Dynamic Query Evaluation Plans

Goetz Graefe, Karen Ward

Oregon Graduate Center

Abstract

In most database systems, a query embedded in a program written in a conventional programming language is optimized when the program is compiled. The query optimizer must make assumptions about the values of the program variables that appear as constants in the query, the resources that can be committed to query evaluation, and the data in the database. The optimality of the resulting query evaluation plan depends on the validity of these assumptions. If a query evaluation plan is used repeatedly over an extended period of time, it is important to determine when reoptimization is necessary. Our work aims at developing criteria when reoptimization is required, how these criteria can be implemented efficiently, and how reoptimization can be avoided by using a new technique called ***dynamic query evaluation plans***. We experimentally demonstrate the need for dynamic plans and outline modifications to the EXODUS optimizer generator required for creating dynamic query evaluation plans.

1. Introduction

In many database applications, queries are embedded in application programs written in a conventional programming language like Cobol or C. These application programs are compiled once and then executed repeatedly over an extended period of time. In most database management systems, the embedded queries are optimized when the programs are compiled. The resulting query evaluation plan is stored in an 'access module' and is activated by procedure calls. This organization avoids the optimization overhead when the programs and the queries are executed, thus allowing for fast query evaluation and high transaction rates.

The disadvantage of query optimization at compile time is that free variables in the query predicate and changes in the system load or in the database cannot be reflected in the query evaluation plan. Since an embedded database query is used to retrieve information pertinent to the current program run, it is natural that program data and the database query are connected by using one or more program variables as constants in the query predicate. The value of the program variables is not known at compile-time, i.e., when the query is optimized. While database query optimization does not genuinely depend on the values of constants in the database query, the optimizer needs to estimate intermediate result sizes, which in turn may depend on the constants in the query predicate. For example, the optimal join strategy depends on the number of tuples to be joined from each input. If the inputs are the results of selections, it is imperative that the optimizer estimate the select output size.

When optimizing an embedded query with program variables in the query predicates, database optimizers estimate selectivities using a guessed "typical" value for each variable, or directly guess selectivities [1]. Costs of alternative query evaluation plans are estimated using the guessed values. In the case of complex queries with inequality constraints involving program variables, the resulting query evaluation plan can be far from optimal.

For example, consider a database query to find all employees with a salary greater than $30,000 and their departments. Assume that this query requires joining the employee relation with the department

This research is supported by National Science Foundation grant IRI-8805200.

relation on the department number, and that the only two indices in the database are for the employee relation on salary and for the department relation on department number. If there are very few employees in this salary range, it probably is best to find qualifying employees using the salary index, to use the result as outer relation in the join and to perform repeated index lookups on the department relation. If there are very many such employees, however, it might be best to read the entire department relation (the smaller of the two relations) into a memory-resident hash table and then probe the hash table using the employee tuples, using file scans for both files. In this example, the scanning strategies, the join order, and the join algorithm depend on the cardinality of the two relations and the selectivity of the selection predicate. Imagine that the above query is embedded in an application program, and that the cutoff value, the constant $30,000, is replaced by a program variable. In this case, a conventional query optimizer cannot satisfactorily optimize the query. We are trying to develop an elegant, dynamic, and efficient solution.

Apart from program variables, the system load can be considered a free variable in query optimization. A number of cost functions, e.g., for sorting or hash join with overflow resolution, depend on the size of main memory available for a particular query. If this size cannot be predicted, the optimizer cannot make reasonable cost calculations and decisions. Considering performance-critical resources, e.g., buffer pages, number of processors, number of channels and disk arms for temporary files, etc., free variables allows using effective optimization techniques outlined in this paper. In the sequel, we will not distinguish between program variables and load variables because they are not significantly different from the standpoint of query optimizations.

Another problem with stored query evaluation plans is that they are optimized for the state of the database at program compilation time, not for the database state at query execution time. In the meantime, the database may change such that the query evaluation plan is no longer optimal. We have to distinguish between changes that make a query evaluation plan infeasible and changes that make a query evaluation plan non-optimal. The former category includes removing relations and indices that are referenced in the query evaluation plan and has been addressed by earlier work [2]. The latter category includes creating new indices, inserting or deleting a large number of tuples, and modifying a large number of attribute values. Our research will later address the problems in this category.

Consider the above example again with a constant of $30,000. If most of the newly hired employees are highly paid specialists, the number of qualifying employees will change, as well as the portion of qualifying employees (i.e., the selectivity). If the query is optimized only once and the resulting access plan reused over a long period of time, the query evaluation plan will eventually become suboptimal.

Other researchers also have identified the benefits of multiple plans for a single query. In a number of early query evaluation papers the problem is considered so significant that plan generation is delayed until query evaluation, e.g., [3, 4, 5]. We believe, however, that selectivity estimation techniques have been sufficiently developed to-date that query plans for completely specified queries can be generated at compile time [6, 7, 8]. Our work addresses the case of incompletely specified queries, e.g., queries with free variables in the predicate.

Most closely related to our work is the work of Lee and Yu [9] who suggest to used index information to decide scan method and join algorithm during query evaluation. Their approach, while useful where applicable, is limited to cases with existing indices (therefore it is not applicable to intermediate results) and to algorithm selection. Our approach also allows reordering operators in a query evaluation plan.

The XPRS project at UC Berkeley includes plans for choosing at run time from a set of prepared plans according to the amount of main memory available at query execution time [10]. The Starburst research group is considering rewriting a query into multiple alternatives and selecting from among them by the second optimization phase according to cost estimates or at run time depending on variable bindings [11], but has as yet not developed techniques for creating multi-plan access modules and appropriate selection criteria.

So far, we have looked at the problem as it appears in conventional, relational database management systems. In a database system used to support logic programs, the problem of unknown predicate constants occurs very frequently. Consider a model of execution that relies on backtracking, particularly Prolog [12, 13]. The same clause is activated repeatedly with different variable instantiations. If the clause contains a database query, the same database query is performed with different constants in the query predicate. The techniques described here are aimed at providing more flexibility and better performance for both conventional and non-conventional database management systems and applications.

Another application that will benefit from multiple or dynamic query evaluation plans are object-oriented database systems. If behavior is encapsulated in type definitions, a selection message sent to a collection object results in repeated method invocation for all members of the collection, e.g., in Smalltalk [14] or GemStone [15]. If a method includes a database lookup, preparing a set of execution plans that covers all possible cases can greatly improve query performance. We have incorporated dynamic query evaluation plans in the Volcano query evaluation system prototype [16] developed at OGC and intend to exploit them in a high-performance object-oriented database system [17].

The remainder of this paper is organized as follows. In Section 2 we propose mechanisms for testing efficiently whether or not a query evaluation plan is optimal, i.e., whether the query plan with the actual constants is optimal for the current state of the database. Section 3 describes how some of these problems can be solved by choosing the scanning strategy dynamically. Section 4 extends these techniques to join processing. In Section 5, we develop a general technique, called **dynamic query evaluation plans**, with the goal of providing minimal overhead, maximal flexibility, or both. The implementation of dynamic query evaluation plans is described in Section 6, which include actual measurements demonstrating the performance gains through dynamic plans. Section 7 describes our current research, including extensions to the EXODUS optimizer generator. Section 8 contains a summary and our conclusions.

2. Test of Optimality

In order to test whether or not a given query evaluation plan is optimal for a set of query constants given in the program variables, a special predicate is associated with each compiled query evaluation plan. When a plan is activated with a record of actual values for the query constants, the associated predicate is evaluated on this record and returns one of the values *TRUE* or *FALSE*. In the case of *TRUE*, the access plan is considered appropriate, and query processing proceeds as in existing database systems. In the case of *FALSE*, the access plan is considered suboptimal, and the database query optimizer is invoked.

It can be argued that the overhead incurred by evaluating the predicate is unacceptable for high-performance database systems. Consider, for example, a banking teller transaction. Probably, an optimized plan finds account records using a hash index on account numbers (assuming such an index exists); testing this access plan's optimality will be wasted effort. While this is true, we would like to alert the reader to two points. First, if the predicate used to test the optimality is compiled into machine code (just as predicates on data records should be), evaluation requires probably in the range of 20 to 100 instructions. Second, in extreme cases like the banking teller example, the predicate can be designed such that it always returns *TRUE* without inspecting the record of actual values.

When the query optimizer has been reinvoked for a certain query after the original query evaluation plan was rejected, it is probably a good idea to keep both plans and choose dynamically among them in future activations of the query. In general, there might be a number of access plans to be chosen from dynamically. It is not even necessary to wait until the original query evaluation plan is rejected; rather, it should be possible to prepare more than one plan when the query is optimized originally. We discuss this concept in more detail in the following sections.

3. Dynamic Decisions on Scanning Strategies

Dynamic decisions on the best scanning strategy are the first step towards dynamic query evaluation plans. The alternative scanning strategies are file scan and index scan, if a suitable index exists. Let us first review the rationale by means of an example.

Consider a large file, e.g., with 10,000 employee records stored in 1,000 pages. If we need to retrieve all records, we should use a file scan, as this method ensures that we inspect each data page only once and allows high-performance techniques like read-ahead. If we retrieve only 10 of the 10,000 records, we do better by using an index (assuming one exists). We would have to read at most 10 data pages, and probably less than 30 index pages. If we retrieve 2,000 records, however, it is quite likely that we eventually have to read all 1,000 data pages and the index adds only to the overhead for three reasons. First, we have to read the index pages, second, if the index is an unclustered index, we inspect many pages more than once, and third, we cannot make best use of read-ahead. Yao [18] gives an estimation formula to determine the number of page accesses from the number of qualifying records; others have expanded on this work, e.g., [19, 20, 21, 22, 23]. In addition to the cost of pages accesses, there are also costs for locking and concurrency control (let us assume that locks are used for concurrency control). If only relevant records are accessed using the index, only those records need to be locked. For a file scan, all records must be locked. However, this can be done with a single call to the lock manager if a hierarchical locking scheme is used. Depending on the selectivity and the current system load, either one of the two strategies can be optimal.

The lesson from this example is that there are many considerations that favor index scan over file scan or vice versa, depending on the actual situation when the query evaluation plan is activated. The choice depends on the selectivity of the predicate and on the current system load, and can be performed efficiently at run time. If the interfaces to the file scan procedure and the index scan procedures are equal (or very similar), a dynamic choice of the scan method can be implemented with reasonable effort and run time overhead. The task of the query optimizer is to determine the break-even point. More exactly, the query optimizer must determine the formulas to find the break-even point and to compare it with the actual selectivity, and include these formulas in the query evaluation plan.

4. Dynamic Decisions on Join Strategies

As we have seen in the example in the introduction, in some cases it is desirable to select the join strategy at run time. If there are more than one join operator cascaded in a query evaluation plan, the decisions on the join strategies are interdependent. Most importantly, the physical (sort) order of intermediate results may affect the cost of algorithms for the next processing step if order-sensitive algorithms such as merge-join are used.

The query optimizer's task for this kind of access plan is more complex. Instead of following a set of assumption and guesses about distributions, selectivities, etc., it must design an efficient decision procedure which can be executed when the query evaluation plan is activated. This decision procedure must have resolved the interdependence of partial decisions into a straight-forward decision tree, and include the break-even values between alternative plans.

Since the number of possible join strategies is very large, even for only moderately complex queries, it is not possible to include all query execution plans in the access module. There are two solutions to this problem. First, the optimizer can select a subset of query execution plans. The plans are selected to allow reasonably efficient query evaluation for any set of parameters. In order to keep this set small, plans with wide range of optimality must be selected. We call this a query evaluation plan's *stability* and will develop this concept is a later paper. Second, instead of storing complete plans, only elements of the plans are stored, and linked together when the access module is activated.

5. Dynamic Query Evaluation Plans

Instead of a set of query evaluation plans, as suggested in the previous sections, we propose to avoid redundancy in the access module by designing the data structure for the query evaluation plans to be more flexible. The access modules of existing database management systems consist of a number of components, e.g., an indicator for file scan with a file name and a search predicate, an indicator for hash join with a hash function and a comparison function, etc. These components are linked together into a static plan by the query optimizer, thus hiding the fact there are several components. **Dynamic access modules** consist of the same components, only the binding between components is more flexible. The only new component is the **decision procedure** used to analyze the actual query constants and the data distribution. When an access module is activated, the first step is to evaluate the decision tree. Concurrently with the evaluation of the decision tree, this step sets up the bindings between the components of the access module.

In addition to the decision tree designed by the optimizer, the access module must also contain the support functions for all possible query evaluation plans. These support functions include comparisons, hash functions, etc. The physical organization of the access module must allow equally efficient execution of any of the query evaluation plans.

Introducing the dynamic choices outlined in the sections above are an important step toward more flexible and efficient database systems. However, instead of considering scanning and join strategies separately, we are investigating how far the concept of **dynamic query evaluation plans** can be generalized. After this more general investigation, we can consider selection of scanning and join strategies as special cases. Other special cases that come immediately to mind are join orders and the placement and execution of aggregate functions.

It can be argued that setting up the bindings dynamically inflicts too much overhead on query processing. Consider the example of a banking teller transaction introduced in Section 3. If there is no gain in using a dynamic access module, the decision tree can be an empty function. In this case, all bindings must be set statically, and "evaluating" the decision tree costs only one instruction. The techniques proposed here do not require that as many choices as possible must be delayed until run time. Their advantage is that they allow delaying exactly as many choices as advisable.

6. A Dynamic Query Evaluation System

In order to assess whether dynamic query evaluation plans are an attractive practical alternative to existing static plans, we have implemented dynamic query evaluation plans in the Volcano query evaluation system [16]. Volcano includes file scan, B-tree scan, select, project, duplicate elimination, sorting, merge join, hash join, naive and hash division [24], and sort- and hash-based aggregate functions. Each algorithm is implemented as an *iterator*, i.e., there are *open*, *next*, and *close* procedures for each algorithm. Associated with each algorithm is a *state record*, allowing multiple use of an algorithm in a query. The arguments for the algorithms, e.g., predicates, are kept in the state record. We also have a facility for parallel plan execution which we will not describe here.

In queries involving more than one operator (i.e., almost all queries), state records are linked together by means of *input* pointers, also kept in the state records. Calling *open* for the top-most operator results in *open* calls for all inputs and in instantiations for the associated state record, e.g., allocation of a hash table. In this way, all iterators in a query are initiated recursively. In order to process the query, *next* for the top-most operator is called repeatedly until it fails with an *end-of-stream* error. Finally, the *close* call recursively "shuts down" all iterators in the query. This model of query execution matches very closely the one being included in the E programming language design [25] and the algebraic query evaluation system of the Starburst extensible relational database system [26].

The current implementation includes a *choose-plan* operator to realize both multi-plan access modules and dynamic plans. This operator provides the same *open, next, close* protocol as the other operators and can therefore be inserted into a query plan at any location. The *open* operation decides which of its several equivalent query plans to use, and invokes the *open* operation for this input. The *next* and *close* operations simply call the appropriate operation for the input chosen during *open*.

The *choose-plan* operator allows considerable flexibility in our experiments. If only one *choose-plan* operator is used as top of a query evaluation plan, it implements a multi-plan access module. If multiple *choose-plan* operators are included in a plan, they simulate a dynamic query evaluation plan. They only simulate a dynamic plan because the decision process is integrated with the open procedure rather than preceding *open*, thus requiring the operator's *next* and *close* operation and their overhead. However, the flexibility and the implementation simplicity of the *choose-plan* operator seem worth this overhead.

Access modules are composed of state records and support functions. We are currently designing the required extensions to the EXODUS query optimizer generator to produce dynamic query evaluation plans, i.e., plans with *choose-plan* operators.

6.1. Experimental Results

The following example demonstrates how considerable savings might be realized by choosing dynamically among multiple query evaluation plans.

The test data are drawn from the billing files of a utility company. The example query selects records from a *site* relation containing location and revenue information for each site where energy meters are installed. Much of the location information is encoded. For instance, the names of the cities served by the utility are found in a separate city table. The address of a particular site is formed by retrieving the street portion of the address from the site record, then using a city code to look up the city name.

The test database contains 13,229 site records, each 252 bytes long, and 77 city records of 28 bytes. Indices have been built for the site relation on revenue amount, and for the city relation on city code.

The example query retrieves address information for those sites whose most recent revenue amount exceeded a certain threshold, where the cutoff amount is a variable. Such a query might be embedded in a program that produces specialized mailing lists. The amount might be set to the minimum bill amount to include all active in the mailing, or it might be set to a high value to identify the largest accounts. When the threshold amount is high, we would prefer to use the amount index to select those few sites that qualify, then use the city code index to complete the address information. If more than a few sites will qualify, however, it would be better to avoid the overhead of reading and repeatedly searching the indices. Instead, we can ensure that each page is read only once by scanning the site file after building a memory-resident hash table from the city file.

We constructed a multi-plan access module containing the state records and support functions needed to execute this query using either B-tree scan and index nested loops join or file scans and hash join.

Currently, an access module is stored apart from both the program that builds it and the program(s) that may execute it. It can be "embedded" in another program by having that program read the access module into a buffer as data. The module is then invoked by passing the buffer address to an execute algorithm. Execute uses displacement information included in the access module to build the input pointers for the topmost state records, until a *choose-plan* operator is encountered. If there are no *choose-plan* operators, execute will link all of the plan's state records and support functions.

Comparison of Query Strategies (time in seconds)								
Tuples	B-tree Scans and Join				File Scans and Hash Join			
Printed	CPU	I/O (1)	I/O (2)	Cost	CPU	I/O (1)	I/O (2)	Cost
0	0.14	0.059	0.130	0.329	1.04	3.410	0.083	4.533
1	0.16	0.115	0.154	0.429	1.07	3.410	0.083	4.563
251	0.67	2.146	0.202	3.018	1.21	3.410	0.083	4.703
501	1.06	3.276	0.274	4.611	1.37	3.410	0.083	4.863
752	1.69	3.726	0.322	5.739	1.39	3.410	0.083	4.883
1001	2.14	4.641	0.370	7.151	1.52	3.410	0.083	5.013
5002	9.72	6.408	1.162	17.290	3.40	3.410	0.083	6.893
10010	19.02	7.090	2.170	28.280	5.77	3.410	0.083	9.263
13229	25.23	8.113	2.914	36.257	7.22	3.410	0.083	10.713

Table 1. Costs of two plans for various output sizes.

A *choose-plan* operator fetches the information it needs to select the best plan. In our implementation of this example, it reads a control record from the database which contains the threshold amount, and chooses the plan to use in the current invocation. It then completes linking the state records for the indicated plan. We have redesigned the *open* procedures to use a second parameter besides the state record, namely a record called *bindings* which contains query predicate constants and system load information.

The cost function used to evaluate the performance of a query is the sum of the CPU time in user mode[1] and the estimated time spent performing I/O (calculated from statistics kept by the file system), expressed in seconds. 3600 KB were allocated as file system buffer space.

For the experiment, we "forced" the *choose-plan* operator to use a plan of our choice, disreganding the threshold amount. (Of course, the selection predicate did use the threshold amount.) Each query plan was invoked using a series of threshold amounts. The results for each plan-amount combination are summarized in Table 1. The columns for I/O reflect the I/O cost on two disk devices. The first one, labeled I/O(1), contains the data files, while the second one contains the indices.

The strategies employing file scans and hash join, shown in the four right-most columns, are comparatively insensitive to the output size. Since all pages of both tables are read into the buffer, the I/O cost is entirely independent of the cutoff value. The CPU cost for resource management (buffers, memory, etc.) and for applying the selection predicate are almost identical for all cutoff values; most of the differences are due to hash and comparison functions and to copying record fields into the output table.

The cost of the index-based access plan is significantly more sensitive to the cutoff value applied. For very few result tuples, the cost grows very steeply with the number of result tuples, and it grows about linearly for large outputs. This result was to be expected as the number of index look-ups and data page fetches is approximately equal to the number of result tuples.

For a small cutoff value and very few result tuples, the first plan using indices is superior to the second. As the number of result tuples increases, however, this advantage diminishes and eventually turns into a disadvantage.

The ratio of costs at the two ends is surprising. If only one account satisfies the predicate, the index strategy is superior by a factor of 10. If all or almost all accounts qualify, it is inferior by a factor of more than 3. This very large range has a significant impact on query optimization. If a cutoff value is not known at compile and query optimization time, as is frequently the case for embedded queries, choosing either one of these plans is wrong. One alternative is to delay query optimization until run time in order to include the actual cutoff value in the selectivity and cost calculations. This alternative increases query cost by the time used for optimization, which could easily double the reponse time for this query. In other words, a conventional query optimizer effectively cannot optimize this simple query. Dynamic query evaluation plans, on the other hand, can decide in very little time which plan is optimal and ensure the less expensive plan is used every time the application program is run.

[1] The queries were run on a Sequent Symmetry which uses Intel 80386 CPU's with a 16 Mhz clock. CPU time was determined using the UNIX system call *getrusage*.

7. Current Work

Our current research consists of two interdependent design and implementation efforts. First, we are augmenting the EXODUS query optimizer generator software to create multiple query evaluation plans and a decision procedure. Second, we plan on changing the search strategy employed by generated optimizers.

7.1. Modifications of the EXODUS Optimizer Generator Software

We are currently designing and implementing modifications to the EXODUS Optimizer Generator [27, 28, 29] that enable generated optimizers to optimize queries with free variables by creating more than one query evaluation plan. The first two modules, which are partially implemented, is actually a pre- and post-processor for the optimizer. The preprocessor parses initial queries coded in QUEL-like syntax and transformed them into an operator tree before optimization. The operator tree consists of Cartesian product operators, followed by a selection and a projection. The postprocessor traverses a query evaluation plan and generates type definitions, support functions, and argument and state records in the C programming language. These programs are then compiled by the standard C compiler, linked with the query evaluation library, and run against the database. The program generated from the optimized query evaluation plan includes a print operator which prints the query result on the screen.

The next step will to allow designation of free variables in queries. The optimizer will not consider values for these variables and will generate multiple query evaluation plans. The *choose-plan* operator will use actual variable bindings to determine which plan to execute. While designating "free" variables in interactive queries is somewhat unrealistic, it reduces the time to set up an environment for implementing and experimenting with multiple plans and dynamic plans, and to investigate the core of the research, the optimizer search strategy.

In a later stage, we intend to augment the selectivity estimation procedures and cost functions with observations from query evaluations. Thus, we will adjust and *learn* optimal values and formulas for these functions, complementing earlier work on learning within the query optimizer [28]. Notice that only dynamic query evaluation plans allow incorporating cost function adjustments in existing access modules effectively, i.e., without recompilation, thus providing the most promising environment for learning and adjusting selectivity estimation and cost functions.

7.2. The Search Strategy

In the current design of optimizers generated with EXODUS, as in most other query optimizers, the task of the query optimization module is to find one best query evaluation plan. In the optimization paradigm proposed here, the goal of query optimization is to find a set of plans and an efficient decision procedure to determine which of these plans is optimal in the actual access module invocation. We call this new optimizer a *multi-plan optimizer*, as opposed to single-plan optimizers found in conventional query optimization. The number of query evaluation plans can be prohibitively large. Developing and using the concept of the stability of query evaluation plans, however, we have reason to believe that the number of required plans in an access module can be kept reasonably low.

The first experiment will involve a binary search scheme similar to the one proposed for XPRS [10]. The independent variable for the binary search can be the buffer size as in XPRS or the cardinality of an intermediate result, calculated from a variable binding. Clearly, only a very limited number of variables can be dealt with in this way, probably only one or two, e.g., buffer size and the number of available CPUs. For the more general problem of embedded queries with multiple free variables, the search space must be pruned more aggressively. We expect that the common subplan analysis implemented in the EXODUS software to detect multiple derivations of the same query evaluation plan will significantly reduce the amount of redundant work.

The other technique we will employ is to split ranges during optimization of subplans only *on demand*. In the first query tree analysis, the possible range of result cardinalities for each of the operations is determined. Instead of keeping just one value for the result cardinality in each node of the operator tree, as is done in most relational optimizers including the one described in [28], cardinalities are specified by their possible range. Similarly, the costs of query evaluation plans and subplans are considered as ranges instead of as single values.

When a cost function is invoked, it first determines whether the cost increases "smoothly" over the range of input sizes, or whether the cost function shows a significant discontinuity. If necessary, the range is split at points of discontinuity, and a *choose-plan* operator is introduced into the plan. In this way, multiple methods may be selected for a single operator distinguished by cardinality ranges.

Consider the example query used in the introduction, which involves two relations with a selection using a free variable and a join. The cost function of hash join may indicate a discontinuity at the point where temporary files must be used to resolve hash table overflow. At this point, the range is split and two sub-plans are used, e.g., one using hash join and one using merge join.

If two methods with smooth cost functions show overlapping cost ranges, i.e., one plan is less expensive than the other at one end of the range and higher at the other end, the optimizer invokes the cost function repeatedly to determine the break-even point of the two plans, and optimizes the subranges separately. A *choose-plan* operator is introduced to determine at run time which of the plans should be used.

Consider the above example again, but assume that no hash table overflow occurs. Since index scan and index nested loops join are superior if very few tuples satisfy the query predicate but are inferior at the opposite extreme, the cost range for the plan using indices overlaps the cost range of the plan using hash join. Thus, the optimizer splits the range after determining the break-even point of the two plans.

Plans that include a *choose-plan* operator i.e., the operators that use the output of this operator as their input, are *reanalyzed* appropriately, as detailed in [27, 28]. Reanalyzing may result in further method splits at higher levels of the operation tree and introduction of more *choose-plan* operators. In this case, lower level *choose-plan* operators may be removed, effectively migrating them up in the operator tree, such that the final plan may include only one *choose-plan* operator at the top of the query.

For example, the query used in the introduction has two optimal plans, depending on the cutoff value for the *salary* constant. Neither plan requires a *choose-plan* operator within the query evaluation plan, i.e., the *choose-plan* operator has migrated up to the root of the plan tree.

8. Summary and Conclusions

In this paper, we have identified a significant problem in current models of query optimization and evaluation. For cases in which a single plan cannot cover the entire possible range of query constants, data distributions, and resource situations, we have suggested a very efficient scheme to decide dynamically when to reoptimize the query or to choose one of several query evaluation plans. We have designed and implemented a query evaluation system that includes dynamic access modules. Dynamic access modules perform the final steps of composing a query evaluation plan very efficiently at run time from fragments prepared by the query optimizer using a decision and linking procedure also included in the access module by the query optimizer.

The concepts presented here can be expected to enhance database systems performance significantly for both conventional and non-conventional application domains. For conventional domains, the emphasis is on better and more flexible support for queries embedded in application programs, and on adaptation to varying system loads. Within non-conventional domains, we envision the new techniques to be particularly advantageous for database systems supporting logic programming relying on backtracking, and in object-oriented database systems that employ query optimization.

Acknowledgements

Guy Lohman provided interesting and stimulating discussions, and pointed out the importance of learning cost functions from run time observations. David DeWitt suggested, supervised, and influenced the development of the EXODUS optimizer generator. Portland General Electric provided the test data, which is gratefully acknowledged.

References

1. P. Griffiths Selinger, M.M. Astrahan, D.D. Chamberlin, R.A. Lorie, and T.G. Price, "Access Path Selection in a Relational Database Management System," *Proceedings of the ACM SIGMOD Conference*, pp. 23-34 (May-June 1979).
2. D.D. Chamberlin, M.M. Astrahan, W.F. King, R.A. Lorie, J.W. Mehl, T.G. Price, M. Schkolnik, P. Griffiths Selinger, D.R. Slutz, B.W. Wade, and R.A. Yost, "Support for Repetitive Transactions and Ad Hoc Queries in System R," *ACM Transactions on Database Systems* **6**(1) pp. 70-94 (March 1981).
3. E. Wong and K. Youssefi, "Decomposition - A Strategy for Query Processing," *ACM Transactions on Database Systems* **1**(3) pp. 223-241 (September 1976).
4. K. Youssefi and E. Wong, "Query Processing in a Relational Database Management System," *Proceedings of the Conference on Very Large Data Bases*, pp. 409-417 (October 1979).
5. C.T. Yu and C.C. Chang, "On the Design of a Query Processing Strategy in a Distributed Database Environment," *Proceedings of the ACM SIGMOD Conference*, pp. 30-39 (May 1983).
6. M.V. Mannino, P. Chu, and T. Sager, "Statistical Profile Estimation in Database Systems," *ACM Computing Surveys* **20**(3)(September 1988).
7. L.F. Mackert and G.M. Lohman, "R* Optimizer Validation and Performance Evaluation for Local Queries," *Proceedings of the ACM SIGMOD Conference*, pp. 84-95 (May 1986).
8. L.F. Mackert and G.M. Lohman, "R* Optimizer Validation and Performance Evaluation for Distributed Queries," *Proceedings of the*

Conference on Very Large Data Bases, pp. 149-159 (August 1986).

9. Y.H. Lee and P.S. Yu, "Adaptive Selection of Access Path and Join Method," *unpublished manuscript,* (1988).

10. M. Stonebraker, R. Katz, D. Patterson, and J. Ousterhout, "The Design of XPRS," *Proceedings of the Conference on Very Large Databases,* pp. 318-330 (August 1988).

11. W. Hasan and H. Pirahesh, "Query Rewrite Optimization in Starburst," *Computer Science Research Report,* (RJ 6367 (62349))IBM Almaden Research Center, (August 1988).

12. D.H.D. Warren, L.M. Pereira, and F. Pereira, "PROLOG - The Language and its Implementation Compared with Lisp," *Proceedings of ACM SIGART-SIGPLAN Symposion on AI and Programming Languages,* (1977).

13. W. Clocksin and C. Mellish, *Programming in Prolog,* Springer, New York (1981).

14. A. Goldberg, D. Robson, and D. Ingalls, *Smalltalk-80: The Language and its Implementation,* Addison-Wesley, Reading, MA. (1983, 1985).

15. G. Copeland and D. Maier, "Making Smalltalk a Database System," *Proceedings of the ACM SIGMOD Conference,* pp. 316-325 (June 1984).

16. G. Graefe, "Volcano: An Extensible and Parallel Dataflow Query Evaluation System," *in preparation,* (February 1989).

17. G. Graefe and D. Maier, "Query Optimization in Object-Oriented Database Systems: A Prospectus," pp. 358-363 in *Advances in Object-Oriented Database Systems,* ed. K.R. Dittrich,Springer-Verlag (September 1988).

18. S.B. Yao, "Optimization of Query Evaluation Algorithms," *ACM Transactions on Database Systems* **4**(2) pp. 133-155 (June 1979).

19. S. Christodoulakis, "Estimating Block Transfers and Join Sizes," *Proceedings of the ACM SIGMOD Conference,* pp. 40-54 (May 1983).

20. S. Christodoulakis, "Estimating Record Selectivities," *Information Systems* **8**(2) pp. 105-115 (1983).

21. S. Christodoulakis, "Estimating Block Selectivities," *Information Systems* **9**(1) p. 69 (1984).

22. B.T. Vander Zanden, H.M. Taylor, and D. Bitton, "Estimating Block Accesses When Attributes Are Correlated," *Proceeding of the Conference on Very Large Data Bases,* pp. 119-127 (August 1986).

23. B.T. Vander Zanden, H.M. Taylor, and D. Bitton, "A general framework for computing block accesses," *Information Systems* **12**(2) p. 177 (1987).

24. G. Graefe, "Relational Division: Four Algorithms and Their Performance," *Proceedings of the IEEE Conference on Data Engineering,* pp. 94-101 (February 1989).

25. J.E. Richardson and M.J. Carey, "Programming Constructs for Database System Implementation in EXODUS," *Proceedings of the ACM SIGMOD Conference,* pp. 208-219 (May 1987).

26. L.M. Haas, W.F. Cody, J.C. Freytag, G. Lapis, B.G. Lindsay, G.M. Lohman, K. Ono, and H. Pirahesh, "An Extensible Processor for an Extended Relational Query Language," *Computer Science Research Report,* (RJ 6182 (60892))IBM Almaden Research Center, (April 1988).

27. G. Graefe and D.J. DeWitt, "The EXODUS Optimizer Generator," *Proceedings of the ACM SIGMOD Conference,* pp. 160-171 (May 1987).

28. G. Graefe, "Rule-Based Query Optimization in Extensible Database Systems," *Ph.D. Thesis,* University of Wisconsin, (August 1987).

29. G. Graefe, "Software Modularization with the EXODUS Optimizer Generator," *IEEE Data Engineering,* (December 1987).