Physical Database Design for Relational Databases

S. FINKELSTEIN, M. SCHKOLNICK, and P. TIBERIO
IBM Almaden Research Center

This paper describes the concepts used in the implementation of DBDSGN, an experimental physical design tool for relational databases developed at the IBM San Jose Research Laboratory. Given a workload for System R (consisting of a set of SQL statements and their execution frequencies), DBDSGN suggests physical configurations for efficient performance. Each configuration consists of a set of indices and an ordering for each table. Workload statements are evaluated only for atomic configurations of indices, which have only one index per table. Costs for any configuration can be obtained from those of the atomic configurations. DBDSGN uses information supplied by the System R optimizer both to determine which columns might be worth indexing and to obtain estimates of the cost of executing statements in different configurations. The tool finds efficient solutions to the index-selection problem; if we assume the cost estimates supplied by the optimizer are the actual execution costs, it finds the optimal solution. Optionally, heuristics can be used to reduce execution time. The approach taken by DBDSGN in solving the index-selection problem for multiple-table statements significantly reduces the complexity of the problem. DBDSGN's principles were used in the Relational Design Tool (RDT), an IBM product based on DBDSGN, which performs design for SQL/DS, a relational system based on System R. System R actually uses DBDSGN's suggested solutions as the tool expects because cost estimates and other necessary information can be obtained from System R using a new SQL statement, the EXPLAIN statement. This illustrates how a system can export a model of its internal assumptions and behavior so that other systems (such as tools) can share this model.

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

General Terms: Algorithms, Design, Performance

Additional Key Words and Phrases: Index selection, physical database design, query optimization, relational database

1. INTRODUCTION

During the past decade, database management systems (DBMSs) based on the relational model have moved from the research laboratory to the business place. One major strength of relational systems is ease of use. Users interact with these systems in a natural way using nonprocedural languages that specify what data

Authors' present addresses: S. Finkelstein, Department K55/801, IBM Almaden Research Center, 650 Harry Road, San Jose, CA 95120-6099; M. Schkolnick, IBM Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598; P. Tiberio, Dipartimento di Elettronica, Informatica e Sistemistica, University of Bologna, Viale Risorgimento 2, Bologna 40100, Italy.

are required, but do not specify how to perform the operations to obtain those data. Statements specify which tables should be accessed as well as conditions restricting which combinations of data from those tables are desired. They do not specify the access paths (e.g., indices) used to get data from each table, or the sequence in which tables should be accessed. Hence, relational statements (and programs with embedded relational statements) can be run independent of the set of access paths that exist.

There has been controversy about how well relational systems would perform compared to other DBMSs, especially in a transaction-oriented environment. Critics of relational systems point out that their nonprocedurality prevents users from navigating through the data in the ways they believe to be most efficient. Developers of relational systems claim that systems could be capable of making very good decisions about how to perform users' requests based on statistical models of databases and formulas for estimating the costs of different execution plans. Software modules called *optimizers* make these decisions based on statistical models of databases. They perform analysis of alternatives for executing each statement and choose the execution plan that appears to have the lowest cost. Two of the earliest relational systems, System R, developed at the IBM San Jose Research Laboratory [4, 5, 10, 11] (which has moved and is now the IBM Almaden Research Center), and INGRES, developed at the University of California, Berkeley [37], have optimizers that perform this function [35, 40]. Optimizer effectiveness in choosing efficient execution plans is critical to system response time. Initial studies on the behavior of optimizers [2, 18, 27, 42] have shown that the choices made by them are among the best possible, for the set of access paths. Optimizers are likely to improve, especially since products have been built using them [20, 22, 29].

A relational system does not automatically determine the set of access paths. The access paths must be created by authorized users such as database administrators (DBAs). Access-path selection is not trivial, since an index designer must balance the advantages of access paths for data retrieval versus their disadvantages in maintenance costs (incurred for database inserts, deletes, and updates) and database space utilization. For example, indexing every column is seldom a good design choice. Updates will be very expensive in that design, and moreover, the indices will probably require more total space than the tables. (The reasons why index selection is difficult are discussed further in Section 2.1.) Database system implementers may be surprised by which index design is best for the applications that are run on a particular database. Since those responsible for index design usually are not familiar with the internals of the relational system, they may find the access-path selection problem very difficult. A poor choice of physical designs can result in poor system performance, far below what the system would do if a better set of access paths were available. Hence, a design tool is needed to help designers select access paths that support efficient system performance for a set of applications.

Such a design tool would be useful both for initial database design and when a major reconfiguration of the database occurs. A design tool might be used when

—the cost of a prospective database must be evaluated,

—the database is to be loaded,

—the workload on a database changes substantially,
—new tables are added,
—the database has been heavily updated, or
—DBMS performance has degraded.

In System R, indices (structured as B^+-trees [14]) are the only access paths to data in a table (other than sequentially scanning the entire table). Each index is based on the values of one or more of the columns of a table, and there may be many indices on each table. Other systems, such as INGRES and ORACLE [34], also allow users to create indices. In addition, INGRES allows hashing methods. One of the most important problems that a design tool for these systems must solve is selecting which indices (or other access paths) should exist in the database [31, 41]. Although many papers on index selection have appeared, all solve restricted versions of the problem [1, 6–8, 16, 17, 23, 25, 26, 28, 30, 36, 39]. Most restrictions are in one of the following areas:

(1) *Multiple-table solutions.* Some papers discuss methodologies for access-path selection for statements involving a single table, but do not demonstrate that their methodologies can be extended effectively to statements on multiple tables. One multitable design methodology was proposed based on the cost separability property of some join methods. When the property does not hold, heuristics are introduced to extend the methodology [38, 39].

(2) *Statement generality.* Many methodologies limit the set of user statements permitted. Often they handle queries whose restriction criteria involve comparisons between columns and constants, and are expressed in disjunctive normal form. Even when updates are permitted, index and tuple maintenance costs are sometimes not considered. When they are, they are usually viewed as independent of the access paths chosen for performing the maintenance.

(3) *Primary access paths.* Often the primary access path is given in advance, and methods are described for determining *auxiliary* access paths. This means that the decision of how to order the tuples in each table has already been made. However, the primary access path is not always obvious, nor is it necessarily obvious which statements should use the primary access path and which should use auxiliary paths.

(4) *Disagreement between internal and external system models.* This problem occurs only in systems with optimizers. The optimizer's internal model consists of its statistical model of statement execution cost and the way it chooses the execution plan it deems best. The optimizer calculates estimates of cost and cardinality based on its internal model and the statistics in the database catalogs. A design tool may use an external model independent of the model used by the optimizer. This approach has several serious disadvantages: The tool becomes obsolete whenever there is a change in the optimizer's model, and changes in the optimizer are likely as relational systems improve. Moreover, the optimizer may make very different assumptions (and hence different execution-plan choices) from those made by the external model. Even if the external model is more accurate than the optimizer's model, it is not good to use an external model, since the optimizer chooses plans based on its own model.

We believe a good design tool should deal with all the above issues. It should choose the best set of access paths for any number of tables, accept all valid input statements, solve the combined problem of record placement and access-path selection, and *use the database system* to obtain both statistics (when the database tables exist) and cost estimates [32]. When the database does not exist yet, the tool should accept a statistical description of the database from the designer and obtain cost estimates based on those statistics from the database system.

In this paper we discuss the basic principles we considered in constructing an experimental design tool, DBDSGN, that runs as an application program for System R. In creating DBDSGN we have attempted to meet all the requirements described above. We have also discovered some general principles governing design-tool construction, and have learned how a DBMS should function to support design tools. These principles have been adopted in the Relational Design Tool (RDT) [19]. RDT is an IBM product, based on DBDSGN, which performs design for SQL/DS [20], a relational system based on System R.

We developed the methodology for the index-selection problem for System R, but did not forget the more general problem of access-path selection for systems with hashing and links as well. We discuss the extension of the DBDSGN methodology to these access paths in Section 7. DBDSGN's major limitation is its assumption that only one access path can be used for each different occurrence of a table in a statement; this assumption is false for systems using tuple identifier (TID) intersection methods. We believe the concepts and results that arose from designing and implementing this tool are also valid for different DBMSs with other access paths; some of the concepts may also be valuable for designing integrated system families where large systems export descriptions of their internal assumptions and behaviors so that other systems (such as tools) can share them.

We assume the reader is familiar with relational database technology and standard query languages used in relational systems. We use SQL [9] as the query language.

2. THE PROBLEM OF INDEX SELECTION IN RELATIONAL DATABASES

2.1 Problem Complexity

Data in a database table can be accessed by scanning the entire table (sequential scan). The execution of a given statement may be sped up by using auxiliary access paths, such as indices. However, the existence of certain indices, although improving the performance of some statements, may reduce the performance of other statements (such as updates), since the indices must be modified when tables are. In System R, some indices, called *clustered* indices, enforce the ordering of the records in the tables they index. All other indices are called *nonclustered* indices. The overall performance of the system depends on the set of all existing indices, as well as on the ways the tables are stored. Although System R supports multicolumn indices (as described in Section 7), this paper focuses on indices on single columns.

Given a set of tables and a set of statements, together with their expected frequencies of use, the *index-selection problem* involves selecting for each table

—the *ordering* rule for the stored records (which determines the *clustered index*, if any), and
—a set of nonclustered indices,

so as to minimize the total processing cost, subject to a limit on total index space. We define the total processing cost to be the frequency weighted sum of the expected costs for executing each statement, including access, tuple update, and index maintenance costs. A weighted index space cost is also added in.

Clustered indices frequently provide excellent performance when they are on columns referenced in a given statement [2, 35]. This might indicate that the solution to the design problem is to have a clustered index on every column. Such a solution is not possible, since (without replication) records can be ordered only one way. On the other hand, nonclustered indices can exist on all columns and may help to process some statements. A set of clustered and nonclustered indices on tables in a database is called an *index configuration* (or more simply a *configuration*) if no table has more than one clustered index and no columns have both clustered and nonclustered indices. We will only be interested in index designs that are configurations. A configuration proposed for a particular index-selection problem it is called a *solution* for that problem.

It may seem that finding solutions to the design problem consists of choosing one column from each table as the ordering column, putting a clustered index on that column, and putting nonclustered indices on all other columns. This fails for three reasons:

(1) For each additional index that exists, extra maintenance cost is incurred every time an update is made that affects the index (inserting or deleting records, updating the value of the index's column). Because of the cost of maintenance activity, a solution with indices on every column of every table usually does not minimize processing costs.

(2) Storage costs must be considered even when there are no updates. Typically, a System R index utilizes from 5 to 20 percent of the space used by the table it indexes, so the cost of storage is not negligible.

(3) Most importantly, a global solution cannot generally be obtained for each table independently. Any index decision that you make for one table (e.g., which index is clustered) may affect the best index choices for another table. Some examples showing the interrelationship among index choices are given in Section 4.

These considerations show that the design problem presented at the beginning of this section does not have a simple solution. Even a restricted version of the index-selection problem is in the class of NP-hard problems [13]. Thus, there appears to be no fast algorithm that will find the *optimal solution*. However, we must question whether the optimal solution is the right goal, since the problem specification and the problem that the designer actually wants solved usually are

not identical. Approximations include

—the statements that are the input for the problem usually represent an approximation to the actual load that will be submitted to the system,

—the frequencies associated with these statements are likely to be approximations,

—the statistics for the data the tool uses (which may be given by the designer or derived from the database itself) represent the data as they exist at the time the design is done and may not accurately reflect future changes, and

—the statistical model used by the optimizer is correct only for some data distributions. Imprecision exists when the actual data do not fit the underlying assumptions of the model [2, 12].

For these reasons, instead of finding the optimal solution to the index design problem, we would like to get a set of *reasonable designs*, each of which has a relatively low performance cost. From this set a designer can choose the one he or she deems best, based on considerations that may not have been completely modeled. By an appropriate use of some heuristics, combined with more exact techniques, DBDSGN can find a set of reasonable solutions quickly. The designer may iterate through several executions of some of DBDSGN's phases, tuning simple heuristic parameters to try to achieve better solutions (at the expense of additional execution time). A discussion of some of these techniques appears in this paper.

2.2 A Methodology for Index Selection

Methodologies for the index-selection problem are based on models of data retrieval and update. Some solve the problem in a wholly analytic way; others use heuristic searches to find a quasi-optimal solution. However, all previous examples compute the estimated costs of retrievals and updates using analytic formulas. Since we assume the database management system uses an optimizer to choose an access-path strategy, it makes sense to *use the optimizer itself to provide the estimated processing cost of a given statement.* The optimizer examines the set of access paths that exist and computes the best expected cost for a statement by evaluating different join orders, join methods, and access choices. By using the optimizer's cost estimates as the basis for our design tool, we obtain three significant advantages.

First, the tool is independent of optimizer improvements. An analytic expression for the cost of performing a given statement must be based on current knowledge of the strategy used by the optimizer and will become invalid if the optimizer computations are altered. For example, suppose a statement includes a predicate on a column for which there is a nonclustered index. An early version of the System R optimizer determined the cost of accessing the tuples using the nonclustered index by assuming that a data page was read for each retrieved tuple [35]. In a later version of the system, the optimizer recognized that the TIDs are stored in increasing order, so a smaller number of estimated page hits results when the number of tuples for a given key value is comparable to the number of data pages [2]. This type of change would have an immediate impact on a tool that used an analytic model of the optimizer's behavior. As another example, two systems based on System R, SQL/DS [20] and DB2 [22], have

different physical data managers, which lead to differences in their optimizer cost models that a design tool should not need to know about.

Second, the query may be transformed to an equivalent form before it reaches the optimizer (or by the optimizer itself). For example, nested queries may be transformed to joins [15, 24]. A tool using an external model may not understand these transformations; even if it does, it will have to be changed when the transformations change.

Third, using the optimizer we can guarantee any proposed solution is one the optimizer will use to its full advantage. Working with an external model could result in a solution that has good performance according to the analytic model. However, when the optimizer is confronted with the set of access paths described in the solution it may choose an execution plan different from the one predicted by the tool, which may result in poor performance. To illustrate this, consider an example involving the table

ORDERS: (ORDERNO, SUPPNO, PARTNO, DATE, QTY, ...)

in the statement

(S1)

```
SELECT   ORDERNO, SUPPNO
FROM     ORDERS
WHERE    PARTNO = 274
AND      DATE BETWEEN 870601 AND 870603.
```

An external model based on more detailed statistics than those available to the optimizer might suggest that an index I_{DATE} on DATE performs much better than an index I_{PARTNO} on PARTNO (which might have been created for another statement). But the optimizer might choose I_{PARTNO} instead, so that the index I_{DATE} is useless. Even worse, the external model could suggest solutions that are poor because the optimizer makes unexpected choices. Thus, we believe that attempts to outsmart the optimizer are misguided. Instead, the optimizer itself should be improved.

A design tool can interact with the DBMS to collect information without physically running a statement by using the SQL EXPLAIN facility [20, 21], a new SQL statement originally prototyped by us for System R. EXPLAIN causes the optimizer to choose an execution plan (including access paths) for the statement being EXPLAINed and to store information about the statement in the database in explanation tables belonging to the person performing EXPLAIN. These tables can then be accessed and summarized using ordinary queries. The system does not actually execute the EXPLAINed statement, nor is a plan for executing that statement stored in the database. Actually executing statements would determine the actual execution costs for a particular configuration, but executing each statement for each different index combination is unacceptably expensive in nontrivial cases. (When we speak of *costs* in the rest of this paper, we mean the optimizer's cost estimates; actual execution costs are explicitly referenced as such.)

The four options for EXPLAIN are REFERENCE, STRUCTURE, COST, and PLAN. EXPLAIN REFERENCE identifies the statement type (Query, Update, Delete, Insert), the tables referenced in the statement, and the columns

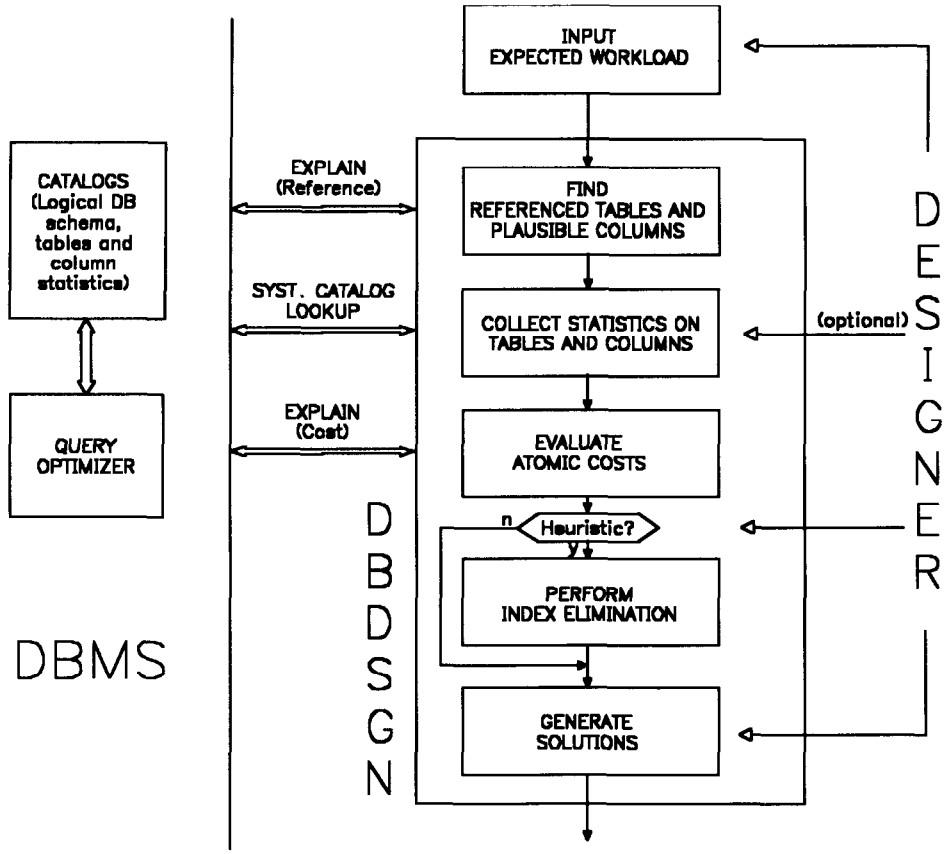

Fig. 1. Architecture of DBDSGN.

referenced in the statement in ways that influence their plausibility for indexing. EXPLAIN STRUCTURE identifies the structure of the subquery tree in the statement, the estimated number of tuples returned by the statement and its subqueries, and the estimated number of times the statement and its subqueries are executed. EXPLAIN COST indicates the estimated cost of execution of the statement and its subqueries in the plan chosen by the optimizer. EXPLAIN PLAN describes aspects of the access plan chosen by the optimizer, including the order in which tables are accessed for executing the statement, the access paths used to access each table, the methods used to perform joins (nested loop, merge scan), and the sorts performed.

DBDSGN has five principal steps. Figure 1 shows an overall description of the architecture of the design tool and identifies its major interactions with the designer and the DBMS.

(1) *Find referenced tables and plausible columns.* Based on an analysis of the structure of the input statements obtained using EXPLAIN, we allow only the columns that are "plausible for indexing" to enter into the design process. (Different columns may be plausible for different statements.) The designer

indicates which tables should be designed for and which should remain as they are.

(2) *Collect statistics on tables and columns.* Statistics are either provided by the designer or extracted from the database catalogs.

(3) *Evaluate atomic costs.* Certain index configurations are called atomic because costs of all configurations can be obtained from their costs. The EXPLAIN facility is used to obtain the costs of these atomic configurations (which are called atomic costs).

(4) *Perform index elimination.* If the problem space is large, a heuristic-based dominance criterion can be invoked to eliminate some indices and to reduce the space searched during the last step.

(5) *Generate solutions.* A controlled search of the set of configurations leads to the discovery of good solutions. The designer supplies parameters that control this search.

3. COST MODEL

3.1 Workload Model

When a designer is asked to supply an index design for a database, he or she must determine the workload that is expected for that system over a specified time period. The expected workload during that period is characterized by a set of pairs

$$W = \{(q_i, w_i), i = 1, 2, \ldots, q\},$$

where each q_i is a statement expressed in the DBMS's language and each w_i is its assigned weight. The term *statement* refers to queries (both single-table queries and multitable joins), updates, inserts, and deletes.

The q_i are the statements that the designer expects to be relatively important during the time period. The statements in the workload W may come from different sources:

—predictable ad hoc statements that will be issued from terminals,
—old application programs that will be executed during the period, or
—new application programs that will be executed during the period.

The weight w_i associated with each statement is a function of

—the frequency of execution of the statement in the period, or
—system load when the statement is run (e.g., statements that can be run off-shift may be given smaller weights, and statements that require particularly fast response time may be given larger weights).

Different statements that are treated identically by the optimizer could be combined, although this requires special knowledge of the optimizer. For example, a System R query with the predicate PARTNO = 274 could be combined with a query with the predicate PARTNO = 956 since the predicates have the same selectivity (the reciprocal of the number of different PARTNO values). Either query could be included in the workload, with the sum of the original weights specified. A query with PARTNO < 274, however, could not be combined with

one requesting PARTNO < 956, since the System R optimizer associates different selectivities with these predicates.

For application programs, the assignment of the weights is a difficult problem. In general, as we mentioned in Section 2.1, frequencies must be approximated. Designers may know how often an application will be run, but may find it difficult to predict the frequency of execution of a statement due to the complexity of program logic. Furthermore, there can be statements like the "CURRENT OF CURSOR" statement in SQL, in which tuples are fetched under the control of the calling program, and the "SELECT FOR UPDATE" statement where the decision to update depends on both program variables and tuple content. For applications that already run on the database, a performance monitor can help solve this problem.

3.2 Atomic Costs

This section describes some aspects of the behavior of the System R optimizer. A tool like DBDSGN could be used for other relational systems if they follow the principles described in this section. It is not the aim of this paper to describe how the optimizer makes its decisions. For a more detailed description, the reader is referred to other papers [2, 35]. The basic principles used by the System R optimizer in processing a given statement are as follows:

Optimizer principles

(P1) Exactly one access path is used for each appearance of a table in the statement.

(P2) The costs of all combinations using one access path per table appearance are computed, and the one with the minimal cost is chosen.

Principle (P1) would not be true of a system that used conjunction of indices on a single table (such as TID intersection, which System R does not support). Principle (P2) might not be true for an optimizer that used heuristics to limit its search for the plan with the smallest expected execution cost. Principle (P2) can be relaxed slightly. It is not necessary for the optimizer to compute all costs, as long as it finds the plan with the smallest expected cost.

The cost of executing a statement consists of three components: tuple access cost, tuple maintenance cost, and index maintenance cost. In this section we consider only the access costs; we deal with maintenance costs in the next section. To clarify the above principles, first consider a statement on a single table that has n indices. The optimizer computes $n + 1$ access costs (n using each single index, and 1 using sequential scan) and chooses the access path with the minimal cost. The access costs are computed independently, since the presence of a given index cannot influence the computation of the cost of accessing the table through another index (since by principle (P1) only one index per table can be used).

Now consider a statement q that is a t-table join, where I_j is the set of indices on the jth table. Let $C_q(\alpha_1, \alpha_2, \ldots, \alpha_t)$ be the optimizer's best (smallest) cost of executing q when the access paths $\alpha_1, \alpha_2, \ldots, \alpha_t$ are used, where α_j is either one of the indices in I_j or sequential scan ρ. The tables may be accessed in many orders, and many join methods are possible even when the access paths are fixed. Because of the Optimizer principles, we can think of the optimizer as if it

calculated each $C_q(\alpha_1, \alpha_2, \ldots, \alpha_t)$ independently. The choice it selects for execution is the one with the minimal estimated cost, so we define

$$\text{COST}_q(I_1, I_2, \ldots, I_t) = \min_{\alpha_j \in I_j \cup \{\rho\}} C_q(\alpha_1, \alpha_2, \ldots, \alpha_t).$$

$\text{COST}_q(I_1, I_2, \ldots, I_t)$ is the cost that the optimizer returns to the design tool. Let ISET denote the collection of indices that exist on a set of tables. For the index configuration ISET, we write $\text{COST}_q[\text{ISET}]$ to represent $\text{COST}_q(I_1, I_2, \ldots, I_n)$, where I_j is the set of indices in ISET that are on the jth table. Indices in ISET on tables not referenced in the statement do not affect $\text{COST}_q[\text{ISET}]$. For a single-table statement against a table with n columns, we can build $n2^{n-1} + 2^n$ different index configurations. (There are n clustering choices, and for each of these, there are 2^{n-1} different nonclustered sets. If no clustered index is chosen, there are 2^n sets of nonclustered indices.) For a join query, the number of configurations is the product of the number of configurations on each table, which is exponential in the total number of columns in the tables.

Configurations with at most one index per table are called *atomic configurations*, and their costs are called *atomic costs*, since (as we shall show) costs for all other configurations can be computed from them.[1] Atomic configurations for a table (or set of tables) are atomic configurations where indices are only on that table (or set of tables). Atomic configurations for a statement are configurations that are atomic for the tables in that statement.

PROPOSITION 1. *The cost of a query (single-table query or join) for a configuration is the minimum of the costs for that query taken over the atomic configurations that are subsets of the configuration. More formally,*

$$COST_q[ISET] = \min_{ASET \subseteq ISET} COST_q[ASET]$$

(where the ASETs are atomic).

This proposition follows from the definition of COST_q. $\text{COST}_q[\text{ISET}]$ is the minimum of the $C_q(\alpha_1, \alpha_2, \ldots, \alpha_t)$ values, where the αs are access paths over appropriate tables (and any α can be sequential scan). Similarly replacing $\text{COST}_q[\text{ASET}]$ by its definition, each $C_q(\alpha_1, \alpha_2, \ldots, \alpha_t)$ appears in the right-hand side minimum at least once, and the C_q terms involving sequential scan appear more than once. Since both minimums are over the same set of C_q terms, they are equal, proving the proposition.

Performing EXPLAIN COST only for atomic configurations significantly reduces the number of cost inquiries to the optimizer performed by DBDSGN. For a query on a table with n columns, there are $2n + 1$ atomic configurations (n with 1 clustered index, n with 1 nonclustered index, and the configuration with no indices), so the number of EXPLAIN COSTs is reduced from exponential to linear in the number of columns. For a t-table join, recall that the number of configurations is exponential in the total number of columns in the joined tables. The number of atomic configurations for a join equals the product of the number

[1] Configurations with more than one index per table are admitted to evaluate statements with self-joins (when a table is joined with itself), but for simplicity we omit discussion of this case.

of atomic configurations for each single table. That is, if we let n_j be the number of columns in the jth table of the join, there are $\Pi_{j=1}^{t}(2n_j + 1)$ atomic configurations for the join. Despite this significant reduction, the computation of all atomic costs may still be impractical for large n_j and t. In Sections 3.4, 4.1, and 4.2, we describe methods to reduce the number of indices considered when atomic costs are computed.

SQL also permits statements with nested subqueries. Each statement subquery can be treated independently from the others (except for the execution frequency [35]), even when a subquery references a table appearing higher up in the subquery tree (a "correlated" subquery). This is because EXPLAIN provides separate information about each subquery in the subquery tree. In particular, DBDSGN uses EXPLAIN STRUCTURE to determine the subquery structure of each statement and the number of times each subquery is performed. DBDSGN uses this together with subquery cost information returned by EXPLAIN COST to compute the cost for the entire query and each subquery.

3.3 Maintenance Costs

Maintenance statements in System R can involve only a single table. (Maintenance statements may have subqueries, but DBDSGN handles them separate from the root of the subquery tree, just as it does when the root is a query.) These statements have three steps:

(1) Using some access path(s), the tuples acted upon are found (or the locations for inserted tuples are found).
(2) The tuples are modified, deleted, or inserted.
(3) Indices on the table are updated, if necessary.

The cost of maintaining indices may be substantial, so a design tool must consider the cost of performing this maintenance when it evaluates a physical design. Furthermore, the maintenance cost cannot be considered constant for every index. In [33] the following is shown:

(1) The maintenance cost depends on the form of the statement, such as the predicates in the **WHERE** clause, and the contents of the **SET** clause for update statements.

(2) Another distinction in cost computation must be made based on the way the tuples and indices to be modified are accessed. In particular, the access path determines the order in which the tuples in the data pages (and the TIDs in the index leaf pages) are scanned. Different formulas apply based on whether or not these objects are scanned in the same order they are stored.

In [33] the different cases are described and formulas for maintenance cost are given. DBDSGN takes all of the above issues into account.

We separate the costs returned by the optimizer for a maintenance statement into two components:

(1) the cost of accessing and modifying tuples, and
(2) the cost of maintaining indices on columns that are affected by the statement.

The notion of atomic cost is also valid for maintenance statements, and we distinguish between the *atomic access costs* (which we define as the sum of the costs of accessing and modifying tuples) and the *atomic index maintenance costs.* Fortunately, a small set of atomic index maintenance costs determines the cost of maintenance statements for any set of indices in the database. DBDSGN must determine the cost of updating any index, no matter what access path is used to access the tuples. The important distinction is not which access path is used, but whether the access path and updated index are ordered in the same way. When they are, this is called an *ordered scan*; when they are not, this is called an *unordered scan.* For instance, for a clustered index the scan is ordered if the access path is either that same index (which can occur for inserts and deletes) or sequential scan;[2] in these cases, the modifications follow the order in which the TIDs are stored in the index leaves. If the clustered index is updated following a scan on a nonclustered index instead, the TIDs may be hit in an unordered way, incurring a higher cost. For updating a nonclustered index, the only ordered scan is the index itself.

Let ISET be a set of indices, and let q be a maintenance statement. Since q can involve only one table (although subqueries can mention other tables), we assume without loss of generality that ISET is only on the modified table. Because of the Optimizer principles, the optimizer's cost estimate for executing maintenance statement q in configuration ISET is

$$\mathrm{COST}_q[\mathrm{ISET}] = \min_{\alpha \in \mathrm{ISET} \cup \{\rho\}} \left[C_q(\alpha) + \sum_{\beta \in \mathrm{ISET}} U_q(\beta, \alpha) \right],$$

where C_q here is the cost of accessing and modifying tuples using access path α, and $U_q(\beta, \alpha)$ is the cost of updating index β if access path α is used as the access path to the table. As with queries, indices in ISET on tables not referenced in a maintenance statement do not affect $\mathrm{COST}_q[\mathrm{ISET}]$. The definition of COST_q above is consistent with the definition of COST_q in the previous section for single-table queries.

Let q be a statement on a single table (including updates, deletes, and inserts, as well as queries on a single table), and let $AP_q(\mathrm{ASET})$ be the access path chosen by the optimizer to process q in atomic configuration ASET (which is either ρ or the one index in ASET that is on the referenced table). The following proposition decomposes the cost of q for configuration ISET into the costs C_q and U_q for atomic configurations ASET included in ISET.

PROPOSITION 2. *Let*

$$COST'_q[ISET] = \min_{ASET \subseteq ISET} \left[COST_q[ASET] + \sum_{\beta \in ISET - ASET} U_q(\beta, AP_q(ASET)) \right]$$

[2] An UPDATE cannot use an index on an updated column as an access path in System R. When a column entry in a given tuple is modified, the TID associated with the tuple is removed from the group of TIDs following the old key value in the index on that column and inserted in the group of the new key value. Accessing the tuples through an index that is currently modified may lead to hitting the same TID more than once.

(where the ASETs are atomic). Then,

$$COST'_q[ISET] = COST_q[ISET].$$

PROOF. Let the n indices in ISET be $\alpha_1, \alpha_2, \ldots, \alpha_n$. By definition, $\text{COST}_q[\text{ISET}]$ is the minimum of the following costs:

$$c_0 = C_q(\rho) + \sum_{i=1}^{n} U_q(\alpha_i, \rho)$$

$$c_1 = C_q(\alpha_1) + \sum_{i=1}^{n} U_q(\alpha_i, \alpha_1)$$

$$\vdots$$

$$c_n = C_q(\alpha_n) + \sum_{i=1}^{n} U_q(\alpha_i, \alpha_n).$$

$\text{COST}'_q[\text{ISET}]$ is the minimum of

$$c'_0 = \text{COST}_q[\{\rho\}] + \sum_{\beta \in \text{ISET}} U_q(\beta, \rho) = c_0,$$

$$c'_1 = \text{COST}_q[\{\alpha_i\}] + \sum_{\beta \in \text{ISET}-\{\alpha_1\}} U_q(\beta, AP_q(\{\alpha_i\})) = \min(c_0, c_1),$$

$$c'_2 = \text{COST}_q[\{\alpha_2\}] + \sum_{\beta \in \text{ISET}-\{\alpha_2\}} U_q(\beta, AP_q(\{\alpha_2\})) = \min(c_0, c_2),$$

$$\vdots$$

$$c'_n = \text{COST}_q[\{\alpha_n\}] + \sum_{\beta \in \text{ISET}-\{\alpha_n\}} U_q(\beta, AP_q(\{\alpha_n\})) = \min(c_0, c_n).$$

Hence, $\text{COST}'_q[\text{ISET}] = \min(c_0, c_1, c_2, \ldots, c_n)$, demonstrating the proposition. □

As we mentioned earlier in this section, for an index β the maintenance cost $U_q(\beta, \alpha)$ depends on whether the access to β is an ordered or unordered scan and is otherwise independent of α's column. (This is also true when α is ρ.) Thus, there are only two costs to be computed for β. Let $U'_q(\beta)$ be the cost of updating β if α determines an ordered scan of β, and let $U''_q(\beta)$ be the cost of updating β if α determines an unordered scan of β. $U_q(\beta, \alpha)$ is either $U'_q(\beta)$ or $U''_q(\beta)$.

Performing EXPLAIN COST for atomic configurations, DBDSGN can collect the maintenance cost of a given index for both ordered and unordered scans. For example, assume q is an UPDATE statement, and we want to evaluate the maintenance cost of an index β. The atomic configurations with β that are of interest for q depend on whether β is clustered or nonclustered. Performing EXPLAIN COST for q with β clustered, we obtain from the optimizer a cost $C_q(\beta)$, for the access and tuple maintenance, and a cost $U_q(\beta, \rho)$, for updating the index. The only possible access path for the optimizer is sequential scan ρ, so $U_q(\beta, \rho) = U'_q(\beta)$ is the cost of maintaining the index β following an ordered scan. Similary the configuration with β nonclustered gives us the cost $U''_q(\beta)$ of

the unordered scan.[3] Similar considerations can be applied for DELETE and INSERT statements. The reader is referred to [33] for details on the cost formulas.

3.4 Columns Plausible for Indexing

Performing EXPLAIN COST only for atomic configurations significantly reduces the number of cost inquiries to the optimizer. This section describes a technique for reducing the number of cost inquiries even further.

The number of index candidates on a table equals twice the number of columns in the table (because indices may be clustered or nonclustered). However, not all columns are plausible candidates for indexing. Columns that appear in a statement in ways that support use of indices are called *plausible* columns (for that statement). Other columns are called *implausible.* The considerations that determine the set of plausible columns for each statement are optimizer dependent. The critical requirement is that, for the statement, implausible columns must have (essentially) the same costs for indices, no matter what other indices exist. For System R the considerations include the following:

(1) A column is plausible if there is a predicate on it and the system can use an index to process that predicate. This happens when the predicate is **AND**ed to the rest of the **WHERE** clause, and it is usable as a search argument to retrieve tuples through an index scan. That is, the predicate has the form **column op** X, where **op** is a comparison or range operator ($>$, $\geq$, $=$, $\leq$, $<$, BETWEEN, IN), and X is a constant, a program variable, or a column in a different table. For example, for the table

PARTS: (PARTNO, DESCRIP, SUPPNO, QONORD, QONHAND, COLOR, WEIGHT, ...)

in the statement

(S2)

```
SELECT   PARTNO, DESCRIP
FROM     PARTS
WHERE    SUPPNO = 274
AND      (COLOR = 'RED' OR WEIGHT > 37)
AND      QONORD = QONHAND + 50,
```

SUPPNO is plausible for statement (S2), but COLOR and WEIGHT are implausible. QONORD is also implausible, because it is compared with the result of an expression.

(2) A column that is not plausible because of selection predicates may still be a plausible candidate for indexing for other reasons. For example, there may be a GROUP BY or ORDER BY clause on the column.[4]

---

[3] We assume here that the cost of updating an index following an unordered scan is always the same, no matter what access path is chosen. This is not always true (see [33] for details), but we think it a reasonable approximation.

[4] The optimizer could even decide that a column that does not appear in the statement is plausible. Moreover, an implausible index might be a better access path than sequential scan in certain cases. Since all indices on implausible columns have almost identical costs, a single implausible representative can be added to the plausible set.

(3) If a table is not mentioned in a statement, all its columns are implausible for that statement.

Using EXPLAIN REFERENCE, DBDSGN identifies the set of plausible columns for each statement. This avoids putting optimizer-dependent information on plausibility into DBDSGN. More generally, the design tool identifies the set of *plausible access paths* for each statement, which includes the clustered and nonclustered indices on the plausible columns,[5] as well as sequential scan. Plausible columns may be unusable for a particular statement because of system constraints. For example, a column that is changed by an UPDATE statement may not be usable even if it appears in an index-processable predicate (see footnote[2]). In a system that supported links and hashing, some links and hashed access paths would also be plausible for a given statement.

We believe the database system (rather than the designer) should determine plausibility. The optimizer is the best judge of its own capabilities. Moreover, it is simpler for designers to let the system automatically determine plausibility rather than to specify plausible columns themselves. Since EXPLAIN REFERENCE is only performed once per statement, determining the columns plausible for indexing is inexpensive.

Limiting access-path choices to plausible access paths greatly reduces the number of cost evaluations requested from the optimizer. A configuration is *plausible* for a statement if all indices in it are plausible for that statement. The following criterion is used to limit the number of times EXPLAIN COST is performed:

(C1) Costs are obtained for each statement only for plausible atomic configurations for that statement.

The validity of this criterion is a consequence of Propositions 1 and 2 of Sections 3.2 and 3.3.

The value of plausibility in reducing the complexity of the index-selection problem is illustrated by the following example: Consider the table **PARTS** mentioned above and the table **ORDERS** of Section 2.2, where each table has 10 columns. Without plausibility a design tool would consider 5,120 (10×2^9) configurations for each table with one index clustered and the others nonclustered, and 1,024 (2^{10}) configurations with all indices nonclustered, for a total of 6,144 configurations. For the two tables together, there are a total of 37,748,736 ($6{,}144^2$) configurations. Plausibility allows us to drastically reduce the number of configurations. Consider the following statement:

(S3)

```
SELECT   P.PARTNO, P.QONORD
FROM     PARTS P, ORDERS O
WHERE    O.PARTNO = P.PARTNO
AND      O.SUPPNO = 15
AND      P.WEIGHT > 200
AND      P.QONHAND BETWEEN 100 AND 150.
```

[5] In an early version of DBDSGN [32], there was no distinction between plausible and implausible columns, and all columns of the table were considered index candidates. This meant less dependence on the optimizer's special properties, but it was also much less efficient.

Of the 20 columns in **PARTS** AND **ORDERS**, only 5 are plausible for statement (S3): PARTNO, WEIGHT, and QONHAND for **PARTS**, and PARTNO and SUPPNO for **ORDERS**. Hence, there are 160 plausible configurations on the two tables, and only 35 of them are atomic plausible configurations. Suppose that another statement in the same workload is

(S4)

```
SELECT   *
FROM     PARTS P, ORDERS O
WHERE    O.PARTNO = P.PARTNO
AND      O.DATE = 840701
AND      P.WEIGHT < 300.
```

All the columns in **PARTS** and **ORDERS** appear in the select list, but only four are plausible. For statement (S4) there are 64 plausible configurations, of which 25 are atomic plausible configurations. Fifteen of those are also atomic plausible for (S3). The total number of different atomic plausible configurations for (S3) and (S4) is 45. In practical workloads many columns in the database are not referenced, and some columns are only referenced in the **SELECT** lists and never in the **WHERE** clauses. The plausible configurations for joins often intersect considerably; it is particularly common for several statements to have the same join columns (because of hierarchical and network relationships that exist in the data tables). Furthermore, as we previously indicated, not all columns referenced in the **WHERE** clauses are plausible. Hence, performing index selection on the basis of the plausible configurations can be practical.

3.5 Catalog Statistics

The cost of executing a statement in the database's current configuration can be obtained using EXPLAIN COST. The optimizer uses statistics in the database catalogs to determine the costs of execution plans and chooses the plan with the lowest cost estimate. Thus, the optimizer will return the cost estimate for a statement in a configuration if the system catalogs describe that configuration. Among the statistics used by the optimizer in making cost estimates are

—for each table, table cardinality (number of tuples in the table) and the number of pages occupied by the table;
—for each column, the average field length, the column cardinality (number of distinct values for the column), and the maximum and minimum values in the column; and
—for each index, the number of leaves and levels.

The obvious way to get the cost of a statement in a configuration is to create that configuration (thereby causing the right statistics to be in the catalog) and to perform that statement. Executing many statements on large tables is typically unacceptable. Creating configurations and performing EXPLAIN COST is better since the optimizer's cost model is used and the statements are not actually executed. But it is still very expensive to create combinations of indices on large tables, and such activity would also interfere with normal operations on tables. DBDSGN uses a different approach that does not have these disadvantages and moreover allows design even when the tables are not yet populated with tuples.

Instead of building configurations, DBDSGN simulates them by changing entries in the database catalogs. To do this it must have statistics describing the configurations, and these statistics can come from several sources. For tables that are already populated, DBDSGN can obtain statistics for tables and columns using the UPDATE ALL STATISTICS statement [3, 20], and based on these can estimate index statistics. If indices already exist, DBDSGN can use their statistics. For tables that have not been loaded yet (or whose contents are expected to change drastically), the designer can supply statistics in a file. Thus, DBDSGN can do design even when there are no tuples in the database (as long as the tables exist).

DBDSGN updates the system catalogs to simulate atomic configurations that are plausible for some statement. If DBDSGN updated the catalog descriptions for the actual tables, applications using these tables would be delayed and find incorrect information in the catalogs. Instead, alterations are made on catalog entries for artificial tables that are created by DBDSGN, which we call *skeleton replicas*. DBDSGN creates these replicas exactly as the actual tables were created. It then updates the catalog entries for the tables and their columns so that their statistics are those of the actual tables (or are the statistics provided by the designer). The skeleton replicas have the same statistics as the actual tables, but contain no tuples and are used only by DBDSGN. Simulating an atomic configuration involves creating the indices in that configuration (on the replicas) and putting the right index statistics into the catalog. (Indices on empty tables are created quickly.) The skeleton replicas are used only for EXPLAIN COST; they are never accessed.

3.6 Computation of Atomic Costs

In order to generate solutions to the index-selection problem, we need to compute the costs of the statements for plausible atomic configurations. One significant component of DBDSGN's execution time is the catalog-update activity required to simulate different index configurations. In System R the system catalogs are stored in the database, so every catalog update affects the database.[6]

We say that one configuration *covers* another for statement q if they have the same indices for all tables referenced in q. A set of configurations covers another for statement q if each configuration in the second set is covered for q by a configuration in the first set. A set of configurations is *minimal* for a workload if it contains no configuration that is covered by the other configurations for every statement in the workload. Since the cost of a statement is independent of indices on tables not referenced in the statement, to obtain all plausible atomic costs it suffices to simulate a (minimal) set of atomic configurations that covers the set of plausible atomic configurations for the workload.

Consider statements on a single table. Since the same index may be plausible for more than one statement, the number of system catalog updates necessary to simulate the configurations equals the total number of different indices that are

[6] The cost of catalog updates would be insignificant if catalog data could be stored outside the database (e.g., in files or program variables). The database system would have to be changed to use this (spurious) cache, rather than the actual catalog data. This would also eliminate the need for the skeleton-replica tables described in the previous section.

plausible for at least one statement. (Sequential scan must be counted once for each table.) For single-table statements, catalog updates could be done efficiently on a table-by-table basis.

For a workload that includes joins, the number of catalog updates may be very high, since the number of atomic configurations to be simulated grow exponentially with the number of tables joined. We want to reduce the number of catalog updates by never simulating a configuration more than once, by simulating a minimal set of configurations for the workload, and by simulating configurations in a sequence that reduces the number of catalog updates. In this section we describe a simple procedure to enumerate (a cover for) the plausible atomic configurations so that DBDSGN can obtain the plausible atomic costs for all statements in the workload.

For a statement q involving t_q tables, let NA_{iq} be the number of plausible access paths to the ith table of the statement. The number of different atomic configurations to be simulated is

$$\prod_{i=1}^{t_q} \mathrm{NA}_{iq}.$$

If atomic configurations are enumerated using Gray coding (any other enumeration scheme generating each configuration once would be acceptable), with table q as the highest order (least frequently changing) column and table 1 as the lowest order (most frequently changing) column, then the number of catalog updates is

$$\psi_q = \sum_{j=1}^{t_q} \prod_{i=1}^{j} \mathrm{NA}_{iq}.$$

ψ_q is minimized by permuting the tables so that the NA_{iq} values are monotonically increasing. Different table permutations may be used for different statements. $\Psi = \sum_q \psi_q$ catalog updates suffice to compute costs for all statements. In many cases the plausible configurations for joins intersect considerably, so performing the cost computations independently for each join risks creating identical configurations more than once. To avoid this (and hence to reduce the number of catalog updates), whenever we simulate an atomic configuration we compute the cost of each statement for which that configuration is plausible. (More generally, we compute the cost for each statement such that the simulated configuration covers a plausible configuration.) Ordering the statements so that the ones with the largest number of tables are processed first also may reduce the number of configurations generated (since a join involving many tables may enumerate configurations needed by simpler statements).

Join cost computation rules

(1) The list of join statements is ordered in decreasing order of the number of tables referenced.

(2) For each join q, all the plausible atomic configurations are enumerated using Gray coding with the tables permuted so that the NA_{iq} values are increasing. A configuration is simulated only if the cost of q for that configuration has not been computed yet.

(3) For each simulated configuration, EXPLAIN COST is performed for every join (after the current join in the join list) such that the simulated configuration covers an atomic configuration that is plausible for that join.

3.7 Required Indices

There are two types of index requirements that may be specified for DBDSGN. The designer may want the indices for some tables to be exactly as they are in the database. The workload W described in Section 3.1 identifies a set of tables T. The designer can partition T into two sets T_{exist} and T_{design}. The indices that already exist in the database for tables in T_{exist} are not be be altered. For example, the designer may not be authorized to redesign those tables (e.g., system catalogs), or the indices for T_{exist} may have already been selected for specific applications. The designer specifies the tables in T_{design}, and DBDSGN suggests index designs for those tables. All other tables are in T_{exist}, and existing indices are used for them. If the indices to the tables in T_{exist} change, the solutions recommended by DBDSGN for the tables in T_{design} might also change.

In addition, the designer may require certain index choices for tables in T_{design}. For example, the clustered index for some table may already have been chosen. Indices may also be required to enforce a constraint; in System R the uniqueness of keys is enforced by creating a "unique" index. DBDSGN allows indices to be required or excluded from all solutions. A variation of this permits fast evaluation of an individual index configuration.

4. INDEX ELIMINATION

In the previous section, we described plausible access paths. Plausibility is based on the appearances of columns in statements, not on the costs of access paths. Plausibility is a valid criterion for restricting the costs evaluated; if the solution-generation procedure described in Section 5 is followed so that the entire space is searched, all configurations are considered, and the optimal index configurations are found (if we assume the costs furnished by the optimizer are the actual execution costs). Analyzing all possible plausible atomic configurations, however, may be impractical when a workload includes joins on many tables where a large number of columns are plausible. Deciding whether or not to do index elimination involves trading improved execution time of the design tool versus finding better (i.e., nearer optimal according to optimizer cost estimates) solutions. As we discussed earlier, finding the (apparently) optimal configuration is not required, since statistics provide an incomplete description of the database, and the optimizer's cost formulas furnish an approximation to actual cost.

In this section we describe heuristic criteria for deciding which plausible indices are likely to be chosen as access paths by the optimizer when other access paths exist in the database. These criteria, based on access cost, can reduce the set of configurations. First we describe index-elimination criteria for statements on one table, and then we consider the multitable case.

4.1 Index Elimination on a Single Table

In this section we assume all statements are on a single table. Hence, all atomic configurations have (no more than) one index. Index elimination is not usually necessary in this case since the number of plausible atomic configurations is

small. However, we begin with a single-table example to help motivate the technique used for index elimination in the multitable case.

DBDSGN simulates configurations column by column (clustered and nonclustered indices), and EXPLAIN COST is performed for all statements for which the column is plausible. Index elimination is carried out by comparing every index choice with every other index choice, as well as with sequential scan. A set of elimination criteria is *valid* if the criteria never eliminate an index that appears in the optimal solution. The elimination criteria we describe here are valid for single-table queries. When the workload also contains maintenance statements and joins, the criteria may not be valid. The problems associated with the maintenance and join costs are discussed at the end of this section and in the next section.

Let $C_i(j)$ be the cost of query q_i with index j. If $C_i(k) < C_i(j)$, then, if both indices exist in the design, the optimizer will prefer k to j. We must also consider the memory cost m_j for each index j, which we define to be the number of pages in the index (multiplied by a storage weight σ supplied by the designer to trade off page costs versus execution costs in computing total cost of a configuration). If $C_i(k) \leq C_i(j)$ for all q_i, then the optimizer will *never* take j if k is in the design. If this is true, k is a better index choice than j, and we can eliminate j from consideration (unless the storage m_k required for index k is more than m_j).

These considerations lead to the following definition:

(DEF1) Given two indices j and k, if $m_k \leq m_j$ and, for all $q_i \in W$, $C_i(k) \leq C_i(j)$, then j is *dominated* (as an index choice) by k. If equality holds for all q_i and m, then k and j are *equivalent*.

When indices are equivalent, all but one can validly be eliminated. The configuration with no indices is represented by a vector $R(\rho)$ of costs $C_i(\rho)$ that corresponds to sequential scan. The optimizer never returns a cost $C_i(j) > C_i(\rho)$, so any index equivalent to sequential scan can validly be eliminated.

Let CLUST be the set of plausible clustered indices over all the q_i, and NONCLUST be the set of plausible nonclustered indices over all the q_i. The following four criteria based on DEF1 can validly be used to eliminate indices from CLUST and NONCLUST. (After an index is eliminated, it cannot eliminate any indices.)

(E1) If $k, j \in$ CLUST and k dominates j, then eliminate j from CLUST.

(E2) If $j \in$ CLUST is equivalent to $k \in$ NONCLUST, and k and j are indices on the same column, then eliminate j from CLUST.

(E3) If $j \in$ NONCLUST is equivalent to ρ, then eliminate j from NONCLUST.

(E4) If $k, j \in$ NONCLUST, and k dominates j, then eliminate j from NONCLUST.

Each of these criteria is valid because the optimizer uses only one access path per table (see Section 3.2) and there is only one table. Criteria (E1) and (E2) should be applied in that order before (E3) and (E4), since otherwise we may eliminate an index before it has the chance to eliminate others. Criteria (E3) and (E4) may be applied in either order. Criteria (E1) and (E4) eliminate dominated indices and keep only one among equivalent indices. Criterion (E2) eliminates

any clustered index that is equivalent to the nonclustered index on the same column, because there is no advantage in keeping the tuples ordered on that column. In (E3), nonclustered indices are compared with the configuration with no indices and eliminated if equivalent. If the corresponding clustered indices are equivalent to ρ, they are eliminated by (E2) (since no nonclustered index can be better than a clustered index on the same column). After the application of the above criteria, CLUST and NONCLUST contain only the indices that are comparatively useful for at least one query (or have small memory cost).

Table I shows costs for a table T_1 with 6 plausible columns and 4 queries. (Normally, different columns might be plausible for different queries.) The single index atomic costs are arranged in a matrix with 4 rows (queries) and 13 columns (6 for the costs of clustered indices, 6 for the nonclustered indices, and 1 for sequential scan). Ignoring memory costs for simplicity, the results of index elimination for that cost matrix are as follows:

Results of elimination

—Criterion (E1): 1c eliminates 6c; 2c eliminates 4c.
—Criterion (E2): 1n eliminates 1c.
—Criterion (E3): ρ eliminates 4n and 6n.
—Criterion (E4): None.

Further elimination criteria may be applied to CLUST and NONCLUST if they still contain many elements. Other indices can be eliminated if they are "almost" dominated by some index. If strict domination is used, indices may survive the elimination process because they are slightly better than others for a few queries, even though they are much worse in most queries. Heuristic elimination criteria may be preferable to strict domination. Let the maximum advantage of k over j for all q_i be

$$\mathrm{MA}_{k,j} = \max_{q_i} \{w_i[\mathrm{C}_i(j) - \mathrm{C}_i(k)]\},$$

and let ε be an elimination coefficient specified between 0 and 1. Heuristic elimination criteria can be based on the following domination definition:

(DEF2) An index k *ε-dominates* an index j if

$$\mathrm{MA}_{j,k} \leq \varepsilon \mathrm{MA}_{k,j},$$

and for storage

$$\sigma(m_k - m_j) \leq \varepsilon \mathrm{MA}_{k,j}$$

where σ is the storage weight supplied by the designer.

Index k ε-dominates index j if the maximum advantage of j over k (over both estimated execution cost and storage cost) is less than or equal to a fraction of the maximum advantage of k over j. Zero-domination is identical to domination ((DEF1)), so index elimination is the same as index elimination with $\varepsilon = 0$. ε-domination might be defined in other ways (e.g., by comparing total advantages or by comparing maximum advantage to total advantage), but we prefer

Table I. Cost Matrix for Index Elimination

	Clustered indices						Nonclustered indices						No index
	1c	2c	3c	4c	5c	6c	1n	2n	3n	4n	5n	6n	ρ
q_1	100	100	50	100	90	100	100	100	50	100	100	100	100
q_2	150	10	50	35	40	150	150	20	50	150	40	150	150
q_3	5	10	10	10	10	5	5	10	10	10	10	10	10
q_4	100	60	100	200	100	200	100	140	200	200	130	200	200

comparison of maximum advantages, since this comparison means that eliminated indices are comparatively unimportant.

Domination increases monotonically as ε increases; that is, if j ε_1-dominates k, then j ε_2-dominates k for $\varepsilon_1 < \varepsilon_2$. Assuming storage costs are equal, for any pair of indices j and k there is a smallest ε between 0 and 1 such that one index ε-dominates the other (based on the ratio of their maximum advantages; if both maximum advantages are 0, the indices are equivalent). ε-domination is not transitive, so the order in which elimination is applied may change the set of eliminated indices.

If the clustered index were chosen on a table, further index elimination could be done based on that choice. This motivates an additional elimination criterion. Fix a particular table. Let G_0 contain all the surviving indices on that table in NONCLUST. For each clustered index k on that table that survived index elimination, let G_k contain clustered index k as well as the nonclustered survivors. Elimination is performed within each group G_k by applying a domination criterion using the clustered index within the group:

(E5) For each group G_k, if k in CLUST ε-dominates an index j in NONCLUST or if k and j are indices on the same column, then eliminate j from G_k (but not from any other group).

Criterion (E5) eliminates the nonclustered index on the clustered column (which is always dominated by the corresponding clustered index) and eliminates other indices that are dominated by the clustered index. Elimination using (E5) can be done only group by group and not globally on NONCLUST, since nonclustered indices dominated by some clustered choices may be useful for other clustered choices. The results of applying (E5) to the G_k are called the *basic groups* for the table.

We previously showed index elimination for Table I, which is the same as index elimination with $\varepsilon = 0$. Index elimination using the ε-domination definition for $\varepsilon = \frac{1}{3}$ yields the following results:

Results of elimination with $\varepsilon = \frac{1}{3}$

—Criterion (E1): 1c eliminates 6c; 2c eliminates 4c; 3c eliminates 5c.

—Criterion (E2): 1n eliminates 1c.

—Criterion (E3): ρ eliminates 4n and 6n.

—Criterion (E4): 2n eliminates 1n.

—Criterion (E5): In the basic group for 2c, 2c eliminates 2n and 5n.
—Criterion (E5): In the basic group for 3c, 3c eliminates 3n and 5n.

Basic groups after elimination with $\varepsilon = \frac{1}{3}$

—(2c, 3n),
—(3c, 2n), and
—(2n, 3n, 5n).

For maintenance statements as well as queries, costs are compared for atomic configurations only. Since the rest of the solution is not determined, DBDSGN cannot include the cost of maintaining other indices in its cost comparisons during the index-elimination phase. Hence, elimination criteria may not be valid heuristics when there are maintenance statements in the workload.

4.2 Index Elimination for Multitable Statements

Most approaches to index selection are restricted to single-table statements. Approximate solutions are obtained by performing the index selection separately table by table. This approach does not work for a system like System R, whose join methods do not have the separability property [38, 39]. System R has two methods for performing joins: "nested loop" and "merge scan" [2, 35]. The optimizer chooses the sequence in which tables are joined, the join methods, and the access path used for each table. For an n-way join, it can use the two methods in any appropriate sequence of 2-way joins. In each join the choice of table order, the join method, and the access paths on tables cannot be done independently.

From our experience with System R and DBDSGN, we concluded that the single-table criteria of the previous section are also good (although not necessarily valid) in the multitable case, when the following (optimizer-dependent) restrictions are obeyed:

(J1) Indices can only eliminate other indices on the same table.
(J2) Clustered indices on join columns can never be eliminated.
(J3) Indices on join columns can never eliminate any other indices.

Restriction (J1) arises because indices are single-table access paths.

Restriction (J2) arises because merge scan is often a very efficient join method when both join columns are clustered. This can seldom be detected from single index atomic costs. Consider, for example, the two tables **ORDERS** and **PARTS** of Sections 2.2 and 3.4, and the following SQL statement:

(S5)

```
SELECT   O.SUPPNO, P.QONORD
FROM     PARTS P, ORDERS O
WHERE    O.PARTNO = P.PARTNO
AND      O.SUPPNO = 15
AND      P.QONHAND BETWEEN 100 AND 150.
```

Assume that a decision has been made to cluster the **PARTS** table on DESCRIP and that a nonclustered index on PARTNO exists for **PARTS**. Given this, the best clustered index for **ORDERS** is probably on SUPPNO. This allows quick retrieval of the tuples from **ORDERS** that have SUPPNO = 15. For each of

these tuples, the corresponding tuples in **PARTS** (having the same PARTNO) can be found using the index on PARTNO on **PARTS** (nested-loop method). The best choice of clustered index for **ORDERS** would be entirely different had the choice for clustered index on **PARTS** been PARTNO. In this case, clustering **ORDERS** on PARTNO would enable even faster processing of statement (S5). The join predicate would be resolved by performing one pass over each table via the clustered index (merge-scan method). This shows that the selection of a clustered index cannot be done independently for each table.

Restriction (J3) arises because some very good solutions would be ignored if indices on join columns were allowed to eliminate indices on nonjoin columns. There are two negative results that could occur without (J3). Suppose the workload contains just statement (S5). Apply index elimination to nonclustered indices on columns SUPPNO and PARTNO of the table **ORDERS**. For simplicity, we only consider the costs of the nested-loop join method. In the nested-loop method for two tables, one table is the outer table, and the other is the inner table. For each qualifying outer-table tuple (satisfying predicates on that table), matching inner-table tuples are found (satisfying join predicates and predicates on the inner table). Let E_X be the expected number of tuples that satisfy predicates on the outer table X (which will also be the number of times the inner table is scanned), let ρ_Y be the cost of the sequential scan on the table Y, let $C(\alpha_j)$ be the cost of accessing the outer table using the index on column j, and let $C'(\alpha_j)$ be the access cost to retrieve tuples matching an outer tuple using the index on column j of the inner table. The optimizer cost estimates are as follows [35]:

—For the index on column O.SUPPNO, the minimum of A1 and A2 are

$$A1 = \rho_{\mathbf{PARTS}} + E_{\mathbf{PARTS}}C'(\text{O.SUPPNO}) \qquad \text{(using } \mathbf{PARTS} \text{ as outer)}$$

and

$$A2 = C(\text{O.SUPPNO}) + E_{\mathbf{ORDERS}}\rho_{\mathbf{PARTS}} \qquad \text{(using } \mathbf{ORDERS} \text{ as outer).}$$

—For the index on column O.PARTNO, the minimum of A3 and A4 are

$$A3 = \rho_{\mathbf{PARTS}} + E_{\mathbf{PARTS}}C'(\text{O.PARTNO}) \qquad \text{(using } \mathbf{PARTS} \text{ as outer)}$$

and

$$A4 = \rho_{\mathbf{ORDERS}} + E_{\mathbf{ORDERS}}\rho_{\mathbf{PARTS}} \qquad \text{(using } \mathbf{ORDERS} \text{ as outer).}$$

Suppose that each index has access cost less than the sequential scan cost and that A3 is less than both A1 and A2. Index elimination would eliminate the index on O.SUPPNO. But, if we put an index on the QONHAND column on **PARTS**, the cost of accessing the **PARTS** table might significantly be reduced. Define B1 and B3 by substituting C(P.QONHAND) for $\rho_{\mathbf{PARTS}}$ in the equations for A1 and A3. Similarly define B2 and B4 by substituting C'(P.QONHAND) for $\rho_{\mathbf{PARTS}}$ in the equations for A2 and A4. The cost reduction from A2 − B2 is $E_{\mathbf{ORDERS}}$ times greater than the cost reduction from A3 − B3. If $E_{\mathbf{ORDERS}}$ is large, the value of A2 will now be much less than the value of A3. Thus, the decision to eliminate the index on O.SUPPNO was poor.

A second outcome that produces even worse results could occur if we ignored restriction (J3). Again we use statement (S5) as an example. The nonclustered index on O.PARTNO could eliminate all other nonclustered indices on **ORDERS** (because cost A3 with **ORDERS** as the inner relation for a nested-loop join is small); similarly P.PARTNO could eliminate all other nonclustered indices on **PARTS** (because executing a nested-loop join with **PARTS** as the inner relation might be cheaper than the alternatives). These join-column indices would not both be used for a nested-loop execution of (S5), since one of them would be on the outer table and there is no nonjoin predicate on either column. (An index can be used to scan all the tuples in a table, but this is typically not profitable.) Thus, the optimizer is forced to choose sequential scan on one of the two tables, and consequently one of the indices will be useless for (S5).

DBDSGN does permit elimination of nonclustered indices on join columns by other indices. If A3 is more than A1 (or A2), then if we put an index on a column on **PARTS** then A3 remains more than A1 (or A2). This is true of A4 as well as A3, and nonclustered indices are poor choices for merge-scan joins. Hence, the nonclustered index on join column O.PARTNO can safely be eliminated when it is dominated by another nonclustered index.

Our discussion of restrictions (J2) and (J3) shows that solutions for the single-table case do not extend to the multitable case in a trivial way (i.e., by combining all the individual solutions for each table) for System R. We have done a series of experiments that show that index elimination is a good heuristic when (J1), (J2), and (J3) are followed.

After DBDSGN does index elimination for a specified value of ε, it indicates how many plausible atomic costs there are for the indices that survived. The designer can then choose to supply a different value of ε.

5. SOLUTION GENERATION

The last step in the design process is a controlled search of the space of subsets of the survivor indices in CLUST and NONCLUST to find good solutions to the index-selection problem. Solutions, which are index configurations, are annotated with the access costs, maintenance costs, and access paths used for each statement in the workload and the total cost. The total cost can also depend on the total storage and the storage weight σ if a designer wants to balance execution time versus the cost of storage. Storage cost is ignored in this section, but is simple to include.

The indices in CLUST and NONCLUST are stored in a list (called the *survivor list*) where the clustered indices precede the nonclustered indices. (If the designer chooses not to do index elimination, CLUST is the set of all plausible clustered indices, and NONCLUST is the set of all plausible nonclustered indices.) The search is done through a tree expansion that enumerates the configurations so that no configuration appears more than once in the tree. The ordering of the survivor list matters; we describe survivor-list ordering rules later in this section.

Before the start of tree expansion, DBDSGN asks the designer whether there is an *index storage limit*; if there is, the designer has to supply the maximum number of pages available for an index configuration in the database.

Table	T_1	T_2
Survivors	2c 3c 2n 3n 5n	7c 9n
Basic groups	2c 3n 3c 2n 5n 2n 3n 5n	7c 9n 9n
Survivor list	2c 7c 3c 9n 2n 5n 3n	

Fig. 2. An example of a survivor list and basic groups.

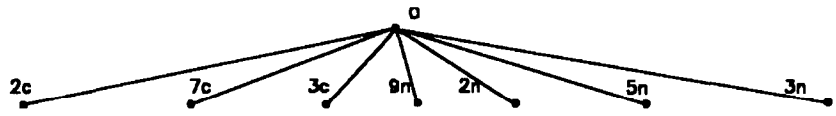

Fig. 3. First expansion.

The root of the tree represents the solution with no indices on any table. A node's children always have one additional index, so the nodes at level k have exactly k indices. Adding a node's children to the tree is referred to as *expanding the node.* The tree's growth proceeds according to the following rules:

Tree expansion rules

(1) The root is expanded with one child for each index on the survivor list. (These nodes represent solutions having only one index on the database.)
(2) For each node, expansion is done with indices that appear later in the survivor list than any index already in the node.
(3) A node can be expanded only with indices that belong to the basic groups of the clustered indices already present in the solution represented by that node.
(4) If a node has no clustered index for a table, any clustered index on that table can be added, but only nonclustered indices in the all-nonclustered basic solution for that table can be added to that node. (Recall that clustered indices precede nonclustered indices in the survivor list.)
(5) Any node that exceeds the index storage limit specified by the designer is pruned.

To explain how the tree grows, we use an example. Suppose design is for the table T_1 of Section 4, and for a table T_2, whose survivor list and basic groups are shown in Figure 2.

Figure 3 shows the first expansion of the tree with solutions having only one index on the database. In this figure the root has no indices. However, when partial designs for some tables have been specified as described in Section 3.7

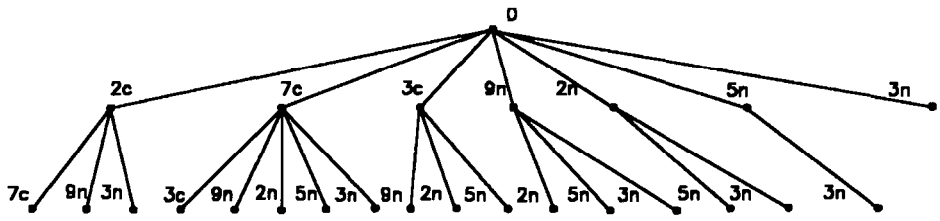

Fig. 4. Second expansion.

(requiring some indices on those tables, but allowing additional indices), the root is the set of required indices.

In Figure 4 the tree is expanded from the first level to the second, and the application of the procedure is shown. For example, the solution represented by 7c is not expanded with 2c; the solution (2c, 7c) is already present and equivalent to (7c, 2c). No expansion takes place for the solution with 3n, which is the last index in the survivor list, but all possible combinations of 3n with other indices appear elsewhere in the tree. Furthermore, 2c is not expanded with 3c or 5n because they do not belong to the same group, and 3c is not expanded with 3n.

For each node a total cost is computed during the expansion. This total cost is the weighted sum of all the costs of the statements (access and maintenance costs) when the database has the set of indices represented by that node. Let ISET be the set of indices in a given node. The total cost of the solution represented by that node is

$$\text{TOTALCOST[ISET]} = \sum_q w_q \text{COST}_q\text{[ISET]},$$

where COST_q[ISET] is the cost of statement q as defined in Section 3.2 for queries and in Section 3.3 for maintenance statements.

The solution in a node can be worse than its parent solution. Assume we start from a node having ISET as a solution and we add index α. The access advantage of a solution ISET′ = ISET ∪ $\{\alpha\}$ is

$$\begin{aligned} A &= \text{TOTALCOST[ISET]} - \text{TOTALCOST[ISET']} \\ &= \sum_q w_q \text{COST}_q\text{[ISET]} - \sum_q w_q \text{COST}_q\text{[ISET']}. \end{aligned}$$

COST_q[ISET′] can be efficiently computed using atomic costs as the minimum of

(1) COST_q[ISET], and
(2) the minimum value of COST_q[ASET], taken over atomic subsets of ISET that contain α.

If α is used for q in configuration ISET′, then the access paths for q correspond to those in some ASET containing α. A cannot be negative. If A is 0, no atomic cost including α is better than those without α, so α is not used as an access path in any statement for configuration ISET′. If A is positive, the disadvantage in

maintenance cost must be considered:

$$D = \sum_q w_q \left[\sum_{\beta \in \text{ISET}'} U_q(\beta, AP_q(\text{ISET}')) - \sum_{\beta \in \text{ISET}} U_q(\beta, AP_q(\text{ISET})) \right].$$

The difference inside the square brackets is not simply $\sum_q U_q(\alpha, AP_q(\text{ISET}'))$, the additional maintenance cost for index α, because the maintenance costs of other indices may have changed based on the change of some access-path choices. Thus, to correctly evaluate maintenance costs, the design tool has to keep track of the actual access paths for each statement. ISET′ has a better total cost than ISET only if A is greater than D.

Furthermore, knowing the actual access paths allows us to detect *wasteful* solutions. These are solutions that contain one or more indices that are never taken as access paths. In the index-elimination phase, we ensure that no index is dominated by any other single index; in solution generation we want to ensure that no index is *wasted* because it is overpowered (in a solution) by a *set* of other indices. Wasteful solutions can arise in two ways:

(1) The most recently added index α is not used in any statement. That is, ISET overpowers α.

(2) When α is added to ISET, some index β in ISET is no longer in an access path for any statement. That is, ISET $\cup \{\alpha\} - \{\beta\}$ overpowers β.

If a wasted index is plausible for some join, future additions of indices on different tables may make it useful for that join. If that index is not plausible for a join (or all the plausible indices for the other tables in the same join are already in the solution), then that index will always be wasted, no matter how the solution is expanded. In that case, the node can be pruned from the tree.

At the end of expansion, the tool displays the S solutions having the smallest cost, where S is a parameter specified by the designer. Wasteful solutions are never displayed (so they may be dropped as soon as they have been expanded). The full expansion of the tree is shown in Figure 5. (No index storage limit is considered in the expansion.) The circles indicate the best solutions ($S = 3$). Some nodes are pruned because the tool detects wasteful solutions. When maintenance statements are present in the workload, the best solutions are not necessarily at the leaves of the tree.

Exploring all the solutions in the tree may be extremely time consuming when there are many tables in the database with a large number of surviving indices. Thus, we allow a controlled partial expansion of the tree, using breadth-first search with heuristic pruning. The search is conducted by expanding all solutions with no more than L indices and keeping only the best N, where N and L are parameters specified by the designer, with $N \geq S$. Each of these N solutions is then expanded by at most L additional indices (assuming they were at the frontier of the previous expansion). This process continues until no further expansion is possible.

Because of this pruning rule, each different ordering of the survivor list determines a different tree (even though the set of solutions examined during the first L index expansion is always the same). This is because the survivor list

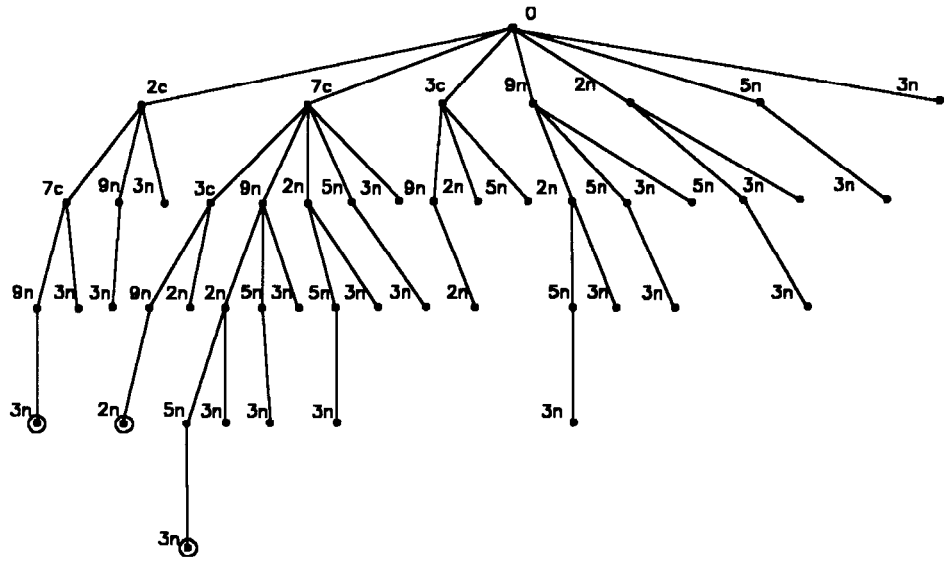

Fig. 5. Full expansion.

imposes a discipline on a node's descendants, denying them certain indices. We want "influential" indices to be early in the survivor list, so that other indices can still be added, despite pruning. (E.g., if the first index in the list was good only for some low-frequency statement, it might be pruned completely after the first L index expansion.) The order of the survivor list is determined by two rules:

Survivor list ordering rules

(1) The clustered indices are stored before the nonclustered ones. (Clustered indices are, in general, the most influential. Thus, they should be considered before the nonclustered ones. They also allow the identification of the basic groups.)

(2) The indices in each set (clustered and nonclustered) are ordered according to their total cost, computed on the basis of their weighted total single index atomic costs: $\sum_q w_q \mathrm{COST}_q(\alpha)$. (Indices with higher total single index costs are usually less influential than indices with lower total single index costs.)

This ordering for the survivor list is likely to be one of the best among the possible permutations of indices.

In Figures 6–8 the expansion of the same tree as in Figures 3–5 is shown with $N = 3$ and $L = 1$. (We assume the order of the survivor list is the same as for Figures 3–5.) The circles indicate the best solutions found at each level of expansion. The best solutions obtained with pruning need not be the same as for full expansion. In this example, however, the number of nodes searched dropped from 48 to 27. If we set $L = 1$, we visit a number of nodes proportional to the number of surviving indices. In general, the number of nodes grows exponentially in L. If we make L the number of surviving indices, we visit the entire tree. The

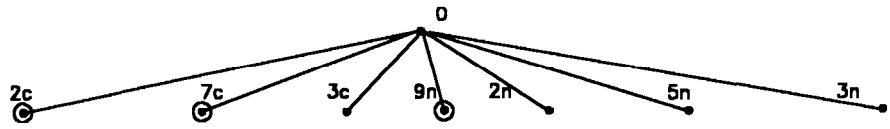

Fig. 6. First expansion with $N = 3$ and $L = 1$.

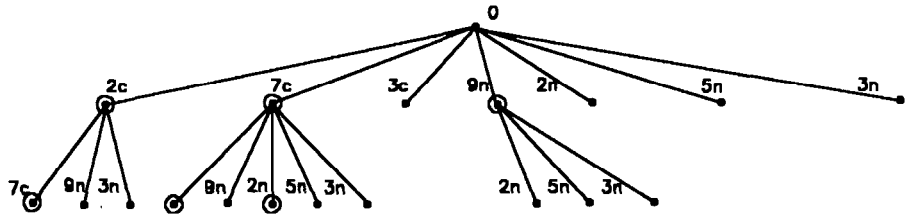

Fig. 7. Second expansion with $N = 3$ and $L = 1$.

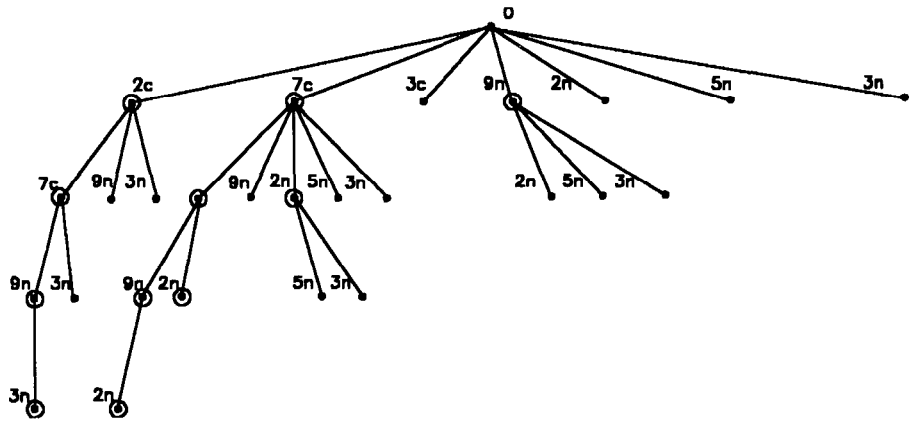

Fig. 8. Full expansion with $N = 3$ and $L = 1$.

trade-off is that some of the low-cost solutions appearing in the unrestricted tree may not appear in the restricted tree, so some of the best solutions may be missed. By controlling these two parameters, a designer can get a good set of choices rather quickly. DBDSGN allows a designer to pursue different choices of N, L, and the index storage limit in the same run. We have found that choosing $L = 3$ usually allows DBDSGN to find the best solutions. Even with much larger values of L, this part of the program is comparatively fast, since all the atomic costs are available.

When a designer chooses to use the heuristic parameters in solution generation, not all solutions are examined, and some join costs may never be needed. A designer may prefer that DBDSGN generate join costs dynamically, as they are needed, rather than before solution generation. Also, after DBDSGN displays solutions for the specified heuristic parameters, the designer may choose to

perform solution generation again with different heuristic parameters. DBDSGN supports this without repeating EXPLAIN COST for any cost previously determined.

6. AN EXAMPLE

We now give an example of the type of input that can be submitted to the tool and the type of output it produces. For the example we use a database containing the following three tables:

PARTS: (PARTNO, QONHAND, ...)
ORDERS: (ORDERNO, PARTNO, SUPPNO, DATE, QTY, ...)
QUOTES: (SUPPNO, PARTNO, MINQTY, MAXQTY, PRICE, ...)

The tables are stored in pages of 4 K bytes. **PARTS** has 8,000 tuples of 10 columns each and is stored in 428 pages; **ORDERS** has 24,000 tuples of 10 columns each and is stored in 618 pages; and **QUOTES** has 72,000 tuples of 8 columns each and is stored in 696 pages. The workload is composed of the ten statements shown in Figure 9 with their weights.

The tool detects that 12 columns are plausible for some statements (these are the columns listed in the table definitions reported above). Index elimination with elimination coefficient $\varepsilon = 0$ and storage weight $\sigma = 1$ eliminates two clustered indices (on O.QTY and Q.MAXQTY) and five nonclustered indices (on P.QONHAND, Q.MAXQTY, Q.MINQTY, Q.PRICE, and O.QTY). In addition to the single-index and sequential scan costs, the tool requests 238 join costs when all costs are precomputed.

Figure 10 shows the solution obtained with a full expansion of the tree without memory limit. The figure also shows the costs and access path chosen for each statement, and the total storage and access cost of the proposed solution. Without actually creating the indices (i.e., by providing statistics to the catalogs for the empty table replicas and the indices through an input file), the CPU time spent to run the example was approximately 12 seconds on an IBM 3081.

7. EXTENSIONS TO OTHER ACCESS PATHS

Multicolumn indices are allowed in System R, and DBDSGN deals with them in a way different from the other indices. It considers multicolumn indices only if the designer specifies them. The group of columns the multicolumn index refers to is treated as if it were a separate column.

Indices are the only access paths (besides sequential scan) supported in System R. Links were implemented in System R, but are used only for system catalogs. Hashing is not supported. In this section we discuss briefly how the principles underlying DBDSGN can be applied to systems that support other access paths.

Dealing with hashing is simple since hashing is a single-table access path that determines the physical placement of the records in a table (just as a clustered index does). Some of the major considerations for extending DBDSGN to consider hashing are as follows:

—Hashing on a column is a primary access path that is incompatible with clustered indices (or hashing on other columns) on the same table.

WEIGHT	WORKLOAD STATEMENTS
5	SELECT SUPPNO, PRICE FROM QUOTES WHERE PARTNO = 8993 AND MINQTY < 1000 AND MAXQTY > 1000
5	SELECT X.ORDERNO, X.PARTNO, Y.DESCRIP, X.DATE, X.QTY FROM ORDERS X, PARTS Y WHERE X.PARTNO = Y.PARTNO AND X.SUPPNO = 'CHR' AND X.DATE BETWEEN 83000 AND 831216
5	SELECT SUPPNO, MIN(PRICE), MAX(PRICE) FROM QUOTES WHERE PARTNO = 11175 GROUP BY SUPPNO
20	INSERT INTO ORDERS VALUES (.....)
10	SELECT PARTNO, QTY FROM ORDERS WHERE ORDERNO = 'BNTJFK' ORDER BY QTY
20	UPDATE QUOTES SET PRICE= PRICE*1.1 WHERE SUPPNO = 'JNS'
20	DELETE FROM ORDERS WHERE SUPPNO = 'JNS'
10	SELECT X.PARTNO, X.DESCRIP,Y.PRICE FROM PARTS X, QUOTES Y WHERE X.PARTNO = Y.PARTNO AND Y.SUPPNO = 'WJM'
2	SELECT Y.SUPPNO,Y.ORDERNO,X.PARTNO,X.DESCRIP FROM PARTS X, ORDERS Y, QUOTES Z WHERE X.PARTNO=Y.PARTNO AND Y.SUPPNO=Z.SUPPNO AND Z.PRICE<5000 AND Y.DATE=840316
5	DELETE FROM ORDERS WHERE DATE>840530 AND PARTNO IN (SELECT PARTNO FROM PARTS WHERE QONHAND>1000)

Fig. 9. Workload statements and their weights.

—Hashing is a plausible access path for the same columns for which indices are plausible, except those referenced in clauses involving order (such as ORDER BY, GROUP BY, and column > "value").

—Hashed columns participate in the elimination heuristic in the same way clustered indices do.

—Basic groups are generated for the hashed columns just as for the clustered indices, except that hashing is compatible with a nonclustered index on the same column.

Links are more complicated; for simplicity we consider only a form of nonclustered binary links. (DBDSGN could also be used for more general types of links.) A link provides paths from parent tuples in one table to child tuples in a second table. The link is a bidirectional access path between two tables that have a $1:N$ relationship. Each parent points to its first child, and each child points to

TABLE	CLUSTERED INDEX	NON-CLUSTERED INDEXES
PARTS	QONHAND	PARTNO
ORDERS	SUPPNO	ORDERNO, DATE
QUOTES	SUPPNO	PARTNO

STATEMENT	ACCESS COST	MAINT. COST	ACCESS PATHS
1	17	0	QUOTES.PARTNO
2	4	0	PARTS.PARTNO ORDERS.SUPPNO
3	28	0	QUOTES.PARTNO
4	0	12	ORDERS.SUPPNO
5	12	0	ORDERS.ORDERNO
6	27	1	QUOTES.SUPPNO
7	11	146	ORDERS.SUPPNO
8	339	0	PARTS.PARTNO QUOTES.SUPPNO
9	163	0	PARTS.PARTNO ORDERS.DATE QUOTES.SUPPNO
10	4	8	ORDERS.DATE
	313	0	PARTS.QONHAND

TOTAL COST = 10095	STORAGE COST = 449

Fig. 10. Solution proposed by DBDSGN.

the next, as well as to the parent. The existence of this $1:N$ relationship is a logical constraint (all children must have parents).

Links can be plausible access paths for the execution of joins. For example, assume **ORDERS** and **PARTS** have a $1:N$ relationship based on PARTNO.

(S6)

```
SELECT   P.PARTNO, P.DESCRIP
FROM     PARTS P, ORDERS O
WHERE    P.TYPE = 'BOLT' AND O.DATE > 840109
AND      P.PARTNO = O.PARTNO.
```

A link on PARTNO between **PARTS** and **ORDERS** would help retrieve all **ORDERS** tuples with a given part number, and such a link is a plausible access path for statement (S6).

Links may also be plausible for statements that are not joins. The *concatenated* access path consisting of an index on the PARTNO column of **PARTS** and the PARTNO link is plausible for the following statement, even though the **PARTS** table does not appear explicitly:

(S7)

```
SELECT   ORDERNO, DATE
FROM     ORDERS
WHERE    PARTNO = 7163.
```

The index on **PARTS** can be used to access the tuple with PARTNO = 7163, and the link used to retrieve the **ORDERS** tuples with the same part number.

Some of the major considerations for extending DBDSGN to consider links are that

—1 : N relationships are specified (in the system catalogs),

—concatenated access paths must be considered,

—atomic configurations never include two links between the same pair of tables (unless a table appears more than once in the **FROM** clause of a statement), and

—the access-path elimination heuristic is restricted for links in the same way as for join columns.

For clustered links, linked tables are interspersed in the same data pages to keep parent tuples close to child tuples. Costs depend on the nesting complexity of interspersion, so atomic configurations must be defined based on different levels of interspersion. This requires a more fundamental change to DBDSGN than hashing or nonclustered links.

8. CONCLUSION

In this paper we presented the fundamental principles used in the design and implementation of a practical physical database design tool, DBDSGN. The design methodology is sufficiently general that it can be applied to other DBMSs with optimizers, since it is nearly independent of the optimizer used and can be extended to handle other types of access paths. DBDSGN obtains cost estimates and other information it needs from the DBMS rather than from an external model whose decisions might be different from those of the optimizer. By restricting cost evaluation for each statement to plausible atomic configurations, the number of configurations is significantly reduced. When the problem space is large, index-elimination and solution-generation heuristics can be used to reduce execution time, but still find good solutions.

Our experience with DBDSGN suggests several observations:

—Database tools should be designed and prototyped as the database system is being designed and prototyped, with people familiar with the system involved in building the tools.

—The size of the design problem can be reduced validly. One method is to identify atomic costs from which all costs can be computed. A second method is to identify choices that are implausible. Heuristics should be applied (if necessary) only after application of these valid methods.

—A tool should not use an independent model of the behavior of the underlying system, even if that model is more accurate than the system's internal model. Instead, the database system should export a description of its behavior. Tools based on that description can be independent of many changes in the system. Improvements in formulas and statistics should be incorporated into the optimizer, not into tools.

—The database system can export descriptions of its behavior by storing the descriptions in database tables. This approach supports extraction and summarization of behavior data using standard database operations.

DBDSGN was built as a prototype for System R, and was the basis for an IBM product, Relational Design Tool (RDT), which performs physical design for SQL/DS.

ACKNOWLEDGMENTS

We would like to thank Laura Haas, Irv Traiger, and the referees for their careful reading of this paper and thoughtful comments. Dave Johnson, Ruth Kistler, George Lapis, and Jim Long contributed to the design and implementation of the EXPLAIN statement.

REFERENCES

1. ANDERSON, H. D., AND BERRA, P. B. Minimum cost selection of secondary indexes for formatted files. *ACM Trans. Database Syst. 2*, 1 (Mar. 1977), 68–90.
2. ASTRAHAN, M. M., KIM, W., AND SCHKOLNICK, M. Performance of the System R access path selection mechanism. In *Info Processing 80: Proceedings of the IFIP Congress 80* (Tokyo, Oct. 1980), pp. 487–491.
3. ASTRAHAN, M. M., SCHKOLNICK, M., AND WHANG, K. Y. Counting unique values of an attribute without sorting. *Inf. Syst. 12*, 1 (1987), 11–16.
4. ASTRAHAN, M. M., BLASGEN, M. W., CHAMBERLIN, D. D., ESWARAN, K. P., GRAY, J. N., GRIFFITHS, P. P., KING, W. F., LORIE, R. A., MCJONES, P. R., MEHL, J. W., PUTZOLU, G. R., TRAIGER, I. L., WADE, B. W., AND WATSON, V. System R: Relational approach to database management. *ACM Trans. Database Syst. 1*, 2 (June 1976), 97–137.
5. BLASGEN, M. W., ASTRAHAN, M. M., CHAMBERLIN, D. D., GRAY, J. N., KING, W. F., LINDSAY, B. G., LORIE, R. A., MEHL, J. W., PRICE, T. G., PUTZOLU, G. R., SCHKOLNICK, M., SELINGER, P. G., SLUTZ, P. R., STRONG, H. R., TRAIGER, I. L., WADE, B. W., AND YOST, R. A. System R: An architectural overview. *IBM Syst. J. 20*, 1 (1981), 41–62.
6. BONANNO, R., MAIO, D., AND TIBERIO, P. Secondary index selection in relational database physical design. *Comput. J. 28*, 4 (Aug. 1985), 398–405.
7. BONFATTI, F., MAIO, D., AND TIBERIO, P. A separability-based method for secondary index selection in physical database design. In *Methodology and Tools for Data Base Design*, S. Ceri, Ed. Elsevier North-Holland, New York, 1983, pp. 148–160.
8. CARDENAS, A. F. Analysis and performance of inverted data base structures. *Commun. ACM 18*, 5 (May 1975), 253–263.
9. CHAMBERLIN, D. D., ASTRAHAN, M. M., ESWARAN, K. P., GRIFFITHS, P. P., LORIE, R. A., MEHL, J. W., REISNER, P., AND WADE, B. W. SEQUEL2: A unified approach to data definition, manipulation and control. *IBM J. Res. Dev. 20*, 6 (Nov. 1976), 560–575.
10. CHAMBERLIN, D. D., ASTRAHAN, M. M., KING, W. F., LORIE, R. A., MEHL, J. W., PRICE, T. G., SCHKOLNICK, M., SELINGER, P. G., SLUTZ, D. R., WADE, B. W., AND YOST, R. A. Support for repetitive transactions and ad hoc queries in System R. *ACM Trans. Database Syst. 6*, 1 (Mar. 1981), 70–94.

11. CHAMBERLIN, D. D., ASTRAHAN, M. M., BLASGEN, M. W., GRAY, J. N., KING, W. F., LINDSAY, B. G., LORIE, R., MEHL, J. W., PRICE, T. G., PUTZOLU, F., SELINGER, P. G., SCHKOLNICK, M., SLUTZ, D. R., TRAIGER, I. L., WADE, B. W., AND YOST, R. A. A history and evaluation of System R. *Commun. ACM 24*, 10 (Oct. 1981), 632–646.
12. CHRISTODOULAKIS, S. Implications of certain assumptions in database performance evaluation. *ACM Trans. Database Syst. 9*, 2 (June 1984), 163–186.
13. COMER, D. The difficulty of optimum index selection. *ACM Trans. Database Syst. 3*, 4 (Dec. 1978), 440–445.
14. COMER, D. The ubiquitous B-tree. *ACM Comput. Surv. 11*, 2 (June 1979), 121–137.
15. GANSKI, R. A., AND WONG, H. K. T. Optimization of nested SQL queries revisited. In *Proceedings of the ACM SIGMOD Conference* (San Francisco, May 1987). ACM, New York, 1987, pp. 23–33.
16. HAMMER, M., AND CHAN, A. Index selection in a self-adaptive database management system. In *Proceedings of the ACM SIGMOD Conference* (Washington, D.C., June 1976). ACM, New York, 1976, pp. 93–101.
17. HATZOPOULOUS, M., AND KOLLIAS, J. Y. On the selection of a reduced set of indexes. *Comput. J. 28*, 4 (Aug. 1985), 406–408.
18. HAWTHORN, P., AND STONEBRAKER, M. Performance analysis of a relational database management system. In *Proceedings of the ACM SIGMOD Conference* (Boston, Mass., May 1979). ACM, New York, pp. 1–12.
19. IBM. Relational Design Tool—Structured Query Language/Data System. Ref. Man. SH20-6415-1, IBM, San Jose, Calif., 1985.
20. IBM. SQL-data system application programming. Man. SH24-5018-2, IBM, San Jose, Calif., Aug. 1983.
21. IBM. SQL-data system planning and administration. Man. SH24-5014-1, IBM, San Jose, Calif., Aug. 1983.
22. IBM SYSTEMS JOURNAL. Database 2. *IBM Syst. J. 23*, 2 (1984), 98–218.
23. IP, M. Y. L., SAXTON, L. V., AND RAGHAVAN, V. V. On the selection of an optimal set of indexes. *IEEE Trans. Softw. Eng. SE-9*, 2 (Mar. 1983), 135–143.
24. KIM, W. On optimizing an SQL-like nested query. *ACM Trans. Database Syst. 7*, 3 (Sept. 1982), 443–469.
25. KING, W. F. On the selection of indices for a file. Res. Rep. RJ1641, IBM, San Jose, Calif., Jan. 1974.
26. KOLLIAS, J. G. A heuristic approach for determining the optimal degree of file inversion. *Inf. Syst. 4*, 1 (1979), 307–318.
27. MACKERT, M., AND LOHMAN, G. R* optimizer validation and performance evaluation for local queries. In *Proceedings of the ACM SIGMOD Conference* (Washington, D.C., May 1986). ACM, New York, 1986, pp. 84–95.
28. PUTKONEN, A. On the selection of access paths in inverted database organizations. *Inf. Syst. 4*, 3 (1979), 219–225.
29. RELATIONAL TECHNOLOGY. *An Introduction to INGRES*. Relational Technology. Berkeley, Calif., Jan. 1984.
30. SCHKOLNICK, M. The optimal selection of secondary indices for files. *Inf. Syst. 1* (1975), 141–146.
31. SCHKOLNICK, M. A survey of physical database methodology and techniques. In *Proceedings of the Very Large Database Conference* (Berlin, Sept. 1978), pp. 474–487.
32. SCHKOLNICK, M., AND TIBERIO, P. Considerations in developing a design tool for a relational DBMS. In *Proceedings of the IEEE COMPSAC Conference* (Chicago, Ill., Nov. 1979). IEEE Press, New York, 1979, pp. 228–235.
33. SCHKOLNICK, M., AND TIBERIO, P. Estimating the cost of updates in a relational database. *ACM Trans. Database Syst. 10*, 2 (June 1985), 163–179.
34. SCHMIDT, J. W., AND BRODIE, M. L. *Relational Database Systems Analysis and Comparison*. Springer-Verlag, New York, 1983.
35. SELINGER, P. G., ASTRAHAN, M. M., CHAMBERLIN, D. D., LORIE, R. A., AND PRICE, T. G. Access path selection in a relational database system. In *Proceedings of the ACM SIGMOD Conference* (Boston, Mass., May 1979). ACM, New York, 1979, pp. 23–34.

36. STONEBRAKER, M. The choice of partial inversions and combined indices. *Int. J. Comput. Inf. Sci. 3*, 2 (June 1974), 167–188.
37. STONEBRAKER, M., WONG, E., AND KREPS, P. The design and implementation of INGRES. *ACM Trans. Database Syst. 1*, 3 (Sept. 1976), 189–222.
38. WHANG, K. J. A physical database design methodology using the property of separability. Rep. STAN-CS-83-968, Computer Science Dept., Stanford Univ., Stanford, Calif., May 1983.
39. WHANG, K. Y., WIEDERHOLD, G., AND SAGALOWITZ, D. Separability—An approach to physical database design. *IEEE Trans. Comput. 33*, 3 (Mar. 1984), 209–222.
40. WONG, E., AND YOUSSEFI, K. Decomposition—A strategy for query processing. *ACM Trans. Database Syst. 1*, 3 (Sept. 1976), 223–241.
41. YAO, S. B. Database storage structure design. In *Proceedings of the Conference on Database Engineering* (Amagi, Japan, Nov. 1979). IBM, San Jose, Calif., 1979, pp. 1–22.
42. YOUSSEFI, K., AND WONG, E. Query processing in a relational database management system. In *Proceedings of the Very Large Database Conference* (Rio de Janeiro, Oct. 1979). pp. 409–417.

Received February 1986; revised June 1987; accepted July 1987