Optimization of Dynamic Query Evaluation Plans

Richard L. Cole, University of Colorado at Boulder
Goetz Graefe, Portland State University

Abstract

Traditional query optimizers assume accurate knowledge of run-time parameters such as selectivities and resource availability during plan optimization, i.e., at compile-time. In reality, however, this assumption is often not justified. Therefore, the "static" plans produced by traditional optimizers may not be optimal for many of their actual run-time invocations. Instead, we propose a novel optimization model that assigns the bulk of the optimization effort to compile-time and delays carefully selected optimization decisions until run-time. Our previous work defined the run-time primitives, "dynamic plans" using "choose-plan" operators, for executing such delayed decisions, but did not solve the problem of constructing dynamic plans at compile-time. The present paper introduces techniques that solve this problem. Experience with a working prototype optimizer demonstrates (i) that the additional optimization and start-up overhead of dynamic plans compared to static plans is dominated by their advantage at run-time, (ii) that dynamic plans are as robust as the "brute-force" remedy of run-time optimization, i.e., dynamic plans maintain their optimality even if parameters change between compile-time and run-time, and (iii) that the start-up overhead of dynamic plans is significantly less than the time required for complete optimization at run-time. In other words, our proposed techniques are superior to both techniques considered to-date, namely compile-time optimization into a single static plan as well as run-time optimization. Finally, we believe that the concepts and technology described can be transferred to commercial query optimizers in order to improve the performance of embedded queries with host variables in the query predicate and to adapt to run-time system loads unpredictable at compile-time.

1. Introduction

Contrary to the assumptions underlying traditional query optimization, parameters important to cost computations can change between compile-time and run-time, e.g., set cardinality, predicate selectivity, available real memory, available disk bandwidth, available processing power, and the existence of associative search structures. The values of these parameters may vary over time because of changes in the database contents, database structures (e.g., indexes are created and destroyed), and the system environment (e.g., memory contention). Moreover, queries themselves may be incompletely specified, e.g., unbound predicates containing "user variables" in an SQL query embedded within an application program. Therefore, the execution of traditionally optimized, static query plans is often sub-optimal when cost-model parameters change, an important unsolved problem in database query optimization [Loh89].

SIGMOD 94- 5/94 Minneapolis, Minnesota, USA
© 1994 ACM 0-89791-639-5/94/0005..$3.50

In previous work, we identified run-time primitives that permit optimization decisions to be prepared at compile-time and evaluated at run-time [GrW89, Gra93]. This primitive was called the *choose-plan* operator. However, our previous work left two all-important questions unanswered, namely how to choose which optimization decisions to delay and how to engineer a query optimizer that efficiently creates dynamic plans for arbitrarily complex queries at compile-time. These two questions are addressed in the present paper, together with experiences gained from a working optimizer prototype.

The key concept in our research is *incomparability of costs* at compile-time. Contrary to traditional query optimization, we acknowledge that missing run-time bindings may render it impossible to calculate and compare the costs of alternative plans at compile-time. If so, alternative plans are only partially ordered by cost (instead of being totally ordered as in traditional query optimization) and we delay the choice between such plans until start-up-time. At start-up-time, when all run-time bindings have been instantiated, cost calculations and comparisons become feasible and the optimal plan can be chosen and evaluated. In other words, we presume in this paper that any compile-time ambiguity in selectivity and cost calculations can be resolved at start-up-time.

A number of sources may prevent accurate and reliable cost estimation and plan comparisons at compile-time. The three most important sources are errors in selectivity estimation [IoC91], unknown run-time bindings for host variables in embedded queries, and unpredictable availability of resources at run-time. While we are currently working to address the first problem, as will be discussed briefly in the final section, the present paper presents a solution for the last two problems. These two problems represent a large number of situations in real database applications, and we believe that the concepts and technology described here can be transferred to commercial query optimizers to improve the performance of embedded queries with host variables in the query predicate and to adapt to run-time system loads unpredictable at compile-time.

In order to validate and experiment with dynamic plan optimization, we extended the Volcano optimizer generator [GrM93] to permit partial orders of costs and have implemented a prototype relational query optimizer using the extended generator. The concept of incomparability is independent of the source of incomparability and of the data model of the generated optimizer. While the prototype is based on the relational data model, the problem of uncertain cost-model parameters and its solution using incomparable costs are also applicable to other data models that require query optimization based on estimation of uncertain parameters. In object-oriented database systems, for example, exact bindings of types (due to the use of subtypes), the existence of path indices, and the current state of object clustering can be modeled as run-time bindings and addressed effectively with our optimization techniques.

If two or more alternative plans are incomparable at compile-time, they are both included in the query evaluation plan and linked together by a *choose-plan* operator, thus creating a

dynamic plan as defined in our earlier research [GrW89, Gra93]. The choose-plan operator allows postponement of the choice among two or more equivalent, alternative plans until start-up-time, when the decision can be based on up-to-date knowledge, e.g., the bindings of user variables unbound at compile-time. Choose-plan operators may be used anywhere as well as multiple times to link alternative subplans in a query evaluation plan.

The overall effectiveness of our approach depends on several factors, namely the optimization time necessary to build dynamic plans, the size of these plans, the speed with which choose-plan decisions can be made at start-up-time, and plan robustness when parameter values change. Experimental analysis of a prototype optimizer permits the following conclusions:

- Plan size increases significantly, but not unreasonably in a start-up-time architecture that supports bringing the entire plan into memory using a small number of I/O requests. (Note that production database systems that support compile-time optimization typically already have such support.)
- The time required for choose-plan decisions is small, and the combined I/O and CPU overhead of starting a dynamic plan is significantly less than the corresponding overhead of completely optimizing queries at start-up-time.
- Finally, dynamic plans are highly robust when actual values of run-time parameter values differ from their values expected at compile-time.

The prime difference of our approach from other work in this area [Ant93, CAB93, DMP93, HaP88, HoS93, MHW90, OHM92] is the extension of compile-time dynamic programming with costs that cannot be compared until run-time. Much of the previous work has focused on developing heuristics applied at start-up-time; therefore, there has been no guarantee of plan optimality. The same is true for (compile-time) parametric query optimization based on randomized optimization algorithms [INS92]. Our approach is capable of guaranteeing plan optimality, while overhead at start-up-time is quite small because most of the optimization effort remains at compile-time.

Other work, in particular [GHK92], has considered dynamic programming for partially ordered plans, formalizing the concept of System R's "interesting orders" [SAC79]. In that model, however, the partial ordering is resolved into a total ordering later in the optimization phase, which does not address or solve the problem of run-time bindings for parameters relevant to the cost calculations. In our work, plans remain partially ordered even after compile-time optimization has completed, until run-time bindings are supplied. Thus, the present work goes beyond previous work and addresses a heretofore unsolved problem in database query optimization. In addition, the Volcano optimizer generator and its extensible architecture ensure that the approach is independent of the data model and the source of incomparability.

The rest of the paper is organized as follows. After we briefly discuss a motivating example and provide background information in Section 2, we describe the effects of incomparable costs on the search engine in Section 3 and then address issues of choose-plan decision procedures in Section 4. In Section 5, we detail our prototype dynamic plan optimizer, followed by an experimental analysis of its performance in Section 6. The final section summarizes our results and outlines future research directions.

2. Motivating Example and Background

To illustrate the benefits of dynamic plans over traditional, static plans, we present a very simple, motivating example. The example query's logical algebra definition and a dynamic plan are illustrated in Figure 1. Diagram (a) shows the logical expression, a Get-Set operator to retrieve data from permanent storage followed by a Select operator with a single predicate. If the predicate is unbound (i.e., it compares a database attribute with a user variable), the optimizer cannot accurately estimate its selectivity until it is bound at start-up-time; therefore, the optimizer cannot accurately determine and compare the costs of the two alternative implementations for this query, a file scan or an index scan. If very few records satisfy the predicate, even an unclustered B-tree scan is far superior to the file scan. The situation is reversed if many records qualify. If the selectivity is not known at compile-time, the choice among these alternative plans should be delayed until run-time using a dynamic plan. In (b) we see a dynamic plan using a Choose-Plan operator to link the two alternative plans for this query.

While this example is very limited in scope (in fact, some but certainly not all relational query optimizers consider dynamic choices among file and index scans for a single relation [Ant93, MHW90]), our optimization technique and prototype implementation are much more general; the creation of this simple dynamic plan is a trivial case. As an example that requires more general dynamic plans, consider a hash join of relations R and S. The size of S is predictable, while the join input from R can be very small or very large depending on a selection of R based on a user variable. Since hash joins perform much better if the smaller of the two inputs is used as the build input [Gra93], two join plans should be included in a dynamic plan for this query. A suitable dynamic plan for this query is shown in Figure 2; this plan can

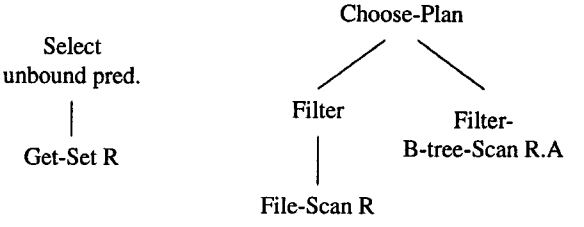

Figure 1. Alternative Implementations of Simple Selection.

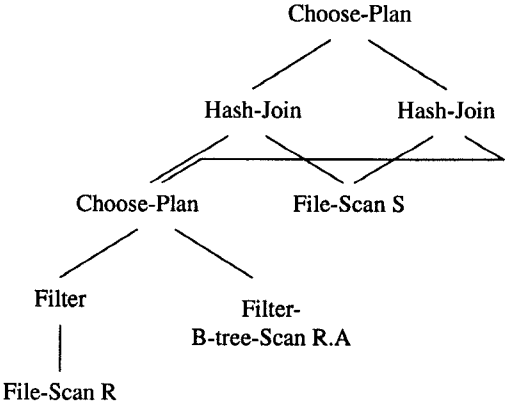

Figure 2. A More Complex Dynamic Plan.

switch among the scan methods and among the join orders at different selectivities. In our work, we consider alternative plans of arbitrary complexity. In other words, generating the dynamic plan of Figure 2 is another trivial case for our optimizer, which permits dynamic choices not only of individual algorithms but also of operator orderings and plan shapes.

Alternative Optimization Scenarios

Figure 3 shows three alternative optimization scenarios that address the problem of inaccurate estimates at compile-time. With traditional static plans, shown in the first horizontal line, optimization is performed once at compile-time. For each invocation at run-time, the plan is activated (including reading an access module and some I/O operations to verify that the plan is still feasible [CAK81]) and then executed. In Figure 3, the optimization time is labeled a, the time to activate a plan is b, and the run-time for a sequence of invocations is c_i.

If, however, a static plan's run-times vary widely due to suboptimal plan execution, as in Figure 3, it might be useful to optimize the query anew for each invocation. This is shown in the second horizontal line of Figure 3. (We assume in Figure 3 that the optimization time a is constant for all run-time bindings, which is realistic for most optimizers.) If a suitable plan can be chosen for each individual run-time invocation, the total run-time will be less than in the first scenario. Thus, the execution times will be $\forall i\ d_i \leq c_i$. Moreover, activation time can be avoided entirely by passing a plan directly from the optimizer to the execution engine. Unfortunately, this approach also implies repetitive and redundant optimization effort, which is troublesome because the most important performance issue is the effort at run-time, i.e., whether $N \times a + \sum_{i=1}^{N} d_i < N \times b + \sum_{i=1}^{N} c_i$, for some number N of query invocations.

A variation on run-time optimization is compile-time optimization with *conditional* run-time re-optimization. This idea was pioneered in System R for plans that have become infeasible [CAK81], but it has also been implemented for plans that have become suboptimal, e.g., in IBM's AS/400 [CAB93]. The problem with this approach is to detect suboptimality reliably. For example, not every change in the available set of indices, in file sizes, or in run-time parameters will render a pre-optimized plan suboptimal. In order to ensure that each query execution always uses the plan that is optimal for its current run-time bindings, however, systems using this approach typically perform many more re-optimizations than truly necessary. As an extreme example, if run-time situations alternate, each query invocation includes re-optimization, even if only two alternative plans are actually used.

The third alternative is the one advocated in the present paper since it avoids many of the problems identified for the other approaches. Optimization is performed only once, at compile-time. The plan produced at compile-time includes plan alternatives, and the actual plan to be executed will be chosen when the entire, dynamic plan is activated. While the optimization time and the plan activation time, tagged e and f in Figure 3, are longer than those for static plans, the total execution times of dynamic plans can be dramatically less than for static plans. In other words, we will argue and demonstrate in this paper that the plans executed using dynamic plans are as good as the ones chosen in run-time optimization, i.e., $\forall i\ g_i = d_i$, and that over many invocations, dynamic plans are more efficient than static plans, i.e., $e + N \times f + \sum_{i=1}^{N} g_i < a + N \times b + \sum_{i=1}^{N} c_i$, and more efficient than run-time optimization, i.e., $e + N \times f + \sum_{i=1}^{N} g_i < N \times a + \sum_{i=1}^{N} d_i$.

Volcano Optimizer Generator

In order to integrate our work on dynamic query evaluation plans with our other research into database query optimization and execution, our work is based on an algebraic transformation model as implemented in the Volcano optimizer generator [GrM93]. The Volcano optimizer generator is an extensible, data model independent tool for creating efficient algebraic query optimizers and has been used in several optimizer prototypes [BMG93, WoG93]. The optimizer generator's modularized components support several optimization abstractions including logical and physical algebra operators, logical and physical properties, a cost model, and a search engine. The logical algebra describes a query as input to the optimizer, while the physical algebra describes the algorithms implemented in the database execution engine. Alternative logical algebra expressions are considered by the optimizer through the application of logical transformation rules, e.g., join associativity and commutativity. The transformation from logical to physical algebra is defined by implementation rules, e.g., join may be implemented by merge-join.

Associated with the logical and physical algebras are abstract data types for logical and physical properties. Logical properties, such as a relation's schema or its cardinality, define logical attributes of the data sets produced by logical algebra operators. Analogously, physical properties such as sort order and

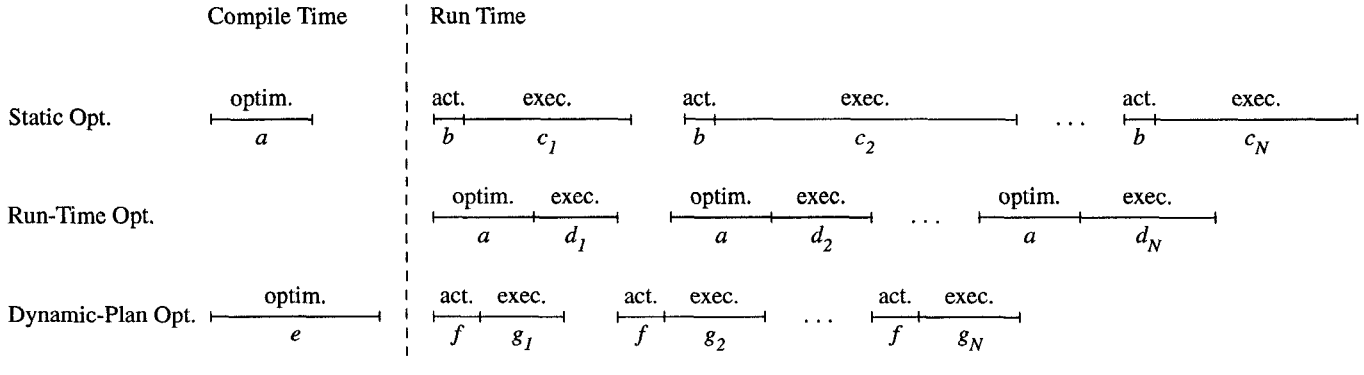

Figure 3. Alternative Optimization Scenarios.

cost define attributes of the physical algebra. While physical properties can be delivered by particular physical algorithms, e.g., merge-join delivers a sort order, they are also delivered by "enforcers," e.g., sort. Enforcers are physical algorithms without equivalent logical operators, instead being associated with the properties they enforce. The application of implementation rules during the search process is guided by considerations of physical properties. An optimization goal is the combination of a logical algebra expression and the desired physical properties, which are a generalization of "interesting orderings" addressed in System R and many other database systems [SAC79].

The Volcano optimizer generator's search engine uses a top-down, memoizing variant of dynamic programming to search the plan space [CLR89], which is a refinement of the bottom-up variant used in traditional optimizers based on the techniques of System R [SAC79]. Therefore, certain sub-problems need not be optimized (depending on the requested physical properties). In addition, the search algorithm prunes the search space using the cost of current feasible plans as upper bounds for new plans. Such branch-and-bound pruning is not a heuristic; it still guarantees that dynamic programming finds the best plan given the cost model and supplied cost-model parameter values.

3. Search and Cost Incomparability

Traditional optimizers have a cost model in which all feasible plans are totally ordered by cost. The dynamic programming model used in the System R optimizer [SAC79] and most other optimizers require that it always be possible to determine which of two equivalent plans is more expensive and therefore can be ignored in further optimization. The cost model of our dynamic plan optimizer and its search engine do not require a total ordering. This is one essential difference between traditional optimization and dynamic plan optimization.

Other researchers have made forays into query optimization using dynamic programming for partially ordered plans, e.g., optimization for first-item or last-item response time [GHK92]. In those models, however, cost incomparability affects only optimization. In other words, all uncertainty is resolved at compile-time, before the optimizer terminates. Our problem is very different, since we address parameter bindings that are not known until run-time. Thus, while nearly all previous optimizers produce fully determined or static query evaluation plans, our optimizer delays some selected optimization decisions until run-time, creating a dynamic plan.

In parametric query optimization [INS92], the problem of unknown parameter bindings has been addressed, but the approach is not based on incomparable costs and the encapsulation of uncertainty within cost. Instead, every possible combination of uncertain parameter values is treated as a separate optimization problem. For example, if each of three cost model parameters can have 10 different values, 1,000 optimizations have to be performed. In order to deal with this combinatorial explosion, randomized search algorithms including side-ways-information-passing are applied in parametric query optimization [INS92]. Unfortunately, randomized algorithms cannot guarantee plan optimality, and part of the goal of parametric query optimization seems defeated.

Extensibility and Generality of Approach

Let us presume that a query optimizer encapsulates cost in an abstract data type, as is done in the Volcano optimizer generator as well as many other optimizers. The functions on this abstract data type include addition, subtraction, comparison, and the cost functions associated with the query processing algorithms. Earlier optimizers require that the cost comparison function return one of the values "greater than," "less than," and "equal to." In support of incomparable costs, the cost comparison function may return "incomparable" in addition to the standard values. The reason for incomparability can be one of many; in order to preserve extensibility, we do not want to assume one reason or another.

If the anticipated query evaluation cost is encapsulated in an abstract data type, the cost model is completely in the control of the database implementor (DBI). Thus, cost can be defined to be the response time for the first, N^{th}, or last result item, CPU, I/O, and network time and effort (time and effort can differ due to parallelism), total resource usage, a time-space product of required memory usage or concurrency locks, a combination of the above, or some other performance measure. If cost is a combination of multiple measures, which are not simply added into a single value such as the weighted sum used in System R [SAC79], cost values are partially, not totally, ordered and costs may therefore be incomparable at compile-time. Our optimization model supports such partially ordered cost measures, although we currently require that two alternative plans' costs be comparable either at compile-time or at start-up-time. As mentioned in the introduction, we are currently working on a query optimization model and prototype that permits delaying optimization decisions beyond start-up-time into run-time.

As a concrete example, our experimental prototype models cost (as well as selectivity, cardinality, and available memory) as an interval rather than a single (expected) value. Thus, a cost value in this model captures the entire range in which the actual cost during query evaluation may fall. If two intervals overlap, it is impossible to claim that one plan is always better than the other; therefore, two overlapping cost intervals are declared incomparable.[1]

Guarantees of Optimality

While delaying some decisions from compile-time to run-time creates a more flexible plan, it may not guarantee that the actual execution plans included in a dynamic plan and chosen at start-up-time are as good as the plans chosen by a run-time optimizer. In the notation of Figure 3: can we guarantee that $\forall i\ g_i = d_i$? The essential consideration focuses on cost comparisons. If truly *all* cost comparisons are declared incomparable, the resulting dynamic plan will include absolutely all possible plans for a query. We call this dynamic plan the "exhaustive plan." Because it includes all plans, it must also include the optimal one for each set of run-time bindings. Presuming that the cost comparisons at start-up-time are correct,[2] the exhaustive plan is optimal for all run-time bindings, and we are assured that $\forall i\ g_i = d_i$.

---

[1] This is merely one (rather simple) way to model uncertain cost model parameters and incomparable costs. The database implementor is free to choose an alternative selectivity and cost model.

[2] In this paper, we deal with unbound cost model parameters. Inaccuracy in the cost functions is a separate issue that has been addressed elsewhere, e.g., by Mackert and Lohman [MaL89]. Any query optimization can only be as good as the cost functions; if the cost formulas do not map file sizes and other cost model parameters correctly to costs, query optimization at compile-time and branch-and-bound pruning cannot possibly work.

In our approach, we do not advocate exhaustive plans. Instead, we only delay those cost comparisons and plan choices from compile-time to run-time that depend on parameters unbound at compile-time. Thus, the decisions made at compile-time do not depend on run-time bindings and their outcomes do not differ whether they are made at compile-time or at run-time. Plans not included in a dynamic plan cannot possibly be optimal for any run-time binding of the unknown parameters, and a dynamic plan is guaranteed to include all potentially optimal plans for all run-time bindings. Thus, a dynamic-plan optimizer can guarantee that a produced dynamic plan is optimally adaptable to all run-time bindings, and we are assured that $\forall i \; g_i = d_i$.

Modifications to Plan Search

Although cost comparisons are performed by DBI-defined functions, the ramifications of incomparable costs must be handled by the search engine. Such plans are linked together using a choose-plan operator, which incorporates a decision procedure to choose among alternative plans at start-up-time using up-to-date knowledge. Thus, there may be more than a single plan for a given combination of a logical algebra expression and desirable physical properties, and it is impossible to prune all but one of them, as is the assumption and foundation of most database query optimizers. Inserting a choose-plan operator into the query evaluation plan's tree creates a dynamic plan, i.e., a plan that is completely generated at compile-time but adapts at start-up-time to changes in cost-relevant parameters.

Since a choose-plan operator and a dynamic plan can be part of a larger plan, the optimizer must be able to calculate their costs. Intuitively, the cost of a dynamic plan is the cost of the decision procedure in the choose-plan operator plus the minimum cost of the equivalent, alternative subplans. Since a choose-plan operator always chooses its cheapest input plan, this minimum reflects the minimal cost for each possible run-time binding. For example, if cost is modeled as minimum-maximum intervals as in our optimizer prototype, the cost of a dynamic plan with two equivalent, alternative subplans is an interval ranging from the smaller of the two minimum costs to the smaller of the two maximum costs. In other words, in the best case, the cost of the dynamic plan is the lower one of the two best-case costs of the two alternative plans, and in the worst case, the cost of the dynamic plan is the lower one of the two worst-case costs (plus the decision cost).

An important concern of partial plan ordering is its effect on search complexity. The worst case complexity of search based on dynamic programming using a total ordering of plans is known to increase exponentially with the number of join operators [OnL90]. An accurate analysis of the effects of partial orders is not possible without an understanding of the dependencies between variables contributing to incomparable costs, but our experimental evaluation demonstrates that robust plans can be optimized for a significant, but reasonable time increase.

In our experiments, reported later in this paper, we observed a marked increase in optimization time, largely due to the fact that branch-and-bound pruning is less effective when cost is modeled as an interval and only the minimum cost can be subtracted when computing bounds. For example, a traditional optimizer can use the expected cost of a known alternative plan as a cost bound. When optimizing the two inputs of a join with a cost bound, it can use the expected cost of the first input to reduce the cost bound for optimizing the second input. Our optimizer, on the other hand, because cost is modeled as an interval, can only use the maximum cost of a known alternative plan as cost bound, and can stop optimizing the second input only when the two inputs' minimum costs together exceed the bound. Clearly, this restriction severely erodes the effectiveness of branch-and-bound pruning. We will come back to this issue when we discuss optimization times in the experimental section.

Techniques to Reduce the Search Effort

Fortunately, there are a number of considerations that reduce the search effort in dynamic plan optimization. The most important among them is sharing of subplans: in our experience, alternative plans often include large common subexpressions. These subplans need to be optimized only once, and their cost can then be used in the cost computation for multiple alternative plans. Notice that the size of access modules representing a dynamic plan typically does not grow exponentially with the complexity of the query in spite of the fact that the number of possible plans grows exponentially [OnL90]. To limit the growth of search effort and access module, all plans and alternative plans must be represented as directed acyclic graphs (DAGs) with common subexpressions, not as trees. Thus, subplans are shared, and the exponential number of combinations in an exhaustive plan is captured by many points at which sharing occurs rather than an exponential number of operators in the plan. The representation of dynamic plans as DAGs also reduces the time to read and activate an access module, as will be discussed in the following section.

Depending on the nature of the cost model that induces cost incomparability and dynamic plans, there may be many situations in which costs seem incomparable but actually are not. We have identified two such types of situations. First, two plans' costs are consistently equal. For example, except in rare circumstances (lots of duplicates), it does not matter which of two merge-join inputs serves as outer and inner input. In those cases, it would be acceptable to make an arbitrary decision.

Second, two plans' costs are similar, but one plan is actually consistently cheaper than the other. In that case, it would be perfectly acceptable to choose the plan that consistently outperforms the other one. However, depending on the detail with which costs are modeled, the two plans' costs may be incomparable during compile-time, and both plans are retained as alternatives of a choose-plan operator, although at run-time the choose-plan operator will always choose one plan and never the other.

These situations could be avoided by an analytical comparison of the two cost functions. Attempting such a deep analysis of cost functions is, unfortunately, not realistic, in particular for an extensible database system in which one does not want to pose any restriction on the form and complexity of cost functions. Thus, analytical comparisons of cost functions is not a viable solution for this problem.

A more realistic, though heuristic, approach is to evaluate the cost function for a number of possible parameter values and to surmise that if one plan is estimated more expensive than the other for all these parameter values, it is always the more expensive plan and therefore can be dropped from further consideration. This solution requires that the DBI implement complex cost comparisons with multiple parameter settings. Nonetheless, it guarantees optimal plans only inasmuch as the DBI can guarantee that cost comparisons are correct. If two plans are actually both optimal for different bindings but the cost comparison does not detect the fact, the optimizer will not find the optimal dynamic plan.

In our prototype, both types of situations discussed above are handled in the most naive manner. In other words, equal-cost plans such as two merge-joins of the same inputs are both included in a resulting dynamic plan, and the costs of alternative

plans are declared incomparable even if one of the two plans actually is consistently better than the other. The reason for this decision is to present our techniques in the most conservative way: if our optimizer prototype can work effectively and outperform static plans as well as run-time optimization without these optimizations, it certainly would do so if these two optimizations were implemented.

4. Start-Up-Time Considerations

As part of the formulation of dynamic plans, we must consider the form of choose-plan decision procedures and their evaluation at start-up-time. For best start-up-time performance, we should attempt to minimize the size of decision procedures by only including the logic necessary to differentiate between alternative plans. For example, if the costs of two alternative plans are incomparable due to an unbound predicate's unknown selectivity, only an updated selectivity is necessary for the decision at start-up-time (one of the proposed approaches in our earlier research [GrW89]). However, it is not easy to build these minimal decision procedures since it requires building inverses for all cost functions.[3] While cost functions are typically fairly simple, they can sometimes be very complex, in particular in object-oriented database systems [KGM91] or if detailed buffer effects are taken into consideration [MaL89]. Moreover, since each cost function is typically a function of many variables such as input cardinalities, selectivity, and resource allocation, multiple inverse functions would be required. In an extensible system, we consider it entirely unrealistic to assume that inverses of cost functions can be provided. Thus, we needed to identify and use a different mechanism for decision procedures in choose-plan operators.

Fortunately, it is not really necessary to minimize the decision procedures and to invert the cost functions. A much simpler approach is to re-evaluate the cost functions associated with the participating alternative plans. The decision procedure is now merely a cost comparison of the plan alternatives with run-time bindings instantiated; thus, the reasons for incomparability of costs at compile-time have vanished. Compared to performing the actual data retrieval and manipulation, evaluating the cost functions takes only a small amount of time. Our experimental results demonstrate that using the original cost functions of the alternative plans contributes only relatively small overhead at start-up-time. Comparing dynamic plans with run-time optimization, re-evaluating the cost functions is much faster than optimizing a query, because optimization requires many more cost calculations in addition to the actual plan search.

Another consideration is the means by which new and updated cost-model parameter values are obtained at start-up-time. Typically, these values are user variables that are passed to a dynamic plan in exactly the same way as to a traditional static plan. In the worst case, these new values require a very small number of system calls or catalog lookups, and again should go unnoticed compared to a query's execution. Also note that the start-up-time effort for a dynamic plan correlates with the complexity of the query, another reason to expect start-up-time overhead to be relatively small relative to a query's execution time.

[3] At best, if the database system were implemented in a programming language in which functions are objects that can be manipulated and modified at run-time (e.g., Lisp or ML), cost functions could be "curried" with the bindings known at compile-time.

In order to reduce the CPU effort, the start-up procedure for dynamic plans can employ two techniques that are well-known to be very effective in optimization, namely sharing of subplans and branch-and-bound pruning. Obviously, if a dynamic plan is represented as a DAG, not as a tree, and if the cost of each subplan is evaluated only once, not as many times as the subplan participates in some larger plan, the start-up-time is much smaller. Moreover, if the cost of one plan has already been computed, it can be used as a bound when computing the cost of an alternative plan; if the cost computation exceeds the bound, cost calculation can be aborted since the alternative plan cannot possibly be less expensive than the plan that created the bound. In our experiments, we represented dynamic plans as DAGs and computed each subplan's cost only once; however, for simplicity, we did not implement branch-and-bound pruning at start-up-time.

Finally, since dynamic plans contain more nodes than static plans, both more data manipulation operators and choose-plan nodes, the I/O cost to read a dynamic-plan access module into memory is larger than the I/O cost for reading a static-plan access module. We presume that either type of plan is in contiguous disk locations; thus, in our performance comparison, we need only consider their difference in transfer times when comparing the start-up-times of static and dynamic plans.

In order to reduce both I/O and CPU effort at start-up-time, we propose a heuristic technique that shrinks dynamic plans over time. During each invocation, the access module keeps statistics indicating which components of the dynamic plan were actually used. After a number of invocations, say 100, the access module analyses which components have been used and replaces itself with a dynamic-plan access module that contains only those components that have been used before. This self-replacement by an access module is similar to that used to execute queries whose compile-time-generated plans have become infeasible [CAK81]; the main difference is that infeasible plans require re-optimization whereas our shrinking heuristic only requires effort comparable to the cost analysis at start-up-time. On the one hand, this technique will eliminate components from a dynamic plan that are never chosen. On the other hand, it is a heuristic technique because it may remove dynamic-plan choices that have not been used in the first invocations but would actually be used, if available, in later runs. We leave an analysis of this technique to later research.

5. Optimizer Prototype

In this section, we present a prototype optimizer that produces dynamic query evaluation plans, including the optimizer's logical algebra, physical algebra, and cost model. Since the objective of this research is optimization of dynamic plans, not data modeling or extensible query processing models, the algebras in this prototype define a basic relational data model and typical execution algorithms. However, our extensions to the search engine for cost incomparability do not depend on these particular definitions; other data models and query algebras are also possible. This observation is equally applicable to the particular cost model we will define; other cost definitions based on different sources of incomparability are possible.

Logical operators and physical algorithms are listed in Table 1. Logical transformations include join commutativity and associativity. The transformation rules permit generation of all "bushy trees," not only the "left-deep trees" of traditional optimizers. The prototype includes two enforcers, also listed in Table 1, one for sort order and one for plan robustness. Plan robustness is the property enforced by choose-plan.

Operator Type	Logical Algebra Operator or Physical Property	Corresponding Physical Algebra Algorithm
Data Retrieval	Get-Set	File-Scan B-tree-Scan
Select, Project	Select	Filter Filter-B-tree-Scan
Join	Join	Hash-Join Merge-Join Index-Join
Enforcer	Sort Order	Sort
	Plan Robustness	Choose-Plan

Table 1. Logical and Physical Algebra Operators.

Like traditional optimizers, our prototype optimizer computes costs from the cardinality of a query's input relations, selectivity of predicates, and availability of resources (e.g., available memory). But unlike traditional optimizers, cost is not restricted to a single point or value. Instead, it is extended from a point to an interval, defined by upper and lower bounds, i.e., [*lower-bound, upper-bound*]. Thus, this cost model acknowledges the inherent error in selectivity and cost estimation [Chr84, IoC91], although only in a somewhat crude form. Comparing the costs of two plans requires a comparison of their cost intervals. Within these intervals, we do not pretend to know precisely (at compile-time) what the cost (at start-up-time) will be. Therefore, when intervals *overlap* one another, we cannot determine whether one is greater or less than the other.

The upper and lower bounds of the cost intervals are computed using traditional cost formulas supplied with the appropriate upper and lower bound values for the parameters of the cost model (e.g., selectivity bounds of [0, 1] for an unbound predicate) and assuming that cost functions are monotonic in all their arguments (input file sizes and available memory). Bounds of the parameters are computed using a parameter's minimum and maximum.

The costs of a choose-plan operator, a dynamic plan, or a subplan are, of course, also intervals in this model. Given two alternative plans, the cost of a dynamic subplan is the minimum of the corresponding values of the alternative plans, plus the evaluation overhead of the choose-plan operator. For example, if two alternative plans have costs [0, 10] and [1, 1] and if a choose-plan introduces an overhead of [0.01, 0.01], the combined plan including the choose-plan operator linking the equivalent alternative plans has the cost [0.01, 1.01]. The addition of two costs simply adds together the respective lower and upper bounds of the costs, but subtracting costs (used to maintain bounds in branch-and-bound pruning) only subtracts the lower-bound, since we can only be sure that the lower-bound cost will be "used up." As mentioned before, subtracting only the lower bound from an overall cost limit has a serious effect on branch-and-bound pruning, as we will see in the next section.

6. Experimental Evaluation

As a preliminary experimental evaluation of dynamic plans, we optimized five queries of increasing complexity and analyzed their average run time, optimization time, start-up time, and number of operator nodes in a plan. The queries were optimized using traditional optimization based on expected value parameters, i.e., with costs as points represented by intervals [*expected-value, expected-value*], and with dynamic-plan-optimization enabled, i.e., with cost intervals [*domain-minimum, domain-maximum*].

It can easily be argued that optimizing only a few queries does not truly reflect the "average" effectiveness (if such a measure can be defined) of dynamic plans relative to static plans or run-time optimization. However, that is not our point here, since it is clear from the motivating examples (e.g., file scan vs. index scan) that dynamic plans can be much faster than static plans. Instead, the purpose of these experiments is to demonstrate that (i) the overhead of dynamic plan optimization is tolerable, even for complex queries (e.g., a 10-way join) and a substantial number of unbound variables (e.g., 10); (ii) while the start-up-time effort is substantial, it is small compared to the possible performance penalty of running static plans only; and (iii) the start-up time of dynamic plans optimized at compile-time is much shorter than the time required for optimization at run-time. In other words, we do not advocate to use dynamic plans at all times and for all queries. However, they are an extremely useful technology in those cases where they apply. We plan on characterizing those cases more thoroughly in the future.

Definitions of the five queries and the number of logical alternative plans (based on the transformation rules) considered by the optimizer's search engine were as follows: query 1 — a query of a single relation with a single predicate (see the earlier motivating example) having one logical algebra alternative (with three different physical algebra expressions); query 2 — a two-way join with two selections (each referring to a different input relation) and 2 logical alternatives; query 3 — a four-way join with four selections and 48 alternatives; query 4 — a six-way join with six selections and 2,432 alternatives; and query 5 — a ten-way join with ten selections and 74,022,912 logical alternatives. In all cases, selectivities of the selection predicates were presumed uncertain, while the join predicates' selectivities were computed using the cross product of the joined relations divided by the larger of the join attribute domain sizes. Thus, query 5 had a total of 10 uncertain cost model parameters based on selectivity. (Although not part of a query's definition, we also considered the effects of uncertain available memory, thus introducing an additional uncertain cost model parameter.)

These five queries were optimized using traditional query optimization at compile-time (resulting in a static query evaluation plan), optimization at run-time, and dynamic plan optimization. For each query, we compare query execution time, optimization time, query plan size, and start-up-time, which are defined as follows. Optimization time, times a and e in Figure 3, is the CPU-time required to optimize and build the query evaluation plan. The reported times are truly measured with our prototype. Start-up-times, times b and f in Figure 3, include I/O and CPU times: I/O effort is required to read the access module and to validate a plan against the current catalogs, and CPU effort is required in dynamic plans to evaluate all choose-plan decision procedures and to choose among alternative plans. I/O costs to read an access module were derived from the plan sizes, which is a count of operator nodes in the directed acyclic graph (DAG), i.e., in the physical representation of the plan. The CPU effort for dynamic-plan-start-up was also measured using our prototype. Since both measured CPU-time and estimated I/O-time are in seconds, we can add them to obtain total start-up-times. Average run-times for static and dynamic plans were determined using $N = 100$ sets of randomly generated values for the uncertain cost-model parameters. The execution times reported are those

predicted by the optimizer.[4] The CPU times for optimization and for dynamic-plan start-up, however, are truly measured. Measured CPU times refer to a dedicated DECstation 5000/125 with 32 MB of memory.

The random values for selectivities of selection operations are chosen from a uniform distribution over the interval [0, 1]. In static-plan query optimization, we used an expected value of 0.05. Note that traditional optimizers often use a small default selectivity for selection predicates, represented here by the value 0.05. Attribute domain sizes varied from 0.2 to 1.25 times the respective relation's cardinality. The expected memory size was 64 pages of 2,048 bytes. When memory was considered an unbound parameter, a run-time value for the number of pages was chosen from a uniform distribution over [16, 112].

The number of records in each relation varied from 100 to 1,000. Using only such a small range of relation cardinalities increases the number of cases in which cost intervals will overlap, thus inducing cost incomparability and pushing our dynamic plan optimizer towards its worst case, i.e., forcing it to generate exhaustive plans. All relations had a record length of 512 bytes. Attributes referenced by the unbound selection predicates as well as all join attributes had unclustered B-tree structures suitable for predicate evaluation. All reported times, both measured (e.g., start-up CPU effort) and estimated (e.g., start-up I/O effort), are given in seconds.

Each of the following diagrams shows four curves. Each curve has five data points, which correspond to the five queries. Curves drawn with solid lines represent the times for traditionally optimized, static query evaluation plans. Curves drawn with dashed lines represent the times of dynamic plans. Curves indicated by o's show uncertainty in the selectivity of query selection predicates; the number of uncertain variables varies from 1 for query 1 (one selection predicate) to 10 for query 5 (10 predicates). Curves indicated by □'s add uncertainty in available memory. Because memory is one more uncertain parameter for each query, these curves are shifted to the right by one in the number of uncertain variables.

First, we compare execution times of static and dynamic plans in Figure 4. In the notation of Figure 3, this diagram quantifies the extent to which $\sum_{i=1}^{N} g_i < \sum_{i=1}^{N} c_i$ by reporting their averages $\bar{g}$ and $\bar{c}$. The x-axis of Figure 4 is the number of uncertain variables as defined by the five experimental queries. For each data point in Figure 4, $N = 100$ sets of random values for the unbound parameters were chosen and execution costs of the optimized static and dynamic plans were determined with each of these sets of values. Obviously, the static plans are not competitive with their equivalent dynamic plans. The performance difference varies between a factor of 5 for query 1 to a factor of 24 for query 5, the most complex query. In absolute numbers, the average run time for query 5 improved from

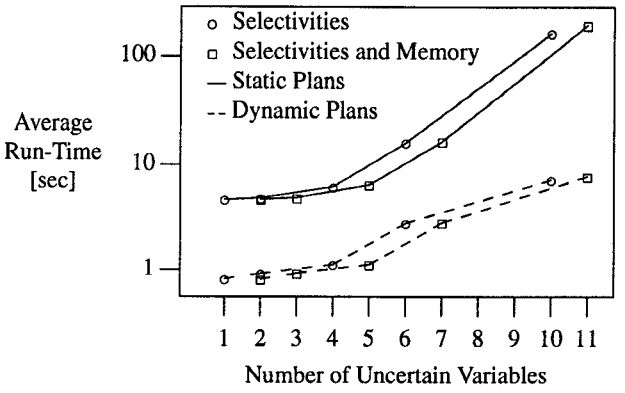

Figure 4. Execution Times of Static and Dynamic Plans.

$\bar{c} = 194.1$ sec to $\bar{g} = 7.8$ sec. Thus, all dynamic plans exhibit vastly improved robustness and performance over the static plans. Notice in particular that the performance advantage of dynamic plans increases with the number of uncertain variables. The additional uncertainty of available memory accentuates the difference between static and dynamic plans even further.

The main cost of dynamic plan optimization is its optimization time. Figure 5 shows the times for exhaustive, bushy-tree optimization generating static and dynamic plans, or times *a* and *e* in Figure 3. For any query, the worst increase in optimization times is less than a factor of 3, 27.1 sec versus 80.6 sec for query 5. This difference is primarily due to the reduced effectiveness of branch-and-bound pruning, as discussed earlier. (Conversely, this experiment demonstrates the effectiveness of branch-and-bound pruning for traditional query optimization using the Volcano optimizer generator's search strategy.) Notice that this increase is paid during compile-time, i.e., only once, and that the dynamic plan contains every potentially optimal plan; therefore, optimality can be guaranteed. Uncertain available memory adds little or no additional optimization time; the effects of uncertain memory size are overshadowed by those of uncertain selectivities.

The start-up-time of dynamic plans, times *b* and *f* in Figure 3, is the CPU time for decisions based on cost function evaluations and increased I/O effort due to the increased size of

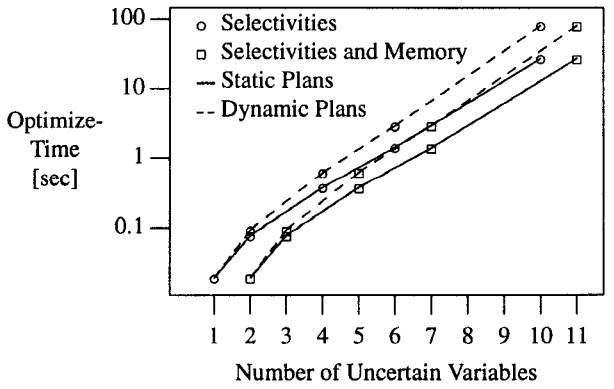

Figure 5. Optimization Time for Static and Dynamic Plans.

[4] We propose and use the following definitions: An optimizer's *efficiency* measures the time to produce a query evaluation plan. Its *effectiveness* measures the quality of the plan, i.e., execution cost, and its *accuracy* measures how well it models resource use. Optimizer accuracy depends on the statistical descriptions of operator inputs (selectivity estimation) and on the mapping of such descriptions and algorithms to cost. In turn, optimizer effectiveness is a combination of optimizer accuracy, search space, and search strategy. In order to evaluate a search strategy without the distorting influences of selectivity estimation and cost calculation, plans should be compared on the basis of anticipated execution costs, not of real execution costs.

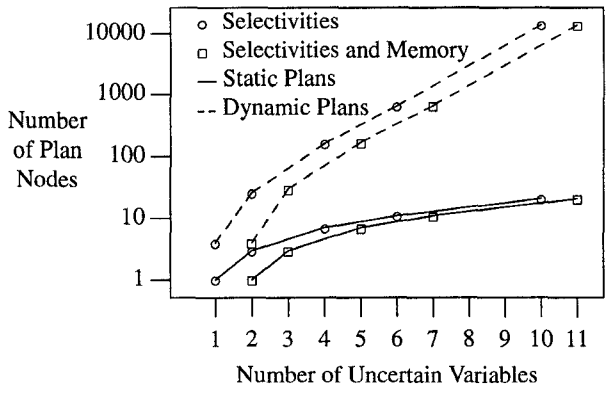

Figure 6. Plan Sizes for Static and Dynamic Plans.

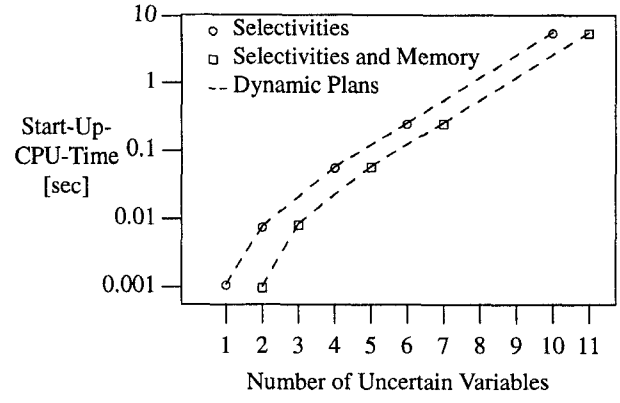

Figure 7. Start-Up-Times for Dynamic Plans, CPU Only.

the plan itself. The latter can be computed from the number of operator nodes in the optimized plan and therefore in the access module. Figure 6 shows the number of operator nodes in static and dynamic plans. As the complexity of the queries increases, the number of nodes also increases. For query 5, which has 11 uncertain variables (10 simple predicates and the size of memory), the difference in plan size is 14,090 versus 21 operator nodes. While this difference might seem alarming, it is important to note that this represents a rather extreme case, a query with as many as 11 uncertain variables. Moreover, we will show that the absolute size of even our largest dynamic plan is acceptable.

As can also be seen in Figure 6, making the amount of available memory uncertain only barely increases the sizes of the dynamic plans. This lack of effect is an argument for believing that there are a limited number of potentially optimal plans, and that additional uncertain variables will not increase the size of the dynamic plans significantly beyond that point. It depends on a number of factors whether or not the dynamic plan including all these potentially optimal plans is, in fact, the exhaustive plan; the much more important issue, however, is the extra effort required to read and start an access module containing a dynamic plan compared to those times for a static plan.

In order to compute the time to read an access module from disk, the number of operator nodes in a plan must be multiplied with the node size and divided by the disk bandwidth. For a node size of 128 bytes and a bandwidth of 2 MB/sec, about 16,000 nodes can be read per second. Thus, the run-time improvement from from 194.1 sec to 7.8 sec discussed above requires additional I/O for the access module of less than 0.9 sec. Catalog validation and one seek operation to read the access module are equal, because both approaches use compile-time optimization. In the following, we will presume that this time is $z = 0.1$ sec, where b and z are almost identical due to the small size of static-plan access modules.

The other component of the start-up-time is the CPU-time for evaluating the decisions in choose-plan nodes, which is shown in Figure 7. Not surprisingly, the increase in start-up CPU time introduced by dynamic plans almost exactly parallels the increase in plan size. More important, however, is the fact that choose-plan decisions can be made quite rapidly: for the most complex dynamic plan in Figure 7, the CPU effort at start-up-time is 5.8 sec, in spite of the fact that a cost function must be evaluated for each node in the dynamic plan. Notice that the dynamic plan is stored as a DAG, not as a tree, and that the cost of shared subexpressions is computed only once, not once for each usage.

Thus, the execution time improvement from $\bar{c} = 194.1$ sec to $\bar{g} = 7.8$ sec required additional start-up effort (CPU + I/O) of only about 7 sec, resulting in an overall improvement in run-time effort from $b + \bar{c} = 0.1 + 194.1 = 194.2$ sec to $f + \bar{g} = 0.1 + 0.9 + 5.8 + 7.8 = 14.6$ sec, or an overall improvement in run-time performance by a factor of 13. In other words, for the most complex query requiring the largest, most complex evaluation plan, the dynamic plan is significantly faster than the static plan.

Given the substantial execution-time savings of dynamic plans, we might want to determine how many query executions are required to justify their extra optimization and start-up effort. Let us call this number of executions $N_{break-even}$, defined to be the smallest N for which $e + N \times (f + \bar{g}) < a + N \times (b + \bar{c})$. If a query is to be executed at least $N_{break-even}$ times, dynamic plans can be expected to be faster overall than static plans. Reformulating the above inequality determines $N_{break-even} = \lceil (e - a) / ((b + \bar{c}) - (f + \bar{g})) \rceil$. Applying this formula to the times measured in our experiments, the break-even points are consistently as low as $N_{break-even} = 1$. Thus, in these experiments, dynamic plans were a better choice than static plans when run-time bindings were not known at compile-time, even if the plan ended up running only once.

While dynamic plans are clearly much more robust and efficient than traditional, static plans, one might ask whether

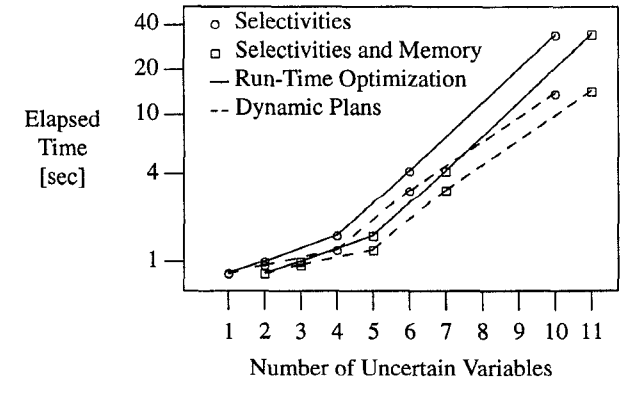

Figure 8. Run-Time Optimization versus Dynamic Plans.

dynamic plans are actually better than delaying all query optimization until start-up-time. The obvious advantage of run-time optimization is that no choose-plan nodes are required, because run-time values for all cost-model parameters are known during optimization. The disadvantage of run-time optimization is that the same query must be optimized repetitively whenever cost model parameters change. In the notation of Figure 3: is $f + \bar{g} < a + \bar{d}$?

In answer to this question, Figure 8 compares the run-time components of run-time optimization with compile-time generated dynamic plans. For other than the simplest queries, there is a significant overall decrease in execution time when using dynamic plans. For query 5, the decrease exceeds a factor of 2. This substantial difference is primarily due to the cost of the start-up-time optimization, which is large when compared to the relatively small run-time overhead of dynamic plans.

Finally, we compute the break-even point $N_{break-even}$ between dynamic plans and run-time optimization, i.e., the smallest number of queries N for which $e + N \times (f + \bar{g}) \leq N \times (a + \bar{d})$. Given that $\forall i \; g_i = d_i$ and therefore $\bar{g} = \bar{d}$, $N_{break-even} = \lceil e / (a - f) \rceil$. For $N_{break-even}$ or more query invocations, dynamic plans require less total computational effort. The largest break-even point in Figure 8 is $N_{break-even} = 4$ (query 5); the smallest break-even point is only $N_{break-even} = 2$, for query 2. These encouraging results are due to the fact that the dynamic-plan start-up-time is much smaller than the optimization effort ($f \ll a$), dominating the differences in optimization time ($e > a$). Thus, even for relatively few invocations of a query, dynamic plans can be much more efficient than run-time optimization.

7. Summary and Conclusions

In this paper, we have addressed an important open problem in database query optimization, namely that the execution of traditionally optimized, "static" query plans is often sub-optimal when cost-model parameters change between compile-time and run-time. While some previous work has considered optimization at run-time as well as run-time mechanisms that facilitate dynamic plans, the present paper is the first to outline a general technique to create dynamic plans at compile-time using exhaustive search in a dynamic programming framework. Our solution is based on the following essential concepts:

- Uncertain cost-model parameters and therefore costs incomparable at compile-time.
- A partial ordering of alternative plans by cost, instead of a total ordering, induced by incomparable costs.
- The choose-plan operator, which enables the building of dynamic plans that incorporate alternative subplans.
- Decision procedures for choose-plan operators based on cost function evaluations with instantiated run-time bindings.
- Extensions to dynamic programming, memoization, and branch-and-bound pruning necessary for optimization and generation of dynamic plans.

Unfortunately, despite using dynamic programming and memoization, dynamic plan optimization is slower than traditional optimization. In dynamic plan optimization, branch-and-bound pruning is less effective than in traditional query optimization, because less knowledge about run-time bindings and about costs prevents tight bounds and cost comparisons. We have found that this is the most important impediment to optimization efficiency, but believe that the added optimization effort is well worth the improved run-time behavior of dynamic plans over traditional, static plans.

We applied these optimization concepts to the construction of a prototype dynamic plan optimizer using the Volcano optimizer generator. As a particular instance of dynamic-plan optimization, we extended plan cost from traditional point data to interval data and defined costs to be incomparable if these intervals overlap, illustrating one possible use of incomparable cost estimates and dynamic plan technology.

Experiments with this prototype demonstrate that our approach to dynamic plan optimization using dynamic programming techniques is an effective and efficient way to build plans exhibiting robust performance characteristics. In particular, our experiments permit the following conclusions:

- Dynamic plan optimization produces robust plans that maintain their optimality even when parameters change between compile-time and start-up-time. The reduced execution time of dynamic relative to static plans more than offsets the additional overhead for plan activation.
- The combined I/O and CPU overhead of dynamic plan evaluation is significantly less than the corresponding overhead of completely optimizing queries at start-up-time. Thus, dynamic plans generated at compile-time are more efficient than run-time optimization, in particular for complex queries.

In summary, we have presented an effective and general solution to the important and heretofore open problem of unknown run-time bindings, particularly for program variables in embedded queries and for other uncertain variables such as join selectivities and memory availability when they are known at start-up-time. Our experiments with a working prototype demonstrate that for queries with run-time bindings unknown at compile-time, the proposed approach is superior to both static plans and run-time optimization.

Our approach is extensible in terms of the data model and in terms of the sources of cost incomparability, because it is based on extensions to the Volcano optimizer generator's search engine and because the criteria for cost incomparability are encapsulated by the abstract data type for cost. The optimizer generator has already been used successfully in optimizing object-oriented queries for the Open OODB system [BMG93] and analysis queries in scientific databases [WoG93]. Because of the optimizer generator's encapsulation of the sources of cost incomparability, it could easily be extended to handle uncertainty in object-oriented clustering techniques [TsN92] and in the existence of path indices [OHM92] as well as other issues of compile-time versus run-time bindings.

While the discussion here has focused on delaying decisions from compile-time to start-up-time, decisions can also be delayed further into run-time. This generalization is necessary if cost-model parameters may change after start-up-time, either because the start-up-time expected values were inaccurate (i.e., selectivity estimation errors [IoC91]) or because parameter values change during run-time. While we have not completed our research into this problem, our initial approach has been to handle inaccurate expected values by evaluating subplans as part of choose-plan decision procedures. When a subplan has been evaluated into a temporary result, its logical and physical properties (e.g., result cardinality and value distributions) are known and therefore may contribute to decisions with increased confidence in the resulting plan's optimality. The novel aspect of this approach when compared with previous approaches to run-time optimization decisions [Ant93, BPR90, MHW90] is that dynamic plan optimization, in combination with statistical decision theory as used by Seppi et al. [SBM89], will provide sound decisions based on the concept of incomparable costs. Thus, our research into compile-time optimization of dynamic

plans creates promising opportunities for further research, in addition to already solving an important open problem in database query optimization.

Acknowledgements

José A. Blakeley, Guy Lohman, David Maier, and Barb Peters made excellent suggestions for the presentation of this paper, which we appreciate very much. — This research was partially supported by the National Science Foundation with grants IRI-8996270, IRI-8912618, IRI-9116547, and IRI-9119446, the Advanced Research Projects Agency (ARPA order number 18, monitored by the US Army Research Laboratory under contract DAAB-07-91-C-Q518), Texas Instruments, and Digital Equipment Corp.

References

[Ant93] G. Antoshenkov, "Dynamic Query Optimization in Rdb/VMS", *Proc. IEEE Int'l. Conf. on Data Eng.*, Vienna, Austria, April 1993, 538.

[BMG93] J. A. Blakeley, W. J. McKenna, and G. Graefe, "Experiences Building the Open OODB Query Optimizer", *Proc. ACM SIGMOD Conf.*, Washington, DC, May 1993, 287.

[BPR90] P. Bodorik, J. Pyra, and J. S. Riordon, "Correcting Execution of Distributed Queries", *Proc. Int'l. Symp. on Databases in Parallel and Distributed Systems*, Dublin, Ireland, July 1990, 192.

[CAK81] D. D. Chamberlin, M. M. Astrahan, W. F. King, R. A. Lorie, J. W. Mehl, T. G. Price, M. Schkolnik, P. G. Selinger, D. R. Slutz, B. W. Wade, and R. A. Yost, "Support for Repetitive Transactions and Ad Hoc Queries in System R", *ACM Trans. on Database Sys. 6*, 1 (March 1981), 70.

[Chr84] S. Christodoulakis, "Implications of Certain Assumptions in Database Performance Evaluation", *ACM Trans. on Database Sys. 9*, 2 (June 1984), 163.

[CAB93] R. L. Cole, M. J. Anderson, and R. J. Bestgen, "Query Processing in the IBM Application System/400", *IEEE Data Eng. Bull. 16*, 4 (December 1993), 19.

[CLR89] T. Cormen, C. Leiserson, and R. Rivest, *Introduction to Algorithms*, McGraw-Hill, New York, NY, 1989.

[DMP93] M. A. Derr, S. Morishita, and G. Phipps, "Design and Implementation of the Glue-Nail Database System", *Proc. ACM SIGMOD Conf.*, Washington, DC, May 1993, 147.

[GHK92] S. Ganguly, W. Hasan, and R. Krishnamurthy, "Query Optimization for Parallel Execution", *Proc. ACM SIGMOD Conf.*, San Diego, CA, June 1992, 9.

[GrW89] G. Graefe and K. Ward, "Dynamic Query Evaluation Plans", *Proc. ACM SIGMOD Conf.*, Portland, OR, May-June 1989, 358.

[Gra93] G. Graefe, "Query Evaluation Techniques for Large Databases", *ACM Computing Surveys 25*, 2 (June 1993), 73-170.

[GrM93] G. Graefe and W. J. McKenna, "The Volcano Optimizer Generator: Extensibility and Efficient Search", *Proc. IEEE Int'l. Conf. on Data Eng.*, Vienna, Austria, April 1993, 209.

[HaP88] W. Hasan and H. Pirahesh, "Query Rewrite Optimization in Starburst", *Comp. Sci. Res. Rep.*, San Jose, CA, August 1988.

[HoS93] W. Hong and M. Stonebraker, "Optimization of Parallel Query Execution Plans in XPRS", *Distr. and Parallel Databases 1*, 1 (January 1993), 9.

[IoC91] Y. E. Ioannidis and S. Christodoulakis, "On the Propagation of Errors in the Size of Join Results", *Proc. ACM SIGMOD Conf.*, Denver, CO, May 1991, 268.

[INS92] Y. E. Ioannidis, R. T. Ng, K. Shim, and T. K. Sellis, "Parametric Query Processing", *Proc. Int'l. Conf. on Very Large Data Bases*, Vancouver, BC, Canada, August 1992, 103.

[KGM91] T. Keller, G. Graefe, and D. Maier, "Efficient Assembly of Complex Objects", *Proc. ACM SIGMOD Conf.*, Denver, CO, May 1991, 148.

[Loh89] G. M. Lohman, "Is Query Optimization a 'Solved' Problem?", in *Proc. Workshop on Database Query Optimization*, G. Graefe (editor), Oregon Graduate Center Comp. Sci. Tech. Rep. 89-005, Beaverton, OR, May 1989, 13.

[MaL89] L. F. Mackert and G. M. Lohman, "Index Scans Using a Finite LRU Buffer: A Validated I/O Model", *ACM Trans. on Database Sys. 14*, 3 (September 1989), 401.

[MHW90] C. Mohan, D. Haderle, Y. Wang, and J. Cheng, "Single Table Access Using Multiple Indexes: Optimization, Execution and Concurrency Control Techniques", *Lecture Notes in Comp. Sci. 416* (March 1990), 29, Springer Verlag.

[OnL90] K. Ono and G. M. Lohman, "Measuring the Complexity of Join Enumeration in Query Optimization", *Proc. Int'l. Conf. on Very Large Data Bases*, Brisbane, Australia, August 1990, 314.

[OHM92] J. Orenstein, S. Haradhvala, B. Margulies, and D. Sakahara, "Query Processing in the ObjectStore Database System", *Proc. ACM SIGMOD Conf.*, San Diego, CA, June 1992, 403.

[SAC79] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price, "Access Path Selection in a Relational Database Management System", *Proc. ACM SIGMOD Conf.*, Boston, MA, May-June 1979, 23. Reprinted in M. Stonebraker, Readings in Database Sys., Morgan-Kaufman, San Mateo, CA, 1988.

[SBM89] K. Seppi, J. Barnes, and C. Morris, "A Bayesian Approach to Query Optimization in Large Scale Data Bases", *The Univ. of Texas at Austin ORP 89-19*, Austin, TX, 1989.

[TsN92] M. M. Tsangaris and J. F. Naughton, "On the Performance of Object Clustering Techniques", *Proc. ACM SIGMOD Conf.*, San Diego, CA, June 1992, 144.

[WoG93] R. H. Wolniewicz and G. Graefe, "Algebraic Optimization of Computations over Scientific Databases", *Proc. Intl' Conf. on Very Large Data Bases*, Dublin, Ireland, August 1993, 13.