Query Optimizers: Time to Rethink the Contract?

Surajit Chaudhuri
Microsoft Research
surajitc@microsoft.com

ABSTRACT

Query Optimization is expected to produce good execution plans for complex queries while taking relatively small optimization time. Moreover, it is expected to pick the execution plans with rather limited knowledge of data and without any additional input from the application. We argue that it is worth rethinking this prevalent model of the optimizer. Specifically, we discuss how the optimizer may benefit from leveraging rich usage data and from application input. We conclude with a call to action to further advance query optimization technology.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems – *Query Processing.*

General Terms

Performance, Design, Experimentation

Keywords

Query optimizer, Cardinality Estimation.

1. INTRODUCTION

Query Optimization remains as relevant a problem as ever before, if not more. Modern day database workloads include On-line Transaction Processing Systems, Enterprise Resource Planning, Customer Relationship Management, On-line Analytical Processing and Data Analysis over Data-warehouses. The queries generated by these workloads are increasingly complex and the databases are larger than ever. Thus, the central role of query optimization that searches the space of different execution strategies and picks a good execution plan remains unquestionable.

It has been thirty years since the publication of the System R paper on Query Optimization [1] that acted as the defining framework for query optimization. As we will review in the next section, significant progress has been made on several aspects of Query Optimization since the early days of relational databases. At the same time, certain fundamental difficulties remain. For example, cardinality estimation remains a difficult problem despite years of research activity, search algorithms in optimizers have significant ad-hoc elements to manage optimization time, and cost estimation is not able to take into account the current state of the server effectively.

SIGMOD'09, June 29–July 2, 2009, Providence, Rhode Island, USA.

While there are no easy solutions to these problems, one line of thinking that has not been explored is revisiting the contract with the optimizer. The contract, as defined in [1], is well-intentioned as it imposed the least burden on applications: The optimizer will produce *high-quality execution plans* for all queries while taking *relatively small optimization time* with *limited additional input* such as histograms. But, by virtue of this contract, optimizers are also by design "closed" to additional information that can potentially help lessen the difficulties of the challenges mentioned above.

In this paper, we share our initial thoughts on "opening up" the query optimizer so that each of its core modules (cardinality estimation, cost estimation and search) is capable of leveraging application input and past usage information. Such an approach also impacts how we build these components of the optimizer and we will discuss these consequences. The goal of this paper is to motivate the idea of opening up the optimizer rather than to present specifics of the interfaces we require of the optimizer.

The first part of the paper is retrospective in nature. We begin by sketching the relevant history of query optimizers and we summarize the key technical challenges (Section 2). We then provide a broad overview of our proposal for enabling the optimizer to leverage additional input from the application or usage information (Section 3). The next two sections provide further elaboration of this approach for cardinality estimation and search components of the optimizer (Sections 4 and 5). We conclude with an outline of the a few steps that can help push the frontiers of query optimization (Section 6).

2. The State of the Art in Query Optimization

2.1 A Brief History

The seminal System-R paper [1] provided a framework for query optimization that consisted of three pillars: (a) cardinality estimation for SQL expressions, (b) cost estimation for SQL execution plans (or partial plans) and (c) a dynamic programming based algorithm to search the space of execution plans. However, the paper also recognized that an ordering of tuples, even if locally suboptimal, may pay off globally. This was referred to as *interesting orders*. In many ways, this paper solved the query optimization problem quite well for the simple execution engines and relatively simple queries of that era.

Over the next fifteen years since [1], the query execution engines became far more sophisticated with the addition of new logical and physical operators. Parallel database technology allowed relational systems to handle complex queries over very large data sets. The SQL engines started being used widely during this era for data warehousing and decision support systems. As a result, database systems started experiencing the need to handle more complex queries. Research on query optimization led to explosion of work on query rewrite rules [21]. Some of the rewrite rules, e.g., de-correlation and Magic sets, commuting Group By and join

operators were significant in terms of their potential effect on performance. The query optimization work during this period focused on building extensible query optimizer architectures to enable incorporation of new logical and physical operators as well as transformation rules. Two of the key research projects that addressed these challenges were Starburst [3] and Volcano/Cascades [2]. Around the same time, there was considerable amount of research on different types of histograms [17] that led to improved histogram structures.

After query optimizer architectures stabilized in late 80s and mid-90s, the research in core search architecture, new transformation rules, and new operators have been relatively less compared to earlier years. The key research direction that received attention since then is *leveraging cardinality information from execution.* To the best of our knowledge, we believe that the first use of execution feedback was made in DEC Rdb/VMS system [18]. The application of execution information in that system focused on single table access path selection. The key observation was that a priori selectivity estimation for a Boolean Expression will always be uncertain. However, by running two access path plans in *competition*, if there is an overwhelmingly better plan, then the winner may be identified in a relatively short time. Another line of work used sampling to estimate cardinality of query sub-expression during query optimization [27]. This required sampling and thus accessing data at optimization time. Despite the appeal of accuracy, this technique was not adopted as it violated the expectation that the optimization time must be small. There was also related work on generation of dynamic plans where the choice among them was resolved at runtime [19]. More recent work on using execution feedback for cardinality can be broken down in two categories:

- *Offline:* There are two offline usage scenarios. The first is use of the feedback information from selection queries to build self-tuning histograms over a single table [4][5]. The other is to use the cardinality information from execution of a more general class of multi-relational query expressions by looking up an "execution feedback cache" [6]. The self-tuning histogram approaches are narrower in scope but they fold the execution feedback into the histogram structures thus resulting in no additional overhead during optimization. In contrast, the execution feedback cache based approach is more general but has higher overhead of optimization as it has to look for matches with query expressions from the cache.
- *During Query execution:* One of the early examples of work in this direction is [8][20]. The key idea is to do mid-flight re-optimization with more accurate knowledge of cardinality, e.g., when the query execution reaches a blocking operator at that time the size of the intermediate result size is accurately known. It can then be used to re-optimize the remainder of the query. A degenerate version of such re-optimization was implemented in Teradata. The query processor of Teradata joined only two tables at a time, (with the results going to a spool/pseudo table). Thus, optimization and execution were interleaved as the optimizer needed to decide the next join at every step.

A far more radical version of feedback-driven query execution is Eddies and its variants [9][16] but they require changes to the query execution engine. In this paper, we have focused only on the query optimizer component without requiring significant changes to the query execution engine. But, certainly in a broader context of redesigning database systems, changes to the execution engine can be considered and Eddies and its variants would be relevant in that context. There have also been proposals to create and maintain statistics on derived expressions [28][29]. However, the technical challenges are in many respects similar to that of leveraging execution feedback cache.

2.2 A Critique of Today's Query Optimizers

Optimizers are able to handle amazingly complex queries often with quite satisfactory solutions. Despite their remarkable success, optimizers continue to have several significant challenges. Some of these have been discussed below.

Cardinality Estimation: While single dimensional histograms over a column works satisfactorily by and large, multi-dimensional histograms have not caught on. No multi-dimensional histogram has been shown to be particularly effective given the inherent technical difficulty of finding appropriate bucket boundaries for n-dimensional space. Also, the space of multi-dimensional histograms is very large and the challenge of determining on which combinations of columns multi-dimensional histograms are to be built, is also nontrivial. In contrast, sampling based techniques on single tables are fundamentally more robust as they can support selectivity estimation of arbitrary predicates on single tables but they require execution of queries over the sample at optimization time. Unfortunately, neither histograms nor sampling based scheme provide any simple answer to the difficult problem of estimating cardinality for expressions beyond the single table case. The independence assumption is used for histograms and it leads to very serious propagation of errors [22]. On the other hand, sampling over results of a complex sub-expression requires pre-computing the join [23][24]. Execution feedback based techniques or statistics on query expressions (discussed in Section 2.1) allow us to avoid relying exclusively on optimizer's built-in statistics estimators. However, to realize the benefit of such execution feedback, we need to make much progress on the current state of the art, as will be discussed in Sections 3 and 4.

Cost Estimation: In today's optimizers, the cost model is largely determined by the optimizer designer with possibilities of scaling a few fixed parameters at installation time. The knowledge of current system state information is not exploited while optimizing ad-hoc queries (that are executed immediately upon optimization) [30]. The inaccuracy in cost modeling, coupled with errors in cardinality estimation, also leads to inappropriate tradeoff in the time spent in optimization vs. its execution time, e.g., a long running query is not optimized sufficiently while a large amount of optimization time is spent on a much simpler query.

Search Algorithm: Extensible optimizers are significantly more capable compared to [1]. But, in their quest to enable optimization of complex queries without excessive optimization time, several pragmatic but ad-hoc elements have been introduced in search algorithms of all optimizers. It is fair to say that while optimizers tend to project the image that their search is "exhaustive", in reality none are**.** For example, optimizers offer different "levels" of optimization where higher levels of setting promise more thorough exploration of the search space. However, beyond that description, the application or the user is left with little

understanding on how the optimization level will impact the quality of the plan and the optimization time. While the powerful set of rewrite rules have expanded the search space considered by the optimizers, they have also made the optimizer's logic more complex. Coupled with the lack of accuracy in cardinality and cost estimation for queries, this has had undesirable effect on the quality of the plans chosen by the optimizer. Plan diagrams [25] have revealed that there are surprisingly many optimal plans as the selectivity is varied even when the value of the cost function changes very little. Figure 1 (borrowed from [25]) illustrates that for TPC-H Query 8 there are a large number of distinct optimal plans at different selectivities of the predicates on Lineitem and Supplier relations. Such Plan diagrams seem undesirable as changes in optimal plans are triggered by relatively little changes in the cost. For parameterized queries, it can result in many *plan changes* as the data evolves. Most DBAs view such a phenomenon warily. This is because frequent changes in plans coupled with the inaccuracies in cardinality and cost estimation, can lead to significant variance in the execution times of the parameterized query. The study of plan diagrams also showed that such changes of the optimal plans can potentially be reduced significantly if we settle for slightly suboptimal plans [26]. However, today's optimizers provide us with no way for DBAs to leverage such a tradeoff.

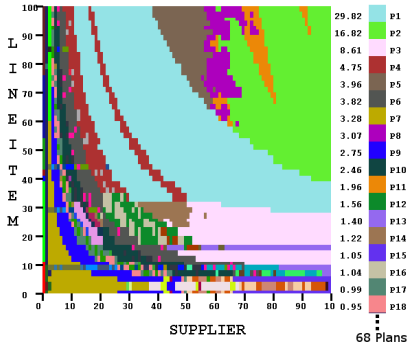

Figure 1: Plan diagram for TPC-H Query 8

To be provocative, one can say that though the optimizers of today's relational databases are able to do surprisingly sophisticated optimization because of the power of transformation rules and their extensible framework; yet they have significant weaknesses that lead to unexpectedly poor selection of execution plans at times.

3. The Case for Rethinking the Contract

The charter of the optimizer is to produce a high-quality execution plan while taking relatively small optimization time with *limited additional input such as histograms*. From the application development perspective, such a contract is convenient as optimizers require no knowledge of the application or data characteristics (beyond histograms). However, as discussed in the last section, there remain fundamental long-standing challenges that don't seem to have any satisfactory solution. Therefore, it may be worthwhile to critically *reexamine the optimizer's contract*.

While the above is a broad issue, we will look at one specific aspect. We suggest that the optimizer is made much more open to *additional information beyond using only histograms that can potentially guide and aid its task*. In the rest of this section, we elaborate this idea in more detail and also outline some of the open challenges that we must solve if we are to leverage the ability of the optimizer to be more open to additional input from the application developer, DBA or the user. We should note that there are already several related initiatives in research and industry. Our proposal should be viewed as a call to accelerate further work in those directions (see also Section 2).

Monitoring and Collection of Usage Information: Compared to database systems a decade ago, today's database systems provide much more capabilities for monitoring the state of the database server and logging such information. For example, in Microsoft SQL Server, dynamic management views can provide snapshots of many facets of the state of the server and such information can be collected in SQL Server's Performance Studio. Oracle, IBM DB2 and other DBMS vendors also provide infrastructure to capture their monitored system state. Such infrastructure allows us to capture SQL workload, their execution plans and observed cardinality of sub-expressions that are part of the final execution plans. However, the above information alone is not rich enough to improve the optimizer's behavior. For example, tracking cardinality of selected query sub-expressions that are not part of the final execution plan [15] can help optimizer improve the plan of a compiled query. Beyond cardinality estimation, another cost model parameter that can significantly affect plan quality is the number of distinct pages of a table accessed using an index lookup. Monitoring this parameter at low overhead during query execution can potentially help correct poor plan choices [33]. Monitoring need not be limited to query execution. There is potential to use monitoring of the query optimization phase to influence the search module. Furthermore, optimizers could also expose information that summarizes its progress when optimization time becomes significant (See Section 5). Understanding the space of low-overhead monitoring that the optimizer can leverage to improve itself deserves more work.

Analysis of Monitored Information: Effective analysis of the data, either during optimization or in background mode, is what makes monitored information valuable. So far, we have made rather shallow use of such analysis. For example, leveraging multi-relation execution feedback for cardinality, as in [6], requires us to use view matching like technology at optimization time which can introduce significant overhead if the feedback cache is large. There is an opportunity to bring to bear statistical inference techniques to the above problem (See Section 4). Oracle's Automatic Tuning Optimizer [31] is an example of an optimization mode that uses execution information to create a SQL Profile for queries. There is a need to expand such work on analysis significantly. This is particularly important as such analysis cannot be done easily by the application developer or DBAs.

Application and Contextual Knowledge: While monitored and usage data can be very valuable, this is not the only source of additional input that the optimizer could leverage. Influencing behavior of the optimizers are of direct interest specifically to performance-sensitive Enterprise Resource Planning (ERP) or Customer Relationship Management (CRM) applications since they have business incentives to optimize performance characteristics of their applications. Often, these applications have a model for their code generation that has implication for characteristics of their queries. Such application knowledge can

be used to guide the optimizer's search space. Section 5 provides more details of such opportunities.

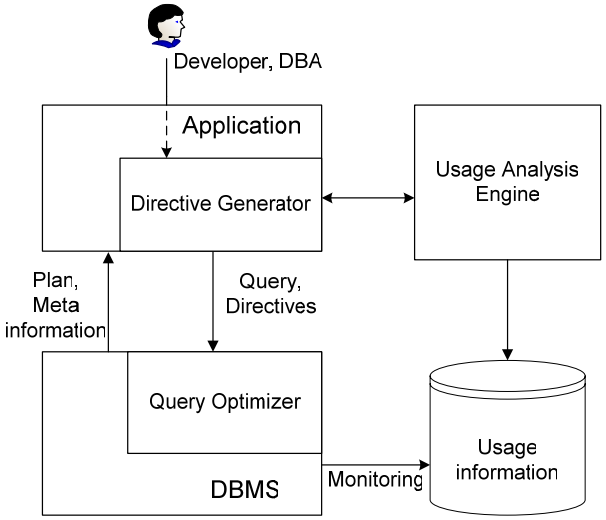

Figure 2: A Core Optimizer that accepts Directives

Need for Injecting Directives: As discussed above, monitored information (appropriately analyzed), application knowledge or a combination of the above two can be sources of additional information for the optimizer. Such information may be related to cardinality or can provide guidance for its search module. Therefore, it is important that the optimizer has appropriate APIs that can be used to influence its behavior using such additional information. In Sections 4 and 5 we discuss the directives related to cardinality estimation and the search module respectively.

A conceptual view of our proposed optimization framework has been captured in Figure 2. Its key characteristics are using monitoring, doing analysis leveraging collected information (in a background mode as well as during optimization) and using APIs for *directives* to influence the behavior of the optimizer. This paper would primarily focus on directives and some limited aspect of the analysis problem.

The directives may target cardinality estimation, cost estimation or the search module. As discussed above, directives can be obtained via analysis of monitored information (by the optimizer itself) or from application or DBA knowledge. One other aspect of the framework that is worth highlighting is that the optimizer does not communicate back only the choice of an execution plan. Rather, we expect the optimizer to provide additional meta-information to the application.

Although the diagram shows the directives to be attached to queries, they can also have broader scope of a session and beyond. We expect this architecture to be of more immediate interest to compiled queries rather than ad-hoc queries although monitored information can also potentially aid the ad-hoc queries.

It may appear that our proposal will increase the cost of building applications by requiring that the applications judiciously pick directives to influence the optimizer. While some sophisticated applications will certainly build custom directive generators that integrate deep application knowledge, the default directive generators may be entirely driven by the optimizer itself through its analysis of monitored information. In such cases, the query directive generation is a component of the query optimizer itself, invoked in a special mode.

Making an optimizer open to additional information has other implications. We need to be thoughtful about what we need to do as a background activity vs. what needs to be done during optimization. Also, we need not assume that the optimizer must have today's simple "query in, execution plan out" model. Rather, for compiled queries (e.g., stored procedures in Microsoft SQL Server) it makes perfect sense to produce an initial plan and then potentially continue additional monitoring to improve the quality of the plan.

The next two sections motivate the use of directives for cardinality and search modules and study the technical challenges for a directive-aware optimizer. Note that such an approach also affects the cost estimation module but we do not include a discussion of that in this paper.

4. Cardinality Estimation Revisited

Our goal is to bring architectural openness to the statistical module, i.e., make the statistics module amenable to richer set of *statistical directives*. Examples of richer directives include:

- *Cardinality Injection:* Explicit sources for cardinality injection can be either from execution feedback [4][6], or from direct application input. Examples of the latter are *validation* predicates that are expected to have a selectivity close to one, e.g., while reporting total sales for winter jackets over the third quarter that is kept in a stored table, a redundant data validation predicate (filter) may be added to the query which checks that the date of the transaction is indeed in the third quarter. In this case, in the absence of a histogram on the transaction date column, we would like the optimizer to assume that the validation predicate has selectivity = 1, instead of using a default magic number to estimate the selectivity.
- *Query-Driven Cardinality Injection:* This represents the class of cardinality information that is obtained by querying at optimization time. All sampling based cardinality estimation techniques fall in this category. For example, cardinality for *Join (S, R)* may be estimated by size of *Join (sample(S), R)* when the foreign key of S is the key of R.
- *Cardinality Constraints:* An application may be aware of correlations in the data that are not explicitly declared or modeled at the database level. The simplest example of a cardinality constraint driven by application input is a key-foreign key constraint specified to help optimizer find a better estimate of join cardinality. Such a constraint might not be declared at a schema level, since enforcing such constraints at the database layer increase the cost of update queries. As another example of application specified input, consider a query containing the predicates (*shipdate* > *@p* AND *receiptdate* > *@p* + *7*). Since the application knows the meaning of these attributes and that about 99% of all the shipments reach within a week of from the *shipdate*, it may issue a directive to ignore the second predicate for selectivity estimation whenever the first predicate is present. This example is reminiscent of semantic query optimization but used narrowly here in the context of cardinality estimation

instead of query equivalence. Cardinality constraints can also be guided by monitored usage information. For example, if p_1 is a predicate on column c_1 and p_2 is a predicate on column c_2, generalization from many instances of execution feedback may lead to simple heuristics such as *selectivity* (p_1 & p_2) = (*selectivity*(p_1) * *selectivity* (p_2))/c_{12}, where c_{12} captures the correlation of columns.

Note that cardinality injection is the simplest of the directives, with an expression on the left hand side and a constant cardinality value on its right. Query-driven cardinality injection is similar but instead of a constant on the right, it specifies a computation that would provide the cardinality information. Finally, cardinality constraints represent the most general case with potentially expressions on both sides. In this paper, we have not attempted to define a language for cardinality constraints. Rather, our objective is to motivate the need and utility for such directives.

It is important to reflect on computational implications of such statistical directives. For example, leveraging an execution feedback cache [6] for cardinality injection can be quite expensive as this may sharply increase the overhead of query expression matching at optimization time. The cost of query-driven cardinality injection depends on the complexity of the query expression and whether part (or all) of the query expression has been pre-computed, e.g., if there is a materialized sample that can be used. The above discussion brings home the point that while statistical directives can greatly improve estimation quality, it is necessary to be mindful of the cost-accuracy tradeoffs. As an extreme example, one can pre-compute statistics on each query expression that the optimizer needs to optimize the query, but this is infeasible. Therefore, statistical directives need to be specified judiciously as they impact performance.

As mentioned in Section 2.2, computing cardinality for expressions for which no cardinality information has been computed explicitly is a known challenge. In traditional histogram based estimators, use of independence and containment assumptions are leveraged to derive such estimation. But, the problem changes as we start leveraging cardinality injection as in [6]. Unlike the traditional framework that uses base table histograms only, now there could be many cardinality expressions whose value may be known via execution feedback. What should be the principle for inferring cardinality of derived expressions? Explicitly matching past cardinality observations (like view matching) to help derive a new cardinality makes minimal use of observed information as it can only answer queries for which an exact match is found among the observations. Recent papers have begun exploring using maximum entropy [5][13] principle to obtain derived cardinality assumption. However, extending such an approach when cardinality is known for complex query expressions (not just selection queries) is not easy. Although a general solution to this problem may not be feasible, exploring the technical problem of inferring cardinality using probabilistic techniques is important (see also [34]). Of course, the problem only gets more difficult for cardinality constraints as they strictly generalize the class of cardinality injection.

Beyond the challenge of estimating cardinality using statistical inference, it is also important to identify what kind of pre-computed structures can be built so that during optimization time the inference of derived expressions is an efficient process. As an example, consider self-tuning histograms. Such histograms integrate past cardinality observations in a histogram structure (in an offline way). This enables the future cardinality expressions to be derived from the histogram during optimization time without any additional overhead for computing cardinality. We need to understand what would be appropriate summary structures in the presence of general statistical directives so that impact on optimization time is reduced. It may be necessary to investigate more general summary structures (e.g., see [32]).

5. Search Framework Revisited

In contrast to statistical directives, providing guidance on search requires a higher degree of sophistication. But, for experienced DBAs, and ERP or CRM applications, *search directives* provide an avenue to constrain the optimizer's search behavior with fine-grained intervention based on the knowledge of the application and data. Given the heuristic nature of search algorithms in an optimizer, search directives are useful in focusing the optimizer's enumeration on a subspace of possible execution plans. Of course, it is very important that the system provides an interactive experience for the application programmer to experiment with appropriate search directives as we expect such directives to be edited and revised before they are finalized.

We distinguish two kinds of search directives. The simpler class of search directives consists of those that constrain the nature of the execution plan chosen by the optimizer. We refer to them as *Execution Plan Directives.* The other class of search directives, called *PlanSpace Directives* makes it possible for the application developer or the database administrator to constrain the way the search algorithm explores the space of potential plans. We discuss each of these now.

Execution Plan Directives*:* Such directives may be identified based on knowledge from past experience, e.g., certain plans or sub-plans that have traditionally worked well for the application. They may also be based on knowledge of the execution environment that the optimizer traditionally doesn't model, such as other queries that run concurrently and their impact on memory contention or contention of physical structures. For example, we may require a query to use a specific index to avoid creating contention with the access plan for another query (that has already been compiled) over the same relation. All commercial optimizers provide the ability to specify optimizer hints that act as execution plan directives, e.g., hints on access methods, join orders, choice of physical operators for joins. However, the languages for such hints have been ad-hoc. Although execution engines differ from one another in the space of operators they support, for a core set of logical and physical operators, we can have a uniform way to represent and reason about execution plan constraints. One possible direction is outlined in [14], which suggests defining hints as *structural patterns* that must be present as a sub-expression in the selected execution plan. Thus, the pattern [(R,S), *, *] indicates that the join among four tables should start with a join between R and S, without having to specify any constraints on other details of the execution plan. Figure 3 shows another example of a tree pattern in the Phints language in [14] that can express a more sophisticated constraint. The query is a join of three tables Customer, Orders and Lineitem. The pattern specifies that there should be an early aggregation on Lineitem using a Stream Aggregate operator, which allows preserving the order of tuples in the Lineitem relation. The constraints in Figure 3 enforces that there should be a Hash Join between Customer and

Orders, where the Customer relation is the build side. Finally, the pattern specifies that the last join should be a Merge Join (e.g. because both inputs will be arriving in sorted order of the join columns). Note also that unlike the join between Customer and Orders, in this case the optimizer is free to choose the ordering for the Merge Join (i.e. the Lineitem relation could be the Outer or Inner).

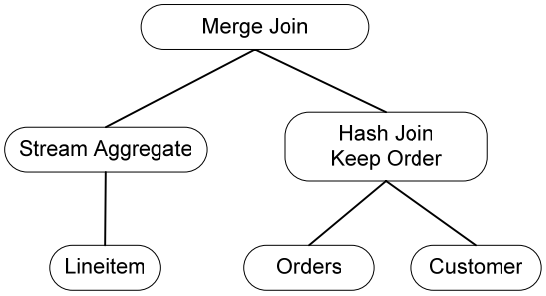

Figure 3: An execution plan constraint for a query

PlanSpace Directives: These directives help influence how the optimizer enumerates the search space. A few motivating examples are given below:

- *Expression Inclusion:* Such a directive ensures that the enumeration algorithm considers the given sub-expression as part of its exploration of alternatives; e.g., expression inclusion for a sub-expression *Join (Sales, Product)* of a query *Q* requires that an optimal execution plan for the sub-expression *Join(Sales, Product)* be generated *as part of enumeration of alternatives for Q*. This ensures that the optimal plan for *Join(Sales, Product)* is at hand when optimization of *Q* is considered. Such a directive may be motivated by the fact that the above sub-plan has been found to have efficient execution and is considered a potentially attractive subplan by the DBA for consideration by the optimizer. Note that unlike Execution Plan directives, an Expression Inclusion directive does not require the final plan to include the subplan mentioned in the directive.
- *Materialized Sub-expressions:* If an expression inclusion directive for an expression E has already been specified, we can further strengthen it by requiring that the search algorithm enumerate a sub-plan that *materializes* the result of the expression E.
- *Initial Plan Specification:* This directive would enable DBA or applications to provide an initial plan for seeding the optimization process. A motivating scenario is plan stability. Consider a case where the current plan for a query is performing adequately. The DBA would like to ensure that in case a recompilation is triggered, the current plan is considered as an alternative by the optimizer as part of its enumeration (if the optimizer determines that the initial plan is still viable).
- *Fixed Plan Set Specification*: This directive constrains the optimizer to pick the lowest cost plan from among a specified set of plans. Observe that in this case the optimizer's search step is considerably simplified – indeed the optimizer needs only to check that the specified plans are valid, and compute their optimizer estimated costs. Such a directive can be useful for parameterized queries when the DBA already knows a set of good plans to choose from. Consider the problem revealed by plan diagrams [25] that there are too many optimal plans as selectivity is varied even when the value of the cost function changes very little (discussed in Section 2.2). Therefore, for a parameterized query, it may be useful to identify first a set of plans through optimizer invocations with different parameters that can approximately have the same effect as that of the "reduced" plan diagrams [26]. Once such plans are identified, the DBA can use the Fixed Plan Set Specification directive to constrain the optimizer to always pick the lowest cost plan among these plans for any instance of that parameterized query.
- *Physical Implementation Preference:* This constraint would enable user/applications to specify trade-off between choice of one physical operator over another, e.g., we can specify that sequential scan on a particular relation is the default access plan unless the gain by using an index seek is 20% or more.

Despite their appeal in providing higher degree of customization for experienced application developers, the key technical challenge in implementing Execution and PlanSpace directives is to make sure that these constraints can be "pushed down" and integrated with the optimizer's enumeration algorithm. This is important not only to leverage the constraints to gain efficiency of enumeration, but also to ensure that we maximize the chances of satisfying the constraints. Unlike statistical directives, search directives also have the added complication that these directives potentially involve physical operators in the engine (e.g., physical implementation) making it hard to standardize such directives across multiple database systems. Therefore, a first step could be to restrict ourselves to search directives that can be specified using elements in the query language (i.e., using logical operators).

Beyond supporting search directives, the optimizer can provide additional information back to the application instead of returning only the final execution plan. Specifically, the optimizer can provide feedback on its *progress of optimization* when optimization time is significant, e.g., for complex decision-support queries. If the optimizer was to provide update on its progress, it can help applications make explicit trade-offs between optimization time and plan quality. To enable such an API, optimizers should have the ability to produce a plan that represents the "current best" execution plan based on its exploration of the space of alternatives so far (perhaps after an initial start-up time). This enables the DBA or the developer to use a more rational basis to make an explicit tradeoff between the optimization time and the quality of plans, e.g., the application may decide to terminate the optimization if there has been a significant reduction from the cost of the initial plan but the average change in the cost of the "current best" plan has been less than 3% over a recent time window. Along with the current best plan, the optimizer can also provide feedback on which of the search directives have been satisfied in the "current best" plan.

The effect of making explicit this concept of progress of optimization also leads to an evaluation framework for the search module of the optimizer. For a given query (with the same statistical input and directives over the same execution engine),

the relative quality of two search strategies may be measured in terms of (a) Time taken to produce an initial plan and its quality and (b) Rate of improvement in the quality of plan with increasing optimization time. Thus, a convenient conceptual framework to evaluate the quality is that of *anytime algorithms* [10]. Such a framework clearly explicates the trade-off between the time to optimize the query and the quality of the plan independent of internal knobs (e.g., optimization level) of optimizers. As an example, consider Figure 4 which contrasts the "ideal" and a "good" search strategy that reach a high quality plan relatively quickly compared to a "bad" search strategy that takes a prolonged time to find a plan with good quality. One impact of adopting such an evaluation framework will be to discourage optimizer designers from adopting less principled features such as "time outs" or "optimizer levels". Such an evaluation framework and specifically the need to support an anytime model along with directives can also impact the choice of the search algorithms deeply. If we are to shift our focus from trying to identify the globally best plan to the paradigm of getting a "current best" plan and improving the selection over time, a fresh look at the search algorithms may be appropriate, e.g., we may revisit randomized combinatorial optimization techniques [11][12].

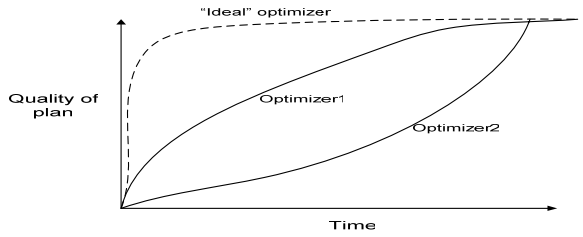

Figure 4: Anytime Framework for Search: Ideal, Good (Optimizer 1) and Bad (Optimizer 2).

6. A Few Next Steps

In order for us to make today's optimizer much more amenable to application and monitored usage information, we see the following as some of the key steps:

1. *Collecting Empirical Information on Optimization:* As discussed in Section 3, we can gather significant information for a database installation on execution as well as the optimization step by wiring the optimization and execution engines appropriately for detailed monitoring. The usage information so obtained can help us understand many facets of optimization such as relative effectiveness of new transformations (or changes in the set of transformations), effect of search space enumeration strategies on search quality (based on time-outs and optimizer levels). Another aspect that is poorly understood at this point is separating the role of inaccurate statistics and search strategy in producing lower quality plans. Overall, these kinds of studies will give us a lot richer information to guide the specifics of redesign of the query optimization architecture.

2. *Developing Analysis Techniques for Usage Information:* This task is directly related to collection of usage information. The raw usage information is not of value unless we can use statistical means to identify patterns. The first step in developing this analytic engine will be to use this to identify statistical directives such as cardinality constraints. There is an opportunity to use statistical machine learning to derive such statistics directives.

3. *Supporting statistical directives and execution plan constraints:* The above two steps provide the necessary infrastructure to help generate directives. The key challenge of course is to be able to leverage the directives for each of optimizer's core modules. Specifically, we need to be able to integrate statistical directives as they can help provide critical input for improving plan quality significantly. Accomplishing this step will need reworking the core of statistics estimation module as discussed in Section 4. After we are able to tackle statistical directives, handling execution plan directives (see Section 5) should be the next goal. Execution plan directives generalize the class of ad-hoc optimizer hints that are supported in today's optimizers.

7. Conclusion

In this article, we briefly reviewed the state of the art in query optimization. We suggested revisiting the contract we have with the optimizer - they should be able to leverage significant additional information from the application developer and usage based analysis such as cardinality and search directives. Ideally, the optimizer should also be able to support an anytime model of query optimization that improves quality of plan as it continues optimization. Finally for compiled parameterized queries, we expect optimizers to improve selection of plans over time rather than be restricted to "query in, plan out" model. There has been work in research as well as in the industry on some facets of the above challenges and we feel work along these directions needs to be further accelerated.

Beyond what we have discussed in this paper, the usage information can be leveraged even more deeply. Just as other software (e.g., Microsoft Windows and Office) aggregate information from large user bases to improve the products, there is a similar opportunity for database vendors to aggregate usage information from its large customer bases to potentially enhance their respective database products.

More than any other time in recent history, database engines and the software platforms today are at the cusp of significant changes. This includes emergence of data analysis infrastructure such as Map-Reduce, Cloud Data Services, Database Appliances and Column-Oriented storage. While all our discussions in this article focused on a traditional SQL query engine, we believe that rethinking query optimization technology can have broad implications for these new platforms as well. In fact, one may argue that the opportunities for rethinking are even more in these emerging platforms as they do not carry legacy of the past.

Acknowledgements

This paper has benefited from input from our colleagues at Microsoft Research and Microsoft SQL Server. We would like to specially thank Vivek Narasayya who gave much critical feedback. Nico Bruno, Christian König, David Maier, Ravi

Ramamurthy, and Anil Nori provided much insightful comments. We also thank David Dewitt, Mike Franklin, Goetz Graefe, Waqar Hasan, James Hamilton, David Lomet, and Jennifer Widom for their thoughtful review.

8. REFERENCES

[1] Selinger, P. et.al.: Access Path Selection in a Relational Database Management System. SIGMOD 1979: 23-34

[2] Graefe, G.: The Cascades Framework for Query Optimization. IEEE Data Eng. Bull. 18(3): 19-29 (1995)

[3] Pirahesh, H., Hellerstein, J.M., Hasan, W.: Extensible/Rule Based Query Rewrite Optimization in Starburst. SIGMOD Conference 1992: 39-48

[4] Aboulnaga, A., Chaudhuri, S.: Self-tuning Histograms: Building Histograms Without Looking at Data. SIGMOD 1999: 181-192

[5] Srivastava, U. et.al. ISOMER: Consistent Histogram Construction Using Query Feedback. ICDE 2006: 39

[6] Stillger, M. et.al.: LEO - DB2's LEarning Optimizer. VLDB 2001: 19-28

[7] Haas, P. J. et al.: Selectivity and Cost Estimation for Joins Based on Random Sampling. JCSS. 52(3): 550-569 (1996)

[8] Kabra, N., DeWitt, D.: Efficient Mid-Query Re-Optimization of Sub-Optimal Query Execution Plans. SIGMOD 1998

[9] Avnur, R., Hellerstein, J. M.: Eddies: Continuously Adaptive Query Processing SIGMOD Conference 2000: 261-272

[10] Zilberstein, S.: Using Anytime Algorithms in Intelligent Systems. AI Magazine 17(3): 73-83 (1996)

[11] Ioannidis, Y.E., Kang, Y. C.: Randomized Algorithms for Optimizing Large Join Queries. SIGMOD Conference 1990

[12] Mayrhofer, R.: Generic Heuristics for Combinatorial Optimization Problems. Proc. of the 9th International Conference on Operational Research 2002

[13] Markl, V. et.al. Consistently Estimating the Selectivity of Conjuncts of Predicates. VLDB 2005: 373-384

[14] Bruno, N., Chaudhuri, S., Ramamurthy, R.: Power Hints for Query Optimization. IEEE ICDE 2009

[15] Chaudhuri, S., Narasayya V.R., Ramamurthy R.: A Pay-As-You-Go framework for Query Execution Feedback. VLDB 2008

[16] Deshpande, A., Ives, Z. G., Raman, V.: Adaptive Query Processing. Foundations and Trends in Databases, 2007

[17] Ioannidis, Y.E.: The History of Histograms (abridged). VLDB 2003: 19-30

[18] Antoshenkov, G. Dynamic Query Optimization in Rdb/VMS, IEEE ICDE 1993

[19] Cole, R.L., Graefe, G.: Optimization of Dynamic Query Evaluation Plans. SIGMOD Conference 1994: 150-160

[20] Urhan, T. Franklin, M.J., Amsaleg, L.: Cost Based Query Scrambling for Initial Delays. SIGMOD Conference 1998

[21] Chaudhuri, S.: An Overview of Query Optimization in Relational Systems. PODS 1998: 34-43

[22] Ioannidis, Y.E., Christodoulakis, S.: On the Propagation of Errors in the Size of Join Results. SIGMOD 1991: 268-277

[23] Acharya, S. et.al. : Join Synopses for Approximate Query Answering. SIGMOD Conference 1999: 275-286

[24] Chaudhuri, S., Motwani, R., Narasayya, V.R.: On Random Sampling over Joins. SIGMOD Conference 1999: 263-274

[25] Reddy, N., Haritsa, J.: Analyzing Plan Diagrams of Database Query Optimizers. VLDB 2005: 1228-1240

[26] Harish D., Darera, P.N., Haritsa, J: On the Production of Anorexic Plan Diagrams. VLDB 2007: 1081-1092

[27] Olken F., Rotem D.: Random Sampling from Database Files: A Survey. SSDBM 1990: 92-111

[28] Galindo-Legaria C., et.al.: Statistics on Views. VLDB 2003

[29] Bruno N., Chaudhuri S.: Exploiting statistics on query expressions for optimization. SIGMOD 2002: 263-274

[30] Lothar F. Mackert, Guy M. Lohman: R* Optimizer Validation and Performance Evaluation for Local Queries. SIGMOD 86

[31] Dageville B. et.al.: Automatic SQL Tuning in Oracle 10g. VLDB 2004

[32] Getoor L., Taskar B., and Koller D.: (2001). Using Probabilistic Models for Selectivity Estimation. SIGMOD 2001

[33] Chaudhuri S., Narasayya V.R., Ramamurthy R.: Diagnosing Estimation Errors in Page Counts Using Execution Feedback. ICDE 2008.

[34] Babcock B., Chaudhuri S.: Towards a Robust Query Optimizer: A Principled and Practical Approach. SIGMOD 2005.