Query Optimization in a Memory-Resident Domain Relational Calculus Database System

KYU-YOUNG WHANG and RAVI KRISHNAMURTHY
IBM Thomas J. Watson Research Center

We present techniques for optimizing queries in memory-resident database systems. Optimization techniques in memory-resident database systems differ significantly from those in conventional disk-resident database systems. In this paper we address the following aspects of query optimization in such systems and present specific solutions for them: (1) a new approach to developing a CPU-intensive cost model; (2) new optimization strategies for main-memory query processing; (3) new insight into join algorithms and access structures that take advantage of memory residency of data; and (4) the effect of the operating system's scheduling algorithm on the memory-residency assumption. We present an interesting result that a major cost of processing queries in memory-resident database systems is incurred by evaluation of predicates. We discuss optimization techniques using the Office-by-Example (OBE) that has been under development at IBM Research. We also present the results of performance measurements, which prove to be excellent in the current state of the art. Despite recent work on memory-resident database systems, query optimization aspects in these systems have not been well studied. We believe this paper opens the issues of query optimization in memory-resident database systems and presents practical solutions to them.

Categories and Subject Descriptors: H.2.2 [**Database Management**]: Physical Design—*access methods*; H.2.3 [**Database Management**]: Languages—*query languages*; H.2.4 [**Database Management**]: Systems—*query processing*

General Terms: Algorithms, Experimentation, Languages, Performance

Additional Key Words and Phrases: CPU-intensive cost model, database query language, domain relational calculus, example element, memory-resident database, query optimization

1. INTRODUCTION

Office-by-example (OBE) [43, 48, 49] is an integrated office system that has been under development at IBM Research. OBE extends the concept of query-by-example (QBE) [50]—a relationally complete database language [36]. It supports various objects needed in a typical office environment: database relations, text processors, electronic messages, menus, graphics, and images. In OBE most applications involving these objects are processed by the database manager,

Authors' current addresses: K.-Y. Whang, Computer Science Department, Korea Advanced Institute of Science and Technology, P.O. Box 150, Cheong-Ryang Ni, Seoul, Korea; R. Krishnamurthy, M.C.C., 3500 West Balcones Center Dr., Austin, TX 78759.

which constitutes the backbone of the entire system. We discuss query optimization in a domain relational calculus language in the context of the OBE system.

The database system consists of three components: a hierarchical relational memory structure (HRMS), the query processor, and the query optimizer. HRMS is the low-level system component that provides elementary access functions that retrieve individual tuples. (Even though the name might indicate otherwise, facilities supporting hierarchies of relations have not been implemented.) Based on these primitives, the query processor implements various types of high-level scans over the relations for retrieving tuples satisfying certain criteria (i.e., predicates). The query optimizer, using these high-level primitives as its options, finds the optimal access strategy for processing a query.

For the design of the database manager, we adopt the unconventional assumption of memory residency of data; that is, part of the database accessed in processing a query resides in main memory. This assumption entails many interesting ramifications. In particular, the data structures and algorithms must be designed to maximize the benefit of memory residency of data. Simply keeping the data in main memory does not automatically guarantee good performance. We discuss in Section 6 how the assumption of memory residency can be reasonably approximated in a practical environment. Since a practical time-sharing system does not always allow a large amount of real memory for each user, we emphasize the importance of proper coupling of the memory-residency idea to the operating system's scheduling algorithm. Especially, pure demand paging is not adequate for memory-residency assumption. We also present the result of measurements on OBE's performance.

The idea of memory-residency of data has also been investigated in [10, 11, 25, 26, 33]. DeWitt et al. [10] compare performance of conventional data structures when a large amount of real memory is available. Garcia-Molina et al. [11] present a new logging/recovery scheme with specialized hardware in a memory-resident database system with infinite real memory. Lehman and Carey [25, 26] introduce a new indexing structure called the *T-tree* and present join algorithms based on the T-trees (we discuss them later). Shapiro [33] introduces a hash-based join algorithm that can be applied efficiently when the size of the main-memory buffer is equivalent to the square root of the size of the relation processed. DeWitt et al. [10] and Shapiro [33] consider the large main memory as a buffer for the disk-resident databases. On the other hand, Garcia-Molina et al. [11] and Lehman and Carey [25, 26] regard main memory as the main depository of data. Here, disks are used only for permanent storage of data and backup. Our approach belongs to the latter, in which different sets of data structures and algorithms have to be devised to take full advantage of the memory-residency of data.

Implementations of Prolog [9] that has been widely used as a logic programming language also have some flavor of memory-resident database systems. Warren [38] points out the similarity between a subset of Prolog and the relational database and proposes an optimization technique based on a variant of the hill-climbing technique [45] with an objective (i.e., optimization criterion) of minimizing tuple accesses. In this heuristic technique, only the goal (relation) ordering is considered; various access structures, such as indexes, typically used in database

management systems are not taken into account in the optimization. In Section 5 we present different optimization criteria suitable for memory-resident database systems and the optimization techniques based on the new criteria. Our technique is not a heuristic since it employs a systematic search based on a complete cost model. In general, Prolog has been emphasized more as a programming language than as a memory-resident database system.

Typically, in a modern relational database system, optimization involves a cost model specific for the system. The cost can be classified into two categories: the CPU computation cost and the I/O access cost. Since I/O devices are typically much slower than the CPU, many traditional approaches to query optimization emphasized the I/O access cost [6, 14, 27, 31, 32]. (The CPU computation cost has been incorporated in System R [32] to a limited extent.) In memory-resident database systems, however, the CPU computation cost becomes the dominant cost. Since all the data are supposed to be in main memory, ideally, only the CPU computation cost will exist. Modelling the CPU computation cost is difficult for various reasons. We discuss the problems in Section 5 and propose a new technique of developing the cost model for CPU-intensive memory-resident database systems.

Many optimization algorithms have been reported in the literature [16, 19, 30, 32, 46]. We classify these techniques into two classes: heuristic optimization and systematic optimization. Heuristic optimization employs heuristic rules to transform the query into an equivalent one that can be processed more efficiently. Typically, heuristic optimization has been widely used for relational algebra optimization [13, 28, 34]. (Heuristic optimization is also used in INGRES [35, 46]—a relational calculus based system.)

Systematic optimization finds the optimal solution by computing and comparing the costs of various alternative access strategies. For the estimation of the costs, it uses a cost model that reflects details of the physical access structures. The systematic optimization has been initially proposed by Selinger et al. [32] for System R [2, 7]. System R's query optimizer uses statistics of data (such as relation cardinalities and column cardinalities) extensively in its cost model. It also incorporates CPU computation in the cost model to a limited extent. Systematic optimization has also been used in a later version of INGRES [23].

Warren's technique [38] falls somewhat in between the two approaches. It uses a rudimentary cost model based on the cardinalities of predicates (relations) and domains and is driven by a heuristic (hill-climbing) technique to find a suboptimal solution.

Our approach employs systematic optimization using a cost model of CPU computation suitable for memory-resident database systems. As in System R, we use statistics extensively to produce a cost estimation. To avoid exhaustive search for alternative access strategies, we use the branch-and-bound algorithm [15]. The branch-and-bound algorithm is functionally equivalent to an exhaustive search in that it produces an optimal solution: it prunes the search space only when it is guaranteed that no optimal solution can be found in that search space.

The purpose of this paper is to present the architecture of the query optimizer, the technique for developing a CPU-intensive cost model, and new aspects of query processing, optimization, scheduling algorithms, and data structures that

are derived from the memory-residency assumption of data. In addition, we present the concept of example-element binding that is unique in domain relational calculus languages. We discuss how this concept is integrated into the optimization process in our system. Despite recent work on memory-resident databases (especially on join algorithms and data structures), the query optimization issues in these systems have not been seriously studied. We believe this paper provides a significant progress toward understanding comprehensive issues involved in designing a practical query optimizer for a memory-resident database system.

The paper is organized as follows. In Section 2 we briefly review the part of the OBE language that is necessary for the discussion of query optimization. In Section 3 we present query processing techniques in our system and define some terminology. Especially, we focus on the aspects specific to domain relational calculus languages. Then, in Section 4, we briefly discuss the storage representation of the database manager. Next, Section 5 presents in detail the optimization strategies, optimization algorithm, and the method for developing the cost model. In Section 6, we present the results of performance measurements and discuss the impact of the operating system's scheduling algorithm on the memory-residency assumption. Finally, Section 7 summarizes the results and concludes the paper.

2. A DOMAIN RELATIONAL CALCULUS LANGUAGE

In this section we illustrate how queries are expressed in the OBE language by using a few examples. We concentrate only on the data manipulation part of the language. The detailed description of the language can be found in Zloof [48].

Example 1. Consider a relation, Sales. Suppose we want a relation of salespersons and their sales to date exceeding $50,000. An OBE program to do this is shown in Figure 1. In Figure 1 the operator P. stands for *present* and specifies projection of the corresponding columns in the output.

Example 2. This is an example of a join query involving three relations. A join can be specified using an *example element* to map the matching columns. An example element begins with an underline followed by any alphanumeric string. The appearance of an example element in two or more positions indicates that the data instances corresponding to those positions should have the same value.

Consider relations, Directory, Sales, and Employees. We want to see the names of salespersons who are not managers and their phone numbers if their sales to date exceed $50,000. The program for this query is illustrated in Figure 2. In Figure 2 the example element _N *maps* the columns in the relations Directory and Sales to specify a join. The operator ¬ specifies *row negation* (meaning NOT EXIST) [50]. The row negation in Employee relation specifies that there be no tuples having the value of _N in the Manager column.

Operators and Commands

The OBE language supports the following operators: P. (present), I. (insert), D. (delete), U. (update), S. (send), X. (execute). The language also provides aggregate

Sales	Salesperson	Sales Quota	Sales To Date
	p.		p.>50,000

Fig. 1. An OBE program containing one relation.

Directory	Name	Phone
	p._N	p.

Sales	Salesperson	Sales Quota	Sales To Date
	_N		>50,000

Employee	Ename	Manager
		_N

Fig. 2. An OBE program containing three relations and negation.

R1	Col1	Col2	Col3
(Node 1)	_X		_Z
(Node 2) ¬		_Y	

R3	Col1	Col2	Col3
(Node 4)	_X	_Y	

R2	Col1	Col2	Col3
(Node 3)	_X+_Z		150

&Out			
P.	_X	_Y	_Z+5

CONDITIONS
_X > 5

Fig. 3. An example query.

functions: SUM. (sum), CNT. (count), AVG. (average), MAX. (maximum), MIN. (minimum), UNQ. (unique), ALL. (duplicate), and G. (grouping). The operator ¬ specifies row negation.

Other Constructs

To project results derived from more than one relation, we use a *user-created output.* We prefix the relation name with the symbol & to indicate that the relation is a user-created output. An example is shown in Figure 3.

3. QUERY PROCESSING

In this section we present the issues involved in processing queries in a domain relational calculus language. We also define terminology that is used throughout this paper. We use the example in Figure 3 for illustrative purposes.

On the screen for a query, multiple rows can appear in a relation skeleton. Each row contains predicates and operators that are to be applied to the tuples of the relation. Each row refers to a different instantiation of the relation. We define a *node* as an instantiation of a relation with associated predicates and operators as specified in a row. For example, in Figure 3, we have four nodes, two of which represent different instantiations of relation R1. Objects that do not have corresponding base relations are not treated as nodes. For example, the user-created output and the condition box are not nodes. We call a node a *negated node* if it involves row negation; otherwise, we call it an *ordinary node*. Node 2 in Figure 3 is an example of a negated node.

A *restriction predicate* is a predicate containing only literal values. Node 3.Col3 = 150 in relation skeleton R2 is an example of the restriction predicate. A *join predicate* is a predicate containing an example element that appears in a node or in a condition box. Node 4.Col1 = _X is an example of the join predicate. It becomes a *bound join predicate* if all of its example elements have been assigned specific values (i.e., *bound*); otherwise, it becomes an *unbound join predicate.*

A join predicate can be used for two purposes. (1) It can bind example elements to specific values contained in the tuple retrieved from the database. For this purpose, the join predicate must be a "bindable instance" of an example element. A join predicate is a *bindable instance* of an example element if, when interpreted with values in a tuple, the example element can be bound unambiguously. For example, the join predicates in Node 1 and Node 4 are all bindable instances, whereas the join predicate in Node 3 is not. In our implementation, we treat only join predicates of the form Column = _X as bindable instances. We do not treat a predicate of the form Column = _X + 5 as a bindable instance for performance reasons since we have to perform an arithmetic operation to bind the example element. (2) A join predicate can select qualified tuples. For this purpose, all example elements in the join predicate must be bound. If any example element is unbound, the evaluation of the predicate is deferred until all the example elements are bound.

We define a column of a node with a restriction predicate as a *restriction column*; a column with a join predicate is a *join column.* For example, Col1 of Node 1 is a join column; Col3 of Node 3 is a restriction column.

Processing a query consists of retrieving the data, checking for satisfaction of the conditions, computing aggregations if necessary, and constructing the output result. The most important part of query processing is evaluation of the join. The query processor treats all the queries as joins; thus, a single-node query is a special case of the join. We first discuss join methods and then describe other aspects of query processing.

There are numerous join methods reported in the literature [6, 21, 32, 46, 47]. Two join methods are frequently used in database systems: the nested-loop join method and the sort-merge join method [32, 40].

In disk-based database systems the sort-merge join method has an advantage over the nested-loop join method when a majority of tuples in the relations are

accessed. Since it processes all the tuples in a block sequentially after sorting, it avoids a situation in which accessing one tuple requires one block access. On the other hand, the nested-loop join method is advantageous when only a small fraction of tuples needs to be processed. Since it accesses the tuples in a random fashion, it is very likely to incur one block access per tuple. However, sorting is not needed in this method. Since only a small fraction of tuples is considered, the disadvantage of random access can be easily compensated for by the saving of the sorting cost. Clearly, the sort-merge join method should be used only when the benefit of sequential access exceeds the cost of sorting the tuples in the relations.

In OBE we choose to implement only the nested-loop join method. We believe that the nested-loop join (with proper usage of indexes) alone is adequate in memory-resident databases.[1] Basically, we argue that an index can be built in time equivalent to that for sorting a relation—both order of $n \log n$, where n is the number of tuples satisfying restriction predicates. A relation already sorted corresponds to an existing index. Once the relation is sorted or an index created, since there is no concept of blocking (i.e., many tuples in one disk block) [44] in a memory-resident database, most of the benefit of using the sort-merge join method can no longer be achieved. In fact, a relation can be viewed as a set of blocks each containing only one tuple. Thus we conclude that, in a memory-resident database, the nested-loop join method is as good as or better than the sort-merge join method in most cases.

Notice that many variations of hash join methods [10, 26] are by our definition nested-loop join methods, but use hash indexes. Such indexes are created on the fly and not permanently stored. One of the access structures we define in Section 5.1 considers creating an index on the fly for a specific query. However, we did not implement hash indexes because we decided to use the same index structure for both permanent and temporary indexes.

Incidentally, Lehman and Carey [26] compare new join algorithms based on the T-tree with other join methods. The T-tree is a compromise between the array index and the AVL tree [22] that achieves more flexibility in updates than the former and less storage space than the latter. However, their conclusion that the T-tree merge join is the best is based on the following assumptions:

(1) They consider only *unconditional* joins (i.e., without any restriction predicates). Typically, the nested-loop join is much preferred if there are strong restriction predicates so that only a small number of tuples need to be joined.
(2) They define the nested-loop join as the method that does not use (or create and use) any indexes. Clearly, the nested-loop join by this definition would be the worst method.
(3) They do not consider the nested-loop join method as we define in this paper (i.e., using the best index available, or creating one if beneficial).
(4) In the T-tree merge join cost, the cost of creating indexes is omitted assuming that the indexes already exist. In all other algorithms compared, the corresponding costs are included.

[1] Some authors [25] *define* the nested-loop join as a method that does not use (nor create and use) any indexes. Clearly, the nested-loop join by this definition would have an unacceptable performance.

In the nested-loop join method, there is a total order among relations to be joined. We define this total order as the *join sequence*. Also, we define the nested loop executing the join according to the join sequence as the *join loop*.

Using the nested-loop join method, the query processor evaluates a query as follows. It processes the nodes in the order of the join sequence. We define the node currently being processed as the *current node*. For the current node, it first evaluates the *column predicates* (the predicates that are specified in the column of the relation skeleton) and then evaluates the *cross-reference predicates* (those in a condition box or the column predicates of the previous nodes, the evaluation of which were deferred because they were not bound). However, not all the cross-reference predicates need to be evaluated; only those containing an example element that gets bound at the current node are considered for evaluation since these are the only ones that can become bound at the current node. We discuss more refinement of this concept in Section 5.

Example 3. Let us evaluate the query in Figure 3 according to the sequence ⟨Node 3, Node 4, Node 1, Node 2⟩. The query processor first retrieves a tuple from Node 3 that satisfies the restriction predicate Col3 = 150. Then it tries to evaluate the join predicate Col1 = _X + _Z. Having found the predicate is unbound, however, it tries to bind the value of the example elements, which also fails because the predicate is not a bindable instance of either _X or _Z. (This paradigm is improved by the query optimizer. We discuss the improvement in Section 5.3.) Thus evaluation of this predicate is deferred by making it a cross-reference predicate for subsequent nodes in the join sequence. Next, the query processor retrieves a tuple from Node 4. Node 4 has two join predicates that are unbound but bindable. Thus the query processor binds the example elements _X and _Y with the values of Col1 and Col2 in the tuple retrieved. The cross-reference predicate Node 3.Col1 = _X + _Z still cannot be evaluated because _Z is not bound. However, the cross-reference predicate _X > 5 in the condition box can be evaluated here. Next the query processor retrieves a tuple from Node 1 that satisfies the bound join predicate Col1 = _X. It subsequently binds the example element _Z with the value of Col3 in this tuple. Note that Col3 = _Z is a bindable instance of _Z. Since both _X and _Z are bound by now, the query processor successfully evaluates the cross-reference predicate Node 3.Col1 = _X + _Z. If the predicate is false, the join loop backtracks to Node 4, retrieves a new tuple in Node 4, and enters Node 1 again. If the predicate is true, the join loop continues to Node 2. At Node 2, the query processor checks whether the value of _Y, which has been bound in Node 4, is in any tuple of Node 2 (i.e., evaluates the row negation). If not, the entire evaluation is successful and the value of projection columns (which are specified with P.; in this case, they are _X, _Y, and _Z + 5) is calculated and returned as a result. This process is repeated until all tuples satisfying the query are constructed.

For operations such as insertions, deletions, and updates, the query processor employs a *deferred update* policy [12, 35]. It first collects the tuples (together with tuple identifiers (TIDs)) to be inserted, deleted, or updated in a temporary relation and then updates the base relation in batch with the tuples in the

temporary relation. This policy makes the update part in an operation independent of the query part. Since the update part is common in all access strategies, optimization involves only the query part. As a result, the optimizer treats all the operations on the database as (read-only) queries.

Aggregations are also evaluated at the end; thus they are independent of optimization. Many DBMSs employ the same strategy. As an exception, System R keeps track of an interesting ordering [32] so that, in the best case, it can save a cost of sorting the result to evaluate the aggregation. Although it may be beneficial at times, it adds significant complexity to the optimization process. In our approach, we do not consider interesting ordering.

We define a *hypothetical result relation* for node N_j in the join sequence $\langle N_1, N_2, \ldots, N_j, \ldots, N_d \rangle$ as the result of joining nodes in the *partial join sequence* $\langle N_1, N_2, \ldots, N_j \rangle$. As we discuss in Section 4, a result relation is never materialized until the completion of the query. We define the *result cardinality* for a node as the cardinality of the hypothetical result relation for that node. Finally, we define *access structures* as the methods of accessing data that the optimizer can select in the process of optimization.

4. STORAGE STRUCTURES

In this section we briefly introduce the storage structures implemented by the low-level system component HRMS. More details on the storage structures can be found in Ammann et al. [1]. HRMS provides the data storage organization and primitive access functions. We introduce here only those features that are relevant for query optimization, that is, the representations of relations and indexes.

4.1 Relations

A relation is represented as a doubly linked list of tuples. Each tuple is an array of pointers to column values. Each column value is stored as a variable-length string with two bytes of information on its length. HRMS tries to store a unique column value only once; thus tuples having the same column value point to the same string. Unique values of a column can be easily detected if an index is maintained for that column.

4.2 Indexes

HRMS provides two types of indexes: the single-column index and the multiple-column index. A single-column index is an index defined for a column of a relation and can be permanently stored in the database. A multiple-column index is an index defined for a list of columns, but cannot be permanently stored in the database.

A multiple-column index is created as follows: first, the tuples of the relation that satisfy the restriction predicates are selected; then, an index is constructed for the list of columns having bound equality join predicates. The lists of restriction and join columns are designated by the optimizer. Since a multiple-column index is always specific to a query, it is not relevant to store the index permanently in the database. The multiple-column index has an advantage of

reducing the size of the relation because it selects only those tuples that satisfy restriction predicates. Due to this reduction both the cost of creating the index and the cost of accessing tuples using this index can be reduced.

Example 4. Consider constructing a multiple-column index for Node 3 in Figure 3. We assume that the join predicate Col1 = _X + _Z is bound by the time Node 3 becomes the current node. We create a multiple-column index by first finding the tuples satisfying the restriction predicate Col3 = 150 and then creating an index on Col1 (having a bound equality join predicate) only for these tuples. In general, there may be more than one bound equality join predicate; in such cases, we build an index on the list of all the join columns having bound equality join predicates. (Hence we have the name "multiple-column index.") Suppose that Node 3 has 10,000 tuples and that 100 tuples satisfy the restriction predicate Col3 = 150. Since the index is created only for these 100 tuples, the costs of creating and accessing the index are reduced. In addition, we have an added advantage of indexing based on more than one column simultaneously.

An index (of either type) is implemented as an array of *tuple identifiers* (*TIDs*) that are pointers to tuples. Accessing an index uses a binary search. To avoid arithmetic computation in the binary search as was used by Lehman and Carey [25], we use the shift instruction instead of the division operator. Lehman and Carey [25] point out that arithmetic operations are the major source of the cost in searching the array index. This scheme contrasts with the conventional method of implementing an index as a binary search tree such as the AVL tree [22]. Since the scheme does not store the tree structure explicitly, it requires less memory space for storing indexes. In our implementation it showed a six-to-one reduction in memory space as compared with the AVL tree. The scheme achieves further reduction in memory space by storing only pointers to TIDs in the index; the key values can be easily found (by two pointer references) from the tuples they point to. In comparison, in conventional disk-based database systems, the index storage cost is a major problem because original key values must be stored in the index.

Updating an index is straightforward. For example, when an index entry is inserted, the upper half of the index is block-copied to a new memory location; a new entry is inserted; and then, the lower half of the index is block-copied next to the new entry. For block-copying we use the instruction MOVE LONG (C) (meaning move a long character string) defined in the IBM 370 architecture. This instruction is a very efficient operation in IBM mainframes. Experiments show that, in a 3081 processor, copying 1 Mbyte of memory using this instruction takes less than 0.1 second. For this reason, updating indexes is not a major problem in our system. We briefly present the performance of update operations in Section 6.2.

5. QUERY OPTIMIZATION

The query optimizer determines the optimal join sequence and specific access structures to be used in processing individual nodes. The results the optimizer produces are recorded in a data structure called the *access plan*. The information in these data structures is subsequently interpreted by the query processor to

evaluate the query. In this section we present the query optimizer's algorithm, access structures, cost model, and data structures.

Let us note that query optimization in OBE is quite simplified due to the nature of the language. Queries written in OBE are in the *canonical form* as defined by Kim [20]. In particular, queries have no substructures such as *nested queries* in SQL. The advantages of the canonical representation are well explained by Kim [20]. (Queries having aggregate operations in a condition box are exceptions; further details on these are beyond the scope of this paper.)

In addition, the concept of example elements provides more flexibility in optimization compared with conventional query languages where the columns participating in a join are explicitly named. For example, consider the join predicate: Column A = Column B AND Column B = Column C. In conventional languages, the optimizer has to derive the fact Column A = Column C. (In many systems, this is not done.) In OBE, since all three columns are represented by the same example element, the information is readily available to the optimizer.

5.1 Access Structures

The optimizer defines the following access structures:

(1) Relation Scan;
(2) Single-Column Index (existing);
(3) Single-Column Index (created);
(4) Multiple-Column Index.

Relation Scan specifies that each tuple in the relation be accessed sequentially. Single-Column Index (existing) specifies that tuples be accessed associatively through an existing single-column index. Single-Column Index (created) also specifies access through a single-column index, but requires that the index be created for each query and be dropped at the completion of the query. Lastly, Multiple-Column Index specifies that tuples be accessed through a multiple-column index.

5.2 Cost Model

We define the minimum query processing cost (response time) as the criterion for optimality. Since the cost of processing a query must be determined before actually executing it, we need a proper model for estimating this cost. Since we assume memory residency of data, we consider only the CPU computation cost.

In many conventional database systems, the cost model has been constructed by simply counting the number of I/O accesses for processing a query. Modelling the CPU cost, however, is not as straightforward for the following reasons. (1) There are many parameters affecting the CPU computation cost; examples are the design of the software, the architecture of the hardware, and even the programming styles of the programmers. Moreover, as the system evolves, these parameters are subject to change. (2) Even if the parameters do not change, analyzing the overall computation cost for the entire program code is next to impossible. For example, the most straightforward method would be to add all the machine cycles of the machine instructions. In a large system program, however, counting machine cycles would be impossible.

As a solution to this problem we propose an approach using both experimental and analytic methods. First, we identify the system's *bottlenecks*, that is, those sections of the program code that take most of the processing time. Then we model the CPU computation cost based on only these system bottlenecks. Next, we improve the bottlenecks to the extent that no tangible enhancement can be made. This step prevents the cost model from drifting frequently as a result of the changes in the program. Since usually there are only a small number of bottlenecks, they can be thoroughly examined. In fact, our experience shows that most bottlenecks include very small numbers of machine instructions. The next step is to find, by experiments, relative weights of different bottlenecks and to determine their *unit costs*. Finally, we develop comprehensive cost formulas based on these unit costs.

In OBE the bottlenecks have been identified with the aid of PLEA8 execution analyzer [29]. The execution analyzer provides the statistics on the percentage of the time spent in individual procedures. Once a procedure is identified, it is relatively straightforward to locate the section of the code that causes the bottleneck.

We have measured the following unit costs:

(1) the cost of evaluating the expressions involved in predicates (unit cost = C_1);
(2) the cost of comparison operations needed to finally determine the outcome of predicates (unit cost = C_4);
(3) the cost of retrieving a tuple from a (memory-resident) relation (unit cost = C_5);
(4) the cost of unit operation in creating an index (unit cost = C_2; there are $n \log_2 n$ unit operations in creating an index, where n is the number of tuples in the relation);
(5) the cost of unit operation in the sorting needed to prepare a multiple-column index (unit cost = C_3).

The five parameters, C_1, C_2, C_3, C_4, C_5 have been determined by experiments using the execution analyzer and have been stable for a long period of program development.

An interesting result of our modeling effort has been to find that the evaluation of the predicates (especially, expressions in the predicates) is the costliest operation in our system. Among the five parameters, C_2, C_3, C_4, and C_5 are of the same order, and C_1 is approximately ten times as large as the others. Basically, C_2, C_4, or C_5 represents a comparison operation. In many conventional database systems the major cost comes from tuple retrieval due to the I/O cost involved and the CPU cost for copying and manipulating the data structures for the tuples. In OBE, however, tuple retrieval (C_3) is as simple as following a pointer. Since all the data are in main memory, we only have to locate the tuple without having to copy or manipulate the tuple data structure. On the other hand, predicate evaluation is much costlier because it requires traversing the expression tree that has a more general form in order to accommodate various forms of expressions. Although the complexity of expressions varies, we encounter the simplest forms (such as _X or 5 for column predicates and _X > 5 for condition boxes) most of the time. The measurement is based on these simple expressions.

The cost formulas are composed using the statistical parameters maintained by HRMS. There are two types of parameters. The *relation cardinality* is the number of tuples in a relation. The *column cardinality* is the number of unique values in a column. The cost formulas are constructed in three steps. First, using the statistical parameters, the selectivities of the predicates are calculated. Next the cardinalities of the hypothetical result relations are calculated. Based on the selectivities of the predicates and result cardinalities thus obtained, the total access cost for processing a query is finally calculated.

Typically, two assumptions are used in many relational database implementations: (1) uniform distribution of data and (2) independence among columns. The uniform distribution assumption states that the values in a column occur with an equal probability. The independence assumption states that a value in a column is in no way related to values in another column, that is, they are not correlated. Although in reality the distribution of data will not conform to these assumptions, we adopt them for practical implementation considerations. Recently, some papers discussed potential deviation of the cost caused by these assumptions. Christodoulakis [8] points out that the uniform distribution represents the worst case in single-relation case (i.e., simple selection). Vander Zanden et al. [37] presents a mechanism to deal with correlation with the clustering column. Nevertheless, the proposed techniques require a significant amount of storage space and overhead to store and maintain the distributions of the data. We believe that the two assumptions, although not absolutely accurate, are sufficiently effective in eliminating any solutions that significantly deviate from the true optimal solution. Our experience with the OBE optimizer supports this belief.

Before concluding this subsection, let us note an interesting consideration in constructing cost formulas. Counting the number of predicates that are to be evaluated is an important part of the cost model. We use the following function for this purpose. Suppose $number_p$ is the number of bound predicates considered for a node. Then, the number of predicates to be evaluated for a tuple in that node is given by

$$f = \begin{cases} 0 & \text{if } number_p = 0 \\ 1 & \text{if } number_p = 1 \\ 1.5 & \text{if } number_p \geq 2. \end{cases}$$

The function f is derived as follows. In OBE predicates specified in columns of a node or in condition boxes are implicitly ANDed. Hence, if the first predicate is false, the second predicate will not be evaluated. Accordingly, the probability that the second predicate is evaluated is the same as the selectivity of the first predicate. Likewise, the probability of the third predicate is evaluated is the product of the selectivities of the first and the second, and so forth. We assume that, in most cases, the selectivity of a predicate is less than one-third. Then, the number of predicates to be evaluated will be bound by $1 + \frac{1}{3} + \frac{1}{9} + \cdots = 1.5$. Estimating the number of predicates based on the selectivities of individual predicates will complicate the optimization process substantially. By using the function f, we can obtain a reasonable estimate without overhead in most practical cases.

For illustrative purposes, we present example cost formulas for the access structure Relation Scan in Appendix A. A complete set of cost formulas can be found in Whang [39].

5.3 Optimization Strategies

As we discussed in Section 5.2, predicate evaluation (especially, expression evaluation) is the costliest operation in processing a query. Thus, query optimization is geared to minimizing the number of predicate evaluations. For this purpose, the optimizer employs the following strategies:

(1) binding an example element as early as possible;
(2) evaluating a predicate as early as possible;
(3) avoiding useless evaluation of a predicate;
(4) avoiding unnecessary evaluation of an expression.

An example element appearing more than once in a query may have multiple places in the join sequence where it can be bound. In other words, it may appear in multiple bindable instances (predicates). For example, the example element _X in Figure 3 may be bound at either Node 1 or Node 4. Given a join sequence, however, an example element should be bound as early as possible for better performance. Early binding of an example element leads to early binding and evaluation of the predicates containing the example element. Early evaluation of predicates, in turn, results in a smaller cost by reducing the number of tuples to be considered in the rest of the join sequence. According to strategies 1 and 2, the optimizer finds the earliest point in the join sequence at which an example element can be bound. For instance, in Figure 3, the optimizer specifies that the example element _X be bound at Node 1 or Node 4, whichever comes earlier in the join sequence. We call such specification by the optimizer *static binding.* Static binding information is subsequently used by the query processor to bind the example elements at run time. We call this process *dynamic binding.* Let us note that these strategies are not heuristics since they always reduce the processing cost.

A *useless evaluation* of a predicate is defined as an evaluation of an unbound (join) predicate. A useless evaluation will eventually be aborted when an unbound example element is encountered. We note that predicate evaluation is the costliest operation. Useless evaluation of a predicate is equally costly because the data structure for the expression has to be interpreted and the predicate partially evaluated until an unbound example element is found. According to strategy 3, the optimizer detects an unbound predicate at optimization time and sets an indicator. The query processor can avoid useless evaluation by simply looking up this indicator. In Example 3, the evaluation of the join predicate Node 3.Col1 = _X + _Z at Node 3 is a useless evaluation, and so is the evaluation of the same predicate as a cross-reference predicate at Node 4.

We use two types of indicators: the column-evaluation flag and the cross-reference-evaluation flag. The *column-evaluation flag* for a node is an array; each entry is associated with a column of the node. An entry is set to 1 if the corresponding column predicate can be evaluated successfully at the node, that is, if it is bound by the time the node becomes the current node. For instance, in

Example 5, column-evaluation flag for Node 3 is ⟨0, 0, 1⟩. The flag for Node 4 is ⟨0, 0, 0⟩ since both join predicates are not bound when Node 4 becomes the current node.

The optimizer sets the column-evaluation flag of a node to 1 if the following conditions are met:

(1) The predicate is bound (a restriction or bound join predicate) at the node (i.e., when the node is the current node).
(2) The predicate is not used in indexing (with either a single-column index or a multiple-column index). This condition applies because the predicate is already evaluated through indexing.

The cross-reference-evaluation flag is an array of ⟨*example element*, join predicate⟩ pair. Each entry in the flag is associated with an example element and a join predicate that contains it. An entry is set to 1 if the three conditions are met:

(1) The *join predicate* is a cross-reference predicate for the node at which the *example element* gets bound.
(2) The *join predicate* becomes bound as the *example element* gets bound.
(3) The *join predicate* is not used to bind the *example element.*

For instance, in Example 3, at Node 4, the cross-reference predicate Node 3.Col1 = _X + _Z (predicate[Node 3, Col1]) does not become bound when example element _X gets bound. Hence, cross-reference-evaluation flag[_X, predicate[Node 3, Col1]] is 0. At Node 1, however, predicate[Node 3, Col1] becomes bound when _Z gets bound. Thus cross-reference-evaluation flag[_Z, predicate[Node 3, Col1]] is 1. Note that predicate[Node 1, Col1] is "not" a cross-reference predicate for Node 4 because Node 4 comes before Node 1 in the join sequence. Since the first condition is not satisfied, cross-reference-evaluation flag[_X, predicate[Node 1, Col1]] is 0 even if the predicate becomes bound when the example element _X gets bound at Node 4. Instead, this predicate is evaluated as a column predicate at Node 1 since it is bound by the time Node 1 becomes the current node.

An interesting special case occurs when there are multiple binding instances of an example element in different columns of the same node. (Imagine there is an example element _X in Node 4, Col3.) In this case, if one is used for binding the example element, the other is treated as a cross-reference predicate.

The query processor evaluates column predicates of a node whenever the node becomes current, but considers only those with their column-evaluation flag entries set to 1. Similarly, it evaluates the cross-reference predicates whenever an example element gets bound but considers only those for which the corresponding ⟨example element, join predicate⟩ entry is set to 1. This way, the query processor avoids useless evaluation completely.

Column predicates are evaluated in two steps. First, the expression is evaluated, and then the result is compared with the value retrieved from the tuple. For instance, in Figure 3, _X + _Z is an expression that is evaluated and compared with the value of Col1. We regard even _X or 150 as an expression because it is stored in the expression structure in the same general form as more complex

expressions. As we discussed in Section 5.2, expression evaluation is the costliest operation, which should be avoided as much as possible. We note that there is no need for reevaluating the expression unless its value changes.

There are two cases where reevaluation is not needed. First, for a restriction predicate, the expression (such as 150 in Node 3.Col3 in Figure 3) needs to be evaluated only once at the initial stage of processing the query. The result is stored in a program variable. Then, at run time, predicate evaluation simply becomes a comparison between the value of the column in the tuple and the one stored in the variable.

Second, the expression for a bound join predicate (except for a cross-reference predicate) needs to be evaluated only once when the node containing the predicate is entered in the join loop (i.e., is made the current node). Once the node is entered, the value of the expression stays constant until the join loop backtracks to the previous node in the join sequence after examining all the eligible tuples in the current node. Thus, the expression does not need reevaluation. For example, the expression in the predicate Node 1.Col1 = _X in Figure 3 is evaluated only when Node 1 is entered. It stays constant until all tuples of Node 1 having the specific value of _X are processed. Cross-reference predicates are exceptions because their example elements are bound to different values every time a new tuple is retrieved from the current node.

According to strategy 4, the optimizer identifies these two cases and set indicators, so that the query processor can avoid unnecessary evaluation by simply looking up the indicators. We use two types of indicators: the comparison-restriction-predicate flag and the comparison-bound-join-predicate flag. Both flags are arrays, each entry being associated with a column of a node. An entry in the comparison-restriction-predicate flag is set to 1 for any column having a restriction predicate. An entry in the comparison-bound-join-predicate flag is set to 1 for any column having a bound join predicate. Columns used for indexing are excepted since the predicates are already evaluated through indexing.

The results of optimization are recorded in the *access plan.* The access plan contains the following information: (1) the optimal join sequence, (2) for each node, the access structure chosen for that node, (3) static binding for each example element, and (4) four predicate-evaluation indicators (the column-evaluation flag, cross-reference-evaluation flag, comparison-restriction-predicate flag, and comparison-bound-join-predicate flag).

Example 5. Figure 4 shows an example of the optimization process. We assume that, initially, the database contains no index permanently defined. As we discuss in Section 5.4, the optimizer employs a systematic search technique called the *branch-and-bound algorithm* [15]. In this example, however, we provide an "intuitive" explanation on the optimality of the solution in Figure 4. Both Node 2 and Node 3 have very selective restriction predicates involving the column DEPT. Since selective predicates reduce the number of tuples to be considered in the rest of the join sequence, it is very likely that the nodes they belong to are chosen to be the first in the join sequence. Here Node 2 is chosen first. Then example elements _X and _Z can be bound in Node 2 since the predicates are bindable instances. The next node in the join sequence is chosen to be Node 1. Since example element _X has been bound, Node 1 can be accessed efficiently

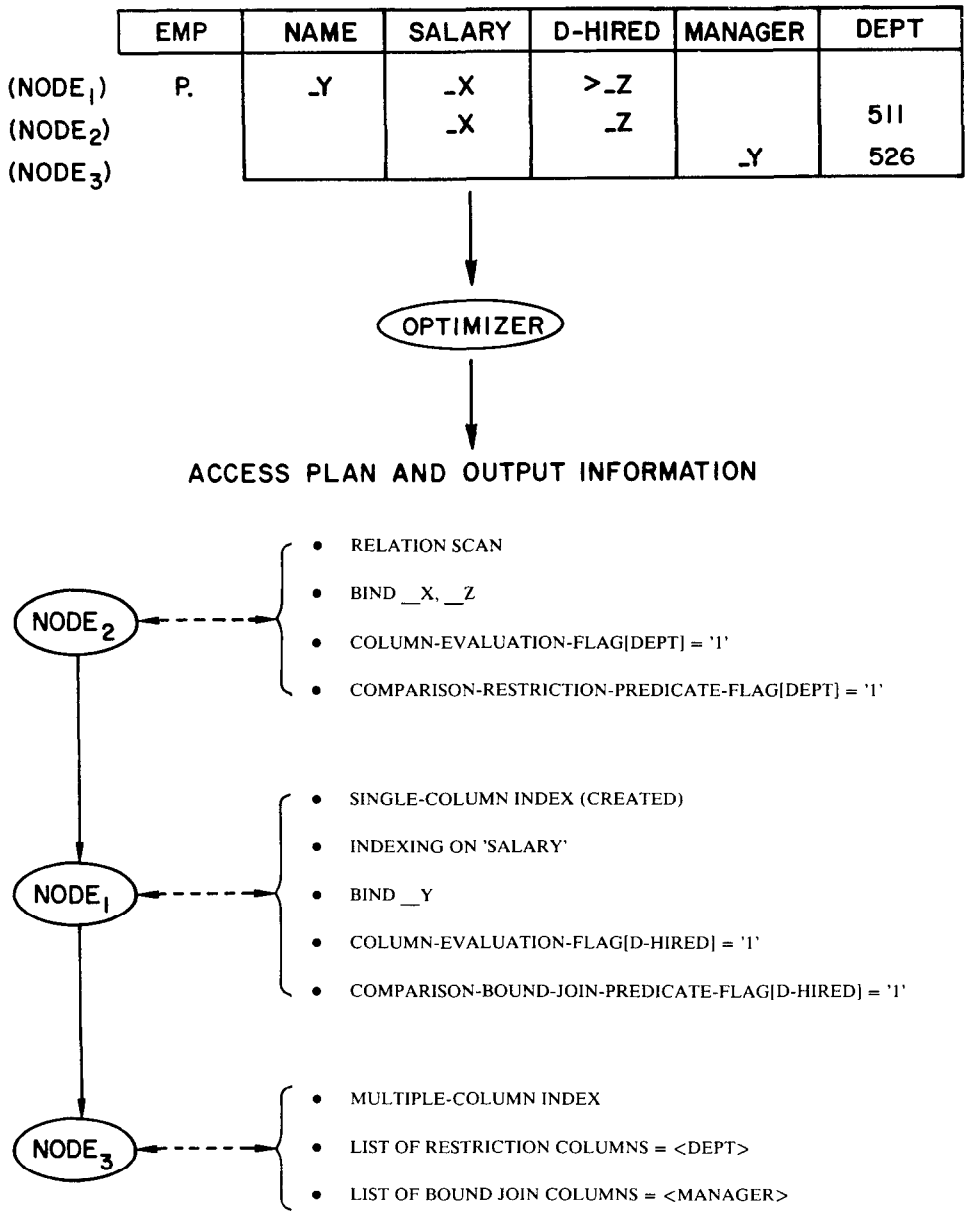

Fig. 4. An example of optimization.

through an index for the column SALARY. Similarly, since example element _Z has been bound, the predicate D-HIRED > _Z can be evaluated. Example element _Y is bound here. The last is Node 3. Since it is the last node in the join sequence, tuples in this node are likely to be accessed repeatedly. Hence, it would be beneficial to reduce the size of the relation by selecting in advance only those tuples satisfying restriction predicates. Multiple-Column Index is chosen as the

access structure to serve this purpose. The list of restriction columns is ⟨DEPT⟩, and the list of bound join columns is ⟨MANAGER⟩. The optimizer's decisions on predicate-evaluation flags and predicate-comparison flags are self-explanatory and are not further discussed.

5.4 Searching Algorithm

The optimizer finds the optimal solution by exploring the search tree using a branch-and-bound algorithm. At each node in the search tree, the optimizer performs the following tasks:

(1) According to the optimization strategies 1 and 2, it statically binds all example elements that can be bound.

(2) It chooses the most efficient access structure. For this purpose, single-column indexes for all the columns having restriction or bound join predicates are considered. If an index is not available, creating and using one is considered. The relation scan and the multiple-column index are also compared.

At the same time, it calculates the total cost of processing the node considering optimization strategies 3 and 4. The cost is multiplied by the result cardinality from the previous node in the join sequence, which represents the number of entering the node in the join loop.

Given a cost model, the branch-and-bound algorithm always produces the optimal solution. Although the branch-and-bound algorithm is functionally equivalent to an exhaustive search, it substantially reduces the search time by means of *pruning*. The pruning of a subtree occurs whenever there is a *clue* that the optimal solution cannot be found in the subtree. The clue can be obtained by comparing a *lower bound* of the cost associated with the subtree with the *global upper bound*, which is maintained as the minimum cost of alternative solutions that have been examined. If the lower bound exceeds or equals to the global upper bound, the subtree is pruned.

A branch-and-bound algorithm can be characterized by three rules: (1) a rule for finding a lower bound for a subtree, (2) a branching rule, and (3) a rule for resolving a tie among candidate subtrees to be explored. For the query optimizer we use the following rules. First, we define the accumulated access cost for the current node as the lower bound for the subtrees to be explored. The *accumulated access cost* of a node is the cost of obtaining the hypothetical result relation for the node according to the join sequence. Second, we employ the newest bound branching rule, which selects the most recently created (or to be created) subtrees as the candidate subtrees. Third, we use a fixed priority scheme to resolve a tie among these candidate subtrees. Let us note that, due to the definition of the lower bound, the candidate subtrees always form a tie.

We define the order of priority among nodes as follows:

(1) An ordinary node with a bound equality predicate.

(2) A negated node with all its predicates bound.

(3) An ordinary node with a bound inequality predicate (i.e., a range or not-equal predicate).

(4) An ordinary node without any bound predicate.

A negated node having an unbound join predicate should never be given a priority because it can be evaluated only when all the predicates are bound.

This priority scheme seems to work reasonably well. In retrospect, however, the authors believe that the priority should not be based on the type of the node; instead, it should be based on combined selectivity of the predicates in the node. In fact, if a range predicate has a better selectivity than an equality predicate, the former should be given a higher priority.

The accumulated access cost for a node is obtained by adding the cost of processing the node to the accumulated access cost for the previous node in the join sequence. The cost of processing a node is calculated as the cost of processing the tuples of the node through the most efficient access structure. The result cardinality for the previous node in the join sequence is implicitly multiplied since it represents the frequency of entering the current node in the join loop.

6. PERFORMANCE

The OBE database manager has been designed with the assumption that data reside in main memory. Naturally, one important question is to what extent this assumption would be satisfied in a real environment. Obviously, a steady tendency toward cheaper memory hardware is encouraging. Yet this will not solve all the problems because the cheaper the memory becomes, the larger the data requirement becomes. Thus we need other means of satisfying the memory-residency assumption.

In OBE the memory-residency assumption is approximated by virtual memory and the working-set scheduling algorithm. We argue that the operating system's scheduling algorithm has a vital effect on the memory-residency assumption. In particular, pure demand paging is not suitable for this assumption because it would suffer from thrashing if the total memory requirement from many users far exceeds the size of the physical memory of the system. However, when coupled with the working-set algorithm, virtual memory provides an excellent approximation of real memory.

6.1 Operating System Environment

In this subsection we describe a very simplified version of VM/370's scheduling algorithm [17] to investigate its effect on the memory-residency assumption. The scheduling algorithm uses two types of time units: *time slice* and *dispatch time slice*. For convenience, let us call them *long time slice* and *short time slice*. In addition, there are two types of queues for virtual machines: *dispatch list* and *eligible list*. The short time slice is the unit time for allocating CPU among members in the dispatch list. A long time slice is a fixed multiple of short time slices, during which a virtual machine is entitled to stay in the dispatch list. If there are other virtual machines with higher priorities when the long time slice expires, a virtual machine may be relocated to the eligible list waiting for another long time slice to be assigned to it.

The scheduler puts virtual machines in the dispatch list only to the extent that the total memory requirement of the virtual machines does not exceed the size of real memory. The memory requirement of a virtual machine (*working set*) is first estimated as the average number of memory-resident pages for that virtual

machine while in the dispatch list. The number thus obtained is adjusted according to some formula that provides a feedback mechanism to stabilize the performance toward the system's global goal for paging activities. Once the set of virtual machines on the dispatch list is determined, paging is controlled on a demand basis. This way, ideally, if access patterns of the virtual machines are constant, the virtual machines must get sufficient real memory with which to work.

Let us summarize the ramifications of the scheduling algorithm.

(1) As long as a query is evaluated within one long time slice, there will be no additional I/O's except for the initial loading of data (one access for each page).

(2) Even when a query spans multiple long time slices, provided that a long time slice is long enough to dominate the cost of initial loading, the I/O time will be negligible compared with the CPU time.

(3) There will be no significant thrashing because the memory requirement of a virtual machine is satisfied while the machine is on the dispatch list.

(4) The system's feedback mechanism, imbedded in the estimation of the working set, stabilizes the overall paging activity (e.g., 15 percent).

We have informally discussed the advantages of the working-set scheduling algorithm, the detailed analysis of which is beyond the scope of this paper. They support our claim that virtual memory, in conjunction with the working-set algorithm, serves as a reasonable approximation to real memory in practical environments. Here we address the problem of the total size of the physical memory (of the entire system, not per user) being less than the size of the data of a single user. Experiments indicate that performance is gradually degraded as the size of data crosses over the physical memory size. This shows that as the data size/physical memory size ratio doubles, the degradation approximately quadruples [4]. Note that, as long as the system's total memory is greater than the data requirement of a *single* user, there is no significant degradation due to thrashing. Of course, the response time would be increased (approximately linearly) as the number of users increases. In a system adopting pure demand paging, the degradation would be far more significant in a multiuser environment due to thrashing.

6.2 Test Results

Extensive tests were made on the performance of OBE; some of the results are presented here. These tests were made on IBM's 3090 processor with the total system's physical memory of 32 Mbytes. They were also tested with SQL/DS, and those results are presented in Table II. For comparison, we also provide a conservative estimation of the performance of a hypothetical I/O bound database system, which will result in an environment adopting demand paging when the system's total memory does not satisfy the total data requirement of all the users.

Table I shows the results of the tests. The tests were made under the following conditions. The machine was lightly loaded. (The tests were made at lunch time, when a large virtual machine size is allowed.) Each relation in the queries contained 10,000 tuples occupying 1.91 Mbytes of storage space. Each relation had indexes on all columns. (The indexes occupied approximately 12 percent of

Table I. Performance Results for OBE on an IBM 3090 Processor

Degree of join	Data volume (Mbytes)	Virtual CPU time (seconds)	Elapsed time (seconds)
1	1.91	1.5	3.7
2	3.82	1.7	5.3
3	5.73	2.0	5.7
4	7.64	11.7	32.0
5	9.56	13.7	42.0

the total space.) The queries were designed as joins with one restriction (range) predicate. The column values of the relations were generated by a random number generator. The intermediate results of the queries ranged from 3,000 to 15,000 tuples before duplicate elimination. (Duplicate elimination is done only at the completion of the query.) Each query was run three times, and the average elapsed time was obtained. Since the virtual CPU time stayed fairly stable over different executions of the same query, only one representative value is shown in Table I.

In Table I, the *degree of join* is the number of relations joined in a query, the *virtual CPU time* is the total CPU time the virtual machine (a user process) consumed in processing a query, and the *elapsed time* is the real time measured from the start of a query to its end.

We tested the same queries with the same data in a disk-resident database management system, SQL/DS, running on the same 3090 processor. Indexes were created only on the join columns and the restriction columns for space considerations. Indexes on the other columns would not have affected the test results since they were not used in processing the test queries. The results of the tests are presented in Table II together with the performance of a hypothetical I/O bound system using the nested-loop join. In this system, we assumed two pages are accessed to retrieve one tuple (one for the index and one for the data) [31, 44]. An average seek and rotation time of 30 ms was used. Since intermediate results were always more than 3,000, we used this figure as a conservative estimate of tuples retrieved from each node. Thus the time to evaluate a query in this system was estimated as 3,000 × 2 × degree of join × 30 ms.

From the results summarized in Tables I and II, we conclude that the performance of OBE justifies memory residency of data. The performance estimation of the I/O bound system indicates that a demand paging should not be used in a memory-resident database system. Formal benchmarking has been done independently of the tests presented in this paper, and the results can be found in Bitton et al. [3] and Bitton and Turbyfill [4]. We briefly summarize the results here.

(1) OBE does very well with selections and joins, as indicated in Tables I and II.

(2) OBE is slow in update, insertion, and deletion. For update of nonkey columns and deletion, maintenance of indexes causes the major cost. However, this problem does not come from our index update algorithm; instead, it comes

Table II. Performance Results for SQL/DS and a Hypothetical I/O Bound System

Degree of join	Elapsed time (seconds) SQL/DS	Elapsed time (seconds) I/O Bound system
1	10	180
2	39	360
3	58	540
4	155[a]	720
5	482[a]	900

[a]The figures were extrapolated using the SQL/DS optimizer's estimation. The queries were not actually run due to space problems.

from our decision to provide indexes for all columns by default, although there is a provision for selective indexing. Experiments indicate that the cost of updating all indexes is comparable to (somewhat slower than) that of updating one index in conventional disk-based DBMSs. Insertion and update of key columns are quite slow; the reason is duplicate elimination by default. Note that many other DBMSs, such as SQL/DS or INGRES do not provide this feature; instead, they allow duplicate records to be stored. Also note that duplicate elimination in a permanent relation is different than that in a temporary relation, for which we use an efficient hash-based method.

(3) OBE is slow for certain types of projections in which our efficient, hash-based duplicate elimination is not used. Nevertheless, this problem is not inherently related to the memory-residency assumption. We believe that this problem can be fixed by more careful design of data structures for these operations.

(4) The test data we provided (Table I) are based on the fact that the relations are already in main memory. If they are not, they have to be brought into main memory at the first reference. Nevertheless, reading a relation into main memory is very fast in our system because we store a relation as *one* record in a file. The file system has a built-in mechanism to try to store a contiguous virtual file address space in contiguous physical disk pages as much as possible. Typically, the system takes 0.67 second to read in 1 Mbyte of data.

6.3 Optimization Cost

While the purpose of the optimizer is to minimize the cost for processing queries, optimization itself accompanies computation cost. Clearly, efficiency of optimization is essential for good overall performance.

We measured the optimization cost in terms of virtual CPU time using the same set of tests as in Table I. The results are summarized in Table III and indicate that the optimization cost is not significant even for the join queries involving as many as five nodes. Join queries of degree five or higher are fairly complex in practice and must be issued very rarely. For this reason, we conclude that the optimization cost can be ignored in our system for most practical situations.

Table III. The Cost of Optimization

Degree of join	Virtual CPU time (milliseconds)
1	3
2	7
3	12
4	22
5	60

6.4 Limitations

OBE's database manager was designed primarily for small databases that are suitable for office application environments. In Section 6.2 we tested the system to its limit by using a volume of data that was close to the system's maximum capability. The conventional IBM 370 architecture without extended addressing provides only 24 bits for addresses—equivalently, 16 Mbytes of address space. The results of the tests indicate that the system performs well with a fairly large volume of data. Nevertheless, we note that, in our system, the amount of data that the system can accommodate is inherently limited by the maximum size of virtual memory. We also note that the performance degrades more than linearly if the memory requirement of a *single* user exceeds the system's physical memory size. This is caused by increased paging activity, which is beyond the control of the operating system's working-set scheduling algorithm.

7. SUMMARY AND CONCLUSIONS

We have presented the design of a query optimizer for a memory-resident database system and have implemented the database system in the context of the OBE project. A single-user version of the system is fully operational at the time of this writing.

The major contribution of this paper is to show that the memory-residency of data is a viable idea in realistic environments. We have proved this claim through a concrete implementation and performance measurement. The results in Table I show that OBE can process a large volume of data with excellent performance concurrently with other transactions in a time-shared system.

We have emphasized that the techniques for query optimization in memory-resident database systems differ significantly from those in conventional disk-based database systems. In particular, we have addressed the following aspects of query optimization:

(1) a CPU-intensive cost model,
(2) optimization strategies,
(3) join algorithms and access structures,
(4) the scheduling algorithm of the operating system.

We summarize our findings on these issues below.

We have presented a new approach to developing a cost model suitable for memory-resident database systems. In such systems CPU computation constitutes the major part of the processing cost. Our technique is based on the system's bottlenecks and their *unit costs*. The complete cost formulas are constructed analytically using these unit costs. We have found an interesting result that the dominant cost in memory-resident database systems is incurred by evaluation of predicates. Thus, in its simplest form, the cost model should reflect the number of predicate evaluations necessary to process a query. This contrasts with the case of disk-based database systems, in which the cost of I/O accesses dominates. A critical aspect of a CPU-intensive cost model is stability. We have proposed a technique for achieving stability of such a model.

We have formalized four optimization strategies for processing queries in memory-resident database systems. They are (1) binding an example element as early as possible, (2) evaluating a predicate as early as possible, (3) avoiding useless evaluation of a predicate, and (4) avoiding unnecessary evaluation of an expression. These strategies are geared to minimizing predicate evaluations, which are the costliest operations. We have presented detailed data structures to implement such strategies. Based on these strategies, the optimizer uses a branch-and-bound algorithm with a fixed priority to search for the optimal access plan.

The four optimization strategies are closely tied to the notion of example-element binding (a simpler version of unification in the predicate logic), which is a unique concept in domain relational calculus languages such as QBE. In Section 3 we presented a complete technique for processing domain relational calculus queries based on example-element binding. The technique differs significantly from those for tuple relational calculus or relational algebra systems, where only the notion of the *column value* exists. Example-element binding is not done in such systems. We have introduced the notion of a cross-reference predicate, whose evaluation is delayed until it becomes evaluable. This notion is an enhancement over many Prolog implementations, in which premature evaluation of such a predicate is considered an error. Query processing in domain relational calculus systems has not been well addressed in the literature. Our approach should provide new insight into these problems.

We have argued that the nested-loop join method is a prevalent technique in memory-resident database systems. Specifically, the benefit of the sort-merge join method disappears in such systems because there is no concept of blocking when the data reside in main memory.

We have presented a simple index data structure suitable for a memory-resident database. The index is implemented as a flat array of TIDs that are pointers to tuples. This structure saves the storage space significantly compared with conventional index structures. The reduction of the storage space allows us to have more indexes with less storage overhead. In fact, in OBE, it is possible to implement the strategy of having indexes for all the attributes in the database. This strategy obviates physical database design problem, which is a nuisance for novice users.

We have emphasized that a proper scheduling algorithm of the operating system is crucial for realizing a memory-resident database system. In particular, we have shown that the working-set scheduling algorithm provides an excellent

approximation for memory-residency of data. By using this algorithm, the system prevents potential thrashing due to heavy usage of virtual memory. In contrast, a pure demand paging scheme would not work in a practical time-shared environment (even with a physical memory size sufficient for a single user) because of potential thrashing.

Finally, we believe that a database system based on the memory-residency assumption is suitable for efficient main-memory applications including many aspects of artificial intelligence and logic programming (such as Prolog [9]). In particular, nonrecursive queries expressed in function-free Horn-clause logic can be directly processed by the techniques proposed in this paper [40].

APPENDIX A

We present cost formulas for the access structure Relation Scan. We first define some notation and introduce a function that is used in calculating the result cardinalities.

Notation

Nrpreds	Number of restriction predicates in Curr-Node.
NbndJpreds	Number of bound join predicates in Curr-Node.
NbndXpreds	Number of cross-reference predicates that are bound at Curr-Node.
Rsel	Joint selectivity of all restriction predicates in Curr-Node.
Jsel	Joint selectivity of bound join predicates in Curr-Node.
Xsel	Joint selectivity of cross-reference predicates that are bound at Curr-Node.
RXsel	Rsel × Xsel.
Sel	Joint selectivity of all bound predicates for Curr-Node.
Rcard	Number of tuples in the base relation of Curr-Node.
Rsltcard(N)	Result cardinality for node N.
Tuplesperaccess	Number of tuples retrieved in one access through an access structure.

Function *b*

Let tuples be partitioned into m groups ($1 \leq m \leq n$), each containing $p = n/m$ tuples. If k tuples are randomly selected from the n tuples, the expected number of groups hit (blocks with at least one tuple selected) is given by

$$\frac{b(m, p, k)}{m} = \left[1 - \left(1 - \frac{1}{m}\right)^k\right] + \left[\frac{1}{m^2 p} \times \frac{k(k-1)}{2} \times \left(1 - \frac{1}{m}\right)^{k-1}\right] + \left[\frac{1}{m^3 p^2} \times \frac{k(k-1)(2k-1)}{6} \times \left(1 - \frac{1}{m}\right)^{k-1}\right].$$

Details of the derivation of this function can be found in Whang et al. [41].

Result Cardinality

We now construct the formulas for the result cardinality. These formulas are independent of the specific access-structures chosen, but dependent on the partial join sequence.

```
IF Curr-Node is a negated node THEN
  Rsltcard(Curr-Node) = Rsltcard(Prev-Node) × (1 − b(1/Jsel, Jsel × Rcard,
      RXsel × Rcard) × Jsel))
ELSE IF Curr-Node is an ordinary node THEN
  Rsltcard(Curr-Node) = Rsltcard(Prev-Node) × (Rcard × Sel).
```

For a negated node, function b gives the number of groups selected according to the restriction predicates and cross-reference predicates. A *group* is a set of tuples having the same join column value. A group is *selected* if one or more of the tuples in the group satisfies these predicates. The value of function b is multiplied by Jsel (or, equivalently, divided by 1/Jsel) to obtain the probability that a specific group is selected. Since the negation specifies nonexistence of such a group, this probability must be subtracted from 1 to produce the probability that the group is not selected. The result is finally multiplied by Rsltcard(Prev-Node) to obtain Rsltcard(Curr-Node) since the selection process for Curr-Node is repeated as many times as Rsltcard(Prev-Node).

For an ordinary node the number of tuples selected according to the predicates is given by Rcard × Sel. Again, this number is multiplied by Rsltcard(Prev-Node) to produce Rsltcard(Curr-Node).

Cost Formulas for Relation Scan

Using the result cardinality, we construct the cost formula for the access structure Relation Scan as follows:

```
IF Curr-Node is a negated node THEN
      Tuplesperaccess = MIN(1/Sel, Rcard)
ELSE IF Curr-Node is an ordinary node THEN
      Tuplesperaccess = Rcard.
```

$Cost_{\text{Relation Scan}}$ =

```
  C1 × Nrpreds                          !expression evaluation for restriction predicates
  + Rsltcard(Prev-Node)
    × (C1 × Nbndjpreds
       + C1 × Tuplesperaccess           !expression evaluation for join predicates
       × (f(Nrpreds + NbndJpreds + NbndXpreds)
       − f(Nrpreds + NbndJpreds))       cross-reference predicate evaluation
     C5 × Tuplesperaccess               tuple retrieval
  + C4 × Tuplesperaccess × f(Nrpreds + Nbndjpreds))
                                        !comparison operations
```

For an ordinary node the search goes through all the tuples in the relation. Therefore, Tuplesperaccess, the average number of tuples to be searched until termination, must be Rcard. For a negated node, however, the search stops as soon as a tuple satisfying the predicates is found (hence, rendering the negated predicate false). Thus, Tuplesperaccess is 1/Sel with the restriction that it cannot be larger than Rcard.

The cost of Relation Scan is the summation of the costs for (1) expression evaluation for restriction predicates, (2) expression evaluation for bound join

predicates, (3) evaluation of cross-reference predicates, (4) tuple retrieval, and (5) comparison operations.

Expressions in restriction predicates are evaluated only once for a given query (according to the comparison-restriction-predicate flags, as explained in Section 5.3). On the other hand, those in bound join predicates must be evaluated every time Curr-Node is entered in the join loop (according to the comparison-bound-join-predicate flags, as explained in Section 5.3). Thus, the number of bound join predicates is multiplied by Rsltcard(Prev-Node). Cross-reference predicates are evaluated for every tuple because example elements in these predicates are bound to different values for different tuples. Thus, for them, the number of predicate evaluations is multiplied by Tuplesperaccess as well as Rsltcard(Prev-Node). Note that the function f is used to calculate the number of evaluations of cross-reference predicates because they are evaluated only "after" all restriction and bound join predicates prove to be true. The next term accounts for the tuple retrieval cost. Finally, the last term is the number of comparison operations for both restriction and bound join predicates.

ACKNOWLEDGMENTS

Arthur Ammann designed HRMS and deserves the credit for initiating the memory-residency idea. This idea was picked up and fully expanded by the authors.

The authors also wish to acknowledge the contributions of other members of the OBE project in designing and implementing the OBE system: Anthony Bolmarcich, Maria Hanrahan, Guy Hochgesang, Kuan-Tsae Huang, Al Khorasani, Gary Sockut, Paula Sweeney, Vance Waddle, and Moshe Zloof. Carolyn Turbyfill and Dina Bitton performed the formal benchmarking. Anil Nigam and John Robinson read an earlier version of this paper and contributed valuable comments.

REFERENCES

1. AMMANN, A. C., HANRAHAN, M., AND KRISHNAMURTHY, R. Design of a memory resident DBMS. In *Proceedings of the International Conference Compcon Spring '85*. IEEE, New York, 1985.
2. ASTRAHAN, M. M., ET AL. System R: Relational approach to database management. *ACM Trans. Database Syst. 1*, 2 (June 1976), 97–137.
3. BITTON, D., HANRAHAN, M. B., AND TURBYFILL, C. Performance of complex queries in main memory database systems. In *Proceedings of the 3rd International Conference on Data Engineering* (Los Angeles, Calif., Feb. 1987).
4. BITTON, D., AND TURBYFILL, C. Main memory database support for office systems: A performance study. In *Proceedings of IFIP WG8.4 Working Conference on Methods and Tools for Office Systems* (Pisa, Italy, Oct. 1986).
5. BITTON, D., AND TURBYFILL, C. Performance evaluation of main memory database systems. Tech. Rep. 85-735, Cornell University, Ithaca, N.Y., Feb. 1986.
6. BLASGEN, M. W., AND ESWAREN, K. P. On the evaluation of queries in a database system. IBM Res. Rep. RJ1945, IBM, San Jose, Calif., April 1976.
7. CHAMBERLIN, D. D., ET AL. SEQUEL2: A unified approach to data definition, manipulation, and control. *IBM J. Res. Dev. 20*, 6 (Nov. 1976), 560–575.
8. CHRISTODOULAKIS, S. Implications of certain assumptions in database performance evaluation. *ACM Trans. Database Syst., 9*, 2 (June 1984), 163–186.

9. CLOCKSIN, W. F., AND MELLISH, C. S. *Programming in Prolog.* Springer-Verlag, New York, 1981.
10. DEWITT, D. J., ET AL. Implementation techniques for main memory database systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (Boston, Mass., 1984). ACM, New York, 1984, pp. 1–8.
11. GARCIA-MOLINA, ET AL. A massive memory database system. Unpublished manuscript, Dept. of Electrical Engineering and Computer Science, Princeton University, Princeton, N.J.
12. GRAY, J., MCJONES, P., BLASGEN, M., LINDSAY, B., LORIE, R., PRICE, T., PUTZOLU, F., AND TRAIGER, J. The recovery manager of the System R database manager. *ACM Comput. Surv. 13*, 2 (June 1981), 223–242.
13. HALL, P. A. V. Optimization of a single relational expression in a relational database. *IBM J. Res. Dev. 20*, 3 (1976), 244–257.
14. HAMMER, M., AND CHAN, A. Index selection in a self-adaptive database management system. In *Proceedings of the ACM International Conference on Management of Data* (Washington, D.C., June 1976). ACM, New York, 1976, pp. 1–8.
15. HILLER, F. S., AND LIEBERMAN, G. J. *Introduction to Operations Research*, 3rd ed. Holden-Day, San Francisco, Calif., 1980.
16. IBARAKI, T., AND KAMEDA, T. On the optimal nesting order for computing N-relational joins. *ACM Trans. Database Syst. 9*, 3 (Sept. 1984), 482–502.
17. IBM. VM/SP: System Logic and Problem Determination Guide (CP), LY20-0892-2, 3rd ed. IBM Marketing, Sept. 1983.
18. JARKE, M., AND KOCH, J. Query optimization in database systems. *ACM Comput. Surv. 16*, 2 (June 1984), 111–152.
19. KAMBAYASHI, Y., AND YOSHIKAWA, M. Query processing utilizing dependencies and horizontal decomposition. In *Proceedings of ACM International Conference on Management of Data* (San Jose, Calif., May 1983). ACM, New York, 1983, pp. 55–67.
20. KIM, W. On optimizing an SQL-like nested query. *ACM Trans. Database Syst. 7*, 3 (Sept. 1982), 443–469.
21. KITSUREGAWA, M., ET AL. Application of hash to data base machine and its architecture. *New Generation Comput. 1* (1983), 62–74.
22. KNUTH, D. *The Art of Computer Programming—Sorting and Searching*, Vol. 3. Addison-Wesley, Reading, Mass., 1973.
23. KOOI, R., AND FRANKFORTH, D. Query optimization in INGRES. *IEEE Database Eng. Bull. 5*, 3 (Sept. 1982), 2–5.
24. KRISHNAMURTHY, R., BORAL, H., AND ZANIOLO, C. Optimization of nonrecursive queries. In *Proceedings of the 12th International Conference on Very Large Data Bases* (Kyoto, Japan, 1986), pp. 128–137.
25. LEHMAN, T., AND CAREY, M. A study of index structures for main memory database management systems. In *Proceedings of the 12th International Conference on Very Large Data Bases* (Kyoto, Japan, Sept. 1986), pp. 294–303.
26. LEHMAN, T., AND CAREY, M. Query processing in main memory database management systems. In *Proceedings of the ACM International Conference on Management of Data* (Washington, D.C., May 1986). ACM, New York, 1986, pp. 239–250.
27. LUK, W. S. On estimating block accesses in database organization. *Commun. ACM 26*, 11 (Nov. 1983), 945–947.
28. PECHERER, R. M. Efficient evaluation of expressions in a relational algebra. In *Proceedings of the ACM Pacific Conference* (1975). ACM, New York, 1975, pp. 44–49.
29. POWER, L. R. EPLEA, using execution profiles to analyze and optimize programs. IBM Res. Rep. RC9932, IBM T. J. Watson Research Center, Yorktown Heights, N.Y., April 1983.
30. REINER, D. S., Ed. *IEEE Database Eng. Bull. 5*, 3 (Sept. 1982).
31. SCHKOLNICK, M., AND TIBERIO, P. Estimating the cost of updates in a relational database. *ACM Trans. Database Syst. 10*, 2 (June 1985), 163–179.
32. SELINGER, P. G., ET AL. Access path selection in a relational database management system. In *Proceedings of the ACM International Conference on Management of Data* (Boston, Mass., May 1979), pp. 23–24.
33. SHAPIRO, L. D. Join processing in database systems with large main memories. *ACM Trans. Database Syst. 11*, 3 (Sept. 1986), 239–264.

34. SMITH, J. M., AND CHANG, P. Y.-T. Optimizing the performance of a relational algebra database interface. *Commun. ACM 18*, 10 (1975), 568–579.
35. STONEBRAKER, M., ET AL. The design and implementation of INGRES. *ACM Trans. Database Syst. 1*, 3 (Sept. 1976), 189–222.
36. ULLMAN, J. D. *Principles of Database Systems.* Computer Science Press, Rockville, Md., 1982.
37. VANDER ZANDEN, B. T., TAYLOR, H. M., AND BITTON, D. Estimating block accesses when attributes are correlated. In *Proceedings of the 12th International Conference on Very Large Data Bases* (Kyoto, Japan, Aug. 1986), pp. 119–127.
38. WARREN, D. H. D. Efficient processing of interactive relational database queries expressed in logic. In *Proceedings of the 7th International Conference on Very Large Data Bases* (Cannes, France, 1981), pp. 272–281.
39. WHANG, K.-Y. Query optimization in office-by-example. IBM Res. Rep. RC11571, IBM T. J. Watson Research Center, Yorktown Heights, N.Y., Dec. 1985.
40. WHANG, K.-Y., AND NAVATHE, S. An extended disjunctive normal form approach for processing recursive logic queries in loosely coupled environments. In *Proceedings of the 13th International Conference on Very Large Data Bases* (Brighton, England, Sept. 1987), pp. 275–287.
41. WHANG, K.-Y., WIEDERHOLD, G., AND SAGALOWICZ, D. Estimating block accesses in database organizations—A closed noniterative formula. *Commun. ACM* 26, 11 (Nov. 1983), 940–944.
42. WHANG, K.-Y., WIEDERHOLD, G., AND SAGALOWICZ, D. Separability—An approach to physical database design. *IEEE Trans. Comput. C-33*, 3 (Mar. 1984), 209–222.
43. WHANG, K.-Y., ET AL. Office-by-example: An integrated office system and database manager. *ACM Trans. Office Inf. Syst. 5*, 4 (Oct. 1987), 393–427.
44. WIEDERHOLD, G. *Database Design.* McGraw-Hill, New York, 1983.
45. WINSTON, P. H. *Artificial Intelligence.* Addson-Wesley, Reading, Mass., 1979.
46. WONG, E., AND YOUSEFFI, K. Decomposition—A strategy for query processing. *ACM Trans. Database Syst. 1*, 3 (Sept. 1976), 223–241.
47. YAO, S. B. Optimization of query evaluation algorithms. *ACM Trans. Database Syst. 4*, 2 (June 1979), 133–155.
48. ZLOOF, M. M. Office-by-example: A business language that unifies data and word processing and electronic mail. *IBM Syst. J. 21*, 3 (1982), 272–304.
49. ZLOOF, M. M. QBE/OBE: A language for office and business automation. *IEEE Comput. 14*, 53 (May 1981), 13–22.
50. ZLOOF, M. M. Query-by-example: A data base language. *IBM Syst. J. 16*, 4 (1977), pp. 324–343.

Received February 1987; revised March 1988; accepted January 1989