A model of storage and access to a relational data base is presented. Using this model, four techniques for evaluating a general relational query that involves the operations of projection, restriction, and join are compared on the basis of cost of accessing secondary storage. The techniques are compared numerically and analytically for various values of important parameters. Results indicate that physical clustering of logically adjacent items is a critical performance parameter. In the absence of such clustering, methods that depend on sorting the records themselves seem to be the algorithm of choice.

Storage and access in relational data bases

by M. W. Blasgen and K. P. Eswaran

E. F. Codd has introduced a relational model of data that permits a high degree of data independence by providing a logical view of the data base.[1] Such a view avoids the details of physical storage of data and the access paths, but places the burden of determining efficient evaluation of methods for queries and updates on the data base management system itself.

This paper describes four methods for evaluating a fairly general query involving the operations of join, projection, and restriction. (A more general treatment of this topic is given in Reference 2.) In References 1 and 2, comparisons are made to discover the method or methods that make the fewest accesses to secondary storage. It is shown that the best method of evaluating such a query depends on the available access paths to the relation (such as by indexes), the physical clustering of logically adjacent items, and the characteristics of the query itself.

Although several relational data base systems have been implemented,[3-6] little has appeared on the performance of such systems. Of the references that have appeared in the literature, few consider the implementation of relational operators. Gotlieb[7] considers only the computation of joins in isolation. Pecherer[8] discusses the evaluation of relational operators in an abstract machine. However, the cost of accesses to secondary storage is not considered by any of them. Since we think that these accesses are the most critical performance parameter, this is the basis of comparison. We consider neither the CPU time nor the cost of virtual storage management. The work described here was initiated during the design of an experimental relational data base system that is known as System R.[9]

Concepts of relational data bases and queries

relational data bases

A *relation* is a table, the entries in which may be thought of as members of a set of *tuples*, each of which consists of a number of *column values*. A relation may also be thought of as a file, in which case a *tuple* is a record and a column is a field. The *size* of a relation R is the number of tuples in R and is denoted by |R|. A relation R with columns A, B, and C is written R(A, B, C). The expression P.R(A, B) represents the *projection* of R on (A, B) and is a relation with two columns A and B. A projection contains a tuple (x, y) if and only if the tuple (x, y, z) is contained in B for some z. Let r be a tuple (entry) of a relation (table) R and let s be a tuple (entry) of S. If A is a column in R, then r(A) is the value of column A in entry r. The *equijoin* of relations R and S on columns A in R and B in S is defined to be $\{(r||s) : r(A) = s(B)\}$, where (r||s) denotes the tuple formed by concatenating r from R and s from S. If for every tuple r of R there exists at most one tuple s of S such that $r(A) = s(B)$, then we say the join is *one to many* with respect to S. If the join, is not one to many with respect to R or S, then the join is *many to many*, i.e., there are distinct tuples r, r′, s, and s′ such that $R(A = r'(A) = s(B) = s'(B)$.

Given a predicate P, the *restriction* operation on a relation R selects the set of tuples {r : r is a tuple of R and P(r) is true}. In addition to arbitrary predicates, our analysis makes reference to *simple predicates*, of the form COLUMN-NAME *op* CONSTANT, where *op* is one of the following set of mathematical operators: {=, ¬=, <=, >=, <, >}. The column involved in a simple predicate is called the restriction column.

queries

The general query that is analyzed in this paper involves restriction, projection, and join. The general query has the following form: Apply a given restriction to a relation R, yielding R′, and apply a possibly different restriction to a relation S, yielding S′. Join R′ and S′ to form a relation (new table) T, and project some columns from T.

This specification of the general query should be thought of as a high-level query. We now consider a particular example and several different methods of evaluating this query.

example

Consider the following relations: PART(PART_NO,SUPPLIER_NO,QTY_ON_HAND,PRICE) and SUPPLIER(SUPPLIER_NO,SUPPLIER_CITY,SUPPLIER_RATING). A part may be supplied by more than one supplier and a supplier may supply more than one part. The example is as follows:

Find PART_NO, SUPPLIER_NO, and SUPPLIER_CITY from the relations PART and SUPPLIER where PRICE is greater than 100 dollars, and the part is supplied by suppliers with SUPPLIER_RATING less than 5.

The query is an equijoin of the two relations on SUPPLIER_NO, and is of the many-to-many type in which the predicates in the restrictions are simple predicates. The result required is a projection of the relation that is a join of relations PART and SUPPLIER, with restrictions applied.

Access and storage models

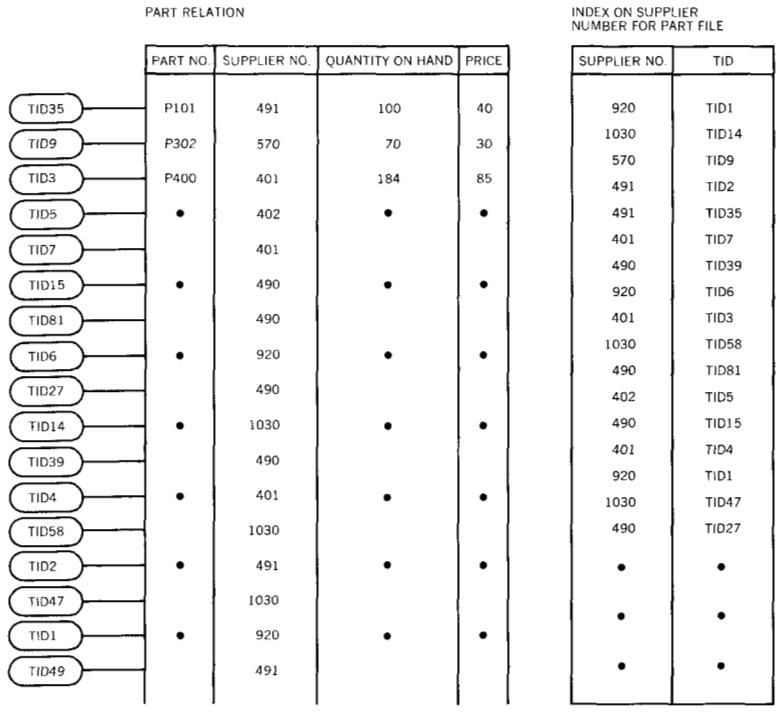
The speed of evaluation of a query depends on the performance of the basic methods of representing and referencing the data base. This section describes our model of storage and of access to the data base.

The data base is assumed to reside on secondary storage, which consists of direct access storage devices. Physical storage space on secondary storage is divided into fixed-size blocks called *pages*, which are the units of secondary storage allocation and the unit of transfer between main storage and secondary storage. Pages are divided into two categories: *data pages* and *index pages*. Data pages are used to store the tuples of the relations in the data base. A page may contain tuples from more than one relation. Each tuple in the data base has a unique identifier called the *Tuple Identifier* or TID. A TID is assumed to give direct access to the tuple, so that at most one page is read if a tuple is referenced using a TID. TIDs also have the property that extracting a set of tuples using a sorted sequence of TIDs accesses a data page at most once.

A *segment* is a large address space that contains one or more relations. The segment is implemented as a set of pages. Each stored tuple in a segment contains the name of the relation of which it is a member. To obtain all the tuples of a relation, the segment can be scanned by fetching the pages one at a time and checking every tuple in the page for a membership in the desired relation. Such a scan is called a *segment scan*, and references each page in the segment once.

A segment scan is potentially slow, has no selectivity, and provides the tuples in a system-determined order. In many cases, it is desirable to have other access paths. The type of path to be described here is that of indexes.[2] Reference 2 also considers another type of access path, termed *links*. IMS[10] and DBTG[11] implementations use the concept of links as well as indexes.

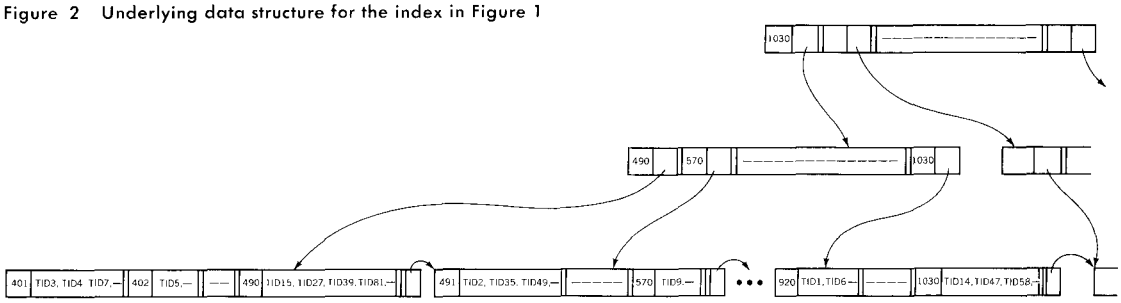
Figure 1 Part relation and index on supplier number



index A single column *index* on a column A of a relation R consists of pairs whose first component is a value from column A of R and whose second component is the TID of a tuple having that value. We say that the index consists of the following pairs: (KEY, TID). An index is stored in a special way to provide rapid access to it. The model we adopt here is that of a VSAM-like tree[12] that is similar to a B-tree.[13] A supplier-number index on a PART relation is shown in Figure 1. Figure 2 shows the storage of the index in Figure 1. The storage design is that of a balanced tree whose nodes are index pages. Leaf pages contain the (KEY, TID) pairs in sorted order, and the higher-level pages contain pairs that consist of the high key on a lower-level page with a pointer to that page. These pairs are also sorted.

An index on column A permits rapid access to a single tuple that has a desired value in column A. To find a TID, the number of index pages referenced is equal to the height of the tree. An index also permits all tuples to be retrieved in sorted order, i.e., the order of increasing values of column A. Subsequences of tuples may also be retrieved in sorted order, e.g., all tuples where column A values are between 10 and 20. Note than an index can provide the TIDs of the tuples that satisfy a simple predicate without access to the data pages.

Figure 2 Underlying data structure for the index in Figure 1



The utility of an index in evaluating a query depends on whether the relation is clustered or unclustered with respect to the index. Suppose an index I is used to extract the tuples (possibly in some range) of a relation R in sorted order. If each data page of R is accessed at most once, then R is *clustered* with respect to the index I. The index I may be termed a *clustering* index with respect to R. On the other hand, if the data pages of R are referenced in a random, approximately uniformly distributed manner, then R is *unclustered* with respect to I. When the number of buffer pages available in main storage is small compared to the number of data pages for R, each fetch of a tuple using a nonclustering index usually results in a secondary storage access to fetch a data page.

To understand the importance of clustering, suppose we must obtain a sequence of M tuples corresponding to an interval of key values in index I of relation R. If I is a clustering index, then this sequence can be obtained by accessing only (approximating to a first order) $(M/|R|) \times D$ pages, where D is the number of data pages of relation R and $|R|$ is the size of R. If it is a nonclustering index, the M data pages will be accessed. The difference in performance is considerable.

sorting

The sorting of tuples on the value in a column (the sort key) forms an important step in one of the algorithms described and analyzed in this paper. For file sizes that are typical in a data base environment, an internal sort is ruled out. Thus we consider an external sort-merge, such as is discussed in Reference 14. The magnitude of a sort-merge may be estimated as follows. Suppose that a given sort/merge algorithm uses a block of Q pages as the unit of transfer between the main storage and secondary storage, and uses a Z-way merge. Sorting a file of M pages requires $(2M/Q) \log_Z (M/Q)$ secondary storage accesses.

Methods

Four methods for evaluating the general query given previously in this paper are described in this section. The methods differ in the way they use TIDs, indexes, and sorting.

Table 1 Boolean variable is true when there is an index on the join column

Boolean Variable	*Corresponding access path*
V_1	Index on the join column of R
V_2	Index on the join column of S
V_3	Index on the restriction column of R
V_4	Index on the restriction column of S

Method 1: indexes on join columns

In this method, the indexes on the join columns of R and S are scanned to determine whether a pair of tuples has the same value in the join columns, and thus is present in the unrestricted join. If it is, the tuple from, say, R, is obtained and checked to see whether it satisfies the restriction predicate. Then by continuing to scan along the index for S, all the tuples from S that have this key value are obtained, and the projections of tuples from S that satisfy the restriction are placed in a temporary storage. By scanning along the index for R, all tuples with this key value are obtained, the restriction is applied, and subtuples of interest are projected. These are then joined with the subtuples in the temporary storage, and the resultant tuples are placed in the output relation. The temporary storage must be sufficiently large to hold the maximum number of restricted subtuples of interest from S that participate in the join with a single tuple from R.

Method 2: sorting both relations

Sorting is used as an aid in joining the two relations. By scanning the relations R and S using some access path, two files W_1 and W_2 are created. File $W_1(W_2)$ contains subtuples corresponding to tuples of R(S) that satisfy the predicate, and consists of columns of R(S) that are in the output or the join predicate. Files W_1 and W_2 are sorted on the join column values. The resulting sorted files are scanned, and the join is performed.

Method 3: multiple passes

A scan is used to obtain the tuples of S. If the access path is not the restriction column index, the restriction predicate is applied as each tuple is obtained. A qualified tuple is projected, and the resulting subtuple s is inserted into a main storage data structure W_2' if there is space. If there is no space and if the join column value in s is less than the current highest join column value in W_2', the subtuples with the highest join column value in W_2 are deleted and s inserted. If there is no room for s and the join column value in s is greater than the highest join column value in W_2', s is not inserted at all. After forming W_2', R is scanned using some access path, and a tuple T_1 of R is obtained. If T_1 satisfies the predicate, then W_2' is checked for the presence of the join column value of T_1. If the join column is present, T_1 is joined to the appropriate subtuples in W_2'.

Table 2 Applicability of methods

Method	*When the following expression is true the corresponding method is applicable*
1	V_1 AND V_2
2	Always applicable since a segment scan can be used
3	Always applicable since a segment scan can be used
4	V_1 AND V_2 AND V_3 AND V_4

If there are more qualitifed tuples in S than can fit in main storage for W_2', another scan of S is made to form a new W_2' that consists of subtuples with join column value greater than the current highest. R is scanned again and the process is repeated. Duplicates need not be kept in W_2'. The data structure for W_2' may be a binary tree, heap, or hash table.

This method is very fast if there is only a single pass: that is, when there is enough main storage to hold all the qualified subtuples of S.

Method 4: simple TID algorithm

This algorithm uses indexes and TIDs as much as possible, and requires that join column and restriction column indexes exist. Using the restriction column indexes, the TIDs of tuples that satisfy the predicates are obtained, and the resultant TIDs are sorted and stored in files R′ and S′. Scanning the join column indexes, TIDs of tuples that jointly participate in the unconstrained join are found. As they are found, each TID pair (TID_1, TID_2) is checked to see whether TID_1 is present in R′ and TID_2 is in S′. If these conditions are met, the tuples are fetched and joined, and the subtuple of interest is obtained.

Applicability of the methods

Whether or not a method can be applied depends on the existence of various access paths. For example, Method 4 is applicable only if there are indexes on the join column of R, on the join column of S, and on the restrictions columns of R and of S. Table 1 defines a boolean variable V_1 to be TRUE when there exists an index on the join column of R, and to be FALSE otherwise. Other variables are similarly defined. Table 2 indicates the conditions under which each method is applicable. For example, Method 1 is applicable when the expression V_1 and V_2 is true.

Systems parameters

One of the most difficult tasks in the analysis and enhancement of system performance is to determine the parameters that sig-

nificantly affect it. After these parameters have been determined they must be measured or estimated. Finally, a model dependent on these parameters must be developed that is useful for predicting performance. A list of parameters that are used in our model is now given. The following list is divided into two parameter types: those that depend only on the data base and those that depend on the query.

Data-base dependent (query independent) parameters:

- $N_1(N_2)$ Cardinality of relation R (S)
- $E_1(E_2)$ Average number of tuples from R(S) in a data page.
- $M_1(M_2)$ Total number of data pages in the segment that contains R(S).
- L Average number of (Key, TID) pairs in a leaf page of an index.
- $C_1(C_2)$ Number of tuples of R(S) that fit in a page of a (temporary) file, as obtained by dividing the size of a page in the file by the average size of a tuple. C_1 may be different from E_1 because a data page in data base may contain tuples from more than one relation. (This is not the case for temporary files.)
- $P_1(P_2)$ Effectiveness of the *join filter* for R(S), i.e., the fraction of tuples of R that participate in the unconditional equijoin.
- G Ratio between the number of tuples in the unconditional equijoin and $N_1 \times N_2$.
- I Number of TIDs that fit in a page of a temporary file.
- K Number of (Key, TID) pairs that fit in a page of a temporary file.
- P Main storage space in page frames that are avilable for sort buffers, TID lists, TID pair lists, W_2', etc.
- Z Merge factor for sort-merge algorithms.
- A Cost of a page transfer.
- B Cost of a block transfer. A block is the unit of transfer for a file.

Query-dependent parameters:

- $H_1(H_2)$ Ratio between the average size of the subtuple of interest from R(S) and the average size of a tuple.
- $F_1(F_2)$ Effectiveness of the *predicate filter* for R(S), i.e., the ratio between the number of tuples that satisfy the predicate and the cardinality of the relation.

The data-base dependent parameters can be estimated at load time, and they need be updated only when the data base is reorganized. Although this condition adds an overhead each time the data base is reorganized, the cost is recovered as quer-

ies are made. The query-dependent parameters can also be estimated. For example, F_1 can be estimated as follows: If the predicate is of the form "column = constant," and there is an index on that column, then estimate F_1 as the ratio between number of distinct values in the index and N_1. If the predicate involves an interval of key values, F_1 can be estimated by using information in the root page of the index tree. In this way, a query evaluator can have estimates of the critical performance parameters.

Cost analysis

Considered in this section are expressions for the cost that some of the methods incur in accessing secondary storage (or, alternatively, the number of accesses to secondary storage). The performance of a method depends strongly on the clustering of relations with respect to access paths. For example, in Method 1 there are two cases. If R is unclustered with respect to the index on the join column of R, then fetching the tuples of R using the index requires N_1 accesses. If, however, the relation is clustered, then it requires N_1/E_1 accesses.

An index can be on the join column, on the restriction column, or on some other column. A relation may be clustered or unclustered with respect to the index. In this way, each of the four methods may have many cases, resulting in a large number of situations to be analyzed. Fortunately, the cost analysis is straightforward when the cost computations for the basic steps are understood. We now illustrate cost analysis for a few methods and cases to exemplify the cost calculation procedure.

In Method 1, if both R and S are clustered on the join column indexes, the cost of scanning the join column index of R is $A \times N_1/L$, and the cost of obtaining the tuples of R is $A \times P_1 \times N_1/E_1$. Similarly, the cost of scanning the join column index of S is $A \times N_2/L$. The cost of scanning S is $A \times P_2 \times F_1 \times N_2/E_2$. (We know that $P_2 \times N_2$ tuples of S qualify for the unconditional join, $F_1 \times N_1$ of R satisfy the predicate, and we fetch only those tuples of S that have the same join value.) Thus the total cost of joining is $A \times (N_1/L + P_1 \times N_1/E_1 + N_2/L + P_2 \times F_1 \times N_2/E_2)$.

On the other hand, if Method 1 is used when both relations are unclustered on the join column indexes, the cost of scanning and obtaining tuples from R is $A \times (N_1/L + P_1 \times N_1)$, and the cost of obtaining the tuples from S is $A \times (N_2/L + P_2 \times F_1 \times N_2)$. Thus the total cost of Method 1 in this case is $A \times (N_1/L + N_2/L + P_1 \times N_1 + P_2 \times F_1 \times N_2)$.

We now consider Method 2, in the case where a clustering index on column X(Y) is used to scan R(S). X(Y) is neither a col-

umn in the prediate not the join column. The cost of obtaining tuples of R is then $A \times (N_1/L + N_1/E_1)$, and the cost of forming a file that consists of subtuples of R is $B \times (N_1 \times F_1 \times H_1/(C_1 \times Q))$, where Q is the size of the block (in number of pages for the file). The cost to sort the file is $2 \times B \times (N_1 \times F_1 \times H_1/(C_1 \times Q) \times (\log(N_1 \times F_1 \times H_1/(C_1 \times Q)) - 1)$. The term -1 arises because the first pass of the sort-merge can be done when the file is formed. The cost of scanning the sorted file is $B \times (N_1 \times F_1 \times H_1/(C_1 \times Q))$. Thus the total cost is $A \times (N_1/L + N_1/E_1 + N_2/L + N_2/E_2) + 2 \times B \times (N_1 \times F_1 \times H_1 \times \log (N_1 \times E_1 \times H_1/(C_1 \times Q)) + N_2 \times F_2 \times H_2 \times \log (N_2 \times F_2 \times H_2/(C_2 \times Q))))$.

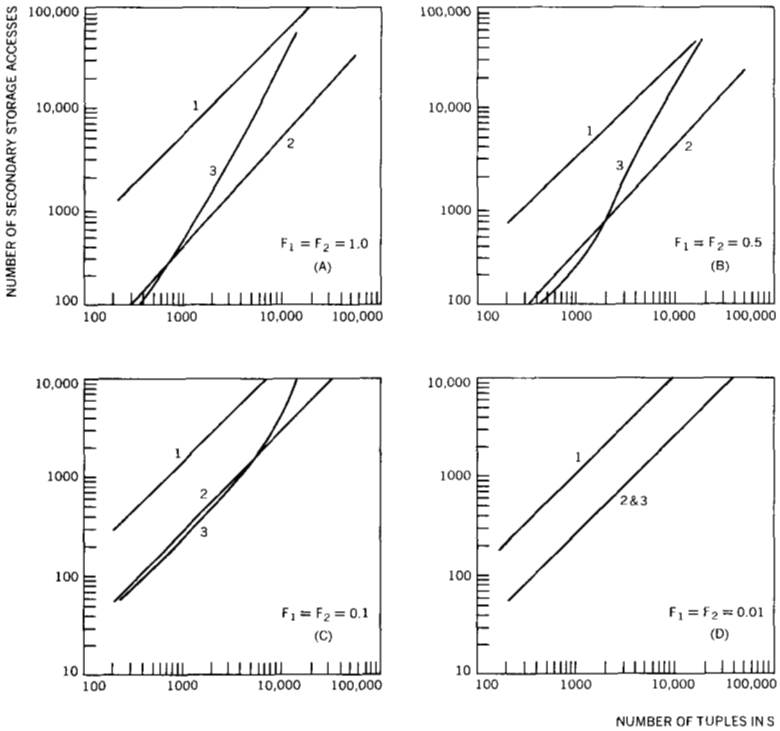
For Method 4, if R and S are clustered with respect to the join column indexes, the total cost is the sum of the following four costs: (1) the cost to form R′ and S′; (2) the cost to scan join column indexes, which is $A \times (N_1/L + N_2/L)$; (3) the cost to check R′ and S′; and (4) the cost to fetch tuples to form the output, which is $A \times F_1 \times F_2 \times (P_1 \times N_1/E_1 + P_2 \times N_2/E_2)$. The costs of (1) and (3) depend on whether R′ and S′ fit in main storage. If they do, the cost for (1) is $A \times (F_1 \times N/L + F_2 \times N_2/L)$, and for (3) the cost is zero. If R′ and S′ do not fit in main storage, the cost for (1) is $A \times (F_1 \times N_1/L + F_2 \times N_2/L) + B \times (2 \times \alpha_1) \times \log (\alpha_1) + 2 \times \alpha_2 \times \log (\alpha_2))$, where α_1 is $N_1 \times F_1/(I \times Q)$ and α_2 is $N_2 \times F_2/(I \times Q)$, and the cost for (3) is $B \times (N_1 \times F_1/(I \times Q) + N_2 \times F_2/(I \times Q))$, since both the probes are sequential.

In Method 4, if neither R nor S is clustered with respect to the join column indexes, then both R′ and S′ are randomly probed. The total cost is then the following sum: (1) the cost to form R′ and S′ as above; (2) the cost to scan the join column indexes, which is $A \times (N_1/L + N_2/L)$; (3) the cost to search randomly both R′ and S′, which is $B \times G \times N_1 \times N_2 \times (\text{Min}(1, (1 - P/(2 \times N_1 \times F_1/I)))) + F_1 \times \text{Min}(1, (1 - P/(2 \times N_2 \times F_2/I))))$; and (4) the cost to fetch the tuples to form the output, which is $A \times (F_1 \times F_2 \times P_1 \times N_1 \times F_1 \times F_2 \times P_2 \times N_2)$.

Comparisons of the four methods

In this section, comparisons are made of the four methods under various conditions. The comparisons take the following form. A certain situation is postulated and the methods and cases that apply in the assumed situation are determined. (For example, if unclustered join column indexes are available, then some case of Method 1 applies.) The cost expressions are then evaluated for the relatively small number of methods that apply. The graphs in Figures 3 to 5, which describe the costs of the applicable methods in the three situations, illustrate this approach.

Figure 3 Comparison of the costs of the applicable methods in Situation A



The situations that we believe are typical and are explored in the graphs are the following:

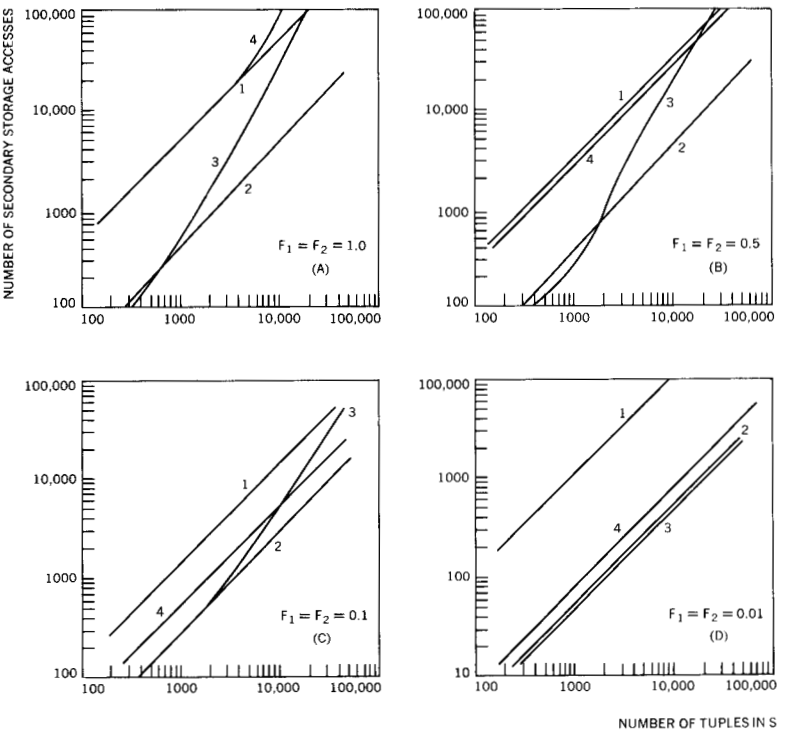
A. There are join column indexes, and indexes on irrelevant columns X and Y. R and S are not clustered on the join column indexes; instead they are clustered on indexes on columns X of R and Y of S.

B. There are join column indexes, restriction column indexes and indexes on irrelevant columns X and Y. R and S are not clustered on the join column indexes or on the restriction column indexes. They are clustered on columns X and Y.

C. There are join column indexes, and indexes on the restriction columns. R and S are clustered on the join column indexes, and not clustered on the restriction columns.

The costs of evaluating the general query in these three situations are indicated in the three figures. For each of the situations, we graph the number of secondary storage accesses as a function of relation size, with the other systems parameters constant. The graphs in a figure show the sensitivity to F_1 and F_2, the selectivity of the predicates. The number on the curves in

Figure 4 Comparisons of the costs of the applicable methods in Situation B



each graph are the appropriate method numbers. When more than one case of a method applies to a situation, only the case with minimum cost is shown.

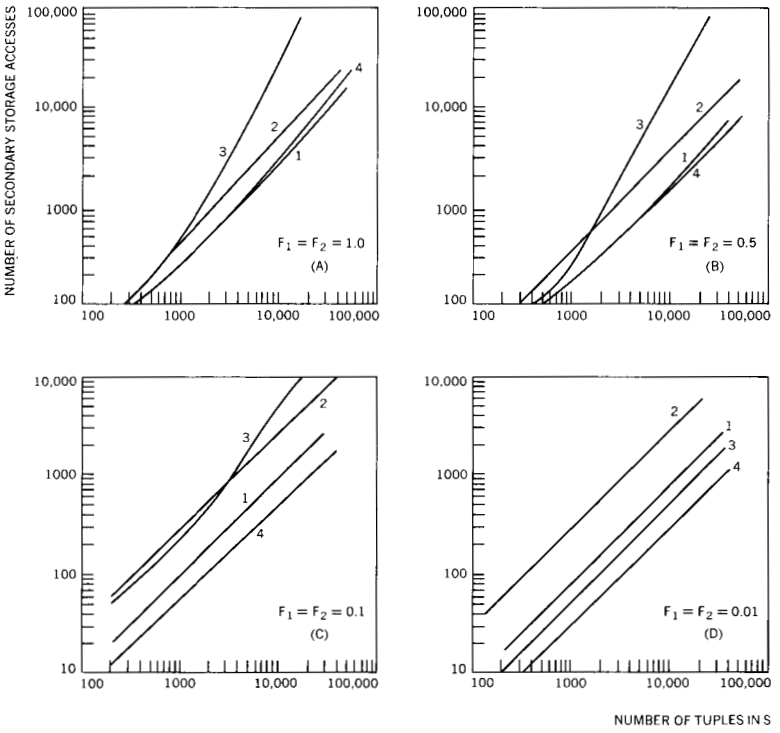
Observations

We have introduced the notion of clustering of indexes with respect to relations, and have observed that this clustering plays an important role in the choice of query evaluation algorithms. Perhaps the most interesting conclusion we can draw from this study is that there are circumstances under which each method is the best. As shown in Figures 3–5, each of the methods is best, given certain conditions. Also, the cost difference between the best method and the second best method is appreciable in most situations.

All methods described in this paper scan a relation to obtain tuples that satisfy some predicate. If several access paths are available, it is possible to determine which is best.

Suppose, for example, that both a clustering index and an index on the restriction column are available. By simple analytical comparisons, the following observations can be made. Use the

Figure 5 Comparison of the costs of the applicable methods in Situation C



restriction column index as the access path if it is the clustering index. Otherwise, if $M_1 < F_1 \times (N_1 + N_1/L)$ and $M_1 < N_1 \times (1/L + 1/E_1)$, then use a segment scan. If not, the nonclustering index on the restriction column is the access path of choice when $F_1 \times (1 + 1/L)$ is less than $(1/L + 1/E_1)$, or approximately, $F_1 < 1/E_1$ (i.e., when less than one tuple per page is retrieved). Otherwise use any clustering index as the access path. This observation should be used in all circumstances to choose an access path to scan a relation satisfying a predicate.

Numerical comparisons have led to the following observations.

Figure 3 illustrates the costs for situation A, where there are nonclustering indexes on the join columns and clustering X and Y indexes. Method 3 is the choice if all of W_2' fits in main storage. Otherwise we choose Method 2, which sorts the tuples. Method 1 should never be employed in this situation. The join column indexes are never used, and thus this conclusion applies even if there are no join column indexes.

Figure 4 deals with situation B, in which there are nonclustering join column indexes, nonclustering restriction column indexes, and clustering X and Y indexes. For situation B, Method 3 is the best (by a small margin) when the number of scans is one.

Otherwise, if $F_1 > 1/E_1$ and $F_2 > 1/E_2$, then Method 3 is preferred with the clustering X and Y indexes as access paths, and in the remaining circumstances, Method 4 is best. Although it is not shown in the figure, if the relations are clustered with respect to the restriction column indexes, the conclusions are similar. Again Method 3 (using the clustering indexes as access paths) is optimal if the number of scans on relation R is one; otherwise Method 2 (also using the clustering indexes) is best. In Figure 4, $N_2 = 4 \times N_1$, $E_1 = E_2 = 20$, $I = 1000$, $C_1 = C_2 = 20$, $L = 200$, $K = 300$, $P_1 = P_2 = 1.0$, $H_1 = H_2 = 0.5$, $G = 7/N_2$, $P = 25$, $A = B = 1.0$, $Z = 3$.

In situation C, graphed in Figure 5, we assume that both relations are clustered with respect to the join column indexes, and there are nonclustering indexes on the restriction columns. Method 4 is a good choice for situation C. In Figure 5, $N_1 = 4 \times N_2$, $E_1 = E_2 = 20$, $I = 1000$, $C_1 = C_2 = 20$, $L = 200$, $K = 300$, $P_1 = P_2 = 1.0$, $H_1 = H_2 = 0.5$, $G = 7/N_2$, $P = 25$, $A = B = 1.0$, and $Z = 3$.

As a general conclustion, when there are clustering indexes on the join columns and there are no restriction column indexes, Method 1 is uniformly the best.

Concluding remarks

Using the observations in this paper, a query translator could operate in the following manner. From the available access paths, determine the applicable methods and cases, eliminate any obviously bad methods, discard any methods that fail to pass certain simple tests (such as, for example, $F_1 < 1/E_1$), and then evaluate the cost estimates for the remaining methods. Choose the method with minimum cost. A query evaluator based on these principles of simple analytic calculations and numeric cost computations could be part of relational data base query systems or other system that uses indexes. A complete model to analyze the cost of various methods that apply to any given situation has been implemented in APL. The time to analyze a particular situation is of the order of a few milliseconds.

In practical implementations, the approach taken here is preferable to solving an analytic model under simplified assumptions, which are usually invalid in practice.

Any higher-level optimization techniques in evaluating queries should take into account the existing access paths and their properties (which are reflected at the low storage level). It is our belief that general high-level transformation techniques such as those applied in programming language compilers may not be of

much use in query language processors unless the access path characteristics and system parameters are taken into consideration.

ACKNOWLEDGMENTS

We thank Professor Rudolf Bayer of the Technical University of Munich, Germany and Dr. E. F. Codd of the IBM Research Laboratory at San Jose for their suggestions. We gratefully acknowledge the many members of the Computer Science Department at the IBM Research Laboratory at San Jose who have contributed their ideas, programming, and critique to our work.

CITED REFERENCES

1. E. F. Codd, "A relational model for data for large shared data banks," *Communications of the ACM* **13,** No. 6, 377–397 (June 1970).
2. M. W. Blasgen and K. P. Eswaran, *On the Evaluation of Queries in a Data Base System*, IBM Research Report FJ 1945, IBM Research Laboratory, San Jose, California 95193 (April, 1976).
3. J. Mylopoulos, et al., "A multi-level relational system," *Proceedings of the AFIPS National Computer Conference, May 1975* **44,** 403–408, AFIPS Press Montvale, New Jersey (1975).
4. S. J. P. Todd, "The Peterlee Relational Test Vehicle—a system overview," *IBM Systems Journal* **15,** No. 4, 285–308 (1976).
5. M. M. Astrahan and D. D. Chamberlin, "Implementation of structured English query language," *Communications of the ACM* **18,** No. 10, 580–588 (October 1975).
6. G. D. Held, et al., "INGRES: A relational data base system," *Proceedings of the National Computer Conference, Anaheim, California, May 1975*, 409–416, AFIPS Press, Montvale, New Jersey (1975).
7. L. Gotlieb, "Computing joins of relations," *Proceedings of the ACM-SIGMOD Conference*, San Jose, California, May 1975, 53–63, Association for Computing Machinery, New York, New York (1975).
8. R. M. Pecherer, "Efficiency evaluation of expressions in a relational algebra," *Proceedings of the ACM Pacific 75 Regional Conference, April 1975*, 44–49, Association for Computing Machinery, New York, New York (1975).
9. M. M. Astrahan et al., "System R: A relational approach to data base management," *ACM Transactions on Data Base Management* **1,** No. 2, 97–137 (June 1976).
10. *Planning for Enhanced VSAM under OS/VS*, Form No. GC26-3842, IBM Corporation, Data Processing Division, White Plains, New York 10604 (1975).
11. R. Bayer and E. M. McCreight, "Organization and maintanence of large ordered indexes," *ACTA Informatica* **1,** No. 3, 173–189 (1972).
12. D. Knuth, *The Art of Computer Programming*, Volume 3, Addison-Wesley, Reading, Massachusetts.
13. *Information Management System, General Information Manual*, Form GH20-0765, IBM Corporation, Data Processing Division, White Plains, New York 10604.
14. *CODASYL Data Base Task Group Report*, April 1971, available from the Association for Computing Machinery, New York, New York.

GENERAL REFERENCE

1. D. D. Chamberlin, "Relational data-base management system," *Computing Surveys* **8,** No. 1, 43–66 (March 1976).

Reprint Order No. G321-5058.