Separability — An Approach to Physical Database Design

KYU-YOUNG WHANG, GIO WIEDERHOLD, AND DANIEL SAGALOWICZ, MEMBER, IEEE

Abstract — **A theoretical approach to the optimal design of a large multifile physical database is presented. The design algorithm is based on the theory that, given a set of join methods that satisfy a certain property called *separability*, the problem of optimal assignment of access structures to the whole database can be reduced to the subproblem of optimizing individual relations independently of one another. Coupling factors are defined to represent all the interactions among the relations. This approach not only reduces the complexity of the problem significantly, but also provides a better understanding of underlying mechanisms.**

Index Terms — **Block accesses, index selection, join methods, physical database design, query optimization, selectivity.**

I. INTRODUCTION

PROBLEMS of access path selection in large integrated databases can be approached from two standpoints. Query optimization seeks the optimal selection of access paths for a specific query being processed, given a certain structure of the underlying physical database [1]–[6]. On the other hand, design of a physical database is concerned with the optimal configuration of physical file and access structures, given the logical access paths that represent the interconnections among objects in the data model, the usage patterns of those paths, the organizational *characteristics of the data* stored in the files, the various features of the particular database management system (DBMS) [7]–[13]. Throughout this paper we use the term *access structures* as the features that a particular DBMS provides for the physical database design. For instance, access structures can be indexes, hashed organizations, links, clustering of records, etc. We also use the term *access configuration* to mean the aggregate of access structures assigned to a relation or to the whole database.

Most past research directed toward optimal design of physical databases has concentrated on single-file cases. This research must be extended to the design of the access configuration of multifile databases. Although some efforts have been devoted to multifile cases [7], [14], [15], the approaches employed fall far short of accomplishing automatic design of optimal physical databases.

A typical approach to the multifile physical database design has been to develop a cost evaluator that produces the total cost of processing queries and updates acting upon a specific access configuration. In this approach, however, selection of an optimal access configuration remained dependent on the designer's intuition or an exhaustive search through all possible configurations. Although an exhaustive search guarantees finding an optimal solution, it is practically impossible even with a small-sized database. This point is illustrated in Example 1.

Example 1: We look into a very simplified design process of a small database based on an exhaustive-search algorithm. We assume that the only access structure available is the clustering property. A column is said to have the *clustering property* if a relation is stored according to the order of the column values. Although the clustering property can be assigned to a combination of multiple columns, we assume for simplicity that it can be assigned only to a single column.

Using this access structure, for a given set of queries as input information, we want to find an optimal access configuration for the database consisting of relations R_1 and R_2, each of which owns two columns. We have nine possible access configurations as in Fig. 1, in which dashed lines show the position of the clustering column. The optimal access configuration can be found as follows:

1) For each of the nine configurations
   a) find the best join method for each query,
   b) obtain the total processing cost;
2) select the configuration that yields the minimum processing cost.

In this simple design example, we have only nine possible access configurations, but the number of access configurations is explosive if we have more relations, more columns in a relation, and various kinds of access structures. For instance, if we have five relations having five columns each, with indexes and the clustering property as available access structures, the number of possible access configurations becomes

$$(6 \times 6 \times 6 \times 6 \times 6) \times (2^5 \times 2^5 \times 2^5 \times 2^5 \times 2^5) = 2.6 \times 10^{11}$$

since we have six possible clustering columns (including the case in which the clustering column does not exist) and 2^5 possible index configurations for each relation. ■

As we see in Example 1, the cost of the exhaustive-search method becomes intolerably high even with a very small

Manuscript received November 1982; revised August 21, 1983. This work was supported by the Defense Advanced Research Project Agency under the KBMS Project, Contract N39-82-C-0250.

K.-Y. Whang was with the Computer Systems Laboratory, Departments of Electrical Engineering and Computer Science, Stanford University, Stanford, CA 94305. He is now with the IBM T. J. Watson Research Center, P. O. Box 218, Yorktown Heights, NY 10598.

G. Wiederhold is with the Computer Systems Laboratory, Departments of Electrical Engineering and Computer Science, Stanford University, Stanford, CA 94305.

D. Sagalowicz was with the Artificial Intelligence Center, SRI International, Menlo Park, CA 94025. He is now with Framentec, Monte Carlo, Monaco.

0018-9340/84/0300-0209$01.00

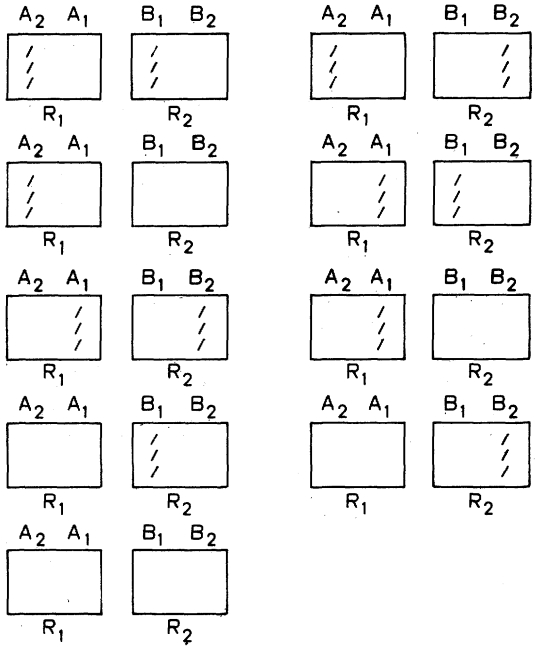


Fig. 1. Nine access configurations.

database. As pointed out in [16], a relevant partitioning of the entire design is necessary to make the optimal design of physical databases a practical matter. For instance, in Example 1, the number of possible alternatives would be reduced to $5 \times 6 \times 2^5 = 960$ if the optimal access configuration could be obtained by finding the optimal access configurations of individual relations independently.

In this paper we shall develop a methodology for the design of multifile physical databases that takes advantage of this partitioning effect. In particular, we discuss the issues involved in designing the access configuration of a physical database so as to minimize the total processing cost of input queries and updates. In calculating the processing cost, we only consider the number of I/O accesses; the cost due to the CPU time is not included. Our approach is somewhat formal and mathematical, deliberately avoiding excessive reliance on heuristics. Our purpose is to render the whole design phase manageable and to facilitate understanding of the underlying mechanisms.

By analyzing an important set of join methods possessing the property we call *separability,* we shall prove that, under certain constraints, optimal design of the access configuration of a multifile database can be reduced to the collective optimal designs of individual relations. In this paper we restrict the available join methods to this set to make the whole approach formally manageable. Extensions to other join methods will be mentioned briefly. The main idea is to set up a *basic design methodology* in accordance with a formal method that includes a large subset of practically important join methods, and then, using some straightforward heuristics, extend this *basic design methodology* to include other join methods as well.

Section II introduces several key assumptions, while Section III describes applicable join methods of interest. In Section V, the design theory will be developed by using the simple cost model introduced for the examples in Section IV. A design algorithm based on the theory will be introduced in Section VI. Extensions of our approach are mentioned briefly in Section VII.

II. Approaches and Assumptions

The design of an optimal physical database is complex for a number of reasons, two of which we shall discuss here. First, we may have several types of access structures available as options. Although some generalized formulas for determining access cost have been devised for certain kinds of file structures [7], [10], [12], [13], it is generally difficult to use them for the selection of optimal file structures without an exhaustive search among all possible alternatives. It therefore becomes necessary to accomplish a judicious separation of design steps and to develop interfaces that will minimize interactions among those steps.

The second source of complexity addressed is the interaction among the access structures assigned to different relations. There are various techniques available, especially join methods, for processing a query, and the choice frequently depends on the access structures available on more than one relation. Therefore, the processing cost of a query associated with one relation depends upon other interacting relations. It is the purpose of this paper to provide a mechanism for coping with these interactions during the design phase.

We choose a relational DBMS and start with the indexes and the clustering property of a single relation as the initially available access structures. The link structure [4] will be included as an extension of the basic result by using heuristics. Clustering of two or more relations, as in many hierarchical organizations, is not considered. For simplicity, we only consider the indexes and the clustering property defined for single columns. Multicolumn indexes and clustering property can be incorporated by defining virtual columns and treating them as separate columns having certain dependencies in the physical database design process [17]. We also assume that all TID (tuple identifier) manipulations can be performed in the main memory without any need to perform I/O accesses.

The database is assumed to reside on disk-like devices. Physical storage space for the database is divided into units of fixed size called blocks [18]. The block is not only the unit of disk allocation, but is also the unit of transfer between main memory and disk. We assume that a block that contains tuples of a relation contains only the tuples of that relation. Furthermore, we assume that the blocks containing tuples of a relation, which comprises a file, can be accessed serially. However, the blocks do not have to be contiguous on the disk.

In principle, we assume that a relation is mapped into a single file. Accordingly, from now on, we shall use the terms *file* and *relation* interchangeably; nor shall we make any distinction between an attribute and a column or between a tuple and a record. This does not mean, however, that we

exclude the possibility of storing prejoined forms of relations directly in the physical database. We believe this can be considered in a separate refining phase after the basic design has been obtained.

We shall develop a simple cost model of the storage structure in Section IV, and shall use various cost formulas based on this model. For convenience, we assume that the size of the available buffer is one block. However, the theory we develop is not dependent on the buffer size if we ignore the contention among many transactions in the buffer pool at query processing time. Not incorporated in our theory is either the effect of the contention in the buffer pool or the scheduling algorithm.

Relations can have various relationships (not necessarily semantically meaningful ones) depending on the characteristics of the domains of the attributes that are related. For example, if we relate a key attribute (or a set of attributes) in relation R_1 and a nonkey attribute (or a set of attributes) in relation R_2, then R_1 and R_2 have a one-to-many relationship with respect to these attributes. In this paper, we consider only one-to-many (including one-to-one) relationships between relations since we believe that many-to-many relationships between relations are less important for the optimization. Let us note that a many-to-many relationship among entity sets is extremely important in representing the semantics of the data [19], but is different from a many-to-many relationship between relations. A many-to-many relationship among entity sets at the conceptual level is often structured with an additional intermediate relation (called an association in [19]) representing the many-to-many relationship; in this case, however, the relationships between relations still remains to be either one-to-many or many-to-one. In contrast, a many-to-many relationship between relations may be relatively trivial; it is shown in [20] that, for a relation scheme R any of whose relation instances is a join of two relation instances whose relation schemes have a many-to-many relationship with respect to a set of attributes A, a multivalued dependency (MVD) [21] holds. We assume here that the only connection relating these two relations is the join based on A. Intuitively, if an MVD $A \rightarrow\!\!\!\!\rightarrow B$ (accordingly, $A \rightarrow\!\!\!\!\rightarrow A \cup B$ and $A \rightarrow\!\!\!\!\rightarrow R - B$) holds for relation scheme R, where A and B are sets of attributes in R, then in a specific relation instance r of R, given a specific value of A, the values of $R - B$ are completely replicated for every distinct value of $A \cup B$. Because of this replication, sets of attributes $A \cup B$ and $R - B$ tend not to have a meaningful relationship, and thus it does not make much sense to have both sets of attributes together in a single relation (say, in the result of a query involving a join).

Finally, we consider only one-variable or two-variable queries and update transactions[1] in this paper. Consideration of transactions of more than two variables needs a more complex treatment. In principle, however, a heuristic approach can be employed to decompose those transactions into sequences of two-variable transactions. A preliminary heuristic and its justifications can be found in [17].

[1]The term *transaction* used here should not be confused with the 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.

III. Transaction Evaluation

A. Queries

The class of queries we consider is shown in Fig. 2. The conceptual meaning of this class of queries is as follows. Tuples in relation R_1 are restricted by restriction predicate P_1. Similarly, tuples in relation R_2 are restricted by predicate P_2. The resulting tuples from each relation are joined according to the join predicate $R_1.A = R_2.B$, and the result projected over the columns specified by ⟨list of attributes⟩. We call the columns that are involved in the restriction predicates *restriction columns*, and those in the join predicate *join columns*. The actual implementation of this class of queries does not have to follow the order specified above as long as it produces the same result.

Query evaluation algorithms, especially for two-variable queries, have been studied in [4] and [5]. The algorithms for evaluating queries differ significantly in the way they use join methods. Before discussing the various join methods, let us define some terminology. Given a query, an index is called a *join index* if it is defined for the join column of a relation. Likewise, an index is called a *restriction index* if it is defined for a restriction column. We use the term *subtuple* for a tuple that has been projected over some columns. The restriction predicate in a query for each relation is decomposed into the form Q_1 AND Q_2, where Q_1 is a predicate that can be processed by using indexes, while Q_2 cannot. Q_2 must be resolved by accessing individual records. We shall call Q_1 the *index-processible predicate* and Q_2 the *residual predicate*.

Some algorithms for processing joins that are of practical importance are summarized below (see also [4], [6]).

- *Join Index Method:* This method presupposes the existence of join indexes. For each relation, the TID's of tuples that satisfy the *index processible predicates* are obtained by manipulating the TID's from each index involved; the resultant TID's are stored in temporary relations R_1' and R_2'. TID pairs with the same join column values are found by scanning the join column indexes according to the order of the join column values. As they are found, each TID pair (TID_1, TID_2) is checked to determine whether TID_1 is present in R_1' and TID_2 in R_2'. If they are, the corresponding tuple in one relation, say R_1, is retrieved. When this tuple satisfies the *residual predicate* for R_1, the corresponding tuple in the other relation R_2 is retrieved and the residual predicate for R_2 is checked. If qualified, the tuples are concatenated and the subtuple of interest is constructed. (We say that the direction of the join is from R_1 to R_2.)
- *Sort–Merge Method:* The relations R_1 and R_2 are scanned either by using restriction indexes, if there is an index-processible predicate in the query, or by scanning the relation directly. Restrictions, partial projections, and the initial step of sorting are performed while the relations are

```
SELECT <list of attributes>
FROM   R1, R2
WHERE  R1.A = R2.B AND
       P1          AND
       P2
```

Fig. 2. General class of queries considered.

being initially scanned and stored in temporary relations T_1 and T_2. T_1 and T_2 are sorted by the join column values. The resulting relations are scanned in parallel and the join is completed by merging matching tuples.

- *Combination of the Join Index Method and the Sort–Merge Method:* One relation, say R_1, is sorted as in the sort–merge method and stored in T_1. Relation R_2 is processed as in the join index method, storing the TID's of the tuples that satisfy the index processible predicates in R_2'. T_1 and the join column index of R_2 are scanned according to the join column values. As matching join column values are found, each TID from the join index of R_2 is checked against R_2'. If it is in R_2', the corresponding tuple in R_2 is retrieved and the residual predicate for R_2 is checked. If qualified, the tuples are concatenated and the subtuple is constructed.
- *Inner/Outer-Loop Join Method:* In the two join methods described above, the join is performed by scanning relations in the order of the join column values. In the inner/outer-loop join, one of the relations, say R_1, is scanned without regard to order, either by using restriction indexes or by scanning the relation directly. For each tuple of R_1 that satisfies predicate P_1, all tuples of relation R_2 that satisfy predicate P_2 and the join predicate are retrieved and concatenated with the tuple of R_1. The subtuples of interest are then projected upon the result. (We say the direction of the join is from R_1 to R_2.)
- *Multiple-Pass Method:* One of the relations participating in the join, say R_1, is scanned, the tuples are obtained, restricted, projected, and inserted into a data structure T_1, whose size is constrained to fit in the available main store. If space in main store is available to insert the resulting subtuple r, this is done. If space is not available, but the join column value in r is less than the current highest join column value in T_1, the subtuples with the highest join column value in T_1 are then deleted and r is inserted. Otherwise, r is not inserted at all. After T_1 has been formed, R_2 is scanned by using an appropriate access path, and every tuple of R_2 that satisfies the predicate is concatenated (if possible) with the appropriate subtuples in T_1 and the result projected. If there are more qualified tuples in R_1 than can fit in the main store for T_1, another scan of R_1 is done to form a new T_1 consisting of subtuples with join column values greater than the current highest. R_2 is also scanned again and the whole process repeated. This method is very fast if only one pass is needed. But processing time increases rapidly when more passes are performed.
- *Link-Based Join Method:* This is conceptually similar to the inner/outer-loop join method, but it takes advantage of existing links [4] between the two relations. The use of links will be mentioned briefly as an extension of our basic methodology.

Let us note that, in the combination of the join index method and the sort–merge method, the operation performed on either relation is identical to that performed on one relation in the join index method or in the sort–merge method. We call the operations performed on each relation *join index method (partial)* or *sort–merge method (partial)*, respectively; whenever no confusion arises, we call these operations simply *join index method* or *sort–merge method*. According to these definitions, the complete join index method actually consists of two join index methods (partial) and, similarly, the complete sort–merge method consists of two sort–merge methods (partial).

B. Update Transactions

We assume that the updates are performed only on individual relations, although the qualification part (WHERE clause) may involve more than one relation. Thus, updates are not performed on the join of two or more relations. (If they are, certain ambiguity arises on which relations to update [22].) The class of update transactions we consider is shown in Fig. 3.

The conceptual meaning of this class of transactions is as follows. Tuples in relation R_2 are restricted by restriction predicate P_2. Let us call the set of resulting tuples T_2. Then, the value for column C of each tuple in R_1 is changed to ⟨new value⟩ if the tuple satisfies the restriction predicate P_1 and has a matching tuple in T_2 according to the join predicate. In a more familiar syntax [23], the class of update transactions can be represented as in Fig. 4. The equivalence of the two representations has been shown for queries in [24].

Deletion transactions are specified in an analogous way. It is assumed that insertion transactions refer only to single relations. From now on, unless any confusion arises, we shall refer to update, deletion or insertion transactions simply as update transactions.

The update transaction in Fig. 3 can be processed just like queries except that an update operation is performed instead of concatenating and projecting out the subtuples after relevant tuples are identified. In particular, all the join methods described in Section III-A can be used for update transactions as well as to resolve the join predicates (ones that relate the two relations) that they have. But there are two constraints: 1) The sort–merge method cannot be used for the relation to be updated since it is meaningless to create a temporary sorted file to update the original relation. 2) When the inner/outer-loop join method is used, the direction of the join must be from the relation to be updated (R_1) to the other relation (R_2) because if the direction were reversed, the same tuple might be updated more than once.

IV. Cost Model of the Storage Structure

To calculate the cost of evaluating a query, we need a proper model of the underlying storage structure and its corresponding cost formula. Although the theory does not depend on the specifics of cost models, it is helpful to have a simple cost model for illustrative purposes.

We assume that a B-tree index [25] can be defined for a column or for a set of columns of a relation. The leaf-level of the index consists of pairs (key and TID) for every tuple in that relation. The leaf-level blocks are chained according to

```
UPDATE R1
SET    R1.C = <new value>
FROM   R1, R2
WHERE  R1.A = R2.B AND
       P1          AND
       P2
```

Fig. 3. General class of update transactions considered.

```
UPDATE R1
SET    R1.C = <new value>
WHERE  P1       AND
       R1.A  IN
         (SELECT R2.B
          FROM   R2
          WHERE  P2 )
```

Fig. 4. An equivalent form of the general class of update transactions.

the order of indexed column values, so that the index can be scanned without traversing the index tree. Entries having the same key value are ordered by TID.

An index is called a *clustering index* if the column for which this index is defined has the clustering property as defined in Example 1. With a clustering index, we assume that no block is fetched more than once when tuples with consecutive values of the indexed column are retrieved. Except for this ordering property, no other difference in the structure is assumed between a clustering and a nonclustering index. The clustering property can greatly reduce the access cost, especially when a join column has a clustering index. Unfortunately, only one column of a relation can have the clustering property since clustering requires a specific order of records in the physical file. One of the objectives of designing optimal physical databases is to determine which column will be assigned the clustering property.

The access cost will be measured in terms of the number of I/O accesses. The following notation will be used throughout this paper:

- n_R: number of tuples in relation R (cardinality)
- p_R: blocking factor of a block containing tuples of relation R
- L_I: blocking factor of an index block containing index I
- F_C: selectivity of the column used or the index thereof
- m_R: number of blocks in relation R, which is equal to n_R/p_R.

By using the simplified model above, the cost of various operations can be obtained as follows:

- Relation Scan Cost—cost for serially accessing all the blocks containing the tuples of a relation

$$RS(R) = n_R/p_R = m_R.$$

- Index Scan Cost—cost for serially accessing the leaf-level blocks of an entire index

$$IS(I,R) = \lceil n_R/L_I \rceil.$$

- Index Access Cost—cost for one access of the index tree from the root

$$IA(I,R) = \lceil \log_{L_I} n_R \rceil + (\lceil F_I \times n_R/L_I \rceil - 1).$$

Here the first term represents the cost of traversing the index tree from the root to a leaf node; the second the cost of scanning the remaining leaf nodes that contain the index entries having the key value.

- Sorting Cost—cost for sorting a relation, or a part thereof, according to the values of the columns of interest

$$\text{SORT}(NB) = 2 \times NB + 2 \times NB \times \log_z \lceil NB \rceil.$$

Here we assume that a z-way sort–merge is used for the external sort [26]. NB is the number of blocks in the temporary relation containing the subtuples to be sorted after restriction and projection have been resolved. It will be noted that SORT(NB) does not include the initial scanning time to bring in the original relation, while it does include the time to scan the temporary relation for the actual join after sorting (see [4]).

V. Design Theory

In this section we develop a theory for the design of optimal physical databases. We shall seek to facilitate comprehension through a series of examples and by case analysis, using the cost model developed in Section IV. Observations resulting from this procedure are formalized and proved in Section V-D.

Our approach to physical database design is based on the premise that at execution time the query processor will choose the best processing method for a given query. We call this processor an *optimizer*. Since the behavior of the optimizer at execution time affects the physical database design critically, we investigate this issue and discuss how it is related to the design.

Since the set of join methods consisting of the join index method, the sort–merge method, and the combination of the two possesses the special property, called *separability* which we shall define later, we regard only those methods as being available for the design theory (the inner/outer-loop join method, the multiple-pass method, and the link-based method are nonseparable join methods with respect to this separable set).

We define the influence of the restriction on one relation to the number of tuples to be retrieved in the other relation the *coupling effect* (which is similar in concept to the *feedback* mentioned in [5]). Starting with a case in which coupling effects between relations are not considered, we then proceed to those cases in which they are included.

A. Cases Without Coupling Effects

Example 2: Fig. 5 describes two relations R_1 and R_2 with their access configurations. Dashed lines (/) represent clustering indexes, the dotted lines (:) nonclustering indexes. Columns without either type of line have no indexes defined for them. We would like to find the best method of evaluation, which the optimizer would choose at query processing time, for the following query:

```
SELECT  A1, A3, B3
FROM    R1, R2
WHERE   R1.A2 = 'a2' AND
        R2.B2 = 'b2' AND
        R1.A1 = R2.B1.
```

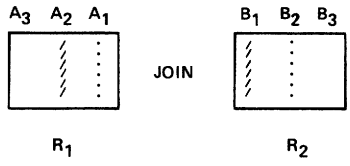


Fig. 5. Relations R_1 and R_2.

For this example only, it is also assumed that all the tuples in each relation participate in the join.

Given these assumptions, the optimizer could try all the possible combinations of the join methods, evaluate the cost of each, and then select the one that costs the least. We have here the following combinations:

	R_1	R_2
1.	Join index method (partial)	Join index method (partial)
2.	Sort–merge method (partial)	Sort–merge method (partial)
3.	Join index method (partial)	Sort–merge method (partial)
4.	Sort–merge method (partial)	Join index method (partial).

Using the cost model developed in Section IV, the following formulas give the cost (number of block accesses) for each of the four cases above. In each formula the first and second bracketed expressions represent the cost of accessing relations R_1 and R_2, respectively. Bracketed expressions in the formulas are given arbitrary values for illustrative purposes. Those expressions whose form is identical are given the same value.

$$\begin{aligned}\text{Cost} = {} & [IA(I_{A2}, R_1) + IS(I_{A1}, R_1) + F_{A2} \times n_{R1}] \quad :100+ \\ & + [IA(I_{B2}, R_2) + IS(I_{B1}, R_2) \\ & \quad + b(m_{R2}, p_{R2}, F_{B2} \times n_{R2})] \quad : 20 \qquad (1)\end{aligned}$$

$$\begin{aligned}\text{Cost} = {} & [IA(I_{A2}, R_1) + F_{A2} \times m_{R1} \\ & \quad + \text{SORT}(F_{A2} \times H_{R1} \times m_{R1})] \quad : 60+ \\ & + [IA(I_{B2}, R_2) + b(m_{R2}, p_{R2}, F_{B2} \times n_{R2}) \\ & \quad + \text{SORT}(F_{B2} \times H_{R2} \times m_{R2})] \quad : 50 \qquad (2)\end{aligned}$$

$$\begin{aligned}\text{Cost} = {} & [IA(I_{A2}, R_1) + IS(I_{A1}, R_1) + F_{A2} \times n_{R1}] \quad :100+ \\ & + [IA(I_{B2}, R_2) + b(m_{R2}, p_{R2}, F_{B2} \times n_{R2}) \\ & \quad + \text{SORT}(F_{B2} \times H_{R2} \times m_{R2})] \quad : 50 \qquad (3)\end{aligned}$$

$$\begin{aligned}\text{Cost} = {} & [IA(I_{A2}, R_1) + F_{A2} \times m_{R1} \\ & \quad + \text{SORT}(F_{A2} \times H_{R1} \times m_{R1})] \quad : 60+ \\ & + [IA(I_{B2}, R_2) + IS(I_{B1}, R_2) \\ & \quad + b(m_{R2}, p_{R2}, F_{B2} \times n_{R2})] \quad : 20. \qquad (4)\end{aligned}$$

Here $b(m, p, k)$ is a function that provides the number of block accesses, where m is the total number of blocks, p is the blocking factor, and k is the number of tuples to be retrieved in TID order. An exact form of this function and various approximation formulas are summarized in [27]. The function is approximately linear in k when $k << n$, and approaches m as k becomes large. A familiar approximation suggested by Cardenas [8] is $b(m, p, k) = m[1 - (1 - 1/m)^k]$. F_{A2} and F_{B2} are the selectivities of the columns $R_1.A_2$ and $R_2.B_2$, respectively. In (1), $F_{A2} \times n_{R1}$ and $b(m_{R2}, p_{R2}, F_{B2} \times n_{R2})$ represent the numbers of blocks accessed that contain data tuples of relation R_1 and R_2, respectively. Since retrieving tuples by scanning a nonclustering join index will access the tuples randomly, the same block will be accessed repeatedly if it contains more than one tuple. Therefore, it is very likely that one block access is needed to retrieve each tuple. Hence, we get $F_{A2} \times n_{R1}$ for the number of data blocks fetched from relation R_1. Note that in this case the tuples cannot be accessed in TID order. For relation R_2, however, the join index is clustering and thus the tuples will be retrieved in TID order, even though they are selected randomly by the restriction. Therefore, even though a block contains more than one tuple, each relevant block of R_2 will be fetched only once. We thus get $b(m_{R2}, p_{R2}, F_{B2} \times n_{R2})$ for the number of data blocks fetched from R_2, where $F_{B2} \times n_{R2}$ is the number of tuples selected by the restriction.

In (2), $F_{A2} \times m_{R1}$ and $b(m_{R2}, p_{R2}, F_{B2} \times n_{R2})$ represent the numbers of blocks accessed during the initial scan of the relation prior to sorting. Since the restriction index is clustering in relation R_1, the initial scan through this restriction index will access $F_{A2} \times m_{R1}$ blocks. In relation R_2, a nonclustering restriction index is used to access the relation initially. This restriction results in random distribution of TID's of the qualified tuples over the blocks. Since these tuples are then accessed in TID order, the access cost is $b(m_{R2}, p_{R2}, F_{B2} \times n_{R2})$.

The factor H_{R2} used in (3) represents the projection effect upon relation R_2. Since the projection selects only part of the attributes from the relations, the tuple is usually smaller after projection. The time required to write the final result is not included since it is the same regardless of the join method used.

With the specific values of the access cost given, (4) gives the minimum access cost. We note that the access costs for each relation do not depend on any parameter of any other relation, and that each part of the cost of (4) becomes the local minimum. That is, the first part of the cost incurred by accessing relation R_1 is the minimum of the costs of the join methods used for R_1, while the second part is the minimum of those for R_2. This implies that the optimizer can determine the optimal join method on one relation without regard to any properties of other relations. ■

The foregoing observation is extremely important because if we can determine the optimal join method for one relation without regard to other relations, we can also use the following method to determine the optimal access configuration for the relation without regard to other relations:

1) Try every possible access configuration for a relation in turn

2) for a given access configuration, find the best evaluation method, which the optimizer would choose at query

processing time, for each given query (this corresponds to the query optimization problem)

3) then calculate the total cost for processing the queries, using their expected frequency of occurrence

4) repeat this procedure for all other possible access configurations, finally selecting the one that yields the minimal total cost.

The result of this will be to reduce designing an optimal access configuration of a database to that of a single relation. Local optimal solutions for individual relations constitute an optimal solution for the entire database. However, the foregoing procedure of making an exhaustive search of all the possible access configurations could yet prove too costly. Therefore, in Section VI we divide the design procedure into two parts: choice of the clustering column and index selection. We shall provide a clean interface between the two steps and discuss deviations from the true optimum.

It should be pointed out here that, despite our assumption that there is no coupling effect between the two relations and despite the fact that the above argument appears to follow directly from that assumption, it will be shown in the following discussion that the problem is similarly reduced even when coupling effects are actually present. Before further discussion, we need the following definition and example.

Definition 1: The *join selectivity* $J(R, JP)$ of a relation R with respect to a join path JP is the ratio of the number of distinct join column values of the tuples participating in the unconditional join to the total number of the distinct join column values of R. A *join path* is a set $(R_1, R_1.A, R_2, R_2.B)$, where R_1 and R_2 are relations participating in the join and $R_1.A$ and $R_2.B$ are the join columns of R_1 and R_2, respectively. An *unconditional join* is a join in which the restrictions on either relation are not considered. ■

Join selectivity is the same as the ratio of the number of tuples participating in the unconditional join to the total number of tuples in the relation (cardinality of the relation). Join selectivity is generally different in R_1 and R_2 with respect to a join path, as shown in the following example.

Example 3: Let us assume that the two relations in Fig. 6 have a 1-to-N partial dependency relationship. Partial dependency means that every tuple in the relation R_2 that is on the N-side of the relationship has a corresponding tuple in R_1, but not vice versa [28]. Let us assume that 50 percent of the employees have at least one child each so that the tuples representing those employees participate in the unconditional join. Every tuple in the children relation R_2 is assumed to have only one corresponding tuple in R_1 and all of them participate in the unconditional join according to the partial dependency. The join selectivity of the employees relation is then 0.5, while that of the children relation is 1.0. ■

B. Cases With Coupling Effects

Let us investigate the four cases shown in Example 2, using the same query, join methods, and access configuration defined as in Fig. 5, but now with coupling effects. In fact, we shall consider coupling effects throughout our subsequent discussions. We shall also assume that R_1 and R_2 have a 1-to-N relationship (1 for R_1 and N for R_2).

R_1: Employees(E#, Job, Age, Salary)
R_2: Children(E#, Name, Hair-color, Sex)

Fig. 6. Employees and children relations.

Case 1: The join index method is applied to both relations R_1 and R_2. With coupling effect, the join will be performed as follows. If a tuple of relation R_1 does not satisfy the restriction predicate for R_1, the corresponding tuples of R_2 that have the same join column values are not accessed. Hence, we have the coupling effect from R_1 to R_2. If there are only index-processible predicates in the query to be evaluated, the situation is then symmetric in the sense that, for the tuples in relation R_2 that do not satisfy the restriction predicate for R_2, the corresponding tuples of R_1 are not accessed either. We have this symmetry because we can resolve all index-processible predicates by using TID's only, without any need to access the data tuples themselves.

Since both $R_1.A_2$ and $R_2.B_2$ have indexes defined for them, the restriction predicates in the WHERE clause are index processible. Therefore, the cost of evaluating this query, including the coupling effect, will be as follows:

$$\begin{aligned}\text{Cost} = {} & [IA(I_{A2}, R_1) + IS(I_{A1}, R_1) \\ & + \{\langle J_1 \times b(1/F_{B1}, F_{B1} \times n_{R2}, F_{B2} \times n_{R2}) \\ & \quad /(1/F_{B1})\rangle \times F_{A2} \times n_{R1}\}] \\ & + [IA(I_{B2}, R_2) + IS(I_{B1}, R_2) \\ & \quad + b(m_{R2}, p_{R2}, \{\langle J_2 \times F_{A2}\rangle \times F_{B2} \times n_{R2}\})].\end{aligned}$$

Here J_1 and J_2 represent the join selectivity of relations R_1 and R_2, respectively, for the join path considered. Expressions in the braces represent the numbers of data tuples accessed in relations R_1 and R_2, respectively. In the first part of the formula, the expression in the braces simultaneously represents the number of blocks accessed in relation R_1. This follows the argument shown in Example 2.

F_{B1} is the selectivity of column $R_2.B_1$ and $1/F_{B1}$ represents the number of groups of tuples that have the same join column values in relation R_2, which is essentially the same as the number of distinct join column values.

The expression $b(1/F_{B1}, F_{B1} \times n_{R2}, F_{B2} \times n_{R2})$ represents the number of groups selected by restriction F_{B2}. Although the b function estimates the number of block accesses in which a certain number of tuples are randomly selected, the same function is used for estimating the number of logical groups selected, if the latter are assumed to be of uniform size. Note that the clustering or nonclustering of tuples in a group is irrelevant. $F_{B1} \times n_{R2}$, the number of tuples in one logical group, plays a role similar to that of the blocking factor.

The expression $b(1/F_{B1}, F_{B1} \times n_{R2}, F_{B2} \times n_{R2})/(1/F_{B1})$ represents the ratio of the number of groups selected by restriction F_{B2} to the total number of groups in relation R_2. Since every tuple participating in the unconditional join in R_1 has a unique join column value and, accordingly, exactly one corresponding group in R_2 (let us recall that R_1 is on the 1-side of the 1-to-N relationship), this ratio correctly represents a special restriction upon R_1 caused by the coupling effect originating in R_2.

In the second part of the cost formula, we simply use F_{A2} to represent the coupling effect directed from R_1 to R_2. Since in R_1 every tuple has a unique join column value, if a tuple is selected according to the restriction, the corresponding group in R_2 that has the same join column value (if it exists) will be selected on the basis of this special restriction resulting from the coupling effect. Hence, F_{A2} represents the ratio of the number of groups selected as a consequence of the coupling effect to the total number of groups in R_2 participating in the unconditional join. That ratio, in turn, has the same value as the ratio of tuples, selected according to the coupling effect, to the total number of tuples participating in the unconditional join in R_2. ■

The coupling effect is formally defined as follows.

Definition 2: The *coupling effect* from relation R_1 to relation R_2, with respect to a type of query, is the ratio of the number of distinct join column values of the records of R_1, selected according to the restriction predicate for R_1, to the total number of distinct join column values in R_1. ■

If we assume that the join column values are randomly selected, the coupling effect from R_1 to R_2 is the same as the ratio of the number of distinct join column values of R_2 selected by the effect of the restriction predicate for R_1 to the number of distinct join column values in R_2 participating in the unconditional join.

Definition 3: A *coupling factor* Cf_{12} from relation R_1 to relation R_2, with respect to a type of query, is the ratio of the number of distinct join column values of R_2, selected by both the coupling effect from R_1 (through the restriction predicate for R_1) and the join selectivity of R_2, to the total number of distinct join column values in R_2. ■

According to the definition, a coupling factor can be obtained by multiplying the coupling effect from R_1 to R_2 by the join selectivity of R_2. This coupling factor contains all the consequences of the interactions of relations in the join operation since it includes both coupling and joining filtering effects. Let us note that, although the coupling factor can be obtained in any case, it does not always contribute to the reduction of the tuples to be retrieved. We will see an example of this in Case 2, below. A coupling factor is said to be *effective* if the coupling effect actually contributes to the reduction of the tuples to be retrieved. In Case 1, the expressions in angle brackets represent the coupling factors from R_2 to R_1 and from R_1 to R_2, respectively, for the type of query considered. By definition, different queries are of the same *type* if they are identical except for their literal values. The same applies to update transactions. For example, INSERT INTO $R_1\langle a, b\rangle$ is of the same type as INSERT INTO $R_1\langle c, d\rangle$. Hence,

$$Cf_{12} = J_2 \times F_{A2},$$

$$Cf_{21} = J_1 \times b(1/F_{B1}, F_{B1} \times n_{R2}, F_{B2} \times n_{R2})/(1/F_{B1}).$$

One important observation here is that the coupling factors do not depend on the specific access structures present in either relation, nor on the specific join method selected, but rather (and solely) depend on the restriction and the data characteristics. Such characteristics include the side the relation is on in the 1-to-N relationship, the average number of tuples in one group, and the join selectivity, which will be known before we start the design phase.

Note that the coupling factors differ according to the specific type of query being considered. Different types of queries have different join paths and different combinations of columns in the restriction predicate, with consequently different selectivities for the calculation of coupling factors.

Now let us investigate the remaining cases in which coupling effects are present between relations.

Case 2: The sort–merge join method is applied to both relations, in the same situation as in Fig. 5. The cost formula is then as follows:

$$\begin{aligned}\text{Cost} = {} & [IA(I_{A2}, R_1) + F_{A2} \times m_{R1} \\ & + \text{SORT}(F_{A2} \times H_{R1} \times m_{R1})] \\ & + [IA(I_{B2}, R_2) + b(m_{R2}, p_{R2}, F_{B2} \times n_{R2}) \\ & + \text{SORT}(F_{B2} \times H_{R2} \times m_{R2})].\end{aligned}$$

It will be noted that the coupling factors do not appear in the cost formula. This is because when the sort–merge join method is used, an initial scan and the sort are performed before the join is resolved; indexes are not used any more while the join is being actually resolved since the relation scan is performed upon the sorted temporary relations. The coupling effect can arise only when the join is being actually resolved and only when the join index is used. Thus, the coupling factor is not effective in this case.

Case 3: The sort–merge join method is used for R_1, the join index method for R_2, in the same situation as in Fig. 5. The join will be performed as described in Section III, under the heading "Combination of the Join Index Method and Sort–Merge Method." Note that the coupling factor is effective from R_1 to R_2. Thus, we obtain the following cost formula:

$$\begin{aligned}\text{Cost} = {} & [IA(I_{A2}, R_1) + F_{A2} \times m_{R1} \\ & + \text{SORT}(F_{A2} \times H_{R1} \times m_{R1})] \\ & + [IA(I_{B2}, R_2) + IS(I_{B1}, R_2) \\ & + b(m_{R2}, p_{R2}, Cf_{12} \times F_{B2} \times n_{R2})].\end{aligned}$$

Case 4: The join index method is used on R_1, the sort–merge method on R_2, in the same situation as in Fig. 5. We obtain the following cost formula:

$$\begin{aligned}\text{Cost} = {} & [IA(I_{A2}, R_1) + IS(I_{A1}, R_1) + Cf_{21} \times F_{A2} \times n_{R1}] \\ & + [IA(I_{B2}, R_2) + b(m_{R2}, p_{R2}, F_{B2} \times n_{R2}) \\ & + \text{SORT}(F_{B2} \times H_{R2} \times m_{R2})].\end{aligned}$$

C. Cases When Restriction Indexes are Absent on One Relation

All four cases that have been discussed so far assume the same situation as in Example 2, except for inclusion of the coupling effect. We still have to consider more general cases in which restriction indexes are absent for the columns specified in the predicate of the query for one relation. The case in which the restriction indexes are absent in both relations will be treated in Section V-D. For clarity of presentation, let us define a shorthand notation for the cost formula.

Definition 4: Cost(R_k, Cf_{jk}, type-of-join) is the cost of a join operation associated with relation R_k when R_k has a coupling factor Cf_{jk} from R_j to R_k, with respect to the query of interest, and the type-of-join is the join method used between R_k and R_j. ■

Although costs differ for different access configurations, this shorthand notation for the cost function does not show that difference explicitly because it is irrelevant to our subsequent discussions. Using this definition, cost formulas for the previous cases can be restated as

$$\text{Case 1: } \text{Cost}(R_1, Cf_{21}, \text{Join-index}) + \text{Cost}(R_2, Cf_{12}, \text{Join-index}) \quad (5)$$

$$\text{Case 2: } \text{Cost}(R_1, Cf_{21}, \text{Sort–merge}) + \text{Cost}(R_2, Cf_{12}, \text{Sort–merge}) \quad (6)$$

$$\text{Case 3: } \text{Cost}(R_1, Cf_{21}, \text{Sort–merge}) + \text{Cost}(R_2, Cf_{12}, \text{Join-index}) \quad (7)$$

$$\text{Case 4: } \text{Cost}(R_1, Cf_{21}, \text{Join-index}) + \text{Cost}(R_2, Cf_{12}, \text{Sort–merge}). \quad (8)$$

If there is no coupling effect between the two relations, as in the case of a query that does not impose a restriction on a relation, say R_2, then the coupling factor Cf_{21} simply becomes the join selectivity J_1 if the join index method is used for R_1. The cost, in this case, will be Cost(R_1, J_1, type-of-join). When the sort–merge join method is used for relation R_k the cost becomes Cost(R_k, 1, sort–merge). But it is identical to Cost(R_k, Cf_{jk}, sort–merge) because, as we observed in Case 2, the coupling factor is not used in the cost formula. According to the same argument, we conclude that the cost of the sort–merge join method can always be written as Cost(R_k, Cf_{jk}, sort–merge).

Case 1-A: Let us assume that the join index method is used for both R_1 and R_2, in the same situation as in Fig. 5, except that the restriction index for column $R_1.A_2$ is missing. The join will be performed as follows. First the TID set R_2' of the tuples that satisfy the restriction on R_2 is obtained by using the restriction on column $R_2.B_2$. TID pairs that have the same join column values are found by scanning the join column indexes according to the order of join column values. As it is found, each TID pair (TID_1, TID_2) is checked to see if TID_2 is present in R_2'. If it is, the corresponding tuple in relation R_1 is retrieved. If this tuple satisfies the restriction upon R_1, the corresponding tuple in R_2 is also retrieved and concatenated, and the result is projected. Note that the coupling factors are effective in both directions. Thus, the cost of evaluating the query will be

$$\begin{aligned}\text{Cost} &= [IS(I_{A1}, R_1) + Cf_{21} \times n_{R1}] \\ &\quad + [IA(I_{B2}, R_2) + IS(I_{B1}, R_2) \\ &\quad + b(m_{R2}, p_{R2}, Cf_{12} \times F_{B2} \times n_{R2})] \\ &= \text{Cost}(R_1, Cf_{21}, \text{join-index}) \\ &\quad + \text{Cost}(R_2, Cf_{12}, \text{join-index}).\end{aligned}$$

Note that since the restriction index on column $R_1.A_2$ is missing, the first part of the cost formula is different from that of Case 1, but the coupling factors remain the same. The case in which $R_2.B_2$ is absent instead of $R_1.A_2$ is treated similarly and will result in the same formula in the shorthand notation.

Case 2-A: The sort–merge method is used for both R_1 and R_2 in the same situation as in Fig. 5, except that the restriction index on the column $R_1.A_2$ is missing. The cost formula becomes

$$\begin{aligned}\text{Cost} &= [m_{R1} + \text{SORT}(F_{A2} \times H_{R1} \times m_{R1})] \\ &\quad + [IA(I_{B2}, R_2) + b(m_{R2}, p_{R2}, F_{B2} \times n_{R2}) \\ &\quad + \text{SORT}(F_{B2} \times H_{R2} \times m_{R2})] \\ &= \text{Cost}(R_1, Cf_{21}, \text{sort–merge}) \\ &\quad + \text{Cost}(R_2, Cf_{12}, \text{sort–merge}).\end{aligned}$$

The case in which the index on $R_2.B_2$ is missing (rather than $R_1.A_2$) is treated similarly and will result in the same formula in the shorthand notation.

Cases 3-A and 4-A: The sort–merge method is used for R_1 and the join index method for R_2, in the same situation as in Fig. 5, except that the restriction index for the column $R_2.B_2$ is missing. In this case, the join is performed as in Case 3. The only difference is that since indexes are now absent for the restriction columns of R_2, the restriction predicate for R_2 can be resolved only after the tuples are retrieved. The cost of evaluating the query becomes

$$\begin{aligned}\text{Cost} &= [IA(I_{A2}, R_1) + F_{A2} \times m_{R1} \\ &\quad + \text{Sort}(F_{A2} \times H_{R1} \times m_{R1})] \\ &\quad + [IS(I_{A1}, R_1) + b(m_{R2}, p_{R2}, Cf_{21} \times n_{R2})] \\ &= \text{Cost}(R_1, Cf_{21}, \text{sort–merge}) \\ &\quad + \text{Cost}(R_2, Cf_{12}, \text{join-index}).\end{aligned}$$

In the case in which $R_1.A_2$ is missing (rather than $R_2.B_2$), it will change the first part of the cost formula we obtained in Case 3, but will result in the same shorthand form. The case in which the join index method is used on R_1 and the sort–merge method on R_2, as in Case 4, is treated similarly.

D. Formalization

In all the cost formulas so far, the coupling factors have been used in both directions, i.e., both bracketed expressions in a formula were of the form Cost(R_k, Cf_{jk}, type-of-join). We shall call the form of these formulas *symmetric*.

Join costs can be written in this form only when the coupling factors are known to be effective for the join method used (as when the join index method was used in the previous cases), or when the cost can be determined regardless of the coupling factors (as when the sort–merge method is used). The reason is that the only ambiguity in determining the cost of a join is whether or not the coupling factor will be included in the calculation since all other information needed is local and is not affected by interaction with other relations. If we know at design phase that coupling factors are effective or that the cost is independent of the coupling factor, we can determine at design phase the costs of various possible joins

on each relation and, using only local information and the coupling factors without ambiguity, accordingly determine the best join method and its cost. There are, however, some cases in which we cannot determine whether the coupling factors are effective at design phase. These will be introduced in Example 4.

If the best join method can be determined with only the local information (the access configuration of the relation and the type of join method used) and coupling factors, without any regard to other relations, the clear implication is that we can design an optimal access configuration of a relation by using only local information and the coupling factors independently of the other relations. The design could be performed by the following procedure:

1) consider each possible access configuration of a relation in turn
2) find the best join method and its cost for the particular configuration
3) repeat this procedure for other access configurations
4) find the one that gives the minimum join cost.

The only nonlocal information used here is furnished by coupling factors. Lumped within them are all the interactions from other relations. We have already observed that the coupling factors do not depend on access configurations of the other relations, nor do they depend on the join methods chosen; they depend exclusively on the properties of given queries and the data characteristics of the relation. Furthermore, these properties can be determined before we start designing any access configuration in the database.

We conclude here that we can design the access configuration of the entire database optimally by designing the optimal access configurations of individual relations one by one, regardless of the remaining relations, when all the information needed is known at design time. The optimal configurations of individual relations will collectively comprise the globally optimal configuration.

To formalize the foregoing observation, we need the following definitions and theorems.

Definition 5: A *partial-join cost* is that part of the join cost that represents the accessing of only one relation, as well as the auxiliary access structures defined for that relation. ■

In the examples above, each expression in square brackets represents a partial-join cost.

Definition 6: A *partial-join algorithm* is a conceptual division of the algorithm of a join method whose processing cost is a partial-join cost. ■

Definition 7: A join method is *symmetric* under certain constraints if, under these constraints, both partial-join costs can be determined with only local information of the pertinent relation and the coupling factor, regardless of the partial-join algorithm used and the access configuration defined for the relation on the other side of the join. ■

Definition 8: A set of join methods is *separable* under certain constraints, if under these constraints

- any partial-join algorithm of a join method in the set can be combined with any partial-join algorithm of any join method in the set, and
- any combination of partial-join algorithms of the join methods in the set produces a symmetric join method. ■

From the discussion at the beginning of Section V-D, we have the following lemma.

Lemma 1: A join method is symmetric if and only if its cost has a symmetric form. ■

Theorem 2: The problem of designing the optimal access configuration of a database can be decomposed into the tasks of designing the optimal access configurations of individual relations independent of one another if the set of join methods used by the optimizer is *separable* with respect to the constraints imposed upon the database system.

Proof: Since the set of join algorithms used is separable, we can choose an arbitrary combination of partial-join algorithms within the set. Thus, we can choose any partial-join algorithm to be used for one relation without regard to the partial-join algorithm used for the other relation. Furthermore, since a join method consisting of any combination of partial-join algorithms is symmetric, the partial-join cost of a partial-join algorithm can be evaluated independently of the partial-join algorithm used and the access configuration defined on the other side of the join. As a result, the specific access methods assigned to and the partial-join algorithm used for one relation cannot affect any design parameters for the other relations. It is therefore guaranteed that there will be no interference among the designs of individual relations. Q.E.D.

Theorem 2 is a generalization of the observation made from Example 2, except that it now includes the coupling effects between relations.

Theorem 3: The set of join algorithms consisting of the join index method and the sort–merge method is *separable* under the constraint that every column in every relation in the database must have an index defined for it.

Proof: Part 1 of Definition 8 is obvious from previous examples and cases. When the join index method is used for both relations, all predicates are index processible since every column has an index. Hence, all predicates are resolved with TID's before the relations themselves are accessed; coupling factors are effective in both directions; and the cost formula has symmetric forms. When the sort–merge method is used for one relation and the join index method for the other, then, by the same reasoning as in Case 3, the cost formula has symmetric forms. If only the sort–merge method is used, the cost formula is always symmetric. Therefore, from Lemma 1, the theorem holds. Q.E.D.

Only symmetric joins have been used in the example and cases presented so far. There are, however, instances of nonsymmetric joins.

Example 4: Let us assume that the join index method is used for both R_1 and R_2, in the same situation as in Fig. 5, but that now restriction indexes for both R_1 and R_2 are missing. In this situation, since there are no restriction indexes, there is no way of resolving the restriction predicate without accessing the tuples themselves. Therefore, if we access relation R_1 first, the access cost would be

$$\begin{aligned}\text{Cost1} &= [IS(I_{A1}, R_1) + J_1 \times n_{R1}] \\ &\quad + [IS(I_{B1}, R_2) + b(m_{R2}, p_{R2}, Cf_{12} \times n_{R2})] \\ &= \text{Cost}(R_1, J_1, \text{join-index}) \\ &\quad + \text{Cost}(R_2, Cf_{12}, \text{join-index}).\end{aligned}$$

On the other hand, if we access relation R_2 first, the access cost would then be

$$\begin{aligned}\text{Cost2} &= [IS(I_{A1}, R_1) + Cf_{21} \times n_{R1}] \\ &\quad + [IS(I_{B1}, R_2) + b(m_{R2}, p_{R2}, J_2 \times n_{R2})] \\ &= \text{Cost}(R_1, Cf_{21}, \text{join-index}) \\ &\quad + \text{Cost}(R_2, J_2, \text{join-index}).\end{aligned}$$

Therefore, we have two expressions each for the partial-join cost of each relation and we cannot determine at the design stage which of them is cheaper. Hence, this join method is not symmetric. The coupling factor is ineffective in one direction in each formula since the join selectivity is used in its place. The cost formula is now also asymmetric relative to the coupling factors. ■

We can still determine which of the two expressions is cheaper at query processing time, but we do not have this knowledge when the physical database is being designed. If we want to ascertain the cheaper expression at design time, we have to analyze simultaneously the relation on the other side of the join, but this violates the definition of symmetry. The design of access configuration for one relation is no longer independent of the other relations. The theory presented in this paper depends entirely on the property of separability, which in turn depends on that of symmetry. The situation depicted in Example 4 is an apparent exception to our theory. However, in our discussion of the index selection problem in Section VI, the justification on the validity of our approach will be amply reinforced.

Theorem 4: The set of join methods consisting of the join index method and the sort–merge method is *separable* under the constraint that whenever the join index method is used for both relations, at least one relation must have indexes for all restriction columns.

Proof: When both relations have indexes on all restriction columns, this theorem reduces to Theorem 3. As before when the sort–merge method is used for both relations, the cost formulas are always symmetric. When the join index method is used for one relation and the sort–merge method for the other, then, by a reasoning similar to Case 3-A, we obtain symmetric cost formulas. If only the join index method is used and one of the relations, say R_1, has incomplete restriction indexes, the join is performed as in Case 1-A except that the restriction on R_1 is now partially resolved by using TID's before accessing the tuples in R_1. We thus get symmetric cost formulas. By Lemma 1, we prove this theorem. Q.E.D.

E. Update Cost

We assumed in Section III-B that the updates are performed only on individual relations, although the qualification part (WHERE clause) may involve more than one relation. Imagine that the qualification part, which can be treated as a query, is segregated. Then, the remaining part (update operation) depends only on the local parameters of the relation to be updated and on the coupling factor because the update operation should only occur after all the predicates are resolved. When processing the qualification part, there are some restrictions as explained in Section III-B. The restriction, however, is independent of the access structures or partial-join algorithms of other relations. Thus, separability can also be applied to the update transactions as well.

VI. Design Algorithm

In this section, an algorithm for the design of optimal access configuration of the database will be presented.

A. Design Step 1

Based mainly on the result of Theorems 2 and 3, the first step of our algorithm is as follows.

Inputs:

- Usage information: A set of various types of queries and update transactions with their respective frequencies.
- Data characteristics (for every relation in the database): Size, blocking factor, selectivities of all columns, relationships with other relations with respect to join paths, join selectivity with respect to join paths.

Outputs:

- Optimal position of the clustering column for each relation.
- Optimal combination of partial-joins for each type of two-variable query.

Condition Assumed:

- Every column of each relation in the database has an index defined for it. Some of these indexes will be dropped in the subsequent index selection step.

Algorithm 1:

1) Segregate the usage information in such a way that if there is a subquery involving more than one relation in the qualification part of an update transaction, it is separated and its frequency is included with that of the same type of query. Thereupon, all the remaining parts of the update transactions will refer to only one relation.

2) Calculate the coupling factors with respect to individual two-variable queries for every relation in the database using the given data characteristics.

3) Pick one relation and determine the optimal position of the clustering column as follows:

a) Assign the clustering property to one column of the relation.

b) Given that position of the clustering column, identify the best partial-join algorithm and calculate its partial-join cost for every two-variable query that refers to this relation, using the given data characteristics and the coupling factors.

c) Utilizing the usage information and the result of

step b), calculate the total cost associated with this relation. This is done by summing up all the partial-join costs identified in step b), multiplied by their respective frequencies, and all costs incurred by one-variable queries and update transactions acting upon this relation.

d) Shift the clustering property to another column of the relation and repeat steps b) and c).

e) Repeat step d) until all the columns of the relation have been considered. The case in which there is no clustering column is also considered. Then determine the one that gives the minimal cost as the clustering column (or none).

4) Step 3) is repeated for every relation in the database. The aggregate of results for all relations comprises the global optimum.

A join path can often have a multiple column as the join column on either relation. In such cases, we consider the multiple join column as a single effective column, independent of its component columns. Therefore, according to the condition in the above algorithm, this effective column is considered to have a multiple-column index defined for it (we do not consider here additional problems involved in the handling of multiple-column indexes).

Although in some cases improvement can be obtained by an adjustment in ordering among the effective column's component columns and by the deletion of overlapping indexes, this is not being considered here. It will be noted that, under the assumptions given, the Design Step 1 algorithm yields a mathematically true optimum.

B. Design Step 2—Index Selection

In the algorithm for Design Step 1, we imposed the restriction that every column of the relations in the database must have an index defined for it. However, not every index is beneficial. Some indexes can increase the total access cost because of their own access and update costs.

The index selection problem has been extensively studied by [9], [11], [29]–[31]. It concerns the method of selecting a set of indexes that will minimize the processing cost in a single-relation environment. The index selection algorithm presented here bears some resemblance to the one introduced by Hammer and Chan [9], but it uses the DROP heuristic [32] instead of the ADD heuristic [33]. The DROP heuristic attempts to obtain an optimal solution by incrementally dropping indexes starting with 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. An extensive test performed for the validation of index selection heuristics shows that the DROP heuristic performs better than the ADD heuristic [17]. In all the cases tested, the DROP heuristic found optimal solutions. In comparison, the ADD heuristic produced nonoptimal solutions in about one quarter of those cases. One possible reason the ADD heuristic does not perform well is the following. In the ADD heuristic, when the first index is added, the cost changes drastically effecting an abrupt change in the design process. In the DROP heuristic, however, the dropping of indexes induces a smooth transition in the design process; the reason for this is that, with the other indexes present that compensate for one another, the dropping of an index has a relatively small effect on cost.

Following is the algorithm for Design Step 2. This algorithm is mainly based on the above discussion, and on Theorems 2 and 4.

Inputs:

- Outputs from Design Step 1: Optimal position of the clustering column for each relation and optimal combination of partial-joins for each type of two-variable query.
- Set of types of one-variable queries and update transactions of interest with their respective frequencies. Here each type of one-variable query represents any Boolean combination of simple predicates. A simple predicate is one that refers to only one column of the relation.
- Data characteristics similar to the ones used in Design Step 1, but only those parameters that pertain to single relations are relevant.

Outputs:

- Set of indexes of each relation that gives the minimum processing time.

Algorithm 2:

1) Select one relation and start with an access configuration with the full index set.

2) Try to drop one index at a time, get the total cost, and 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 of the cost.

5) Try to drop two indexes at a time, get the total cost, and 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 of the cost.

8) Repeat steps 5), 6), and 7) with three indexes, four indexes, $\cdots$, up to k indexes at a time. (The variable k must be provided beforehand and can be adjusted to obtain a desired accuracy.)

C. Separability in Design Step 2

The implicit meaning of the index selection is that those indexes that do not compensate for their own maintenance and access cost should be dropped. In Design Step 2 we again considered relations singly and independently of one another. This was based on the separability theory of Theorem 2, i.e., that the access structures assigned to one relation do not affect cost calculations for other relations. However, since, in contrast to Design Step 1, we are eliminating some indexes, we can encounter situations that were excluded as exceptions in Example 4 and Theorem 4. In these situations, calculation of cost is no longer separable. Nevertheless, it turns out the calculation error caused by the assumption of separability, even in these exceptional situations, is not significant.

If we look at Example 4 again, the actual cost at query

processing time will be

$$\begin{aligned}\text{Cost} &= \min(\text{Cost1}, \text{Cost2}) \\ &= \min[\{\text{Cost}(R_1, J_1, \text{join-index}) \\ &\qquad + \text{Cost}(R_2, Cf_{12}, \text{join-index})\}, \\ &\qquad \{\text{Cost}(R_1, Cf_{21}, \text{join-index}) \\ &\qquad + \text{Cost}(R_2, J_2, \text{join-index})\}] .\end{aligned}$$

But because we assumed symmetry, the sum of the costs we used implicitly in Design Step 2 is

$$\begin{aligned}\text{Cost}' &= \text{Cost}(R_1, Cf_{12}, \text{join-index}) \\ &\quad + \text{Cost}(R_2, Cf_{21}, \text{join-index}) .\end{aligned}$$

Thus, the total error in cost estimation will be

$$\begin{aligned}\text{Error} &= g \times (\text{Cost} - \text{Cost}') \\ &= g \times \min[\{\text{Cost}(R_1, J_1, \text{join-index}) \\ &\qquad - \text{Cost}(R_1, Cf_{21}, \text{join-index})\} \\ &\qquad \{\text{Cost}(R_2, J_2, \text{join-index}) \\ &\qquad - \text{Cost}(R_2, Cf_{12}, \text{join-index})\}] \qquad (9)\end{aligned}$$

where g is the frequency of this join.

Remember, however, that the restriction indexes for both relations had been dropped because their benefits did not compensate for their update and access cost. Hence, it must be either that the frequency of access to the column is not significant, or that the effect of selectivity is small. Therefore, either the frequency of the join we are concerned with is insignificant or the coupling factor approaches the join selectivity, making the error insignificant [see (9)]. It is true that because of an unusually high update cost, an index can be dropped despite that the column has an effective selectivity, or the frequency of access to the column is high. We believe, however, that this situation does not frequently occur on both sides of a join in practice, and that the effect of this peculiarity to the overall design may not be significant. This argument has been supported in part by an extensive test performed for the validation of design algorithms based on the theory [17]. Following this argument, we claim that separability can be applied to all the cases of concern without causing any significant error. Similar situations arise when, on both relations, only some of the restriction columns specified in a query have indexes assigned, while others do not. A similar argument holds for such cases.

VII. Extensions and Further Study

An extension of nonseparable joins, for instance, the inner/outer-loop join method and the multiple-pass method described in Section III, could be made by means of the following heuristic method. After Design Step 1, each type of two-variable query is considered in turn and its join cost, as determined in Design Step 1, is compared to possible nonseparable joins. If a nonseparable join is cheaper, that query type should be marked to note that this nonseparable join must be used. For a possible shift of the clustering column, after completion of this step, Design Step 1 should be repeated, with the join method for a marked query type fixed to be the nonseparable join method assigned previously. This whole procedure (Design Step 1 and the refinement step with nonseparable join methods) is repeated until the refinement becomes insignificant.

The link structure [4] can be considered next. For every join path, the total cost of all queries using this join path is compared to the cost based on a hypothetical link. If the latter is less, a link is assigned to that join path. If the join column on the N-side relation of the 1-to-N relationship is a clustering column, the link is endowed with the clustering property; otherwise not.

The most attractive prospects for the inner/outer-loop join methods are those queries that use the sort–merge method for the relation on the 1-side of the 1-to-N relationship, but use the join index method for the other side. Use of the inner/outer loop join method in these cases has the advantage of saving sorting time on one relation and index-searching time on the other (if it has a strong coupling factor). On the other hand, join paths that support many queries using the inner/outer-loop join method would be the most promising prospects for the link structure. Index selection could be done at the conclusion of these steps.

Finally, although we have developed our theory in terms of the relational system, it should be pointed out that the basic concept of separability is applicable to network database systems as well (Theorem 2 holds for any system, while Theorems 3 and 4 are relevant only for relational systems).

VIII. Conclusion

It has been observed and proven that with a separable set of join methods, the problem of designing the optimal physical database can be reduced to one of designing optimal individual relations. This can be done independently of one another by using the coupling factors that represent all interactions among the relations. This substantially diminishes the complexity of the problem by partitioning it into disjoint subproblems. The task is made even more manageable by dividing the procedure into two steps — one for determining the optimal positions of clustering columns, the other for index selection. A proper interface between the two steps was introduced.

Design Step 1 results in a true mathematical optimum. Although because of the heuristics used in Design Step 2 and for the interface between the two steps the overall design does not provide a true optimum, it was argued that the deviation would be insignificant.

The key objective of this paper is to propose a formal approach to the design of physical databases that simplifies the problem considerably, and at the same time provides better insight into underlying mechanisms. We believe that this novel approach can enable substantial progress to be made in the optimal design of multifile physical databases.

Acknowledgment

The authors wish to thank the referees for their many thoughtful comments on the completeness of the contents and

the presentation of the paper, which helped to enhance the readability significantly. The authors also wish to acknowledge invaluable discussions with S. Finkelstein and R. El-Masri. T. Pickering helped with editing.

REFERENCES

[1] J. Smith and P. Chang, "Optimizing the performance of a relational algebra database interface," *Commun. Ass. Comput. Mach.*, vol. 18, pp. 568–579, Oct. 1975.
[2] R. M. Pecherer, "Efficient evaluation of expression in a relational algebra," in *Proc. Ass. Comput. Mach. Pacific 75 Regional Conf.*, San Francisco, CA, Apr. 1975, pp. 44–49.
[3] L. Gotlieb, "Computing joins of relations," in *Proc. Int. Conf. on Management of Data*, San Jose, CA, May 1975, pp. 55–63.
[4] M. W. Blasgen and K. P. Eswaran, "On the evaluation of queries in a database system," IBM Res. Rep. RJ1945, IBM, San Jose, CA, Apr. 1976.
[5] S. B. Yao, "Optimization of query evaluation algorithm," *Ass. Comput. Mach. Trans. Database Syst.*, vol. 4, no. 2, pp. 133–155, June 1979.
[6] P. G. Selinger *et al.*, "Access path selection in a relational database management system," in *Proc. Int. Conf. on Management of Data*, Boston, MA, May 1979, pp. 23–34.
[7] D. S. Batory and C. C. Gotlieb, "A unifying model of physical databases," Computer Systems Research Group, University of Toronto, Tech. Rep. CSRG-109, Apr. 1980.
[8] A. F. Cardenas, "Analysis and performance of inverted database structures," *Commun. Ass. Comput. Mach.*, vol. 18, pp. 253–263, May 1975.
[9] M. Hammer and A. Chan, "Index selection in a self-adaptive database management system," in *Proc. Int. Conf. on Management of Data*, Washington, DC, ACM SIGMOD, June 1976, pp. 1–8.
[10] D. Hsiao and F. Harary, "A formal system for information retrieval from files," *Commun. Ass. Comput. Mach.*, vol. 13, no. 4, pp. 67–73, Feb. 1970; also see *Commun. Ass. Comput. Mach.*, vol. 13, no. 4, p. 266, Apr. 1970.
[11] M. Schkolnick, "The optimal selection of secondary indices for files," *Inform. Syst.*, vol. 1, pp. 141–146, Mar. 1975.
[12] D. G. Severance, "A parametric model of alternative file structures," *Inform. Syst.*, vol. 1, no. 2, pp. 51–55, 1975.
[13] S. B. Yao, "An attribute based model for database access cost analysis," *Ass. Comput. Mach. Trans. Database Syst.*, vol. 2, no. 1, pp. 45–67, Mar. 1977.
[14] T. J. Gambino and R. Gerritsen, "A database design decision support system," in *Proc. Int. Conf. on Very Large Databases*, Tokyo, Japan, IEEE, Oct. 1977, pp. 534–544.
[15] R. H. Katz and E. Wong, "An access path model for physical database design," in *Proc. Int. Conf. on Management of Data*, Santa Monica, CA, ACM SIGMOD, May 1980, pp. 22–29.
[16] R. Gerritsen *et al.*, "Cost effective database design: An integrated model," Decision Sciences Working Paper 77-12-03, Wharton School, Univ. of Pennsylvania, 1977.
[17] K. Whang, "A Physical Database Design Methodology Using the Property of Separability," Ph.D. dissertation, Stanford University, Stanford, CA, 1983, Rep. STAN-CS-83-968.
[18] G. Wiederhold, *Database Design*. New York: McGraw-Hill, 1983.
[19] G. Wiederhold and R. El-Masri, "The structural model for database design," in *Proc. Int. Conf. on Entity Relationship Approach*, Los Angeles, CA, Dec. 1979, pp. 247–267.
[20] K. Whang, G. Wiederhold, and D. Sagalowicz, "Separability as an approach to physical database design," Tech. Rep. STAN-CS-81-898, Stanford University, Stanford, CA, Oct. 1981. Also numbered CSL TR-222.
[21] J. Ullman, *Principles of Database Systems*. Potomac, MD: Computer Science Press, 1982.
[22] A. M. Keller, "Updates to relational databases through views involving joins," IBM Res. Rep. RJ3282, IBM, San Jose, CA, Oct. 1981.
[23] D. D. Chamberlin *et al.*, "SEQUEL2: A unified approach to data definition, manipulation, and control," *IBM J. Res. Devel.*, vol. 20, no. 6, pp. 560–575, Nov. 1976.
[24] W. Kim, "On optimizing an SQL-like nested query," *Ass. Comput. Mach. Trans. Database Syst.*, vol. 7, no. 3, pp. 443–469, Sept. 1982.
[25] R. Bayer and E. McCreight, "Organization and maintenance of large ordered indices," *Acta Inform.*, vol. 1, 1972.
[26] D. Knuth, *The Art of Computer Programming — Sorting and Searching*, vol. 3. Reading, MA: Addison-Wesley, 1973.
[27] K. Whang, G. Wiederhold, and D. Sagalowicz, "Estimating block accesses in database organizations — A closed noniterative formula," *Commun. Ass. Comput. Mach.*, vol. 26, pp. 940–944, Nov. 1983.
[28] R. El-Masri and G. Wiederhold, "Properties of relationships and their representation," in *Proc. Nat. Comput. Conf.*, AFIPS, vol. 49, May 1980, pp. 191–192.
[29] W. F. King, "On the selection of indices for a file," IBM Res. Rep. RJ1341, IBM, San Jose, CA, 1974.
[30] M. Stonebraker, "The choice of partial inversions and combined indices," *Int. J. Comput. Inform. Sci.*, vol. 3, no. 2, pp. 167–188, 1974.
[31] G. Held, "Storage structures for relational data base," Ph.D. dissertation, University of California, Berkeley, 1972.
[32] E. Feldman *et al.*, "Warehouse location under continuous economies of scale," *Management Sci.*, vol. 12, no. 9, pp. 670–684, July 1966.
[33] A. A. Kuehn and M. J. Hamburger, "A heuristic program for locating warehouses," *Management Sci.*, vol. 10, pp. 643–657, July 1963.
[34] K. Whang, G. Wiederhold, and D. Sagalowicz, "Physical design of network model databases using the property of separability," in *Proc. Int. Conf. on Very Large Databases*, Mexico City, Mexico, Sept. 1982, pp. 98–107.

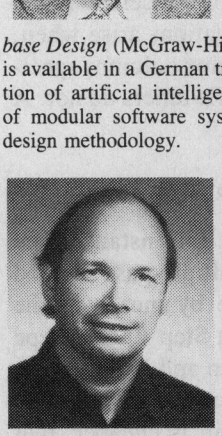
Kyu-Young Whang was born in Seoul, Korea on March 2, 1951. He received the B.S. degree in electrical engineering from Seoul National University in 1973 from which he graduated summa cum laude, the M.S. degree in electrical engineering from the Korea Advanced Institute of Science and from the Computer Systems Laboratory, Stanford University, Stanford, CA, in 1975 and 1982, respectively, and received the Ph.D. degree from the Computer Systems Laboratory, Stanford University in 1984.

From 1975 to 1978, he served as a Research Engineer and a Senior Research Engineer in the Agency for Defense Development, Korea. In October 1983 he joined the IBM T. J. Watson Research Center, Yorktown Heights, NY, as a Research Staff Member where he has been engaged in research and development of an office system based on a relational database management system. During his study at Stanford, he worked in various projects including compiler development, development and analysis of various counting algorithms with an application to relational database systems (work done at IBM San Jose Research Center), and development of a VLSI design system (in cooperation with Silvar-Lisco), as well as his dissertation research on physical database design. His research interests encompass physical and logical database design, database systems, distributed database systems, computer networks, and database applications to VLSI design systems.

Dr. Whang is a member of the Association for Computing Machinery and the IEEE Computer Society.

Gio Wiederhold completed his undergraduate education in The Netherlands. He received the Ph.D. degree from the University of California, San Francisco, in 1976.

After coming to the U.S. he worked as a Programmer, Project Leader, Manager, Director, and Consultant for a number of companies, as well as for the University of California, Berkeley, and Stanford University, Stanford, CA. He is currently an Associate Professor of Medicine and Computer Science (Research) at Stanford University. His book *Database Design* (McGraw-Hill) is widely used as a textbook, and its first edition is available in a German translation. His current research includes the application of artificial intelligence techniques to databases, and the development of modular software systems, as well as the development of a database design methodology.



Daniel Sagalowicz (M'73) earned the engineering degree from the Ecole Nationale Superieure des Telecommunications in France in 1966, and the M.S. and Ph.D. degrees from Stanford University, Stanford, CA, in 1967 and 1970, respectively.

After doing some systems development at Honeywell-Bull in Paris, he joined the Artificial Intelligence Center at SRI International, Menlo Park, CA, where he was Assistant Director when the research and the writing of this paper occurred. His general area of interest is the application of artificial intelligence to the database area. In particular, he has been involved in research on natural language access to databases, conceptual modeling of information contained in databases, use of semantic knowledge to improve information retrieval, and use of heuristic techniques in the design of physical databases. He is currently working for Teknowledge. He serves as the Chief Technical Officer of Framentec, a joint subsidiary of Teknowledge and Framatome based in Monaco.