Efficient Mid-Query Re-Optimization of Sub-Optimal Query Execution Plans*

Navin Kabra
Computer Sciences Department
University of Wisconsin, Madison
navin@cs.wisc.edu

David J. DeWitt
Computer Sciences Department
University of Wisconsin, Madison
dewitt@cs.wisc.edu

Abstract

For a number of reasons, even the best query optimizers can very often produce sub-optimal query execution plans, leading to a significant degradation of performance. This is especially true in databases used for complex decision support queries and/or object-relational databases. In this paper, we describe an algorithm that detects sub-optimality of a query execution plan during query execution and attempts to correct the problem. The basic idea is to collect statistics at key points during the execution of a complex query. These statistics are then used to optimize the execution of the query, either by improving the resource allocation for that query, or by changing the execution plan for the remainder of the query. To ensure that this does not significantly slow down the normal execution of a query, the Query Optimizer carefully chooses what statistics to collect, when to collect them, and the circumstances under which to re-optimize the query. We describe an implementation of this algorithm in the Paradise Database System, and we report on performance studies, which indicate that this can result in significant improvements in the performance of complex queries.

1 Introduction

One of the key reasons for the success of relational database technology is the use of declarative languages and query optimization. The user can just specify what data needs to be retrieved and the database takes over the task of finding the most efficient method of retrieving that data. It is the job of the query optimizer to evaluate alternative methods of executing a query, and selecting the cheapest alternative.

Notwithstanding the tremendous success of this approach, query optimization still remains a problem for database systems. Modern database systems are placing an increasingly heavy burden upon their query optimizers. Relational database systems are increasingly being used to execute complex decision support queries. In addition, commercial vendors are all scrambling to add object-relational features to their database systems. Unfortunately optimizer technology has not kept pace with these advances, and a number of the inadequacies of traditional query optimizers have become obvious. Due to the inability of query optimizers to accurately estimate the cost of executing complex query evaluation plans, they often produce sub-optimal plans.

There are a number of reasons why estimating the cost of query execution is difficult. Query optimizers use statistics stored in the system catalogs to estimate sizes and cardinalities of tables that participate in the query. This introduces an error in the estimates either due to the approximations involved, or because statistics are not kept up-to-date. As the number of joins in the query increases, these errors multiply and grow exponentially [9]. Another source of errors is the lack of sufficient information about the run-time system at query optimization time. The amount of available resources (especially memory), the load on the system, and the values of host language variables are things that differ for every execution of the query, and, in some cases, change in the middle of query execution.

The problem is further aggravated in the case of object-relational database systems that allow users to define data-types, methods, and operators. Collection and storage of statistics (for example, histograms) for user-defined data-types (for example, spatial data-types like polygon, point) is an area that has not yet been addressed by the database research community. There are some primitive methods that have been proposed to deal with the estimation of the cost of execution for user defined functions/methods written in an external language (like C++) [23], but these are far from adequate. Similarly, selectivity estimation for predicates involving user-defined methods/functions is another area that is poorly understood. All of this makes it really difficult to properly estimate the cost of executing object-relational queries. Although recent advances in estimation techniques (for example, the histograms of [19] and [11]) and the parameterized/dynamic query evaluation plans of [10, 8, 7] address some of the issues, many problems still remain to be solved.

In this paper, we describe *Dynamic Re-Optimization*, an algorithm that can detect the sub-optimality of a query execution plan while executing the query in order to re-optimize it and improve its performance. During query optimization, the plan produced by the query optimizer is annotated with the various estimates and statistics used by the optimizer. Actual statistics are collected at query execution time. These observed statistics are compared against the estimated statistics and the difference is taken as an indicator of whether the query-execution plan is sub-optimal. The new statistics (much more accurate than the initial optimizer estimates) can now be used to optimize the execution of the remainder of the query.

*This research was supported by NASA under contracts NAGW-3895 and NAGW-4229.

Collection of statistics at run-time can significantly slow down the execution of a query. Further, re-optimizating part of the query and modifying the query execution plan at run-time also incurs overheads. This can actually cause the performance of a query to deteriorate instead of improving. To prevent such problems, we use hints from the optimizer to determine the most strategic places in the query where statistics should be collected, and to determine the conditions under which to re-optimize a query.

Our approach is quite different from the competition model proposed by Antoshenkov [2, 3], the dynamic query plans of [8] and [7], or the parametric query optimization algorithms proposed in [10]. The differences between these algorithms and our approach are further described in Section 4 when we discuss related work.

The remainder of this paper is organized as follows. In Section 2 we describe the details of our algorithm. In Section 3, we describe an implementation of the algorithm in the Paradise database system, and report the results of a performance study that validates our algorithm. In Section 4, we contrast our approach with previous work described in the database literature. Section 5 presents our conclusions and directions of future research.

2 Algorithm Overview

The *Dynamic Re-Optimization* algorithm tries to detect sub-optimality of a query execution plan while the query is being executed. If a query execution plan is believed to be sub-optimal, it dynamically changes the execution plan of the remainder of the query (the part that hasn't been executed yet) leading to an improvement in performance.

These are the salient features of the algorithm:

1. **(Annotated) Query Execution Plans**: We assume that a conventional query optimizer is used to produce a query execution plan for a given query. The only requirement on the plan generated by the query optimizer is that the plan produced by the optimizer should include information about the optimizer's estimates of the sizes of all the intermediate results in the query, and the execution cost/time for each operator in the query. We refer to such a plan as an *annotated query execution plan* in the remainder of this paper.
2. **Runtime Collection of Statistics**: At specific intermediate points in the query, various statistics are collected during query execution. These statistics are used to obtain improved estimates of the sizes of intermediate results and execution costs. These improved estimates can be compared against the optimizer's estimates to detect sub-optimality of the query execution plan.
3. **Dynamic Resource Re-allocation**: The improved estimates are used to improve the allocation of shared resources (like memory) to the various operators of the query, leading to improved performance.
4. **Query Plan Modification**: The improved estimates are also used to determine whether the remainder of the query execution plan would benefit from re-optimization. If so, then the remainder of the query is re-optimized.
5. **Keeping Overheads Low**: Collection of statistics at query execution time can result in a significant overhead if used indiscriminately. To prevent this from happening, at query optimization time, the most effective points to collect statistics are determined, and statistic collection operators are inserted into the query execution plan at only those points.

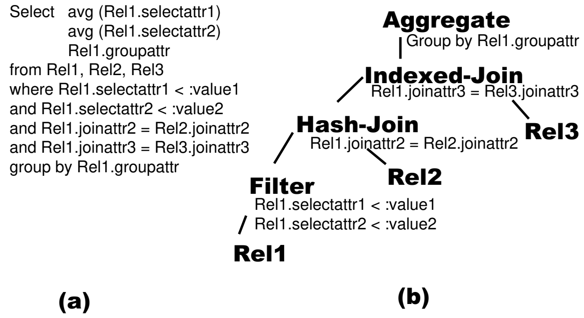

Figure 1: A query and its query execution plan

In the remainder of this section, we describe each of the above items in detail. We end the section with an overview of the whole dynamic re-optimization process, and how it all fits together.

2.1 Query Execution Plans

The job of a query optimizer in a database system is to take as input a query (which is declarative) and produce an execution plan for that query. Figure 1(a) shows an example SQL query. We will use this query as a running example throughout this section for illustrative purposes. Figure 1(b) shows a possible execution plan for this query that might be produced by a query optimizer. An execution plan is essentially a tree in which each node represents some database operator (like *hash-join*, *index-scan*) being applied to its inputs.

During the course of optimization, the query optimizer estimates the sizes of various intermediate results that might be produced, and the cost/time taken by each operator. As part of the *Dynamic Re-Optimization* algorithm, we modify the query optimizer so that these estimates are included in the query evaluation plan that it produces, and are sent to the database execution engine. In the remainder of this paper, we refer to such a plan as an *annotated query execution plan*. The kind of estimates we expect the plan to be annotated with are sizes and cardinalities of intermediate results, selectivities of selection and join predicates, and estimates of the number of groups in case of aggregate operators.

2.2 Run-time Collection of Statistics

In this sub-section, we describe how statistics can be collected at specific points during the execution of a query plan. We describe the kinds of statistics that we can collect, and how this can be done without any I/O overhead. These statistics can then be used to get improved estimates for intermediate result sizes and operator execution costs. In this section, we deal only with the method of collecting the statistics. The question of determining what statistics to collect and at what points in the query execution plan is deferred to a later section.

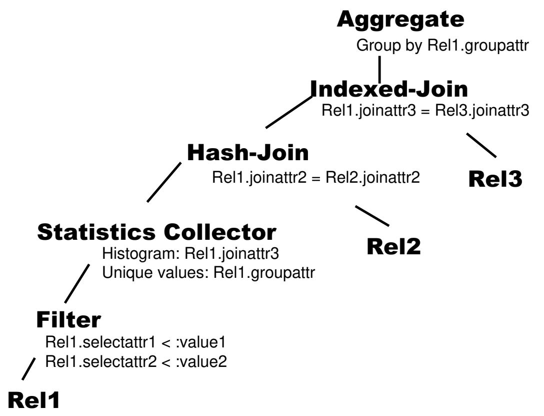

Figure 2: Collection of Statistics at run-time

We now describe how statistics can be collected for an intermediate result of a query without any I/O overhead. Consider Figure 2. There is a *filter* operation that applies selection predicates to the Rel1 relation. Just after the *filter* operation, a *statistics collector* operator is inserted into the query execution plan. As the tuples are being produced by the *filter* operator, they can be examined by a statistics collection routine, and the required statistics can be gathered without interrupting the normal execution of the query. Thus, for example, the cardinality of the result of the *filter* operation can be computed by keeping a running count of the number of tuples that stream past the statistics collection routine, and the average tuple size can be computed by keeping a running average.

There are two limitations of this approach. First, this approach cannot be used to collect any statistics that cannot be gathered in just one pass of the input. This is not a severe limitation because, the statistics that we need to gather for *Dynamic Re-Optimization* can be computed with reasonable accuracy using this approach. To compute cardinality and average tuple-size of a relation, a single pass is obviously enough. Using reservoir sampling [24], histograms can also be computed with reasonable accuracy [19]. The number of unique values of a particular attribute (or a set of attributes) can be computed using the bitmap approach of [6] or reservoir sampling ([24] as described in [20]).

The second limitation of this approach has to do with the pipelining of operators in a query execution. If statistics collection is being done in the middle of a pipelined execution of a row of operators, then none of the operators in the pipeline can benefit from those statistics. This is because all the operators in the pipeline are executing concurrently with the statistics collection routine. Hence, the statistics will not be ready until all the operators in the pipeline have completed a significant portion of their execution[1]. This problem is inherent in our approach, but we will see that, in spite of this limitation, *Dynamic Re-Optimization* performs well in practice.

An alternative to this would be to actually break the pipeline, and force materialization of intermediate results at points where statistics need to be collected. This, however, can significantly slow down the execution of a query, and we consider this to be too high a price to pay.

It should be noted that there is a significant difference between conventional statistics that are computed and stored in system catalogs, and the statistics gathered for the *Dynamic Re-Optimization* algorithm. Conventional statistics need to be rather general in the sense that they are computed once and then used for estimations in various different types of queries. Consider a histogram built on an attribute a of relation R. This same histogram might be used to estimate the selectivity of an equality predicate of the form 'R.a = 10', a range predicate of the form 'R.a between 10 and 100', a join operation such as 'R.a = S.b' and to estimate the number of unique values of R.a (for aggregation). By contrast, histograms constructed for the *Dynamic Re-Optimization* algorithm can be very specific because the exact purpose for which the statistics are being computed is known. This can be exploited to increase the accuracy of the estimates. [19] indicates that different histograms are suited for different purposes. Hence, the type of histogram and method of computation can be adapted to the problem at hand.

After statistics are gathered in this fashion during query execution, they can be used to obtain new estimates for intermediate result sizes and operator execution costs for the remainder of the query. We note that the statistics collected at run-time are actually observed statistics, as opposed to estimates (which the optimizer uses). Further, as described in the previous paragraph, these statistics can be "specific" to the query being executed. Due to this, the new estimates can be a significant improvement over the *optimizer's estimates* that are included in the annotated query execution plan. We refer to these estimates as the *improved estimates* in the remainder of this paper. In the next two sub-sections, we describe exactly how these *improved estimates* can be used to improve the execution of the query.

2.3 Dynamic Resource Re-allocation

In this sub-section, we describe how *improved estimates* can be used to improve the allocation of shared resources to a query, leading to an improvement in performance. We first briefly comment upon resource allocation algorithms, and then discuss how they can benefit from *improved estimates.*

Most of the state-of-the-art algorithms for basic relational operators like sort, join and aggregate require a large amount of main memory to perform well with large datasets. The performance of these algorithms depends critically upon the actual amount of memory allocated. Assuming a workload of complex queries consisting of a number of memory-consuming operations, it is unrealistic to expect that the memory requirements of all the queries can be satisfied. This gives rise to the problem of deciding how to divide available memory among different queries in the system, and different operators in the query.

Memory allocation strategies for complex queries can be classified into two categories. The memory allocation is either decided at query optimization time by the optimizer [22, 4], or it is determined at execution time based upon estimated memory characteristics of the query [14, 26] (or individual operators of the query [15]). In either case,

[1] It should be noted that a blocking operator, like *hash-join*, acts as a natural break in a pipeline, because it consumes all of its first input before producing any tuples of output.

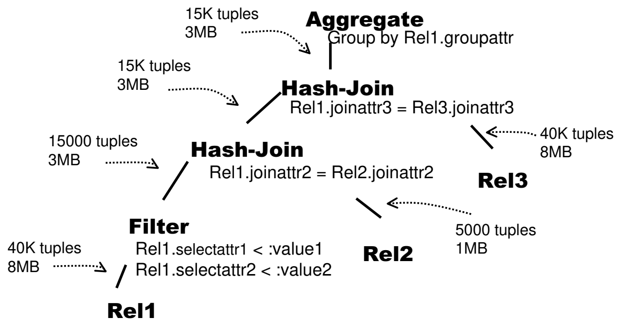

Figure 3: Use of improved statistics to improve memory allocation

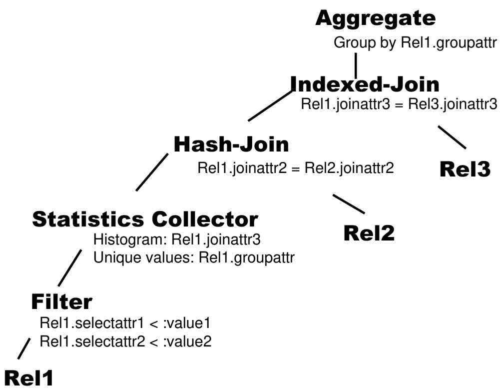

Figure 4: A potentially sub-optimal query plan

these algorithms estimate the memory requirements using statistics, and decide upon an allocation of memory based on the trade-offs involved. Allocating too little memory to a particular query or operation implies that it has to do more I/0 to make up for the lack of memory, and its performance suffers. On the other hand, allocating too much memory results in under-utilization of memory (which could have been better used by another operator), again leading to sub-optimal performance. The discussion of actual algorithms for memory management and allocation is beyond the scope of this paper, but we note that any memory management algorithm that intelligently allocates memory based on estimated memory requirements runs into the same problems that face a conventional query optimizer: *i.e.* inaccurate estimates.

As discussed in the previous sub-section, during the course of query execution, statistics about intermediate query results can be gathered and used to improve upon the estimates of the query optimizer. These improved estimates can be used to improve the allocation of memory to the various operators of the query. Specifically, when improved estimates are available, the memory management module can be re-invoked and supplied with the new estimates. The memory management module uses these new estimates to produce a new memory allocation for the remainder of the query. Overall performance is expected to improve since the new memory allocation is based on improved estimates.

Consider for example the query execution plan in Figure 3. We now describe how this actually works in a specific database system (such as Paradise [17]). In this plan the *filter* operator produces 15000 tuples that require 3MB of memory. Based on this estimate, the maximum memory requirement for each join is estimated at 4.2 MB (size of left input plus overhead), and the minimum requirement is 250KB. Let us assume that at run-time only 8MB of memory is available for this query. In this case the Memory Manager believes that the maximum memory requirement for both joins cannot be satisfied. Hence, it allocates 4.2 MB to the first *hash-join* (its maximum memory requirement), allocates only 250KB to the second *hash-join* (its minimum memory requirement), and allocates the left over memory to the *aggregate* operator. This causes the second *hash-join* to execute in two passes.

If a statistics collector operator is now inserted into the query execution plan just after the *filter* operator, (as shown in Figure 2), the exact number of tuples resulting from the *filter* operation can be observed. Let us assume that the actual number of tuples satisfying the selection predicate is 7500, and not 15000. Now, the maximum memory requirement for the second *hash-join* is re-computed and is found to be 2.05MB. The Memory Manager can satisfy this requirement. Using the new memory allocation, the hash- join of *Rel3* can be completed in one pass, resulting in a significant improvement of performance.

In this paper, we assume that once an operator starts executing, its memory allocation cannot be changed. In other words, improved statistics can only be used to improve the memory allocation for operators that have not begun executing. If, however, the operators in the database system have been implemented in such a manner that they can respond to changes in memory allocation in mid-execution, our algorithm can be extended to take advantage of this.

Throughout this paper, we have concentrated only on dynamically improving the memory allocation for a query. However, similar techniques can be applied to handle the allocation of any shared resources (e.g. processors in an SMP).

2.4 Query Plan Modification

In the previous sub-section, we described a relatively simple change to improve the execution of a query. The allocation of memory to the various operators in the query was modified without actually modifying the query execution plan. While that can result in significant savings in some cases, a much more serious problem with query execution is that the query execution plan itself might be sub-optimal. For example, the join order might be sub-optimal, or the choice of algorithms (*e.g. hash-join vs. indexed nested-loops join*) could be improved. In this case, tremendous savings can be achieved by modifying the query execution plan.

Consider the query execution plan shown in Figure 4. Let us assume that the query optimizer's estimate for the number of tuples resulting from the *filter* operation has a significant error[2] . Since the remainder of the query plan is based on

[2] There are a number of reasons why this can happen even if there are state-of-the-art histograms on the base relation. The histograms might be out-of-date. The *filter* might involve two different correlated attributes of the relation (*e.g.* 'R1.a = 10 and R1.b = 20') and the histograms do not capture the correlation. Or, the selection predicate might have a user-defined function in a external language, in which case there is no way for the database system to estimate the selectivity of the *filter*.

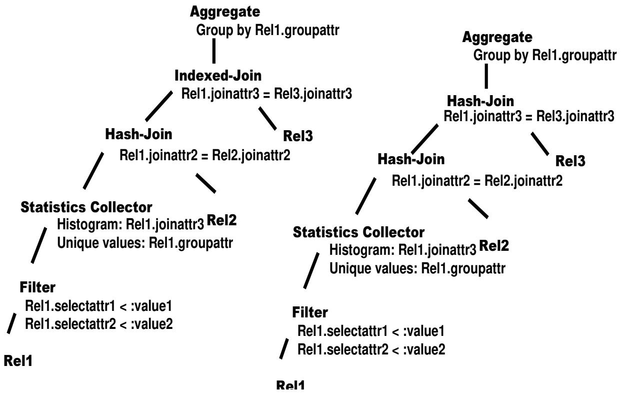

Figure 5: Re-optimization of a plan without discarding any work

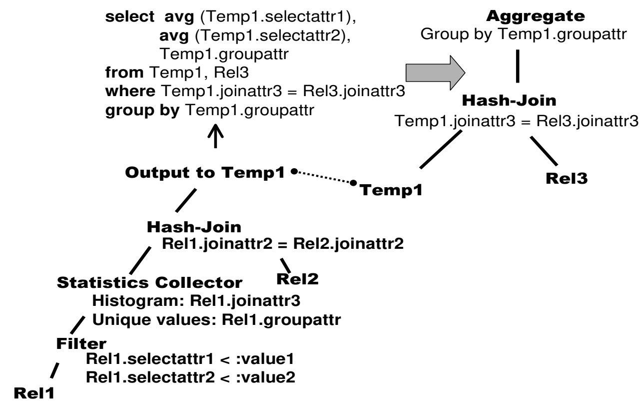

Figure 6: Re-optimization of a plan by materializing intermediate results

this estimate, it is quite possible that the plan might be sub-optimal. At this point it is possible to use the new statistics to re-invoke the query optimizer and generate a better execution plan for the query.

We note that at the time the new statistics for the result of the *filter* become available, the *filter* operation has already completed execution, and the build phase of the *hash-join* algorithm is also complete. However, the probe phase of the *hash-join* has not yet started, and the none of the other operators have even started execution. Under the circumstances, there are three options that the re-optimization algorithm might consider.

The simplest course of action is to completely discard the current execution, generate a completely fresh new execution plan for the query, and execute it from the beginning. This approach has the major disadvantage that it completely discards a significant amount of work that has been already performed and starts out afresh. For this approach to succeed, the combined amount of work done by the new query execution plan and the work that was discarded should be less than the work that would have been done by the previous plan. It is conceivable that this could be the case for some query plans, especially if the sub-optimality is detected early. However, we believe that this approach is too risky, and we do not consider it further in the remainder of this paper.

The second option is to suspend execution of the query, and only re-optimize those parts of the query that have not started executing. In the example above, the *filter* operation is already complete and *hash-join* is also partially done. However, the *indexed nested-loops join* and *aggregate* have not yet begun execution. Hence, a plan involving these two operators can be modified without having to discard any work. Specifically, the query optimizer is re-invoked with new statistics. It is also given the information that the *filter* and the build phase of the *hash-join* is done. The query optimizer then produces a new plan in which the *filter* and the *hash-join* are left as they are, but the remainder of the plan is re-optimized. This situation is pictured in Figure 5.

While we believe that this approach is the best under the circumstances, it does require significant modifications to the query scheduler of the database system to make it work. Specifically it requires the ability to suspend a query in mid-execution, and to modify the query execution plan of the remainder of the query (without the knowledge of the operators that are already halfway through their execution) and to resume execution using the new plan. While this concept is easy to grasp, actually implementing it in a real system can be a problem, especially if the scheduler was not initially designed to handle situations like this.

To tackle this problem, we modified the algorithm slightly to get a new algorithm that is less efficient, but is much easier to implement. Figure 6 shows how this works. In this approach, we do not suspend the execution of the query, but let the currently executing operators (*i.e.* the *hash-join* involving *Rel2*) run to completion. However, instead of piping output to the next operator in the query execution plan, it is re-directed to a temporary file on disk. Now, SQL corresponding to the remainder of the query is generated in terms of this temporary file. This modified query is then re-submitted to the parser/optimizer like a regular query[3].

When to re-optimize: Re-optimizing a query has a significant overhead associated with it. First, there is a non-trivial cost associated with re-parsing and re-optimizing a query. Second, if the re-optimization forces an extra materialization of an intermediate result, the cost of writing and reading that result is incurred. For this reason, re-optimization of a query is not triggered every time the statistics of an intermediate result are observed to be different from the optimizer's estimates. Instead, this decision is made using some heuristics based on the (estimated) costs involved.

Let $T_{cur-plan,optimizer}$ be the optimizer estimate for the time required to execute the current plan. Let $T_{cur-plan,improved}$ be the *improved estimate* for the same. Let $T_{materialize,estimated}$ be the estimated overhead for materializing (writing and reading) the intermediate relation. Let us assume that the optimizer is actually re-invoked and it produces a new plan for the remainder of the query. In this case, let $T_{opt,actual}$ be the time that would be required to re-parse and re-optimize (the remainder of) the query. Let $T_{new-plan,total}$ be the total estimated time for executing the new plan (including the time for work already completed, the time for optimization, the time for materialization, and the time to execute the remainder of the query using the

[3] Of course, care has to be taken to ensure that the new query executes in the same transaction context as the previous one.

new plan).

Obviously, re-optimization should be considered only if $T_{cur-plan,improved} > T_{new-plan,total}$. Unfortunately, is not known until the optimizer is actually re-invoked. Let us, for the moment, assume that $T_{opt,actual}$ is always negligibly small. In that case, the solution is easy. When observed statistics are found to be different from the estimated statistics, the optimizer is invoked to produce a new plan (since this step is considered negligibly cheap) and an estimated $T_{new-plan,total}$. Now, if $T_{new-plan,total} < T_{cur-plan,improved}$, the new plan is accepted and we take the steps required for dynamic modification of the plan (*i.e.* materializing the intermediate result, and then executing the new plan using the materialized result). If, however, this is not the case, then we reject the new plan, and continue execution as before. In this case, no materialization of the intermediate result needs to be done, and the only overhead incurred is the $T_{opt,actual}$ required to getting an estimate for $T_{new-plan,total}$.

Unfortunately, $T_{opt,actual}$ is not always negligibly small and overhead can be significant. We note that it is not too difficult to get a conservative estimate for $T_{opt,actual}$. Let us call this estimate $T_{opt,estimated}$. The time taken to optimize a query does not depend upon the sizes of the datasets involved. Rather, it depends upon the number of operators in the query. Mainly, the cost is dominated by the cost of enumerating the various join orders for the query. Assuming the worst case, a query containing n joins requires the most time for optimization if it is a star-join query [16]. The time taken to optimize a star-join query containing n joins is usually rather stable for a given optimizer and database system. Hence, an optimizer for a particular database system can be calibrated to obtain these estimates.

Now, we use a couple of heuristics to determine whether it is worth spending $T_{opt,estimated}$ time to re-invoke the optimizer. First, we note that re-optimizing is probably not worth the trouble unless the query execution time is much higher than the optimization time. Specifically, we use the heuristic,

$$\frac{T_{opt,estimated}}{T_{cur-plan,improved}} > \theta_1 \qquad (1)$$

In this equation θ_1 is a parameter for the Dynamic Re-Optimization algorithm, and should be a small quantity like 0.05. The optimizer is *not* re-invoked if equation 1 holds.

Another point to be noted is that re-optimization is probably not worthwhile unless there is reason to believe that the current plan might be sub-optimal. To model this, we use the difference between $T_{cur-plan,optimizer}$ and $T_{cur-plan,improved}$ as an indicator of whether the current plan is likely to be sub-optimal. Specifically we re-optimize only if

$$\left| \frac{T_{cur-plan,improved} - T_{cur-plan,optimizer}}{T_{cur-plan,optimizer}} \right| > \theta_2 \qquad (2)$$

In this equation θ_2 is another parameter for the algorithm, and is set at approximately 0.2.

2.5 Keeping overheads low

So far in this section, we have described the *Dynamic Re-Optimization* algorithm, based on the assumption that "statistics" are collected at "key" points during the execution of a query. In this sub-section, we describe exactly what "statistics" are collected, and what are the "key" points in the query.

Obviously, the decision about what statistics to collect needs to be made at query optimization time. After a conventional optimizer has produced a query execution plan, we process this plan and insert statistics-collection operators at various points in the query execution plan. We will refer to this algorithm as the *statistics-collectors insertion algorithm.* This algorithm determines what are the "most effective" statistics to collect, and produces a plan containing the appropriate statistics collection operators. A simple solution would be to measure cardinalities, sizes, and histograms at all intermediate points during the execution of the query. As described in Section 2.2, collection of statistics at query execution time is relatively cheap since there is no I/O overhead. Nevertheless, the CPU overhead itself can be significant in some cases. For the queries that benefit from *Dynamic Re-Optimization*, the savings achieved by re-optimization out-weigh the overheads associated with statistics collection, but for queries that do not get re-optimized, the overhead actually results in an increase in the query execution time.

The *Dynamic Re-Optimization* algorithm is useful for detecting certain kinds of sub-optimalities in complex queries. However, there are a number of queries for which *Dynamic Re-Optimization* does not help. Obviously, if the plan produced for a particular query is already optimal, or close to optimal, re-optimization does not help. Another possibility is that the query might be too simple (for example, consisting of just one join). In this case, even if the query plan produced by the optimizer is sub-optimal, *Dynamic Re-Optimization* is not useful, because by the time collection of statistics is complete, most of the query is also done executing. Thus, even though the new statistics may indicate that the query plan was sub-optimal, it is too late to do anything about it.

The *Dynamic Re-Optimization* algorithm is not targeted towards these queries. However, it is important that their performance does not suffer if the *Dynamic Re-Optimization* algorithm is used. If possible, statistics collection should be entirely avoided for such queries. If not, steps should be taken to ensure that the overhead introduced is kept acceptably low.

Due to these considerations, it becomes important to carefully choose what statistics are collected at query execution time. There is an important trade-off to be considered here. Collecting statistics at too many points in the query can lead to an unacceptably high overhead. On the other hand, if statistics are collected at too few points in the query, some of the sub-optimalities in the query execution plan might not get detected, leading to the loss of some optimization opportunities.

We now describe the *statistics-collectors insertion algorithm* that is used to determine what statistics to collect during query execution. For the remainder of this paper, we assume that the time required for measurement of cardinality and size (in pages) of a table, and the minimum and maximum values for its attributes, is negligible. Hence, we assume that these statistics are measured for all intermediate results in a query. The *statistics-collectors insertion algorithm* will be restricted only to computations of histograms and estimations of number of unique values of a particular attribute (or set of attributes). If, however, in a particular database system, measuring cardinality/size has a significant overhead associated with it, the same techniques can be applied to them as well.

The *statistics-collectors insertion algorithm* starts by making a list of all the potentially useful statistics that can be computed. For a given intermediate table, a histogram on a particular attribute is potentially useful if that attribute is part of a join predicate or a selection predicate later on in the query execution plan. Similarly, computing the number of unique values of an attribute (or set of attributes) is potentially useful if that attribute (or set) is part of a group-by clause of an *aggregate* operation later in the query execution plan. Given this list of potentially useful statistics, we need to determine which ones should be discarded, and which ones computed.

The *maximum acceptable overhead*, μ (specified as a fraction of the total execution time of the query), is an external parameter supplied to the algorithm. Thus, if $T_{cur-plan,optimizer}$ is the optimizer's estimate of the query execution time, then $\mu \times T_{cur-plan,optimizer}$ is the maximum time that can be allocated to the collection of statistics. Now, we need to determine a subset of the potentially useful statistics that take less than $\mu \times T_{cur-plan,optimizer}$ time to compute, and which are "most effective" in detecting the sub-optimality of a plan. To be able to do this, we need to estimate two things. First, we need to estimate the cost of computing each of the statistics. This can be easily estimated using the optimizer's estimates of the sizes of intermediate results. Second, we need some measure of the "effectiveness" of a particular statistic in detecting sub-optimality of a plan.

Two key factors are considered while deciding the effectiveness of statistics in detecting sub-optimality of a query execution plan. The first factor is the probability that the corresponding optimizer estimates are inaccurate. If there is a high probability that the optimizer's estimates are accurate, then there is not much reason to actually observe the statistics at run-time. The second factor is the fraction of the query execution plan that is affected by that particular statistic. The larger the fraction of the query that might be affected by a statistic, the more effective is the statistic at detecting sub-optimality of a plan.

The first question that we ask is what are the chances that the optimizer's estimates corresponding to that attribute are inaccurate? For example, if there is an equality selection on a particular attribute of a base table, and there exists a serial histogram on that attribute, then chances are very high that the optimizer's estimates for the result of the selection operator are very accurate [19]. On the other hand, if there is neither a histogram nor an index on that attribute, chances are very high that the optimizer's estimates are rather inaccurate. In that case, computing a histogram on the result at run-time is likely to be very useful.

The *statistics-collectors insertion algorithm* assigns an *inaccuracy potential* level of *low*, *medium* or *high* to the various *optimizer estimates* in a query execution plan using the following rules. An *inaccuracy potential* of *high* for a particular statistic indicates that there is a high possibility of the corresponding optimizer estimate being inaccurate. We first assign *inaccuracy potential* levels to the statistics on base tables found in catalogs. Then the *inaccuracy potential* levels are propagated upwards in the query execution plan. The following are a set of rules for determining the *inaccuracy potentials*:

- The *inaccuracy potential* for a histogram on an attribute of a base table is *low* if it has a serial histogram, *medium* for equi-width and equi-depth histograms, and *high* if there is no histogram.
- If the system catalogs contain estimates for the number of unique values of a particular attribute of a base table, the inaccuracy potential for this estimate is *low*. The inaccuracy potential for the number of unique values of an attribute (or set) at any intermediate point in a query execution plan is always *high*. (In other words, the inaccuracy potential for number of unique values is *low* only for attributes in a base table, and is *high* in all other cases.)
- Some database systems have information available about the update activity on a table since the last time statistics were updated. In this case, the inaccuracy potential level for all statistics is increased one level if there has been significant update activity since the last time statistics were collected.
- The *inaccuracy potential* for the output of a selection operator involving a simple predicate is the same as the *inaccuracy potential* of its input. In other words, *inaccuracy potential* is *low* if there exists a serial histogram on the input, *medium* for equi-width and equi-depth histograms and *high* when there are no histograms.
- If a selection predicate involves two or more attributes of the relation, then *inaccuracy potential* of the output is one level higher than the *inaccuracy potential* of the input. In other words, if the inaccuracy potential for input is *low*, then *inaccuracy potential* for output is *medium*, and if the input is *medium* or *high*, *inaccuracy potential* is *high*. This increase in *inaccuracy potential* is due to the possibility of correlations that are not captured by the histograms.
- If a selection predicate involves user-defined methods, the *inaccuracy potential* of output is always *high*.
- Consider an equi-join where the join attributes are keys for the corresponding tables. In this case, the output can be estimated rather accurately if the input is known. Due to this, the *inaccuracy potential* for the output of an equi-join on key attributes is the same as the maximum *inaccuracy potential* of its inputs. If the equi-join is on a non-key attribute, then the *inaccuracy potential* is one level higher than the *inaccuracy potential* of its inputs.
- The *inaccuracy potential* for non-equi-joins is always *high*.
- The *inaccuracy potential* for the output of an aggregate operator is the same as the *inaccuracy potential* with which the number of unique values for the grouping columns is known in the input.

The other factor in determining effectiveness of computing a particular statistic is the fraction of a query execution plan that is affected by that statistic and has not yet executed. Consider Figure 7. This figure shows two statistics being collected at query execution time. One is a histogram on

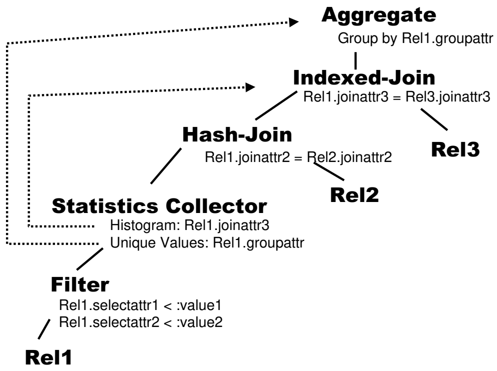

Figure 7: Fraction of a query affected by statistics

the attribute *Rel1.joinattr3* in the output of the *filter* operation, and the other is the number of unique values of of the *Rel1.groupattr* attribute. Now the *joinattr3* attribute is part of the join predicate in the *indexed nested-loops join* pictured in the figure, and hence the corresponding histogram is useful in estimating the cost of that join and the size of its output. Hence, the portion of the query execution plan affected by the histogram on the *joinattr3* attribute consists of all the operators after that join. On the other hand, the number of unique values for the *groupattr* attribute is only useful for the *aggregate* operation. Hence the portion of the query execution plan affected by this statistic consists only of the *aggregate* operation.

Now the relative effectiveness of two different statistics is compared as follows. If one statistic has a higher *inaccuracy potential*, then that statistic is considered to be more effective in detecting sub-optimality of a plan. If the *inaccuracy potentials* for two statistics are the same, then the statistic that affects a larger portion of the query execution plan is considered more effective. Using these rules, the list of all potentially useful statistics is ordered according to increasing effectiveness. Now, we begin deleting the least effective statistics from this list one by one until the total estimated time for computing all the statistics drops below the maximum acceptable overhead ($T_{cur-plan,optimizer}$).

2.6 Summary

To summarize, this is how the entire *Dynamic Re-Optimization* algorithm works. First a conventional optimizer is used to generate a conventional query execution plan for a query. Then the *statistics-collectors insertion algorithm* is invoked to insert statistics-collection operators into the query execution plan. The *statistics-collectors insertion algorithm* ensures that the statistics-collection operators inserted into the query plan do not slow down the query by more than a fraction μ. The output of the *statistics-collectors insertion algorithm* is the final static plan for the query that can be stored in the database system. We note that this plan contains all the optimizer's estimates for the sizes of various intermediate results and the execution times for the operators in the query.

At query execution time, the statistics-collector operators that have been inserted into the query gather statistics on the intermediate results of the query execution. These statistics are then used to obtain improved estimates for the execution times for the remaining operators of the query. These estimates are compared with the optimizer estimates that are stored as a part of the query plan. If the estimates are significantly different, and the query is expensive enough to warrant re-optimization, then the query optimizer is re-invoked to obtain a new plan for the remainder of the query. If the estimated total execution time for the new plan (including overhead of re-optimization and materialization of intermediate results) is less than the estimated execution time of the old plan, then the execution plan for the remainder of the query is replaced with the new plan.

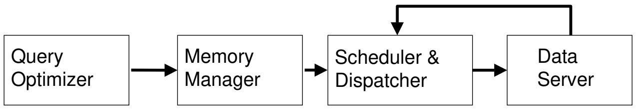

Figure 8: Query Execution in Paradise

3 Implementation and Performance

As an experimental validation of the *Dynamic Re-Optimization* algorithm we implemented it in the Paradise database system [17]. In this section, we report some of the results of our experiments. First we describe the details of the actual implementation of the *Dynamic Re-Optimization* algorithm in the context of Paradise, and its interactions with the memory management module of Paradise. Then we study the performance of the *Dynamic Re-Optimization* algorithm using datasets and queries based upon the TPCD benchmark specification [21]. We also performed some experiments using skewed datasets to measure the effect of skew on performance.

3.1 Implementation in Paradise

Paradise is a database system designed to handle rich datatypes through the use of Abstract Data Types (ADTs) and provides scalability through the use of parallelism. In our experiments, we concentrated mainly on the relational features of Paradise.

Figure 8 shows some of the components of the Paradise system that are involved in optimizing and executing a query. The query optimizer is built using the OPT++ architecture [13], and uses a conventional dynamic programming algorithm based on the System-R optimizer [22]. The cost estimates in the optimizer are based on histograms stored in the system catalogs. The system uses MaxDiff histograms as described in [19]. This produces a static plan that contains the query execution strategy as well as the optimizer's estimates of the sizes of intermediate query results. This annotated plan is submitted to the database engine for query execution.

At query execution time, the Memory Manager of the database engine determines the allocation of memory to the various operators of the query. It determines the memory requirements (minimum and maximum memory demands) of each operator using the estimates provided by the optimizer. Based on the memory requirements of each operator, and by considering the trade-offs involved, it allocates some amount of memory to each operator. The amount of memory thus allocated to an operator represents the maximum memory that the operator is allowed to use during execu-

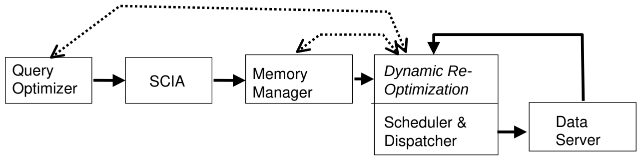

Figure 9: Query Execution with *Dynamic Re-Optimization*

tion. If all the data required by the operator does not fit into the allocated amount of memory, it has to spill some of the data to disk. Details of the Memory Management module of Paradise are described in [15].

The Memory Manager annotates a query execution plan with memory allocation values, and hands over the plan to the query scheduler and dispatcher for execution. The query scheduler and dispatcher executes a complex query execution plan in phases by partitioning it into a number of *segments*. Each segment is a subset of the operators in the query execution plan that can be executed concurrently. Typically, a segment consists of a set of consecutive operators that can be executed in a pipelined fashion. The different segments of a query execution plan are executed one after another in sequence. The dispatcher dispatches a segment of operators to the data-servers and waits for them to complete execution. When all the operators of a segment complete execution, a message is sent to the dispatcher, and it advances to the next segment in the execution plan.

Figure 9 shows how we modified Paradise to incorporate *Dynamic Re-Optimization*. First, the *statistics-collectors insertion algorithm* (SCIA) was added as a post-processing phase after the query optimizer. This takes the query execution plan produced by the optimizer and inserts statistics collection operators in it as described in Section 2.5. The scheduler-dispatcher is modified to take into account the *Dynamic Re-Optimization* algorithm. As in the previous design, after the Memory Manager is done with memory allocation, it hands over the plan to the scheduler and dispatcher. This partitions the plan into segments and begins dispatching each segment in sequence.

In the new scheme, when a segment is dispatched to the data-servers to be executed, it might contain *statistics-collector* operators. As far as the data-servers are concerned, these are regular operators similar to *hash-join* or *index-scan*. The only difference is that when a *statistics-collector* completes execution, it sends back to the dispatcher a message containing the statistics collected. At this point, the *Dynamic Re-Optimization* algorithm in the dispatcher is invoked. This can do one of three things at this point. First, it uses Equation 1 and Equation 2 (discussed in Section 2.4) to determine whether to consider re-optimizing the query. If the answer is yes, it invokes the query optimizer and obtains a new plan for the remainder of the query, using the new statistics. Then it uses the optimizer estimate of the cost of execution of the new plan to determine whether the cost of the new plan is actually less than the estimate for the old plan in spite of the re-optimization overhead. If this is true, the *Dynamic Re-Optimization* algorithm instructs the data-server to finish execution of the last operator and write the result to a temporary file. It deletes all the state information for the old plan from the dispatcher data-structures and then submits the new query plan for execution. If the new plan is not cheaper than the old plan, then the dynamic re-optimizer continues working with the old plan. However, it uses the new estimates to invoke the Memory Manager again to obtain an improved memory allocation for the plan based on the improved statistics. This process continues until the query completes execution.

In addition to implementing the *Dynamic Re-Optimization* and the *statistics-collectors insertion algorithms* in the system, we had to add the *statistics-collector* operator to the data-server. The *statistics-collector* operator was added as a regular streamed operator (similar to the *filter* operator). It took a stream of tuples as its input and produced exactly the same stream of tuples as its output. Since this operator just needs to examine the tuples without modifying or discarding any of them, it can be implemented without requiring an extra copy. To compute the size of the relation, the number of tuples, and the minimum and maximum value for an attribute, we maintain a single value that is updated after each tuple is examined. For computing a histogram, one database page is allocated to hold a reservoir sample [24] for the histogram. As each tuple is examined, the value of the corresponding attribute is copied into the reservoir according to the sampling technique described in [19]. When all the tuples from the input are exhausted, the reservoir is examined to build the histogram.

3.2 Experimental Results

To study the effect of *Dynamic Re-Optimization* on real queries, we performed experiments using some TPC-D queries. The TPC-D dataset generator was used with a scale factor of 3 to generate a 3GB database. Using this database, we ran queries Q1, Q3, Q5, Q6, Q7, Q8, Q10 described in the TPC-D specification [21][4]. All the experiments were run on a cluster of 4 PCs each configured with dual 133 Mhz Pentium processors, 128 MB of memory, dual fast and wide SCSI-2 adapters (Adaptec 7870P), and one Seagate Barracuda 2.1 GB disk drive (ST32500WC). Solaris 2.5 was used as the operating system. The processors were connected using 100Mbit/second ethernet and a Cisco Catalyst 5000 switch that has an internal bandwidth of 1.2 GB/second. The buffer pool was kept at 32MB at each node of the system. We purposely chose not to have a larger buffer pool since we wanted to study the effect of memory management techniques on query optimization. Refer to [21] for the specifications of the queries. We ran each query with and without the use of *Dynamic Re-Optimization*. Each query was executed 5 times and the average execution time was reported.

In all these queries, we set the value of μ (maximum allowable overhead) to 0.05 ensuring that none of the queries ever performed 5% worse than normal. The parameters θ_1 and θ_2 were kept at 0.05 and 0.2 respectively. An analysis of the sensitivity of the *Dynamic Re-Optimization* algorithm to the values of μ, θ_1, and θ_2 is contained in [12].

[4] The other queries in the TPC-D benchmark specification were not included in our experiments because some of the necessary features were not supported by Paradise. For the same reason, minor modifications were made to the queries that where included. In all cases where a query contained aggregates over expressions (*e.g.* SUM (L_EXTENDEDPRICE*(1-L_DISCOUNT))) we replaced it with a simpler aggregate expression (*e.g.* SUM (L_EXTENDEDPRICE)).

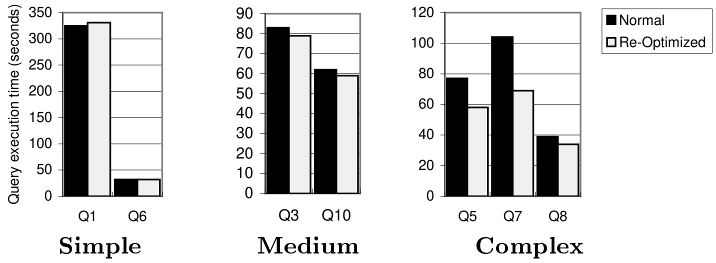

Figure 10: Performance of *Dynamic Re-Optimization*

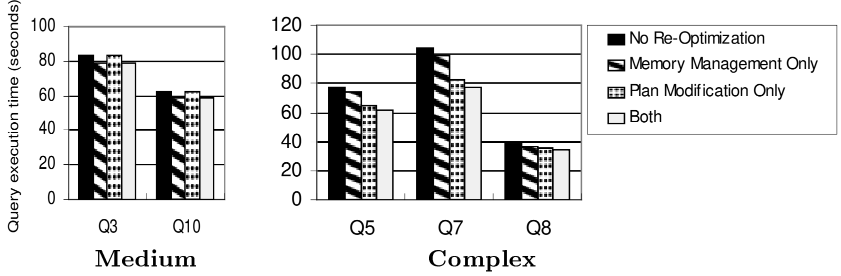

Figure 11: Isolating the effect of improvements due to memory management and plan modification

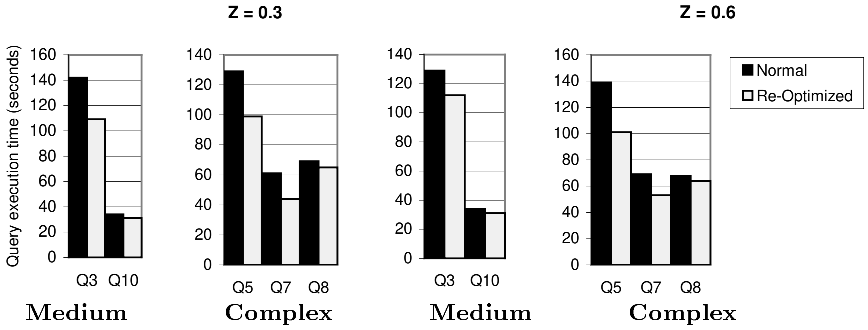

Figure 12: Effect of skew

Based on the expected effects of *Dynamic Re-Optimization* on different types of queries, we can classify all queries into three categories. Queries that contain zero or one joins will never get re-optimized. We refer to such queries as *simple* queries. Queries containing two or three joins will usually not benefit much from plan modification, but might see some benefits from improved memory management. We refer to this category of queries as *medium* queries. Finally, all queries containing four or more joins are the primary targets for which *Dynamic Re-Optimization* is designed. We will refer to them as *complex* queries. In the query set that we used, Q1 and Q6 are simple, Q3 and Q10 are *medium*, while Q5, Q7 and Q8 are complex.

Figure 10 shows the results of our experiments. We see that queries Q1 and Q6 do not benefit at all from *Dynamic Re-Optimization*. This is an expected result, since these are *simple* queries. We see a small increase in the execution time for Q1, indicating the overhead of statistics collection. Q3 and Q10 show modest improvements (upto 5%) in performance, while the *complex* queries show larger improvements (10 to 30%).

From the previous experiment, it is unclear how much of the performance improvement is due to improvements in the memory allocation for the query and how much is due to plan modification. To isolate these effects we performed another experiment in which the *Dynamic Re-Optimization* was run in two different modes. In one mode, the improvements in statistics were used solely for improving the memory management of the query, and plan modification was turned off. In the second mode, dynamic re-allocation of memory was turned off and only plan modification was used to improve the performance of the query. The results of this experiment are shown in Figure 11. A couple of interesting observations can be made about these results. First, we see that all the *medium* queries benefit only from improved memory management. Second, the *complex* queries benefit from both, improved memory management, as well as plan modification. They see a small improvement (5 to 10%) due to memory management and a larger one (10 to 20%) due to plan modification. Since the *simple* queries are not really affected by *Dynamic Re-Optimization* we have not included them in this or later experiments.

We also ran some experiments to study the effect of skew on the performance of *Dynamic Re-Optimization*. For this, we used the same queries with skewed data. Instead of generating TPC-D data with uniform distributions, we modified the data generator to skew all non-key using generalized Zipfian distribution ([27] as described in [18]). We ran two sets of experiments with values for the Zipf factor (z) value set at 0.3 and 0.6. The results of these experiments are plotted in Figure 12. Comparing these charts with the charts for the uniform data (Figure 10) we see that the relative performance of *Dynamic Re-Optimization* improves slightly as more skew is introduced in the system. In some cases the benefit from re-optimization actually decreases when skew increases (for example Q10). This can be attributed to the fact that in some cases, the accuracy of serial histograms actually increases when skew is increased.

4 Related Work

One of the earliest query optimizers [25] was, in some sense, a dynamic query optimizer. However, after the publication of [22], most of the work on query optimization has focussed on optimization of a query at compile time. Since the late 80s, however, the limitations of this approach have begun to be felt, and there has been an emergence of a number of different query optimization schemes in which some of the optimization decisions are postponed to query execution time.

[5] describes a scheme in which query execution plans generated by an optimizer are re-optimized just before query execution time if they are believed to be sub-optimal. At query optimization time, the statistics used by the optimizer to generate the optimal plan are stored with the plan in the database system. At query execution time, the actual statistics from the system catalogs are compared against the statistics stored in the plan. If they are found to differ significantly the query is re-optimized before execution. This differs significantly from our approach. First, the query is only re-optimized before execution begins. In their approach, there is no collection of statistics, or modification of the plan in the middle of query execution.

The competition model of Antoshenkov [2, 3] represents another way of dynamically determining the plan of a query. In his approach, competing executions start executing using different plans. After a while, it becomes clear that one of the plans is better than the others, and the execution of the sub- optimal plan is stopped. While this approach might work well for determining which access method to use for a particular table-scan, or which join algorithm to use for executing a particular join, it cannot be extended easily to the case where the join order for a complex query might be sub-optimal. Further, the competition model cannot be used for dynamically improving the resource allocation of a query.

Query Scrambling described in [1] also does some dynamic re-optimization of a queries but it is directed towards a very different problem. Execution of queries that access data from widely distributed data sources can get stalled if the data from some data source is arrives very slowly. Query Scrambling dynamically re-schedules operators to tackle such unexpected delays, and in some cases, adds new operators. This is a very specific technique to tackle a specific problem found in distributed databases. Unlike our algorithm, Query Scrambling is not intended to be a general algorithm for dynamic re-optimization of sub-optimal queries

One important reason for sub-optimality of query execution plans is that a lot of information about the run-time system (availability of memory, bindings of host language variables, existence of indices) is not available at query compile time. The dynamic execution plans of [8, 7], and the parametric query optimization algorithm of [10] try to tackle this problem. Their approach is to produce a composite plan that is in effect a combination of a number of different plans, each of which is optimal for a given set of values of run-time parameters. One of the problems with this approach is that as the number of things that are unknown at query optimization time increases, the space/time complexity of the optimization algorithm, and the complexity of the parameterized/dynamic plan produced by the algorithm increases. Given the limited amount of time that is available for query optimization, these approaches either have to resort to the use of randomization for exploring the vast search space [10], or to make simplifying assumptions [7]. Another shortcoming of these approaches is that they do not address the issue of statistical and propagational errors in estimates. Thus, if a histogram-based estimate of the selectivity of a predicate is inaccurate, the corresponding sub-optimality in a plan cannot be detected using these approaches. However, they do have an advantage over Dynamic Re-Optimization in that they do not impose any overheads on query execution at run-time.

A hybrid algorithm that combines the parametric/dynamic query plans approach and the *Dynamic Re-Optimization* algorithm could possibly combine the best features of both approaches. The query optimizer can try to anticipate the most common cases that might arise at run-time and produce a parameterized plan that covers these possibilities. At query execution time, statistics can be observed/collected to determine which plan to choose for query execution. If a situation arises at run-time that is not covered by the common cases anticipated by the query optimizer, dynamic re-optimization can be used. This approach suggests a possible direction of future research.

5 Conclusions

In this paper, we have described an algorithm that can detect sub-optimalities in query execution plans for complex queries, and improve the performance of such queries by dynamically re-optimizing the execution plan. Strategically placed statistics collectors are inserted into query execution plans to observe sizes and data distributions of intermediate query result sizes at run-time. These run-time statistics are used for improving the allocation of shared resources (memory) to the query, and for modifying the query execution plan if need be. We also describe how this can be done efficiently without placing too much of an overhead on the execution of the query. We have demonstrated experimental results to support our claim that Dynamic Re-Optimization can significantly improve the performance of complex queries if their query execution plans are sub-optimal without significantly slowing down the queries whose plans do not benefit from re-optimization.

As emerging new applications force databases to support complex decision support queries, complex data-types and user-defined methods, it will become more and more difficult for query optimizers to statically produce good query execution plans. Some form of re-optimization of query execution plans at run-time will become necessary in such cases. We believe that the techniques we have presented, possibly in combination with parameterized plans will form the basis for the future evolution of query optimizers to meet this challenge.

Declarative query languages and automatic query optimization were an important reason for the success of relational database systems. Lack of good query optimizers could very well lead to the downfall of the next wave of innovations in database system technology. In this paper, we have examined the inadequacies of traditional query optimizers in dealing with issues raised by modern database systems and demonstrated ways to overcome them. We believe that the ideas contained in this paper represent an important step in ensuring that query optimizers keep up with the other advances in database systems.

Acknowledgements

We would like to thank the Paradise team for their help with the use of the Paradise Database System. We would also like to thank Yannis Ioannidis and Joseph Hellerstein for useful discussions, and the anonymous referees for their comments on drafts of this paper.

References

[1] Amsaleg, L., Franklin, M. J., Tomasic, A., and Urhan, T. "Scrambling Query Plans to Cope with Unexpected Delays". In *The 4th International Conference on Parallel and Distributed Information Systems (PDIS)* (Miami Beach, Florida, Dec. 1996).

[2] Antoshenkov, G. "Dynamic Query Optimization in Rdb/VMS". In *In Proceedings of the IEEE Conference on Data Engineering* (1993), pp. 538–547.

[3] Antoshenkov, G. "Dynamic Optimization of Index Scan Restricted by Booleans". In *In Proceedings of the IEEE Conference on Data Engineering* (1996), pp. 430–440.

[4] Chen, M. S., et al. "Using Segmented Right-Deep Trees for Execution of Pipelined Hash Joins". In *Proc. of the 18th VLDB Conf.* (1992).

[5] Derr, M. A., Morishita, S., and Phipps, G. "Adaptive Query Optimization in a Deductive Database System". In *In Proceedings of the Proceedings of the Second International Conference on Information and Knowledge Management* (Washington D. C., USA, 1993).

[6] Flajolet, P., and Martin, G. N. "Probabilistic Counting Algorithms for Database Applications". In *Journal of Computer and System Sciences* (1985), vol. 31(2), pp. 182–209.

[7] Graefe, G., and Cole, R. "Optimization of Dynamic Query Evaluation Plans". In *Proceedings of the 1994 ACM-SIGMOD Conference* (1994).

[8] Graefe, G., and Ward, K. "Dynamic Query Evaluation Plans. In *SIGMOD Proceedings* (June 1989), ACM, pp. 377–388.

[9] Ioannidis, Y., and Christodoulakis, S. "On the Propogation of Errors in the Size of Join Results". In *Proceedings of the 1991 ACM-SIGMOD Conference* (Denver, Colorado, May 1991).

[10] Ioannidis, Y., Ng, R. T., Shim, K., and Sellis, T. "Parametric Query Optimization". In *Proc. of the 18th VLDB Conf.* (1992).

[11] Ioannidis, Y., and Poosala, V. "Balancing Histogram Optimality and Practicality for Query Result Size Estimation". In *Proceedings of the 1995 ACM-SIGMOD Conference* (San Jose, California, May 1995).

[12] Kabra, N. *"Query Optimization for Relational and Object-Relational Database Systems"*. PhD thesis, University of Wisconsin, Madison, 1998.

[13] Kabra, N., and DeWitt, D. J. "Opt++: An Object Oriented Implementation for Extensible Database Query Optimization". In *to appear in The VLDB Journal* (1998).

[14] Mehta, M., and DeWitt, D. J. "Dynamic Memory Allocation for Multiple Query Workloads". In *Proc. of the 19th VLDB Conf.* (Dublin, Ireland, 1993).

[15] Nag, B., and DeWitt, D. J. "Memory Allocation Strategies for Complex Decision Support Queries". Submitted for publication.

[16] Ono, K., and Lohmann, G. "Extensible Enumeration of Feasible Joins for Relational Query Optimization". In *Proc. of the 16th VLDB Conf.* (Aug. 1990).

[17] Patel, J. M., et al. "Building a Scalable Geo-Spatial DMBS: Technology, Implementation, and Evaluation". In *Proceedings of the 1997 ACM-SIGMOD Conference* (Tuscon, Arizona, May 1997).

[18] Poosala, V. "Zipf's Law". Tech. rep., University of Wisconsin, Madison, 1995.

[19] Poosala, V., and Ioannidis, Y. "Histogram-Based Solutions to Diverse Database Estimation Problems". In *Data Engineering Bulletin* (1995), vol. 18(3), pp. 10–18.

[20] Poosala, V., Ioannidis, Y., Haas, P. J., and Shekita, E. "Improved Histograms for Selectivity Estimation of Range Predicates". In *Proceedings of the 1996 ACM-SIGMOD Conference* (Montreal, Canada, June 1996).

[21] Raab, F. *"TPC Benchmark D – Standard Specification, Revision 1.0"*. Transaction Processing Performance Council, May 1995.

[22] Selinger, P., Astrahan, M., Chamberlin, D., Lorie, R., and Price, T. "Access Path Selection in a Relational Database Management System". In *Proceedings of the ACM SIGMOD Conference on Management of Data* (May 1979).

[23] Stonebraker, M., Anton, J., and Hirohama, M. "Extendability in POSTGRES". In *Data Engineering Bulletin* (1987), vol. 10(2), pp. 16–23.

[24] Vitter, J. S. "Random Sampling with a Reservoir". In *ACM Transactions on Mathematical Software* (1985), vol. 11, pp. 37–57.

[25] Wong, E., and Youssefi, K. "Decomposition – A Strategy for Query Processing". In *ACM Transactions on Database Systems* (Sept. 1976).

[26] Yu, P. S., and Cornell, D. W. "Buffer Management Based on Return on Consumption in a Multi-Query Environment". In *VLDB Journal* (Jan. 1993), vol. 2(1).

[27] Zipf, G. K. *"Human Behavior and the Principle of Least Resistance"*. Addison-Wesley, Reading, MA, 1949.