Design-By-Example: A Design Tool for Relational Databases

Dina Bitton ¹
Heikki Mannila ²
Kari-Jouko Räihä ³

TR 85-692
July 1985

Department of Computer Science
Cornell University
Ithaca, NY 14853

¹Cornell University, Department of Computer Science.
²University of Helsinki, Department of Computer Science.
³Cornell University, Department of Computer Science. On leave from the University of Tampere, Department of Mathematical Sciences.
Design-By-Example:
A Design Tool for Relational Databases

Dina Bitton

Heikki Mannila

Kari-Jouko Räihä

ABSTRACT

In recent years, research in relational design theory and in query optimization has established a firm ground for designing well-structured logical and physical database schemes. However, the design process requires mastering a considerable amount of theoretical results. Furthermore, even for the initiated database designer, many of the known algorithms for logical design do not provide constructive guidelines for generating a database scheme that would prevent update anomalies and data inconsistencies. Nor do the algorithms and evaluation methods for file structures and query processing provide constructive physical design rules.

We propose an expert tool that would make knowledge in relational design theory and query optimization automatically and transparently available to the database designer. This tool is a system with an interactive, graphical interface that uses examples to guide the designer through several phases of logical and physical database design. Logical design is based on example relations, and physical design on example queries. The example relations are automatically generated by the system. They contain sample data and satisfy the data dependencies that the designer specifies with the assistance of the expert tool. The example queries and their expected frequency are specified by the designer, using graphically displayed skeleton queries. The system generates a physical design scheme that optimizes the mix of queries expected by the designer, and computes a performance forecast. Both example relations and example queries can be modified by the designer, until the expert tool generates a satisfactory design.

---

1 Cornell University, Department of Computer Science.
2 University of Helsinki, Department of Computer Science.
3 Cornell University, Department of Computer Science. On leave from the University of Tampere, Department of Mathematical Sciences.
1. Introduction

In the last decade, the design theory of relational databases has been the focus of intensive research. The notion of "good" logical design (that is the design of relational schemes that minimize data redundancy and are free of update anomalies) has been formalized through sound theoretical concepts such as logical dependencies, normal forms and lossless decompositions. As a result, there is now a better understanding of what constitutes a "good" or a "bad" database design, and several algorithms exist for generating relational database schemes that have desirable properties ([Ul82], [Ma83]).

Unfortunately, while our knowledge of design theory has become extensive, little has been done to make this theoretical knowledge readily available to the designer of a real world database. Understanding the consequences of the theoretical results requires a fair amount of sophistication, and applying these results is certainly beyond what can be expected of the growing population of database designers [Gr83].

The same gap exists in the area of physical design, that is the organization of the data in storage and the access methods. A large body of research has resulted in considerable knowledge about many file structures, performance characteristics of access methods and query optimization [TF82], [SACL79]. However, at the best, some of this knowledge is only incorporated in a query optimizer, by the database system programmer. The database designer, on the other hand, must often make uninformed choices in the physical design process, for the lack of systematic assistance. In most cases the physical design process is manual, and prone to errors. As a consequence, after the database becomes operational, performance problems may require the physical design to be changed.

We contend that the gap between theoretical knowledge in database design and bad database design practice can only be filled by making the design process interactive and by using automatic scheme generators. Thus, we propose an expert tool, DBE (for Design By Example), that automatically makes available to the designer important knowledge in rela-
tional database theory and in performance evaluation of database systems. We see the design process as an interactive dialog, where the designer is guided by graphically displayed examples. During an interactive design session, candidate logical and physical schemes for the database are automatically generated by the system, and illustrated through example relations and queries.

For the logical design process (Figure 1), the designer initially specifies the set of attributes with their domains and the data dependencies that exist between these attributes. The system then automatically generates a candidate database scheme with a set of example relations, and assists the designer in verifying that the scheme correctly models the reality. If it does not, the designer will modify the specifications. As some uncertainty usually exists in the specification of the logical dependencies, several iterations through the process will be necessary to reach a desirable database scheme. It is in the automatic generating process that the design theory can be integrated, in a manner that can be completely transparent to the user of the design tool.

For the physical design process (Figure 2), the designer provides critical information about the size of the database, the distribution of the data and the expected pattern of access and retrieval. The system then generates a candidate physical scheme and performance forecasts. Again the process is interactive, and several candidate schemes will be proposed and tested until a desirable design is reached.

Many systems have been previously proposed to support the database design process, but they all differ from DBE in one or more major aspects - usually in the data model. The IDBD system [DP82] meets criteria similar to ours, but it was conceived for the network model and concentrates more on physical than logical design. Toronto’s Taxis system [MBGW84] as well as the DATAID system [BDD84] are based on the Entity-Relationship model, which we have abandoned because of our aim to stay within one data model. GAMBIT [BDRZ84] is based on an extended relational model. Only the RED1 system [BH84] is
FIGURE 1: LOGICAL DESIGN BY EXAMPLE
FIGURE 2: PHYSICAL DESIGN BY EXAMPLE
based on the pure relational model, but it provides neither the integrated example-based environment nor the graphical interface in the style of [We84] that we are aiming at.

The recent trend on graphical design tools may be compared to the development of user interfaces in the last decade. In early database systems, interactive access to databases was usually provided through special front-ends built on top of the database management systems. More recently, the emphasis on user-friendly interfaces has motivated the development of tools (such as QBE [ZI77] or gdl/ER [ZM83]) that help the database user specify a query. We conjecture that what has already happened with query interfaces will now happen with design tools. In the endeavor of developing database design tools, we may use some of the lessons learned in the development of query languages and user interfaces.

In the remainder of this report, we will describe more specifically our approach to database design, and explain what makes it particularly attractive in the development of an automatic design tool. The report is organized as follows. Section 2 is devoted to logical design. First, we take a critical look at previously proposed approaches to logical database design and identify important features that a design tool should provide. Then, we explain the role of Armstrong relations as example relations in the logical design process, and present recent theoretical results that establish the ground for generating and using these relations. In Section 3, we describe our approach to physical design and present a preliminary design for an interactive physical design system. In Section 4 we discuss a number of implementation issues and describe our environment for a prototype of DBE. Finally, in Section 5, we summarize our research goals.

2. Logical design

2.1. Approaches to relational database design

Three main approaches have been proposed for designing a relational database scheme:
(1) A higher-level data model is used by the designer to describe the conceptual database scheme. That scheme is then mapped into a relational scheme.

(2) The designer expresses the conceptual scheme using predicates that describe relationships among the attributes. The relation scheme is then extracted from these predicates.

(3) The designer specifies a set of attributes and a set of data dependencies. The relational scheme is generated by a decomposition or a synthesis algorithm, with these sets as input.

Although we are aware that the relational model is not as semantically powerful as several "higher-level" data models ([Co79], [Ch76], [WE80], [AH84]), we have rejected the first alternative, mainly because of our emphasis on ease of use. A great asset of the relational model is its simplicity, and we want our design tool to be based on a single data model that is simple and widespread. For the purpose of developing an automatic scheme generator, another strong argument in favor of the pure relational model (and against less mature data models) is its solid mathematical foundation.

The second approach is related to the recent interest in the universal relation model and one of its key concepts, objects ([FMU82], [MU83]). Here, the designer defines predicates describing the relationships within subsets of attributes that have a meaning independent of the other attributes. Each subset gives rise to an object that can be independently updated. In some respects this is a very attractive approach to database design. It produces database schemes that by their very construction do not suffer from the infamous update anomalies. In addition, it can produce database schemes where the set of relations as a whole enjoys some desirable property such as acyclicity [Fa83]. However, there are also several problems associated with this approach. One emerges when the desired degree of acyclicity is not obtainable with the given input: it is not known how the design should be changed. Another problem is that the quality of the resulting scheme depends crucially on
whether the designer has been able to specify all the relevant predicates. Thus, any uncertainty in the design specifications cannot be dealt with.

The third approach is the most traditional, and it is the one that we have chosen to pursue. There exist several well-understood algorithms based on synthesis or decomposition that generate normalized relation schemes for a specified set of data dependencies. This traditional approach has been criticized because of the difficulty of finding the proper input information for the algorithms (that is the data dependencies). In fact, we just used this same argument against the predicate approach to database design. However, our design tool solves this problem by assisting the designer in the task of specifying the logical data dependencies. This is done by automatically generating example relations that illustrate a candidate database scheme. Indeed, seeing example relations is a very effective way to detect anomalies. Moreover, by adding or modifying tuples in an example relation, the database designer can provide information needed to infer dependencies that may have been missing in the initial specification of the database. Thus, in addition to the traditional algorithms for database design, we propose to use example relations, that are generated by appropriate algorithms and modifiable by the database designer. However, this concept can only be made practical if we know how to efficiently generate example relations and infer dependencies. These issues are addressed in detail in Section 2.2.

Database design based on synthesis and decomposition algorithms has also been criticized for producing one of several possible database schemes, without providing the option to choose one that may be more desirable in practice. It is important that a database design tool leave the choices between alternative schemes to the designer [Ke82], and this is a criteria that we have adopted: our system will be designer-driven. In particular, the designer will have the final word on which design to use, even though the design tool will have tried to help him avoid bad designs.
2.2. Example relations in a logical design tool

In this section, we motivate the use of example relations in logical design, and introduce algorithms that provide the ground for building a design tool based on examples. Many theoretical questions had to be answered before we could decide that the logical design process could indeed be based on the use of example relations. How can examples help the designer in testing a candidate database scheme? Given a set of attributes and their dependencies, what constitutes a "good" example relation? How can example relations be generated? If we modify an example relation, how can we infer a new set of dependencies obeyed by the modified relation? We will show that these questions can be well answered by exploiting properties of Armstrong relations ([Ar74], [Fa82]).

Detecting bad database design

Suppose we are designing a database scheme for storing information about COURSEs, lecture HOURs, lecture ROOMs, and TEACHERs. Suppose that besides the attributes, the database designer has been able to identify the functional dependencies HOUR ROOM \rightarrow COURSE (only one course can be taught in any classroom at any given time) and TEACHER \rightarrow COURSE (each teacher only teaches one course). Let us further suppose that this information is given as input to an algorithm that decomposes the universal relation scheme losslessly into relation schemes in Boyce-Codd normal form. If the first dependency is used as the basis of decomposition, the result would consist of two relation schemes: \{HOUR ROOM COURSE\} and \{HOUR ROOM TEACHER\}. A relation stored according to the first scheme should satisfy the dependency HOUR ROOM \rightarrow COURSE, whereas nothing is required from the second relation.

Suppose that this database scheme is proposed to the designer. Since he has already expressed the data dependencies to the best of his ability, it is unlikely that he will find any

---

4 Given a relation scheme with a set of functional dependencies F, an Armstrong relation for this scheme is a relation that satisfies exactly the dependencies in the closure F."
logical anomalies in the proposed scheme. On the other hand, given the choice, he might have preferred a decomposition that uses the second dependency, yielding the schemes {TEACHER COURSE} and {TEACHER HOUR ROOM}. However, this design also contains the scheme {TEACHER HOUR ROOM} with no enforced dependencies. The latter may produce anomalous relation instances, simply because all the relevant dependencies were not found by the designer. Thus this simple example illustrates the two disadvantages that we have discussed in Section 2, in conjunction with the traditional approach to database design. One is the inability to deal with uncertainty in the specifications of data dependencies, the other is the lack of choice between alternative designs.

**Good example relations**

Let us now try to assist the designer in detecting the anomalies of a candidate scheme by displaying an example relation. Choosing an arbitrary relation is not going to be very helpful; on the contrary, it can be downright dangerous. For suppose that we show the designer the following example relation filled with arbitrary tuples.

<table>
<thead>
<tr>
<th>TEACHER</th>
<th>HOUR</th>
<th>ROOM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gries</td>
<td>Tu 10:10</td>
<td>Ives 120</td>
</tr>
<tr>
<td>Hopcroft</td>
<td>Th 10:10</td>
<td>Ives 120</td>
</tr>
</tbody>
</table>

Since no data dependencies were required to hold in the relation, this is an example of a legal relation. However, it should not be shown to the designer, since it gives the false impression that the suggested design is acceptable. The reason for this is that even though no dependencies were required to hold in the relation, this particular relation satisfies quite a few nontrivial dependencies: \( \text{HOUR} \rightarrow \text{ROOM} \), \( \text{TEACHER} \rightarrow \text{HOUR} \), and all the dependencies derivable from these.

A good example relation should not leave the designer any illusions about what can be stored in the database. Therefore the example relation should satisfy exactly the dependencies that can be derived from the given set of dependencies: no more, no less. In other
words, the example relation should be an Armstrong relation [Ar74].

Returning to our example database, an Armstrong relation for the empty set of dependencies in the relation scheme \{TEACHER HOUR ROOM\} is given below.

<table>
<thead>
<tr>
<th>TEACHER</th>
<th>HOUR</th>
<th>ROOM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gries</td>
<td>Tu 10:10</td>
<td>Ives 120</td>
</tr>
<tr>
<td>Hopcroft</td>
<td>Tu 10:10</td>
<td>Ives 120</td>
</tr>
<tr>
<td>Hopcroft</td>
<td>Th 10:10</td>
<td>Ives 120</td>
</tr>
<tr>
<td>Gries</td>
<td>Tu 10:10</td>
<td>Hollister 110</td>
</tr>
</tbody>
</table>

> From this relation it is easy to see that everything is not as it should be: Gries is supposed to be in two rooms at the same time. Thus, this example relation clearly shows that the candidate scheme was wrong.

**Generating an example relation**

Since we usually require from the relation schemes that they be in normal form, the problem of generating Armstrong relations for normalized relation schemes becomes particularly interesting. The reader is referred to [MR85] for a detailed description and analysis of related algorithms. As an example of the usefulness of putting the theory into a context (i.e. integrating it in a design tool), we can quote some of the results related to normalized schemes. In [BDFS84] it is shown that the size of an Armstrong relation can in general be exponential both in the number of attributes in the scheme and in the number of dependencies. This is a discouraging result; it seems to imply that Armstrong relations are useless in a practical tool. However, once the normal form property is taken into account, much tighter bounds can be derived [MR85]. Even though still exponential in pathological cases, our initial experiments suggest that real-world relation schemes and dependency sets do indeed have Armstrong relations of reasonable size.

---

5 The idea of using Armstrong relations in database design was first proposed by Silva and Melkanoff [SM81]. However, their results concentrate on the universal relation scheme, while we contend that from the designer’s point of view, it is more useful to produce Armstrong relations for all the relation schemes in the database.
For a given set of dependencies there exist many Armstrong relations. Which of these is most useful? Clearly, the relation should be small if the designer is going to draw any conclusions from it. What, then, is the proper measure of size? The number of tuples in the relation is an obvious candidate, which we shall adopt here, too; an alternative might be the number of different values appearing in the relation. Even if we aim at Armstrong relations with few tuples, it is not clear that the minimal relation is the most useful for the designer. For instance, it might be illustrative to show the nonexistence of a functional dependency \( X \rightarrow Y \) by a pair of consecutive tuples that agree on \( X \) and disagree on \( Y \). In our example the anomaly would be even easier to notice if the first and fourth tuples were adjacent.

In a minimal relation one tuple may be (and often is) used as a pair of many other tuples for showing that dependencies do not hold. Therefore it is impossible to juxtapose all the tuples that form pairs. Consequently, it becomes more difficult for the designer to locate the anomalies. This is clearly a case where a suitable structure of an Armstrong relation simply cannot be decided without practical experiments. It is just one example of how the implementation can lead theoretical work to the correct direction: once we know what a good Armstrong relation should look like, we can attack the problem of how such relations are generated efficiently.

**Inferring dependencies from example relations**

Let us now turn to the next step in the design process. Returning to our example again, the reason for the anomalous relation is that the designer had forgotten the dependency \( \text{TEACHER HOUR} \rightarrow \text{ROOM} \) from the set of dependencies. The conventional solution would be to modify the dependency set and run the algorithm again. However, since the anomaly is found by inspecting the Armstrong relation, it appears more natural to remove the anomaly by modifying the relation.
This possibility gives rise to some interesting new problems. As an example, suppose that the designer decides to remove the problem by deleting the fourth tuple. The result has no anomalies; therefore the designer asks the design tool to treat the modified relation as the Armstrong relation. Thus we have a problem that is opposite to the basic generation problem: given a relation $r$, the system should find a (small) set of dependencies $F$ such that $r$ is an Armstrong relation for $F$. Note that this problem always has a solution. An algorithm that solves this dependency inference problem is necessary if we wish to directly exploit the information in the relations without asking the designer to modify the dependencies for a rerun of the algorithm. Such an algorithm is given in [MR85]. Again, this is a problem that previously had not received any attention. simply because Armstrong relations had been studied in isolation of a tool that exploits them.

Open problems

Until now, we have concentrated on logical design using functional dependencies, and in this context, investigated three problems: size bounds for Armstrong relations, efficient generation of small Armstrong relations and efficient inference of dependencies. Partial solutions to these problems are presented in [MR85], and we are currently working on extending these results. On the other hand, we are also beginning to investigate whether other types of dependencies, such as multivalued and inclusion dependencies, can be also dealt with in a design by example process.

Previously known algorithms for generating Armstrong relations (e.g. [GJ82], [BDFS84]) were quite inefficient, sometimes requiring exponential time even in the best case. The new generation algorithm in [MR85] was essential before one could even imagine using Armstrong relations in a practical tool. Further experiments are now needed to improve the behavior of the algorithms in practice.

One open problem is the exact complexity of the dependency inference problem. Another interesting question is the exact complexity of generating minimum Armstrong
relations. At present, the algorithm produces Armstrong relations whose size is at most the square of the size of a minimum relation. A variant of this minimization problem arises when an Armstrong relation is being modified. The modified relation, while certainly an Armstrong relation for some dependency set, may not be minimal. Minimizing the relation would better illustrate to the designer the effect of the modifications. Note that the problem is different from the problem of producing a minimum relation: here we want a smallest possible relation that is contained in the input relation and still satisfies exactly the same set of dependencies.

Besides the basic normalization algorithms, we also intend to include in the system various tests that the designer can activate, such as the test for acyclicity [Fa83]. Their basic incorporation should be easy once the above problems (related to the interface) have been solved. However, the design tool should also assist the designer in achieving the desired property should the test fail. Although some ideas have been proposed (e.g. [BK83], [Sc83b]), the proper way of doing this is still an open question. Undoubtedly, experiments with a prototype system will prove useful here, too.

Although we have only designed and implemented algorithms for databases with functional dependencies, we feel that it is important to extend our work and include at least those dependencies that are easy to understand intuitively and yet extend the semantic power of functional dependencies. Multivalued dependencies [Fa77] satisfy this requirement. Inclusion dependencies ([CFP84], [Sc83a]) are simple, and also occur frequently in modeling properties of databases. Together functional and inclusion dependencies already form a powerful combination for describing the semantics of real-world applications.

3. Physical design and example queries

Our approach to physical design is based on principles similar to those of logical design. Furthermore, we see logical and physical design as two interleaved processes with mechanisms to switch back and forth between logical and physical design phases. Both the
logical and physical design processes rely on an interactive dialog, where the system automatically generates candidate schemes and produces examples that illustrate these schemes for the designer. While the logical design process is driven by example relations, the physical design process is driven by example queries. In the latter, the system assists the database designer in specifying a representative mix of example queries and generates a candidate physical organization of the database. This organization is one that optimizes the expected performance of this mix of queries. Once the system determines an optimal physical scheme, it presents the database designer with its computed forecast for the response time of every example query. In the event that this performance forecast is unacceptable to the database designer, the candidate physical scheme is rejected and a new phase of the physical or logical design process is initiated.

A diagram of this physical design process is pictured in Figure 2. The main features of the system are the query generator and the index selector. The query generator assists the database designer in specifying representative mixes of queries. Based on these queries and on information on the distributions of the attribute values, the index selector selects the attributes on which it is desirable to maintain a clustered or a secondary index. While results in relational design theory constitute the basis for automatically generating candidate logical schemes, knowledge about access methods and query execution algorithms constitute the basis for this automatic index selection process. In existing database management systems, this knowledge is often integrated in the design of a query optimizer. However, physical design is more complex than query optimization. A query optimizer operates in a context where the physical scheme exists, and for each individual query, chooses among a number of possible access paths and query execution algorithms. On the other hand, a physical design system must choose, among a large number of possible storage and indexing schemes, a scheme that optimizes the performance of all possible queries. Finding an optimal scheme in the general case would require an exhaustive evaluation of all possible query execution algorithms against all possible physical schemes. Clearly, the number
of possibilities is too large and our knowledge on performance evaluation is too limited to develop a physical design algorithm that would be theoretically optimal. A number of studies have previously addressed this problem under very restrictive conditions on the type of queries and file organizations that are considered in the design process [HC76, Sc77, SC77, Ya77, Wh83]. Most cost evaluators are also based on the simplifying assumption that all data is uniformly distributed, and define a partial measure of cost such as the number of disk accesses.

In our physical design process, we use a simple heuristic to estimate how the cost of queries is affected by the availability of index structures. We depart from the uniform distribution assumption and use refined cost formulas that we have developed for a number of non-uniform attribute distributions [BV85]. We have limited ourselves to those physical organizations where each relation is stored as a contiguous sequence of records, and the only access methods are single-attribute indices. For each relation, the physical design process selects one clustering attribute, and a number of attributes on which it is desirable to maintain a secondary index.

3.1. The query generator

In the logical design process, we account for the fact that the database designer may not be able to specify correctly the data dependencies. In order to help him refine his initial specification of these dependencies, we use example relations which he can search for anomalies that may result from missing or incorrect dependencies. In the physical design process, we assume that the database designer does not know precisely what the queries should be. Furthermore, he may want to modify or replace a query if there is an indication that it will take too long to execute. Thus, an automatic physical design tool should provide assistance to the designer in the specification of representative queries. This assistance relies on two features. First, we provide graphical aids for the formulation of example queries. Second, after the system selects a desirable physical scheme, we present the
designer with performance forecasts for these queries. Since performance is the goal of physical design, we let the designer accept or reject the candidate physical scheme, based on the expected performance of the example queries.

Our graphical interface is similar to the OBE [Zil82] interface. The system provides graphically displayed skeleton queries and asks the designer to fill in constant values and arithmetic operators to define selection and join predicates.

In order to test our implementation of the physical design procedure, we are using a synthetic database and queries that we have previously developed for a performance evaluation system [BDT83, BT84]. This system is based on a test database which is synthetically generated and provides systematic control over selectivity factors, sizes of projections and joins. Modeling example queries as highly-controlled synthetic queries presents obvious advantages in testing a physical design tool. For instance, the database contains a 10,000 tuples relation tenKtup that has a number of integer and string valued attributes. One attribute, \textit{key}, has unique integer values, in the range 0 to 9,999. Another attribute, named \textit{hundred}, has non-unique values, which are randomly generated in the range 0 to 99. For the random number generator we use either a uniform (in which case there are 100 tuples with the same value of attribute \textit{hundred}), or a skewed distribution (e.g. Zipf-like or normal). An example of a selection query that selects 10\% of the tuples in the relation might be (in the query language QUEL):

\begin{verbatim}
range of t is tenKtup
retrieve (t.all) where (t.key > 99 and t.key < 200)
\end{verbatim}

And an example of a projection query that produces exactly 100 tuples and eliminates 9,900 duplicates could be:

\begin{verbatim}
range of t is tenKtup
retrieve (t.hundred)
\end{verbatim}

Associated with each example query, the designer is asked to specify weights that indicate the relative frequency of each query in a representative mix. At the end of the
query generation phase, the example queries and their associated weights are sent to the index selector.

3.2. The index selector

Selecting clustering attributes and deciding on which other attributes relations should be inverted is the core of the physical design system. This selection process is based on the following information supplied by the database designer:

(1) Relation Structure: For each relation, the designer must input the following data:

(a) Relation Cardinality.

(b) Attribute Distributions: Minimally, the designer must indicate for each attribute its domain and whether or not its values are unique. The designer may also choose to specify a theoretical distribution (e.g., uniform, normal) or an empirical distribution. The default distribution is uniform.

(2) Dependencies: The data dependencies as defined in the logical design process are used to refine estimates of the number of tuples accessed by queries.

The procedure for this selection process is as follows:

(1) Select a clustering attribute for each relation.

(2) Assume that a secondary index exists for all other attributes and estimate the number of blocks accessed by each query.

(3) Repeat the following steps until expected performance degrades:

(a) Compute the expected number of disk accesses for each example query, and for the query mix.

(b) Select the least helpful index, and drop it.

Our main measure of performance for a query is the expected number of blocks it would access, given a set of available indices. Our evaluation of this number is based on formulas previously developed by Selinger et al [SACL79] and Whang [Wh83]. However, the
novelty of our approach is in a method for reducing the number of access paths that are evaluated. Rather than evaluating the cost of a query for all possible configurations of indices, we use a heuristic to identify the most beneficial clustering index, and the least beneficial secondary indices. This heuristic takes into account the frequency distribution of attribute values. We use the input information about the distribution of attribute $A_i$ to estimate the number of tuples that an example query would access if it searched a source relation with a predicate defined on $A_i$. For each pair of $(A_i, Q_j)$, where $A_i$ is an attribute and $Q_j$ is an example query, we define an attribute selectivity:

$$v(A_i, Q_j)$$

to be the cost of an access path for query $Q_j$, which uses a secondary index on $A_i$. This would approximately be equal to the number of tuples that would be accessed by query $Q_j$ if it searched a source relation for tuples satisfying a condition on attribute $A_i$. The attribute selectivities are computed differently depending on the type of relational operation in which the attribute is involved. Currently, we are only considering simple one or two-variable relational queries, and updates that modify only one relation. For instance, suppose that query $Q_j$ is a selection on relation $r$, and $A_i$ is an attribute of $r$. If the selection selects all tuples $t$ of $r$ such that

$$t.A_i = C$$

Then

$$v(A_i, Q_j) = \text{expected frequency of value } C$$

If the selection is defined by a boolean combination of predicates

$$t.A_{i_1} <op> C_1 \ AND \ t.A_{i_2} <op> C_2 \ AND \ \cdots \ OR \ t.A_{i_n} <op> C_n$$

then a selectivity measure is charged to each of the attributes $A_{i_1}, A_{i_2}, \ldots, A_{i_n}$ on the account of query $Q_j$. For a join query $Q_{j'}$, computing the join attribute selectivity requires knowing a joint probability distribution for this attribute in the operand relations. For a projection on attribute $A_i$, we assume that the operation is performed by sorting the
relation. Thus, if the relation’s cardinality is $n$, we assign:

$$v(A_i, Q_j) = n \log_2 n$$

After the matrix of attribute selectivities

$$v(A_i, Q_j), \ i = 1, \ldots, n, \ j = 1, \ldots, k$$

for all the attributes $A_1, \ldots, A_n$ and all the example queries $Q_1, \ldots, Q_k$, is computed, these selectivities are weighted according to the composition of the representative query mix. This produces a weighted average selectivity for every attribute:

$$V(A_i) = \sum_{j=1}^{k} w_j \cdot v(A_i, Q_j)$$

These computed frequencies constitute the basis for selecting one clustering attribute for each relation. Basically, performance can be most improved by clustering with respect to the attribute that has the highest selectivity. However the cost of reorganizing an index may counterbalance the benefit of clustering. Thus for each update query, we compute $u(A_i, Q_j)$, which is the number of block accesses that may be required to reorganize the index on $A_i$ as a result of tuple insertions or deletions performed by $Q_j$, and like for the $v(A_i, Q_j)$, we use the query weights $w_j$ to compute a global $u(A_i)$. For each relation, the attribute candidate for a clustering index corresponds to the highest value of:

$$V(A_i) - u(A_i)$$

Thus, we have a quantitative criterion for completing step (1) in the index selection procedure above. After this step, for each clustering attribute $A_i$, we replace $v(A_i, Q_j)$ by the number of actual blocks on which the tuples accessed by $Q_j$ are clustered (that is, we divide by the number of tuples that fit on one block).

In the selection of secondary indices, we use one additional performance measure: the cost of scanning a relation in the absence of an index. For each non-clustering attribute $A_i$, we compute the cost $scan(A_i, Q_j)$ of query $Q_j$ when an index on $A_i$ is not available. For a selection query with a condition on $A_i$, this corresponds to a sequential scan of the relation.
For a projection or a join, we assume that the algorithm used in the absence of an index is based on sorting the relation. Clearly, in the index selection process, we want to drop first the indices for which \( u(A_i) \) exceeds the benefit of using the index, that is:

\[
     u(A_i) > \text{cost of scan} - V(A_i)
\]

After these indices have been eliminated, the cost of the example queries is computed and the current configuration of indices defines a candidate physical scheme. Should the performance of the queries be unsatisfactory, the next least beneficial index may be dropped. Since the attribute selectivities are only a rough indicator of the cost of an access path, it may be the case that the cost of the query mix or of an individual query can be slightly improved by dropping an attribute which satisfies our approximate measure of effectiveness, that is:

\[
     u(A_i) < \text{cost of scan} - V(A_i)
\]

It should be noted that the index selection procedure is based on a heuristic, rather than on a theoretically optimal algorithm. Also, at this point, we can only compute the attribute selectivities for simple queries. In spite of its limitations, we have chosen to pursue this approach because of its potential for dealing with non-uniformly distributed data. Further research is needed to investigate physical design algorithms under non-uniform assumptions [Ch84, Va85].

4. Implementation issues

The DBE project originated at the University of Helsinki, where a number of logical design algorithms have been implemented as Pascal or Prolog programs. This includes normalization algorithms, various tests (e.g., membership tests and tests for satisfaction of normal form properties) as well as an algorithm for generating Armstrong relations. On the other hand, at Cornell, interest in automated physical design has stemmed from research in performance evaluation of database systems and in query optimization. A set of programs for benchmarking relational database systems has been developed [BDT83, BT84], and
research on query evaluation and optimization is on-going [BV85, Va85]. Many fundamental problems (some of which we have mentioned in Sections 2 and 3) remain to be investigated, in order to formalize our approach to both logical design and physical design. However, we expect the implementation of a prototype to help us in the development of new algorithms.

The kind of interaction that we wish to provide between the design tool and the designer is best implemented on a system that provides pull-down menus and a mouse. We need to be able to display and modify with ease the tables for the example relations, and the graphical representation of queries. We have recently begun an implementation of DBE on the Xerox Dandelion workstations. The programming environment that we have selected is the Interlisp environment. It provides both the advantage of fast prototyping in Lisp and a very high-level graphical interface. We are now concentrating on the user interface component of the design tool. To begin with, this requires dealing with displaying example relations and their dependencies, and with designing menus for modifying database schemes. We are also developing a set of data structures for representing database schemes, example relations, relational queries and updates. After these building blocks are in place, we will turn our attention to a robust implementation of the design algorithms.

5. Summary of research goals

We have proposed an expert tool for the logical and physical design of relational databases. We envision the database design process as a Design by Example process, where the designer can interactively specify logical and physical schema information, and test candidate schemes proposed by the system. The main features of our design tool are summarized below.

Ease of use is guaranteed by the extensive use of examples. The designer is assisted by graphical displays of example relations (containing sample data) and example queries. These examples can be manipulated by the designer and the system may be requested to
use the modified examples to infer a new candidate scheme.

The design process is designer-driven. The designer input (on data semantics, projected access patterns and performance requirements) drives the logical and physical design processes. Although the system guides the designer and avoids clearly bad designs, the final choice among candidate schemes is given to the designer.

Candidate database schemes and example relations are automatically generated. We have several efficient algorithms to support this generation. We also have an algorithm to infer logical dependencies from example relations. Although more work is needed to generalize these algorithms, we have already established a firm ground to implement a robust scheme generator.

Physical design is guided by example queries and timing estimates. Like logical design, physical design is viewed as an iterative process, where candidate schemes are automatically generated by the system.

By following our plan of research and implementation, we hope to show that important knowledge in relational database theory and query optimization can be made automatically and transparently available to the uninitiated database designer. Finally, we are also finding out in our preliminary experiments with automating the logical design process that this implementation can be a valuable tool for research on algorithm design.

References


BDD84 C. Batini, V. De Antonellis, A. Di Leva, Database design activities within the DATAID project. *Database Engineering* 7, 4 (December 1984), 16-21.


Ch76  P.P. Chen, The entity-relationship model - toward a unified view of data. *ACM Transactions on Database Systems* 1, 1 (1976), 9-36.


Co79  E.F. Codd, Extending the database relational model to capture more meaning. *ACM Transactions on Database Systems* 4, 4 (1979), 397-434.


Fa77  R. Fagin, Multivalued dependencies and a new normal form for relational databases. *ACM Transactions on Database Systems* 2, 3 (September 1977), 262-278.


MU83  D. Maier, J.D. Ullman, Maximal objects and the semantics of universal relation databases. ACM Transactions on Database Systems 8, 1 (March 1983), 1-14.


MBGW84 J. Mylopoulos, A. Borgida, S. Greenspan, H.K.T. Wong, Information system design at the conceptual level - the Taxis project. Database Engineering 7, 4 (December 1984), 4-9.


We84  H. Wedekind, Supporting the design of conceptual schemata by database systems. IEEE 1984 International Conference on Data Engineering, April 1984, 434-438.


