Robust Heuristics for Scalable Optimization of Complex SQL Queries

A project report submitted in partial fulfilment of the
requirements for the Degree of

Master of Science (Engineering)

in

Internet Science and Engineering

by

Gopal Chandra Das

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

JULY 2006

to my grandfather, parents and family

Acknowledgements

I sincerely express my gratitude to my advisor Prof. Jayant R. Haritsa for his guidance and enduring support. His constant encouragement and confidence in me has helped a great deal in the completion of this effort. I would also like to thank all my friends Akshat, Aslam, Prateem, Sandeep, Debmalya and others who directly or indirectly helped me in completing the project.

Abstract

Modern database systems typically use a dynamic-programming-based search strategy to identify optimal execution plans for SQL queries. However, due to its exhaustive nature, resulting in exponential time and space overheads, this approach does not easily scale to complex queries with a large number of base relations, such as those found in current decision-support and enterprise management applications. To address this problem, a variety of heuristics to prune the search space to a manageable size have been proposed in the literature. However, as we will empirically demonstrate in this paper, even the best heuristics currently available can often result in either extremely poor choices of execution plans, or an inability to sufficiently control the overheads.

Accordingly, we revisit the search-space issue in dynamic programming here, and present a new pruning strategy. The strategy is based on (a) selectively applying pruning to only local segments of the join graph that are expected to be difficult to optimize, and not to the entire join graph; and (b) adopting a skyline-based pruning heuristic on a feature vector that incorporates sub-plan costs, cardinalities and selectivities. Through a detailed study, running to millions of complex queries on rich relational schemas implemented on an industrial-strength database system, the new strategy is shown to always efficiently produce high-quality plans, oftentimes the optimal itself – that is, it is robust unlike its predecessors. Further, this improvement is achieved with comparable or reduced optimization overheads. Finally, the strategy is easily amenable to implementation in current database systems.

Contents

List of Figures

Chapter 1

Introduction

1.1 Introduction

Modern database systems use a *query optimizer* to identify the most efficient strategy, called "query plan", to execute the declarative SQL queries that are submitted by users. A key component of this computationally intensive process is to use a dynamic-programming-based approach to exhaustively enumerate the combinatorially large search space of plan alternatives and, using a cost model, identify the optimal choice. While dynamic programming **(DP)** works very well for moderately complex queries with up to around a dozen base relations, it usually fails to scale beyond this stage in current systems due to its inherent exponential space and time complexity. Therefore, DP becomes practically infeasible for complex queries with a large number of base relations, such as those found in current decision-support and enterprise management applications.

To address the above problem, a variety of approaches have been proposed in the literature. Some completely jettison the DP approach and resort to alternative techniques such as randomized algorithms (e.g. [3, 9]) or genetic techniques (e.g. [6]), whereas others have retained DP by using heuristics to prune the search space to computationally manageable levels. In the latter class, the best strategy currently available is "Iterative Dynamic Programming" **(IDP)** [4, 8] wherein DP is employed bottom-up until it hits its feasibility limit, and then *iteratively* restarted with a *significantly reduced subset* of the execution plans currently under consideration. An experimental study over a variety of queries and datasets demonstrated that by appropriate choice of algorithmic parameters, it was possible to almost always obtain, as per the characterization in [10], "good" (within a factor of twice of the optimal) plans, and in the few remaining cases, mostly "acceptable" (within an order of magnitude of the optimal) plans, and rarely, a "bad" plan.

Robustness of IDP

While IDP is an innovative and powerful approach, we will show in detail in this paper that there are a variety of common query frameworks wherein it can fail to consistently produce good plans, let alone the optimal choice. This is especially so when *star* or *clique* components are present, increasing the complexity of the join graphs. Worse, this shortcoming is exacerbated when the number of relations participating in the query is scaled upwards.

Example. To make the above concrete, consider the 15-relation "Star-Chain" join graph shown in Figure 1.1, where relation R_1 star-joins with relations R_2 through R_{11}, and R_{11} through R_{15} join in a chain formation – this join graph is structurally similar to Queries 8 and 9 of the TPC-H benchmark [11]. A hundred different instances of the Star-Chain join graph (generated by using various combinations of relations for R_1 through R_{15} from a 25-relation schema) were implemented on the PostgreSQL engine [5], and optimized with DP and IDP (for a representative IDP parameter setting of $k = 7$, where k determines the number of DP levels executed in each iteration). [1]

The relative performance results are shown in Table 1.1. Here, the classification of Good (G), Acceptable (A), and Bad (B) plans [10], is refined with the addition of Ideal (I), meaning the recommended plan is either identical to that produced by DP, or within 1% of this optimal. Additionally, the Worst-case (W) plan-cost increase ratio w.r.t. DP is given, and an overall plan-quality factor, ρ, defined as the Geometric Mean of the

[1]The IDP implementation is the best variant [4], described in Section 3.1.

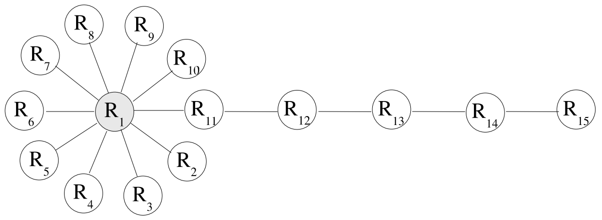

Figure 1.1: Star-Chain Join Graph

plan-costs normalized to the same metric w.r.t. DP, is tabulated.

Query Join Graph	Tech-nique	Plan-Quality					
		I	G	A	B	W	ρ
Star-Chain-15	DP	100	0	0	0	1	1
	IDP	2	44	54	2	10.96	2.83
	SDP	80	20	0	0	1.22	1.02

Table 1.1: Plan Quality (DP, IDP, SDP)

In this table, we see that, relative to DP, for which all plans are Ideal by definition, a sizeable fraction of the plans delivered by IDP are *rather inefficient* – totally, 56% plans are beyond a factor of 2 with regard to the optimal, and 2% are beyond a factor of 10. Further, IDP produces the ideal plan only for a very few (2%) queries. In the worst-case, the IDP plan is about 11 times slower than the optimal plan, and the ρ overall plan-quality metric is close to 3, way above the ideal value of 1.

Skyline Dynamic Programming

We attempt here to address the above problem of consistency in plan quality, by proposing a new pruning strategy for the DP search space. Our heuristic, hereafter referred to as "Skyline Dynamic Programming" (SDP), is based on two novel premises: (a) Selectively applying pruning to only *local* segments of the join graph that are expected to be difficult to optimize, and not to the entire join graph; and, (b) Adopting a multi-way *skyline*-based pruning strategy on a sub-plan feature vector that incorporates *costs*, *cardinalities* and *selectivities*.

Through a detailed study, running to millions of complex queries on rich relational schemas implemented on the PostgreSQL engine, we show that SDP is comparatively very *robust* with regard to consistently providing high-quality plans – in fact, for a large fraction of the queries, it produces *ideal plans*. A quantitative instance is shown in Table 1.1, where SDP gives the ideal plan in 80% or more of the cases for Star-Chain-15, while the remaining sub-optimal choices are all good plans – in fact, very good plans since in the worst-case, the plan selected by SDP is only 22% slower than the optimal. Finally, the ρ value is 1.02, very close to the ideal of 1.

Equally important, SDP's improvement is not achieved at the cost of increasing the optimization time and space overheads – on the contrary, due to its aggressive pruning strategy, SDP is able to complete the optimization process with overheads that are perceptibly lower than that of IDP. This is quantitatively shown in Table 1.2 where the space and time overheads of SDP are at least a third lower than that of IDP. Another point to note is that the overheads of IDP and SDP are an order of magnitude lower than DP, and this is due to the effect of pruning, which is quantified by calibrating the number of plans costed in each strategy, which is also shown in Table 1.2 – the heuristics cost only around 10% of DP's search space.

Query Join Graph	Technique	Memory (in MB)	Time (in sec)	Costing (in plans)
Star-Chain-15	DP	32.39	1.00	8.3E5
	IDP	7.39	0.20	1.3E5
	SDP	4.33	0.10	0.5E5

Table 1.2: Optimization Overheads

To put the above results in perspective, Figure 1.2 shows a plot of the plan-quality ρ against the the optimization overhead, for DP, IDP (for both $k = 4$ and $k = 7$) and SDP. We see here that SDP produces a much better "knee-of-the-tradeoff" between input effort and output quality, as compared to IDP.

Effect of Scaling. When the Star-Chain join graph is scaled up to 23 relations, DP becomes computationally infeasible, due to running out of physical memory. However,

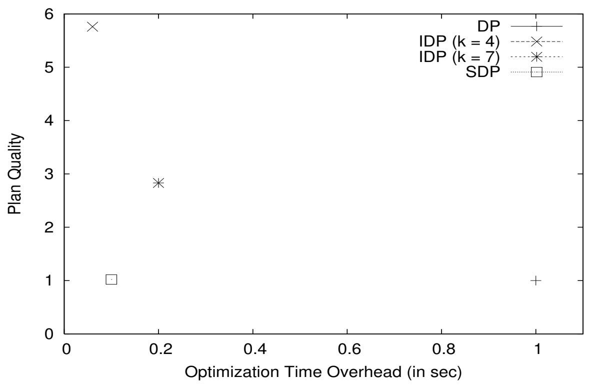

Figure 1.2: Plan Quality (ρ) vs. Effort Tradeoff

both IDP and SDP are able run to completion and in Table 1.3, we show for this query IDP's performance relative to SDP, that is, treating SDP as ideal. We see there that the quality gap between SDP and IDP *increases*, with close to 90% of the IDP plans falling in the bad category relative to SDP. Further, with regard to the overheads, shown in Table 1.4, we see that the differences between SDP and IDP are about an *order of magnitude*.

Query Join Graph	Technique	Plan-Quality					
		I	G	A	B	W	ρ
Star-Chain-23	DP	*	*	*	*	*	*
	IDP	0	0	12	88	25.3	16.2
	SDP	100	0	0	0	1	1

Table 1.3: Scaled Join Graph: Plan Quality

Query Join Graph	Technique	Memory (in MB)	Time (in sec)	Costing (in plans)
Star-Chain-23	DP	*	*	*
	IDP	460.37	54.7	4.5E6
	SDP	55.33	1.08	0.4E6

Table 1.4: Scaled Join Graph: Overheads

In a nutshell, SDP consistently and efficiently produces high-quality query execution plans, as compared to prior pruning approaches. Moreover, like IDP, it can be easily integrated with current optimizers – in fact, as mentioned earlier, all our experiments have been conducted through *direct implementation* on the PostgreSQL engine.

Organization

The remainder of this paper is organized as follows: In Section 2.1, we present our new SDP pruning approach. The experimental framework is described in Section 3.1, and the results are presented in Section 3.2. Finally, in Section 4.1, we summarize our results and outline future research avenues.

Chapter 2

The SDP Algorithm

2.1 The SDP Algorithm

In this section, we present the salient features of the SDP algorithm, closing with the rationale for its design. Broadly, SDP augments the classical DP approach with an additional localized pruning filter that is brought into play in some of the levels of the DP iteration to minimize the subsequent search space.

Definitions. To aid in the presentation, we start with the following definitions:

Hub Relation: Any relation that joins with *three or more* relations in the join graph is defined to be a "hub relation". For example, in Figure 2.1, featuring a nine relation join, the hub relations are **1** and **7**. The identification of hub relations is not restricted only to the original join graph, but is computed *afresh* in each iteration of SDP with the *current* version of the join graph. For example, in Figure 2.1, if after the first iteration, a combination 12 is retained for further expansion, then in the second iteration this combination is treated as a single relation, and it turns out to be a hub relation since it has 3 join edges (to relations 3, 4 and 5).

Where necessary to make a distinction, the hub relations that occur in the original join graph are referred to as *root hubs*, while those that appear in the intermediate levels are referred to as *composite hubs*.

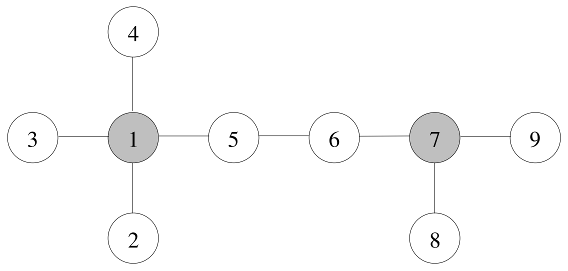

Figure 2.1: Hub Relations in Join Graph

Join Composite Relation: A Join-Composite-Relation (JCR) is defined, similar to [7], as any group of relations that are joined together during the optimization process. For example, 12, 13, 156, etc. are all JCRs in Figure 2.1. Each JCR is associated with a set of plans – the lowest cost plan for producing the JCR and also the incomparable plans that produce "interesting orders" [7] which could be used in subsequent joins or for ordering the query result.

Skyline: Given a set O of objects $O_1, ..., O_n$ characterized by a feature-vector $FV = f_1, ..., f_p$, where each f_i is from an ordered domain, the *skyline* [1] of this set consists of objects that are not *completely dominated* by other objects in the data-set. That is, the skyline is the set of objects O_k such that [1]

$$\nexists O_i \in O \text{ s.t. } \forall j \; f_j(O_i) \leq f_j(O_k)$$

2.1.1 Selective Pruning Locations

A common characteristic of the previous approaches to limiting the DP search space was to apply the pruning universally over the *entire join graph*. However, in practice, only some segments of the join graph turn out to be responsible for high overheads, and by selectively pruning only in such regions, we do not run the risk of unnecessarily and adversely impacting the high-quality of the DP approach on the simpler regions.

Our observation has been that, given a generic query graph, it is the presence of *hub*

[1] Assuming scoring function is for minimization.

relations that are primarily responsible for the high overheads of DP in the optimization process, whereas the remaining segments can be handled relatively efficiently. This observation is quantified in Table 2.1, which shows DP completing optimization (on PostgreSQL) of a 28-relation chain query (where there are no hubs) in less than a second using only about 6 MB of memory, whereas, in marked contrast, a much smaller 16-relation star query requires close to two minutes of processing time and over 300 MB of memory.

Number	Chain		Star	
of Relations	Time (in sec)	Memory (in MB)	Time (in sec)	Memory (in MB)
4	0.0016	0.02	0.0016	0.02
8	0.0050	0.14	0.0200	0.67
12	0.0240	0.45	0.8000	16.15
16	0.0550	1.10	111.1000	326.03
20	0.0900	1.85	–	–
24	0.1700	3.10	–	–
28	0.3300	6.20	–	–

Table 2.1: DP Overheads (Chain and Star)

Based on the above observation, in our SDP approach, we apply pruning selectively *only to JCRs containing hub relations*, leaving the remaining JCRs to be optimized under the aegis of the traditional exhaustive DP.

2.1.2 SDP Iterations

We will explain the operation of the SDP algorithm through the pictorial overview shown in Figure 2.2 for the example join graph of Figure 2.1.

In its first iteration (Level 1), SDP applies the standard DP algorithm, identifying the best access plan for each individual relation. Then, in the second iteration (Level 2), all pair-wise join-composites (excluding, of course, cartesian products) of the base relations are enumerated, as in standard DP. These JCRs are split into two sets: **PruneGroup (PG)** and **FreeGroup (FG)**, with the splitting based on whether or not the JCR includes a complete hub from the immediately previous level – i.e. a "hub-parent". Subsequently the pruning strategy (described later in this section) is applied, and the output is the set of length-2 "survivor JCRs". These survivor JCRs, along with all the survivor JCRs of previous levels (in this case, Level 1), then form the input to DP of Level 3, and the process continues in this manner until we complete Level 7, where the length-7 survivor JCRs have been identified. At this stage, there are only two additional relations to be joined for each composite, which means that, by definition, there cannot be any hub-relations present (recall that a hub relation has to be connected to at least 3 relations). Therefore, we employ the standard DP algorithm in levels 8 and 9. Generalizing the above, given a query graph of N relations, the SDP algorithm utilizes standard DP in Level 1 and Levels $N-2$ and $N-1$, whereas in all the other levels the pruning function comes into play if and only if there is at least one hub available in that level. Further, the input to the DP algorithm in each level is composed of not just the survivor JCRs of the immediately preceding level, but also the survivor JCRs of all prior levels, thereby supporting the identification of *bushy joins*.

2.1.3 Pruning Strategy

The pruning strategy in SDP has two steps: First, the JCRs in the PruneGroup are *partitioned* into sub-groups within which the pruning function is employed. Plausible alternatives for the partitioning are the following:

Parent-Hub Partitioning: Here, the partitioning is based on the "hub-parents" relative to the current level. For example, in Level 3 of Figure 2.2, the partitioning would be based on the hubs **12** and **13**.

Root-Hub Partitioning: Here, the partitioning is based on the root hubs, that is, the hubs of the original join graph in Level 1. For example, in Figure 2.2, the root-hub partitioning will be based on **1** and **7** at all levels.

A point to note here is that with either Parent-Hub or Root-Hub partitioning, if a JCR happens to have *multiple* hub-parents, then it appears in *all* the associated partitions.

For example, in Figure 2.2, the JCR 123 will appear (with Parent-Hub partitioning) in the partitions corresponding to both parent-hubs 12 and 13.

After partitioning, the second step is to apply the function, described next, *within each partition*, to prune a subset of the JCRs present in the partition.

Skyline Pruning Function

In the evaluation of IDP [4], the basic plan evaluation functions that were considered included: (a) *MinCost* (choose subplan with the cheapest result cost); (b) *MinRows* (choose subplan with the fewest result rows); and (c) *MinSel* (choose subplan with the lowest result selectivity). They also mentioned that they evaluated several combinations of these functions, but did not find any combination to perform noticeably better than the best among the basic functions, which turned out to be *MinRows*.

However, our experience has been rather different – as described below, we find that a carefully constructed multi-way function that takes all three parameters (*cost*, *rows* and *selectivity*) into account can provide extremely robust performance as opposed to considering only *MinRows*.

We characterize JCRs with a feature-vector comprised of the following attributes: [Rows(R), Cost(C), Selectivity(S)], corresponding to the number of rows output by the JCR, the <u>lowest</u> cost of producing this output, and the output selectivity of the JCR relative to the product of the sizes of its base relations, respectively. For example, consider the JCR 12345 shown in Figure 2.3, whose feature-vector is FV(12345) = [184736,57726,2.54E-10] since it is estimated by the optimizer to produce 184736 rows at a cost of 57726, with the selectivity being $\frac{184736}{242223*100*200*300*500} = 2.54E - 10$.

We now use the *skyline concept* on the above feature vector for pruning JCRs. Specifically, we compute a *disjunctive multiway skyline* on pairwise combinations of the RCS attributes in the feature vector. That is, we first find the skyline set of JCRs based on their RC values, then the skyline set on the CS values, and finally the skyline set on the RS values. The JCRs featured in the three skylines are unioned, and all remaining JCRs are pruned. That is, we retain only those JCRs that are able to survive in at least *one of* *the three skylines.*

An example of the pruning process is shown in Table 2.2, where from the Prune Group's partition on root hub 1, which consists of JCRs {123,125,135,145,156}, the survivor JCRs are 123, 125, 145 and 156 (Y indicates that the JCR is part of the skyline), while 135 is pruned.

Prune	Feature Vector	*Skylines*		
Group$_1$	[R,C,S]	*RC*	*CS*	*RS*
123	[187638, 49386, 3.9E-5]	Y	Y	-
125	[122879, 52132, 1.0E-5]	Y	Y	Y
135	[242620, 56021, 1.0E-5]	-	-	-
145	[241562, 55388, 6.65-6]	-	-	Y
156	[385375, 52632, 4.5E-6]	-	Y	Y

Table 2.2: Multi-way Skyline Pruning

As mentioned earlier, it is possible that a JCR may contain multiple parent-hubs in which case it is present in all the associated PruneGroup partitions. Given this, it is possible that a given JCR may survive in only some of its parent-hub partitions, but not all – in SDP, such JCRs are *pruned* since they are not universally considered, by all parent-hubs, to be JCRs that are worth pursuing further.

2.1.4 Handling Interesting Orders

It has been well-established that an important factor in choosing an efficient execution plan is accounting for "interesting orders" [7]. This property arises when

1. The query output is explicitly required to be sorted by the user (through Order-By or Group-By statements); or

2. When there are "shared join columns" – that is, relational attributes participating in multiple joins – in the join graph.

The reason for its importance is that costlier sub-plans that happen to provide interesting orders may reduce the cost of later operations as compared to the lowest-cost sub-plan, and should therefore be retained for further processing.

The ordering arising out of shared join columns is beneficial to SDP since such columns result in additional edges being introduced in the join-graph. For example, the presence of $R.a \bowtie S.b$ and $R.a \bowtie T.c$ in the join-graph, where R, S and T are relations with attributes a, b and c, respectively, directly implies $S.b \bowtie T.c$. In most industrial-strength query optimizers, including PostgreSQL, the optimizer rewriter itself performs the inclusion of these additional edges. The presence of the extra edges has the potential to create new hubs, and therefore provides additional opportunity for SDP to determine the choice of survivor JCRs. With respect to user-specified orders, only those that are on join columns are of relevance to our context. To ensure that the skyline pruning does not inadvertently remove JCRs that could subsequently utilize the benefits of such interesting orders, we modify the partitioning function described previously to include additional partitions in the PruneGroup at each level. Specifically, a separate partition is formed for each relation that has an interesting join column, and all JCRs that *do not* contain this relation are included in the partition. This is because we then retain the ability to combine these JCRs, at a later point in the optimization process, with join relations that can produce interesting orders. The skyline pruning function is applied as before, individually to these additional partitions, and the survivors are included in the Survivor JCR group output at that level.

2.1.5 Rationale for SDP Design Choices

We have described above the mechanics of the SDP algorithm. In the remainder of this section, we explain the rationale behind our design choices.

Localized Pruning

The standard DP algorithm considers all meaningful JCRs (i.e. excluding cartesian products), and for each JCR identifies the optimal choices (lowest-cost plan plus interesting order plans). In SDP, our goal is to raise the level of comparison from plans associated with a JCR, *to JCRs themselves.* Specifically, given a hub relation, all "spokes" of the hub are treated as being in the same equivalence class since they have to eventually join with the rest of the relations connected to the hub. Further, as we have already seen, it is the optimization of hub regions in the query graph that are the most complex, and therefore applying pruning to the equivalent-class JCRs in this region will have maximum impact on the optimizer performance.

Note that with SDP, there is no pruning at all for a chain or cycle query, whereas for star and clique queries, there is a strong pruning effect.

Feature Vector

The attributes chosen in our JCR feature vector are [ROWS(R),COST(C),SELECTIVITY(S)], for the following reasons: Firstly, these attributes are already available from virtually all industrial-strength query optimizers, and are also easy to compute. Secondly, and more importantly, Rows is a measure of the *output* of the JCR, whereas Cost is a measure of the *join sequence of the inputs* to the JCR, while Selectivity is a measure of the *transfer function* between the output and inputs. Therefore, it seems natural to use these attributes to characterize a JCR.

Pruning Function

In addition to characterizing a JCR, the three attributes of the feature vector are also related with regard to their *cumulative effect* on query optimization: The Rows attribute is a measure of the *future* impact that the JCR will have in the optimization process, since it forms the input to the next join level, whereas the Cost attribute is a measure of the *current* impact of the JCR with regard to the plan efficiency, and the Selectivity is a measure of the inherent *power* of the JCR, in terms of reducing the intermediate relation cardinalities, a prime objective in query optimization. That is, the three attributes express complementary facets of the optimization process, and ideally we would like to retain JCRs that cheaply produce minimal output on the largest inputs. From the literature, we know that the skyline function [1] provides a natural and elegant mechanism to identify the best among a set of objects described by a feature vector that has attributes over orthogonal ordered domains. We therefore use this concept to ensure that a “complete and precise” set of such JCRs are effectively retained. Fast techniques for computing skyline functions have been presented in several papers (see [2] for a recent survey), and we assume the use of such techniques. With regard to the specific skyline function employed in SDP, two choices are available:

Option 1: Computing a single skyline set on the entire RCS vector; or

Option 2: Computing the (union of) pairwise skyline sets on RC, CS, and RS, respectively.

Our experiments indicated that Option 1 produces high-quality plans but provides very little pruning since most JCRs would survive in this skyline. In contrast, Option 2 produces "the best of both worlds", simultaneously producing high-quality plans along with strong JCR pruning. As a case in point, the number of JCRs processed across all levels for the example query using Options 1 and 2, is shown in Table 2.3, as also their associated plan-quality metrics. We see here that Option 2 provides virtually the same plan-quality as Option 1 but processes only about half the number of JCRs.

Query Join Graph	JCRs Processed	Plan Quality (ρ)
Prune Option 1	1646	1.0148
Prune Option 2	862	1.0151

Table 2.3: Performance of Skyline Options

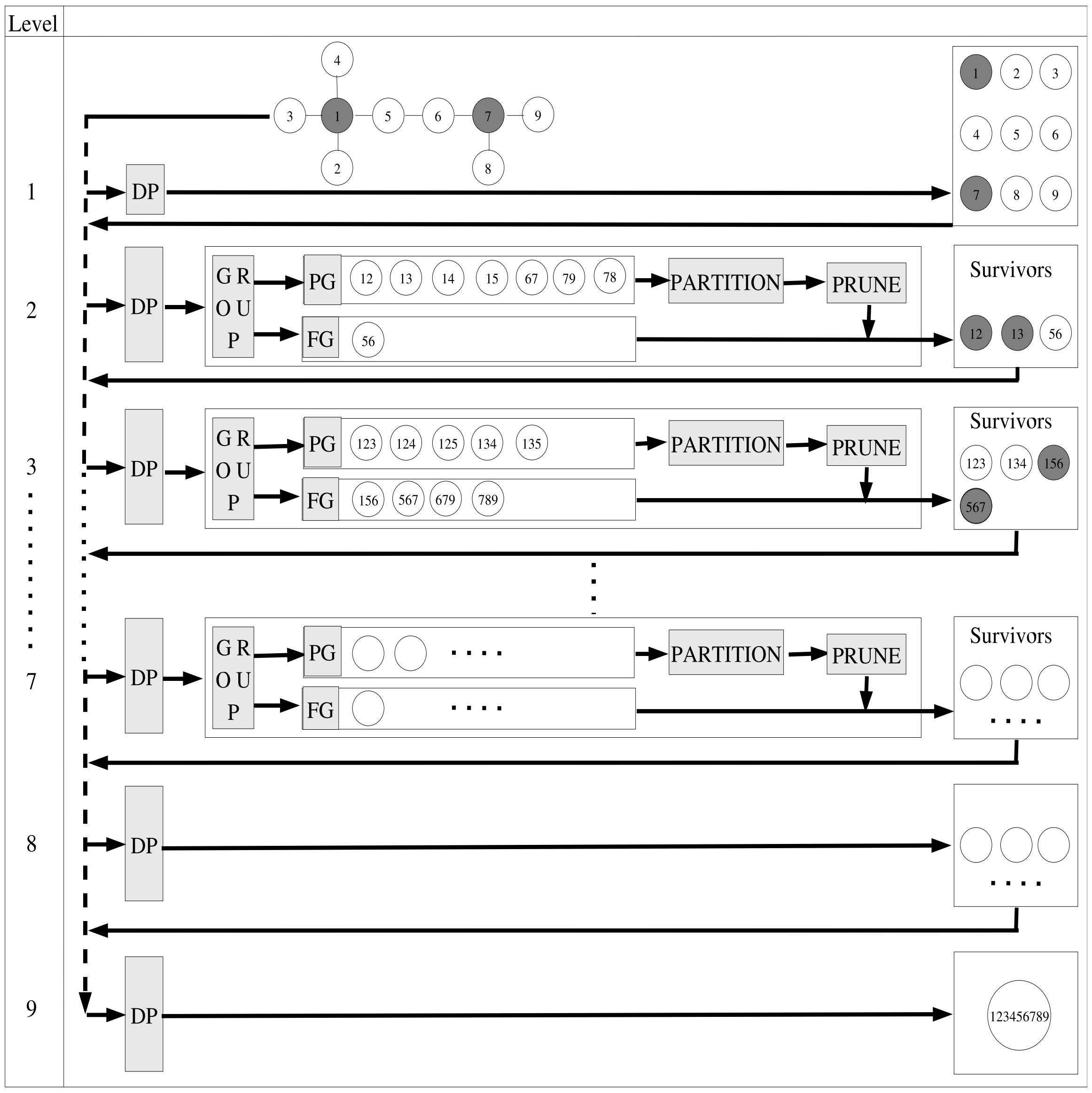

Figure 2.2: SDP Iterations

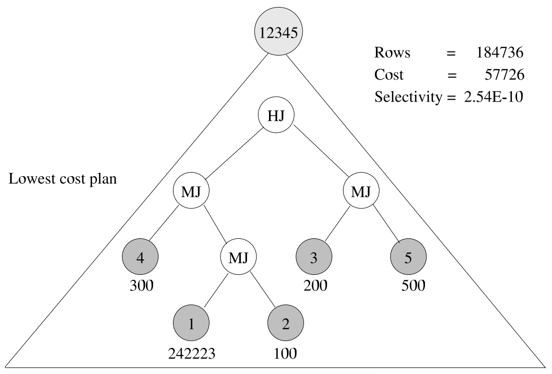

Figure 2.3: Feature-Vector for JCR

Chapter 3

Performance Framework

3.1 Performance Framework

We now move on in this section to describing the experimental framework under which the relative performance of the three alternative approaches, namely, DP, IDP and SDP, was evaluated. The experiments were conducted on vanilla Pentium-IV PCs with 1 GB of memory, 80GB of disk and running the Linux operating system, using the PostgreSQL 8.1.2 engine [5].

Optimizer Algorithms. For DP, we used the implementation that is already available in the PostgreSQL engine, whereas IDP and SDP were implemented by us through modifications to the source-code. The IDP implemented is the one found to be the best overall performer in [4] – specifically, the IDP_1-balanced-bestRow variant, with a hybrid plan evaluation function that selects 5% of the subplans based on *Minimum Intermediate Result*[1] plan evaluation function for ballooning to complete plans, and during ballooning again uses the *Minimum Intermediate Result* plan evaluation function. IDP also has a parameter k that determines the number of levels to which DP is utilized in each iteration, and we have experimented with both $k = 4$ and $k = 7$, which were found to perform reasonably well in [4] – these are referred to as IDP(4) and IDP(7) in the sequel. With

[1]MinRows in our terminology.

regard to SDP, we consider only the Root-Hub partitioning variant, since we found that it provides plan quality close to that of Parent-Hub with much lesser overheads.

Database. The database is approximately 1.5 GB in size, and is composed of twenty-five relations with a geometric distribution (parameter 1.5) of the relational cardinalities, ranging from 100 to 2.5 million rows. Each relation has twenty-four columns, among which a random column has an index built on it. The domain sizes of the columns also have a geometric distribution going from 100 to 2.5 million. With regard to the distribution of data values in these columns, we have experimented with both uniform and skewed (exponential) distributions.

The Analyze command of PostgreSQL was used to generate the database statistics that are used by the optimizer in the plan identification process.

Query Templates. While we experimented with a wide variety of query join graph topologies, for ease of presentation and space reasons, the representative results presented here are with respect to *pure-star* queries and *star-chain* join graphs – our results for the other topologies are similar in flavor.

A very large number of queries, running to several millions, were generated through a combinatorial enumeration of the relational choices – for example, with the 15-relation pure-star query, the hub relation was chosen to be the largest, as in usually the case in data warehousing applications, and ${}^{24}C_{14} \approx 2M$ query instances were created through selection of 14 of the 24 remaining relations, and then each of these queries was optimized with the various feasible optimization strategies.

In the star-component of the queries, the join of the "spoke" relations with the hub relations is on indexed columns, while in the chain-component of the query, each relation in the chain joins on an indexed column with its left neighbor.

Further, for each generated query, we also generated an ordered variant, where the user requests ordered output on a randomly chosen join column.

3.2 Experimental Results

Having described the framework, we now move on to presenting the results of our experiments evaluating the various optimization strategies.

3.2.1 Star Join Graphs

For the star join graphs, the plan-quality statistics are shown in Table 3.1. Here, we see that DP is feasible only for the 15-relation join but not for the 20 and 23-relation versions. For Star-15, we observe that both IDP(7) and IDP(4) have a very large proportion (over 95%) of plans that are more than twice as costly as the optimal plan. In marked contrast, SDP provides the optimal plan in over 50 percent of the queries, and a good plan for all the remaining queries. Overall, SDP has a plan-quality factor, ρ, very close to 1, whereas both IDPs are much beyond this mark (3.23 and 6.47, respectively).

When we scale up the join graph complexity to 20 relations, the performance of both IDPs (relative to SDP as the ideal since DP is not feasible) noticeably worsens, especially IDP(4) which has more than 80% bad plans. Finally, when we go up to 23 relations, IDP(7) itself becomes infeasible, while IDP(4) continues to perform very poorly with respect to SDP.

The above results were on plan quality – now we consider the associated optimization overheads, which are shown in Table 3.2. Here we see that the overheads of SDP are always substantially lower than those of the other algorithms with regard to all three metrics, and that even for a 23-way join, it is able to complete in less than a second using only about 40MB of memory. The reason for these reduced overheads is clear from the number of plans costed, which for SDP is about one-third that of IDP(4) and about 20-30 times smaller than that of IDP(7), testifying to the efficacy of its pruning function.

Maximum Scaleup

We conducted another experiment, with an extended database schema, to determine the *maximum* number of relations in a star join graph that could be handled by the various

Query Join	Tech-	Plan-Quality					
Graph	nique	I	G	A	B	W	ρ
Star-15	DP	100	0	0	0	1	1
	IDP(7)	0	2	98	0	6.00	3.23
	IDP(4)	0	1	96	3	11.35	6.47
	SDP	57	43	0	0	1.17	1.02
Star-20	DP	$*$	$*$	$*$	$*$	$*$	$*$
	IDP(7)	0	0	100	0	9.7	6.10
	IDP(4)	0	0	16	84	18.46	12.19
	SDP	100	0	0	0	1	1
Star-23	DP	$*$	$*$	$*$	$*$	$*$	$*$
	IDP(7)	$*$	$*$	$*$	$*$	$*$	$*$
	IDP(4)	0	8	17	75	31.88	12.93
	SDP	100	0	0	0	1	1

Table 3.1: Star Join Graphs: Plan Quality

optimization algorithms before they exceeded the physical memory of the system. The results for this experiment are shown in Table 3.3, along with the associated optimization times. We see here that SDP is capable of optimizing as large as *45-relation* star join graphs, whereas the other algorithms run out of steam much earlier. Further, even for this massive query, SDP completes the optimization process in *under a minute*, again testifying to the efficacy of its pruning function.

Interesting Orders

The queries considered so far did not have any interesting orders. We now turn our attention to their ordered variants, for which the results are shown in Table 3.4.

We see in this table that qualitatively the performance of the three algorithms remains similar to that seen earlier for pure-star queries, with IDP(4) and IDP(7) having a substantial share of plans that are more than twice slower than the optimal, whereas SDP almost always produces the optimal plan.

Query Join Graph	Technique	Memory (in MB)	Time (in sec)	Costing (in plans)
Star-15	DP	172.94	24.82	3.79E6
	IDP(7)	45.36	1.38	0.61E6
	IDP(4)	10.68	0.21	0.10E6
	SDP	5.44	0.11	0.04E6
Star-20	DP	*	*	*
	IDP(7)	506.14	88.70	4.95E6
	IDP(4)	66.59	1.11	0.49E6
	SDP	22.94	0.41	0.17E6
Star-23	DP	*	*	*
	IDP(7)	*	*	*
	IDP(4)	167.84	2.7	0.99E6
	SDP	43.63	0.84	0.30E6

Table 3.2: Optimization Overheads

Technique	Maximum Scale (# of Relations)	Optimization Time (in sec)
DP	16	111
IDP(7)	21	179
IDP(4)	29	15
SDP	45	51

Table 3.3: Maximum Scalability

3.2.2 Star-Chain Join Graphs

Since the performance of the basic Star-Chain join graphs was already presented in the Introduction, we restrict ourselves to presenting the results of the ordered variants here. These results are shown in Table3.5 and indicate that IDP(7) and IDP(4) have a noticeable fraction of bad plans, and a substantial number of plans that are more than twice slower than the optimal. Further, the percentage of optimal plans is rather low. On the other hand, SDP provides the optimal plan on all but a few queries.

Query Join Graph	Technique	Plan-Quality					
		I	G	A	B	W	ρ
Star-15	DP	100	0	0	0	1	1
	IDP(7)	0	74	26	0	3.55	1.55
	IDP(4)	1	38	61	0	3.37	2.00
	SDP	99	1	0	0	1.01	1.01
Star-20	DP	*	*	*	*	*	*
	IDP(7)	0	56	44	0	3.33	1.70
	IDP(4)	0	58	42	0	2.79	1.74
	SDP	100	0	0	0	1	1
Star-23	DP	*	*	*	*	*	*
	IDP(7)	0	62	38	0	4.67	1.77
	IDP(4)	0	64	36	0	2.95	1.86
	SDP	100	0	0	0	1	1

Table 3.4: Ordered Star: Plan Quality

3.2.3 Need for Localized Pruning

While the previous experiments demonstrated the utility of the skyline-based pruning function, in our final experiment, we demonstrate the need for the other primary feature of SDP, namely, localized pruning. To do this, we compare the performance of SDP with global pruning of the JCRs output from DP, based solely on the skyline functions, to that obtained with local hub-based pruning. A sample set of results are shown in Table 3.6, for the (unordered) Star-Chain-20 join graph, where we see that the plan quality deteriorates perceptibly to around 1.4 from 1.05 with Global pruning, and the worst-case cost ratio jumps to 6 from 1.3.

Query Join Graph	Technique	Plan-Quality					
		I	G	A	B	W	ρ
Star-Chain-15	DP	100	0	0	0	1	1
	IDP(7)	24	26	40	10	40.64	2.33
	IDP(4)	4	33	54	9	41.21	3.01
	SDP	100	0	0	0	1.00	1.00
Star-Chain-20	DP	100	0	0	0	1	1
	IDP(7)	3	67	15	15	726.54	3.14
	IDP(4)	3	32	51	14	728	3.98
	SDP	98	2	0	0	1.17	1.01
Star-Chain-23	DP	*	*	*	*	*	*
	IDP(7)	0	62	38	0	4.67	1.77
	IDP(4)	0	22	78	0	5.05	2.35
	SDP	100	0	0	0	1	1

Table 3.5: Ordered Star-Chain: Plan Quality

Technique	Plan-Quality					
	I	G	A	B	W	ρ
SDP/Global	73	0	27	0	6	1.4
SDP/Local	80	20	0	0	1.3	1.05

Table 3.6: Local vs Global Pruning

Chapter 4

Conclusion

4.1 Conclusions

We have attempted, in this paper, to present a robust and scalable heuristic for the efficient optimization of complex SQL queries that are not easily amenable to the traditional dynamic-programming approach. Our new heuristic, SDP, has two distinctive features with respect to prior proposals for limiting the search space: First, it prunes selectively on the hub regions in the query join graph, since it is these relations that cause optimization complexity, and allows the other simpler regions to retain the full power of DP; Second, it incorporates a feature-vector that includes (intermediate) cardinalities, costs and selectivities, which are the primary factors in query optimization, and a three-way disjunctive skyline pruning function on this feature vector intended to retain all the potentially useful plans, while pruning away the poor strategies.

Experiments over a vast number of queries with a spectrum of join-graph topologies, on rich schemas implemented on the PostgreSQL engine, demonstrated that our approach can perform significantly better than IDP, the best heuristic currently available, with regard to both the robustness of the plan quality and the overheads involved in plan selection. In fact, in all our experiments, we have found SDP to *always provide at least a good plan* (within twice of the DP-optimal), if not the optimal itself. Further, in situations where DP became infeasible, we found that SDP performed close to an order of magnitude better than IDP. Even more attractively, due to its low overheads, SDP could scale up to handling about double the number of join relations in hard-to-optimize star queries as compared to the corresponding limit of the IDP approach. We also showed that even in the presence of interesting orders in the join graph, SDP maintains its good performance profile.

In closing, SDP achieves a substantially improved tradeoff between plan quality and optimization overheads that makes it feasible to efficiently optimize the complex queries with tens of relations that arise in today's industrial-strength data-processing environments. Our future research plans include investigating the impact of using "strong skyline" functions [12] on the optimization process.

Bibliography

[1] S. Borzsonyi, D. Kossmann and K. Stocker. *The Skyline Operator.* Proc. of Intl. Conf. on Data Engineering (ICDE), 2001.

[2] P. Godfrey, R. Shipley and J. Gryz. *Maximal Vector Computation in Large Data Sets.* Proc. of Intl. Conf. on Very Large Data Bases (VLDB), 2005.

[3] Y. E. Ioannidis and Y. Kang. *Randomized algorithms for optimizing large join queries.* Proc. of ACM SIGMOD Intl. Conf. on Management of Data, 1990.

[4] D. Kossmann and K. Stocker. *Iterative dynamic programming: a new class of query optimization algorithms.* ACM Trans. on Database Systems (TODS), 25(1), 2000.

[5] PostgreSQL Database System. *www.postgresql.com.*

[6] Postgres Genetic Optimizer. *www.postgresql.org/docs/7.4/static/geqo-intro2.html*

[7] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie and T. G. Price. *Access Path Selection in a Relational Database Management System.* Proc. of ACM SIGMOD Intl. Conf. on Management of Data, 1997.

[8] E. Shekita and H. Young. *Iterative Dynamic Programming.* IBM Tech. Report, 1998.

[9] M. Steinbrunn, G. Moerkotte and A. Kemper. *Heuristic and Randomized Optimization for the Join Ordering Problem.* Intl. Journal on Very Large Data Bases (VLDB), 1997.

[10] A. Swami. *Distribution of query plan costs for large join queries.* IBM Tech. Report RJ 7908, 1991.

[11] Transaction Processing Performance Council. *http://tpc.org/*.

[12] Z. Zhang, X. Guo, H. Lu, A. K. H. Tung and N. Wang. *Discovering strong skyline points in high dimensional spaces.* Proc. of Intl. Conf. on Information and Knowledge Management (CIKM), 2005.