueryGraph($Query$)
4: $seedList$ = makeSeedList($QueryGraph$)
5: **for** $iteration = 0$ to $numOfIterations - 1$ **do**
6: **for** $lev = iteration1$ 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: $Plans[lev]$ = makeGreedySelection($Plans[lev]$)
19: $participatingRels$ = relationsIn($Plans[lev]$)
20: $Plans$[1] = $Plans$[1] - 1-wayPlansFor($participatingRels$) + $Plans[lev]$
21: **end for**
22: **return** $Plans[lev]$

CHAPTER 4

PERFORMANCE STUDY

In this chapter, we are going to measure the performance of our approach and compare it with the existing methods. We fix the default values to the parameters and vary each one of them thus producing a comparative study of the behaviors of different schemes with respect to the particular parameter thus knowing how sensitive that parameter is in influencing performance.

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. While populating the database, the user can either choose to set the relation sizes manually for each relation or set the relation size of the first table and the percentage difference between relation sizes of any two consecutive tables R_i and R_{i+1}. The way the relation sizes are set and the relations are populated determines the seed list and common subgraphs that are generated eventually. Therefore, we cover various scenarios in these experiments that bring about the structural differences in the database. In all the scenarios, the table sizes vary from 1000 to 8,000,000 tuples.

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 our scheme, 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).

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).

4.1 Experiment 1: Varying the number of relations

Figure 4.1 portrays how our scheme SRIDP "Subquery plan Reuse based IDP" pushes the value of "k" beyond what IDP is capable of. The default settings are listed in Table 4.1.

Table 4.1: Default Parameter Settings

Density Level	**Similarity relaxation**	**prune factor**
2	30,30	30

We have various density levels with which queries are generated, namely, 1,2,4,8. It is a randomized manner of generating queries by fixing a lower and upper bound on the number of allowable predicates for a particular density level. The default density setting used is 2 which is the 2nd level of density from the highest. 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 "k" the maximum allowed degree is defined as $\#Relations/k$.

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's denominator is listed as 30, which means that similar subgraph sets of strength with 1/30 th fraction of the highest populated subgraph set should be pruned off. The denominator of the prune factor is listed in the table. 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".

It can be seen in Figure 4.1 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. It must be noted that Skyline DP 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.

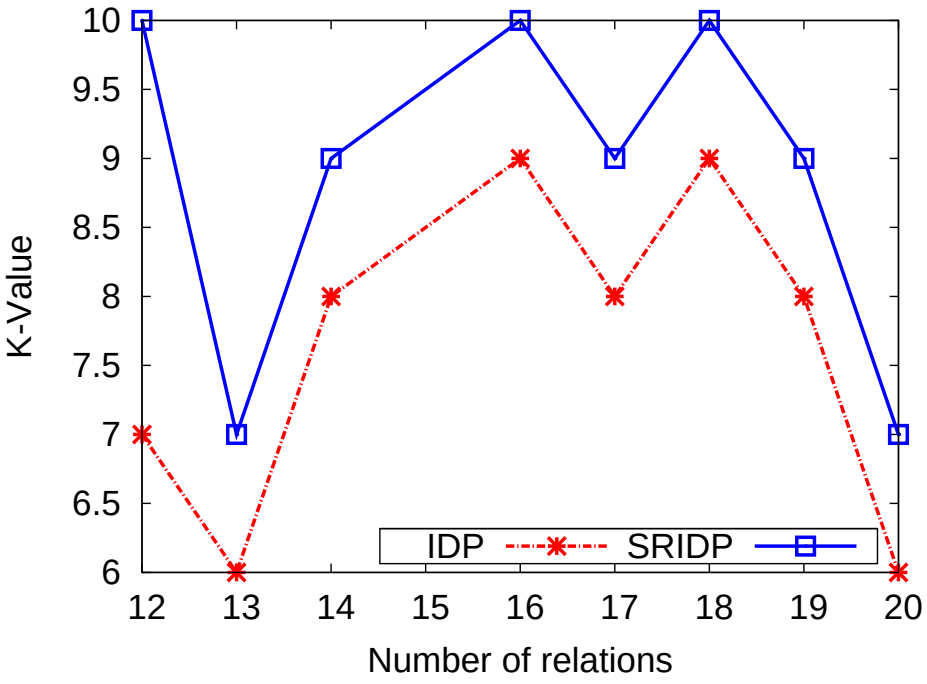

Figure 4.1: K-value versus number of relations.

Figure 4.2 shows whether the increase in "k" using our scheme has translated to improved plan quality. Actual costing (in plans) is being listed here in Table 4.2 since it is difficult to make out the values from the figure.

It can be noted that plan quality using SRIDP has been better than that of IDP at most of the points. Except that at 14,16 and 19 plan quality has been worse slightly. The plan quality using our scheme was better manifold at 12, 13, 17, 18 and 20. This highly depends on the effect of similar subplan reuse over the

Table 4.2: Costing in Plans for medium dense queries

Number of relations	Plan cost IDP	Plan Cost SRIDP	Skyline DP(pruning)
12	34571.29	33174.26	out of memory
13	40316.21	37849.12	out of memory
14	48755.22	49652.81	out of memory
16	392045.96	407081.06	out of memory
17	544761.71	85452.12	out of memory
18	415910.6	359533.91	out of memory
19	340097.2	422955.86	out of memory
20	566904.82	420731.09	out of memory

specific queries at those points. Increasing the value of "k" is combated by the similarity relaxation. At number of relations=14, the value of "k" using SRIDP has risen from 6 to 7. But, probably the sub plans that were reused for that query may not be the ideal ones. This led to a drop in plan quality but by a meager percentage. Figure 4.3 shows the optimization time in seconds. This sacrifice is worthwhile given the enhancement in plan quality. SRIDP takes longer than IDP because cover set generation needs time. 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 4.4 and 4.5 show the execution time and total running time respectively for a set of medium dense (density level 2) queries. It should be noted that this is a different random query set from what has been used for Figure 4.3. Nevertheless these readings emphasize that the gain in execution time is worthwhile the optimization time overhead, thus making SRIDP win in overall query running time.

Figure 4.6 shows the savings in plan cost and enhancement in plan quality using SRIDP as compared to IDP for high-density queries of density level 1 . It should be noted that skyline DP based on pruning of subplans has only one point plotted

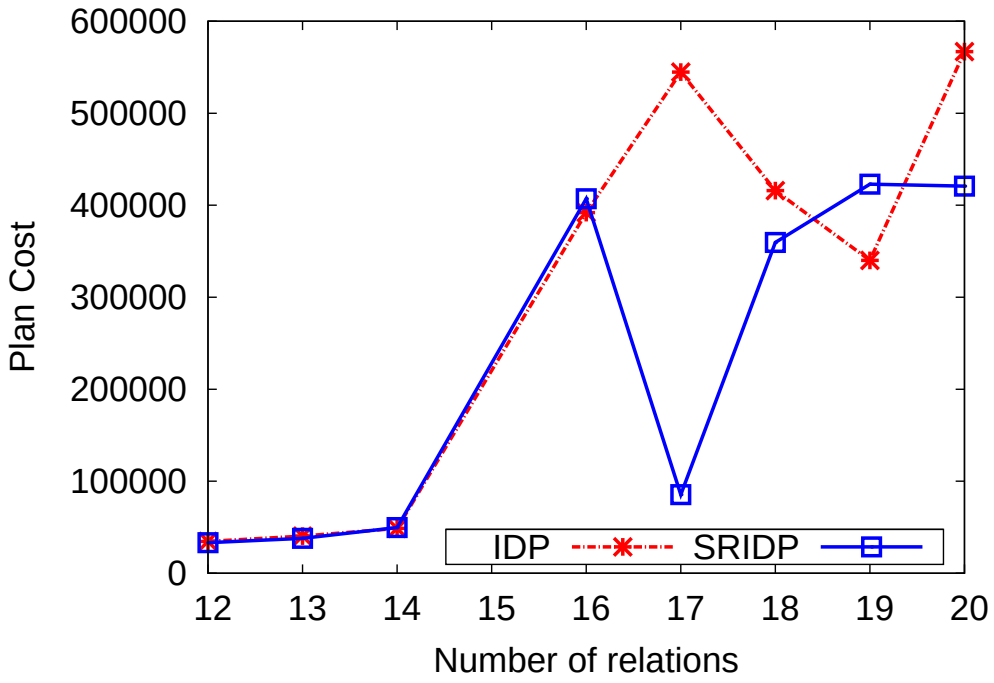

Figure 4.2: Plan cost versus number of relations for medium density.

since that is the only query for which that scheme has finished plan generation. For the subsequent points it ran out of memory. Table 4.3 lists the values of plan costs in detail. Figure 4.7 plots the total running time for a high density query set against the number of relations.

Table 4.3: Costing in Plans for highly dense queries

Number of relations	**Plan cost IDP**	**Plan Cost SRIDP**	**Skyline DP(pruning)**
11	31903.28	12486.45	7729.86
12	36729.46	23866.3	out of memory
13	46183.5	44360.81	out of memory
14	17192.68	17714.26	out of memory
15	326531.96	325158.31	out of memory
16	82425.42	47818.18	out of memory

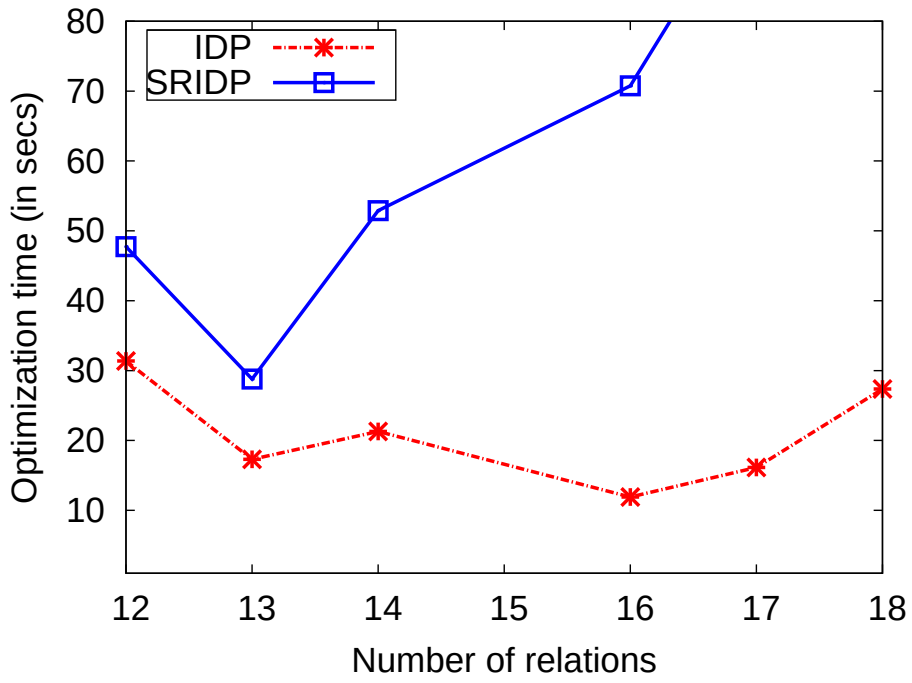

Figure 4.3: Optimization time versus number of relations.

4.2 Experiment 2: Varying density

Figure 4.8 shows the variance in plan cost with the difference in density level. This is for queries whose number of relations is 13 and the remaining default settings do not vary. 13 table queries were chosen to cover a wide range of density levels.

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

4.3 Experiment 3: Varying similarity parameters

The parameters to adjust similarity among subqueries are allowed percentage difference in table size and selectivity. For a 13-table query of default settings, similarity relaxation was varied and the effect it had on plan cost was studied. Figure 4.9 shows the changes in plan cost with respect to variance in similarity parameters.

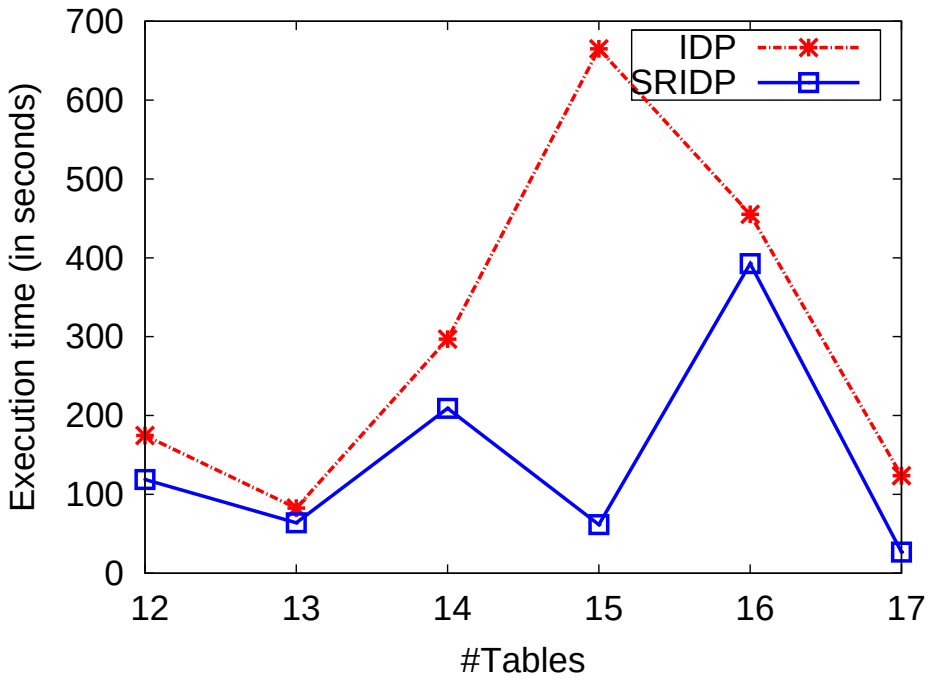

Figure 4.4: Query Execution time versus number of relations.

We noticed that a relaxation of 10% was insufficient for SRIDP to work at a "k" value of 7 as compared to IDP whose maximum reachable "k" was 6. Since it ran out of memory, that point is not plotted. From 20% to 40% relaxation, plan cost increased thereby worsening plan quality due to increased relaxation. At 60% it suddenly peaks to a plan cost level more than that of IDP before dropping down to a static optimal for 70% to 90%.

Figure 4.10 plots the change in plan quality with similarity relaxation for an 18-table query of default settings.

We expected that as relaxation increases, the plan cost becomes higher and higher thereby worsening plan quality. But in all the cases, SRIDP performed better than IDP for the 18-table query. 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

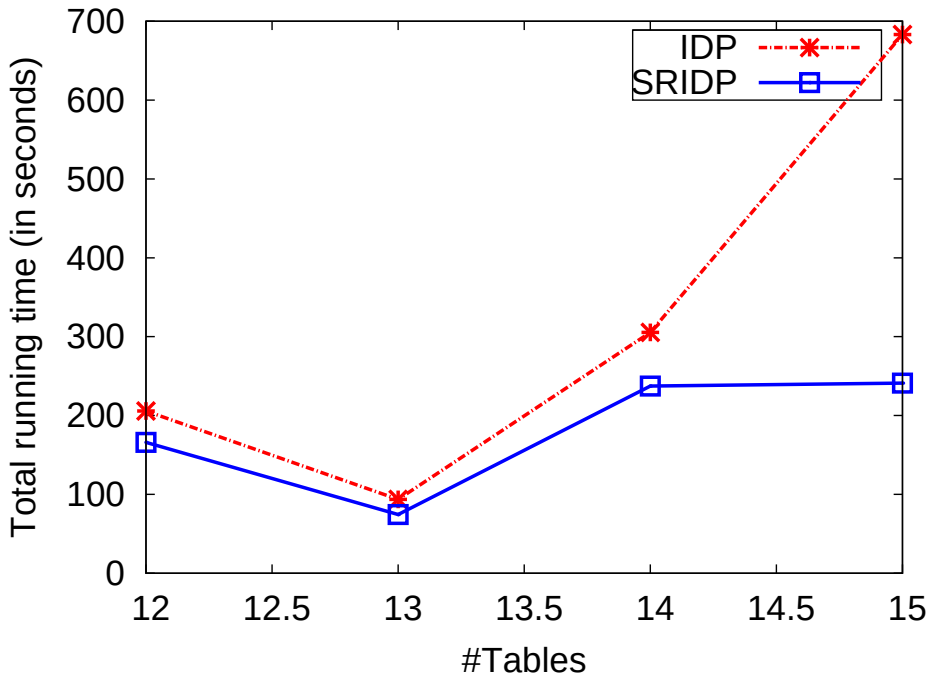

Figure 4.5: Total Query Running time (optimization + execution) versus number of relations.

in the final plan. Also we cannot be sure that copied plans are always worse, they might actually be optimal enough.

4.4 Experiment 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

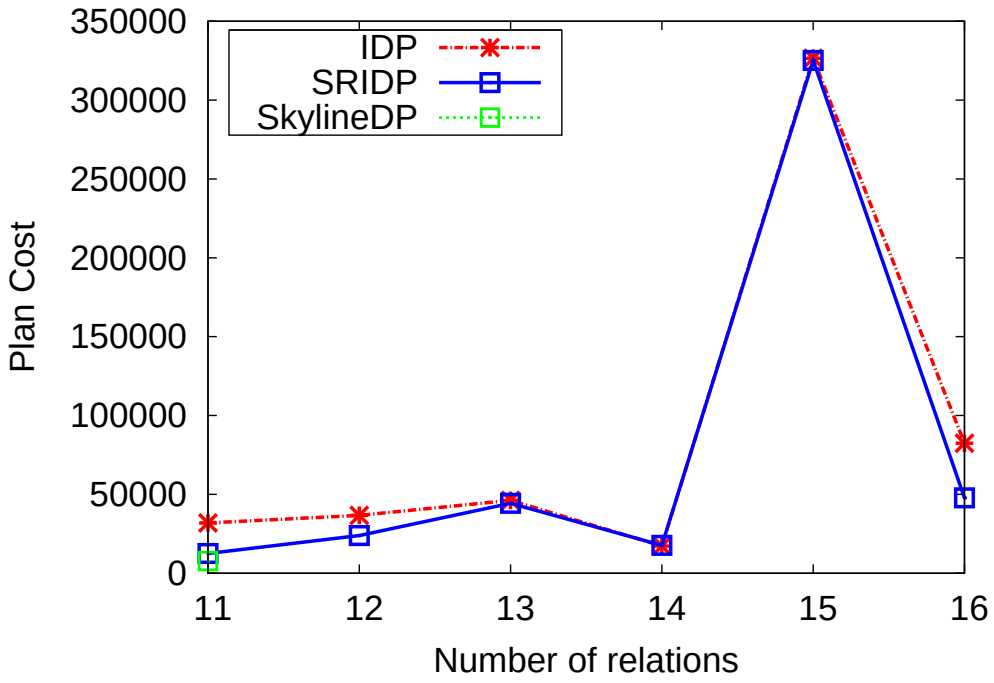

Figure 4.6: Plan cost versus number of relation for high density.

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 4.11 plots plan cost against prune factor while Figure 4.12 depicts optimization time versus prune factor for a 13-table query with density level 1 (highly dense) and similarity relaxation at 70%. SRIDP pushes "k" to 7 as against IDP which can reach only a maximum "k" of 6 for this query. Because the default settings of density level 2 and 30% relaxation show extremely minor (insignificant changes) in optimization time as well as plan quality with variance in prune factor. So we chose a highly dense and a higher similarity relaxation to measure the changes.

In Figure 4.11, we can observe that prune factor may change but the plan cost

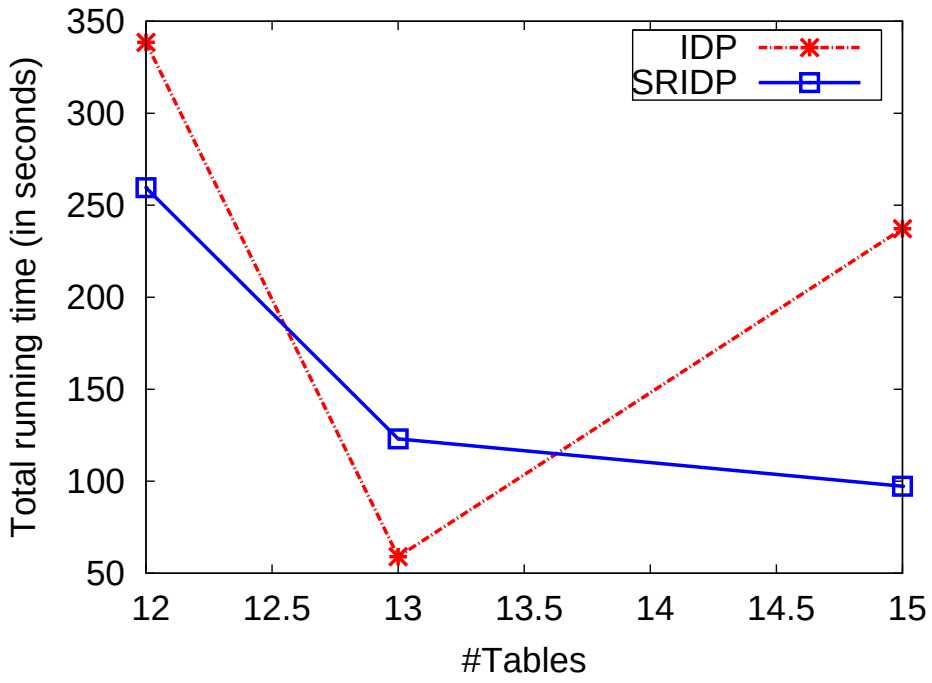

Figure 4.7: Total running time (optimization + execution) versus number of relations for high density.

of SRIDP remains constant. The plan generated by IDP has also been plotted for cost comparison. Whereas in Figure 4.12, 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.

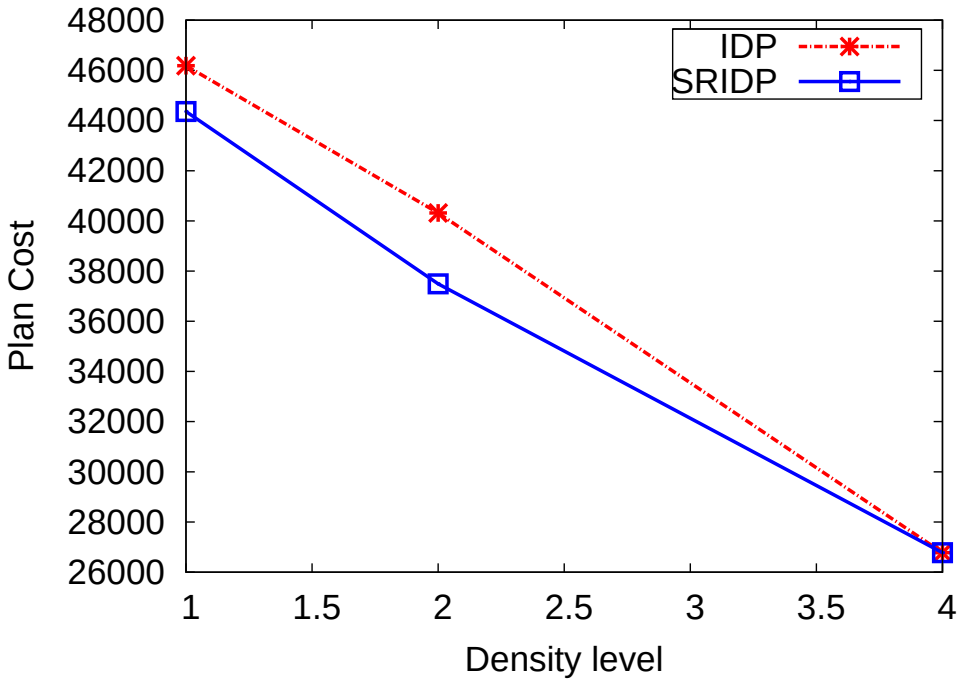

Figure 4.8: Plan cost versus number of relation for various density levels.

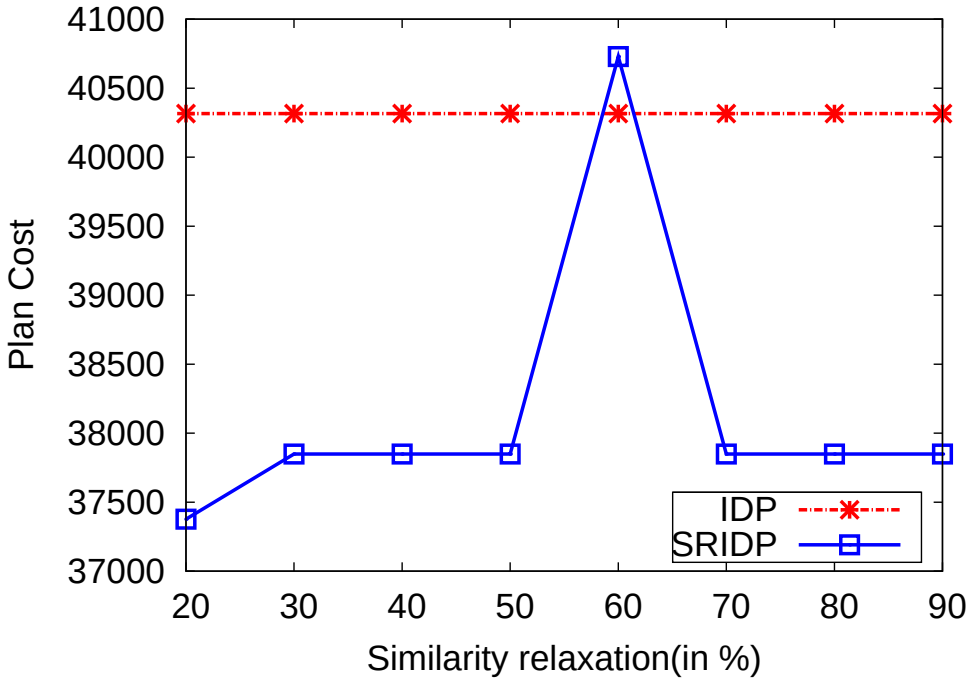

Figure 4.9: Plan cost versus table size and selectivity relaxation in % for a 13-table query.

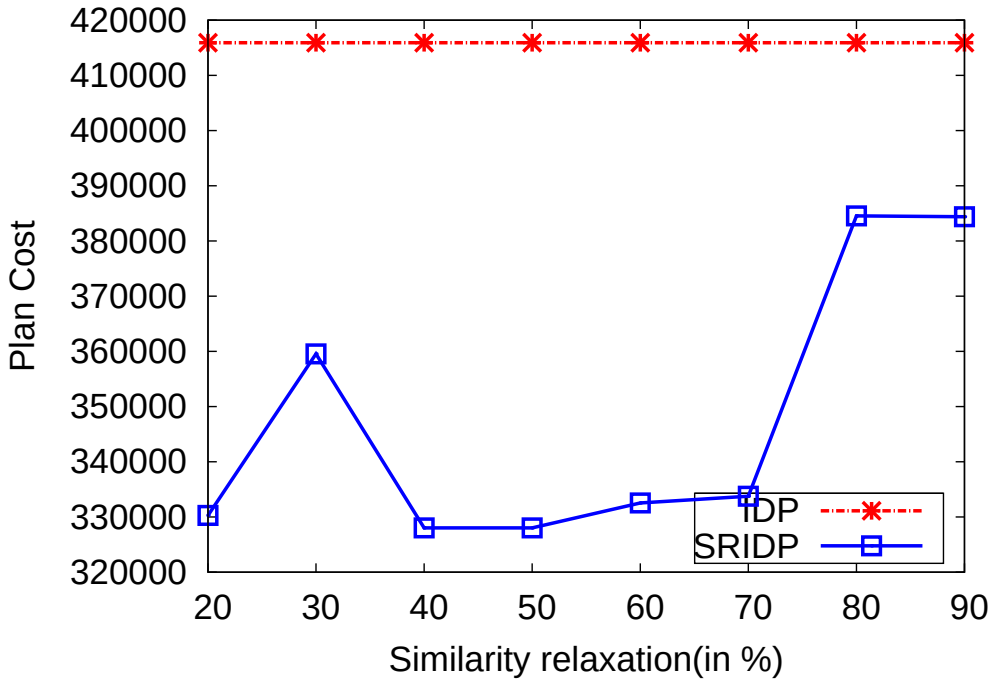

Figure 4.10: Plan cost versus table size and selectivity relaxation in % for an 18-table query.

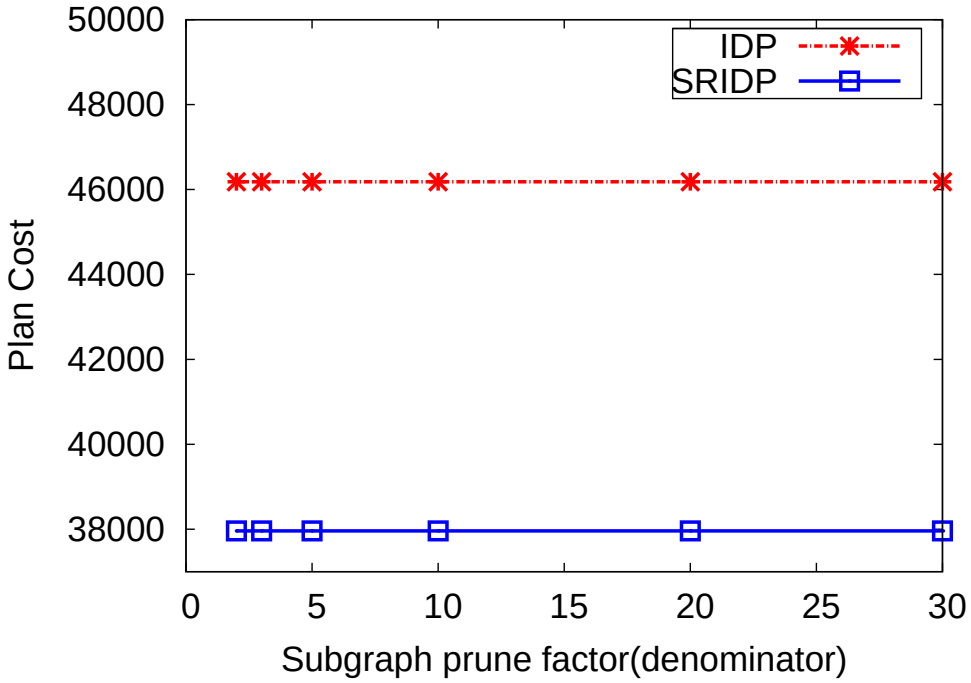

Figure 4.11: Plan cost versus prune factor.

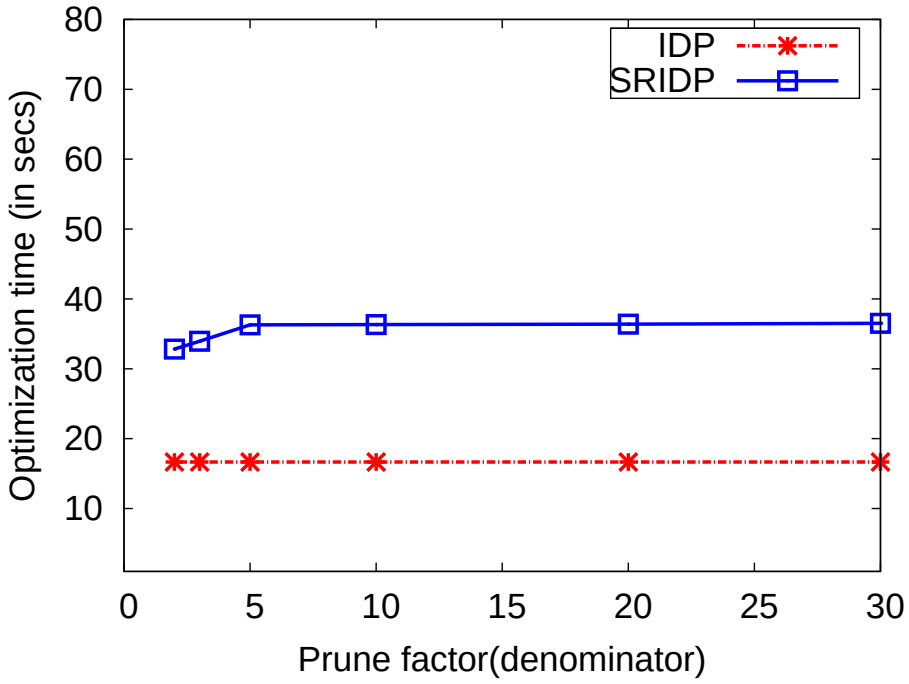

Figure 4.12: optimization time versus prune factor.

CHAPTER 5

CONCLUSION

In our work, we proposed and implemented a memory efficient approach, SRIDP, to generate high quality plans 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 for a particular candidate in the DP lattice. The innovative techniques embodied in SRIDP include:

- The generation of the collection of similar subgraph sets over the entire DP lattice termed as cover set is done as efficiently as possible. Incremental construction of subgraph sets on the fly and deletion of sets which are no longer required has improved the time and memory efficiency of our scheme.

- Plan construction has been done by re-using the query plans among similar subqueries identified by cover set, again by avoiding multiple plan construction for each join candidate and hence making it memory efficient.

Without mere pruning of join candidates to do a straight forward reduction in search space, we resorted to reuse combined with reduction in the number of plans constructed for each join candidate by finding the ideal join orders from similar subqueries which helped in retaining optimality and achieving memory efficiency.

Our results report a consistent increase in the value of "k" 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.

In our future work, we intend to extend our work by multi-threading our optimization algorithm for modern hardware like multi-core architecture and distributing the join candidates ideally among various threads.

BIBLIOGRAPHY

[1] Yu Wang 0014 and Carsten Maple. A novel efficient algorithm for determining maximum common subgraphs. In *International Conference on Information Visualisation*, pages 657–663, 2005.

[2] Jeffrey Baumes, Mark K. Goldberg, Mukkai S. Krishnamoorthy, Malik Magdon-Ismail, and Nathan Preston. Finding communities by clustering a graph into overlapping subgraphs. In *IADIS AC*, pages 97–104, 2005.

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

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

[5] Gopal Chandra Das and Jayant R. Haritsa. Robust heuristics for scalable optimization of complex sql queries. In *ICDE*, pages 1281–1283, 2007.

[6] David DeHaan and Frank Wm. Tompa. Optimal top-down join enumeration. In *SIGMOD Conference*, pages 785–796, 2007.

[7] César A. Galindo-Legaria and Milind Joshi. Orthogonal optimization of subqueries and aggregation. In *SIGMOD Conference*, pages 571–581, 2001.

[8] César A. Galindo-Legaria, Arjan Pellenkoft, and Martin L. Kersten. Fast, randomized join-order selection - why use transformations? In Jorge B. Bocca, Matthias Jarke, and Carlo Zaniolo, editors, *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile*, pages 85–95. Morgan Kaufmann, 1994.

[9] César A. Galindo-Legaria and Arnon Rosenthal. How to extend a conventional optimizer to handle one- and two-sided outerjoin. In *ICDE*, pages 402–409, 1992.

[10] Arianna Gallo, Pauli Miettinen, and Heikki Mannila. Finding subgroups having several descriptions: Algorithms for redescription mining. In *SDM*, pages 334–345, 2008.

[11] Goetz Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2):73–170, 1993.

[12] Jorng-Tzong Horng, Baw-Jhiune Liu, and Cheng-Yan Kao. A genetic algorithm for database query optimization. In *International Conference on Evolutionary Computation*, pages 350–355, 1994.

[13] Yannis E. Ioannidis and Younkyung Cha Kang. Left-deep vs. bushy trees: An analysis of strategy spaces and its implications for query optimization. In *SIGMOD Conference*, pages 168–177, 1991.

[14] Yannis E. Ioannidis and Eugene Wong. Query optimization by simulated annealing. In *SIGMOD Conference*, pages 9–22, 1987.

[15] Matthias Jarke and Jürgen Koch. Query optimization in database systems. *ACM Comput. Surv.*, 16(2):111–152, 1984.

[16] H. Jiang and C. W. Ngo. Image mining using inexact maximal common subgraph of multiple args. In *Int. Conf. on Visual Information Systems*, 2003.

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

[18] Rosana S. G. Lanzelotte, Patrick Valduriez, and Mohamed Zaït. On the effectiveness of optimization search strategies for parallel execution spaces. In *VLDB*, pages 493–504, 1993.

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

[20] Guido Moerkotte. Analysis of two existing and one new dynamic programming algorithm for the generation of optimal bushy join trees without cross products. In *In Proc. 32nd International Conference on Very Large Data Bases*, pages 930–941, 2006.

[21] Guido Moerkotte. Dp-counter analytics. Technical report, 2006.

[22] Guido Moerkotte and Thomas Neumann. Dynamic programming strikes back. In *SIGMOD Conference*, pages 539–552, 2008.

[23] Tadeusz Morzy, Maciej Matysiak, and Silvio Salza. Tabu search optimization of large join queries. In *EDBT*, pages 309–322, 1994.

[24] Kiyoshi Ono and Guy M. Lohman. Measuring the complexity of join enumeration in query optimization. In Dennis McLeod, Ron Sacks-Davis, and Hans-Jörg Schek, editors, *16th International Conference on Very Large Data Bases, August 13-16, 1990, Brisbane, Queensland, Australia, Proceedings*, pages 314–325. Morgan Kaufmann, 1990.

[25] Jun Rao, Bruce G. Lindsay, Guy M. Lohman, Hamid Pirahesh, and David E. Simmen. Using eels, a practical approach to outerjoin and antijoin reordering. In *ICDE*, pages 585–594, 2001.

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

[27] John W. Raymond and Peter Willett 0002. Maximum common subgraph isomorphism algorithms for the matching of chemical structures. *Journal of Computer-Aided Molecular Design*, 16(7):521–533, 2002.

[28] John W. Raymond, Eleanor J. Gardiner, and Peter Willett. Rascal: Calculation of graph similarity using maximum common edge subgraphs. *The Computer Journal*, 45:2002, 2002.

[29] Patricia G. Selinger and Michel E. Adiba. Access path selection in distributed database management systems. In *ICOD*, pages 204–215, 1980.

[30] Timos K. Sellis. Global query optimization. In *SIGMOD Conference*, pages 191–205, 1986.

[31] Leonard D. Shapiro, David Maier, Paul Benninghoff, Keith Billings, Yubo Fan, Kavita Hatwal, Quan Wang, Yu Zhang, Hsiao min Wu, and Bennet

Vance. Exploiting upper and lower bounds in top-down query optimization. In *IDEAS*, pages 20–33, 2001.

[32] Arun N. Swami. Optimization of large join queries: Combining heuristic and combinatorial techniques. In *SIGMOD Conference*, pages 367–376, 1989.

[33] Arun N. Swami and Balakrishna R. Iyer. A polynomial time algorithm for optimizing join queries. In *ICDE*, pages 345–354, 1993.

[34] Akutsu 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.

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

[36] Bennet Vance and David Maier. Rapid bushy join-order optimization with cartesian products. In *In Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 35–46, 1996.

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

[38] Xifeng Yan and Jiawei Han. Closegraph: mining closed frequent graph patterns. In *KDD*, pages 286–295, 2003.

[39] Xifeng Yan, Philip S. Yu, and Jiawei Han. Substructure similarity search in graph databases. In *SIGMOD Conference*, pages 766–777, 2005.

[40] Qiang Zhu, Yingying Tao, and Calisto Zuzarte. Optimizing complex queries based on similarities of subqueries. *Knowl. Inf. Syst.*, 8(3):350–373, 2005.