The Fittest Survives: An Adaptive Approach to Query Optimization

Hongjun Lu Kian-Lee Tan
Dept Information Systems and Computer Science
National University of Singapore
Lower Kent Ridge, Singapore 0511

Son Dao
Hughes Research Laboratories
Information Sciences Lab.
Malibu, CA 90265, USA

Abstract

Traditionally, optimizers are "programmed" to optimize queries following a set of build-in procedures. However, optimizers should be robust to its changing environment to generate the fittest query execution plans. To realize adaptiveness, we propose and design an adaptive optimizer with two features. First, the search space and search strategy of the optimizer can be tuned by parameters to allow the optimizer to pick the one that fits best during the optimization process. Second, the optimizer features a "learning" capability for canned queries that allows existing plans to be incrementally replaced by "fitter" ones. An experimental study on large multi-join queries based on an analytical model is used to demonstrate the effectiveness of such an approach.

1 Introduction

With the widespread adoption of database management systems comes greater expectations from the database user community. In particular, there is an increasing demand for high performance (high throughput and low response time). At the same time, databases are growing in size and queries are becoming more complex. For a large number of new applications (e.g. deductive databases and object-oriented database systems), complex queries that involve very large number of joins (> 30) are not uncommon. For example, in object-oriented database systems, queries typically involve "path expressions". One way of executing these path expressions is to turn them into joins. In these cases, each query that navigates through the database is likely to involve many joins. While using more powerful hardware helps, the design of query processing and optimization algorithms that utilize the system resources effectively is equally crucial for maximizing the system performance.

In this paper, we revisit the problem of optimizing large number of joins. We propose a new adaptive query optimizer with two novel features to optimizing large multi-join relational queries. First, the search space of the optimizer can be tuned by parameters. Varying these parameters gives rise to a family of optimization heuristics of varying complexities. This allows the optimizer to pick the one that fits best during the optimization process. Second, the optimizer features a "learning" capability for canned queries. By repeatedly optimizing a canned query, a different (and possibly better) plan than previous optimizations is obtained, and the query plan can be refined accordingly. An experimental study based on an analytical model is used to demonstrate the effectiveness of the approach. The results of the study show that the proposed technique is competitive for ad-hoc queries, and produces more optimal plans for canned queries eventually.

Throughout this paper, execution plans for multi-join queries will be depicted as trees, with each internal node corresponding to a join and each leaf node corresponding to a base relation. To simplify the discussion, only hash-based join methods [DKO+84, Sha86] will be considered.[1] As in [SD90], each join in an exe-

Proceedings of the 21st VLDB Conference
Zürich, Switzerland, 1995

[1] All the algorithms studied in this paper are not limited to

cution tree will have its *building* relation to the left and its *probing* relation to the right. Hash tables are built on building relations and probed by probing relations. If all the internal nodes of an execution tree have at least one leaf (i.e., base relation) as a child, then the tree is called *deep* [IK91]. Otherwise it is called *bushy*. A *left-deep* tree is a deep tree whose probing relations are restricted to base relations. Conversely, a *right-deep* tree is a deep tree whose building relations are restricted to base relations. As defined, the search space of bushy trees includes deep trees, which in turn includes left-deep and right-deep trees.

The rest of this paper is organized as follows. In the next section, we give an overview of adaptive query optimization. Section 3 proposes an optimization framework for multi-join queries. In Section 4, we describe how the framework may be extended with learning capability. Section 5 presents the experiments and results that were conducted, and finally, conclusions are drawn in Section 6.

2 Adaptive Query Optimization

In this section, we give an overview of adaptive query optimization. After motivating the need for adaptive query optimizer, we propose the design of such an optimizer and discuss issues that must be addressed for it to be realized. We shall also compare such an approach with existing optimization techniques.

2.1 The Fittest Survives

Most of today's optimizers are "programmed" to perform in a certain way. For example, reoptimizing a query will go through the same optimization process and produce the same query plan, and nothing can be done to improve a sub-optimal plan. As another example, it will consume the same amount of resources to optimize a query regardless of the system load. As such, it lacks the ability to accommodate the ever-changing environment.

However, it is crucial for an optimizer to be able to adapt to different situations because it can lead to better overall system performance (we shall illustrate with examples in the following subsections). To "revolutionize" the current technology, a new generation of optimizers must be built that can adapt to different scenarios, such as the query, the system resource, and the optimization objectives. We shall discuss some of these here.

hash-based join methods. Considering k join methods require choosing the best join methods for each join. This will increase the search space and complexity of the algorithm by a factor of $O(k^n)$ where n is the number of joins [TL91].

Adaptive to Query

The optimizer must adapt to different queries differently depending on the query type (whether it is an ad-hoc query or a canned query), its complexity (number of relations) and the cardinalities of the relations.

The cost of processing an ad-hoc query really comprises of two components: the optimization cost and the processing cost. There is clearly a tradeoff between the optimization cost and the processing cost – a long optimization cost generally leads to a short processing cost whereas a short optimization cost may lead to a long processing cost. The optimizer must adapt accordingly depending on the complexity of the query, and the cardinalities of the relations. For example, consider two multi-join queries with the same join graph and same number of relations – query 1 involves small relations while query 2's relations are large. Under a traditional optimizer, the optimization cost for both queries are equally high. While it is beneficial to spend a high optimization cost for query 2 since its execution time is expected to be longer than query 1, query 1's optimization cost may be too high compared to its processing cost since it involves only small relations and its execution time is expected to be short. Had the optimizer been capable of adapting, it might have restricted the search space for query 1 to keep down the optimization time and enlarged the search space for query 2 to generate better plans for both queries. In other words, the search space that fits best for the particular query should be selected during the optimization.

For complex canned queries, once a plan is generated, it is traditionally left unchanged until some critical statistics are modified. If a plan is sub-optimal, it will remain so. Moreover, we may not even know how far or close is the plan from the optimal solution. The sub-optimality of a plan will become more common for complex queries and more crucial since repeated executions of the queries may degrade the system performance. An adaptive optimizer, however, may improve the plan of a canned query by "learning" from previous optimizations of a canned query and incrementally refine the existing plan. In other words, "fitter" plan will survive and replace the less fit ones.

Adaptive to System Resource

An adaptive optimizer should be adaptive to the system workload. For example, when the system workload is high, then the optimizer may traverse a smaller search space. On the contrary, a large search space (and hence higher optimization time) can be tolerated when the system workload is low.

The optimizer should also treat the optimization time as a critical resource. In this way, the longer

the optimization time a query is assigned, the better the quality of the plan will be.[2] Complex canned queries have traditionally been assigned high optimization cost because the high cost can be amortized over multiple runs of the queries. However, it may not be practical to assign a long optimization time to a query (even canned query) in a single optimization process because today's applications are mission critical and runs 24 hours a day, 7 days a week, 365 days a year. To overcome the problem of overcommitting optimization resource, an adaptive optimizer can "break up" a long optimization time to a query into shorter *timeslices*. The query is then optimized multiple times each with a optimization period equivalent to a timeslice. To avoid generating the same plan (as is done traditionally), the optimizer adapts and learns from previous optimizations so that the quality of the plan of a query can be improved at each optimization of the query. The effect is equivalent to that of optimizing the query using a long optimization time. Note that the query is not optimized consecutively otherwise it is no different from existing techniques. Instead, the query is optimized at different runs, i.e. if the query is submitted 10 times, then it may be optimized 10 times, each resulting in a different plan.

Adaptive to Optimization Objectives

Another important aspect of an adaptive optimizer is that it must adjust to different optimization objectives for different applications and queries. An urgent request requires the optimizer to minimize the response time. On the other hand, the optimizer may want to maximize the resource usage for "batch" queries. Some applications may require optimizing a weighted sum of the response time and resource consumption. Yet another optimization objective that is becoming increasingly important is that of minimizing dollar charges. This objective is particularly important to users whose queries involve accessing other organization's resources.

Even if the user does not specify an objective, the optimizer should adapt the optimization objective in some ways. For example, if the system load is heavy, it may be more appropriate to optimize resource utilization. When the system load is light, minimizing the response time may become acceptable.

2.2 The Architecture of an Adaptive Query Optimizer

An adaptive query optimizer must be able to

1. traverse a different search space at different invocation (even for the same query).

2. enhance the plans of canned queries over times

Figure 1 shows the design of such an optimizer. The optimizer comprises two main components: the *tuner* and the *plan generator*.

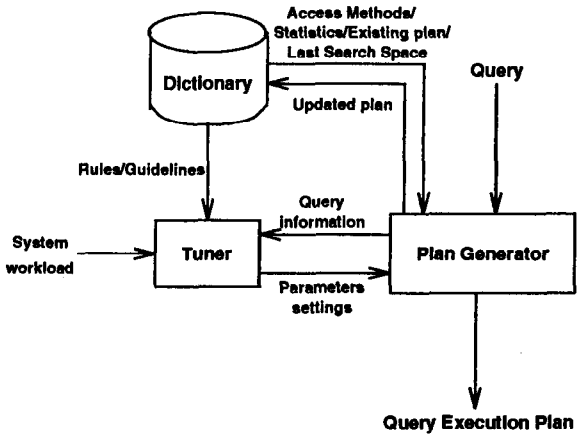

Figure 1: An Adaptive Query Optimizer.

Our plan generator is highly parameterized. We shall leave the discussion of the parameters to the next section. It suffices for the moment to know that varying the set of parameters results in different optimization heuristics of varying complexity. Depending on the setting at the time the query is submitted, the particular algorithm will be used to optimize the query.

The tuner is a new component which acts as the "administrator" of the optimizer. Its responsibility is to set the values of the parameters for different queries, applications and resources. The decision of the tuner is based on several pieces of information: (a) the query to be optimized, (b) the system workload at time of optimization, and (c) the rules/guidelines that has been established. For example, a simple rule may be to assign the amount of optimization time according to the number of relations in the query. Other rules may be determined by the complexity of the queries, the system load, etc.

Now, let us walk through the optimization process of a query. For an ad-hoc query, the plan generator (a) passes the query information to the tuner, (b) the tuner determines the parameter settings for the plan generator, and (c) the plan generator looks up the dictionary to obtain the statistics and access methods available, and produced a query plan based on the optimization algorithm (obtained by the parameter settings).

For a canned query, if it is the first time the query is submitted, then it is treated as an ad-hoc query. However, the generated plan is retained together with

[2] Randomized algorithms are examples of existing algorithms that treat optimization time as a resource though it expended all in a single optimization process.

the query. Additionally, information about the search space traversed is kept. If it is not the first run, after obtaining the parameter settings from the tuner, the plan generator will optimize the query using a different search space from that which the query was last optimized. The cost of the generated plan is compared with that of the existing plan, and the better plan is retained and used.

Such an adaptive optimizer is highly desirable, and has the following advantages:

1. It can act as an optimizer generator. A system can choose a fixed set of parameters, and "generate" its own optimizer. This corresponds to traditional optimizers.

2. More importantly, the variability allows the system to apply different algorithms to different applications and queries. The flexibility to vary the optimization algorithm (and hence the optimization cost) allows us to tune an overall minimum cost for different queries and applications.

2.3 The Issues

To realize the full potential of an adaptive query optimizer such as the one discussed above, several issues has yet to be addressed:

1. What are the set of parameters that can be used to tune the optimizer search space?

2. How should these parameters be tuned?

3. How can the optimizer "learn" and model optimization time as a resource for canned queries?

In this paper, we address the first and last issues which we describe in greater details in the next two sections respectively. The challenging task of tuning the optimizer depends on applications, and requires some amount of research before a set of rules/guidelines can be developed.

2.4 Related Work

Several researches have addressed the issue of adaptive query optimization, most of which focus on increasing the search space. Earlier work on increasing the search space used by the query optimizers have mainly addressed the benefits of allowing composite inners [Koo80, RR82].

The Exodus' rule-based optimizer generator can vary the search space considered by an optimizer by changing the rules and methods of its input [GD87]. As such, different optimizers with different search space can be adapted to different applications. However, once an optimizer is generated, its search space cannot be varied without generating a new optimizer.

In [OL90], Starburst's extensible join enumerator adopts a generate and filter approach. The join generator, which is based on dynamic programming, generates a set of feasible plans. This set can be adjusted by parameters that control the use of Cartesian products and composite inners. A set of filters may then be used to further pruned the generated plans. However, the exponential complexity of dynamic programming may limit the optimizer to queries that involve not more than 15 relations. The Starburst optimizer also has a greedy join enumerator that can generate left-deep, right-deep and bushy execution trees. It also has a tuning knob to control just how bushy an execution tree can get.

Lanzelotte and Valduriez [LV91] proposed a promising approach in which a hierarchy may be used to represent optimization algorithms. Each internal node of the hierarchy represents a category in which its children nodes belong to. The leaf nodes represent the set of optimization algorithms used by the optimizer. In this way, an optimizer can be built that will vary its search space for different queries by simply selecting the desirable algorithms within the desirable categories. However, this will mean that the number of different search space will be proportional to the number of optimization algorithms. Moreover, how the search space for each of the optimization algorithms may be varied was largely ignored.

3 The Tunable Optimization Framework

A first step towards adaptive query optimization is to design an optimizer whose search space can be tuned to adapt to different scenarios. In this section, we propose an optimization framework with tuning capability. The framework iteratively selects a set of relations to be optimized, and the optimization process may be guided by a smaller set of the selected relations. The framework is targeted at ad-hoc queries. Because it is highly tunable by parameters, it can be set to generate a good plan in a reasonable amount of time. We shall describe how to extend the framework to support incremental optimization for canned queries in Section 4.

3.1 Tuning Parameters

Before we look at the framework, let us consider the parameters that we have identified for tuning the search space. In this section, we describe the three categories that were chosen. The three categories, which are described in more detail below, are:

1. search space based,

2. algorithm and algorithm specific, and

3. optimization objective.

3.1.1 Search Space Based

The search space can be tuned by (1) the shape of the execution tree (*TreeShape*), (2) enabling or avoiding cross products (*AvoidCrossProducts*), and (3) the number of relations to be optimized in each iteration (*NumRel*). Varying any of these parameters affects the search space directly.

Parameter 1: TreeShape. This parameter restricts the shape of the execution tree. It is set to either *left-deep*, *right-deep*, *deep*, or *bushy*. Left- and right-deep trees have the smallest search spaces, followed by deep and bushy trees [OL90].

Parameter 2: AvoidCrossProducts (ACP). This parameter is set to either *True* or *False*. If it is set to *True*, then cross products are avoided whenever possible. Otherwise, two relations may be joined regardless of whether there is any join predicate between them. In the event that **AvoidCrossProducts** is set to *True* and cross products are unavoidable, then they are deferred until the end of the query.

Parameter 3: NumRel. This parameter determines the maximum number of relations to be optimized in each iteration. For example, given a 40-relation join, setting $NumRel$ to 15 leads to 3 iterations needed to produce the final plan. While it is independent of the optimization algorithm employed, it may be restricted by practical constraints. For example, if dynamic programming is used as the optimization algorithm, then the maximum number of relations should not be more than 15 because of the exponential complexity of dynamic programming. On the other hand, if a greedy heuristic with polynomial time complexity is used, then it can be set to as large as the number of relations in the query.

3.1.2 Algorithm and Algorithm Specific

The framework is designed such that existing algorithms can be extended easily with tuning features. This is achieved by specifying the parameter *OptAlgo*. Depending on the set of algorithms that is supported, it can be set to DP for dynamic programming, RA for randomized algorithms, and GD for greedy-based heuristics. Among the randomized algorithms, one can choose to use iterative improvement (II) [SG88], simulated annealing (SA) [SG88], genetic algorithm (GA) [LV91], or two phase optimization (2PO) [IK90].

Some algorithms also have algorithm specific parameters. For example, randomized algorithms generally require a termination criterion. A commonly used measure is the number of iterations whereby there is no further improvement over the best plan. As another example, the heuristic algorithms proposed in [SYT93] increases the search space by keeping a number of alternative subplans for the same set of relations. The maximum number of alternative subplans to be kept is specified by a parameter.

3.1.3 Optimization Objective

The parameter *OptObj* specifies the optimization objective to be used. It can be set to *ResponseTime*, *ResourceConsumption*, *WeightedSum*, or *Monetary*. While it doesn't affect the search space, as we have seen, the flexibility to vary the optimization objective for different queries or applications is highly desirable. For simplicity, for this paper, we restrict our discussion to minimizing elapsed time. Varying the optimization objectives require building a parameterized cost models only.

3.2 The Framework

Figure 2 shows the proposed framework. It comprises several iterations, each of which finds an optimal subplan. For simplicity in presentation, some details of the algorithm are abstracted. Before walking through the algorithm, let us describe the functions that are called by it. The function $selRel(G, R_i, R_j, k, NumRel, ACP)$ returns S, a set of $NumRel$ relations from the join graph G. The argument k is a "flag" that indicates whether R_i and R_j must be in S. This is needed only for the first iteration. For the subsequent iterations, R_i and R_j are just "dummies" and they are not used anymore. If ACP is False, then the relations in S need not form a connected subgraph in G; otherwise they must lead to a connected subgraph in G. Function $optimizeGraph(G, S, TreeShape, NumRel, OptAlgo)$ employs the optimization algorithm determined by *OptAlgo* to produce either (1) a partial plan for G base on S if $NumRel < n$ or (2) a full plan for G if $NumRel = n$. *TreeShape* determines the shape of the generated plan. Function $postProcess(G, S, subPlan)$ derives a new plan, *newSubPlan*, from *subPlan* for a subset of the relations in S. It is motivated by observing that fixing the number of relations to pick at each iteration may lead to poor performance.

Let us walk through the algorithm in Figure 2. As shown on line 2, the algorithm tries all possible pairs of relations for its first join. For each starting pair, an execution plan is generated. At each iteration, all the unjoined pairs of relations are considered (line 6). Function *selRel* is then invoked to obtain a set of relations that *optimizeGraph* uses to generate a subplan (line 7,8). The generated subplan is then postprocessed (line 9). Next, all the relations that has been

Input: A join graph, G, of n relations R_1, ..., R_n
Output: Optimal multi-join plan, $OptPlan$

1. $OptPlan.cost = \infty$
   // try all relation pairs as a starting point
2. for R_i, R_j in G
3. if (R_i, R_j connected by an edge or ACP is False) do
4. $CurrentPlan = \emptyset$
5. k = 1
6. while $|G| > 1$ do
7. $S = selRel(G, R_i, R_j, k, NumRel, ACP)$
8. $subPlan = optimizeGraph(G, S, TreeShape, NumRel, OptAlgo)$
9. $newSubPlan = postProcess(G, S, subPlan)$
10. add $newSubPlan$ to $CurrentPlan$
11. collapse R_x in G, forall $x \in newSubPlan$
12. k = k + 1
13. endwhile
14. endif
15. if $CurrentPlan.cost < OptPlan.cost$ do
16. $OptPlan = CurrentPlan$
17. endif
18. endfor
19. output $OptPlan$

Figure 2: The tunable optimization framework.

selected are collapsed into one node in G to reflect the fact that they have been joined (line 11). This process is repeated until all the relations have been joined, and the least costly plan generated among all starting pairs of relations forms the final output (line 15-17).

3.3 Tuning the Search Space

From the description of the framework, we see that varying the tuning parameters can lead to a family of optimization algorithms of varying complexities, some of which have appeared in the literature and others are new. For example, by setting $NumRel$ to all the relations, and using dynamic programming as the optimization algorithm, and setting $TreeShape$ to left-deep, we effectively have the traditional system R algorithm [SAC+79] with $O(2^n)$ complexity for n relations. As another example, by setting $NumRel$ to all the relations, and using the greedy heuristic in [SYT93] as the optimization algorithm, we effectively have a greedy algorithm with complexity $O(n^3)$. On the other hand, setting $NumRel$ to different values (smaller than the number of relations) leads to a family of new optimization algorithms.

4 Incremental Optimization

As mentioned, traditional optimizers generate the same plan for the same query given the same amount of resources. If the quality of the plan is far from optimal, then the performance of the system suffers. This may be unacceptable for canned queries since they are repeatedly executed. We propose that canned queries be incrementally optimized. To do so, the optimizer must be "intelligent" enough to "learn" from previous optimizations of a query so that the subsequent optimizations can generate different and better plan for the query.

4.1 Incremental Optimization using Different Search Space

Our solution is to organize the search space into subspaces (possibly overlapping), so that at each optimization of a query, the optimizer works on a different subspace. All that is needed of the optimizer is to "remember" the subspaces that it has searched in previous optimizations. To illustrate the idea, suppose we can order the subspaces, say S_1, S_2, and so on, such that the i^{th} optimization will search subspace S_i. The basic idea works as follows. When the optimizer is invoked to optimize a canned query for the first time, it searches S_1 to produce a plan. At the same time, the optimizer will remember the plan. Some time later, the same query is to be processed again. The optimizer is called again but this time, it searches S_2 (knowing that S_1 has already been searched). It compares the new plan with the existing plan, keeps the better one and uses it. This process is repeated until all the subspaces have been searched, in which case, subsequent execution of the query requires no further optimization.

The main issue lies in organizing and ordering the search space. We shall demonstrate how this can be done in the context of the proposed optimization framework described in Section 3. Suppose we have n relations and k is the predetermined number of relations to be selected for optimization in each iteration of the algorithm. We also made several interesting observations and conclusions.

Observation 1: If the k relations selected at each iteration of the optimization framework is different at each *run* (we use the term "run" to mean an invocation of the optimization framework), we expect a different plan to be generated. For example, suppose in a run, the first iteration selects R_1 to R_k. Let the plan generated be plan A. Now, consider another run that selects R_{k+1} to R_{2k} in the first iteration. Let this plan be plan B. Clearly, plan A is different from plan B.

This observation leads to the following conclusions:

- The set of relations selected at each iteration can be used to partition the subspace that the optimizer has searched.
- By making the optimizer remembers the set of relations selected, the optimizer can generate different plans at different runs of the same query.

Observation 2: The iterations of the optimization framework can be ranked in the following sense. The first iteration of the optimization framework is the most important because a different set of relations selected in the first iteration affects all subsequent iterations. The second iteration ranked second, and so on.

This leads us to the following heuristic:

> To cut down on the complexity of the optimizer, we can always restrict it to remember only the relations for the first few iterations, and not all the iterations.

Observation 3: To target the optimizer for both ad-hoc and canned queries, the most important run of the optimizer is the first run. Ad-hoc queries need a good plan from the first and only run. On the other hand, canned queries can hope to incrementally improve on the plans that have been generated form earlier runs.

This also leads to the following heuristic:

> The optimizer should search a larger space for the first run, but for subsequent runs, it can restrict to a smaller but different search space.

In the current study, we restrict the optimizer to just remembering the set of relations selected in the first iteration. All that is needed is to try all combinations of choosing k relations from n. This value effectively represents the total number of runs before we can say that the plan is "optimal". So, we begin by considering R_1 to R_k. The next run considers R_1 to R_{k-1} and R_{k+1}, and the next run considers R_1 to R_{k-1} and R_{k+2}, and so on. Generating all these sequences is straightforward.

Figure 3 shows the framework extended with the "learning" capability. It comprises two parts. The first part (lines 2-4) corresponds to the case when a canned query is optimized in the first run or an ad-hoc query is optimized, in which case, the optimization framework described earlier is called to produce the plan. The parameter *SST* represents the information remembered by the optimizer – the parameter settings and the relations already considered in the first iteration – for it to know the search spaces that have been traversed. The generated plan and the *SST* are then stored.

Input: A join graph, G, of n relations R_1, ..., R_n
Output: Optimal multi-join plan, *OptPlan*

```
1. if planExists(G) = FALSE do
2.    plan = optFramework(G, SST)
3.    store SST
4.    store plan
5. else
6.    CurrentPlan = ∅
7.    k = 1
8.    while |G| > 1 do
9.       S = selRel(G, k, SST, NumRel, ACP)
10.      subPlan = optimizeGraph(G, S, TreeShape,
                    NumRel, OptAlgo)
11.      newSubPlan = postProcess(G, S, subPlan)
12.      add newSubPlan to CurrentPlan
13.      collapse R_x in G, forall x ∈ newSubPlan
14.      k = k + 1
15.   endwhile
16.   retrieve existing plan, OptPlan
17.   store SST
18.   if CurrentPlan.cost < OptPlan.cost do
19.      OptPlan = CurrentPlan
20.      store OptPlan
21.   endif
22.endif
23.output OptPlan
```

Figure 3: Incremental optimization framework.

The second part (lines 5-23) is executed when an existing plan exists. It is essentially a simpler version of the optimization framework, i.e. it did not try all possible pairs of relations for its first join and the starting set of relations is determined by *SST*.

4.2 Incremental Optimization with Randomized Algorithms

While randomized algorithms can be used with the incremental optimization framework described above, it can also be used to optimize queries incrementally in a different and straightforward way. Randomized algorithms generally works iteratively as follows: (1) generate an initial plan (randomly), (2) apply transformation rules randomly to improve the plans, (3) repeat steps 1 and 2 until some termination criterion is met.

Clearly, if the optimizer can ensure that each invocation generates a different initial plan in step 1, then we have an incremental randomized optimizer. All that is required is for the optimizer to remember the set of initial plans that has been generated. Instead

of a random generation, we can systematically generate an initial plan by specifying the order of the plan. This can be done in a manner similar to that described for the incremental optimization framework, i.e. the optimizer remembers a subset of relations that must be joined to generate the initial plan.

4.3 Some Simple Optimizations

There are some optimizations that can be incorporated to cut down the optimization cost, i.e. we don't have to try all combinations that may be generated for *SST*. First, some combinations may not generate good plans and such cases can be pruned away immediately. For example, if the selected relations results in a disconnected join subgraph, we can skip this case, and try the next one that results in a connected graph.

Second, some combinations may lead to the same plan, and we can terminate "prematurely". For example, suppose the first run selects R_1 to R_k, and the second run results in choosing R_1 to R_{k-1} and R_{k+1}. And it turns out that in the first iteration of both runs, the actual number of relations used are 9, and they are both R_1 to R_9. then, in the second run, we can bypass all subsequent iterations because the generated plan is definitely the same as the existing plan.

Finally, it may not be necessary to try all combinations. If the database administrator is comfortable with the quality of a plan (for example, there is no improvement after some threshold number of runs), then incremental optimization can be turned off.

5 Preliminary Results

In this section, we describe the experiments that were conducted based on an analytical cost model, and the results of the experiments. The cost model is similar to the one used in [SYT93]. Two optimizers were implemented in C and the experiments were tested on a Sun Sparc-2 machine. The purpose of the experiment is to demonstrate the necessity and feasibility of adaptive query optimization.

5.1 Optimizers Studied

In this study, two adaptive query optimizers were built. One was based on dynamic programming, while the other used a randomized algorithm. For the former, we will refer to it as *Incremental Dynamic Programming (IDP) optimizer*, and for the latter, we will call it the *Incremental Randomized Algorithm (IRN) optimizer*.

The IDP optimizer iteratively selects a set of *NumRel* relations, and employs the modified dynamic programming algorithm proposed in [SYT93] as the optimization algorithm to produce a subplan for the selected relations. The modified dynamic programming algorithm extends the traditional dynamic programming to include memory consumption while using elapsed time as the cost metric [SYT93]. For practical reasons, *NumRel* is restricted to a maximum value of 15.

For the IRN optimizer, the optimization algorithm used in each iteration is the two phase optimization algorithm (2PO) [IK90]. 2PO comprises of two phases. In the first phase, the randomized algorithm Iterative Improvement is applied to produce a quick solution. In the second phase, the randomized algorithm Simulated Annealing is used to further improve on the plans generated in the first phase. As shown in [IK90], the optimization cost can still be very large for large number of relations too. As such, we have restricted *NumRel* to at most 40 in our study. Therefore, if the number of relations is less than or equal to 40, the straightforward technique is used.

5.2 Workload

We experimented with acyclic multi-join queries only. For each query, the optimizers took a join graph along with statistics on relation cardinalities and join selectivities as their input. The data for a query was generated in two steps as in [SYT93]. First an acyclic join graph was generated, then relation cardinalities and join selectivities were assigned to the graph.

As in [SYT93], three relation types (small, medium, and large) were used. The cardinality of small, medium, and large relations were uniformly distributed over [10K, 20K], [100K, 200K], and [1M, 2M] records, respectively. Using these relation types, the algorithm assigns cardinalities and join selectivities as follows:

1. The type (small, medium, or large), but not the cardinality, of the final result was picked.

2. Each node (i.e., relation) R was randomly assigned a type (small, medium, or large) and then randomly assigned a cardinality in that type's range.

3. The join selectivity js of each edge (R_1, R_2) was chosen by first picking a value v for $\|R_1 \times R_2\|$ in $[0.5 \cdot min(\|R_1\|, \|R_2\|), (1.5 \cdot max(\|R_1\|, \|R_2\|)]$ and then solving $js \cdot (\|R_1 \times R_2\|) = v$

If the product of all the relation cardinalities and all the join selectivities (i.e., the cardinality of the final result) fell within the range of the final result type chosen in step 1), then the algorithm exited. Otherwise it backtracked to step 2), and tried new join selectivities. The algorithm restarted itself if it found that it had backtracked over 500 times. The calculation for

js reflects the fact that a join's size is often a function of its input relations. The multipliers of 0.5 and 1.5 were added to increase the variance in the size of intermediate results. All join selectivities were treated as independent.

Three sets of tests were conducted and are summarized in Table 1. For example, in *test-small*, 80% of the relations were small relations, 10% were medium and 10% were large.

Test Name	Ratio of relation sizes
test-small	80% Small, 10% Medium, 10% Large
test-large	10% Small, 10% Medium, 80% Large
test-mixed	33% Small, 34% Medium, 33% Large

Table 1: Types of tests

Because of the high optimization cost, in our experiments, we run 200 queries of 20-relation join, 50 of 40-relation join, and 20 of 60-relation, 80-relation and 100-relation join.

5.3 Experiment 1: Dynamic Programming Based Optimizer

This experiment studied the performance of the IDP optimizer that is based on dynamic programming. All the algorithms studied avoid cross products. We derived the following heuristics by varying the parameters as follows:[3]

BY-n: bushy tree with n as $NumRel$

DE-n: deep tree with n as $NumRel$

LD-n: left-deep tree with n as $NumRel$

We studied two values of n: 10 and 15. All the results (the processing cost and optimization cost) are scaled with respect to the processing cost of BY-15. Figure 4 shows the result for 40 relation joins. Looking at the results for test-small, we see that all the algorithms are relatively close in terms of processing cost. This is so because most of the relation sizes are small, and pipelining can be exploited without incurring excessive cost to write intermediate results back to disk. LD-10 which performed worst in terms of processing cost is, on average, about 50% worse than BY-15. However, the optimization cost for BY-15 is the highest while that for LD-10 is the lowest. In fact, the optimization cost for BY-15 is very much larger its processing cost. As a result, it turns out that the algorithm that performs best overall is DE-10. Though DE-10's processing cost is high, its low optimization cost is more than enough to outweigh the effect of the poor plans.

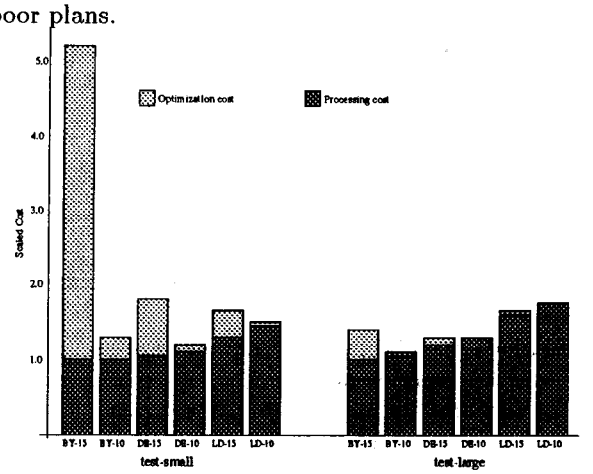

Figure 4: Comparison of IDP-based heuristics for 40-relation joins.

Turning to the results for test-large, we see a sightly different picture. First, the relative difference between the quality of the plans produced by the algorithms increases. This is because the relations are larger in size, and the effect of join orders become more important. Second, the ratio of optimization cost to processing cost decreases because of the high processing cost. As a result, the winning algorithm (in terms of overall cost) is now BY-10.

Both results pointed out two very interesting observations. First, no single algorithm dominates performance. While BY-15 generates the best quality plan, DE-10 performs best overall for test-small and BY-10 performs best overall for test-large. This implies that using only a single optimization algorithm (as is done in traditional query optimizer) is not sufficient – there is a need for adaptive query optimization. Second, it is not the algorithm that generates the best quality plan wins, but it is the "fitter" one (which may vary from query to query) that generates the best overall cost wins.

To further demonstrate the effectiveness of adaptive query optimization, we employ a simple tuning rule:

> Let M denote the available memory and S be the total size of the base relations. Then, if S < 5 M, employ DE-10, else if S < 15 M, employ BY-10 else employ BY-15.

This experiment was conducted with test-mixed and Table 2 summarizes the results. In the table, each entry represents the number of plans (in terms of percentage) that performed best overall. For example, BY-10 generated the best overall plans 59% of the time, while BY-15's high optimization lead it to perform best overall for only 8%. The tuned optimizer,

[3] We have not studied right-deep trees since the results in [SYT93] showed that the average performance of right-deep trees are the worst among all the four tree shapes.

however, performed best for 74% of the queries. The results clearly show that the ability to adapt is a crucial component in future optimizer. Even a simple rule as the one we have suffices to lead to good performance overall.

Tuned IDP	BY-15	BY-10	DE-10
74%	8%	59%	33%

Table 2: Results of tuning IDP optimizer on test-mixed.

To study the effect of the IDP optimizer on canned queries, we experimented with 10 queries from test-mixed (due to the high optimization cost). Figure 5 shows the results. Here we used only algorithm BY-15. The axis indicates the average number of times the optimizer has been invoked before a better plan was obtained. The first bar in the figure (when there is only 1 run) is the first run. The results of subsequent runs are scaled with respect to this run. In other words, a value of 0.9 implies that the cost of the plan is 90% that of the first run. We made two observations. First, the first run provides a good plan within a reasonable optimization cost. Second, unlike traditional optimizer, IDP can improve on plans of a canned query (though it may take a long time to do so). This is especially important since canned queries are executed repeatedly and a sub-optimal plan may degrade performance. Note that even a small percentage improvement could be a lot in terms of base time.

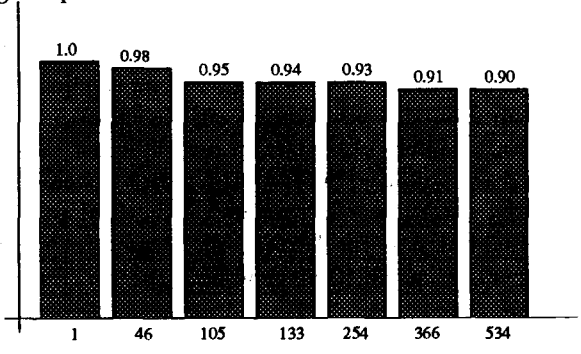

Figure 5: Plan refinements for 40-Relation Joins using BY-15.

5.4 Experiment 2: Randomized-Based Optimizer

This experiment studied the performance of the IRN optimizer that is based on 2PO. All the algorithms studied avoid cross products and uses bushy trees. We derived five heuristics by controlling the running time of the algorithms. This is achieved by varying the termination criteria of the 2PO algorithm (see Appendix II), i.e. we set k to 1, 2, 4, 8 and 16. We denote the derived algorithms as IRN-k, for k = 1, 2, 4, 8 and 16. All the results (the processing cost and optimization cost) are scaled with respect to the processing cost of IRN-16.

Figure 6 shows the result for 40 relation joins. Looking at the results for test-small (Figure 6(a)), as expected, we see that though IRN-16 produces the best quality plans, its high optimization cost lead it to produce the worst overall plans. Instead, we see that the heuristic IRN-1 that produces the worst plans performed best overall. Like its IDP counterpart, the main reason is because we are dealing with small relations. Turning to the results in Figure 6(b), we see that a higher optimization cost may be necessary for large relations to generate a better overall plan. As shown, the best algorithm is now IRN-8. It is interesting to note that IRN-16 remains the worst. Once again, the results demonstrate the need for an optimizer to adapt, especially for algorithms that have high variability in optimization cost.

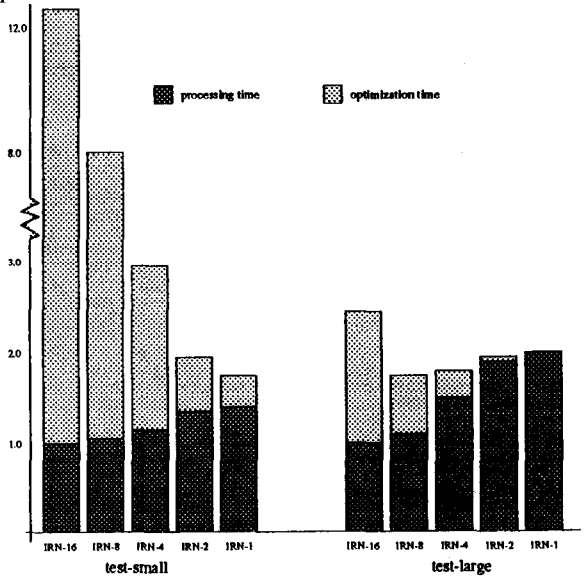

Figure 6: Comparison of IRN-based heuristics for 40-relation joins.

We also employ a simple tuning rule to study the effectiveness of adaptive query optimization:

> Let M denote the available memory and S be the total size of the base relations. Then, if S < 4 M, employ IRN-1, else if S < 10 M, employ IRN-2 else if S < 20 employ IRN-4, else employ IRN-8.

This experiment was conducted with test-mixed and Table 3 summarizes the results. As can be seen, the tuned optimizer performed best (in terms of the percentage of "optimal" overall cost). Several reasons account for the low percentage (only 63%). First, the

optimization cost for randomized algorithms is harder to predict. Second, the rule that we have adopted is a very simple one. However, the results clearly shows the need for adaptability in future optimizer.

Tuned IRN	IRN-1	IRN-2	IRN-4	IRN-8
63%	42%	0	23%	35%

Table 3: Results of tuning IRN optimizer on test-mixed.

Finally, we studied the effect of the IRN optimizer on canned queries. 10 queries from test-mixed were used. And we only considered algorithm IRN-8. Figure 7 shows the results. The axis indicates the average number of times the optimizer has been invoked. We see that IRN can improve on plans of a canned query by up to 20%.

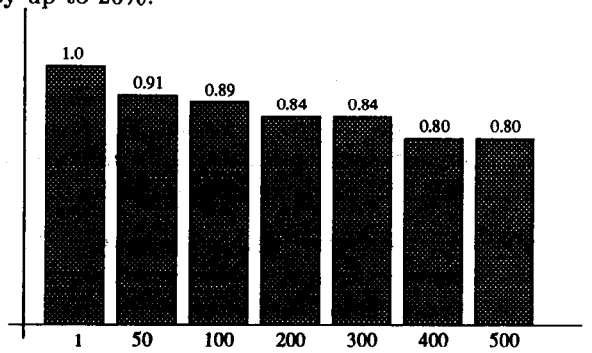

Figure 7: Plan refinements for 40-Relation Joins using IRN-8.

5.5 Experiment 3: Comparison of Optimizers

We also conducted a performance study between IDP and IRN optimizers to give us a flavor of the relative performance between the two optimizers. The following three heuristics were studied:

- IDP-BY-10 employs the IDP optimizer that considers a bushy search space with each iteration working on at most 10 relations
- IRN-8-20 employs the IRN optimizer that considers a bushy search space. In addition, k is set to 8, and the number of relations to be considered in each iteration is 20.
- IRN-8-40 is similar to IRN-8-20 except that the number of relations in each iteration is 40.

Before we present the results, we would like to remind the readers that the numbers of queries used are fairly small, and so the results should be seen as valid in the context of the queries (though we believe it should hold for larger number of queries).

Figure 8 shows the results of the experiments on the overall costs on test-mix. Clearly, none of the algorithms performed best. IDP-BY-10 performed best for queries involving 20 and 40 relations, IRN-8-20 performed best for queries involving 60 relations, and IRN-8-40 performed best for 80-relation and 100-relation join queries. For small number of relations, the optimization cost is a crucial determinant in the overall cost. For larger number of relations, the choice of an algorithm that produces good quality plans is more critical. Again, this result shows the importance of adapting an optimizer. In fact, more than one basic optimization algorithm (as in dynamic programming, randomization algorithms) should be supported.

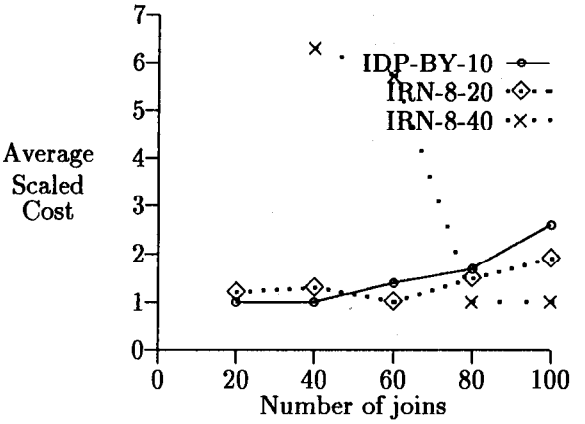

Figure 8: Comparison of IDP and IRN optimizers.

6 Conclusion

In this paper, we have described adaptive query optimization, a novel approach to optimizing large multi-join relational queries. Unlike traditional optimizers which are "programmed" to perform in a certain way, an adaptive query optimizer must be robust to the ever-changing environment to maximize the system performance. To realize adaptiveness, we designed an optimizer with two features. First, the search space of the optimizer can be tuned by parameters. Varying these parameters gives rise to a family of optimization heuristics of varying complexities. This allows the optimizer to pick the one that fits best during the optimization process. Second, the optimizer features a "learning" capability for canned queries such that repeated optimizations of a canned query may lead to different (and possibly better) plans than previous optimizations. This allows existing plans to be incrementally replaced by "fitter" ones. Our experimental study based on an analytical model demonstrated the necessity and effectiveness of such an approach.

We plan to extend this work in several directions. First, we want to develop a set of rules to tune the

optimizer automatically. Second, the proposed incremental optimization strategy has a limitation. For example, if *NumRel* is set to the number of relations in the query, it is not clear how and what information should be maintained to facilitate incremental optimization. Third, the number of spaces that must be searched remains exponential. Controlling this number without sacrificing the optimality of the solution is topic for further study. Finally, we have assumed that a query is optimized for a fixed set of run-time parameters such as buffer size. As argued in recent work [CG94, IN92], the actual buffer available at runtime may be different from that assumed by the optimizer during compilation/optimization. Previous work have sought to generate a set of plans, each targeting a different parameter setting [CG94, IN92]. At runtime, the most suitable plan is then picked and processed. The concept of incremental optimization can be applied here too. For example, for a given query, a set of k plans targeted at different buffer availability is generated and organized using an existing index (such as a B^+tree on the buffer size) to facilitate fast access. When the query is to be processed, new plans can be generated incrementally and replaced those less fit ones. We plan to look into this.

References

[CG94] R.L. Cole and G. Graefe. Optimization of dynamic query evaluation plans. In *1994 ACM-SIGMOD Conference*, pages 150–160, Minneapolis, Minnesota, May 1994.

[DKO+84] D. DeWitt, R. Katz, F. Olken, L. Shapiro, M. Stonebraker, and D. Wood. Implementation techniques for main memory database systems. In *1984 ACM-SIGMOD Conference*, Boston, NY, June 1984.

[GD87] G. Graefe and D.J. DeWitt. The exodus optimizer generator. In *1987 ACM-SIGMOD Conference*, May 1987.

[IK90] Y.E. Ioannidis and Y. Kang. Randomized algorithms for optimizing large join queries. In *1990 ACM-SIGMOD Conference*, pages 312–321, Atlantic, NJ, June 1990.

[IK91] Y.E. Ioannidis and Y. Kang. Left-deep vs bushy trees: An analysis of strategy spaces and its implications for query optimization. In *1991 ACM-SIGMOD Conference*, pages 168–177, Denver, Colorado, May 1991.

[IN92] Y.E. Ioannidis and R.T. Ng. Parametric query optimization. In *18th VLDB Conference*, pages 103–114, Vancouver, Canada, August 1992.

[Koo80] R.P. Kooi. The optimization of queries on relational databases. Technical Report Report No. CES-80-8, Case Western Reserve University, Cleveland, Ohio, September 1980.

[LV91] R.S.G. Lanzelotte and P. Valduriez. Extending the search strategy in a query optimizer. In *17th VLDB Conference*, Barcelona, Spain, September 1991.

[OL90] K. Ono and G. Lohman. Measuring the complexity of join enumeration in relational query optimization. In *16th VLDB Conference*, pages 314–324, Brisbane, Australia, August 1990.

[RR82] A. Rosenthal and D. Reiner. An architecture for query optimization. In *1982 ACM-SIGMOD Conference*, June 1982.

[SAC+79] P.G. Selinger, M.M. Astrahan, D.D. Chamberlin, R.A. Lorie, and T.G. Price. Access path selection in a relational database management system. In *1979 ACM-SIGMOD Conference*, pages 23–34, Boston, Massachusetts, June 1979.

[SD90] D.A. Schneider and D.J. DeWitt. Tradeoffs in processing complex join queries via hashing in multiprocessor database machines. In *16th VLDB Conference*, Brisbane, Australia, September 1990.

[SG88] A. Swami and A. Gupta. Optimization of large join queries. In *1988 ACM-SIGMOD Conference*, pages 8–17, Chicago, Ilinois, June 1988.

[Sha86] L. Shapiro. Join processing in database systems with large main memories. *ACM TODS*, 11(3):239–264, September 1986.

[SYT93] E.J. Shekita, H.C. Young, and K.L. Tan. Multi-join query optimization for symmetric multi-processors. In *19th VLDB Conference*, pages 479–492, Dublin, Ireland, August 1993.

[TL91] K.L. Tan and H. Lu. On the strategy space of multi-way join query optimization. *SIGMOD RECORD*, 20(4):81–82, December 1991.