DECEMBER 1986 VOL. 9 NO. 4

a quarterly bulletin of the
Computer Society of the IEEE
technical committee on

Database Engineering

CONTENTS

Letter from the Editor 1
G. Lohman

Issues in the Optimization of a Logic Based Language 2
R. Krishnamurthy, C. Zaniolo

Optimization of Complex Database Queries Using Join Indices 10
P. Valduriez

Query Processing in Optical Disk Based Multimedia Information Systems 17
S. Christodoulakis

Query Processing Based on Complex Object Types 22
E. Bertino, F. Rabitti

Extensible Cost Models and Query Optimization in GENESIS 30
D. Batory

Software Modularization with the EXODUS Optimizer Generator 37
G. Graefe

Understanding and Extending Transformation-Based Optimizers 44
A. Rosenthal, P. Helman

SPECIAL ISSUE ON RECENT ADVANCES IN QUERY OPTIMIZATION

Editor-in-Chief, Database Engineering

Dr. Won Kim
MCC
3500 West Balcones Center Drive
Austin, TX 78759
(512) 338-3439

Associate Editors, Database Engineering

Dr. Haran Boral
MCC
3500 West Balcones Center Drive
Austin, TX 78759
(512) 338-3469

Prof. Michael Carey
Computer Sciences Department
University of Wisconsin
Madison, WI 53706
(608) 262-2252

Dr. C. Mohan
IBM Almaden Research Center
650 Harry Road
San Jose, CA 95120-6099
(408) 927-1733

Prof. Z. Meral Ozsoyoglu
Department of Computer Engineering and Science
Case Western Reserve University
Cleveland, Ohio 44106
(216) 368-2818

Dr. Sunil Sarin
Computer Corporation of America
4 Cambridge Center
Cambridge, MA 02142
(617) 492-8860

Chairperson, TC

Dr. Sushil Jajodia
Naval Research Lab.
Washington, D.C. 20375-5000
(202) 767-3596

Vice-Chairperson, TC

Prof. Krithivasan Ramamrithan
Dept. of Computer
and Information Science
University of Massachusetts
Amherst, Mass. 01003
(413) 545-0196

Treasurer, TC

Prof. Leszek Lilien
Dept. of Electrical Engineering
and Computer Science
University of Illinois
Chicago, IL 60680
(312) 996-0827

Secretary, TC

Dr. Richard L. Shuey
2338 Rosendale Rd.
Schenectady, NY 12309
(518) 374-5684

Database Engineering Bulletin is a quarterly publication of the IEEE Computer Society Technical Committee on Database Engineering. Its scope of interest includes: data structures and models, access strategies, access control techniques, database architecture, database machines, intelligent front ends, mass storage for very large databases, distributed database systems and techniques, database software design and implementation, database utilities, database security and related areas.

Contribution to the Bulletin is hereby solicited. News items, letters, technical papers, book reviews, meeting previews, summaries, case studies, etc., should be sent to the Editor. All letters to the Editor will be considered for publication unless accompanied by a request to the contrary. Technical papers are unrefereed.

Opinions expressed in contributions are those of the individual author rather than the official position of the TC on Database Engineering, the IEEE Computer Society, or organizations with which the author may be affiliated.

Membership in the Database Engineering Technical Committee is open to individuals who demonstrate willingness to actively participate in the various activities of the TC. A member of the IEEE Computer Society may join the TC as a full member. A non-member of the Computer Society may join as a participating member, with approval from at least one officer of the TC. Both full members and participating members of the TC are entitled to receive the quarterly bulletin of the TC free of charge, until further notice.

Letter from the Editor

From the earliest days of the relational "revolution", one of the most challenging and significant components of relational query processing has been query optimization, which finds the cheapest way to execute procedurally a query that is (usually) stated non-procedurally. In fact, a high-level, non-procedural query language has been — and continues to be — a persuasive sales feature of relational DBMSs.

As relational technology has matured in the 1980s, increasingly sophisticated capabilities have been added: first support for distributed databases, and more recently a plethora of still more ambitious requirements for multi-media databases, recursive queries, and even the nebulous "extensible" DBMS. Each of these advances poses fascinating new challenges for query optimization.

In this issue, I have endeavored to sample some of this pioneering work in query optimization. Research contributions, not surveys, were my goal. Space constraints unfortunately limited the number of contributors and the scope of inquiry to the following:

Although the processing of recursive queries has been a "hot topic" lately, few have explored the impact on query optimization, as Ravi Krishnamurthy and Carlo Zaniolo have done in the first article. Patrick Valduriez expands upon his recent ACM TODS paper on join indexes to show how a query optimizer can best exploit them, notably for recursive queries.

Multi-media databases expand the scope of current databases to include complex objects combining document text, images, and voice, portions of which may be stored on different kinds of storage media such as optical disk. Stavros Christodoulakis highlights some of the unique optimization problems posed by these data types, their access methods, and optical disk storage media. Elisa Bertino and Fausto Rabitti present a detailed algorithm for processing and resolving the ambiguities of queries containing predicates on the *structure* as well as the *content* of complex objects, which was implemented in the MULTOS system as part of the ESPRIT project.

The last three papers present alternative approaches to extensible query optimization. Don Batory discusses the "toolkit" approach of the GENESIS system, which uses parametrized types to define standardized interfaces for synthesizing "plug-compatible" modules. Goetz Graefe expands upon his "optimizer generator" approach that was introduced in his 1987 ACM SIGMOD paper with Dave DeWitt, in which query transformation rules are compiled into an optimizer. And Arnie Rosenthal and Paul Helman characterize conditions under which such transformations are legal, and extensible mechanisms for controlling the sequence and extent of such transformations.

I hope you find these papers as interesting and significant as I did while editing this issue.

Guy M. Lohman
IBM Almaden Research Center

Issues in the Optimization of a Logic Based Language

R. Krishnamurthy *Carlo Zaniolo*

MCC, 3500 Balcones Center Dr., Austin, TX, 78759

Abstract

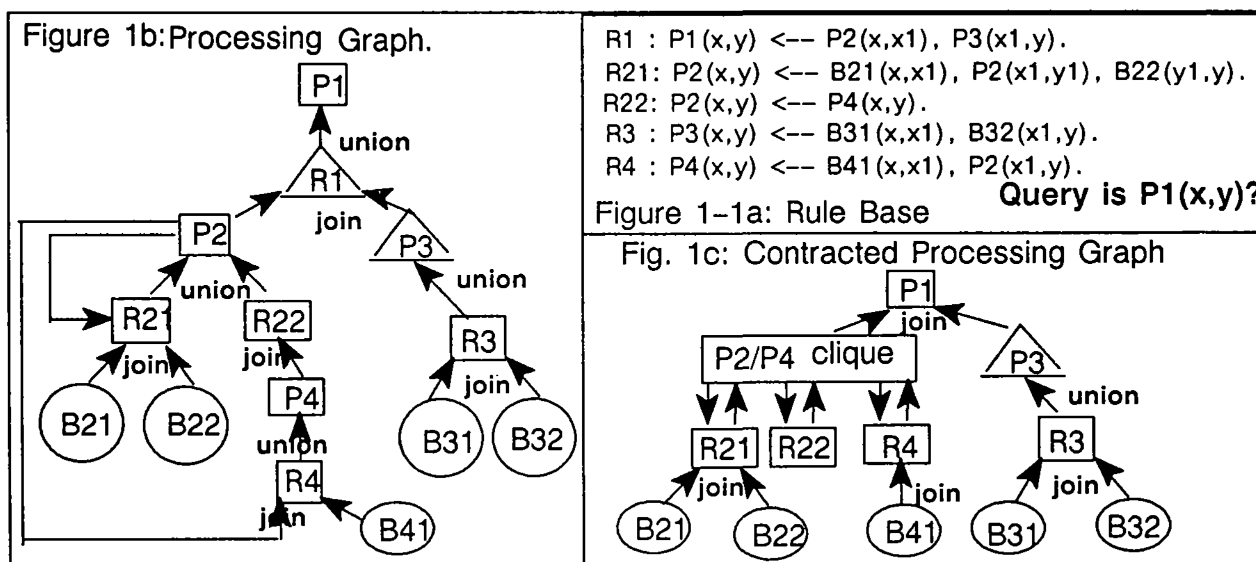
We report on the issues addressed in the design of the optimizer for the Logic Data Language (LDL) that is being designed and implemented at MCC. In particular we motivate the new set of problems posed in this scenario and discuss one possible solution approach to tackle them.

1. Introduction

The Logic Data Language, LDL, combines the expressive power of a high-level logic-based language (e.g., Prolog) with the non-navigational style of relational query languages, where the user need only supply a query (stated logically), and the system (i.e., the compiler) is expected to devise an efficient execution strategy for it. Consequently, the query optimizer is delegated the responsibility of choosing an optimal execution.--a function similar to that of an optimizer in a relational database system. The optimizer uses the knowledge of storage structures, information about database statistics, estimation of cost, etc. to predict the cost of various execution schemes chosen from a pre-defined search space, and selects a minimum cost execution.

As compared to relational queries, LDL queries pose a new set of problems which stem from the following observations. First, the model of data is enhanced to include complex objects; e.g., hierarchies, heterogeneous data allowed for an attribute [Z 85]. Secondly, new operators are needed not only to operate on complex data, but also to handle new operations such as recursion, negation, etc. Thus, the complexity of data as well as the set of operations emphasize the need for new database statistics and new estimations of cost. Finally, the use of evaluable functions, and function symbol [TZ 86] in conjunction with recursion, provides the ability to state queries that are *unsafe* (i.e., do not terminate). As unsafe executions are a limiting case of poor executions, the optimizer must guarantee the choice of a safe execution.

The knowledge base consists of a *rule base* and a *database*. An example of a rule base is given in Figure 1. Throughout this paper, we follow the notational convention that Pi's, Bi's, and f's are *(derived) predicates, base predicates* (i.e., predicate on a base relation), and *function symbols*, respectively. The tuples in the relation corresponding to the Pi's are computed using the rules. Note that each line in Figure 1a is a rule that contains a *head* (i.e., the predicate to the left of the arrow) and the *body* that defines the tuples that are contributed by this rule to the head predicate. A rule may be *recursive* (e.g., R21), in the sense that the definition in the body may depend on the predicate in the head, either directly by reference or transitively through a predicate referenced in the body.



In a given rule base, we say that *P -> Q*, if there is a rule with *Q* as the head predicate and the predicate *P* in the body, or there exists a *P'* where *P->P'* and *P'->Q* (transitivity). Then a predicate *P*, such that *P->P*, will be called *recursive*. Two predicates, *P* and *Q* are called *mutually recursive if P->Q and Q ->P.* This implication relationship is used to partition the recursive predicates into disjoint subsets called *recursive cliques*. A clique C1 is said to *follow* another clique C2 if there exists a recursive predicate in C2 that is used to define the clique C1. Note that the follow relation is a partial order.

In a departure from previous approaches to compilation of logic [KT 81, U 85, N 86], we make our optimization query-specific. A predicate P1(c,y), (in which c and y denote a bound and unbound argument respectively), computes all tuples in P1 that satisfies the constant, c. A *binding* for a predicate is the bound/unbound pattern of its arguments, for which the predicate is computed. Throughout this paper we use x,y to denote variables and c to denote a constant. A predicate with a binding is called a *query form* (e.g., P1(c,y)?). We say that the optimization is query-specific because the algorithm is repeated for each such query form. For instance, P1(x,y)? will be compiled and optimized separately from P1(c,y)?. Indeed the execution strategy chosen for P1(c,y)? may be inefficient (or even unsafe) for P1(x,y)?.

In this paper we limit the discussion to the problem of optimizing the pure fixpoint semantics of Horn clause queries [Llo 84]. In Section 2, the optimization is characterized as a minimization problem based on a cost function over an execution space. This model is used in the rest of the paper to discuss the issues. In Section 3, we discuss the problems in the choice of a search space. The cost model considerations are discussed in section 4. The problem of safety is addressed in section 5.

2. Model

An execution is modelled as a '*processing graph*', which describes the decisions regarding the methods for the operations, their ordering, and the intermediate relations to be materialized. The set of logically equivalent processing graphs is defined to be the *execution space* over which the optimization is performed using a cost model, which associates a cost for each execution.

2.1. Execution Model

An execution is represented by an AND/OR graph such as that shown in Figure 1b for the example of Figure 1a. This representation is similar to the predicate connection graph [KT 81], or rule graph [U 85], except that we give specific semantics to the internal nodes as described below. In keeping with our relational algebra based execution model, we map each AND node into a *join* and each OR node into a *union*. Recursion is implied by an edge to an ancestor or a node in the sibling subtree. A *contraction* of a clique is the extrapolation of the traditional notion of an edge contraction in a graph. An edge is said to be *contracted* if it is deleted and its ends (i.e., nodes) are identified (i.e., merged). A clique is said to be *contracted* if all the edges of the clique are contracted. Intuitively, the contraction of a clique consists of replacing the set of nodes in the clique by a single node and associating all the edges in/out of any node in the clique with this new node (as in Figure 1c).

Associated with each node is a relation that is computed from the relations of its predecessors, by doing the operation (e.g., join, union) specified in the label. We use a square node to denote materialization of relations and a triangle node to denote the pipelining of the tuples. A pipelined execution, as the name implies, computes each tuple one at a time. In the case of join, this computation is evaluated in a lazy fashion as follows: a tuple for a subtree is generated using the binding from the result of the subquery to the *left* of that subtree. This binding is referred to as *binding implied by the pipeline.* Note that we impose a *left to right* order of execution. This process of using information from the sibling subtrees was called *sideways information passing* in [U 85]. Subtrees that are rooted under a materialized node are computed bottom-up, without any sideways information passing; i.e., the result of the subtree is computed completely before the ancestor operation is started.

Each interior node in the graph is also labeled by the method used (e.g., join method, recursion methods etc.). The set of labels for these nodes are restricted *only* by the availability of the techniques in the system. Further, we also allow the result of computing a subtree to be filtered through a selection/restriction predicate. We extend the labeling scheme to encode all such variations due to filtering.

In summary, an execution is modeled as a processing graph. The set of all logically equivalent processing graphs, $\mathcal{PG}$, (for a given query) defines the *execution space* and thus defining the search space for the optimization problem. In order to find practical solutions, we would like to restrict our search space to the space defined by the following equivalence-preserving transformations:

1) **MP:** *Materialize/Pipeline:* A pipelined node can be changed to a materialized node and vice versa.

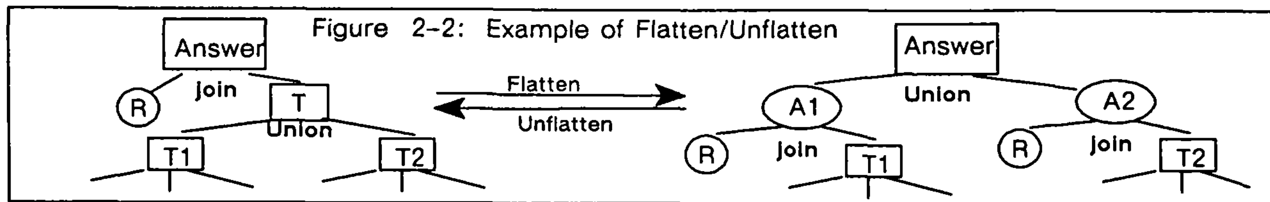


Figure 2-2: Example of Flatten/Unflatten

2) **FU:** *Flatten/Unflatten:* Flattening distributes a join over union. The inverse transformation will be called unflatten. An example of this is shown in Figure 2.
3) **PS:** *PushSelect/PullSelect:* A select can be piggy-backed to a materialized or pipelined node and applied to the tuples as they are generated. Selects can be pushed into a nonrecursive operator (i.e., join or union that is not a part of a recursive cycle) in the obvious way.
4) **PP:** *PushProject/PullProject*: This transformation can be defined similar to the case of select.
5) **PR:** *Permute*: This transforms a given subtree by permuting the order of the subtrees. Note that the inverse of a permutation is defined by another permutation.

Each of the above transformational rules map a processing graph into another equivalent processing graph, and is also capable of mapping vice versa. We define an equivalence relation under a set of transformational rules T as follows: a processing graph p1 is equivalent to p2 under T if p2 can be obtained by zero or more applications of rules in T. Since the equivalence class (induced by said equivalence relation) defines our execution space, we can denote an execution space by a set of transformations, e.g., {MP, PS, PR}.

2.2. Cost Model:

The cost model assigns a cost to each processing graph, thereby ordering the executions. Typically, the costs of all executions in an execution space span many orders of magnitude. Thus "it is more important to avoid the worst executions than to obtain the best execution", a maxim widely assumed by query optimizer designers. Experience with relational systems has shown that even an inexact cost model can achieve this goal reasonably well. The cost includes CPU, disk I/O, communication, etc, which are combined into a single cost that is dependent on the particular system [D 82]. We assume that a list of methods is available for each operation (join, union and recursion), and for each method, we also assume the ability to compute the associated cost and the resulting cardinality.

Intuitively, the cost of an execution is the sum of the cost of individual operations. In the case of nonrecursive queries, this amounts to summing up the cost for each node. As cost models are system- dependent, we restrict our attention in this paper to the problem of estimating the number of tuples in the result of an operation. For the sake of this discussion, the cost can be viewed as some monotonically increasing function on the size of the operands. As the cost of an unsafe execution is to be modeled by an infinite cost, the cost function should guarantee an infinite cost if the size approaches infinity. This is used to encode the unsafe property of the execution.

2.3. Optimization Problem:

We formally define the optimization problem as follows: "Given a query Q, an execution space E and a cost model defined over E,find a processing graph pg in E that is of minimum cost. " It is easy to see that an algorithm exists that enumerates the execution space and finds the execution with a minimum cost. The main problem is to find an efficient strategy to search this space. In the rest of the paper, we use the model presented in this section to discuss issues and design decisions relating to three aspects of the optimization problem: search space, cost model, and safety.

3. Search space:

In this section, we discuss the problem of choosing the proper search space. The main trade-off here is that a very small search space will eliminate many efficient executions, whereas a large search space will render the problem of optimization intractable. We present the discussion by considering the search spaces for queries of increasing complexity: conjunctive queries, nonrecursive queries, and then recursive queries.

3.1. Conjunctive queries:

The search space of a conjunctive query can be viewed based on the ordering of the joins (and therefore the relations) [Sel 79]. The gist of the relational optimization algorithm is as follows: " For each permutation of the set of relations, choose a join method for each join and compute the cost. The

result is the minimum cost permutation." This approach is based on the fact that, for a given ordering of joins, a selection or projection can be pushed to the first operation on a relation without any loss of optimality. Consequently, the actual search space used by the optimizer reduces to {MP, PR}, yet the chosen minimum cost processing graph is optimal in the execution space defined by {MP, PR, PS, PP}. Further, the binding implied by pipelining will also be treated as selections and handled in a similar manner. Note that the definition of the cost function for each individual join, the number of available join methods, etc. are orthogonal to the definition of the optimization problem. This approach, taken in this traditional context, essentially enumerates a search space that is combinatoric on *n*, the number of relations in the conjunct. The dynamic programming method presented in [Sel 79] only improves this to *O(n*(2**n))* time by using *O(2**n)* space. Consequently, database systems (e.g., SQL/DS, commercial INGRES) limit the queries to no more than 10 or 15 joins, so as to be *reasonably efficient.*

In logic queries it is expected that the number of relations can easily exceed 10-15 relations. In [KBZ 86], we presented a quadratic time algorithm that computes the optimal ordering of conjunctive queries when the query is acyclic. Further, this algorithm was extended to include cyclic queries and other cost models. Moreover, the algorithm has proved to be heuristically very effective for cyclic queries once the minimum cost spanning tree is used as the tree query for optimization [V 86].

Another approach to searching the large search space is to use a stochastic algorithm. Intuitively, the minimum cost permutation can be found by picking, randomly, a "large" number of permutations from the search space and choosing the minimum cost permutation. Obviously, the number of permutations that need to be chosen approaches the size of the search space for a reasonable assurance of obtaining the minimum. This number is claimed to be much smaller by using a technique called simulated annealing[IW 87] and this technique can be used in the optimization of conjunctive queries.

In summary, the problem of enumerating the search space is considered the major problem here.

3.2. Nonrecursive Queries:

We first present a simple optimization algorithm for the execution space{MP,PS,PP,PR} (i.e., any flatten/unflatten transformation is disallowed), using which the issues are discussed. As in the case of conjunctive query optimization, we push select/project down to the first operation on a relation and limit the enumeration to {MP,PR}. Recall that the processing graph for any execution of a nonrecursive query is an AND/OR tree.

First consider the case when we materialize the relation for each predicate in the rule base. As we do not allow the flatten/unflatten transformation, we can proceed as follows: optimize a lowest subtree in the AND/OR tree. This subtree is a conjunctive query, as all children in this subtree are leaves (i.e., base relations), and we may use the exhaustive case algorithm of the previous section. After optimizing the subtree, we replace the subtree by a "base relation" and repeat this process until the tree is reduced to a single node. It is easy to show that this algorithm exhausts the search space {PR}. Further, such an algorithm is reasonably efficient if number of predicates in the body does not exceed 10-15.

In order to exploit sideways information passing by choosing pipelined executions, we make the following observation. Because all the subtrees were materialized, the binding pattern (i.e., all arguments unbound) of the head of any rule was uniquely determined. Consequently, we could outline a bottom-up algorithm using this unique binding for each subtree. If we do allow pipelined execution, then the subtree may be bound in different ways, depending on the ordering of the siblings of the root of the subtree. Consequently, the subtree may be optimized differently. Observe that the number of binding patterns for a predicate is purely dependent on the number of arguments of that predicate. So the extension to the above bottom-up algorithm is to optimize each subtree for all possible bindings and to use the cost for the appropriate binding when computing the cost of joining this subtree with its siblings. The maximum number of bindings is equal to the cardinality of the power set of the arguments. In order to avoid optimizing a subtree with a binding pattern that may never be used, a top-down algorithm can be devised. In any case, the algorithm is expected to be reasonably efficient for small numbers of arguments, *k,* and of predicates in the body, *n.*

When *k* and/or *n* are very large, it may not be feasible to use this algorithm. We expect that *k* is unlikely to be large, but there may be rule bases that have large *n*. It is then possible to use the polynomial time algorithm or the stochastic algorithm presented in the previous section. Even though we do not expect *k* to be very large, it would be comforting if we can find an approximation for this case too. This remains a topic for further research.

In summary, the technique of pushing select/project in a greedy way for a given ordering (i.e., a sideways information passing) can be used to reduce the search space to {MP, PR} as was done in the

conjunctive case. Subsequently, an intelligent top-down algorithm to exhaust this search space can be used that is reasonably efficient. But this approach disregards the flatten/unflatten transformation. Enumerating the search space including this transformation is an open problem. Observe that the sideways information passing between predicates was done greedily; i.e., all arguments that can be bound are bound. An interesting open question is to investigate the potential benefits of partial binding, especially when flattening is allowed and common subexpressions are important.

3.3. Recursive queries:

We have seen that pushing selection/projection is a linchpin of non-recursive optimization methods. Unfortunately, this simple technique is inapplicable to recursive predicates [AU 79]. Therefore a number of specialized implementation methods have been proposed to allow recursive predicates to take advantage of constants or bindings present in the goal. (The interested reader is referred to [BR 85] for an overview.) Obviously, the same techniques can be used to incorporate the notion of pipelining (i.e., sideways information passing). In keeping with our algebra-based approach however, we will restrict our attention to *fixpoint* methods, i.e., methods that implement recursive predicates by means of a *least fixpoint operator.* The *magic set method* [BMSU 85] and *generalized counting method* [SZ 86] are two examples of fixpoint methods.

We extend the algorithm presented in the previous section to include the capability to optimize a recursive query, using a divide and conquer approach. Note that all the predicates in the same recursive clique must be solved together--they cannot be solved one at a time. In the processing graph, we propose to contract a recursive clique into a single node (materialized or pipelined) that is labeled by the recursion method used (e.g., magic set, counting). The fixpoint of the recursion is to be obtained as a result of the operation implied by the clique node. Note that the cost of this fixpoint operation is a function of the cost/size of the subtrees and the method used. We assume such cost functions are available for the fixpoint methods. The problem of constructing such functions are discussed in the next section.

The bottom-up optimization algorithm is extended as follows: choose a clique that does not follow any other clique. For this clique, use a nonrecursive optimization algorithm to optimize and estimate the cost and size of the result for all possible bindings. Replace the clique by a single node with the estimated cost and size and repeat the algorithm. In Figure 3 we have elucidated this approach for a single-clique example. Note that in Figure 3b the subtree under P3 is computed using sideways information from the recursive predicate P2; whereas in Figure 3c, the subtree under the recursive predicate is computed using the sideways information from the P3. Consequently, the tradeoffs are cost/size of the recursive predicate P2 versus the cost/size of P3. If, eavluating the recursion is much more expensive than nonrecursive part of the query and the result of P3 is restricted to a small set of tuples, then Figure 3c is a better choice.

Unlike in the non-recursive case, there is no claim of completeness presented here. However, it is our intuitive belief that the above algorithm enumerates a majority of the interesting cases. An example of the incompleteness is evident for the fact that the ordering of the recursive predicates from the same clique is not enumerated by the algorithm. Thus, an important open problem is to devise a reasonably efficient enumeration of a well-defined search space. Another serious problem is the lack

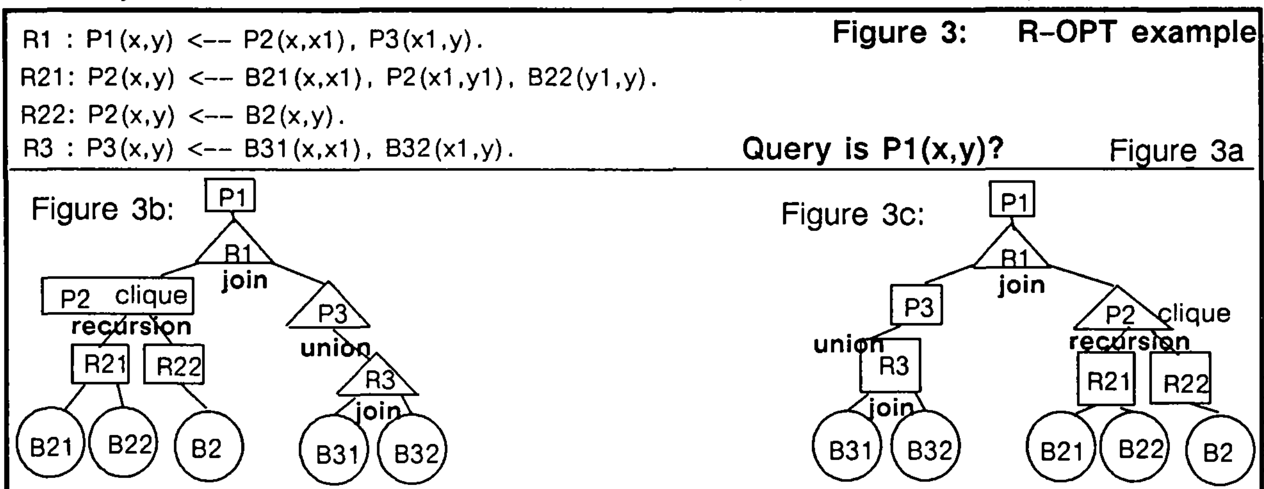


Figure 3: R-OPT example

of intuition in gauging the importance of various types of recursion, which leads to treating all as equally important.

4. Cost Model:

As mentioned before, we restrict our attention to the problem of estimating the number of tuples in the result of an operation. Two problems discussed here are: the estimation for operations on complex objects, and the estimation of the number of iterations for the fixpoint operator (i.e., recursion).

Let the employee object be a set of tuples whose attributes are Name, Position, and Children, where Children is itself a set of tuples each containing the attributes Cname, and Age. All other attributes are assumed to be elementary and the structure is a tree (i.e., not a graph). The estimations for selection, projection, and join have to be redefined in this context as well as defining new formulae for flattening and grouping. One approach is to redefine the cardinality information required from the database. In particular, define the notion of bag cardinality for the complex attributes. The bag cardinality of the children attribute is the cardinality of the bag of all children of all employees, where bag is a set in which duplicates are not removed. Thus, average number of children per employee can be determined by the ratio of the bag cardinality of the children to the cardinality of the employees. In other words, complex attributes have the bag cardinality information associated while the elementary attributes have the set cardinality information associated. Using these new statistics for the data, new estimation formulas can be derived for all the operations, including operations such as flattenning and grouping which restructure the data. In short, the problem of estimating the result of the operations on complex objects can be viewed in two ways: 1) inventing new statistics to be kept to enable more accurate estimations; 2) refining/devising formulae to obtain more accurate estimations.

The problem of estimating the result of recursion can be divided into two parts: first, the problem of estimating the number of iterations of the fixpont operator; second, the number of tuples produced in each iteration. The tuples produced by each iteration is the result of a single application of the rules, and therefore the estimation problem reduces to the case of simple joins. To understand the former problem, consider the example of computing all the ancestors of all persons for a given Parent relation. Intuitively, this is the transitive closure of the corresponding graph. So we can restate the question of estimating the number of iterations to be the estimation of the diameter of the graph. Formula for estimating the diameter for a graph parameterized by the number of edges, fan-out/fan-in, number of nodes etc. have been derived using both analytical and simulation models. Preliminary results show that a very crude estimation can be made using only the number of edges and number of nodes in the graph. Refinement of this estimation is the subject of on-going research. In general, any linear recursion can be viewed in this graph formalism, and the result can be applied to estimate the number of iterations. In short, formulae for estimating the diameter of the graph are needed to estimate the number of iterations of the fixpoint operator. The open questions are the parameters of the graph, the estimation of these parameters for a given recursion, and extension to complex recursions such as mutual recursion.

5. Safety Problem:

Safety is a serious concern in implementing Horn clause queries. Any evaluable predicates (e.g., comparison predicates like x>y, x=y+y*z), and recursive predicates with function symbols are examples of potentially unsafe predicates. While an evaluable predicate will be executed by calls to built-in routines, they can be formally viewed as infinite relations defining, e.g., all the pairs of integers satisfying the relationship x>y, or all the triplets satisfying the relationship x=y+y*z [TZ 86]. Consequently, these predicates may result in unsafe executions in two ways: 1) the result of the query is infinite; 2) the execution requires the computation of a rule resulting in an infinite intermediate result. The former is termed the lack of *finite answer* and the latter the lack of *effective computability or EC*. Note that the answer may be finite even if a rule is not effectively computable. Similarly, the answer of a recursive predicate may be infinite even if each rule defining the predicate is effectively computable.

5.1. Checking for safety:

Patterns of argument bindings that ensure EC are simple to derive for comparison predicates. For instance, we can assume that for comparison predicates other than equality, all variables must be bound before the predicate is safe. When equality is involved in a form "*x= expression*", then we are ensured of EC as soon as all the variables in *expression* are instantiated. These are only sufficient conditions and more general ones - e.g., based on combinations of comparison predicates - could be given (see for instance [M 84]). But for each extension of a sufficient condition, a rapidly increasing

price would have to be paid in the algorithms used to detect EC and in the system routines used to support these predicates at run time. Indeed, the problem of deciding EC for Horn clauses with comparison predicates is undecidable [Z 85], even when no recursion is involved. On the other hand, EC based on safe binding patterns is easy to detect. Thus, deriving more general sufficient conditions for ensuring EC that is easy to check is an important problem facing the optimizer designer.

Note that if all rules of a nonrecursive query are effectively computable, then the answer is finite. However, for a recursive query, each bottom-up application of any rule may be effectively computable, but the answer may be infinite due to unbounded iterations required for a fixpoint operator. In order to guarantee that the number of iterations are finite for each recursive clique, a well-founded order (also known as Noetherian order[B 40]) based on some monotonicity property must be derived. For example, if a list is traversed recursively, then the size of the list is monotonically decreasing with a bound of an empty list. This forms the well-founded condition for termination of the iteration. In [UV 86], some methods to derive the monotonicity property are discussed. In [KRS 87], an algorithm to ensure the existence of a well-founded condition is outlined. As these are only sufficient conditions, they do not necessarily detect all safe executions. Consequently, more general monotonicity properties must be either inferred from the program or declared by the user in some form. These are topics of future research.

5.2. Searching for Safe Executions:

As mentioned before, the optimizer enumerates all the possible permutations of the goals in the rules. For each permutation, the cost is evaluated and the minimum cost solution is maintained. All that is needed to ensure safety is that EC is guaranteed for each rule and a well founded order is associated with each recursive clique. If both these tests succeed, then the optimization algorithm proceeds as usual. If the tests fails, the permutation is discarded. In practice this can be done by simply assigning an extremely high cost to unsafe goals and then let the standard optimization algorithm do the pruning. If the cost of the end-solution produced by the optimizer is not less than this extreme value, a proper message must inform the user that the query is unsafe.

5.3 Comparison with Previous Work

The approach to safety proposed in [Na 85] is also based on reordering the goals in a given rule; but that is done at run-time by delaying goals when the number of instantiated arguments is insufficient to guarantee safety. This approach suffers from run-time overhead, and cannot guarantee termination at compile time or otherwise pinpoint the source of safety problems to the user -- a very desirable feature, since unsafe programs are typically incorrect ones. Our compile-time approach overcomes these problems and is more amenable to optimization.

The reader should, however, be aware of some of the limitations implicit in all approaches based on reordering of goals in rules. For instance a query: p(x, y, z), y= 2*x ?, with the rule

p(x, y, z) <-- x=3, z=x*y

is obviously finite (x=3, y=6, z=18), but cannot be computed under any permutation of goals in the rule. Thus both Naish's approach and the above optimization cum safety algorithm will fail to produce a safe execution for this query. Two other approaches, however, will succeed. One, described in [Z 86], determines whether there is a finite domain underlying the variables in the rules using an algorithm based on a functional dependency model. Safe queries are then processed in a bottom up fashion with the help of "magic sets", which make the process safe. The second solution consists in flattening , whereby the three equalities are combined in a conjunct and properly processed in the obvious order refered to earlier.

6. Conclusion

The main strategy we have studied proposes to enumerate exhaustively the search space, defined by the given AND/OR graph, to find the minimum cost execution. One important advantage of this approach is its total flexibility and adaptability. We perceive this to be a critical advantage, as the field of optimization of logic is still in its infancy, and we plan to experiment with an assortment of techniques, including new and untested ones.

A main concern with the exhaustive search approach is its exponential time complexity. While this should become a serious problem only when rules have a large number of predicates, alternate efficient search algorithms can supplement the exhaustive algorithm (i.e., using it only if necessary), and also these alternate algorithms should make extensive flattening of the given AND/OR graph practically feasible. We are currently investigating the effectiveness of these alternatives.

Among the optimization aspects not covered in this paper we find that of common subexpression elimination[GM 82], which appears particularly useful when flattening occurs. A simple technique using a hill-climbing method is easy to superimpose on the proposed strategy, but more ambitious technique provide a topic for future research. Further, an extrapolation of common subexpression in logic queries can be seen in the following example: let both goals P(a,b,X) and P(a,Y,c) occur in a query. Then it is conceivable that computing P(a,Y,X) once and restricting the result for each of the cases may be more efficient.

Acknowledgments: We are grateful to Shamim Naqvi for inspiring discussions during the development of an earlier version of this paper.

References:

[AU 79] Aho, A. and J. Ullman, Universality of Data Retrieval Languages, *Proc. POPL Conf.*, San Antonio, TX, 1979.

[B 40] Birkhoff, G., " Lattice Theory", American Mathematical Society, 1940.

[BMSU85] Bancilhon, F., D, Maier, Y. Sagiv and Ullman, Magic Sets and other Strange Ways to Implements Logic Programs, *Proc. 5-th ACM SIGMOD-SIGACT Symposium on Principles of Database Systems*, pp. 1-16, 1986.

[BR 86] Bancilhon, F., and R. Ramakrishan, An Amateur's Introduction to Recursive Query Processing Strategies, *Proc. 1986 ACM-SIGMOD Intl. Conf. on Mgt. of Data*, pp. 16-52, 1986.

[D 82] Daniels, D., et. al., "An Introduction to Distributed Query Compilation in R*", Proc. of Second International Conf, on Distriuted Databases, Berlin, Sept. 1982.

[GM 82] Grant, J. and Minker J., On Optimizing the Evaluation of a Set of Expressions, *Int. Journal of Computer and Information Science, 11, 3 (1982), 179-189.*

[IW 87] Ioannidis, Y. E, Wong, E, Query Optimization by Simulated Annealing, SIGMOD 87, San Francisco.

[KBZ 86] Krishnamurthy, R., Boral, H., Zaniolo, C. Optimization of Nonrecursive Queries, *Proc. of 12th VLDB*, Kyoto, Japan, 1986.

[KRS 87] Krishnamurthy, R, Ramakrishnan, R, Shmueli, O., "Testing for Safety and Effective Computability", Manuscript in Preparation.

[KT 81] Kellog, C., and Travis, L. Reasoning with data in a deductively augmented database system, in *Advances in Database Theory: Vol 1*, H.Gallaire, J. Minker, and J. Nicholas eds., Plenum Press, New York, 1981, pp 261-298.

[Llo 84] Lloyd, J. W., *Foundations of Logic Programming*, Springer Verlag, 1984.

[M 84] Maier, D., *The Theory of Relational Databases*, (pp. 542-553), Comp. Science Press, 1984.

[Na 86] Naish, L., Negation and Control in Prolog *Journal of Logic Programming*, to appear.

[Sel 79] Sellinger, P.G. et. al. Access Path Selection in a Relational Database Management System., *Proc. 1979 ACM-SIGMOD Intl. Conf. on Mgt. of Data*, pp. 23-34, 1979.

[SZ 86] Sacca', D. and C. Zaniolo, The Generalized Counting Method for Recursive Logic Queries, *Proc. ICDT '86 --Int. Conf. on Database Theory*, Rome, Italy, 1986.

[TZ 86] Tsur, S. and C. Zaniolo, LDL: A Logic-Based Data Language,*Proc. of 12th VLDB*, Kyoto, Japan, 1986.

[U 85] Ullman, J. D., Implementation of logical query languages for databases, *TODS*, 10, 3, (1985), 289-321.

[UV 85] Ullman, J.D. and A. Van Gelder, Testing Applicability of Top-Down Capture Rules, Stanford Univ. Report STAN-CS-85-146, 1985.

[V 86] Villarreal, M., "Evaluation of an O(N**2) Method for Query Optimization", MS Thesis, Dept. of Computer Science, Univ. of Texas at Austin, Austin, TX.

[Z 85] Zaniolo, C. The representation and deductive retrieval of complex objects, *Proc. of 11th VLDB*, pp. 458-469, 1985.

[Z 86] Zaniolo, C., Safety and Compilation of Non-Recursive Horn Clauses, *Proc. First Int. Conf. on Expert Database Systems*, Charleston, S.C., 1986.

OPTIMIZATION OF COMPLEX DATABASE QUERIES USING JOIN INDICES

Patrick Valduriez

Microelectronics and Computer Technology Corporation
3500 West Balcones Center Drive
Austin, Texas 78759

ABSTRACT

New application areas of database systems require efficient support of complex queries. Such queries typically involve a large number of relations and may be recursive. Therefore, they tend to use the join operator more extensively. A join index is a simple data structure that can improve significantly the performance of joins when incorporated in the database system storage model. Thus, as any other access method, it should be considered as an alternative join method by the query optimizer. In this paper, we elaborate on the use of join indices for the optimization of both non-recursive and recursive queries. In particular, we show that the incorporation of join indices in the storage model enlarges the solution space searched by the query optimizer and thus offers additional opportunities for increasing performance.

1. Introduction

Relational database technology can well be extended to support new application areas, such as deductive database systems [Gallaire 84]. Compared to the traditional applications of relational database systems, these applications require the support of more complex queries. Those queries generally involve a large number of relations and may be recursive. Therefore, the quality of the query optimization module (query optimizer) becomes a key issue to the success of database systems.

The ideal goal of a query optimizer is to select the optimal access plan to the relevant data for an input query. Most of the work on traditional query optimization [Jarke 84] has concentrated on select-project-join (SPJ) queries, for they are the most frequent ones in traditional data processing (business) applications. Furthermore, emphasis has been given to the optimization of joins [Ibaraki 84] because join remains the most costly operator. When complex queries are considered, the join operator is used even more extensively for both non-recursive queries [Krishnamurthy 86] and recursive queries [Valduriez 86a].

In [Valduriez 87], we proposed a simple data structure, called a join index, that improves significantly the performance of joins. In this paper, we elaborate on the use of join indices in the context of non-recursive and recursive queries. We view a join index as an alternative join method that should be considered by the query optimizer as any other access method. In general, a query optimizer maps a query expressed on conceptual relations into an access plan, i.e., a low-level program expressed on the physical schema. The physical schema itself is based on the storage model, the set of data structures available in the database system. The incorporation of join indices in the storage model enlarges the solution space searched by the query optimizer, and thus offers additional opportunities for increasing performance.

Join indices could be used in many different storage models. However, in order to simplify our discussion regarding query optimization, we present the integration of join indices in a simple storage model with single attribute clustering and selection indices. Then we illustrate the impact of the storage model with join indices on the optimization of non-recursive queries, assumed to be SPJ queries. In particular, efficient access plans, where the most complex (and costly) part of the query can be performed through indices, can be generated by the query optimizer. Finally, we illustrate the use of join indices in the optimization of recursive queries, where a recursive query is mapped into a program of relational algebra enriched with a transitive closure operator.

2. Storage Model with Join Indices

The storage model prescribes the storage structures and related algorithms that are supported by the database system to map the conceptual schema into the physical schema. In a relational system implemented on a disk-based architecture, conceptual relations can be mapped into base relations on the basis of two functions, partitioning and replicating. All the tuples of a base relation are clustered based on the value of one attribute. We assume that each conceptual tuple is assigned a surrogate for tuple identity, called a TID (tuple identifier). A TID is a value unique for all tuples of a relation. It is created by the system when a tuple is instantiated. TID's permit efficient updates and reorganizations of base relations, since references do not involve physical pointers. The partitioning function maps a relation into one or more base relations, where a base relation corresponds to a TID together with an attribute, several attributes, or all the conceptual relation's attributes. The rationale for a partitioning function is the optimization of projection, by storing together attributes with high affinity, i.e., frequently accessed together. The replicating function replicates one or more attributes associated with the TID of the relation into one or more base relations. The primary use of replicated attributes is for optimizing selections based on those attributes. Another use is for increased reliability provided by those additional data copies.

In this paper, we assume a simple storage model where the partitioning function is identity, defining a primary copy, and the replicating function defines one or more *selection indices*. The primary copy of a relation R(A, B, ...) is a base relation F(TID, A, B, ...) clustered on TID. Clustering is based on a hashed or tree structured organization. A selection index on attribute A of relation R is a base relation F(A, TID) clustered on A.

Let R_1 and R_2 be two relations, not necessarily distinct, and let TID_1 and TID_2 be identifiers of tuples of R_1 and R_2, respectively. A *join index* on relations R_1 and R_2 is a relation of couples (TID_1, TID_2), where each couple indicates two tuples matching a join predicate. Intuitively, a join index is an abstraction of the join of two relations. A join index can be implemented by two base relations F(TID_1, TID_2), one clustered on TID_1 and the other on TID_2. Join indices are uniquely designed to optimize joins.

The join predicate associated with a join index may be quite general and include several attributes of both relations. Furthermore, more than one join index can be defined between any two relations. The identification of various join indices between two relations is based on the associated join predicate. Thus, the join of relations R_1 and R_2 on the predicate ($R_1.A = R_2.A$ and $R_1.B = R_2.B$) can be captured as either a single join index, on the multi-attribute join predicate, or two join indices, one on ($R_1.A = R_2.A$) and the other on ($R_1.B = R_2.B$). The choice between the alternatives is a database design decision based on join frequencies, update overhead, etc.

Let us consider the following relational database schema (key attributes are bold):

CUSTOMER (**cname**, city, age, job)
ORDER (**cname**, **pname**, qty, date)
PART (**pname**, weight, price, spname)

A (partial) physical schema for this database, based on the storage model described above, is (clustered attributes are bold)

C_PC (**CID**, cname, city, age, job)
City_IND(**city**, CID)
Age_IND (**age**, CID)
O_PC (**OID**, cname, pname, qty, date)
Cname_IND(**cname**, OID)
CID_JI (**CID**, OID)
OID_JI (**OID**, CID)

C_PC and O_PC are primary copies of CUSTOMER and ORDER relations. City_IND and Age_IND are selection indices on CUSTOMER. Cname_IND is a selection index on ORDER. CID_JI and OID_JI are join indices between CUSTOMER and ORDER for the join predicate (CUSTOMER. Cname = ORDER.Cname).

3. Optimization of Non-Recursive Queries

The objective of query optimization is to select an access plan for an input query that optimizes a given cost function. This cost function typically refers to machine resources such as disk accesses, CPU time, and possibly communication time (for a distributed database system). The query optimizer is in charge of decisions regarding the ordering of database operations, and the choice of the access paths to the data, the algorithms for performing database operations, and the intermediate relations to be materialized. These decisions are undertaken based on the physical database schema and related statistics. A set of decisions that lead to an execution plan can be captured by a *processing tree* [Krishnamurthy 86]. A processing tree (PT) is a tree in which a leaf is a base relation and a non-leaf node is an intermediate relation materialized by applying an internal database operation. Internal database operations implement efficiently relational algebra operations using specific access paths and algorithms. Examples of internal database operations are exact-match select, sort-merge join, n-ary pipelined join, semi-join, etc.

The application of algebraic transformation rules [Jarke 84] permits generation of many candidate PT's for a single query. The optimization problem can be formulated as finding the PT of minimal cost among all equivalent PT's. Traditional query optimization algorithms [Selinger 79] perform an exhaustive search of the solution space, defined as the set of all equivalent PT's, for a given query. The estimation of the cost of a PT is obtained by computing the sum of the costs of the individual internal database operations in the PT. The cost of an internal operation is itself a monotonic function of the operand cardinalities. If the operand relations are intermediate relations then their cardinalities must also be estimated. Therefore, for each operation in the PT, two numbers must be predicted: (1) the individual cost of the operation and (2) the cardinality of its result based on the selectivity of the conditions [Selinger 79, Piatetsky 84].

The possible PT's for executing an SPJ query are essentially generated by permutation of the join ordering. With n relations, there are n! possible permutations. The complexity of exhaustive search is therefore prohibitive when n is large (e.g., $n > 10$). The use of dynamic programming and heuristics, as in [Selinger 79], reduces this complexity to 2^n, which is still significant. To handle the case of complex queries involving a large number of relations, the optimization algorithm must be more efficient. The complexity of the optimization algorithm can be further reduced by imposing restrictions on the class of

PT's [Ibaraki 84], limiting the generality of the cost function [Krishnamurthy 86], or using a probabilistic hill-climbing algorithm [Ioannidis 87].

Assuming that the solution space is searched by an efficient algorithm, we now illustrate the possible PT's that can be produced based on the storage model with join indices. The addition of join indices in the storage model enlarges the solution space for optimization. Join indices should be considered by the query optimizer as any other join method, and used only when they lead to the optimal PT.

In [Valduriez 87], we give a precise specification of the join algorithm using join index, denoted by JOINJI, and its cost. This algorithm takes as input two base relations $R_1(TID_1, A_1, B_1, \ldots)$ and $R_2(TID_2, A_2, B_2, \ldots)$, and a join index JI (TID_1, TID_2). The algorithm JOINJI can be summarized as follows:

```
for each pair (tid1, tid2) in JI do
    t1 := read (R1, tid1)
    t2 := read (R2, tid2)
    concatenate the tuples t1 and t2
endfor
```

The actual algorithm optimizes main memory utilization and clustered access to relations R_1 and R_2. The cost of JOINJI can be abstracted in terms of Yao's function [Yao 77]. This function, denoted by Y, gives the expected number of page accesses for accessing k tuples randomly distributed in a relation of n tuples stored in m pages:

$$Y(k, m, n) = m * \left(1 - \prod_{i=1}^{k} \frac{n - (n/m) - i + 1}{n - i + 1}\right)$$

Assuming that relations R_1 and R_2 are clustered on TID, and ignoring the access to indices and the production of the result, the cost of JOINJI can be summarized as:

$$cost\ (JOINJI) = (Y(k_1, |R_1|, ||R_1||) + Y(k_2, |R_2|, ||R_2||)) * IO$$

where k_i is the number of tuples in R_i that participate in the join, $|R_i|$ is the number of pages of R_i, $||R_i||$ is the cardinality of R_i, and IO is the time to read a page on disk.

The combined use of join indices and selection indices may generate PT's that defer to the last part of the query the access of the primary copies of relations (much larger). Let us first consider a type of query that involves a selection and a join. We suppose a join index exists for the join and an selection index exists for the selection. An example of such a query is *"give the name and age of customers in Paris with the part names ordered"*. Many alternative PT's may be found for such a simple query. Figure 1 illustrates two interesting PT's for this query, represented by a query tree on conceptual relations. Both PT's provide clustered accesses. For simplicity, we have left the algorithms unspecified for the operations in the PT's. PT_1 describes a traditional strategy in which only the relevant tuples of CUSTOMER are accessed and joined with the primary copy of relation ORDER using the index on the join attribute. PT_2 describes a strategy based on a join index. The join of the primary copies of CUSTOMER and ORDER is performed with the relevant subset of the join index (semi-joined by the list of relevant CID's) using the algorithm JOINJI. Both PT_1 and PT_2 can dominate depending on select and join selectivities.

Let us now consider a type of query involving multiple joins, possibly with selections. If every join can be processed using a join index, then an interesting PT consists of first joining all the join indices, thus providing all the identifiers of relevant tuples, and finally accessing the relevant tuples based on those identifiers. Therefore, the primary copies of the relations are only accessed through clustered TID's in a final phase.

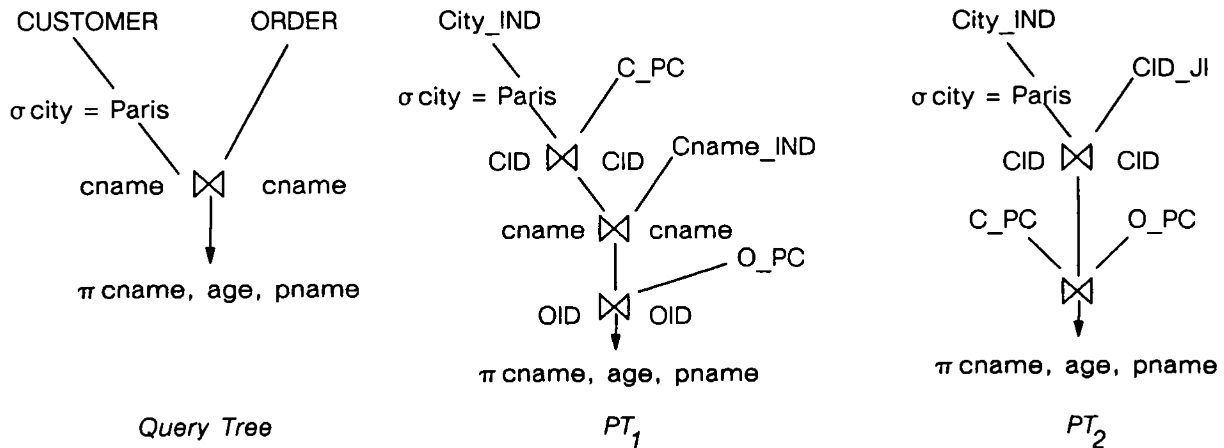


Figure 1: Alternative Processing Trees for A Non-Recursive Query

If not every join can be processed using a join index, then joins with join indices may be combined with more traditional join algorithms. Let us consider the query whose query tree is given in Figure 2. Suppose that only one join index exists for that query. Two cases can occur: there is a join index on ORDER and PART, or a join index on CUSTOMER and ORDER. In the first case, a traditional join precedes the join using join index. The traditional join produces relation R, which is then used both in a semi-join with the join index (to select the relevant subset of the join index) and in the final join using the join index. In the second case, the join using the join index precedes the traditional join. Figure 2 shows the PT's corresponding to each case.

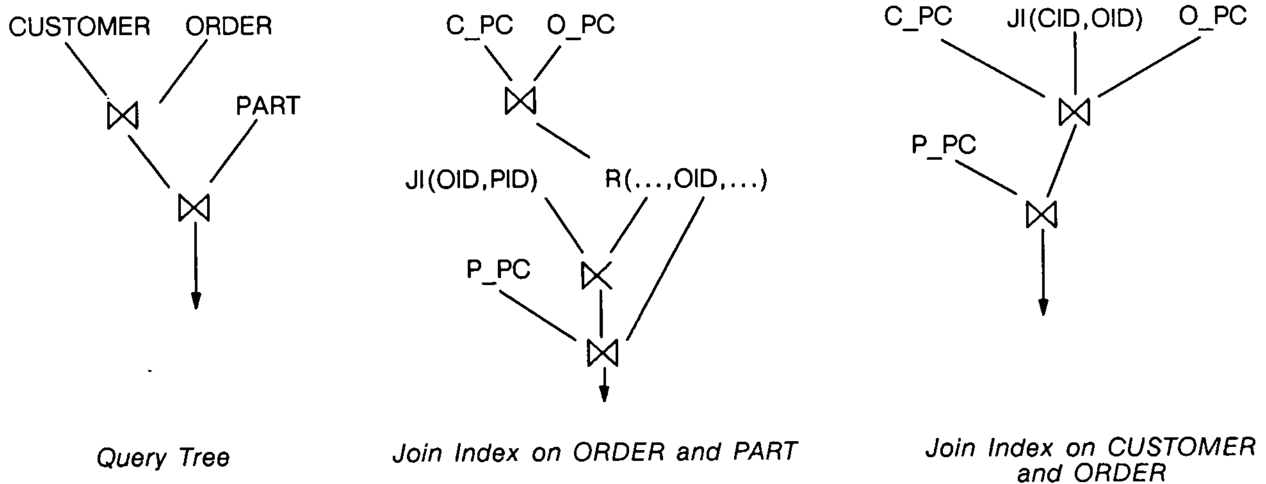


Figure 2: Processing Trees for Different Join Indices

4. Optimization of Recursive Queries

Recursive queries can be mapped into loops of relational algebra operations [Bancilhon 86], where the operations of iteration i use as input the results produced by iteration (i-1). In [Jagadish 87], it is shown that the most important class of recursive queries, called linear queries, can be mapped into programs consisting of relational algebra operations and transitive closure. Thus, the transitive closure operator, extensively used for fix-point computations, is of major importance and requires efficient implementation. In [Valduriez 86a], we have illustrated the value of join indices for optimizing recursive queries, and particularly transitive closure.

A join index captures the semantic links that exist between tuples. If we view the join of two tuples as an arc connecting those tuple identifiers, a join index can represent directed graphs in a very compact way. Therefore, it will be very useful to optimize graph operations like transitive closure. Let us consider again the PART relation:

PART (pname, weight, price, spname)

where spname is the name of a subpart (or component part). Assuming that PID and SPID stand for PART tuple identifiers, then we can have two join indices (each clustered on its first attribute)

JI_1 (**PID**, SPID)
JI_2 (**SPID**, PID)

JI_1 associates a part_id with its subpart_id's, while JI_2 associates a subpart_id with its parent part_id. Therefore, JI_1 is well suited for traversals in the part-subpart direction. JI_2 allows efficient traversals that follow the subpart-part direction.

Assuming that a recursive query is mapped into a conceptual query tree of relational algebra operators and transitive closure, the query optimization algorithm discussed in Section 3 still applies. However, the introduction of transitive closure yields a larger solution space, since transitive closure may be permuted with other relational operators (e.g., select and join). The transitive closure operator can be implemented efficiently by a loop of joins, unions, and possibly difference (for cyclic relations). Superior performance is consistently attained when transitive closure is applied using join index rather than the primary copy of the relation [Valduriez 86a]. For instance, let us consider the recursive query on the PART relation *"list the component parts and their prices for part A"*. Figure 3 illustrates the corresponding query tree and a possible processing tree, in which transitive closure (noted TC) is applied to the join index. The selection "pname = A" precedes the transitive closure so that only those parts that are (transitively) components of part A are produced. In the PT, the result of the transitive closure on join index JI is a set of pairs (PID of A, PID of a subpart of A). Therefore, an additional join with relation PART is necessary to complete the query. Thus, the most complex part of the query is done on small data structures (selection index, join index). The value of performing the transitive closure using join index is to avoid repeated access to relation PART, which is potentially much larger than the join index.

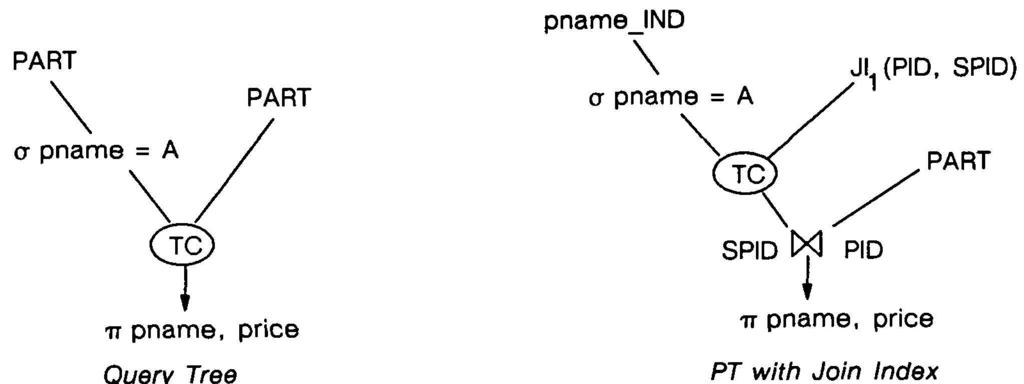


Figure 3: Processing of a Recursive Query with Join Index

5. Conclusion

Join indices are data structures especially designed to speed up join operations. The incorporation of join indices in a storage model provides the query optimizer with a larger solution space and hence more opportunities for optimization. We have illustrated the use of join indices to optimize non-recur-

sive and recursive queries, which typically involve many joins. Using join indices permits the generation of execution strategies in which the complex part of the query can be performed through indices, and the primary copies (base relations) can be accessed in a final phase. Since indices are much smaller than base data, a substantial gain may be obtained.

However, there are cases where classical indexing (selection indices on join attribute) outperforms join indices. First, if the query only consists of a join preceded by a selection with high selectivity, then the indirect access to the join index will incur additional index accesses. Second, join indices require systematic access to the relation primary copy for projection. If the only projected attributes are join attributes, then selection indices on join attribute will save having to access the primary copy. Intuitively, join indices are more suitable for complex queries than for simple queries.Therefore join indices should be considered as an additional access method by the query optimizer.

References

[Bancilhon 86] F. Bancilhon, R. Ramakrishnan, *"An Amateur's Introduction to Recursive Query Processing Strategies"*, ACM-SIGMOD Int. Conf., Washington, D.C., May 1986.

[Gallaire 84] H. Gallaire, J. Minker, and J.M. Nicolas, *"Logic and Database: A Deductive Approach"*, ACM Computing Surveys, Vol. 16, No. 2, June 1984.

[Ibaraki 84] T. Ibaraki, T. Kameda, *"On the Optimal Nesting Order for Computing N-Relation Joins"*, ACM TODS, Vol. 9, No. 3, September, 1984.

[Ioannidis 87] Y.E. Ioannidis, E. Wong, *"Query Optimization by Simulated Annealing"*, ACM-SIGMOD Int. Conf., San Francisco, CA, May 1987.

[Jagadish 87] H.V.Jagadish, R. Agrawal, L. Ness, *"A Study of Transitive Closure as a Recursion Mechanism"*, ACM-SIGMOD Int. Conf., San Francisco, CA, May 1987.

[Jarke 84] M. Jarke and J. Koch *"Query Optimization in Database Systems"*, ACM Computing Surveys, Vol. 16, No. 2, 1984.

[Krishnamurthy 86] R. Krishnamurthy, H. Boral and C. Zaniolo *"Optimization of Non-Recursive Queries"*, Int. Conf. on VLDB, Kyoto, Japan, 1986.

[Piatetsky 84] G. Piatetsky-Shapiro, C. Connell, *"Accurate Estimation of the Number of Tuples Satisfying a Condition"*, ACM-SIGMOD Conf., Boston, MA, 1984.

[Selinger 79] P. Selinger et al., *"Access Path Selection in a Relational Database Management System"*, ACM SIGMOD Conf., Boston, MA, May 1979.

[Valduriez 86a] P. Valduriez and H. Boral, *"Evaluation of Recursive Queries Using Join Indices"*, 1st Int. Conf. on Expert Database Systems, Charleston, SC, 1986.

[Valduriez 86b] P. Valduriez, S. Khoshafian, and G. Copeland, *"Implementation Techniques of Complex Objects"*, Int. Conf. on VLDB, Kyoto, Japan, 1986.

[Valduriez 87] P. Valduriez, *"Join Indices"* ACM TODS, Vol. 12, No. 2, June 1987.

[Yao 77] S.B. Yao, *"Approximating Block Accesses in Database Organizations"*, CACM, Vol. 20, No. 4, April 1977.

QUERY PROCESSING IN OPTICAL DISK BASED MULTIMEDIA INFORMATION SYSTEMS

Stavros Christodoulakis

Department of Computer Science
University of Waterloo
Waterloo, Ontario N2L 3G1
Canada

ABSTRACT

MINOS is a project in multimedia information management. In this project we investigate issues related to the integrated magnetic and optical disk storage management for multimedia objects, content addressibility for multimedia objects, access methods, user interfaces, editing and formatting tools, distributed system aspects, and finally query optimization in a multimedia server environment.

In this project we have implemented and demonstrated a series of prototypes. We are using the prototypes for experimentation and evaluation of our ideas. Currently we are involved in the implementation of a high-performance multimedia object server based on optical disk technology. Optical disks have been chosen because of their ability to store inexpensively large volumes of multimedia information. We are studying various aspects of query processing in such an environment analytically and experimentally. The results of our investigations will be incorporated in our system implementation. In this report we outline our research efforts in multimedia query processing.

Issues in Optical Disk Based Multimedia Query Processing

In the environment described above, a number of new issues and problems in query processing appear. First, performance estimates for retrieval must be derived. Such estimates have to take into account the nature of the storage media (e.g., for optical disks), the distribution of the lengths of the objects in the data base, the selectivities of the queries (mainly text-based), the placement of the qualifying objects on the disk (block boundaries may be crossed), the interactive nature of the retrieval of multimedia objects, as well as the characteristics of the access methods that MINOS uses. These issues and some preliminary results of our studies are described in more detail below.

Retrieval Performance of CAV Optical Disks

Optical disks present different performance characteristics than magnetic disks. For Constant Angular Velocity (CAV) disks, a major performance difference from magnetic disks is the existence of a mirror with small inertia that can be used to deflect the reading beam very fast. As a result, it is much faster to retrieve information from tracks that are located near the current location of the reading head. We call this a **span access capability**. The span access capability of optical disks has implications for scheduling algorithms and data structures that are appropriate for optical disks, as well as significant impact on retrieval performance [Christodoulakis 87a].

In [Christodoulakis 87] we also derive exact analytic cost estimates as well as approximations that are cheaper to evaluate, for the retrieval of records and longer objects such as text, images,voice, and documents (possibly crossing block boundaries) from CAV optical disks. These estimates may be used by query optimizers of traditional or multimedia data bases.

Retrieval Performance of CLV Optical Disks

Constant Linear Velocity (CLV) optical disks have different characteristics than the CAV optical disks. CLV optical disks vary the rotational speed so that the unit length of the track which is read passes under the reading mechanism in constant time, which is independent of the location of the track. This has implications on the rotational delay cost which, in CLV disks, depends on the track location. This also implies that, in CLV disks, the number of sectors per track varies (outside tracks have more sectors). The latter (variable capacity of a track) has many fundamental implications on selection of data structures that are desirable for CLV optical disks and the parameters of their implementation, for the selection of access paths to be supported for data bases stored on CLV disks, as well as for the retrieval performance and the optimal query processing strategy to be chosen. (These implications are studied in detail in [Christodoulakis 87b], in which is shown that these decisions depend on the location of data placement on the disk.)

Analytic cost estimates for the performance of retrieval of records and objects from CLV disks are also derived in ([Christodoulakis 87b]). These estimates may be used by traditional or multimedia query optimizers. It is shown that the optimal query processing strategy depends on the location of files on the CLV disk. This implies that query optimizers may have to maintain information about the location of files on the disk.

Estimation of Selectivities in Text

In multimedia information systems much of the content specification will be done by specifying a pattern of text words. Queries based on the content of images are difficult to specify, and image access methods are very expensive. Voice content is transformed to text content if a good voice recognition

device is available. Thus accurate estimation of text selectivities is important in query optimization in multimedia objects.

There is another important reason why accurate estimation of text selectivities is important. Frequently the user wants to have a fast feedback of how many objects qualify in his query. If too many objects quality, the user may want to restrict the set of qualifying objects by adding more conjunctive terms. If too few objects qualify, the user may want to increase the number of objects that he receives by adding more disjunctive terms. (Tradeoffs of precision versus recall are extensively described in the information retrieval bibliography.) Although such statistics may be found by traversing an index on text (possibly several times for complicated queries) indexes may not be the desirable text access methods in several environments [Haskin 81].

Given a set of stop words (words that appear too frequently in English to be of a practical value in content addressibility), it is easy to give an analytic formula that calculates the **average** number of words that qualify in a text query [Christodoulakis and Ng 87]. This analytic formula uses the fact that the distribution of words in a long piece of text is Zipf with known parameters.

However, the average number of documents may not be a good enough estimate (in some cases) for query optimization or for giving an estimate of the size of the response to the user [Christodoulakis 84]. More detailed estimates will have to consider selectivities of individual words and queries. This can be done using sampling. A sampling strategy looks at some blocks of text, counts the number of occurrences of a particular word or text pattern, and based on this extrapolates the probability distribution of the number of pattern occurrences to the whole data base. A potential problem with this approach is that in order to be confident about the statistics a large portion of the file may have to be scanned.

Instead of blocks of the actual text file, blocks of the text signatures could be used when signatures are used as text access methods. Since more information exists in blocks of signatures than in blocks of the actual text file, fewer blocks would have to be looked at; alternatively, by sampling the same number of signature blocks, sharper probability distributions (for the number of occurrences of the pattern in data base) can be obtained.

Query Optimizers for Interactive Multimedia Retrieval

The multimedia retrieval environment has the following important difference with the traditional data base environments: it is very difficult for the user to specify precisely what he wants to see. (It is difficult, for example, to specify content in images, and there are many synonyms of text words. Voice segments may also be only partially recognized by a voice recognizer.) It is frequently the case that the user has to look carefully within a document in order to decide if a document is relevant or not, and which parts of it are relevant. Retrieving all qualifying documents at once may not be the best query processing strategy, because users frequently quit when they find what they want or when they look at

the first few pages of a document. The time that a user spends on a document may depend on the order of the document in the retrieved ordered set of documents. In addition, users frequently want to reformulate their queries if the filter was not good enough (too many or too few documents qualify). The above observations may have significant impact in the structure of query optimizers for multimedia data. We are experimenting with a user model in order to integrate it in our performance studies [Christodoulakis and Ng 87]. We are also experimenting with several query processing strategies in this environment.

A second aspect of the interactive multimedia retrieval has to do with the retrieval of delay-sensitive data such as voice, video, annimations. For long voice segments for example, it may not be desirable to prefetch all the voice information. This may require many block accesses from secondary storage, and it may occupy large main memory resources for long time intervals. Care however in scheduling must be taken to guarantee that enough voice information is delivered to user workstations so that voice interruptions are minimized. This implies that performance measures used should also take into account delay-sensitive data (data type), unlike traditional data bases.

It is however hard to study reliably the performance of the system using analytical methods in such an environment. We are currently implementing a distributed testbed for multimedia management based on a server architecture [Christodoulakis and Velissaropoulos 87]. The testbed is modular so that we can easily replace components for experimentation. We will be using the testbed for experimenting with various scheduling algorithms and performance measures in a multimedia server environment with delay-sensitive data.

Processing of Multiple Requests

MINOS extensively uses signature techniques as access methods ([Christodoulakis and Faloutsos 84], [Faloutsos and Christodoulakis 87]). Signature files are mainly sequential access methods (however, multilevel signature methods may also be used). Sequentially accessed files are good candidates for parallel processing of several requests at a time (unlike tree organizations). This may be particularly useful for jukebox optical disk based architectures where disk interchanges to the reading device(s) may be slow and therefore requests queued. New requests may also arrive during processing and it may be desirable to join the queue of requests in progress. We are currently experimenting with several algorithms for processing requests in parallel in such a multimedia environment using signatures of one or more levels. The best of these algorithms will be incorporated into the query processing component of MINOS.

Integrated System Implementation

The system under implementation is based on a server architecture using SUN workstations, Ethernet, and magnetic and optical mass storage devices for the server. The multimedia presentation manager resides in the workstations. A high-performance object filing system that combines magnetic and optical disk technology has been implemented [Christodoulakis et al. 87]. The filing system is general, in that it allows the designer to choose from a variety of access methods and implementations of access methods, data placement strategies and implementations of data placement strategies to be defined for a file. This set of access methods and placement strategies is extensible. We are currently testing the system. The filing system will be used for low-level support of the archival component of the server. The query processing strategies that will perform best in the performance studies outlined in this paper will be incorporated in the system.

References

[Christodoulakis 84] S. Christodoulakis: "Implications of Assumptions in Database Performance Evaluation", ACM TODS, June 1984.

[Christodoulakis 87a] S. Christodoulakis: "Analysis of Retrieval Performance for Records and Objects Using Optical Disk Technology", ACM TODS, June 1987.

[Christodoulakis 87b] S. Christodoulakis: "Analysis and Fundamental Performance Tradeoffs for CLV Optical Disks", Technical Report, Department of Computer Science, University of Waterloo, 1987.

[Christodoulakis and Velissaropoulos 87] S. Christodoulakis and T. Velissaropoulos: "Issues in the Design of a Distributed Testbed for MINOS", Transactions on Management Information Systems", 1987.

[Christodoulakis and Ng 87] S. Christodoulakis and R. Ng: "Query Processing in a Multimedia Retrieval Environment", in preparation, 1987.

[Christodoulakis et al. 87] S. Christodoulakis, E. Ledoux, R. Ng: "An Optical Disk Based Object Filing System", Technical Report, Department of Computer Science, University of Waterloo, 1987.

[Christodoulakis and Faloutsos 84] S. Christodoulakis and C. Faloutsos: "Performance Analysis of a Message File Server", IEEE Transactions on Software Engineering, March 1984.

[Faloutsos and Christodoulakis 87] C. Faloutsos and S. Christodoulakis: "Analysis of Retrieval Performance of Signature Access Methods", ACM TOOIS, 1987.

[Haskin81] L.A. Haskin: "Special Purpose Processors for Text Retrieval", Database Engineering 4,1, Sept 1981,16-29.

Query Processing Based on Complex Object Types

Elisa Bertino, Fausto Rabitti

Istituto di Elaborazione della Informazione
Consiglio Nazionale delle Ricerche
Via S.Maria 46, Pisa (Italy)

ABSTRACT

In application areas where the data management system has to deal with a large number of complex data objects with a wide variety of types, the system must be able to process queries containing both conditions on the schema of the data objects and on the values of the data objects. In this paper we will focus on a particular phase in query processing on a data base of complex objects called Type-Level Query Processing. In this phase, the query is analyzed, completed, and transformed on the basis of the the definitions of the complex object types. We will present, in particular, the techniques used in the ESPRIT project MULTOS. In this project, a data server has been implemented in which data objects are constituted by multimedia documents with complex internal structures.

1. Introduction

Many applications, such as office information systems (OIS), particularly filing and retrieval of multimedia documents [IEEE84], computer-aided design and manufacturing (CAD/CAM), and artificial intelligence (AI) in general and knowledge-based expert systems in particular, need to deal with a large number of data objects having complex structures. In such application areas, the data management system has to cope with the large volumes of data and to manage the complexity of the structures of these data objects [BANE87]. An important characteristic of many of these new applications is that there is a much lower ratio of instances per type than in traditional data base applications. Consequently, a large number of objects implies a large number of object types. The result is often a very large schema, on which it becomes difficult for the users to specify queries. The data management system must be able to process queries containing both conditions on the schema (i.e. partial conditions on type structures of the complex data objects to be selected) and on the data objects (i.e. conditions on the values of the basic components contained in the complex data objects).

In this paper we will focus on a particular phase in query processing on a data base of complex objects. In this phase the query is analyzed, completed, and transformed based on the information contained in the definitions of the complex object types. We call this phase Type-Level Query Processing. With this phase, the system realizes a two-fold functionality:

- The system does not force the user to specify exactly the structures (i.e. the types) of the complex data objects to select. On the contrary, it allows the user to specify only partial structures of these complex objects, so making queries on content is much more flexible. In fact, the user can specify the type of only a few components of the complex objects (and giving conditions on the values), without specifying the complete type of the complex objects.
- The system exploits the complex structures of the data objects, described according to a high-level model, for query transformations which simplify the rest of the query processing.

During Type-Level processing, some transformations allow pruning of the query, so that the resulting query contains fewer predicates to evaluate. In other words, for a given query, the Type-Level processor checks whether there are conjuncts or disjuncts in the query that are always true for instances of the object types referenced in the query. In certain cases, during this phase, it may also be deduced that the query is empty without having to access the data. In this paper, we will describe such transformations and also in which cases the Type-Level processor deduces that a query is empty.

For this purpose, we will present the techniques used for this phase of query processing in Project MULTOS, for the implementation of a data server in which data objects are constituted by multimedia documents with complex internal structures.

2. The MULTOS System

The MULTOS multimedia document server has been designed and implemented within project MULTOS [BERT85] [BERT86], which is part of the European Strategic Programme for Research in Information Technology (ESPRIT).

The internal structure of the document server consists of a certain number of components: the type handler maintains type definitions and manages all the operations on types; the storage subsystem provides access methods for document retrieval and allows the storage of large data values, the operation and structure translator maps document level operations onto the data structures of the storage subsystem. The query processor is responsible for the execution of queries. It checks the syntactic correctness of the query and performs query decomposition and query optimization. The result of query execution is the set of identifiers of all documents satisfying the query. The query processor is also the module which performs Type-Level processing of the queries.

2.1. The Document Model

A multimedia document is a collection of components which contain different types of multimedia information, and may be further structured in terms of other components (such as the body of a paper that is composed of sections and paragraphs and contains images and attributes embedded in text). For these reasons, we can consider multimedia documents as complex data objects. These complex structures, which can vary greatly from one document instance to another, cannot be adequately described with the structuring mechanisms of traditional data models. Thus, an important issue concerns the adoption of a suitable conceptual model. The data model adopted in MULTOS is defined in [MULT86], and is based on the ideas expressed in [RABI85] and [BARB85].

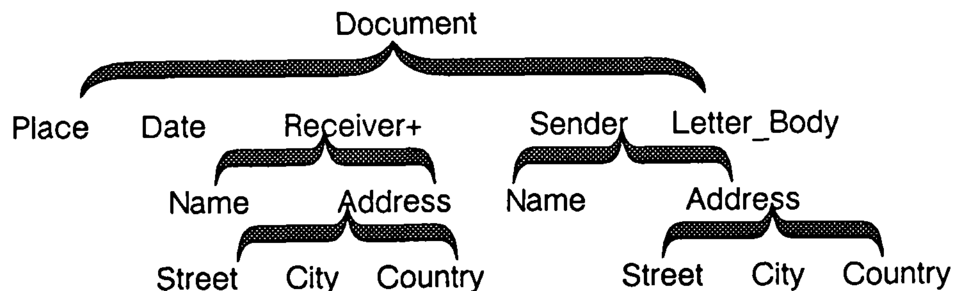


Figure 1: Example of Document Type: Generic_Letter

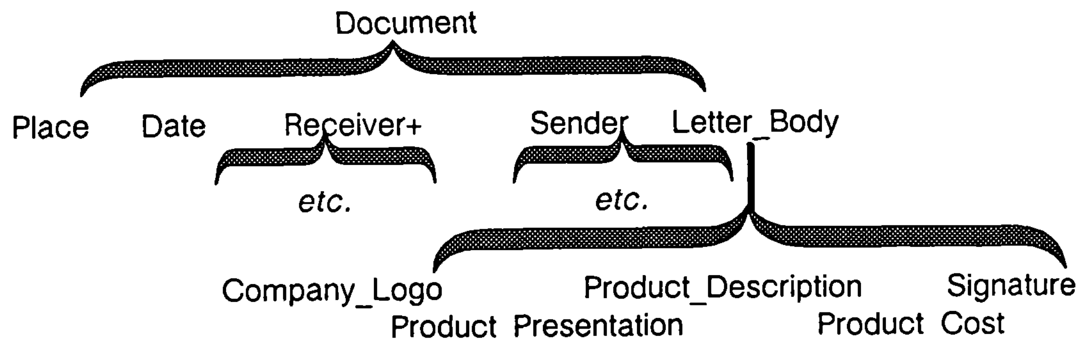


Figure 2: Example of Document Type: Business_Letter

In order to support different operations (i.e., editing, presentation, retrieval), the document model supports several structural descriptions of a multimedia document. The **logical structure** determines how logical components, such as sections and paragraphs, are related [HORA85]. The **layout structure** describes how document components are arranged on an output device at presentation time [HORA85]. There may be couplings between logical and layout structures. The document model adopts a standardized representation based on ODA (the Office Document Architecture, a standard under definition by ISO, ECMA and CCITT [ECMA85]) for the logical and layout structures.

It is also important to see a document in terms of its conceptual components: a conceptual component is a document component which has some commonly understood meaning for the community of users. The **conceptual structure** is used for query specification, since conceptual components are more meaningful to the user than logical or physical components and, also, the conceptual structure is often less complex than the logical and layout structures. The main type constructor used in this data model is **aggregation**, which allows the definition of a complex component as the composition of its sub-components. In this model, the type definition of complex objects (i.e. multimedia documents) can be organized in a **is–a** hierarchy, where the inheritance of (conceptual) components between complex object types is described.

In Fig.1 the conceptual structure of the type Generic_Letter is sketched. In Fig.2 the conceptual structure of the type Business_Letter is sketched. The second type is a specialization of the first type, since the component Letter_Body has been specialized into five new components. In conceptual modelling terms, we can say that Business_Letter is–a Generic_Letter.

In the examples of document types, the "+" symbol attached to the component Receiver means that it is a multi-valued conceptual component. It should be noticed also that the conceptual components Name and Address appear in two subtrees having as roots respectively the conceptual components Receiver and Sender.

2.2. The Query Language

Queries may have conditions on both the content and the conceptual structure of documents. Expressing conditions on the document's conceptual structure means to ask for documents having the conceptual components whose names are specified in the condition. The query language is fully described in [MULT86a]. In general, a query has the following form:

find documents version ... scope ... type *TYPE-clause*; where *COND-clause*;

One or more conceptual types can be specified in the *TYPE* clause. The conditions expressed in the query apply to the documents belonging to the specified types. If the types indicated in the query have subtypes, then the query applies to all the documents having as type one of these subtypes. When no type is specified, the query will apply to all possible document types. The conditions expressed in the *COND* clause are a Boolean combination of conditions which must be satisfied by the documents retrieved. Conditions on text components and conditions on attribute components, of different types, can be mixed in the *COND* clause. A text condition on the special component named "text" is applied to the entire document.

In order to reference a conceptual component in a document, a path-name must be specified. A path-name has the form:

$$name_1[.|*]name_2[.|*]\dots name_{n-1}[.|*]name_n$$

where each $name_i$ is a simple name.

The path-name specifies that the conceptual component being referenced is the component having simple name $name_n$ which is contained within the conceptual component whose name is $name_{n-1}$. The conceptual component $name_{n-1}$ is in turn contained in $name_{n-2}$, and so forth. Component names within a path-name can be separated by either a "." or a "*". When a "." is used, it means that the conceptual component of the left side of the "." contains directly the component on the right. When a "*" is used, there may be one or more intermediate components between the component on the left side and the one on the right.

In our query language, conditions are usually expressed against conceptual components of documents, that is, a condition has the form: "*component restriction*", where *component* is the name (or path-name) of a conceptual component and *restriction* is an operator followed by an expression. This expression may contain other component names.

It should be noticed that any component name (or path-name) may refer to a conceptual component which is contained in several conceptual components. For instance, in the example in Fig.2, we could have a

condition of the form: "Name *restriction*". The restriction applies to both components whose path-names are Sender.Name and Receiver.Name. The problem is to decide how such a condition is satisfied. There are four possibile interpretations:

(1) Name *restriction* = True if
(Sender.Name *restriction* = True) $\wedge$ ($\exists$ (Receiver.Name *restriction* = True))

(2) Name *restriction* = True if
(Sender.Name *restriction* = True) $\wedge$ ($\forall$ (Receiver.Name *restriction* = True))

(3) Name *restriction* = True if
(Sender.Name *restriction* = True) $\vee$ ($\forall$ (Receiver.Name *restriction* = True))

(4) Name *restriction* = True if
(Sender.Name *restriction* = True) $\vee$ ($\exists$ (Receiver.Name *restriction* = True))

Our system uses the third interpretation, since it is the most general: the answer to query (3) contains the answers to queries (1), (2), and (3). This choice reflects the approach of giving to the user the most general answers, when there are ambiguities in the query. Then, the user, who did not know exactly the types defined in the document base, can refine the the original query specifying exactly the meaning of the query. The four different semantics, in our query language, can be specified explicitly as:
(1) Sender.Name *restriction* and some Receiver.Name *restriction*
(2) Sender.Name *restriction* and every Receiver.Name *restriction*
(3) Sender.Name *restriction* or every Receiver.Name *restriction*
(4) Sender.Name *restriction* or some Receiver.Name *restriction*

In addition to the previous types of conditions, the language must allow conditions on the existence of conceptual components within documents. This allows expressing queries on the conceptual structure of documents. Therefore we have defined the operator "with". A condition containing the "with" operator has the form: "with *component*". This condition expresses the fact that the component whose name (or pathname) is given must be a conceptual component of the documents to be retrieved. To express conditions that require that a conceptual component having name $name_i$ is contained in a conceptual component having name $name_j$, the path-name $name_i * name_j$ is used.

The "with" operator is conceptually very important in our query language. While the other operators allow the definition of conditions on data (ie. document instance), the "with" operator allows the definition of conditions on meta-data (ie. document types).

An example query that will be used throughout this paper is:

```
    find documents where Document.Date > /1/1/1987/ and
(*Sender.Name = "Olivetti" or *Product_Presentation contains "Olivetti") and
*Product_Description contains "Personal Computer%" and
(*Address.Country = "Italy" or text contains "Italy") and
with *Company_Logo;
```

It should be noticed that no type is specified for this query. As we will see, one of the tasks associated with Type-Level Processing is to determine the type(s), if any, to which the query applies.

3. Initial Steps in Query Processing

The task of query processing consits of several steps, some of which are concerned with query optimization [BERT87]. In this paper we consider the pre-processing steps, in which some initial activities are performed, such as query parsing and accessing the type catalog. Also during this phase, the query is modified in light of the type hierarchy.

3.1. Parsing

The query is parsed by a conventional parser. The parser verifies that the query has a correct syntax. The parser output is a query parse tree, which is augmented and modified by the subsequent steps in query processing. The *COND* clause (the boolean combination of conditions) is expressed in the parse tree in Conjunctive Normal Form (CNF).

3.2. Type-Level Processing

If a list of types is specified in the query (clause *TYPE*), then it is checked that the conceptual components present in the query belong to those types. Also, each conceptual component name is expanded to its complete path name. If there are several paths corresponding to a given name, the condition C in which the component appears is substituted by a disjunction of conditions $C_1, \ldots, C_n$, where n is the number of path names. Each C_i has the same form as C, except that the name of the conceptual component appearing in C is substituted by the i-th path name.

If no type is specified, the type catalog is accessed to determine the document types containing the conceptual components whose names appear in the query conditions (clause *COND*). The list of these types is added to the query tree. If no document type exists containing such conceptual components, the query results in an empty set and query processing stops.

The transformations on the query parse tree include the elimination of all "with" conditions (in most cases), the reorganization (i.e. addition and/or deletion) of disjuncts in some conjuncts and the possible elimination of some conjuncts. If the conjuncts have mutually exclusive requirements (i.e. no document type exists containing the required conceptual components) the query results in an empty set and the query processing stops. If all conjuncts are eliminated the answer to the query is the set of documents belonging to one of the types determined in the Type-Level query processing, and no query optimization is necessary any more. The algorithms used in this query processing phase will be described in the following section.

The example query, resulting from the transformations performed by the Type-Level query processor, is the following (this query applies only to the type Business_Letter):

```
    find documents type Business_Letter; where Document.Date > /1/1/1987/ and
(Document.Sender.Name = "Olivetti" or
Document.Letter_Body.Product_Presentation contains "Olivetti") and
Document.Letter_Body.Product_Description contains "Personal Computer%" and
(some Document.Receiver.Address.Country = "Italy" or
Document.Sender.Address.Country = "Italy" or text contains "Italy");
```

4. The Algorithm for Type-Level Query Processing

The input to the algorithm is the initial list of types L contained in the clause *TYPE* and the query parse tree, derived from the *COND* clause and expressed in CNF:

$$COND = \bigwedge_i (\bigvee_j (r_{i,j}))$$

where $r_{i,j}$ is the j-th condition in disjunction in the i-th conjunct.

In order to illustrate the algorithm for Type-Level Query Processing, we show how each step applies to the query of the previous example. We suppose that the types in the catalog are:
t_1 = Generic_Letter, t_2 = Business_Letter.

The algorithm is composed of the following steps:

1) All path-names, $P = \{p_1, p_2, \ldots, p_p\}$, identifying document components on which query conditions are defined, are extracted from the parse tree.

   In the example, we have $P = \{p_1, p_2, \ldots, p_6\}$ where:
   p_1 : Document.Date
   p_2 : *Sender.Name
   p_3 : *Product_Presentation
   p_4 : *Product_Description
   p_5 : *Address.Country
   p_6 : *Company_Logo

2) A p_i can either be a complete path-name (i.e. without "*") or a partially specified path-name (i.e. with "*"). For each p_i, we determine the set $N_i = \{n_{i,1}, n_{i,2}, \ldots, n_{i,n_i}\}$ of all complete path-names corresponding to p_i according to the applicable type definitions in the document type catalog (if p_i is already a complete path-name, N_i contains only p_i itself, provided that the component name is found in at least one type definition). N_i can be empty. Then, for each $n_{i,j}$, the list of document types

$T_{i,j} = \{t_1, t_2, \ldots, t_{t_{i,j}}\}$ containing this component name is determined by accessing the type catalog (including sub-types found in the type hierarchy).

In the example, for each path-name p_i in set P determined in the previous step, the set N_i is:
N_1 = {Document.Date}
N_2 = {Document.Sender.Name}
N_3 = {Document.Letter_Body.Product_Presentation}
N_4 = {Document.Letter_Body.Product_Description}
N_5 = {Document.Receiver.Address.Country, Document.Sender.Address.Country}
N_6 = {Document.Letter_Body.Company_Logo}

Also, the sets $T_{i,j}$ are:
$T_{1,1} = \{t_1, t_2\}$, $T_{2,1} = \{t_1, t_2\}$, $T_{3,1} = \{t_2\}$, $T_{4,1} = \{t_2\}$, $T_{5,1} = \{t_1, t_2\}$, $T_{5,2} = \{t_1, t_2\}$, $T_{6,1} = \{t_2\}$

3) The set of all types referenced in the query conditions is determined by $T' = \bigcup_i(\bigcup_j(T_{i,j}))$

   In the example, $T' = \{t_1, t_2\}$.

4) If the *TYPE* clause exists, the types listed in it are assigned to the set L. Let $T'' = L^*$ be the transitive closure, according to the type is–a hierarchy of all the types listed in L. Otherwise T'' is the set of all types in the catalog.

   In the example, since there is no *TYPE* clause in the query, T'' is the set of all types in the catalog: $\{t_1, t_2\}$.

5) The initial set of types for the query is determined by $T = T' \bigcap T''$.

   In the example, $T = \{t_1, t_2\}$.

6) Now the query is expanded. Let $p_{i,j}$ be the path-names of the condition $r_{i,j}$, as specified in the query. We discuss only the case in which each condition contains only one path-name. The extension to the case in which more path-name are contained in the same condition is straightforward.

   If $N_{i,j}$ is empty, disjunct $r_{i,j}$ must be eliminated. Otherwise, if $N_{i,j}$ contains k complete path-names $n_1, \ldots, n_k$, the disjunct $r_{i,j}$ is replaced by k new disjuncts (inside the same i–th conjunct), where $p_{i,j}$ has been replaced by $n_1, \ldots, n_k$.

   In the example, the following disjuncts are modified by this step:
   $r_{2,1}$ (name expansion): Document.Sender.Name = "Olivetti"
   $r_{2,2}$ (name expansion): Document.Letter_Body.Product_Presentation contains "Olivetti"
   $r_{3,1}$ (name expansion): Document.Letter_Body.Product_Description contains "Personal Computer%"
   $r_{4,1}$ is replaced by two new disjuncts, one for each complete path-name contained in set N_5. They are as follows:
   $r_{4,1'}$: Document.Receiver.Address.Country = "Italy"
   $r_{4,1''}$: Document.Sender.Address.Country = "Italy"
   $r_{5,1}$ (name expansion): with Document.Letter_Body.Company_Logo.

7) For each conjunct i, the set of types which contains at least one component referenced by a "non-with" condition of this conjunct is determined by $T_i = \bigcup_j(X_{i,j})$ where $X_{i,j}$ is the list of types associated with the name $n_{i,j}$ of the condition $r_{i,j}$, provided that this is not a "with" condition.

   In the example, for each conjunct i ($i = 1, \ldots, 5$), the set T_i is determined:
   $T_1 = \{t_1, t_2\}$, $T_2 = \{t_1, t_2\}$, $T_3 = \{t_2\}$, $T_4 = \{t_1, t_2\}$, $T_5 = \emptyset$.

8) For each conjunct i, the set of types which contains at least a component referenced by a "with" condition of this conjunct is determined by $W_i = \bigcup_j(Y_{i,j})$, where $Y_{i,j}$ is the list of types associated with the name $n_{i,j}$ of the condition $r_{i,j}$, provided that this is a "with" condition. Let the set S_i be the list of "with" conditions $r_{i,j}$ whose $Y_{i,j}$ constitute the minimal covering of the set W_i.

In the example, for each conjunct i (i = 1,. . .,6), the set W_i is:
$W_1 = \emptyset$, $W_2 = \emptyset$, $W_3 = \emptyset$, $W_4 = \emptyset$, $W_5 = \{t_2\}$
For conjunct 5 the set S is: $S_5 = \{r_{5,1}\}$.

9) Let $W = \bigcap_i (W_i)$.

In the example, we obtain the following set W: $W = \emptyset$.

10) The set of types which contain at least one component in each conjunct is determined by:
$T = T \bigcap (\bigcap_i (T_i \bigcup W_i))$. If T is empty, the query is empty and query processing stops.

In the example, we obtain:
$T = \{t_1, t_2\} \bigcap ((\bigcap(\{t_1, t_2\}, \{t_1, t_2\}, \{t_2\}, \{t_1, t_2\}, \{t_2\})) = \{t_2\}$.
Therefore, the set of types to which the query applies contains only the type t_2 = Business_Letter.

11) Now query reduction starts. For each conjunct i, each disjunct $r_{i,j}$ is examined. If the associated set of types $T_{i,j}$ is not contained in T, condition $r_{i,j}$ cannot become True and must be eliminated. If the whole conjunct become empty, the query becomes empty and query processing stops.

In the example, this step has no effect, since there are no disjuncts that satisfy the condition expressed in the algorithm.

12) The set of types T_i and W_i are now restricted: $\forall i, \quad T_i = T_i \bigcap T \quad W_i = W_i \bigcap T$

In the example, the sets T_i and W_i are restricted by this step to be:
$T_i = \{ t_2 \} \quad for\ i = 1, \ldots, 4$
$T_5 = \emptyset$
$W_i = \emptyset \quad for\ i = 1, \ldots, 4$
$W_5 = \{ t_2 \}$

13) For each conjunct i, each disjunct $r_{i,j}$ which is a "with" condition and which is not a member S_i is eliminated.

In the example, this step has no effect since there are no disjuncts that satisfy this condition.

14) For each conjunct i, each disjunct $r_{i,j}$ which is not a "with" condition and whose $T_{i,j}$ is contained in W_i is eliminated.

In the example, this step has no effect since there are no disjuncts that satisfy this condition.

15) If conjunct i contains only "with" conditions and W_i is contained in W, then it is always True, since it is implied by the other conjuncts. Thus, this conjunct can be eliminated.

In the example, this step has no effect since there is only one conjunct which is constituted by "with" conditions, conjunct 5, but W_5 is not contained in W.

16) If conjunct i contains only "with" conditions and W_i contains T, then it always True, since it is implied by the list of types T. Thus, this conjunct can be eliminated.

In the example, conjunct 5 is eliminated, since it contains only a "with" condition and W_5 contains T.

The output of the algorithm is the list of types T on which the query is to be restricted and the modified query parse tree: $\bigwedge_i (\bigvee_j (r_{i,j}))$. If the query parse tree is empty (i.e. all conjuncts have been eliminated), the answer to the query is the set of documents belonging to one of the types in T, and no query optimization is necessary.

In the example, the output of the algorithm yields the set of types T = {Business_Letter} and the modified *COND* clause:
$r_{1,1}$: Document.Date > /1/1/1987/
$r_{2,1}$: Document.Sender.Name = "Olivetti"
$r_{2,2}$: Document.Letter_Body.Product_Presentation contains "Olivetti"

$r_{3,1}$: Document.Letter_Body.Product_Description contains "Personal Computer%"
$r_{4,1'}$: Document.Receiver.Address.Country = "Italy"
$r_{4,1''}$: Document.Sender.Address.Country = "Italy"
$r_{4,2}$: text contains "Italy".

5. The MULTOS Prototype

The query processing techniques (including Type-Level query processing) described above have been implemented in project MULTOS. The design and implementation of the MULTOS prototype will require a total effort of about 100 man-years, distributed over five years. The implementation of the first prototype of the MULTOS multimedia document server was completed as of June 1987. In this prototype, query processing is based on data and text components of documents. A second prototype will follow, in which image and audio components will also be considered. In particular, graphics and image components should play an active role in the query processing.

References

[BANE87] Banerjee, J., et al., *Data Model Issues for Object-Oriented Applications*, ACM Trans. on Office Information Systems, Vol.5 N.1, Jan. 1987.

[BARB85] Barbic F. and Rabitti F., *The Type Concept in Office Document Retrieval*, Proc. VLDB Conference, Stockholm, August 21-23, 1985.

[BERT85] Bertino, E., Gibbs, S., Rabitti, F., Thanos, C., Tsichritzis, D., *Architecture of a Multimedia Document Server*, Proc. 2nd ESPRIT Technical Week, Brussels, Sept. 1985.

[BERT86] Bertino E., Gibbs S., Rabitti F., Thanos C., and Tsichritzis D., *A Multimedia Document Server*, Proc. Advanced Database Symposium, Japan, August 29-30, 1986.

[BERT87] Bertino E., Rabitti F., Gibbs S., *Query Processing in a Multimedia Document System*, IEI-CNR Technical Report B4-18, Pisa, Italy, 1987.

[ECMA85] ECMA TC-29, *Office Document Architecture*, Standard ECMA-101, European Computer Manifacturer Association, Paris, Sept. 1985.

[HORA85] Horak W., *Office Document Architecture and Office Document Interchange Formats: Current Status of International Standardization*, Computer, Vol.18, N.10, pp.50-62, October 1985.

[IEEE84] IEEE Database Engineering, Special Issue on Multimedia Data Management, Vol.7, N.3, September 1984.

[MULT86] Rabitti F., *Document Model*, MULTOS Tech. Deliverable IEI-86-02, IEI-CNR, Pisa, March 1986.

[MULT86a] Bertino E., Rabitti F., *Definition of the Query Language*, MULTOS Tech. Deliverable IEI-86-03.1, IEI-CNR, Pisa, March 1986.

[RABI85] Rabitti F., *A Model for Multimedia Documents*, in *Office Automation: Concepts and Tools*, D.Tsichritzis ed., Springer-Verlag, 1985.

Extensible Cost Models and Query Optimization in GENESIS *

D.S. Batory
Department of Computer Sciences
The University of Texas at Austin

Abstract

The GENESIS extensible DBMS is founded on the premise that customized DBMSs can be synthesized from primitive and prewritten modules. In this article, we explain how cost models used in query optimization can be synthesized from primitive cost functions.

1. Introduction

GENESIS is an extensible database system. Its distinguishing feature is that it is a realization of a theory of DBMS implementation [Bat82-87a] whose principles were reviewed in an earlier DBE article [Bat87b]. The underlying premise of the theory is that DBMSs can be synthesized from primitive, prewritten modules that have standardized interfaces. If all components are available, GENESIS can synthesize a target DBMS in minutes, in contrast to the man-years of effort required to customize existing DBMSs.

Query optimization is among the most important features of DBMSs. One of the fundamental problems in extensible DBMS technologies is the development of extensible query optimizers [Fre87, Gra87]. Query processing algorithms and their cost functions are at the heart of query optimization; a DBMS uses the cost functions to identify the cheapest available algorithm to process a query. In GENESIS, complex algorithms are compositions of primitive algorithms. By symmetry, complex cost functions can be seen as compositions of primitive cost functions.

In this article, we briefly outline the steps that are taken to process queries in GENESIS. We review the ideas of how DBMS software can be constructed from components, and show how these same ideas can be applied to the construction of cost models that are required by query optimizers.

Keep in mind that the GENESIS approach is not primarily aimed at the development of new query processing algorithms, but rather at the development of a *framework* in which recognized and new algorithms can be cast. (This point should be evident in the following sections). The framework that we outline provides the means for standardizing interfaces, defining plug-compatible modules, and demonstrating a building-blocks technology for extensible DBMSs. We show in [Bat87a] how many different query processing algorithms (e.g., join algorithms) conform to a very simple algebraic framework.

2. Overview

Nontraditional database applications require new data types and operators. A functional data model and data language, we feel, provides the greatest flexibility for addressing these needs. GENESIS users see databases consisting of objects that are related by functions [Bat86b]. User requests, such as queries and updates, are functional expressions, called **GDL-expressions**, that map streams of objects.

In contrast, the algorithms and storage structures of DBMS implementations are traditionally expressed in terms of record/tuples, rather than objects. A record-based model with a functional data language, we feel, provides the best description of DBMS implementations [Bat87a]. GENESIS DBMS implementors (DBIs) see databases as networks of files and links. Database algorithms, such as file retrieval and joins (i.e, link traversals), are defined by functional expressions called record expressions or **r-expressions** that map streams of records. The precise syntax and semantics of GDL-expressions and r-expressions are not essential to this article. We note, however, that the distinction between GDL-expressions and r-expressions is similar to the

* This work was supported by the National Science Foundation under grant DCR-86-00738.

difference between SQL statements and their relational algebra counterparts.

Query processing and optimization in GENESIS involves a translation between these representations and is accomplished in three steps: 1) mapping a user's GDL-expression to an equivalent, but not optimized, r-expression, 2) optimizing the r-expression, and 3) evaluating the optimized expression. Figure 1 shows these steps and their intermediate results.

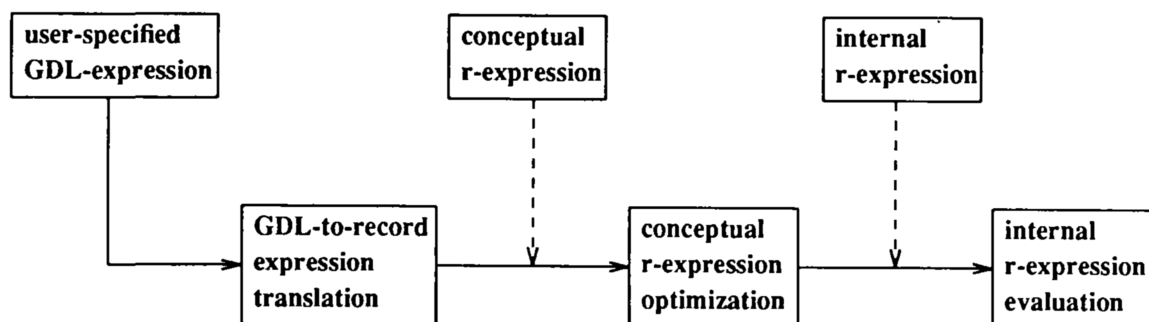


Figure 1. Query Optimization in GENESIS

Object-to-record data mappings are needed in Step 1. These mappings are provided by database administrators (DBAs) when an object-based schema is initially declared. (Database design tools could be invented to automate such mappings). GENESIS provides a good deal of flexibility here, as it supports non-1NF records, i.e., records with repeating fields and nested repeating fields [Bat86a].

Mapping GDL-expressions to r-expressions does not involve query optimization. The r-expression that is produced in Step 1 consists solely of operations on conceptual files, conceptual links, and record streams. *No reference is made to the algorithms that implement these operations* (e.g., which join algorithm to use, which indices to use, etc.).

Optimizing an r-expression and selecting the algorithms to implement the conceptual operations is the task of Step 2. This can be accomplished by rule-based optimizers or by traditional query optimizers. (GENESIS has no fixed optimizer; the optimizer that is used in a GENESIS-produced DBMS is part of the DBMS's specification).

Query optimization in GENESIS clearly requires an extensible cost model which estimates the cost of executing conceptual operations. In the following sections, we outline a framework in which software for DBMSs can be synthesized from components, and how extensible cost models can be produced in a similar manner.

3. DBMS Software and Cost Model Building Blocks

The underlying concepts of GENESIS can be explained in terms of parameterized types [Gog84]. A classical example of a parameterized type is STACK_OF[x]; stack algorithms can be defined independently of the objects that are placed on a stack. Nonparameterized types, such as INT and STRING, can be used as parameters of STACK_OF to define stacks of integers (STACK_OF[INT]) and stack of strings (STACK_OF[STRING]). The modules that implement the STACK_OF, INT, and STRING types, can be recognized as building blocks of more complicated systems.

DBMS implementations can be viewed in the same way. One can define the nonparameterized types BPLUS, ISAM, and HEAP. Instances of these types are B+ tree, indexed-sequential, and heap structures. Declaring a file F to be of type BPLUS, for example, means that file F has a B+ tree implementation.

Parameterized types arise in all but the most trivial of DBMSs. INDEX[df,xf] is a parameterized type that defines a partial implementation of an inverted file. When a file F is defined to be of type INDEX, the INDEX module maps F (henceforth called an **abstract file**) to a data file and a set of zero or more index files (henceforth called **concrete files**). INDEX also maps operations on F to data file and index file operations. The key idea behind the parameterization is that the data and operation mappings of INDEX *do not rely* on the

implementations of the concrete data file and concrete index files. For this reason, the file types of the data and index files are parameters to INDEX. Specifically, the df parameter specifies the implementation of the data file and xf the implementation of the index files. A composition of types is a **module expression**. The module expression of a typical inverted file is INDEX[HEAP,BPLUS]: a HEAP structure implements the data file and BPLUS trees implement the index files of an inverted file. Of course, assigning different file structure implementations to parameters df and xf yield different inverted file implementations. A large list of parameterized types (or **elementary transformations**) is given in [Bat85].

A consequence of describing DBMS implementations as module expressions is a rather unusual notion of software layering. Normally, different layers in a system have completely different interfaces. In the GENESIS approach, each parameterized and nonparameterized file type corresponds to a distinct layer. Layers are stacked in the order in which their corresponding types are nested in a DBMS's module expression. Because each type defines a file implementation, all types/modules/layers support *exactly* the same interface, and are thus plug-compatible.

When modules (types) are composed, the notions of abstract and concrete are relative. The most abstract files are conceptual files, and the most concrete files are internal files. If file C is of type INDEX[HEAP,BPLUS], it is a conceptual file. The heap data file and the B+ tree index files that are generated by this module expression are the internal files.

Every GENESIS module or layer maps operations on an abstract file to operations on concrete files. To illustrate operation mappings, let's first consider two simple file types: BPLUS and HEAP. Let AF be an abstract file and Q be a selection predicate. The abstract operation RET(AF,Q) generates the stream of AF records that satisfy Q. The implementations of RET in the BPLUS and HEAP modules are:

```
RET(AF,Q)                          /* taken from the BPLUS module */
{
     RET_BPLUS(AF,Q);              /* B+ tree retrieval algorithm */
};

RET(AF,Q)                          /* taken from the HEAP module */
{
     RET_HEAP(AF,Q);               /* heap retrieval algorithm */
};
```

That is, if file AF is a BPLUS tree, the RET operation on AF is realized by a B+ tree retrieval algorithm (i.e., RET_BPLUS). If AF is a HEAP, the RET operation on AF is realized by a heap retrieval algorithm (RET_HEAP). Similar discussions hold for other file structures (e.g., ISAM, RTREE, etc.).

Another retrieval operation which accesses an abstract record given its pointer p, denoted ACC(AF,p), would be implemented in a similar manner. (ACC_BPLUS would be the access algorithm for B+ trees, ACC_HEAP for heaps, etc.). So too would other abstract operations, such as inserting, deleting, and updating abstract records.

From an extensibility viewpoint, if a new file structure is invented, a new file type can be defined. The module that implements the type supports the same interface as all other file modules and encapsulates the retrieval, access, insertion, deletion, etc. algorithms that are associated with the file structure. Adding new file structures in this way is simple, and is called **storage structure extensibility**. [1]

Query optimization requires cost estimates of file structure retrievals. An obvious organization to impose on an extensible optimizer is to express costs of abstract operations in terms of costs of concrete operations. Let $RET(AF,Q) denote the cost of performing the RET(AF,Q) operation. For the BPLUS and HEAP types,

[1] A slightly different, but equivalent, description of storage structure extensibility was given in [Bat87b], where BPLUS, HEAP, ISAM, and other storage structure algorithms were encapsulated in a single module (INTERNAL), where different algorithms were distinguished within a large case statement. Our use of the parameterized type analogy eliminates the need for this case statement.

we can supply definitions for $RET:

```
$RET(AF,Q)                              /* taken from BPLUS module */
{
        $RET_BPLUS(AF,Q);               /* B+ tree retrieval cost function */
};

$RET(AF,Q)                              /* taken from HEAP module */
{
        $RET_HEAP(AF,Q);                /* heap retrieval cost function */
};
```

As before, when a $RET(AF,Q) is to be executed, the implementation of the file AF determines the cost function to be evaluated. Thus, if AF is a heap, then $RET_HEAP(AF,Q) is executed.

In general, every GENESIS file type supports a common set of operations and cost functions. The operations are basic operations (e.g., record retrieval, insertion, deletion, etc.) on abstract files. The algorithm(s) for each abstract operation are expressed in terms of operations on concrete files. By analogy, there is one cost function for each abstract operation. The expression that defines such a cost function is expressed in terms of cost functions for concrete file operations.

Now let's look at a more complicated file type: INDEX. INDEX maps an abstract file and its operations to an inverted file, which consists of a data file and $n \geq 0$ index files, each of which is connected to the data file by a link (Fig. 2). Let AF be the abstract file, D be the (concrete) data file, and I_j be an index file (j=1..n). A possible implementation of RET(AF,Q) in INDEX is:

```
RET(AF,Q)                                         /* taken from INDEX module */
{       case (use_$RET(AF,Q)_to_choose_cheapest) of
        {       Alg1:   RET(D,Q);                                    /* scan data file */
                Alg2:   USE_1_INDEX(AF, Q, RET(Ij, Qj), ACC(F,p));   /* use one index file */
                Alg3:   USE_n_INDICES(AF, Q, RET(Ij, Qj), ACC(F,p)); /* use many indices */
        };
};
```

The three retrieval algorithms listed above are among the most popular in inverted file implementations. The first processes query Q by scanning the data file, the second uses one index (as is done in System R), and the third is the classical algorithm that takes the intersection and union of multiple inverted lists.

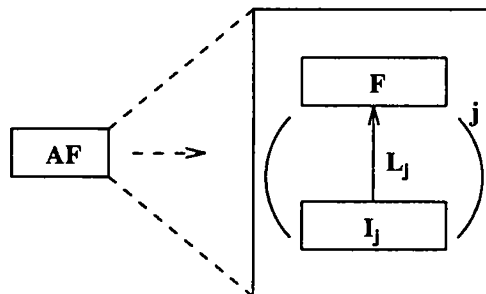


Figure 2. Indexing Layer Data Mappings

Three comments. First, note that the USE_1_INDEX and USE_n_INDICES algorithms have the operations RET(I_j, Q_j) and ACC(F,p) as parameters. These operations represent, respectively, index file retrievals and data file accesses. Note that the algorithms that implement these operations are *not* specified; only when the implementation of these files is given in a module expression are algorithms actually assigned to these

operations. As we will see, this is critical to a plug-compatible software technology.

Second, there are other algorithms besides the three listed above. To add a new inverted file retrieval algorithm, one simply adds another case. The same applies for other abstract operations (e.g., abstract record insertion, deletion, etc.). They too are implemented as case statements, and new algorithms are included as additional cases. Introducing another algorithm to process an abstract file or abstract link operation is called **algorithm extensibility.**

Third, an element of query optimization can be seen in this example. When a RET(AF,Q) is to be processed, the cheapest available algorithm is selected. This is accomplished by evaluating cost functions for each algorithm and choosing the cheapest algorithm. The code template for making this optimization decision has the following organization:

```
$RET(AF,Q)                                          /* taken from INDEX module */
return(    minimum_of
           (    $RET(F,Q),                                      /* Alg1 cost function */
                $USE_1_INDEX(AF, Q, $RET(Ij, Qj), $ACC(F,p)),   /* Alg2 cost function */
                $USE_n_INDICES(AF, Q, $RET(Ij, Qj), $ACC(F,p))  /* Alg3 cost function */
           )
      );
```

Two points. First, the cost functions $USE_1_INDEX and $USE_n_INDICES have algorithms to determine *which* indices to use to process Q in order to achieve the optimal performance. (As there are many such algorithms, the algorithm that is actually used could be a parameter to the $RET(AF,Q) cost function. Doing so would provide a simple means by which optimizing algorithms could be changed). The information about what indices to use is retained as part of the optimization process, and is later used when the selected algorithms are executed.

Second, note that actual expressions for the cost functions for index file retrieval $\$RET(I_j, Q_j)$ and data file accessing $ACC(F,p) are *not* specified; only when the implementation of these files is given in a module expression can actual cost expressions be assigned to these functions.

In summary, the building blocks of DBMSs are parameterized and nonparameterized file types. The software building blocks are the algorithms that map operations on abstract files to operations on concrete files. The cost model building blocks are cost functions that are associated with these operation mappings. Composition of building blocks is considered in the next section.

4. Synthetic DBMSs and Synthetic Performance Models

As mentioned in the previous section, a simple inverted DBMS, similar to INGRES, is described by the module expression INDEX[HEAP,BPLUS]. In this particular example, data file operations referenced in the INDEX algorithms are mapped to heap file operations, and index file operations are mapped to B+ tree operations. Let C be a conceptual file, D be its corresponding data file, I_j be an index file, and Q be a selection predicate. The algorithms to retrieve conceptual records are a composition of INDEX, BPLUS, and HEAP retrieval and access algorithms:

```
RET(C,Q)
{    case (use_$RET(C,Q)_to_choose_cheapest) of
     {    Alg1:     RET_HEAP(D,Q);
          Alg2:     USE_1_INDEX(C, Q, RET_BPLUS(Ij, Qj), ACC_HEAP(D,p));
          Alg3:     USE_n_INDICES(C, Q, RET_BPLUS(Ij, Qj), ACC_HEAP(D,p));
     };
};
```

The above algorithms were derived by replacing RET(D,Q) with RET_HEAP(D,Q), $RET(I_j,Q_j)$ with $RET_BPLUS(I_j, Q_j)$, and ACC(D,p) with ACC_HEAP(D,p).

A cost model for the inverted DBMS can be constructed in an identical manner by composing inverted file and internal cost functions. We obtain by function substitution:

```
$RET(C,Q)
return(    minimum_of
           (    $RET_HEAP(D,Q),                                        /* Alg1 */
                $USE_1_INDEX(C, Q, $RET_BPLUS(Ij, Qj), $ACC_HEAP(D,p))    /* Alg2 */
                $USE_n_INDICES(C, Q, $RET_BPLUS(Ij, Qj), $ACC_HEAP(D,p))  /* Alg3 */
           )
      );
```

In principle, the software and cost models of a DBMS that has a more complicated module expression (and thus has more complicated storage structures) can be synthesized in an identical manner by composing prewritten software/cost components. A model of DBMS software composition and examples of more elaborate DBMSs are given in [Bat87a-b].

Another way to understand synthetic query optimizers is in terms of a hierarchy of optimizers. The top-most optimizer is identified with the conceptual level. Given a query, it generates access plans (i.e., r-expressions) that reference *only* operations on conceptual files and conceptual links. This optimization, as we stated earlier, could be accomplished by rule-based optimizers or by traditional heuristic or enumerative means. (The actual algorithm to search the solution space is a parameter to the DBMS's specification). The cost of an access plan is computed by estimating the frequency with which conceptual operations are executed, times the cost of their execution. The latter is estimated by the cost functions that are generated in the composition process, such as $RET(C,Q) in our above example.

Query optimization in synthetic DBMSs is functionally no different than in traditional DBMSs: first, the solution space of possible access paths for processing the query is searched, the cheapest plan is selected (and in the process, information about what indices to use, etc, is retained), and finally an expression (i.e., composition of algorithms) is generated to implement the plan.

Acknowledgements. I thank Jim Barnett, Brian Twichell, and Tim Wise for their comments and suggestions on earlier drafts of this article.

References

[Bat82] D.S. Batory and C.C. Gotlieb, 'A Unifying Model of Physical Databases', **ACM Trans. Database Syst.** 7,4 (Dec. 1982), 509-539.

[Bat84] D.S. Batory, 'Conceptual-To-Internal Mappings in Commercial Database Systems', **ACM PODS 1984,** 70-78.

[Bat85] D.S. Batory, 'Modeling the Storage Architectures of Commercial Database Systems', **ACM Trans. Database Syst.** 10,4 (Dec. 1985), 463-528.

[Bat86a] D.S. Batory, J.R. Barnett, J.F. Garza, K.P. Smith, K.Tsukuda, B.C. Twichell, T.E. Wise, 'GENESIS: An Extensible Database Management System', to appear in **IEEE Trans. Software Engineering.**

[Bat86b] D.S. Batory and T.Y. Leung, 'Implementation Concepts for an Extensible Data Model and Data Language', TR-86-24, University of Texas at Austin, 1986.

[Bat87a] D.S. Batory, 'A Molecular Database System Technology', TR-87-23, Dept. Computer Sciences, University of Texas, Austin, 1987.

[Bat87b] D.S. Batory, 'Principles of Database Management System Extensibility', **IEEE Database Engineering,** 10, 2 (June 1987), 40-46.

[Fre87] J.C. Freytag, 'A Rule-Based View of Query Optimization', **ACM SIGMOD 1987,** 173-180.

[Gog84] J. Goguen, 'Parameterized Programming', **Trans. Software Engr.,** SE-10, 5 (September 1984), 528-543.

[Gra87] G. Graefe and D.J. DeWitt, 'The EXODUS Optimizer Generator', **ACM SIGMOD 1987,** 160-172.

[Gut77] J. Guttag, 'Abstract Data Types and the Development of Data Structures', **Comm. ACM,** 20,6 (June 1977), 396-404.

Software Modularization
with the EXODUS Optimizer Generator

Goetz Graefe

Department of Computer Sciences and Engineering
Oregon Graduate Center[1]

Abstract

In this paper, we describe the outline of the rules and procedures that need to be written to create a query optimizer in the extensible database system EXODUS. The emphasis is put on the support the EXODUS optimizer generator architecture provides for software modularity and incremental evolution of the data model and the query optimizer.

1. Introduction

In [Graefe1987], we presented the design and an initial performance evaluation of the EXODUS optimizer generator. EXODUS is an extensible database system currently under design and implementation at the University of Wisconsin. An overview of its architecture is given in [Carey1986]. EXODUS is not a database system in itself. It is a collection of software tools and libraries that allows for rapid and structured development of database systems. To implement a new database system, the database implementor, henceforth called the DBI, invokes EXODUS software tools with appropriate description files, and links the generated code with EXODUS libraries and some of her or his own code into an executable database system, ready to assist the users. Besides the optimizer generator, there is the implementation language E [Richardson1987], the storage system [Carey1986a], the type and dependency manager, the access plan translator to generate E programs for queries, and the user interface generator.

2. Rule-Based Optimization of Operator Trees

The design goal of the EXODUS optimizer component is twofold. First, it is imperative to provide a very general and powerful model of optimization. Second, it should be easy to specify the optimization process, and the DBI should be encouraged to use a modular design in his or her implementation efforts.

While pondering what a suitable model of optimization would be, we let ourselves be guided by the model of execution that we anticipate for future data models. Incidentally, this model of execution also received special attention in the design of the E programming language [Richardson1987]. A run-time system for a database typically consists of a limited set of procedures. Each of these procedures transforms a data stream according to an argument that was derived from the original query. A typical example is a selection operator which eliminates those tuples, records, or objects from a stream that do not satisfy a predicate provided in the query. To evaluate complex queries, such procedures can be nested, i.e. the output of one of them can be the input of another one. We call the transfer of data between such procedures a data stream or simply a stream, without making any assumptions about how this transfer is physically arranged, e.g. by temporary files, shared memory, or messages, and how the procedures are synchronized.

If we assume that this is how queries will be evaluated in EXODUS based databases, we can infer that queries can always be expressed as trees of operators. For EXODUS, we decided to require the user interface to produce a tree of operators that represents one correct sequence of operations to answer a query. Notice that we intend to automate this process using the user interface generator which is still in the early design phase. The operator tree is on the logical level only, it does not specify which algorithms and implementation methods are to be used. The optimizer will transform the presented query tree into one that promises more efficient execution of the query. This is done step by step, each step being the transformation of a query tree or a part of it into an equivalent query tree. These transformations include replacement of operators (e.g. Cartesian product and selection by a join), insertion of new operators (e.g. an additional project to eliminate fields as early as possible), and rearrangement of operators to achieve lower processing cost.

---

[1] This work was done at the Computer Sciences Department, University of Wisconsin — Madison.

Frequently, there is more than one implementation procedure for a given operation. For example, a number of join methods have been developed for the relational equi-join [Blasgen1977], and most database systems employ a repertoire of them. Therefore, the optimizers for EXODUS based database systems will distinguish between operators and methods. Operators are on the logical level, i.e. an operator and its argument determine the mapping from the input stream(s) to the output stream. Methods are on the physical level, i.e. a method specifies the algorithm employed. For example, a relational equi-join is an operator, and nested loops join, merge join, and hash join are corresponding methods for this operator. Part of the optimization process is to find the least expensive set of methods to implement a particular operator tree.

It is important to notice that the correspondence between operators and methods can be complex. A single method can implement more than one operator, or a single operator can require more than one method. Consider a relational equi-join as an example. If duplicate elimination is not considered a part of the project operator, it is easy to include a projection in any procedure that implements the join. A single method (algorithm, procedure) can perform more than one operator. Similarly, a merge join is only possible if both inputs are sorted, otherwise extra sort procedures are required. This potentially complex relationship between operators and methods must be captured in the optimization process. In EXODUS, it is part of the optimization to select which of several implementation methods for an operation is the most suitable in each case.

The easiest way to describe tree transformations and the correspondence of operator trees and method trees is by means of rules. Rules for operator reordering are termed **transformation rules**, and rules for method selection **implementation rules**. In our framework for optimization, the data model of the target DBMS is described by a set of transformation rules and implementation rules together with a set of cost functions to predict the processing cost for implementation alternatives.

The rule set must have two formal properties — it must be **sound** and **complete**. **Sound** means that it allows only legal transformations. If a rule is not correct, no optimizer employing this rule can work properly. It is impossible for a software tool such as the EXODUS optimizer generator to determine whether a set of rules is sound; this can only be determined by the DBI who defines the data model by specifying the rules. Verifying the soundness of the rules would only be possible if the data model can be described independently, and the two descriptions could be compared. **Complete** means that the rule set must cover all possible cases, such that all equivalent query trees can be derived from the initial query tree using the transformation rules. If the rule set is not complete, the optimizer will not be able to find optimal access plans for all queries. Again, this cannot be verified automatically because the set of equivalent query trees and access plans is defined only once.

To summarize, in our optimization model the optimizer maps a tree of operators into an equivalent tree of methods, and it does so by operator reordering and method selection. The set of operators and the set of methods are data-model-dependent, and hence must be specified by the DBI. Similarly, the rules that govern the tree transformations and the operator-to-method mappings must also be specified by the DBI.

2.1.1. Cost Model

The purpose of the optimization step is to find the least expensive execution plan for a given query. Clearly, the cost model plays a crucial role in the optimization process. It is very important that the cost model accurately anticipates execution cost for a given query. Cost measures (CPU, I/O, network transfer) and formulas to anticipate query processing cost in database systems have received significant attention [Selinger1979, Lohman1985, Mackert1986, Mackert1986a]. For generality, the only assumption about the cost model is that the sum of the costs of all methods in a plan is the cost of the entire plan. For the methods that are applied in a query evaluation plan, the appropriate cost function is invoked to calculate the cost for this method with the particular arguments and inputs.

2.1.2. Search Strategies

The search strategy adopted for EXODUS is a best first search [Barr1981]. A state in the general formulation of the search strategy is a query tree. Every state can, in general, be expanded (or transformed) in several directions. It is quite likely that the benefits of the various expansions differ considerably, hence we decided that not all states will be expanded fully in our search strategy. At any time during the search, there will be states which are exhaustively expanded, others which are partially expanded, and some which are unexpanded. The expansions which are possible but not applied yet are called the *OPEN* set. Each entry in *OPEN* is a pair of a query tree and a transformation rule.

All query trees and access plans explored so far are stored in a data structure called *MESH*. Since as many nodes as possible are shared among query trees, and since several different searches are supported on

MESH, *MESH* is a complex network of pointers.

At each step in the search, the transformation performed is the one which carries the ***most promise*** that it will eventually, via subsequent transformations, lead to the optimal query evaluation plan. The crucial element in this search strategy is the promise calculation, called the ***promise evaluation function***. It must include the current query and plan, other queries and plans which have were found earlier in the search process, and information about the transformation rule involved. The most natural measure for promise is the cost improvement of the access plans.

3. Modularization of DBI Code

In an extensible database system, there are always some parts in the optimizer (and in other components as well) that cannot be expressed in a restricted, e.g. rule-based language. These parts are best written in the DBI's implementation language. A software tool is used to combine the rules and the DBI's source code.

For easy extensibility, it is very important to assist the DBI in dividing the code into meaningful, independent modules. Not only is a modular optimizer easier to implement, we envision this as an aid for maintaining a database management system that evolves over time.

Some optimizer components can only be defined after the data model has been defined (data-model-dependent components), and hence must be provided by the DBI. In this section, we will briefly review these components, and how they are broken into modules. We generally associate these procedures with one of the concepts that we have introduced earlier, namely operators, methods, and rules.

3.1. Data-Model-Dependent Data Structures

There are two types of data-model-dependent data structures that are important in the optimization process. First, there are **arguments** for operators and methods. Second, in almost all cases it is desirable to maintain some dictionary information for intermediate results in a query tree. We term such dictionary information **properties** of the intermediate results. Since defining these data structures is part of customizing an extensible database system, the optimization component of such a system must treat these structures as "black boxes". In EXODUS, we define and use a procedural interface to maintain and query properties. Furthermore, we distinguish between operator and method arguments, and between operator-dependent and method-dependent properties. As an example from a relational system, cardinality and tuple width are operator-dependent properties, whereas sort order is a method-dependent property.

3.2. Rules and Conditions

In the EXODUS optimization concept, the set of operators, the set of methods, transformation rules, and implementation rules are the central components that the DBI specifies to implement an optimizer. The rules are non-procedural; they are given as equivalence laws that the generator translates into code to perform tree transformations. Each of these rules should be self-contained. Only then is it possible to expand the rule set safely as the data model evolves.

The rules express equivalence of query trees. Tree expressions, i.e. algebraic expressions, embody the shape of a tree and the operators in it. For some rules, however, applicability does not depend on the tree's shape and the operators alone. For example, some transformations might only be possible if an operator argument satisfies a certain condition. Since operator arguments should be defined by the DBI, such conditions cannot be expressed in a data model independent form. We allow the DBI to augment rules with source code to inspect the operator arguments, the data dictionary, etc.

3.3. Cost Functions

As mentioned earlier, processing cost occurs by executing a particular algorithm. The cost calculation is closely related to the processing method being executed. Hence, we associate cost functions with the methods, and calculate the cost of a query execution plan as the sum of the costs of the methods involved. The parameters of a cost function are the characteristics of the data streams serving as inputs into the method, e.g. the number of data objects in each input data stream, and the method argument, e.g. a predicate.

3.4. Property Functions

The characteristics of the data stream which are needed as parameters to the cost functions are data-model-dependent. Thus, they must be defined by the DBI. We attach characteristics, which we call properties, with both the operators and the methods. Operators (and their arguments) determine the logical properties of a node in a query tree, e.g. cardinality. A particular algorithm or method chosen defines physical properties of an

intermediate result, e.g. sort order.

Besides the conceptual differentiation of operator and method properties, there is also a practical reason why the two should be separated. The operator properties should be computed as early as possible, in particular before the cost functions are invoked. Determining the cost can be easier if the operator properties, e.g. cardinality, have been derived already. The method properties, on the other hand, can only be computed after the method has been determined, which is after the least expensive method has been found using the cost functions.

3.5. Argument Transfer Functions

Arguments to operators and methods, e.g. predicates, are also data-model-dependent, and can, therefore, only be modified by DBI code. If a transformation involves only operator reordering, there is a correspondence between operators in the old query tree and in the new query tree. In these cases, arguments can be copied between corresponding tree nodes. If this is not possible, the DBI is allowed to associate a function with a transformation rule. When the rule is applied to transform a query tree, this function is invoked to determine the arguments of the new operator nodes. For example, if a complex selection predicate must be broken up into smaller pieces, a function associated with the transformation rule is called to perform the necessary manipulations of the argument data structure.

3.6. Promise Estimation Functions

In the search strategy, a promise evaluation or estimation function is used to anticipate how beneficial the application of a rule will be, and to decide which transformation to apply next. It is not easy to design a general scheme to do this, and any general scheme will suffer from its generality; sometimes it might be necessary to consider data-model-dependent aspects of the query tree to be transformed. For example, join commutativity will, on the average, have a neutral effect. If there are asymmetric join methods like hybrid hash join [DeWitt1984], however, the benefit of join commutativity depends on whether one or both of the relations will fit into a main memory hash table. Therefore, an extensible database system should provide a general scheme to estimate the benefit of applying a transformation to a query tree, and leave the option to the DBI to augment this scheme with specialized estimation procedures.

4. The EXODUS Design and Implementation

We implemented a prototype of the optimizer generator. It was intended to serve several purposes. First, it shows the feasibility of the approach. Second, it is used to get preliminary performance figures. Third, it helps to identify the important parameters, their influences, and their ranges. Finally, it will constitute the optimization component of the EXODUS extensible database system.

In addition to the concepts introduced in the last section, we also allowed the DBI to organize the optimization as several phases. For each phase, there can be a different rule set, different search parameters, and different stopping criteria. The query tree corresponding to the optimal access plan of one phase serves as the input tree of the next phase.

To produce an optimizer, the DBI invokes the generator on a **model description file**. This is done only once, at **database system generation time**. The resulting query optimizer can then be used indefinitely. The output of the generator are two files: one with C code and one with C definitions to be included in other source code files written by the DBI. The model description file has two required parts and one optional part. The first required part is used to declare the operators and the methods of the data model. It can include C code and C preprocessor declarations to be used in the generated code. The second required part consists of transformation rules and implementation rules. The optional third part contains C code which is appended to the output file. The format of the model description file and the rule language used in the second part are described in [Graefe1987].

The third part of the model description file can be used to append source code to the optimizer code. For instance, it is a good place to put cost functions and other support functions. It is also possible, however, to put the source code for the cost functions and the other support functions in separate files. This section briefly describes the support functions and their purposes.

A cost function is associated with each method. The name is built by the generator by concatenating the word *cost* and the name of the method. To find the least expensive implementation method for a query tree, the procedure *ANALYZE* matches the query against the implementation rules, and calls the cost functions of the matching methods. The input arguments to a cost function are the node of the query tree in *MESH* and the nodes in *MESH* which would produce the input stream(s) according to the implementation rule.

With each operator or method, an argument specifies exactly how the operator or method should be applied to the input streams. An argument appears once in every node in the original query (for the operator) and in the access plan (for the method), and twice in every node in *MESH* (for the operator and for the method). Its type is defined by the DBI in a code section of the declaration part in the model description file. Typically, this field will have a *union*-type (C keyword for variant record).

When an argument transfer function is included in a transformation rule, this function is called to set all the argument fields in the new query tree immediately after the tree is created. To compare argument fields when searching for duplicate nodes in *MESH*, the DBI must provide a Boolean function called *DIFFERENT_ARGUMENTS*. When a duplicate node is removed, the function *UNDO_NODE* is called to allow the DBI to do some housekeeping if necessary, e.g. deallocate space that a pointer in an argument field points to.

By default, argument fields are copied from the initial query tree into *MESH*, between corresponding nodes in *MESH*, and from *MESH* to the final access plan. If this is inappropriate for some reason, these defaults can be overwritten. To do so, the DBI defines function names *COPY_IN* for transfers from the initial query tree into *MESH*, *COPY_OUT* for transfers from *MESH* to the final access plan, *COPY_OPERATOR_ARGUMENT* for use in transformation rules, and *COPY_METHOD_ARGUMENT* for use in implementation rules in a code section of the declaration part.

In each node in *MESH*, there are two fields called *OPERATOR_PROPERTY* and *METHOD_PROPERTY* which allow the DBI to store data dictionary information about intermediate results, for example cardinality and sort order. The types of these fields are defined by the DBI in a code section of the model description file. Information which can be derived from the operator, the operator argument, and the properties of the input stream(s) belongs in the operator properties, whereas information that depends on the chosen implementation method belongs in the method properties. These two are distinguished, as this allows the cost functions to use operator properties before method properties can be derived.

To derive the properties of *MESH* nodes, the DBI associates a property function with each operator and each method in the system. The names of these functions are the word *property* concatenated with the name of the operator or method respectively. The generator inserts calls to these functions into the generated code at the appropriate places.

As mentioned earlier, the DBI may wish to assist the optimizer in estimating the benefit of a transformation before the transformation is performed. A function name given in a transformation rule with the keyword *estimate* will be called by the optimizer to calculate the cost of a query after a transformation from the query tree, the operator arguments, and the operator and method properties.

At first glance, there seems to be a lot of code to write. Not all of the functions outlined above, however, are required. Only the cost functions are absolutely necessary. If no type for operator property fields is specified, operator property functions are not necessary. Similarly, calls to method property functions are only inserted into the generated code if a type for a method property has been specified. Argument transfer functions and estimation functions are only used if they are indicated in a rule.

Some or all of the functionality provided by these procedures is required in any optimizer. Since they are data-model-dependent, they cannot be provided for the DBI. What we have done, though, and what we consider important, is to provide a framework that breaks the data-model-dependent code in a query optimizer into small but meaningful pieces.

We expect that our system allows incremental development of database systems. Hopefully, even small subsets of operators, methods and rules can be designed, implemented, and tested independently.

5. Conclusions

To support query optimization in extensible database systems, the data model specific part must be separated from those parts that can be used for any data model. The reusable parts are part of the EXODUS effort, whereas the data model specific parts are defined by the Database Implementor (DBI). An optimization model general enough to fit all modern data models is provided to specify the data model specific parts for the optimization component. Our optimization model is based on an algebra view of the data model, i.e. on operator trees and method trees. Optimization in this model consists of operator reordering and method selection.

The data model specific part is captured in a rule language designed for this purpose and in a set of support functions written in the DBI's implementation language. The rules are transformation rules for operator reordering and implementation rules for method selection. In general, the rules are non-procedural, but the rule

language allows the DBI to support the optimizer by estimating the promise of a transformation. Support functions may be associated with operators, methods, and rules. They include the cost functions for methods, property functions for operators and methods, and argument transfer and estimation functions for rules.

Using a rule set as a base for the optimizer builds a very clear and concise framework for modularization of the DBI's optimizer code, and facilitates incremental development and testing.

The rules are translated by the optimizer generator into executable source code, which is compiled and linked with the support functions and other database software. Interpretative techniques were ruled out by our research because the resulting optimizers, including one prototype implementation undertaken in Prolog, are limited to the interpreter's search strategy, and bound to be slow.

One interesting design issue that remains is to provide general support for predicates as some form of predicate is likely to appear in all data models. Writing the DBI code for predicates, and operator arguments in general, was the hardest part of developing our optimizer prototypes. In the current design, the DBI must design his or her own data structures and provide all the operations on them for both rule conditions and argument transfer functions. It may be difficult to invent a generally satisfying definition and support for predicates, but it would be a significant improvement of the optimizer generator. The fact that predicates are a special case of arguments poses an additional challenge, since the overall design of the argument data structure must still remain with the DBI.

More generally, we realize that the optimizer generator works largely on the syntactic level of the algebra. The semantics of the data model are left to the DBI's code. This has the advantage of allowing the DBI maximal freedom with the type of data model implemented, but it has the disadvantage of leaving a significant amount of coding to the DBI. We therefore would like to incorporate some semantic knowledge of the data model into the description file. This, however, is a long-term goal to which we have not yet given much attention.

6. Acknowledgements

The author appreciates the generous support and advice by David DeWitt, Mike Carey, and the student members of the EXODUS project.

References

Barr1981.
A. Barr and E.A. Feigenbaum, *The Handbook of Artificial Intelligence*, William Kaufman, Inc., Los Altos, CA. (1981).

Blasgen1977.
M. Blasgen and K. Eswaran, "Storage and Access in Relational Databases," *IBM Systems Journal* **16**(4)(1977).

Carey1986.
M.J. Carey, D.J. DeWitt, D. Frank, G. Graefe, J.E. Richardson, E.J. Shekita, and M. Muralikrishna, "The Architecture of the EXODUS Extensible DBMS: A Preliminary Report," *Proceedings of the Int'l Workshop on Object-Oriented Database Systems*, pp. 52-65 (September 1986).

Carey1986a.
M.J. Carey, D.J. DeWitt, J.E. Richardson, and E.J. Shekita, "Object and File Management in the EXODUS Extensible Database System," *Proceedings of the Conference on Very Large Data Bases*, pp. 91-100 (August 1986).

DeWitt1984.
D.J. DeWitt, R. Katz, F. Olken, L. Shapiro, M. Stonebraker, and D. Wood, "Implementation Techniques for Main Memory Database Systems," *Proceedings of the ACM SIGMOD Conference*, pp. 1-8 (June 1984).

Graefe1987.
G. Graefe and D.J. DeWitt, "The EXODUS Optimizer Generator," *Proceedings of the ACM SIGMOD Conference*, pp. 160-171 (May 1987).

Lohman1985.
G. Lohman, C. Mohan, L. Haas, D. Daniels, B. Lindsay, P. Selinger, and P. Wilms, "Query Processing in R*," pp. 31-47 in *Query Processing in Database Systems*, ed. J.W. Schmidt,Springer, Berlin (1985).

Mackert1986.
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).

Mackert1986a.
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).

Richardson1987.
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).

Selinger1979.
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*, (June 1979).

Understanding and Extending Transformation-Based Optimizers

Arnon Rosenthal
Computer Corporation of America, 4 Cambridge Center,
Cambridge MA 02142. arnie@cca.cca.com (arpanet) or cca!arnie (uucp)

Paul Helman
Computer Science Dept., University of New Mexico,
Albuquerque NM 87131. unmvax!helman (uucp)

ABSTRACT

We present elements of a framework for describing query optimizers, especially optimizers based on transformations. Using the framework, we discuss correctness of the optimizer, modeling the computational steps of current optimizers, convenient specification of search strategies, analyzing the search process, and addition of new transformations.

1. Introduction

We provide formalisms for modelling query optimizers, and discuss ways of making optimizers extensible. A major barrier to better optimizer architectures is the inadequacy of formal frameworks in textbooks and the research literature. For example, many designers still learn the subject by studying the classic papers (e.g. [SELI79]), which present algorithm fragments and intuitions, but few unifying formalisms. In our experience, designs produced after such study tend to use too low a level of abstraction in data structures and operations, and therefore to lack generality and modularity. In fact, designers often lack criteria for determining whether an optimizer modification is valid. A more thorough formalization would aid understanding of both conventional and transformation-based optimizers.

Optimizer extensibility is needed to accomodate new query language operators (e.g., recursion, outerjoins), new datatypes, semantic query optimization, multiple query optimization, and contingency planning for uncertain environments. Researchers are actively developing new optimization algorithms for these purposes. But with current optimizer architectures, adding a new algorithm requires modifying considerable existing code.

For example, suppose one has a "semantic query optimizer" that suggests alternative translations of various subexpressions of a query, to exploit integrity constraints. The efficiency of each alternative can be judged only after checking low-level details such as availability of indexes to support each join. Even a system programmer would have difficulty integrating this new capability into most existing optimizers, except by evaluating each combination of alternatives as a separate query, leading to an exponential number of queries.

To alleviate these problems, we try to organize or model optimizers as having three nearly-orthogonal aspects -- *operator graphs*, *transformations of those graphs*, and *search-control rules*. Like other researchers [BATO87,FREY87,GRAE87], our approach is to use transformation rules that can be added (nearly) independently.

Sections 2 and 3 discuss operator graphs and transformations, respectively. Our definitions of operator graphs and transformations provide some refinements of previous graph formalisms (such as data nodes, a view of operators as predicate transformers, and correctness conditions at any stage of optimization). Section 4 discusses theoretical and practical issues in searching a space

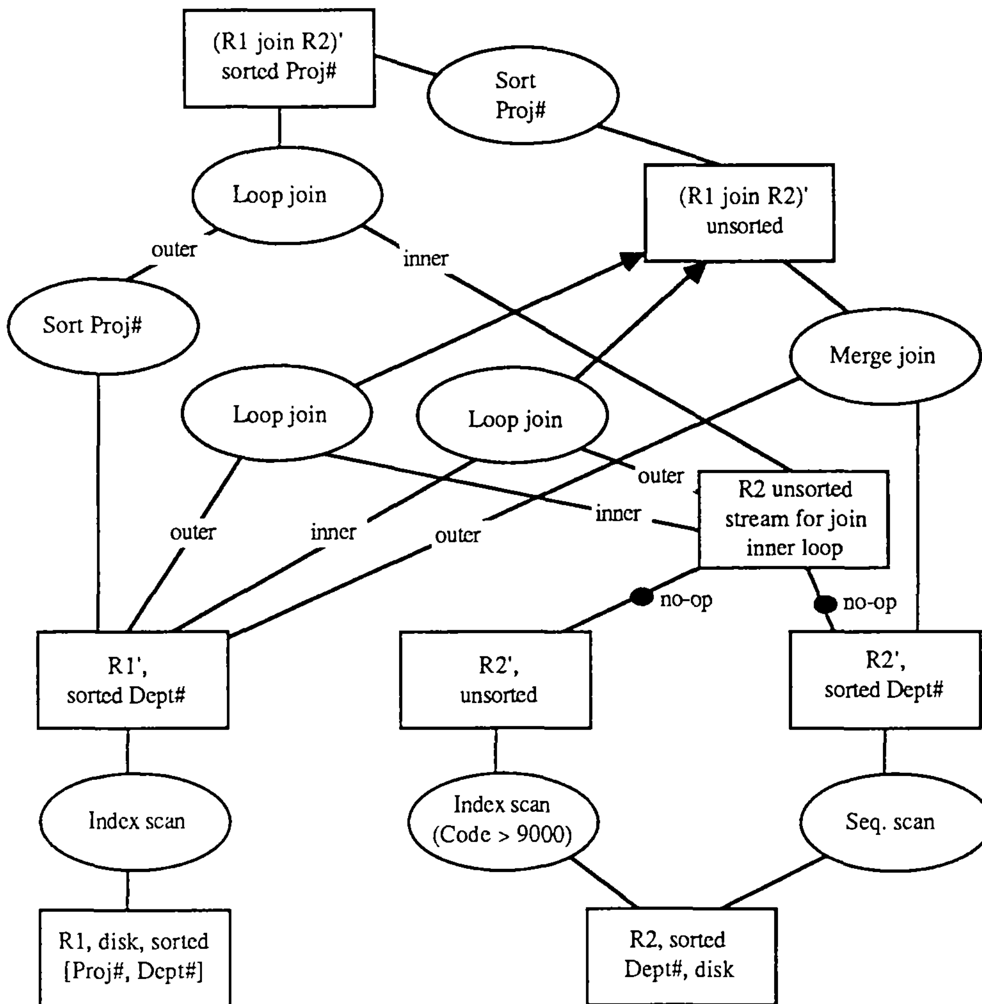
described by transformations.

Our results include the ability to directly model computational strategies in the optimizer, a basis for verifying a wide variety of optimizers, ways of controlling search in an extensible optimizer, and criteria for comparing the breadth and thoroughness of the searches carried out by different optimizers.

2. Operator Graphs

In this section we describe how an operator graph is used to model alternative query processing strategies. When the graph is "well formed," we can guarantee the correctness of any strategy selected from the graph. We also explain ways that our node and predicate formalisms differ from [BATO87,FREY87,GRAE87].

An *operator graph* is an acyclic digraph composed of *operator nodes*, which may represent arbitrary operators such as join or sort, and *data nodes*, which represent inputs to -- and results from -- operators. An *expression* (or, synonymously, *strategy*) is an operator graph in which each data node has only a single incoming path; it is analogous to one solution of an AND/OR graph. The figure below illustrates a sample operator graph at the detailed level, for the query shown.



Select R1.A, R2.B
From R1, R2
Where (R1.Proj# = 123) and
(R2.code = 9000) and
(R1.Dept# = R2.Dept#)
Order By R2.Age

Note: After query inputs are scanned, all operands are tuple streams. Apostrophes on relation names indicate that all relevant selections (e.g., R1, Proj # = 123) have been applied.

Data nodes are labelled with *node predicates* (not to be confused with query predicates) that describe the result available at the node. Conjuncts of the predicates (called *node properties*) include logical data content as a relational expression over the inputs, sort ordering, location, etc. The predicates are an interface specification, a promise to later operators about the results of all expressions entering the node. When a data node has multiple incoming edges, each edge represents an alternative computation which produces a result satisfying the node's predicate.

An *elementary graph* consists of one operator, its input and output data nodes, and edges connecting them. An elementary graph is *well formed* if [inputs satisfy the predicates on the input node(s)] is sufficient to imply [outputs satisfy the predicates on its output data node(s)]. The condition can be tested by using a description of each operator's semantics, expressed as a predicate transformation (in the sense of program verification). For example, in the previous figure the R2 input node, sequential scan operator, and (R2' sorted on Dept#) form an elementary subgraph; sequential scan transforms the input predicate to stream and (in our interpretation) applies relevant selections.

Well-formed graphs are the basic "correctness" condition to be preserved during algorithm modifications and handled by a verifier. An arbitrary graph is called well-formed if each elementary subgraph is well-formed. It is easily shown that if the inputs to a well-formed subgraph G obey the predicates on G's input nodes, the output of any expression entering an output data node D will obey the predicates on D. In particular, when the query inputs satisfy the predicates on input nodes, and the output node predicate implies the desired query result, then any expression for the graph is a correct computation for the query.

Operator graphs are helpful in understanding and verifying two very different optimization approaches. *Query transformers* start with an operator graph for the query and repeatedly apply transformations, always keeping a well formed operator graph that implements the entire query. This approach is used in many proposals for extensible optimizers. In contrast, older optimizers are bottom-up *strategy generators* that gradually produce an operator graph. The building process typically runs off an auxiliary data structure (e.g., a "query graph," showing relations and incident predicates).

A further advantage of operator graphs is that they can contain operators at different conceptual levels. This is useful when optimizing a global query to be executed over several different DBMSs (e.g., relational and Codasyl) that have interfaces at different levels.

Data Nodes: Data nodes are not explicit in most transformation-based optimizer formalizations, but can be quite useful, as described below. When their power is not needed, they can be omitted, with incoming edges routed directly to the next operator.

A data node provides a location for information such as the node predicate, result size, and best known cost. Transformations that insert unary operators into the graph are more naturally applied to a data node rather than to all binary operators that may use the node. For results that may be produced in several ways, and used by several follow-on operators (or by several independent queries), a data node removes the need for pairwise connections. Alternative (or parallel) plans for a subquery can be attached to a data node, to cope with failures or to optimize unpredictable subqueries by running alternative algorithms on separate processors. Data nodes also are useful in multiple query optimization, since they help to identify common intermediate results.

A data node can be created anytime we wish to describe a convenient intermediate result. For example, one can create a separate data node representing "cheapest data node for R1, regardless of sort order". Operators that are oblivious to the input's sort order (Sort, inner loop of nested join) can use just this node instead of each of the R1 data nodes separately.

3. Operator Graph Transformations

A transformation-based optimizer applies a series of transformations to an operator graph. The transformations can *decompile* subgraphs to a higher level of abstraction (such as combining a query and its subqueries so as to allow more join orderings), *generate alternatives* (such as alternative forms based on semantic query optimization), *elaborate a strategy to a more detailed level* (multi-argument joins to two argument joins, and then to nested loops), or can *postprocess* the strategy (performing local improvements that do not affect the optmizer's search, e.g., minimizing the labor of projections).

A transformation is described by an *initial graph* and *result graph*, also called *patterns*. The result graph's data nodes are labelled by predicates; predicates on its input nodes are called "applicability conditions." Conceptually, applying a transformation consists of two steps.

1. Find an instance of the initial pattern within the current operator graph. For a match to be found, the predicates on the input nodes must imply the applicability conditions. Properties of the result nodes may receive values from corresponding properties of the initial pattern.

2. Add the result graph to the operator graph, and connect its outputs to appropriate data nodes in the result.

Correctness conditions: The formalism suggests a feasible approach to verify strategies -- simply verify that all procedures that add nodes to the graph preserve well-formedness. (Correctness is not usually verified when the query is compiled -- rather, builders of the optimizer verify that each routine that modifies the graph preserves these conditions.) While this approach does not verify the efficiency of the generated strategy or termination of the optimizer, correctness of compiled code is often the main goal, especially in critical or secure applications.

To verify a transformation-based optimizer, three conditions must be proved about each transformation: First, the transformation's result graph must be well-formed. Next, the input data nodes' predicates in the operator graph must imply the predicates on the input nodes of the input pattern (often by possessing a superset of the node properties). Finally, predicates on the result graph's output nodes must imply the predicates on data nodes to which they are attached. If the above conditions hold, the transformation introduces no violations of "well-formedness." The conditions apply to transformations used in either query transformation or strategy generation approaches. The first condition can be verified when the transformation is defined; the other two involve simple tests to be included in the pattern matching. [1]

An alternative way of justifying transformations is to show that each transformation's result pattern preserves all properties of the result node in the input subgraph. However, this formulation does not handle transformations that build a strategy graph bottom-up, creating new result nodes that do not correspond to nodes of the input pattern.

It seems desirable to express as much as possible of compilation as transformations of operator graphs. When we leave the theoretical framework we lose both conceptual tools (e.g., "well-formed graphs") and software tools (e.g., pattern matchers, prettyprint debugging routines).

---

[1] Most properties are atomic (e.g., "sort-field=A1") and can be checked by direct inspection. Non-atomic properties are only slightly more difficult to check. For example, if subexpressions place the query's restrictions and projections as early as possible, then checking for a match between relational subexpressions means determining whether they involve the same set of relations.

By modelling optimization as adding alternatives to a single operator graph, we are able to minimize redundant optimization and complex tests for common subexpressions. To see this, consider the alternative scenario of adding transformations above a conventional optimizer (whose facilities for managing alternatives and recognizing common subexpressions are embedded deep within Select/Project/Join optimization). If a high level transformation like semantic query optimization simply generates alternative queries, one cannot tell which is best without expanding the alternative implementations of each. Suppose the original query involves N nodes, and each of m such transformations (applied to different subgraphs) generates an alternative form of its input graph. Then 2**m complete queries will be generated, each containing roughly N nodes; a total of 2**m N-node graphs need to be expanded, an exponential explosion. In contrast, operator graph transformations generate alternatives within the same subgraph. If s denotes be the number of nodes added in an average subgraph, one graph containing N+m*s nodes needs to be expanded. [2]

4. Searching

Section 4.1 provides a theoretical analysis of searches that guarantee optimality within a strategy space. Section 4.2 discusses the management of an extensible set of transformations, and mechanisms for choosing the next transformation to apply.

4.1 Analysis of the search -- definitions from combinatorial optimization

Describing the thoroughness of the search: This section attempts to provide a vocabulary for analyzing design decisions in the search strategies of an optimizer. Like the proverbial drunk who looks for his keys only under streetlights (the light is better), we focus here on the more tractible analysis of searches that guarantee not to miss an optimum.

In classical (operations research) optimization theory, an optimization problem is defined by its search space (i.e., the feasible solutions) and the cost model. Unfortunately, for query optimization the search space is hard to describe. Feasible solutions are *Query Evaluation Plans (QEPs)* -- well-formed expression with appropriate predicates on input and output nodes, and operators described in sufficient detail for cost modelling and execution. But QEPs are still subject to more global constraints on execution, constraints that may be difficult to describe or exploit. For example, if an expression is not a tree, the strategy must be checked to see if it correctly synchronizes multiple uses of the same tuple stream.

Transformation-based formalisms are a more procedural way of describing the optimization problem. They can describe much of the processing in current optimizers, and can naturally define each transformation at an appropriate level of abstraction. However, it is difficult to get an overview of the entire set of candidate strategies. And there is still no explicit model of execution limitations to help the transformation-definer ensure that all generated strategies are executable.

We now present some definitions that are useful in understanding optimizers' search mechanisms. The *potential operator graph* of a set of transformations is the operator graph generated by applying transformations until no more apply. Assuming that transformations do not delete their input graphs, the definition is order independent. The *potential strategy space* for a query Q consists of all expressions in the potential operator graph. It has operators at many different levels of

---

[2] The number of detail-level data nodes obtained from a high level node may increase slightly, if more sort orders become "interesting".

abstraction, and may be conveniently described level by level. (For example, what are the legal sequences of joins; what subgraph can be the implementation of a join, what interesting sort orders should be considered?)

The *searched strategy space* consists of those strategies that have been considered (explicitly or implicitly) in the search. Typically, we restrict attention to detail-level strategies, since this is where strategy elimination occurs. (Rules for eliminating suboptimal strategies are discussed below). An optimizer *fully searches* its potential strategy space if it performs sufficient strategy generation and cost evaluation guarantee that it has found the lowest cost expression in the space. (In contrast, *directed searches* try to quickly find a good solution). If all elaborations of some abstract-level subgraph have been searched, we say that that abstract level has been searched.

System R uses dynamic programming to fully search its potential strategy space. Elimination is based on *substitutability* (the principle of optimality): If two partial expressions are sufficiently similar that one can be substituted for the other, then the more expensive partial solution cannot appear as part of an optimal expression. All extensions of the loser are eliminated from consideration. Substitutability requires that the cost function for an operator know nothing about the operator's inputs except information determined by predicates on its input data nodes. Also, cost of an expression must be the sum of individual operator costs.

Another rule is *bounding*: Any partial QEP that costs more than some complete QEP must be suboptimal. The major saving from bounding is that a good bound can eliminate *all* strategies into some data nodes (e.g., which can only be produced using sequential scans on large relations). The eliminated data node then does not appear as an input to other patterns. A low cost QEP can be generated by a directed search [GRAE87,ROSE86].

A third elimination rule is for the implementer to assert *dominance*, that a transformed subgraph T(S) dominates S. (S' *dominates* S if the best expression derived from S' is as cheap as the best derived from S). For example, transformations that move selections earlier, remove inessential joins, or replace outerjoins by ordinary joins often exhibit dominance. Dominance is a higher-level elimination rule than substitutability or bounding in the sense that it relies on an implementor assertion rather than on a cost model.

How not to speed an optimizer: For optimizers that thoroughly search their strategy space, simplicity and flexibility should probably govern whether the optimizer generates all alternatives at less detailed levels. We expect that *most of the optimization time will be spent in generating implementation-level strategies, and especially in evaluating operator costs.*

Several researchers have mentioned the possibility of looking ahead from a data node to determine the cost of completing the computation above the data node. That is, one somehow finds a lower bound L on the cost of completing the computation above the data node. Then if B is the cost of the best known QEP, B-L (rather than B) is the maximum allowable cost for strategies into the data node. We suspect that such techniques will be "cleverness sinks" with little effect on performance. It seems unlikely that the improved cutoff of B-L will eliminate, say, 50% of the nodes that survive simple cutoffs and substitutability.

In general, accurate cost estimates seem obtainable only by analyzing detailed strategies. Accurate estimates are not possible until all large transformations have been completed. (For example, in [ROSE85] semantic transformations exploit constraints to eliminate unnecessary joins). Inaccuracy in estimating an operator's cost will be multiplied by the number of iterations over that operator within a QEP. For example, an error in the cost of a Scan used on the inner relation of nested-loop join is multiplied by the number of outer tuples.

4.2 Transformation Selection in an Extensible Optimizer

This section focuses on the *selection mechanism* that selects the pair (transformation, matched input graph) for the next transformation, and on how this mechanism is affected by optimizer extensions. The selection mechanism requires speed, flexibility for adding new operators, and the intelligence to expand the most promising parts of the operator graph. [3]

Extensibility approaches

The Postgres approach to extensibility [STON86] does not add new optimizations, but instead extends existing rules to new data types. For example, the implementer of a spatial datatype may identify generic optimization techniques (retrievals via index, sorting, etc.) that are to be applied to the new datatype. The search process and strategy space are not really changed by the new types, and the optimizer need not be transformation-based.

A more difficult problem is to add optimization techniques that do not have analogues in the current optimizer [GRAE87, BATO87]. New methods must be added to handle operators not currently subject to optimization (e.g., outerjoins, nested subqueries, recursion), and for adding alternative strategies for existing joins (e.g., index intersections, which greatly improve performance of some queries [ROSE82]).

A transformation-based optimizer is relatively easy to extend, because transformations can be written and validated separately. In our framework, adding new transformations requires mechanisms for *extending the set of data node properties* [easily accomplished by representing properties using a set of pairs (property_name, legal values)] and for *extending the transformation selector to include the new transformations.*

Transformation Selection

A monolithic program for selecting transformations is too difficult to modify as new types of transformations are added. Below, we speculate about extending a more promising approach. [GRAE87] uses *expected cost factors* (denoted "ECFs") to evaluate the likelihood of getting a useful result from a transformation. Their selection mechanism is very extendable, because ECFs are learned from previous queries, without implementer intervention. In experiments on select/project/join queries, ECFs provided excellent results.

Nevertheless, we suspect that larger rule bases will follow the same path as other rule-based applications, and will need to be structured. This structure must then be considered when extensions are made. (It is preferable to define appropriate structuring constructs rather than to subtly encode control information into the rules.) EXODUS [GRAE87] and GENESIS [BATO87] include some limited implementer-supplied structuring, in the form of declarations that certain rules are 1-way or nonrepeatable. Even these simple structuring rules may need to be modified when the optimizer is extended. For example, a transformation that elaborates multiple-input joins to 2-input joins will be "one-way" in a select-project-join optimizer. But when optimizing a user query over a view-relation, one begins by collapsing two-input joins from the query and the view definition into a single three-way join.

[3] Database design systems need very fast optimization, in order to find the optimum strategy for a wide range of physical designs.

To illustrate the need for implementer help in the overall control, imagine that independent groups have supplied transformations for semantic query optimization, outerjoins, recursion optimization, and nested subqueries (including views). The ECF for a transformation depends on whether the resulting graph has an efficient implementation. For transformations whose result is at a fairly abstract level, "implementation" involves both simplification transformations (e.g., detection of unnecessary joins), and low-level details such as available buffer space or indexes. Hence the abstract transformations are justified by good solutions many steps ahead. This long lookahead makes it difficult to learn good orderings automatically; for example, could a program efficiently learn whether collapsing nested subqueries to multiway joins should be given higher priority than selecting a recursion implementation? Instead, the implementer might specify the relative priorities for "clusters" of transformations.

5. References

[BATO87] D. Batory, "A Molecular Database Systems Technology", Computer Science Department TR-87-23, University of Texas at Austin.

[FREY87] J.C. Freytag, "A Rule-Based View of Query Optimization", *Proc. ACM-SIGMOD Conference on Management of Data*, San Francisco, 1987.

[GRAE87] G. Graefe and D. DeWitt, "The Exodus Optimizer Generator", *Proc. ACM-SIGMOD Conference on Management of Data*, San Francisco, 1987.

[ROSE82] A. Rosenthal and D. Reiner, "An Architecture For Query Optimization," *Proc. ACM-SIGMOD Conference on Management of Data*, Orlando, FL, 1982.

[ROSE85] A. Rosenthal and D. Reiner, "Querying Relational Views of Networks," in W. Kim, D. Reiner, and D. Batory, eds., *Query Processing in Database Systems* (New York: Springer-Verlag, 1985).

[ROSE86] A. Rosenthal, U. Dayal, and D. Reiner, "Fast Query Optimization over a Large Strategy Space", (draft, available from the authors)

[SELI79] P. Selinger et. al, "Access Path Selection in a Relational Database Management System", *Proc. ACM-SIGMOD Conference on Management of Data*, 1979.

[STON86] M. Stonebraker and L. Rowe, "The Design of POSTGRES", *Proc. ACM-SIGMOD Conference on Management of Data*, Washington DC, 1986.

THE COMPUTER SOCIETY
OF THE IEEE
1730 Massachusetts Avenue, N W
Washington, DC 20036-1903

Non-profit Org.
U.S. Postage
PAID
Silver Spring, MD
Permit 1398