Index Selection in Relational Databases

Kyu-Young Whang

IBM T.J. Watson Research Center, P.O. Box 218, Yorktown Heights, NY 10598

ABSTRACT: An index selection algorithm for relational databases is presented. The problem concerns finding an optimal set of indexes that minimizes the average cost of processing transactions. This cost is measured in terms of the number of I/O accesses. The algorithm presented employs a heuristic approach called *DROP heuristic*. In an extensive test to determine the optimality of the algorithm, it found optimal solutions in all cases. The time complexity of the algorithm shows a substantial improvement when compared with the approach of exhaustively searching through all possible alternatives. This algorithm is further extended to incorporate the clustering property (the relation is stored in a sorted order) and also is extended for application to multiple-file databases.

1. Introduction

We consider the problem of selecting a set of indexes that minimizes the transaction-processing cost in relational databases. The cost of a transaction[†] is measured in terms of the number of I/O accesses.

The index selection problem has been studied extensively by many researchers. A pioneering work based on a simple cost model appeared in [1]. A more detailed model incorporating index storage cost as well as retrieval and index maintenance cost was developed in [2]. Some approaches [3], [4] attempted to formalize the problem to obtain analytic results in certain restricted cases. In a more theoretical approach, Comer [5] proved that a simplified version of the index selection problem is NP-complete. Thus, the best existing algorithm for finding an optimal solution would have an exponential time complexity. In an effort to devise a more efficient algorithm, Schkolnick [6] discovered that, if the cost function satisfies a property called *regularity*, the complexity of the optimal index-selection algorithm can be reduced to one that is less than exponential. Hammer and Chan [7] took a somewhat different approach and developed a heuristic algorithm that drastically reduced the time complexity. However, no attempt has yet been made to establish the validity of this algorithm.

Although considerable effort has been devoted to developing algorithms for index selection, most past research has concentrated on single-file cases. Furthermore, the problem of how to incorporate the primary structure (the clustering property) of a file is still awaiting a solution. The purpose of this paper is to introduce an index selection algorithm with a reasonable efficiency and accuracy, which is subsequently extended to include multiple-file cases as well as to incorporate the clustering property.

The approach presented in this paper bears some resemblance to the one introduced by Hammer and Chan [7]; but there is one major modification: the DROP heuristic [8] is employed instead of the ADD heuristic [9]. The DROP heuristic attempts to obtain an optimal solution by dropping indexes incrementally, starting from a full index set. On the other hand, the ADD heuristic adds indexes incrementally, starting from an initial configuration without any index, to reach an optimal solution.

Since we are pursuing a heuristic approach for index selection, the actual result is suboptimal. However, in an extensive test performed for validation, the DROP heuristic found optimal solutions in all cases. (On the other hand, the ADD heuristic found suboptimal solutions on several occasions.)

In determining optimality, we consider only the cost of accessing and maintaining the database and indexes; i.e., the cost of storing indexes in the storage medium is not considered. If desired, however, the index storage cost can be incorporated by making it part of the index maintenance cost (or, equivalently, the two costs can be combined to constitute the *overhead cost* as defined in [2]), so that it can fit in the framework of the algorithms to be presented. Thus, the general validity of our approach should remain intact.

[†] The term *transaction* used here should not be confused with the atomic transaction as a unit of consistency and recovery. Here, it is used as a generic term for both queries and update activities against the database.

We first present the index selection algorithm for single-file databases without the clustering property. This algorithm is tested for validation with 24 randomly generated input situations, and the result compared with the optimal solutions generated by searching exhaustively through all possible index sets. The algorithm is then extended to incorporate the clustering property. Extension to multiple-file cases is considered subsequently.

Section 2 introduces assumptions regarding the cost model and a simple index structure, while Section 3 describes the classes of transactions we consider and their cost formulas. Next, in Section 4, we present the index selection algorithm and its time complexity. In Section 5, we discuss the result of the test performed for the validation of the algorithm. Extension of the algorithm to incorporate the clustering property is the principal topic of Section 6. Finally, in Section 7, we discuss an extension of the algorithm for application to multiple-file databases.

2. Assumptions

We assume that the relation is stored in a secondary storage medium, which is divided into fixed-size units called blocks [10]. For simplicity, we assume that a relation is mapped into a single file, an attribute to a column, and a tuple to a record. Accordingly, we shall use the terms *file* and *relation* interchangeably.

In processing a transaction the number of I/O accesses necessary to bring the blocks into the main memory depends on the specific buffer strategy. We assume, however, the following simple strategy: no block access will be necessary if the next tuple (or index entry) to be accessed resides the the same block as that of the current tuple (or index entry); otherwise, a new block access is necessary. We also assume that all TID (tuple identifier) manipulations can be performed in the main memory without any need for I/O accesses.

We consider only conjunctive predicates consisting of simple equality predicates (e.g., Column A = 'a'). The selectivity of each simple predicate is estimated as the inverse of the corresponding column cardinality (the number of unique values in a column). If a predicate is a conjunction of simple predicates, its selectivity is obtained by multiplying the individual selectivities of those simple predicates. More general predicates can be incorporated if a more elaborate scheme for estimating selectivity [11] is employed.

We assume that a B^+ -tree index [12] can be defined for a column of a relation. The leaf level of the index consists of key-TID pairs for every tuple in that relation and the leaf-level blocks are chained so that the index can be scanned without traversing the index tree. Entries having the same key value are ordered by TID. When index entries are inserted or deleted, we assume that splits or concatenations of index blocks are rather infrequent, so that modifications are done mainly on leaf-level blocks.

A simple model of the storage structure has been chosen for the purpose of illustration and implementation. Although detailed models must be dependent on different implementations, we believe it is unlikely that the validity of the algorithms to be presented is influenced by these variations. In support of this argument, the validity of the DROP heuristic (in a form slightly different than presented here) has been well established in reference [8] in relation with the Warehouse Location Problem which involves a significantly different cost model.

3. Transaction Model

We consider four types of transactions: query, update, deletion, and insertion. The classes subsumed under these types are shown in Figures 1 to 4.

```
SELECT  <list of columns>
FROM    R
WHERE   P
```

Figure 1. General class of queries considered.

UPDATE R
SET R.A = < new value_A >,
SET R.B = < new value_B >,
.
.
.
WHERE P

Figure 2. General class of update transactions considered.

```
DELETE  R
WHERE   P
```

Figure 3. General class of deletion transactions considered.

INSERT INTO R: <list of column values>

Figure 4. General class of insertion transactions considered.

In figures 1 to 4, "P" stands for the restriction predicate that selects the relevant tuples. We call the columns appearing in P *restriction columns.*

Cost formulas for those transactions are now introduced in the form of functions. In calculating the cost of a query we do not include the cost of writing the result, since that cost is independent of the index set and accordingly irrelevant for optimization purposes. We also assume that, in resolving predicates, all the available indexes are utilized even if some of them might increase the processing cost because of their own access cost.

We define the following notation:

C	A column.
n	Number of tuples in the relation (cardinality).
p	Blocking factor of the relation.
L_C	Blocking factor of the index for column C.
F_C	Selectivity of Column C or of its index.
m	Number of blocks in the relation, which is equal to n/p.
t	A transaction
Restricted Set	Set of tuples that satisfy all the restriction predicates. Equivalent to $(\prod_{C \in \{\text{all restriction columns}\}} F_C) \times n.$
Partially Restricted Set	Set of tuples that satisfy the restriction predicates that can be resolved via indexes. Equivalent to $(\prod_{C \in \{\text{all restriction columns having indexes}\}} F_C) \times n.$

- **function b(m,p,k):** cost of accessing k randomly selected tuples in TID order

$$b(m,p,k) = m\left[1 - \binom{n-p}{k} / \binom{n}{k}\right] \qquad (1)$$
$$= m\left[1 - ((n-p)!(n-k)!)/((n-p-k)!n!)\right]$$
$$= m\left[1 - \prod_{i=1}^{k} (n-p-i+1)/(n-i+1)\right]$$

when $k \leq n-p$, and

$b(m,p,k) = m$

when $k > n-p$.

The function is approximately linear when $k << n$ and approaches m as k becomes large. Equation 1 is an exact formula derived by Yao [13]. Variations of this function and approximation formulas for faster evaluation are summarized in [14].

- function IA(C,mode): cost of accessing a B^+ -tree index from the root

  A. mode = Query mode
  $$IA = \lceil \log_{L_C} n \rceil + (\lceil F_C \times n/L_C \rceil - 1) \qquad (2)$$

  B. mode = Insertion mode
  $$IA = \lceil \log_{L_C} n \rceil$$

  C. mode = Update mode
  $$IA = \lceil \log_{L_C} n \rceil + (\lceil 0.5 \times F_C \times n/L_C \rceil - 1)$$

The function IA has three modes, depending on the purpose of accessing the index. In query mode, all the index entries with the same key value are retrieved. The first term in Equation 2 is the height of the index tree, while the second is the number of leaf-level index blocks accessed. One block access is subtracted from the second term since the cost of accessing the first leaf node is already included in the first term. In insertion mode, an index entry corresponding to the inserted tuple is placed after the last entry that has the same key value. Thus, only one leaf-level block will be accessed. This cost, however, is included in the first term. In update mode, the index entries containing the old value have to be searched to find the one with the TID of the updated tuple; thus, on the average, about half of the index entries will be searched.

- function Query(t): cost of processing a query

$$\text{Query} = b(m,p,|\text{partially restricted set}|) + \sum_{C \in \{\text{all restriction columns having indexes}\}} IA(C,\text{query mode}) \qquad (3)$$

Queries are processed as follows. Indexes of all restriction columns are accessed in query mode to obtain the sets of TIDs that satisfy the corresponding simple restriction predicates. The intersection of these TID sets is formed subsequently to locate tuples in the partially restricted set. These tuples are retrieved and produced as output after the remaining restriction predicates are resolved. The first term in Equation 3 represents the cost of accessing data tuples, the second the cost of accessing indexes.

- function Update(t): cost of processing an update transaction

$$\text{Update} = \text{Query}(t) + b(m,p,|\text{restricted set}|) + |\text{restricted set}| \times 2 \times \sum_{C \in \{\text{all updated columns having indexes}\}} [\text{IA}(C,\text{update mode}) + 1] \quad (4)$$

The update cost consists of three parts: the first term of Equation 4 represents the cost of reading in blocks containing the tuples to be deleted, the second term the cost of writing out modified blocks, and the third term the cost of updating corresponding indexes. The third term is again divided into two parts: the cost of deleting index entries for old values and that of inserting index entries for new values. Since these two parts have the same value, a factor of two is introduced. In either part, one block access is added for each index entry modified to account for writing out the updated block. Let us note that, even for insertion of new index entries, the update mode is specified for function IA because index entries having the same key value must be ordered according to their TIDs.

- function Delete(t): cost of processing a deletion transaction

$$\text{Delete} = \text{Query}(t) + b(m,p,|\text{restricted set}|) + |\text{restricted set}| \times \sum_{C \in \{\text{all columns having indexes}\}} [\text{IA}(C,\text{update mode}) + 1]$$

The deletion cost is the same as the update cost except that the third term of the cost function represents the cost of deleting index entries for all existing indexes.

- function Insert(t,Ntuples_inserted): cost of processing an insertion transaction

$$\text{Insert} = \text{Ntuples\_inserted} \times (1 + 1 + \sum_{C \in \{\text{all columns having indexes}\}} [\text{IA}(C,\text{insertion mode}) + 1])$$

Three different mechanisms contribute to the insertion cost: locating the place to insert a new tuple (one I/O access); writing out the modified block access (one I/O access); and modifying all existing indexes accordingly. In the third, function IA is called in insertion mode since the new index entry is always added at the end of the list of index entries that have the same key value.

4. Index Selection Algorithm (DROP Heuristic)

Input:

- Usage information: A set of various query, update, insertion, and deletion transactions with their relative frequencies.
- Data characteristics: Relation cardinality, blocking factor, selectivities, and index blocking factors of all columns.

Output:

- A near-optimal index set.

Algorithm 1:

1. Start with a full index set.
2. Try to drop one index at a time and, applying the cost evaluator, obtain the total transaction-processing cost to find the index that yields the maximum cost benefit when dropped.
3. Drop that index.
4. Repeat Steps 2 and 3 until there is no further reduction in the cost.
5. Try to drop two indexes at a time and, applying the cost evaluator, obtain the total transaction-processing cost to find the index pair that yields the maximum cost benefit when dropped.
6. Drop that pair.
7. Repeat Steps 5 and 6 until there is no further reduction in the cost.
8. Repeat Steps 5, 6, and 7 with three indexes, four indexes, ..., (up to k+1 indexes at a time) until there is no further improvement.

The variable k is the maximum number of indexes that produce incremental cost benefits when dropped together at a time. (Dropping k+1 indexes at a time provides no improvement; thus, the algorithm will terminate.) We need to consider dropping more than one index together because the presence of an index may have influence on the benefit of having other indexes. This interaction occurs when more than one column appears in the predicate specified in a transaction. For example, in processing a query having a predicate, (Column A = 'a' AND Column B = 'b'), the selectivity of either conjunct may not significantly reduce the number of blocks to be accessed; nevertheless, the joint selectivity of the two might. Let us note that this situation can occur because of nonlinearity of function b(m,p,k), which returns the number of blocks to be accessed to retrieve a given number of tuples. When this happens, dropping either index alone may cause a heavy penalty in cost because the other index alone is not very useful, whereas dropping both together may produce benefit because maintenance costs of both indexes are eliminated for the same amount of loss in benefit.

Intuitively, if only one predicate is involved, the set of indexes that must be considered together are those specified together in a predicate. In a typical input situation, however, there are many predicates with different combinations of columns; thus, the presence of an index can affect process of dropping other indexes indirectly through more than one predicate. For example, suppose an index for Column A has an influence on an index for Column B via predicate P_1, while the index for Column B has an influence on an index for Column C via predicate P_2. Then, naturally, the index for Column A has an influence on the index for Column C indirectly via predicates P_1 and P_2. Thus, in principle, the variable k may have any value from one to the total number of columns in the relation. Nevertheless, we believe that, in most practical cases, k=2 is sufficient to find optimal or near-optimal solutions. In fact, in all the tests performed to validate the index selection algorithms, the maximum value of k we encountered was 2.

The time complexity of the algorithm is $O(g \times v^{k+1})$, where g is the number of transactions specified in the usage information, v the number of columns in the relation, and k the maximum number of columns considered together in the algorithm. The time complexity is estimated in terms of the number of calls to the cost evaluator, which is the costliest operation in the design process. In the algorithm, the cost evaluator is called for every k-combination of columns of the relation and for every transaction in the usage information. This contributes the order of $g \times v^k$. The procedure is repeated until there is no further reduction in the cost. Since the number of iterations is proportional to v, the overall time complexity is $O(g \times v^{k+1})$.

5. Validation of the Algorithm

An important task in developing heuristic algorithms is their validation. In this section the result of an extensive test performed to validate the index selection algorithm (DROP heuristic) will be presented. In particular, we try to measure the deviations of the heuristic solutions from the optimal ones for various input situations generated by using different parameters. (These parameters are chosen from the ranges that are important in practical applications.) For a relation with many columns, it is a difficult, often impossible, task to identify the optimal solution itself. Therefore, the number of columns in a relation is restricted to ten in the sets. Optimal solutions are then obtained by searching exhaustively through all possible alternatives (2^{10} combinations).

The input situations are generated as follows:

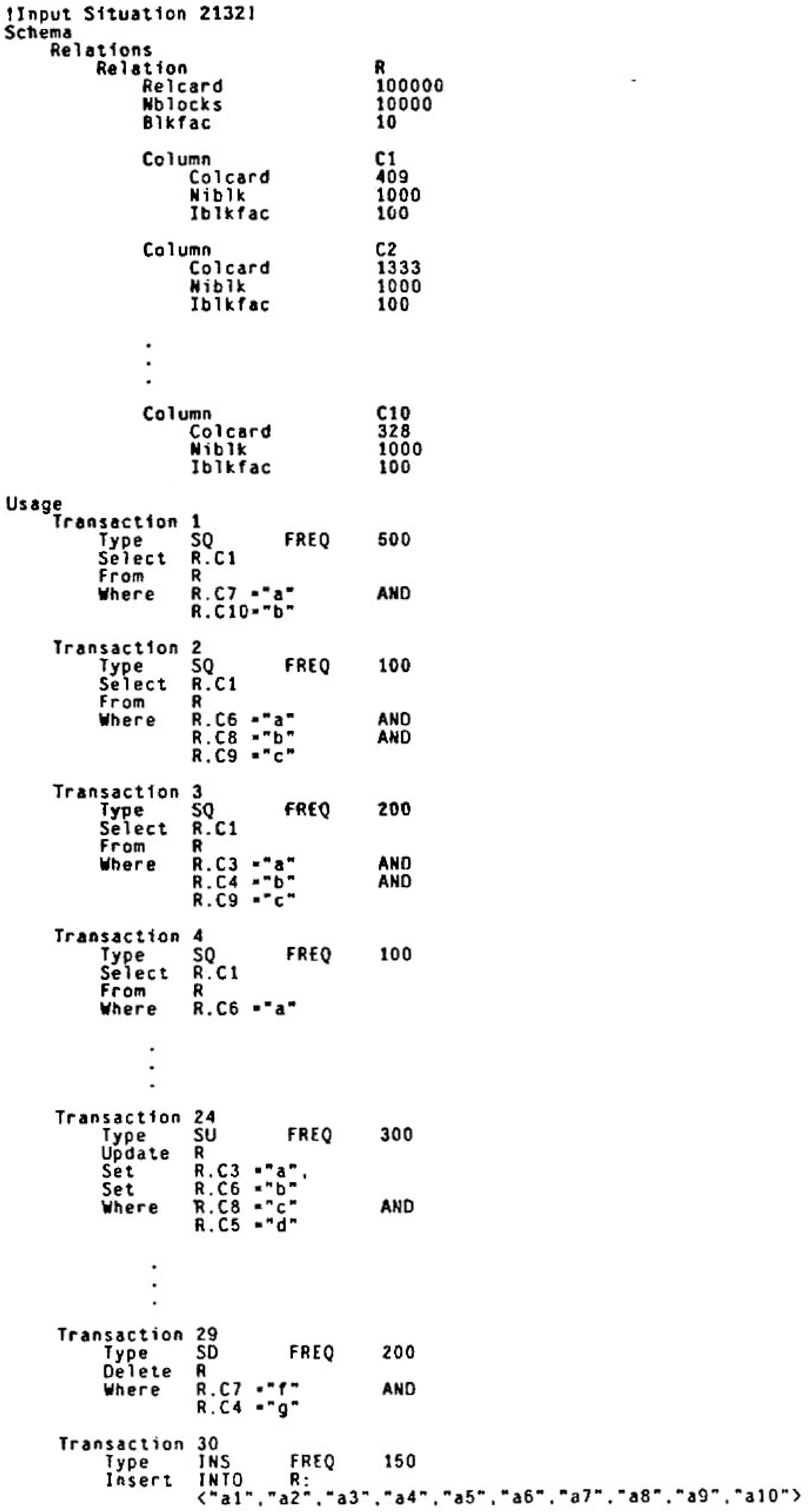
1) Two sets of the relation cardinality and column cardinalities are used: in the first set the relation cardinality is 1000; in the second it is 100,000. The column cardinalities are randomly generated between 1 and the relation cardinality, with a logarithmically uniform distribution.

2) Two sets of blocking factor and index blocking factors are used: 1) 10 and 100; 2) 100 and 1000. The index blocking factors are assumed to be identical for all columns in a particular input situation.

3) An input situation includes 30 transactions and their relative frequencies. Among them are 21 queries, 4 to 5 update transactions, 3 to 4 deletion transactions, and 1 insertion transaction. Using this template, three different sets of transactions are created to provide different mixture of transactions. For each set, transactions are randomly generated as follows: for queries and deletion transactions, 1 to 3 (this number is randomly selected for each transaction) columns are randomly selected as restriction columns; for update transactions, 1 to 3 (this number is also randomly selected) columns are randomly selected as updated columns, and another randomly selected set of columns as restriction columns.

4) Two sets of relative frequencies are used. In the first set all transactions initially have identical frequencies. Later, the frequencies of deletion and insertion transactions are multiplied by an adjustment factor to keep the number of indexes in the result between 3 and 7. This adjustment is made to avoid extreme cases in which a full index set or an empty index set is the optimal solution. For the second, the relative frequencies of transactions are randomly generated between 100 and 500, with an interval of 50 between adjacent values.

The scheme described above generates 24 different input situations with varying statistics of the database and usage information as well as random mixtures of transactions. An example situation is shown in Figure 5 in an abbreviated form. The test results for both DROP and ADD heuristics are summarized in Table 1. In the first column of Table 1, the first digit of the input situation number represents the set of the relational cardinality, the second the set of the main-file blocking factor and the index blocking factor, the third the set of transactions, and the last the set of relative frequencies of transactions. The second column of the table shows the number of indexes present in the optimal solution. The CPU time shows the performance of the algorithms when run on a DECSYSTEM-2060. Percentage deviations are shown for the situations in which any deviation occurred. Marked by "opt" are the situations in which optimal solutions were found.



```
!Input Situation 2132!
Schema
    Relations
        Relation                R
            Relcard             100000
            Nblocks             10000
            Blkfac              10

            Column              C1
                Colcard         409
                Niblk           1000
                Iblkfac         100

            Column              C2
                Colcard         1333
                Niblk           1000
                Iblkfac         100

            .
            .
            .

            Column              C10
                Colcard         328
                Niblk           1000
                Iblkfac         100

Usage
    Transaction 1
        Type    SQ      FREQ    500
        Select  R.C1
        From    R
        Where   R.C7 ="a"       AND
                R.C10="b"

    Transaction 2
        Type    SQ      FREQ    100
        Select  R.C1
        From    R
        Where   R.C6 ="a"       AND
                R.C8 ="b"       AND
                R.C9 ="c"

    Transaction 3
        Type    SQ      FREQ    200
        Select  R.C1
        From    R
        Where   R.C3 ="a"       AND
                R.C4 ="b"       AND
                R.C9 ="c"

    Transaction 4
        Type    SQ      FREQ    100
        Select  R.C1
        From    R
        Where   R.C6 ="a"

        .
        .
        .

    Transaction 24
        Type    SU      FREQ    300
        Update  R
        Set     R.C3 ="a",
        Set     R.C6 ="b"
        Where   R.C8 ="c"       AND
                R.C5 ="d"

        .
        .
        .

    Transaction 29
        Type    SD      FREQ    200
        Delete  R
        Where   R.C7 ="f"       AND
                R.C4 ="g"

    Transaction 30
        Type    INS     FREQ    150
        Insert  INTO    R:
                <"a1","a2","a3","a4","a5","a6","a7","a8","a9","a10">
```

Figure 5. An input situation.

Table 1. Accuracy and Performance of the Index Selection Algorithm.

Input Situation	Number of Indexes	CPU time(seconds) / Deviation(%)				
		DROP Heuristic		ADD Heuristic		Ex. Search
1111	7	2.3	opt	2.0	0.21	36
1112	5	2.2	opt	2.1	opt	36
1121	6	2.4	opt	2.0	1.23	37
1122	6	2.3	opt	2.1	1.17	37
1131	6	2.5	opt	2.1	opt	39
1132	7	2.5	opt	2.1	1.17	39
1211	5	3.1	opt	1.7	opt	32
1212	3	1.9	opt	1.6	opt	31
1221	4	2.1	opt	1.7	opt	32
1222	5	2.0	opt	1.7	opt	32
1231	4	2.3	opt	1.7	opt	35
1232	5	2.2	opt	1.7	opt	35
2111	4	2.4	opt	2.1	16.71	39
2112	5	2.5	opt	2.1	21.17	40
2121	6	2.3	opt	2.0	opt	38
2122	7	2.5	opt	2.0	opt	37
2131	6	2.6	opt	2.2	opt	40
2132	6	2.7	opt	2.2	opt	40
2211	6	2.6	opt	2.0	opt	36
2212	4	2.4	opt	1.9	opt	36
2221	6	2.4	opt	1.9	opt	34
2222	6	2.3	opt	1.9	opt	34
2231	5	2.4	opt	2.0	opt	38
2232	5	2.4	opt	2.0	opt	38

In all situations tested, the DROP heuristic found optimal solutions. Although the test is by no means exhaustive, the result is a good indication that the DROP heuristic will perform well in many practical situations. In comparison, the ADD heuristic produced suboptimal solutions in six cases; the maximum deviation encountered was 21.17%.

The reason why the ADD heuristic does not perform as well as the DROP heuristic is the following. With the ADD heuristic, a potentially most important index is selected first. Since the presence of an index affects selection of other indexes in such a way as to maximize its benefit, selection of the first index is tantamount to dictating the solution finally to be generated. Thus, a possible suboptimal decision made at the beginning may well persist in the result. On the other hand, in the DROP heuristic, the least significant index is dropped first. Accordingly, even if the decision made at the beginning is not optimal, it is unlikely that the resulting solution is significantly affected. Example 1 illustrates this point further.

Example 1: Figure 6 shows intermediate index sets at each step of the ADD heuristic and the DROP heuristic. For convenience, we chose a relation with only four columns. The symbol '1' in the figure indicates the presence of an index, and 'X' its absence. Only the first two iterations of the design process are shown since there is no more improvement in the third.

In this example, the DROP heuristic found the optimal solution. The ADD heuristic, however, resulted in a slight deviation from the optimal. Compared with the optimal solution, the solution that the ADD heuristic produced has an index on Column 1, but lacks one on column 4. Column 1 was assigned an index during the first iteration because the index set (1 X X X) was less costly than (X X X 1); this index subsequently stayed until the algorithm terminated. The index on Column 1 is absent in the optimal solution, however, since the presence of indexes on columns 3 and 4 renders it less significant than it would be without them. □

Algorithm	ADD heuristic				DROP heuristic			
Column	1	2	3	4	1	2	3	4
Initial	X	X	X	X	1	1	1	1
Iteration 1	1	X	X	X	1	X	1	1
Iteration 2	1	X	1	X	X	X	1	1

Figure 6. Intermediate index sets during a design process.

This example shows the error caused by selecting an insignificant index that looks as if it were important at the initial stage of the design using the ADD heuristic because the interaction among indexes on different columns has not been well established. In contrast, since all indexes are initially present in the DROP heuristic, the influence of an index on others is taken into account from the beginning, thereby reducing the probability of reaching an incorrect solution.

As we can see in Table 1, an exhaustive search takes excessive computation time; in comparison, the DROP heuristic is far more efficient without significant loss of accuracy. Obviously, for larger input situations, the exhaustive-search method will become prohibitively time-consuming. In these cases, heuristic algorithms such as the DROP heuristic may be the only ones applicable.

6. Index Selection when there is a Clustering Column

In this section we extend the index selection algorithm to incorporate the clustering property. We present two algorithms for this extension.

Algorithm 2:

1. For each possible clustering column in the relation, perform index selection.
2. Save the best configuration.

Algorithm 3:

1. Perform index selection with the clustering column determined in Step 2 of the last iteration. (During the first iteration it is assumed that there is no clustering column.)
2. Perform clustering design with the index set determined in Step 1. The clustering property is assigned to each column in turn, and then the best clustering column is selected.
3. Steps 1 and 2 are iterated until the improvement in cost through one loop cycle is less than a predefined value (e.g., 1%).

Algorithm 2 is a pseudoenumeration since index selection is repeated for every possible clustering-column position. Naturally, Algorithm 2 has a higher time complexity than Algorithm 3, but has a better chance of finding an optimal solution. Both algorithms have been implemented and tested as a part of Physical Database Design Optimizer–an experimental system for developing various heuristics for the multiple-file physical database design [15]. In most cases tested the algorithms found optimal solutions. (The validation of these algorithms is combined with those of Algorithms 4, 5, and 6 in Section 7.) Let us note that the cost formulas have to be modified to take the clustering column into account. A complete set of cost formulas for multiple-file relational databases with the clustering property can be found in [16].

7. Index Selection for Multiple-File Databases

We present, in this section, an extension of the index selection algorithm for application to multiple-file databases. The extended algorithm (Algorithm 4) is almost identical to Algorithm 1 except for the following:

1) The entire database is designed all together. This is done by treating all columns in the database uniformly, as if they were all in a single relation.

2) Clustering columns are incorporated by a technique similar to the one employed in Algorithm 3. In addition, multiple clustering columns are allowed with the restriction that at most one can be assigned to a relation. Accordingly, clustering design is repeated until as many clustering columns as are beneficial in reducing the overall cost are assigned to the database.

Let us note that, when considering a transaction involving more than one relation, the optimizer [17],[18] has to be invoked to find the optimal sequence of access operations as well as to determine the cost of evaluating the transaction.

Algorithm 4 has also been implemented and tested as a part of the Physical Database Design Optimizer. For the purpose of comparison, we now briefly introduce other multiple-file physical database design algorithms (Algorithms 5 and 6), which are based on the property of separability. The results of the tests for their validation are then compared with those for Algorithm 4.

The separability approach was proposed by Whang, Wiederhold, and Sagalowicz in [19] and, subsequently, in [20] and [21]. Other work based on this approach appeared in [22]. The separability-based approach enables the physical design of the entire database to be performed relation by relation independently of one another (we call this phase of design *Phase 1*)—if certain conditions are met. Features that violate these conditions can be incorporated by adding an adjustment step (which we call *Phase 2*) during each iteration. Thus, Phase 1 of Algorithm 5 is identical to Algorithm 2, and that of Algorithm 6 to Algorithm 3. The description of Phase 2, however, is beyond the scope of this paper and will not be discussed further. Interested readers are referred to the reference [15].

The three multiple-file design algorithms were tested with 21 different input situations. The input situations tested consisted of seven schemas, each of which was accompanied by three variations of usage specification generated as follows. First, the transactions and their frequencies were defined in such a way that by intuition they looked most "natural". Second, according to the test results from the first usage specification, the frequencies were modified so that the costs of individual transactions were of the same order. This modification prevented a few most costly transactions from dominating the results of the design. Third, all the queries were eliminated from the usage specification leaving only update, insertion, and deletion transactions. This modification simulated a situation in which there was a high frequency of updates.

The test schemas were selected with various statistics. Among them, four of them were arbitrarily chosen, while the remaining three were extracted from the Ships-Monitoring-Database–a research vehicle for the Knowledge-Base Management Systems (KBMS) Project [23], [24] at Stanford University. Two schemas were defined as small subsets with the third encompassing the entire KBMS database. The skeleton of the KBMS schema is shown in Figure 7, using the notation defined in the Structural Model [25]. In Figure 7, the symbol --> represents a many-to-one relationship between relations, and --* a one-to-many relationship with different structural constraints.

The results of the tests thus obtained are summarized in Table 2. In the first column the first digit of the input situation number represents the schema, and the second the usage input. In the description, r stands for the number of relations, c the number of columns in the database, and t the number of transactions in the usage input. The CPU time shows the performance of the algorithms when run in a DECSYSTEM-2060. Marked by "*" are the situations in which any deviation occurred.

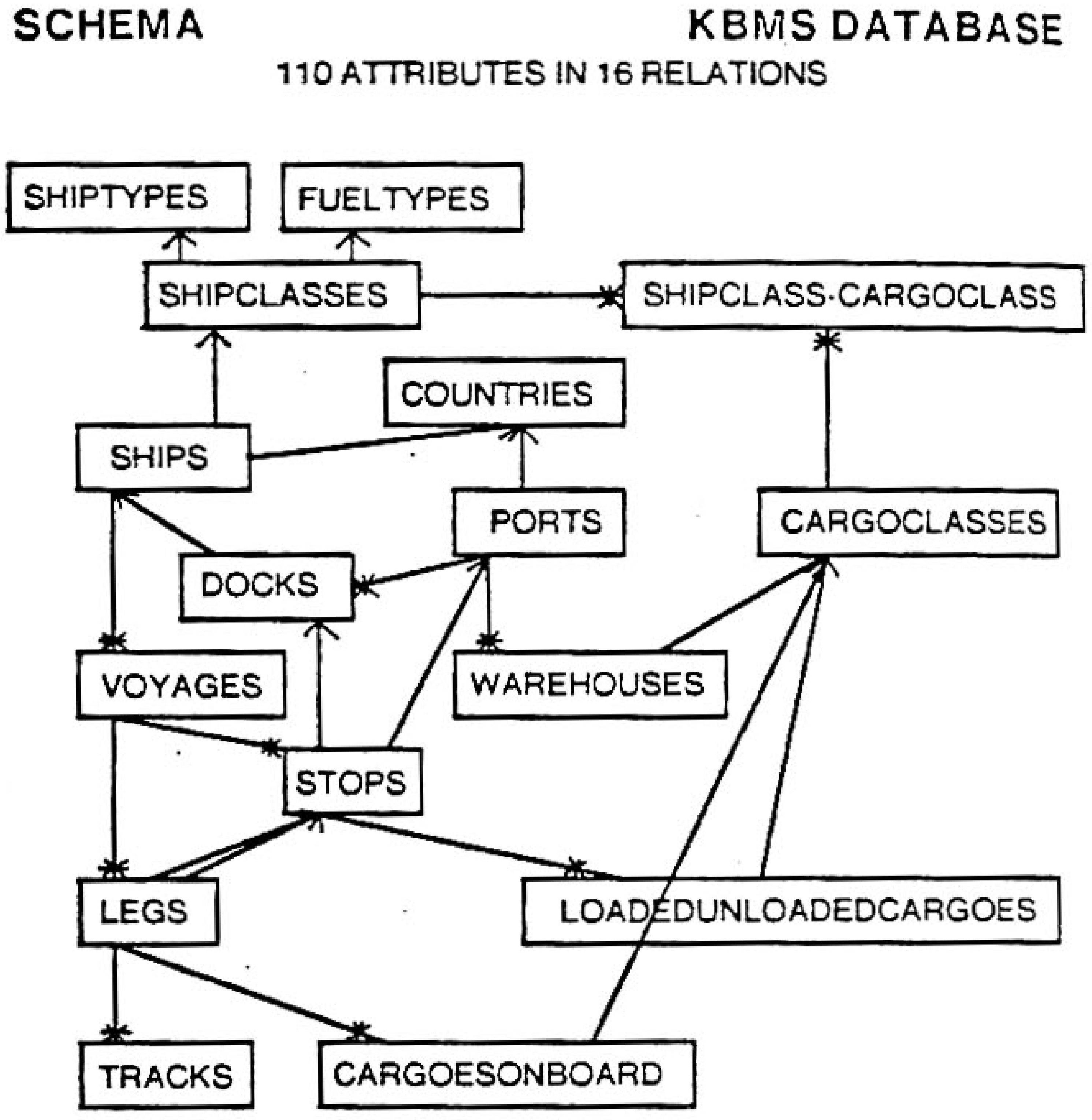


Figure 7. The KBMS schema.

Table 2. Accuracy and Performance of Multiple-File Design Algorithms

Input Situation	Description	CPU time(s:seconds;m:minutes;h:hours;y:years)						
		Algorithm4		Algorithm5		Algorithm6		Ex.Search
10	2r, 6c, 7t	1.83s		1.25s		0.86s		26.91s
20	4r, 9c,10t	5.41s		1.48s		1.23s		36.75m
30	4r, 12c,12t	10.51s		3.44s		2.09s		13.93h
40	4r, 11c,13t	6.62s		2.73s		2.04s		3.65h
50	5r, 12c,12t	12.51s		4.89s		2.32s	•	25.85h
60	4r, 11c,15t	13.93s		3.54s		2.63s		8.52h†
70	16r,110c,81t	2.00h		4.80m		1.63m		10^{35} y†
11	2r, 6c, 7t	1.81s		1.26s		0.84s		26.46s
21	4r, 9c,10t	5.91s		1.67s		1.35s		42.83m
31	4r, 12c,12t	10.67s		3.43s		2.17s		14.00h
41	4r, 11c,13t	9.90s		1.88s		1.42s		3.62h
51	6r, 12c,12t	13.13s		5.00s		3.54s		26.63h
61	4r, 11c,15t	21.51s		3.74s		2.71s		8.04h†
71	16r,110c,81t	2.02h		4.60m		2.13m		10^{35} y†
12	2r, 6c, 5t	1.23s		0.86s		0.57s		17.23s
22	4r, 9c, 5t	1.50s		0.55s		0.43s		10.43m
32	4r, 12c, 6t	4.65s		1.73s		1.08s		6.95h
42	4r, 11c, 6t	0.95s	•	0.43s		0.25s	•	29.95m
52	6r, 12c, 8t	6.04s		2.41s		1.49s		9.95h
62	4r, 11c, 6t	3.72s		1.81s		1.23s		2.12h†
72	16r,110c,38t	24.40m		1.77m		21.76s		10^{35} y†

† Values are estimated.
• Situations that produced nonoptimal solutions.

Optimal solutions were obtained by running the exhaustive-search algorithm. For Situations 70, 71, and 72, where exhaustive search was nearly impossible, however, the results of three design algorithms were compared; if they produced the same result, it was considered to be optimal. According to this criterion, in most situations tested, all three algorithms produced optimal solutions. Even in the situations that produced suboptimal solutions, the deviations were far from being significant. (Algorithm 4 yielded 6.6% in Situation 42; Algorithm 1 yielded 3.1% of deviation in Situation 50 and 6.6% in Situation 42. These situations are fully analyzed in [15].)

As we can see in Table 2, Algorithm 4 performs well with reasonable efficiency. Compared with the exhaustive search algorithm, it takes a negligible amount of time to complete the design without a significant loss of accuracy. For a very large database (for example, one consisting of 250 relations and 5000 columns), however, even Algorithm 4 can become intolerably time-consuming. In these cases, Algorithms 5 and 6, which are based on the separability property, are the only algorithms applicable. Indeed, when a very large database is involved, the entire database design somehow has to be partitioned to achieve a reasonable performance in the design process.

Nevertheless, Algorithm 4 has its own advantages. Because it does not require that the database management system satisfy the conditions for separability, it can be easily implemented on top of any relational system that supports indexes and clustering columns, although, for the other algorithms, the system should satisfy these conditions as closely as possible to achieve better accuracy, and any violations of the conditions should be explicitly identified through analysis. Besides, it is worth mentioning once again that the performance of Algorithm 4 falls into a practically feasible range, especially when small to moderate-sized databases are considered.

8. Summary and Conclusion

Algorithms for the optimal index selection in relational databases have been presented. Algorithm 1, which employs the DROP heuristic, has been introduced for single-file databases and compared with the ADD heuristic. In an extensive test performed for its validation, the DROP heuristic has found optimal solutions in all cases. In comparison, the ADD heuristic has found suboptimal solutions in several occasions.

The index selection algorithm using the DROP heuristic has been extended to incorporate the clustering property (Algorithms 2 and 3) and to include cases for multiple-file databases (Algorithm 4). Further, it has been embedded in multiple-file design algorithms based on the separability property (Algorithms 5 and 6). Algorithm 4 has subsequently been tested for validation and compared with Algorithms 5 and 6 as well as the exhaustive-search algorithm. The result shows that Algorithm 4 takes a negligible amount of time with no significant loss of accuracy in comparison with exhaustive search; furthermore, although it is not as fast as the other two algorithms, its performance is in a practically usable range.

Index selection has long been a subject of intensive research. Nevertheless, no successfully validated and reasonably efficient algorithm has been reported. We believe that our approach provides useful, easy-to-implement, and reliable algorithms for practical applications.

Acknowledgments

Most of the work presented in this paper was done as part of the author's dissertation research at Stanford University, Stanford, California. The research was supported by the Defense Advanced Research Project Agency under the KBMS project, Contract N39-82-C-0250.

The author wish to thank Ravi Krishnamurthy and Steve Morgan for reading earlier versions of this paper and providing many thoughtful comments.

References

[1] Lum, V.Y. and Ling, H., "An optimization problem of the selection of secondary keys," in *ACM Natl. Conf.*, ACM, 1971, pp.349-356.

[2] Anderson, H.D. and Berra, P.B., "Minimum cost selection of secondary indexes for formatted files," *ACM Trans. Database Systems,* Vol. 2, No. 1, March 1977, pp. 68-90.

[3] King, W.F., "On the selection of indices for a file," IBM Research Report RJ1341, IBM, San Jose, Calif., 1974.

[4] Stonebraker, M., "The choice of partial inversions and combined indices," *Intl. Journal of Computer Information Sciences.* Vol. 3, No. 2, 1974, pp.167-188.

[5] Comer, D., "The difficulty of optimum index selection," *ACM Trans. Database Systems,* Vol. 3, No. 4, Dec. 1978, pp.440-445.

[6] Schkolnick, M., "The optimal selection of secondary indices for files," *Information Systems,* Vol. 1, March 1975, pp.141-146.

[7] Hammer, M. and Chan, A., "Index selection in a self-adaptive database management system," *Proc. Intl. Conf. on Management of Data,* Washington, D.C., ACM SIGMOD, June 1976, pp. 1-8.

[8] Feldman, E., et al., "Warehouse location under continuous economies of scale," *Management Science,* Vol. 12, No. 9, July 1966, pp.670-684.

[9] Kuehn, A.A. and Hamburger, M.J., "A heuristic program for locating warehouses," *Management Science,* Vol. 10, July 1963, pp. 643-657.

[10] Wiederhold, G., *Database Design,* McGraw-Hill Book Company, New York, 1983.

[11] Demolombe, R., "Estimation of the number of tuples satisfying a query expressed in predicate calculus language," in *Proc. Intl. Conf on Very Large Databases,* Montreal, Canada, 1980, pp. 55-63.

[12] Comer, D., "The ubiquitous B-tree," *ACM Trans. Database Systems,* Vol. 11, No. 2, June 1979, pp.121-137.

[13] Yao, S.B., "Approximating block accesses in database organizations," *Commun. ACM,* Vol. 20, No. 4, 1977, pp. 260-261.

[14] Whang, K., Wiederhold, G., and Sagalowicz, D., "Estimating block accesses in database organizations–a closed noniterative formula," *Commun. ACM,* Vol. 26, No. 11, Nov. 1983, pp. 940-944.

[15] Whang, K., "A physical database design methodology using the property of separability," Ph.D. dissertation, Stanford University, Stanford, Calif., 1983, Rep. No. STAN-CS-83-968.

[16] Whang, K., "Transaction-processing costs in relational database systems," IBM Res. Rep. RC 10952, Jan. 1985.

[17] Selinger, P.G. et al., "Access path selection in a relational database management system," in *Proc. Intl. Conf. on Management of Data,* Boston, Mass., May 1979, pp.23-34.

[18] Kooi, R. and Frankforth, D., "Query optimization in INGRES," *IEEE Database Engineering Bulletin,* Vol. 5, No. 3, Sept. 1982, pp. 2-5.

[19] Whang, K., Wiederhold, G., and Sagalowicz,
"Separability—an approach to physical database desi
in *Proc. Intl. Conf. on Very Large Data Bases,* Can
France, IEEE, Sept. 1981, pp. 320-332.

[20] Whang, K., Wiederhold, G., and Sagalowicz,
"Separability–an approach to physical database desig
IEEE Transactions on Computers, Vol. C-33, No. 3, M
1984, pp. 209-222.

[21] Whang, K., Wiederhold, G., and Sagalowicz, D., "
property of separability in physical design of netw
model databases," *Information Systems.* Vol. 9, No.
1984.

[22] Bonfatti, F., Maio, D., and Tiberio, P., "A separabili
based method for secondary index selection in physi
database design," in *Methodology and tools for data b
design,* Chapter 6, North-Holland, 1983.

[23] Wiederhold, G., Kaplan, S.J., Sagalowicz, D., "Physi
database research at Stanford," *IEEE Database Engine
ing Bulletin,* Vol. 5, No. 1, Mar. 1982.

[24] Wiederhold, G. and Milton, J., "Knowledge-based ma
agement dystems project," *IEEE Database Engineeri
Bulletin,* Vol. 6, No. 4, Dec. 1983.

[25] Wiederhold, G. and El-Masri, R., "The Structural Moc
for database design," In *Proc. Intl. Conf. on Entity Re
tionship Approach,* Los Angeles, Calif., Dec. 1979, p
247-267.