Automating Physical Database Design

Automating Physical Database Design: An Extensible Approach

Steve Rozen

March 1993

A dissertation in the Department of Computer Science submitted to the faculty of the Graduate School of Arts and Sciences in partial fulfillment of the requirements for the degree of Doctor of Philosophy at New York University

Approved: ______________________________

Research Advisor

Abstract

In a high-level query language such as SQL, queries yield the same result no matter how the logical schema is physically implemented. Nevertheless, a query's cost can vary by orders of magnitude among different physical implementations of the same logical schema, even with the most modern query optimizers. Therefore, designing a low-cost physical implementation is an important pragmatic problem—one that requires a sophisticated understanding of physical design options and query strategies, and that involves estimating query costs, a tedious and error-prone process when done manually.

We have devised a simple framework for automating physical design in relational or post-relational DBMSs and in database programming languages. Within this framework, design options are uniformly represented as "features", and designs are represented by "conflict"-free sets of features. (Mutually exclusive features conflict. An example would be two primary indexes on the same table.) The uniform representation of design options as features accommodates a greater variety of design options than previous approaches; adding a new design option (e.g. a new index type) merely entails characterizing it as a feature with appropriate parameters.

We propose an approximation algorithm, based on this framework, that finds low-cost physical designs. In an initial phase, the algorithm examines the logical schema, data statistics, and queries, and generates "useful features"—features that might reduce query costs. In a subsequent phase, the algorithm uses the DBMS's cost estimates to find "best features"—features that belong to the lowest-cost designs for each individual query. Finally, the algorithm searches among conflict-free subsets of the best features of all the queries to find organizations with low global cost estimates.

We have implemented a prototype physical design assistant for the INGRES relational DBMS, and we evaluate its designs for several benchmarks, including ASSSAP. Our experiments with the prototype show that it can produce good designs, and that the critical factor limiting their quality is the accuracy of query cost estimates. The prototype implementation isolates dependencies on INGRES, permitting our framework to produce design assistants for a wide range of relational, nested-relational, and object-oriented DBMSs.

To my parents, Barbara L. Rozen and Jerome G. Rozen, Jr.,

and to my daughter, Kate.

Contents

List of Figures

Acknowledgments

Above all, I thank Dennis Shasha, my advisor. Without his patience and encouragement this dissertation would not have been written. To the extent that it is clear, simple, and relevant, it owes an incalculable debt to his guidance. I also thank my readers, Benjamin Goldberg and Edmond Schonberg.

Tom Cheatham of Harvard University and Software Options provided much encouragement and support, as did Glenn Holloway, Judy Townley, and Mike Karr of Software Options. Glenn Holloway also helped me resolve some thorny implementation difficulties. Chip Morris was a source of moral support throughout, and also helped by reading drafts of a conference paper that reported earlier stages of this work (as did Glenn Holloway and Judy Townley).

Anina Karmen-Meade of the Computer Science Department, through her sanity and competence, made dealing with administrative matters a pleasure. NYU's Academic Computing Facility provided the hardware and software for running INGRES; I relied especially on Jeff Barry for advice and information.

This work was also supported by U.S. Office of Naval Research grants N00014-92-J-1719, N00014-91-J-1472, and N00014-90-J-1110, and by U.S. National Science Foundation grants IRI-89-01699 and CCR-9103953, and by DARPA under contract N00014-85-C-0710 with the U.S. Office of Naval Research.

Chapter 1

Motivation and Objectives

Relational database management systems today dominate sales of new database management systems (DBMSs). The success of relational DBMSs is due to the convenient, high-level abstractions they provide to application developers. These are:

- The relational model itself—the relational model is inherently simple, and provides high-level, set-oriented query languages.
- Atomic transactions—atomic transactions guarantee consistency of data that is used by concurrent processes, even in the face of system failures.
- Physical data independence—the formulation of queries does not depend on the particular access paths available (e.g. indexes). Consequently, the database administrator can modify many aspects of the physical database without compromising the correctness of existing queries.

Because of these high-level abstractions, relational DBMSs support a classic "programming by refinement" paradigm, in which there is a separation of correctness concerns from performance concerns [DGLS79]. In this paradigm, developers can first concentrate on producing a system in which queries produce correct results, and where the logical schema provides a clean and intellectually straightforward representation of the real-world entities and relationships being modeled. Then, developers concentrate on improving the performance of the applications that depend on the DBMS, partly by adjusting the physical organization of the database to improve query performance.[1]

1.1 Rationale for a Physical Database Design Assistant

The job of improving performance in DBMS applications is sometimes called "database tuning", of which [Sha92] provides a system-independent overview. Physical database design is an important component of database tuning. Our working definition of the physical database design problem will be the following:

> Given a logical database schema and data statistics, such as table size, together with a set of queries on the schema and their frequencies, find a good physical database design—one that a competent human database designer might produce given the same information. (1)

This formulation of the problem is especially suitable for the many database applications where most of the queries are "canned" (executed by application programs). Of course, a physical design for canned applications can also accommodate

[1]In reality, of course, the separation of correctness from performance concerns is not absolute, and we discuss some of the implications of this fact below.

ad hoc queries; execution plans for these can be computed relative to a fixed physical design as in current practice.

Finding a good physical design involves deciding

- which columns should be primary and secondary index keys,
- how to vertically partition logical tables,
- which queries should be materialized views,

and so forth. Physical database design is difficult for a number of reasons:

- Even simple subproblems of physical database design are hard in a formal sense. For example, [Com78] shows that a restricted form of secondary index selection is NP-complete.
- Considerable expertise is needed to understand the performance impact of physical design options offered by a particular DBMS. Vendors's documentation on this performance impact is often sketchy, and advice on critical aspects of physical design, such as selecting appropriate indexes, often leaves the database administrator with many alternatives and no way of deciding among them.
- Many important design options are not actually supported by the DBMS. Salient examples include the ability to materialize aggregate views or to vertically split a table into two physically distinct sub-tables. To employ these strategies the database administrator must breach the separation between logical and physical database design, and revise both the queries and the logical schema. This breach detracts from the clarity of the logical

schema, makes application maintenance more complex, and requires that the designer transform queries on one logical schema to queries on another schema. (And SQL is surprisingly difficult to transform correctly, as evidenced by the difficulty of devising correct algorithms for unnesting nested SQL queries [Kim82, GW87, Mur92].)

- It is tedious and error-prone to evaluate manually the cost of a particular design. One must understand the workings of a particular DBMS's query optimizer, and understand what plan the optimizer is likely to use. Then, one must be able to estimate the cost of the optimizer's plan on the organization. In practice, designers often simply start with a plausible design based on rules of thumb, load it with data, and then run queries on variations of that plausible design.

Additional evidence of the difficulty of physical database design is an empirical study reported in [EHR80]. In this study 11 groups of graduate students (who had already taken introductory database courses) were given three tries to produce a physical database design for a simple CODASYL [COD71] database. In the results the lowest-cost design of the worst group of students was over twice as expensive as the design that had the lowest cost overall. The mean lowest-cost design among all the groups was 37% more expensive than the design that had lowest cost overall. We conjecture that the interposition of a query planner in relational DMBSs makes the task of physical database design more, rather than less, difficult than in CODASYL DBMSs.

Our own experience in database application development [RS89] also suggests that database performance is often a concern, and that a software assistant for physical database design would be a useful adjunct. DBMSs are complex, and demand for experienced database administrators is high; software that can support database administrators by suggesting physical designs and by performing the cost estimates needed to evaluate them can help avoid time-consuming missteps. Software that can help solve the physical design problem (1) might also be used

- for capacity planning,
- to simply estimate the cost of a particular design,
- to produce designs that are partly specified by the database administrator, or
- to perform sensitivity analysis of a given design in the face of varying query frequencies.

In view of the difficulty and importance of physical database design, we set ourselves the goal of providing software to automatically solve (1).

1.2 Related Work

Although there has been much work on relatively restricted subproblems of problem (1), relatively little work has focused on a pragmatic approach to solving (1) itself. We primarily build on efforts reported in [FST88, IBM85] and in [DJCM89, CMD83]. To provide a solution to problem (1), it seems we need to enrich the solution space explored in [FST88]. Also, we want an approach that is applicable to a variety of relational DBMSs, and also to post-relational DBMSs such as nested-relational ($\neg$1NF) and object-oriented DBMSs. Therefore our approach must be more independent of the characteristics of any particular DBMS than that taken in [FST88].

The papers [DJCM89, CMD83] report work in the context of CODASYL DBMSs, and therefore their approach is necessarily somewhat different from our own in many details. Nevertheless there are broad similarities, notably in the reliance both on knowledge-based generation of possible designs and on cost-based searching.

An alternative is to rely primarily on a knowledge-based approach. For example, RdbExpert is a commercial system (for DEC's Rdb) that takes this approach [DEC92, DEC]. Another effort, [CBC93], describes how physical designs produced by a set of rules can be ranked according to a measure of confidence in design quality. Chapter 7 discusses in more detail the relationship of our work to other work.

1.3 Is the Problem Too Open-Ended?

A possible objection to our goal of producing a physical database design assistant is that solutions to the problem are too open-ended to be captured within any fixed set of design strategies. For example, strategies could include moving part of the application out of the DBMS or even buying new hardware. We argue that, even in these cases, having a design assistant such as the one we describe is useful. For example, before moving part of an application out of the DBMS or buying new hardware, one would like to be reasonably sure that the design being used is already close to the best possible, and that a drastic solution is unavoidable.

We also know that one physical design assistant, RdbExpert, is popular among Rdb consultants, even though they may second-guess its designs.

1.4 Formal Statement of the Problem

We now formally define problem (1). Let Cost(Q, D) denote the cost of computing query Q on a physical design D, and let Cost(D) denote the storage cost of the physical design itself. In our prototype system, Cost is a linear function of the CPU time, disk access, and disk pages needed for storage; Section 3.1 presents the details. The coefficient of each term in Cost is a parameter to the system, so the database administrator could, for example, arrange to effectively ignore storage costs if this is reasonable for a particular application. Other cost functions would be possible, for example one based on estimated response time.

Given Cost, the physical database design problem is defined by:

Input: Logical Schema, S, with data statistics, for example the number of rows, the maximum and minimum values in each column, and the number of distinct values in each column.

Workload, $W = \{\langle Q_1, \phi_1\rangle, \ldots \langle Q_n, \phi_n\rangle\}$; each Q_j is a query (or update); each ϕ_j is the frequency of Q_j—the number of times Q_j is executed each hour.

Output: Physical design, D, for S, with low weighted Cost:

$$\mathrm{Cost}(W, D) \overset{\mathrm{def}}{=} \mathrm{Cost}(D) + \sum_{\langle Q_j, \phi_j\rangle \in W} \phi_j \mathrm{Cost}(Q_j, D). \qquad (2)$$

We prefer to use frequencies rather than abstract weights (as [FST88] used) because frequencies allow us to make an informed trade off between query costs and storage costs. In particular, the more frequent a query is the more storage we would be willing to use to make it execute efficiently.

1.5 Example Physical Design Problem

As mentioned above, we want to develop a general framework for physical database design that is applicable to relational DBMSs from different vendors, and also to post-relational DBMSs. For concreteness, however, we apply our methodology to the commercial INGRES DBMS[SKWH76, RS86, Dat87, ING90], and use a running example in describing our framework.

Figure 1.1 shows a simple logical schema[2] annotated with statistics. In this example the logical schema consists of three tables: `parts`, `orders`, and `quotes`. For each table we have the number of tuples (e.g. `quotes` has 10000 tuples) and information about the table's columns. The columns that are part of a logical key are underlined in Figure 1.1; for `quotes` the logical keys are {`sno`, `pno`, `minqty`} and {`sno`, `pno`, `maxqty`}.

For each column in the tables, Figure 1.1 shows the column's type, the number of different values in the column, and the column's minimum and maximum values; the distributions are uniform. For example, the `descrip` column of `parts` has type char(184) (i.e. fixed-length character field of length 184), containing 4000 different values distributed uniformly with the smallest possible value of `"0"` and the largest possible value being a string of 184 'Z's.

[2]Inspired by an example in [FST88], adapted for expositional purposes.

`parts` #tuples=4000

column	<u>pno</u>	qonhand	descrip
type	integer	integer	char(184)
# values	4000	2000	4000
min, max	1, 4000	1, 4000	`"0"`, `"Z..."`

`orders` #tuples=10000

column	<u>ono</u>	pno	sno	date	qty	oprice
type	char(6)	integer	char(3)	integer	integer	money
# values	10000	3000	40	400	5000	1000
min	`"0"`	1	`"AAA"`	19850101	1	.00
max	`"Z..."`	3000	`"ZZZ"`	19930101	1000000	1000.00

`quotes` #tuples=10000

column	<u>sno</u>	<u>pno</u>	<u>minqty</u>	<u>maxqty</u>	price	remarks
type	char(3)	integer	integer	integer	money	char(15)
# values	40	1200	1000	1000	4500	10000
min	`"0"`	1	1	1	0.10	`"0"`
max	`"ZZZ"`	8000	1000000	1000000	1000.00	`"Z..."`

Figure 1.1: Example Schema

Q_0: "Enter an order."

```
insert into orders values (···)
```

Q_1: "Find the smallest number of parts on hand."

```
select min(qonhand) from parts
```

Q_2: "Find a particular order."

```
select * from orders where ono = :hostvar
```

Q_3: "Find orders (and the corresponding part information) for which we might be paying too much, i.e. orders where the order price is greater than some quote for a number of parts no greater than the number of parts ordered."

```
select orders.ono, parts.pno, parts.descrip
from parts, orders, quotes
where parts.pno     =  orders.pno
  and orders.pno    =  quotes.pno
  and quotes.minqty <= orders.qty
  and quotes.price  <  orders.oprice
```

$$\phi_0 = \phi_1 = \phi_2 = \phi_3 = 1/\text{hour}$$

Figure 1.2: Example Workload

Figure 1.2 shows a four-query workload on the schema of Figure 1.1. One might think that the query Q_1 can be computed using the minimum column value supplied with the annotated schema in Figure 1.1. However, minimum column values need only be approximate, and the result of the query must be exact, so this statistic cannot be used to answer Q_1. INGRES never uses such a statistic as the result of a query, because the statistic may be out-of-date.

In query Q_2, `:hostvar` is a variable in a host program that is set to a literal value before the query is executed. In other words, if Q_2 is executed while `:hostvar` has the value `'X23-S6'`, the effect is of executing

```
select * from orders where ono = 'X23-S6'
```

To simplify exposition, we take the frequency of each query in our example to be 1/hour; this does not reduce the complexity of the design problem, and is a not a restriction in our prototype.

Chapter 2

A Framework

As discussed above, we want an approach that is applicable to different relational DBMSs and also to post-relational models such as nested-relational ($\neg$1NF) and object-oriented databases. Retargetability is important because of the variety of relational systems available today, and because of the intense research on extensible DBMSs and DBMS toolkits, which will likely form the basis of tomorrow's post-relational systems [BBG$^+$90, MJC88, Haa90, HHR90, LKD$^+$88, RC87, SCF$^+$86]

Therefore we will develop a general framework that can be instantiated for a particular DBMS. We first characterize a manual approach, then show our approach to automating it.

2.1 Manual Design Methods

A manual approach [ING90, Sha92] to the physical database design problem posed by Figures 1.1 and 1.2 would work something like this:

- Look at the logical schema and the workload.

- Observe that Q_0 is an insert on `orders`; this implies that, as far as Q_0 is concerned, `orders` shouldn't have any indexes, since maintaining the indexes will increase the cost of Q_0.

- Observe that Q_1 references only the `qonhand` column of `parts`. This suggests that Q_1 could be computed efficiently if there were a dense index on `qonhand`.

- Observe that Q_2 is a point query on `ono`, so a primary or secondary index on `ono` would probably help Q_2. (A sparse primary index would likely be preferable to a secondary index.)

- Observe that Q_3 is a "bulk join"—a join where a large number of tuples from two or more tables are joined. This implies that `btree` indexes on the join columns (`parts.pno`, `orders.pno`, `quotes.pno`) might be a good idea.

- Since the requirements of different queries are different, (e.g. Q_0 is better off without an index on `parts`, but Q_2 is helped by an index on `parts`) the designer has to rely on experience or informal cost estimation to determine which queries's "needs" should be satisfied.

- Load the database into the chosen design, and reconsider if the performance of any of the queries is inadequate.

We can characterize this approach as involving two parts:

1. generating design possibilities based on inspection of the queries and logical schema, and

2. some kind of search among the possibilities generated, possibly involving generation of additional design possibilities if there are performance problems.

Our approach will derive from two key intuitions of the manual approach:

1. For a single query, database designers can often rely on rules of thumb to quickly produce a small set of candidate representations that might be advantageous. For example, if a query involves only columns a and b, both in equality selections, then only indexes on a or b or both need be considered. Furthermore, a query plan to go with the representations is also available by rule of thumb. Of course rules of thumb must sometimes be backed up by cost estimation and search, as in a multi-way join. For such cases one can generate potentially useful data organizations and plans (in a way analogous to rule-based generation of candidate query plans as in [Fre87, GD87, Loh88]) and then fall back on cost estimates.

2. To find a good physical organization for several queries one can often narrow the search to a space that is in some sense intermediate between good organizations for the individual queries. For example, suppose one query can be computed efficiently on a table indexed by column a and another can be computed efficiently if the table is indexed by column b, and that neither query can be computed using an index on column c. Then when searching for an organization good for both queries one must consider indexes on a, b, or both, but one need not consider an index on c.

Below, we formalize these two intuitions enough to provide the basis of a practical system.[1]

The broad outlines of our framework for automating (1), then, are:

- Use features to represent different designs options (e.g. indexes).
- Use knowledge-based methods to generate possibly good features for each query.
- Among the features generated for each query, search to find those that are best for that query.
- Search among the union of the best features for all the queries for a set of features that is good for the workload as a whole.

We discuss in detail in Chapter 4 how we can generate features and perform the search.

2.2 Features

We formalize the notion of a feature as follows. We take a particular physical organization (usually one that minimizes storage space), and call this the *basic (physical) schema*. This we represent by the empty set of features. In our example, this would be to store each of the tables `parts`, `orders`, and `quotes` as an unordered sequence of records without any indexes—in INGRES terminology, as a `heap`.

[1]The preceding paragraph is taken from [RS91a], ©VLDB Endowment; reproduction here is permitted under copyright agreement with VLDB Endowment.

To this can be added various additional features. For example, let us denote a primary (resp. secondary) index feature of type τ on columns $\bar{c}$ of table t by $idx(1, \tau, t, \bar{c})$ ($idx(2, \tau, t, \bar{c})$, resp.), $\tau \in \{\texttt{hash}, \texttt{ISAM}, \texttt{btree}\}$. Then a design in which there is a primary `btree` index on `orders.ono` would simply contain a feature $idx(1, \texttt{btree}, \texttt{orders}, \texttt{ono})$, and a good feature set for Q_1 might be $\{idx(1, \texttt{btree}, \texttt{orders}, \texttt{ono})\}$. The aforementioned good design for Q_3 (consisting of primary `btree` indexes for each of the join columns) would be represented by the feature set

$$\{idx(1, \texttt{btree}, \texttt{parts}, \texttt{pno}),\ idx(1, \texttt{btree}, \texttt{orders}, \texttt{pno}),\ idx(1, \texttt{btree}, \texttt{quotes}, \texttt{pno})\}$$

Clearly not every set of features represents a physical schema. For example, the feature set $\{idx(1, \texttt{btree}, \texttt{orders}, \texttt{ono}),\ idx(1, \texttt{btree}, \texttt{orders}, \texttt{pno})\}$ cannot correspond to a physical schema, because a table cannot have two different primary indexes.

To accommodate this fact we refine our notion of feature set as follows.

Definition 1 A feature set that represents a physical schema is a *realizable* feature set.

Definition 2 A set of features, F, is *compressible* if for all realizable $F' \subseteq F$, $F'' \subseteq F'$ implies that F'' is realizable.

Henceforth in this thesis we assume that all feature sets are compressible.[2] This assumption is natural, because it says that whenever a given feature set is realizable, so are its subsets.

Another advantage of this assumption is that it simplifies determination of whether a feature set is realizable; we can restrict ourselves to a notion of conflict among features to determine realizability. For example, $idx(1, \texttt{btree}, \texttt{orders}, \texttt{ono})$ and $idx(1, \texttt{btree}, \texttt{orders}, \texttt{pno})$ conflict.

We now can frame the physical database design problem as follows. Let $\mathrm{Cost}(Q, F)$ denote the cost of computing query Q on a physical schema represented by feature set F, and let $\mathrm{Cost}(F)$ denote the storage cost of the physical design represented by F. We can then replace D in (2) by F, since we are representing designs by feature sets.

To generate features, the design assistant inspects each query in the workload to yield a set of potentially useful features for that query. We want each of the features in this set to be useful in the following sense:

Definition 3 A feature, f, is *existentially useful* to a query, Q, if there exists a realizable feature set, F, such that

1. $F \cup \{f\}$ is realizable, and
2. $\mathrm{Cost}(Q, F \cup \{f\}) < \mathrm{Cost}(Q, F)$.

[2]Given a non-compressible set of features, it is possible to define a different, compressible, set that can express the same physical schemas. For example, if we consider the materialization of a join query as one kind of feature, and consider an index on the materialized join as another kind of feature, then a feature set containing only an index on a materialized join (but not the materialized join itself) is not realizable. (To make the feature set compressible, we would have to consider the index in conjunction with the materialized join as a single feature.)

We next show (in Theorem 1) that we can confine our search to subsets of existentially useful features, and still find a lowest-cost solution.

Definition 4 The set of all existentially useful features for a query, Q, is termed the *complete feature set* of that query, denoted $\mathrm{cfs}(Q)$.

Recall that a workload is a set of queries associated with their frequencies.

Definition 5 Let $W = \{\langle Q_1, \phi_1\rangle, \ldots \langle Q_n, \phi_n\rangle\}$, be a workload. Its complete feature set, denoted $\mathrm{cfs}(W)$, is defined as

$$\bigcup_{\langle Q_j, \phi_j\rangle \in W} \mathrm{cfs}(Q_j).$$

We now show a few results (first shown in [RS91a]) about feature sets

Definition 6 An *ideal feature set* for workload W is any realizable feature set, F, such that for every realizable feature set F', $\mathrm{Cost}(W, F) \leq \mathrm{Cost}(W, F')$.

Theorem 1 *Given a workload, W, there must be some ideal feature set, F, for W such that $F \subseteq \mathrm{cfs}(W)$.*

Proof. To show a contradiction, suppose not, i.e. that for all ideal feature sets, F, $F - \mathrm{cfs}(W) \neq \emptyset$. Consider a minimal ideal feature set, F'. Let $D = F' - \mathrm{cfs}(W)$. By our assumption, $D \neq \emptyset$, so take any $f \in D$, and let $F'' = F' - \{f\}$. Now $\mathrm{Cost}(W, F'') \not> \mathrm{Cost}(W, F')$ (otherwise f would be existentially useful for some query in W). Furthermore F'' is a strict subset of F', implying that F' is not minimal. Contradiction. □

This theorem tells us then, that in searching for an ideal feature set for W, we can confine our attention to subsets of $\mathrm{cfs}(W)$.

Corollary 1 *Given a workload $W = \{\langle Q_1, \phi_1 \rangle, \ldots \langle Q_n, \phi_n \rangle\}$ we can find, for each j, $1 \leq j \leq n$, a realizable $S_j \subseteq \mathrm{cfs}(Q_j)$ such that $\bigcup_{1 \leq j \leq n} S_j$ is an ideal feature set.*

Proof. Consider an ideal feature set, F, that is also a subset of cfs(W). Such an F must exist by Theorem 1. If we let each $S_j = \mathrm{cfs}(Q_j) \cap F$ we satisfy the corollary, since by the assumption that all feature sets are compressible, each $F \cap \mathrm{cfs}(Q_j)$ is realizable, and

$$\begin{aligned} F = F \cap \mathrm{cfs}(W) &= F \cap \bigcup_{1 \leq j \leq n} \mathrm{cfs}(Q_j) \\ &= \bigcup_{1 \leq j \leq n} (F \cap \mathrm{cfs}(Q_j)). \end{aligned}$$

$\square$

Thus, if we could somehow find such S_j's, we could simply take their union to find an ideal feature set for W.

Our algorithm, presented in Chapter 5, does not guarantee that it has selected an S_j for each Q_j. But it uses the heuristic of selecting for each Q_j a set, $best_I(j)$, containing a restricted number features found in the lowest-cost feature sets for that Q_j (see Section 5.1). The hope is that within $\bigcup_{1 \leq j \leq n} best_I(j)$ there will be, if not an ideal set, at least one with low Cost for W. Our experiments suggest that the solutions found do indeed have costs close to that of an ideal set.

The next chapter instantiates this abstract framework for INGRES.

Chapter 3

Instantiation for INGRES

In order to evaluate the utility of the physical database design framework presented above and in [RS91a], we instantiated and implemented it for the commercial INGRES[1] relational database management system [SKWH76, RS86, Dat87, INGa]. We call this instantiation DAD-I (for **DA**tabase **D**esigner—**I**NGRES).

DAD-I operates in three main phases, which we briefly summarize here:

Feature Generation For each query, Q_j in the workload, generate a set of features that are likely to be existentially useful to Q_j. Denote this *useful*(Q_j) or *useful*(j).

Search I Search among each *useful*(j) for a small number of features that are part of the best feature sets for Q_j. Call these $best_I(j)$.

Search II Search among the union of the $best_I(j)$ for a feature set, F, with low weighted aggregate Cost over the entire workload.

[1]We used INGRES v5.0, the most recent release available to us, on a VAX 8650 running VMS.

The remainder of this chapter discusses particulars of the instantiation for INGRES: the Cost function and feature kinds. The following chapters discuss feature generation and Search I and II.

3.1 Costs

We base our Cost function on the cost estimates that INGRES's query optimizer produces. (It would be impractical to try to re-estimate the queries's cost by parsing the ASCII representation that INGRES uses to display its query plans.) The INGRES query optimizer yields its cost estimate in two separate components:

- number of disk access (DRA), and
- "C"s, a measure of CPU utilization.

INGRES documentation does not reveal the relative weights the query optimizer assigns to CPU utilization and disk accesses when choosing a plan. In any case, INGRES's users can do little to influence the plans chosen by the optimizer, so we accept as optimal the plans and estimates that INGRES produces.

However, in order to be able to compare the costs of executing a query on different physical database designs, we must be able to convert the cost of disk access, "C"s, and of storage space to a common currency, which we do with the following formula:

$$Cost_{\$} = K_C C_C + K_D C_D + K_S C_S \tag{3}$$

where C_C is the number of "C"s used per hour, C_D is the number of disk accesses used per hour, and C_S is the number of INGRES pages used. The values for K_C, K_D, and K_S assign relative weights to CPU utilization, disk-access rate, and storage costs. These will likely vary from installation to installation, so the values of these coefficients are parameters to DAD-I. We call the units of Cost "nominal $".

To determine realistic values for the coefficients in 3, one could conduct an analysis similar to that in [GP87]. But in many cases these coefficients are provided by the local accounting mechanism. For example, for our experiments we based these coefficients on the charges made by NYU's Academic Computing Facility, on whose computers we ran the experiments.

3.2 Storage Structures and Query Plans

A brief description of INGRES's storage structures and of the kinds of query plans it generates should provide context for understanding the features and feature-generation procedures used by DAD-I. INGRES constitutes a stress test for our framework in that it offers a richer repertoire of physical table organizations and indexes than any other commercial relational DBMSs, and therefore presents us with a larger design space.

3.2.1 INGRES Storage Structures

INGRES offers three organizations for both primary and secondary indexes: `btree`, `ISAM`, and `hash`. (The next-richest offering is from DEC's RDB, which offers primary and secondary `btree` indexes, and a primary `hash` index; Oracle's version 7 will also offer `hash` indexes.) A primary `btree` index is a dense index, the leaves of which contain pointers to the data pages. A primary `ISAM` index is a multi-level sparse index that potentially requires periodic, explicit reorganization to maintain efficiency. A primary `hash` index organizes a table by the value of a hash function on key values. Tables with a `hash` organization must also be periodically reorganized as the table grows. If reorganizations are required, their frequency depends on the pattern of insertions. Because `ISAM` and `hash` indexes require reorganization as their table grows they are called "static". On the other hand, `btree` organizations adjust automatically to most update patterns, and are called "dynamic".

In `ISAM` tables, insertions involving a sequential index key (i.e. with new values for the index key that tend to monotonically increase or decrease over time) are particularly egregious, because all updates go to the last index page, transforming it into a long overflow chain. Column updates that have the effect of deleting an index key and always inserting it at the end of the index have the same effect.

An INGRES table need not have a primary index, in which case the table is said to be stored as a `heap`. In a `heap` table, tuples are simply stored in the order they are inserted, and there is no data structure to support associative access.

Secondary indexes can have a `btree`, `ISAM`, or `hash` organization. Secondary indexes are implementationally similar to tables, with the leaf pages of a secondary index being analogous to the data pages of a primary table.

In addition to primary and secondary indexes, INGRES offers the option of compressing character data by truncating trailing blanks, but there is no compression of key data in primary indexes (see [ING90], page 10-29).

Unlike some DBMSs (for example Oracle [ORA88]), INGRES provides no mechanism for co-locating (clustering) the rows of more than one table.

3.2.2 INGRES Query Plans

INGRES accepts queries (including data manipulation statements) in a dialect of SQL [Cha76]. The query optimizer translates SQL statements to query plans that specify the actual operations to be performed to compute the query. To understand when a particular design option might be useful to a query, we have to understand INGRES's query optimizer and the plans it produces.

INGRES's query optimizer estimates query costs using detailed data distribution statistics (if these are available). Such data statistics guide query cost estimates. For example, consider the queries

```
select * from employees where married=1                (4)
```

and

```
select * from employees where birth_date='9-apr-88'    (5)
```

and suppose there is a secondary index on both `married` and `birth_date`. If there were three possible values for `married` and 9845 birth dates for 10000 employees in the table, INGRES can store the information that there are approximately 3333.3 records for each `married` value and approximately 1.0157 records for each `birth_date` value. (This information is computed by INGRES only at the database administrator's request.) In this case INGRES would not use the secondary index on `married` to compute query (4), but it would use the secondary index on `birth_date` to compute query (5). (The index on `married` would yield virtually every page in the table anyway, so a scan, with readahead, is preferable for (4).[2])

[2]INGRES can also compute and use information on non-uniform data distributions. However,

In addition to using data statistics, INGRES can

- internally rewrite nested SQL queries as joins [Kim82, GW87],
- use TID intersection (which allows INGRES to use more than one index on a single occurrence of a table in a query's `from` list), and
- generate plans that access only secondary indexes, (rather than the primary table) when the indexes include the necessary columns.

Secondary indexes in INGRES are much like tables. The leaf pages of the secondary index act like the data pages of a table. When a query, Q, can be computed so that all tuples from a correlation name,[3] t, are drawn from secondary index rather than the index's base table, we say the index is *column-sufficient* for t in Q.

In processing joins, the optimizer uses several strategies: nested loop, two variants of sort-merge, index join,[4] and, for SQL nested subqueries, a so-called "subquery" join, where a sub-query is evaluated first, and its output is used as the inner table of the join [ING90].

3.3 Kinds of Features

In designing a set of features to capture the design options available in INGRES, we tried to strike a balance between the conflicting requirements

- of having few features in order to reduce the search space, and

in INGRES v5.0, the version we used, there is no way to enter these statistics without actually loading all the rows into the table. This would be prohibitive for physical database design, because reorganizing and building indexes on fully loaded tables is too expensive. Therefore we restricted ourselves to uniform distributions.

[3]See Section 4.1.2 for a definition of "correlation name".

[4]Index joins are called "key lookup" joins in [ING90].

- of having a large number of features to get the best possible design.

Every conflict-free feature set represents a state of INGRES's data dictionary.[5] A state of the data dictionary is the set of tables known to the system, what organizations they have (e.g. `heap`, `btree`), and what secondary indexes are available on them. Some feature sets require DAD-I to transform the original queries into semantically identical queries in light of the state of the data dictionary. For example, if one is maintaining a query as a materialized view, updates must be rewritten to maintain the materialized view (since INGRES provides no built-in facility for materialized views).

3.3.1 Features in DAD-I

DAD-I employs the following repertoire of features:

Primary Indexes A single table can have only one primary organization, so different primary index features on the same table conflict. Each primary index feature represents either a dense `btree`, or a sparse `ISAM` or `hash` index.

Secondary Indexes In principle, there can be any number of secondary indexes on a table. However INGRES sometimes produces unrealistically low cardinality estimates when joining two secondary indexes. (Appendix A.1 contains an example.) To avoid designs that rely too heavily on secondary indexes (due to this anomaly) we made secondary indexes on the same table with intersecting sets of columns conflict.

[5]The data dictionary is called the "catalog" in INGRES documentation.

Also, as a heuristic, we do not allow a feature set to contain a primary and a secondary index where the columns of one index are a prefix of the columns of the other. The only exception occurs when one of the indexes has more columns than the other and the other is a `hash` index (for reasons adduced below). Formally, this heuristic states that an `ISAM` or `btree` index, χ, on $a_1, a_2, \ldots, a_n$ conflicts with any other index, χ', on $a_1, a_2, \ldots, a_m$ when $n > m$; if $m = n$ then χ and χ' conflict regardless of their organizations. The rationale for this heuristic is that INGRES can always use an `ISAM` or `btree` index on columns $a_1, a_2, \ldots, a_n$ to compute a selection involving $a_1, a_2, \ldots, a_m$ when $n > m$. Therefore, if there is already an `ISAM` or `btree` index on $a_1, a_2, \ldots, a_n$, an additional index $a_1, a_2, \ldots, a_m$ provides only marginal advantage. (The marginal advantage derives from the fact that the m-column index might have fewer levels, so using it might involve fewer disk accesses.) By contrast, a `hash` index on $a_1, a_2, \ldots, a_n$ is useless unless one has a value for each a_i, whence the exception when the shorter index has a `hash` organization.

Vertical Partitioning (Vertical Splitting, Vertical Declustering) This is the representation of a table as two physical tables, whose columns share a logical key. Formally, let R be a table with columns $A = \{a_1, a_2, \ldots, a_n\}$ and minimal logical keys $K_1, K_2, \ldots, K_m$. Then a *vertical partitioning* of R is a set, $\{A_1,\ A_2\}$, such that

1. $A_1 \cup A_2 = A$, and
2. $\exists K_i.(K_i = A_1 \cap A_2)$

The sets A_1 and A_2 contain the columns of two physical tables which together stand in for R. The second condition guarantees that queries can recover the original table by joining on K_i. Any index on the original table whose columns are contained entirely in one (or both) of A_1 or A_2 is understood to be an index on A_1 or A_2 (or both, respectively). Any other index conflicts with the vertical partitioning $\{A_1, A_2\}$. As a heuristic, a feature set can contain only a single vertical partitioning of a logical table.

An additional heuristic is that a secondary index on exactly the columns of A_1 (or A_2) conflicts with the vertical partitioning on $\{A_1, A_2\}$. The rationale is that (as described below) a query, Q, that can be computed using only A_1 (or A_2) can also be computed using only such an index (i.e. the index is column-sufficient for Q).

Materialized Aggregates This is the maintenance of certain simple aggregate queries as materialized views. For example, consider the following query from the AS^{3}AP benchmark [TOB91]):

```
select min(key) from hundred group by name
```

This query could be stored in a materialized view, that is, in a table mapping each `name` in `hundred` to the minimum `key` associated with it in `hundred`. The implementation of DAD-I materializes simple and `group by`[6] aggregates on a single table for queries that involve `where` selections only on columns in the `group by` clause. Techniques for maintaining materialized views for larger classes of queries are straightforward [Koe81, Pai84, Han87], though

[6]Because of a technical problem with getting cost estimates from INGRES, DAD-I cannot handle `group by` queries. See Section 6.1.

maintaining more complex aggregates is likely to be more expensive, and therefore less useful.

3.3.2 Other Possible Feature Kinds

In addition to the design strategies captured by the features above, there are some design strategies that might be useful to include in DAD-I as features, but that we did not have the resources to implement. These strategies are all considered "last-resort" strategies by expert human designers [Sha92, McG89].

Duplicate Tables This is the replication of (possibly partial) rows from a single table in an additional physical table with a different primary organization.

One of our reasons for deferring generation of duplicate tables is that, in many cases, a column-sufficient secondary index can be as good.

However, we would like to include duplicate tables, eventually, because in some situations table duplication is preferable to a secondary index. For example, a duplicate table can require fewer pages than a secondary index, because the duplicate table need not store TIDs of the primary table.

Join Denormalization By "join denormalization" we mean materializing a join. This strategy can be a good for certain queries. For example, an equi-join where

- the join column is a key of one of the tables,
- there is a two-way inclusion dependency on the join columns (that is, there are no unmatched rows), and

- the equi-join also involves equality selection on the join column or the join column is a key for both tables

would be a good candidate for join denormalization. One good reason for including join denormalization would be to undo a vertical partitioning in the logical schema.

Ad Hoc Nesting (Vertical Anti-Partitioning in [Sha92]) A common pattern in a normalized relational design is to have one table that contains master (or "header") records for some entity, and another table containing a set of zero or more detail records that relate to the master record. For example, in a (financial) bond database one might have a master record describing the static characteristics of a bond, such as its coupon. Another table would contain a set of date/price records representing the bond's price history [RS89]. In this situation one might want to allocate in the master record columns for detail records frequently requested with the master. For canned applications, this can be very advantageous, because a single page access retrieves both master and detail information. On the other hand, this makes query formulation awkward.

Encoding Sparse Domains If a column or set of columns takes on only a few values relative to the total number of tuples in the table, one can save space by encoding each value by short identifier, perhaps an integer. A disadvantage of this is that a join is required to reconstitute the data. Sometimes the mapping from identifier to value changes little over time, and is complied into application programs that use the database.

Other Design Parameters

As a default heuristic, DAD-I uses a compressed organization except on tables subject to `update`s of a variable-length character field. (The database administrator can override the default.) The reason is that (at least in INGRES 5.0 [INGb]) the query optimizer does not consider the cost of expanding compressed data.[7] Therefore, INGRES's query-cost estimates on a compressed organization are at least as low as on the corresponding non-compressed organization.

For both compressed and uncompressed tables with `insert`s or `update`s, by default, DAD-I uses effective fill factors[8] of 90% for `btree` or `ISAM` and 75% for `hash`.[9]

Neither DAD-I's cost model nor the workload presented by the database administrator is likely to capture important considerations that would make non-default compression or fill factors desirable; such considerations include

- the frequency of updates that are absent from the workload, and
- the cost and feasibility of taking the database off-line for reorganizing tables.

Because of these imponderables, DAD-I's user can specify non-default compression or non-default effective fill factors.

Another INGRES design parameter that we leave outside of DAD-I because it is invisible in INGRES's query-cost model is the physical location of tables on

[7]Indeed, much of this cost occurs in updates to the compressed fields, because when the value in a compressed field is lengthened, INGRES might move the record, requiring secondary indexes on the table to be updated. (This is true even if the record is only moved within its page.)

[8]By "effective fill factor" we mean the percentage of each storage page that is used, averaged over the lifetime of a table. This is different from INGRES's "fill factor", which is the percentage of storage page usage when the table is first loaded.

[9]The default (initial) fill factors that INGRES uses for uncompressed tables are 80% for `btree` or `ISAM` and 50% for `hash`.

disk. For example, a table can be located on several disks. Storing a table on several disks is crucial if disk access rates exceed the capacity of a single disk (usually around 30–50 random disk access per second). DAD-I implicitly assumes that tables are located on as many disks as needed to support any given disk access rate.

Other relational DBMSs offer other design options that could be represented as features. For example, Oracle provides the option of co-locating tuples from two tables on the same page, so that they share the same primary organization [ORA88]. If offered in future relational systems, design options such as join indexes [Val87], could easily be represented as features. Indexes for object-oriented databases (for example path indexes [MS86] or class-hierarchy indexes [KKD89]) as well as various clustering disciplines would also be good candidates for representation as features.

Chapter 4

Feature Generation

Recall that DAD-I's first step given a workload, $W = \{\langle Q_1, \phi_1\rangle, \ldots \langle Q_n, \phi_n\rangle\}$, is to generate, for each Q_j, *useful*(j)—a set of features that are likely to be existentially useful to Q_j. We want DAD-I to produce a *useful*(j) that is large enough to contain a good physical design, but not so large that searching among *useful*(j)'s subsets becomes intractable.

4.1 Preliminaries

4.1.1 Static Indexes

Even though it might be necessary to periodically reorganize static indexes, it is important to include them in DAD-I's repertoire, because an INGRES `ISAM` or `hash` index can be much more efficient than a `btree`. For example, in a simple `select` that we tested, the average number of disk accesses was 8.3 on a primary `btree` as opposed to 3.3 on a primary `ISAM`.[1]

It would be technically straightforward to extend DAD-I's cost model to include the costs of table and index reorganization. These costs include not only the machine resources to perform the reorganization (i.e. CPU and disk-access costs), but also the cost of table or index unavailability during the reorganization and the administrative and managerial costs of insuring that the reorganization occurs when needed. This extension would allow DAD-I to estimate the cost of reorganization if it could estimate the rate at which a static structure became disorganized. However, it is not clear how realistic such a cost model would be, since reorganizations typically take place during periods of low load, so we have decided to defer this as a possible future enhancement. In the mean time, our approach is to allow the database administrator to supply two pieces of information on every table:

1. the columns in the table that have sequential update patterns, and
2. whether the database administrator is willing to use a static organization even if the table grows.

This information is used in the following predicates:

Definition 7 Given a table, t, and a workload (understood from the context) $static(t)$ is true iff either

- there are no inserts in the workload, or

[1]The reasons include the fact that the readahead factor is greater for `ISAM` than for `btree` tables, and the fact that primary `btree` indexes are dense while primary `ISAM` indexes are sparse.

- the designer is willing to use a static organization even if the table grows.

Definition 8 *sequential-key*$(t.c)$ is true iff column c on table t has a sequential update pattern in a given workload (where the workload is understood from the context).[2]

If *static*(t), then DAD-I usually generates `ISAM` or `hash` indexes, unless for the first column, c, of the index, *sequential-key*$(t.c)$ is true. (In a few situations DAD-I generates a `btree` index on t when *static*(t); see 4.2 and 4.4 below.) When *sequential-key*$(t.c)$, DAD-I never generates an `ISAM` index on t with first column c, even if *static*(t). DAD-I does sometimes generate a `btree` index on a sequential key. In high-contention workloads this can be problematic, because the transactions serialize on the insertion point. A future version of DAD-I might warn the database administrator about potential situations of this kind, and allow the database administrator to suppress consideration of the `btree`.

4.1.2 Correlation Names

There is a distinction in SQL between *correlation names* and tables (and table names). Briefly, in an SQL query of the form

$$\texttt{select} \cdots \texttt{from}\, t_1\ t'_1,\ \ldots,\ t_n\ t'_n\ \ldots$$

each t'_i is a correlation name for t_i. Correlation names allow the query to mention the same table twice in the `from` list, and still be able to distinguish the two mentions in other parts of the query. If a t'_i is absent (it is optional), then the correlation name for t_i is t_i.

[2]The terminology “incremental key” or “incrementing key” is also used [ING90].

Rules for generating features are often best formulated in terms of correlation names. As a shorthand we often use a correlation name, t', where we would use a table, t. In such cases t' is understood to refer to the table named (or aliased) by t'. For example, "columns of t'" means "columns of t" and "placing an index on t'" means "placing an index on t".

If no table occurs more than once in a query, we can just think of correlation names as table names.

4.2 Indexes for Join Predicates

An *(equality) join predicate* for a pair of correlation names, t, t', in a query, q, is a predicate of the form $t.c$=$t'.c'$.[3] In addition, a join predicate for t and t' is implicit if there is a nested subquery connected by `in`, that is, of the form

$$t.c \texttt{ in (select } t'.c' \texttt{ from } \cdots)$$

There are two ways that INGRES can use an index to evaluate a join predicate:

1. For bulk joins (i.e. joins involving large numbers of tuples from both tables), INGRES can use the ordering properties of a primary `btree` index: if a table is ordered by a join column, a merge-join method does not have to sort.

2. When a join can be computed in such a way that the outer operand has few tuples, INGRES can use a primary index on the inner table to look up

[3]In our experience, INGRES does not use indexes to compute joins where the join predicates involve inequalities.

matching values. This join method is sometimes termed an “index join”, and in [ING90], “key lookup join”.

In tests with INGRES v5.0, every join plan we observed satisfied the following properties:

Property 1 *INGRES does not use a secondary index in evaluating a join, except when the index in question is column-sufficient for one of the join arguments (i.e. when all the necessary columns for one of the join arguments can be read from the index.)*

Property 2 *INGRES uses only the first column in the index key when planning a sort-merge query. For example, if the* `where` *clause contains*

$$t.a = t'.a \texttt{ and } t.b = t'.b$$

then INGRES sorts even if there is a primary `btree` *on* $\langle a, b\rangle$.

Property 3 *INGRES does not recognize that an* `ISAM` *organization maintains data in approximately sorted order by the index key; INGRES estimates the cost of sorting it as if the table had a* `heap` *organization.*[4]

DAD-I's feature-generation procedures are based on these properties.

DAD-I generates one or more primary indexes for every join column, c, on correlation name t, with index types take from the matrix

	sequential-key($t.c$)	
static(t)	Y	N
Y	`hash`, `btree`	`hash`, `ISAM`, `btree`
N	`btree`	`btree`

[4]This is consistent with INGRES 6.2 documentation, [ING90] pages 9-25, 9-26, and 9-29.

where *static*(t) and *sequential-key*$(t.c)$ are as defined in Section 4.1.1.[5] A sequential key probably stresses even a `btree`; nodes split often and their storage utilization is low. However, for a join predicate DAD-I generates a potential `btree` even on a sequential key, because `btree` is the only structure that INGRES will read as ordered input to a merge join (as discussed above).

DAD-I relies on INGRES cost estimates to discard indexes that are unprofitable because they are on small tables.

Property 2 implies that a concatenated index would be preferable to a single-column index only when the following two conditions hold:

- There are predicates of the form

$$r.a = s.b \texttt{ and } r.c = s.d$$

- A primary index on any of $r.a$, $s.b$, $r.c$, or $s.d$ is infeasible because of the possibility of overflow.[6]

Since this is presumably a rare occurrence, and in light of Property 1, DAD-I generates only single-column primary indexes to support join predicates. Were this to prove a problem it could be easily corrected.

For the schema of Figure 1.1 and the workload of Figure 1.2, DAD-I generates the following features for join predicates for query Q_3 (the only query with join predicates):

[5] As a technical elaboration, note that some indexes are intrinsically poor because there are so many tuples per index-key value that either data pages or index-leaf pages suffer from overflow. Before generating any index DAD-I always estimates whether the prospective index key would lead to overflow. If so, DAD-I tries to add additional columns to the index key. This may involve changing a `hash` index to an `ISAM` index (provided the initial column is not a sequential key). See Section 4.6 below for more discussion.

[6] "Overflow" in this case would be due to having too high a "repetition factor" (in INGRES terminology)—so many tuples with a particular value for, say, a, that they would not all fit in a single page.

$idx(1, \texttt{hash}, \texttt{parts}, \texttt{pno})$

$idx(1, \texttt{ISAM}, \texttt{parts}, \texttt{pno})$

$idx(1, \texttt{btree}, \texttt{parts}, \texttt{pno})$

$idx(1, \texttt{btree}, \texttt{orders}, \texttt{pno})$

$idx(1, \texttt{hash}, \texttt{quotes}, \texttt{pno})$

$idx(1, \texttt{ISAM}, \texttt{quotes}, \texttt{pno})$

$idx(1, \texttt{btree}, \texttt{quotes}, \texttt{pno})$

Here `orders` has only a `btree` index because *static*(`orders`) is false. (Also, there are no sequential keys in this workload.)

4.3 Indexes for Selection Predicates

A *selection predicate* is one of the form

$$t.c \; \texttt{between} \; v_1 \; \texttt{and} \; v_2$$

or

$$t.c \, \theta \, v_1 \tag{6}$$

where t is a correlation name, $t.c$ is a column specification involving t, θ is one of `=`, `>`, `>=`, `<`, or `<=`, and v_1 and v_2 are values, in other words, expressions that involve no column specification.[7] A selection predicate of the form (6) is an

[7]We omit certain predicates for the following reasons:

1. $t.c_1 \, \theta \, t.c_2$. We have never observed INGRES to use an index to find tuples satisfying such predicates.
2. $E(t.c) = v_1$, where $E(t.c)$ is some expression in $t.c$ other than $t.c$ itself (e.g. $t.c + 3$), so that $t.c$ is not the immediate operand of the comparison operator. INGRES never uses an index to find tuples satisfying such predicates [ING90].

equality selection predicate if θ is `=`, and every other selection predicate is a *range selection predicate*; $t.c$ is an *equality column* in a query, q, if $t.c$ is involved in an equality selection predicate in q, and $t.c$ is a *range column* if it is involved in a range selection predicate. In either case, $t.c$ is a *selection column.*

The *selectivity* of a conjunctive set of selection predicates, S, on a correlation name, t, is the fraction of the tuples of t that satisfy each predicate in S, and the *yield* is the number of such tuples. Similarly, *page selectivity* of S is the fraction of the pages of t that contain a tuple that satisfies each predicate in S, and the *page yield* is the number of such pages. Page selectivity is at least as large a fraction as selectivity, and, in general, approaches selectivity only when t's table is sorted or clustered by the selection columns in S. When this is not the case, the page yield of S can be approximated from the tuple yield as $P(1-(1-1/P)^{Y_t})$, where P is the number of pages in t's table and Y_t is the tuple yield [Car75]. We also define selectivity, yield, page selectivity, and page yield on a set, A, of columns to be the corresponding selectivity or yield on a conjunctive set of equality predicates, one on each column in A.

We do not want DAD-I to generate an index that has such a high page yield that it will always be better to compute a query using a full-table scan than using the index. We therefore define the function *selective-enough* to yield true when an index is likely to be selective enough to use. In defining *selective-enough*, we make optimistic estimates of the costs of using an index and pessimistic estimates

3. $t.c \neq v_1$. INGRES does not use an index for such a predicate unless it knows from data statistics that the predicate is very selective. However, INGRES v5.0 provides no reasonable interface for loading a fictitious non-uniform distribution, so this cannot arise in DAD-I's implementation for INGRES v5.0.

4. $t.c$ `in` $(v_1, \ldots, v_n)$. This can be rewritten as a disjunct.

of table-scan costs (since we don't want to discard useful indexes).

1. For a primary index $I = idx(1, \tau, t, A)$ of type τ on columns A of table t:

   (a) Let y be the page-yield of A assuming that I provides t's primary organization.

   (b) Let z be an estimate of the number of levels of I that are not cached, and that must therefore be read from disk.

   (c) Let Pages(t) be the total number of pages in t assuming a `heap` organization, and let *heap-readahead* be 3.8—an empirically determined estimate of the readahead (prefetch) factor for scanning an INGRES `heap` organization. (Since the alternative to a primary index is a `heap` organization, we will compare the number of page accesses needed for a primary index to the number needed for scanning a `heap`.)

   selective-enough(I) is defined to be true iff $(y + z) \times \textit{heap-readahead} \leq$ Pages(t).

2. For a secondary index $I' = idx(2, \tau, t, A)$ of type τ on columns A of table t:

   (a) Let y be the page yield of A assuming that t has a `btree` organization. (We will compare the use of the index against the scan of a `btree` primary organization, because this is the most pessimistic assumption for the scan—a `btree` stores relatively few records per page (because of the extra storage needed for the leaf pages), and there is no readahead (prefetch) on a `btree`.)

   (b) Let z be as in 1b, except for I' rather than I.

(c) Let Pages(t) be the number of pages of t assuming a `btree` primary organization.

selective-enough(I') is defined to be true iff $(y + z) \leq \text{Pages}(t)$.

We have observed the following property to hold for INGRES query plans:

Property 4 *INGRES never uses a* `hash` *index to find tuples satisfying a range predicate.*

Because of Property 4, and because we assume that it is not usually advantageous to use index columns to the right of a range column when computing a range query, DAD-I uses a straightforward procedure for generating indexes to support selection predicates. We first require the following definition.

Definition 9 Let S be a set of columns. $\Pi(S)$ is defined to be the set of every sequence of columns that can be drawn from S (not necessarily sequences containing every element of S). In other words, $\Pi(S)$ is the set containing every permutation of every element of 2^S.

Procedure 1

```
Input:   $Q$, a query.
         $t$, a correlation name in $Q$.
         $S(Q,t)$, the set of selection columns for $t$ in $Q$.
         $E(Q,t)$, the set of equality columns for $t$ in $Q$.
Output:  A set, $X$, of indexes on $t$ to be added to $useful(Q)$.

1. For every sequence of columns, $\bar{c} \in \Pi(E(Q,t))$ do:
   (a) If $\bar{c} \neq \langle\rangle$ then
       i. If $static(t)$ let $\tau$ be hash; otherwise let $\tau$ be btree.
       ii. If $selective\text{-}enough(idx(1,\tau,t,\bar{c}))$ then include $idx(1,\tau,t,\bar{c})$ in $X$.[8]
       iii. If $selective\text{-}enough(idx(2,\tau,t,\bar{c}))$ then include $idx(2,\tau,t,\bar{c})$ in $X$.
   (b) For every $r \in S(Q,t) - E(Q,t)$ do:
       i. Let $\bar{c} : r$ the column sequence formed by appending $r$ to $\bar{c}$.
       ii. Let $c$ be the first column in $\bar{c} : r$.
       iii. If $static(t)$ and $\neg sequential\text{-}key(t.c)$ then let $\tau'$ be ISAM; otherwise
            let $\tau'$ be btree.
            A. If $selective\text{-}enough(idx(1,\tau',t,\bar{c}{:}r))$ then include $idx(1,\tau',t,\bar{c}{:}r)$
               in $X$.
            B. If $selective\text{-}enough(idx(2,\tau',t,\bar{c}{:}r))$ then include $idx(2,\tau',t,\bar{c}{:}r)$
               in $X$.
```

[8]Recall from footnote 5 that whenever DAD-I generates an index that suffers from overflow DAD-I will actually try to add additional columns to it.

Because of the definition of Π, this procedure generates a number of indexes that is exponential in the number of selection columns on a correlation name in a query. In the workloads we tested (see Section 6.2) this works fine, because most correlation names have one or no selection column. (In the tests, the maximum number of selection columns per correlation name is six, and in this case five out of the six are range columns.)

For the schema of Figure 1.1 and workload of Figure 1.2, DAD-I generates the following features for selection predicates for Q_2, the only query with a selection predicate:

$idx(1, \texttt{btree}, \texttt{orders}, \texttt{ono})$

$idx(2, \texttt{btree}, \texttt{orders}, \texttt{ono})$

4.3.1 Sharpening Feature Generation for Selections

Although Procedure 1 was adequate for our tests, for workloads with more complicated selection predicates, DAD-I might need a more sophisticated procedure for generating possible indexes. (Such queries might arise in so-called decision-support or strategic-data-access applications [O'N91].)

We might be able to improve Procedure 1 in two ways:

- By generating a single index on a particular set of columns when the order of the columns in the indexes makes no difference in the performance of the index.

  Unless a range column is involved, the order of columns in an index key is unimportant for a single correlation name in a single query. This makes sense, even when compression is used, because the storage requirement for index keys is independent of the order of columns in the keys.

However, some orders will be better than others if one takes several queries into account. For example, if we have the queries

```
select * from t where a = 4 and b = 5          (7)
```

and

```
select * from t where b = 3                    (8)
```

then an index on $\langle b, a\rangle$ might be useful for both (7) and (8), whereas an index on $\langle a, b\rangle$ would be useful only to (7).

- By not generating an index on columns $c_1, \ldots, c_n$ when using one an index $c_1, \ldots, c_n, c_{n+1}$ would require fewer disk access.

  For a given correlation name in a query, it is probably always better to use a single, concatenated index as opposed to using TID intersection on several indexes. (This is not to say, however, that the concatenated index should always involve as many selection columns as possible.) For example, for query (7), a concatenated index on a and b would likely be better than separate indexes on a and b.

  Here again, as with the case of column orders for indexes, when we consider the possibility of more than one occurrence of t's table in Q, or of occurrences of t's table in other queries, we see that there are situations where, for several queries, several indexes, each involving fewer columns, are better than one index using all the selection columns. For example, if in addition to (7) and (8) we had

`select * from` t `where` b `= 3`

then separate indexes on each of a and b might be the best choice.

4.4 Indexes for `order by` Clauses

INGRES generates query plans that take into account the ordering properties of primary `btree` indexes (though not of `ISAM` indexes). In figuring out which indexes will support a given `order by` clause, the main subtlety arises in the interaction between the `order by` clause and selection predicates in the `where` clause. For example, if we have

```
select * from t where t.x = :v order by y
```

and there is an index *idx*(1, `btree`, `t`, `x`, `y`), then INGRES can use this index both to find the qualifying tuples and to guarantee the desired order. However, if there were an index on `t.x` and a `btree` index on `t.y` then INGRES could not use both indexes; it would have to either use the index on `t.y` that supports the order, or use the index on `t.x` that supports the selection predicate.

So, for equality selection predicates the principle is simple: If the `order by` clause begins with columns $c_1, \ldots, c_n$ drawn from correlation name t, and if we have equality predicates on columns $s_1, \ldots, s_q$ of correlation name t, then INGRES can use a `btree` index on columns $s_1, \ldots, s_q, c_1, \ldots, c_n$ of t's table. (We can assume without loss of generality that $\{c_1, \ldots, c_n\} \cap \{s_1, \ldots, s_q\} = \emptyset$, since any columns in the intersection can be removed from the `order by` clause without changing the query's result.)

For *range* selection predicates and for join predicates, we usually cannot use the same index to both find qualifying tuples and guarantee their order. For example, if we have

```
select * from t where t.x > :v order by y
```

there is no index that INGRES can use to both find tuples satisfying the predicate `t.x > :v` and maintain `t` sorted by `y`.

The only exception occurs when the range selection predicate or join predicate is on the first column of the `order by` clause. In this case, since the index generated for the `order by` will also support the join or range selection predicate, there is no need to do anything beyond generating the index for the `order by` clause.

DAD-I generates `btree` indexes to support a query, Q, with an `order by` clause as follows:

1. Let the `order by` clause of Q_j be "`order by` B", and let the *sort specification*, B, be $x_1, \ldots, x_n, x_{n+1}, \ldots$.
2. Let $B' = x_1, \ldots, x_n$ denote the maximal prefix of B such that each x_i, $1 \leq i \leq n$, corresponds to a column of a single correlation name, t, in Q's `from` list.
3. Let $S = \{s_1, \ldots, s_p\}$ be the equality-selection columns for t in Q.
4. $\Pi(S)$ is as defined in Definition 9.
5. For every $\sigma \in \Pi(S)$ add the index $\mathit{idx}(1, \texttt{btree}, t, \sigma|B')$ to $\mathit{useful}(Q)$, where $\sigma|B'$ is the concatenation of σ and B'.[9]

[9] A possible refinement would be to trim columns from the right if this index is needlessly selective.

We generate only `btree` indexes because INGRES does not recognize the ordering properties of `ISAM` indexes, and because `hash` indexes do not order their data.

The workload of Figure 1.2 contains no `order by` queries. However, consider the following query (adapted from [FST88]) on the schema of Figure 1.1.

```
select orders.pno,orders.qty from orders
where orders.ono=:hostvar
order by qty
```

DAD-I would generate the following features to support the `order by` clause in this query:

idx(1, `btree`, `orders`, `ono,qty`)
idx(1, `btree`, `orders`, `qty`)

4.5 Vertical Partitionings

The basic idea of generating partitionings is simple. Let $vp(R, \{A, A'\})$ denote the vertical partitioning of table R into two tables with column sets A and A', as discussed in Section 3.3, above. For query Q on table R let $used(R, Q)$ denote the set of columns of R that are needed to compute Q. Then to generate the vertical partitionings for $useful(Q)$ the conceptual procedure is as follows:

Procedure 2

```
Input:    $Q$, a query in a workload.

Output:   $P$, a set of vertical partitionings to be added to $\mathit{useful}(Q)$.

  1. For each table, $R$, in $Q$ do:
     (a) Let $U$ be the set of column names of $R$.
     (b) Let $Z$ be $\mathit{used}(R, Q)$.
     (c) If $Z \neq U$ and $R$ has at least one logical key then
         i. If $Z$ contains a logical key of $R$, let $K$ be a minimal such key (in
            terms of the number of columns it contains). Add
                 $\mathit{vp}(R, \{Z, (U - Z) \cup K\})$
            to $P$.
        ii. If $Z$ does not contain a logical key of $R$, let $K$ be a logical key
            of $R$ such that the number of columns in $K - Z$ is minimal. If
            $Z \cup K \neq U$ then add
                 $\mathit{vp}(R, \{Z \cup K, (U - Z) \cup K\})$
            to $P$.
```

Given the schema and workload of Figures 1.1 and 1.2, DAD-I adds the following vertical partitioning to *useful*(1):

$\mathit{vp}(\texttt{parts}, \{\{\texttt{qonhand}, \texttt{pno}\}, \{\texttt{pno}, \texttt{descrip}\}\})$

and the following to *useful*(3):

$vp(\texttt{parts},\ \{\{\texttt{qonhand}, \texttt{pno}\},\ \{\texttt{pno}, \texttt{descrip}\}\})$

$vp(\texttt{orders},\ \{\{\texttt{ono}, \texttt{oprice}, \texttt{pno}, \texttt{qty}\},\ \{\texttt{date}, \texttt{ono}, \texttt{sno}\}\})$

$vp(\texttt{quotes},\ \{\{\texttt{minqty}, \texttt{pno}, \texttt{price}, \texttt{sno}\},\ \{\texttt{maxqty}, \texttt{minqty}, \texttt{pno}, \texttt{remarks}, \texttt{sno}\}\})$

Here DAD-I generates $vp(\texttt{parts},\ \{\{\texttt{qonhand}, \texttt{pno}\},\ \{\texttt{pno}, \texttt{descrip}\}\})$ twice: once for Q_1, using step 1(c)ii of Procedure 2, and once for Q_3, using step 1(c)i of Procedure 2. It so happens that INGRES can compute Q_1 using only columns `qonhand` and `pno`, whereas it can compute Q_3 using only columns `pno` and `descrip`.

DAD-I generates no vertical partitionings for Q_0 and Q_2 because the test at step 1c of Procedure 2 fails: Q_0 and Q_2 both use[10] all the columns of `orders`.

The procedure that DAD-I actually uses to generate vertical-partitioning features is complicated by the fact that DAD-I creates the INGRES tables needed for each vertical partitioning at the beginning of the search phase. Therefore, before it starts searching, DAD-I must predict which vertical partitionings it might generate during the search phase. For example, suppose that the procedure above generates $vp(R,\ \{\{a,b,c\},\ \{a,d,e,f,g\}\})$ for a query, Q, with $used(Q) = \{a,b,c\}$, and generates $vp(R,\ \{\{a,b,d\},\ \{a,c,e,f,g\}\})$ for another query, Q', with $used(Q') = \{a,b,d\}$. Then $vp(R,\ \{\{a,b,c,d\},\ \{a,e,f,g\}\})$ would be good for both Q and Q'. And in fact DAD-I might generate this feature during Search II, when DAD-I tries to find a feature set that is good for both Q and Q' (which we discuss below). Such vertical partitionings are *not* added to *useful*(Q) or *useful*(Q'), because if $vp(R,\ \{\{a,b,c\},\ \{a,d,e,f,g\}\})$ turns out to be a poor choice for Q, then so will $vp(R,\ \{\{a,b,c,d\},\ \{a,e,f,g\}\})$.

[10]For an `insert` or `delete` query we consider all columns in the updated table to be *used*.

4.6 Column-Sufficient Indexes

As mentioned in Section 3.2.2, in some cases INGRES can compute a query, Q, by reading only a secondary index for a correlation name, t, in Q. Such an index is said to be "column-sufficient for t in Q". For example, INGRES can compute Q_1 in Figure 1.2 by scanning only the index $idx(2, \texttt{hash}, \texttt{parts}, \texttt{qonhand})$ and without reading any data from the `parts` table. This is advantageous because $idx(2, \texttt{hash}, \texttt{parts}, \texttt{qonhand})$ would involve far fewer pages than the entire table—around 30 as opposed to the 400 pages needed were `parts` stored as a `heap`.

DAD-I generates column-sufficient indexes either

- from an index previously generated for another reason (i.e. for a join or selection predicate, or for an `order by` clause), or
- from scratch, when necessary.

4.6.1 Generating Column-Sufficient Indexes from Other Indexes

Whenever DAD-I generates an index, $idx(k, \tau, R, c_1, \ldots, c_n)$ (of type τ on columns $c_1, \ldots, c_n$ of correlation name t of table R, $k \in \{1, 2\}$) for query Q that is not an update of R, DAD-I also attempts to add a column-sufficient index to $useful(Q)$, using the following procedure, with Z bound to $used(t, Q)$:

Procedure 3

Input: $idx(k, \tau, R, c_1, \ldots, c_n)$, an index.

Z, a subset of R's columns.

Output: $idx(2, \tau', R, c_1, \ldots, c_n, c_{n+1}, \ldots, c_{n+m})$, if this index can be created, where $\{c_{n+1}, \ldots, c_{n+m}\} = Z - \{c_1, \ldots, c_n\}$, and $c_{n+1}, \ldots, c_{n+m}$ are sorted by column name (to canonicalize the column-sufficient indexes).

1. Let the type of the new index be

$$\tau' = \begin{cases} \tau, & \text{if } \tau \in \{\texttt{ISAM}, \texttt{btree}\} \text{ or } Z - \{c_1, \ldots, c_n\} = \emptyset \\ \texttt{btree}, & \text{if } \tau = \texttt{hash} \text{ and } \textit{sequential-key}(R.c_1) \\ \texttt{ISAM}, & \text{otherwise} \end{cases}$$

2. If INGRES can create a secondary index with $n + m$ columns, generate the index

$$idx(2, \tau', R, c_1, \ldots, c_n, c_{n+1}, \ldots, c_{n+m})$$

At step 1, it is important to "convert" `hash` indexes to some other index type when adding more columns to it. The reason is that the new columns render the `hash` index useless to the purpose for which DAD-I originally generated it. For example, $idx(2, \texttt{hash}, t, c)$ can support the predicate $t.c = \texttt{:v}$, but $idx(2, \texttt{hash}, t, c, c')$ will not necessarily support it. If the initial column of the `hash` index being extended is a sequential key, DAD-I must use a `btree` structure for the generated column-sufficient index; otherwise DAD-I can use an `ISAM` index.

In the example of Figures 1.1 and 1.2, DAD-I extends indexes to column-sufficient indexes as follows:

- For Q_2, DAD-I extends the two indexes $idx(k, \texttt{btree}, \texttt{orders}, \texttt{ono})$, $k \in \{1, 2\}$ to

  $$idx(2, \texttt{btree}, \texttt{orders}, \texttt{ono,date,oprice,pno,qty,sno})$$

  Although this column-sufficient index involves *all* the columns of `orders`, DAD-I generates it because it might be useful if some other primary organization is useful to another query—for example, in the case that another query were well-served by a primary `btree` index on `pno`.

- For Q_3, DAD-I performs the following index extensions ($\leadsto$ denotes "extends to"):

  $$\begin{array}{rcl} idx(1, \texttt{btree}, \texttt{parts}, \texttt{pno}) & \leadsto & idx(2, \texttt{btree}, \texttt{parts}, \texttt{pno,descrip}) \\ \left.\begin{array}{r} idx(1, \texttt{ISAM}, \texttt{parts}, \texttt{pno}) \\ idx(1, \texttt{hash}, \texttt{parts}, \texttt{pno}) \end{array}\right\} & \leadsto & idx(2, \texttt{ISAM}, \texttt{parts}, \texttt{pno,descrip}) \\ idx(1, \texttt{btree}, \texttt{quotes}, \texttt{pno}) & \leadsto & idx(2, \texttt{btree}, \texttt{quotes}, \texttt{pno,minqty,price}) \\ \left.\begin{array}{r} idx(1, \texttt{ISAM}, \texttt{quotes}, \texttt{pno}) \\ idx(1, \texttt{hash}, \texttt{quotes}, \texttt{pno}) \end{array}\right\} & \leadsto & idx(2, \texttt{ISAM}, \texttt{quotes}, \texttt{pno,minqty,price}) \end{array}$$

  and

  $$idx(1, \texttt{btree}, \texttt{orders}, \texttt{pno}) \leadsto idx(2, \texttt{btree}, \texttt{orders}, \texttt{pno,ono,oprice,qty})$$

4.6.2 Generating Column-Sufficient Indexes from Scratch

If—after generating column-sufficient indexes by extending indexes originally generated to support join predicates, selection predicates, and `order by` clauses—it turns out that some correlation name, t, for query Q, has no column-sufficient index and that Q does not write to R, then DAD-I generates an arbitrary index on $\mathit{used}(t, Q)$ (with columns in a canonical order).

If possible, DAD-I ensures that the leading column is not a sequential key. If this is possible and $\mathit{static}(t)$, DAD-I generates an `ISAM` index (which has the lowest fill factor and best scanning performance). If this is not possible (because every column in $\mathit{used}(t, C)$ is a sequential key) and $\mathit{static}(t)$, then DAD-I generates a `hash` index. If $\neg\mathit{static}(t)$ DAD-I always generates a `btree` index.

For the example schema and queries of Figures 1.1 and 1.2, INGRES generates a single column-sufficient index from scratch, for Q_1:

$\mathit{idx}(2, \texttt{ISAM}, \texttt{parts}, \texttt{qonhand})$

4.7 Materialized Aggregates

DAD-I generates a materialized-view feature for queries of the form

`select` *aggregate-list* `from` *t*

or

`select` *aggregate-list non-aggregate-list* `from` *t* `where` *W* `group by` *c-list*[11]

where

1. t is a single table reference,

2. every column mentioned in W is also in *c-list*,

[11]Because of a technical problem with getting cost estimates from INGRES, DAD-I cannot handle `group by` queries. See Section 6.1.

3. *aggregate-list* is a list of aggregate-column-function applications—i.e applications of one of `max`, `min`, `count`, `sum`, `avg`, and

4. *non-aggregate-list* is a list of column specifications without aggregate operators.[12]

However, if the estimated cardinality of the table needed to contain the materialized view is no smaller than that of the original table, no feature is generated.

In the example of Figures 1.1 and 1.2, DAD-I would generate a materialized-aggregate feature for query Q_1. Using this feature would involve creating a one-column, one-tuple table and using it to compute Q_1.

Ideally, materialized views would be provided by the DBMS, and designers of many next-generation DBMS seem to be at least considering supporting them (e.g. [CW91, SJGP90]). Restrictions 1–2 allowed us to concentrate our implementation effort on cases that demonstrate the flexibility and breadth of the feature-set framework while not requiring us to implement a fully general view-materialization facility.

Materializing `max` and `min` queries is especially useful in INGRES, which does not use indexes to compute these queries ([ING90], page 9-26).

[12]The elements *aggregate-list* and *non-aggregate-list* can be intermixed.

Chapter 5

Search

After generating *useful*(j) for each query, Q_j, in the workload, W, DAD-I has to find a feature set, $F \in 2^{\bigcup_j useful(j)}$, with low Cost$(W, F)$. DAD-I does this in two phases, Search I and Search II. Search I's job is to find, for each Q_j in the workload, a small number of the best features from *useful*(j). Search II's job is to look for good features in the union of these "best" feature sets.

Figure 5.3 shows all the features generated for the example of Figures 1.1 and 1.2; each feature is labeled for future reference. Figure 5.4 shows *useful*(j) for each query in Figure 1.2.

5.1 Search I

In Search I, DAD-I searches for the "best" features for Q_j in each *useful*(j). The basic idea is to start with at least a lowest-cost feature set, and then to keep adding additional low-cost feature sets (in ascending order of cost), until adding one more would cause the cardinality to exceed $max\text{-}best_I$, as described below.

Label	Feature
0	$idx(2, \texttt{ISAM}, \texttt{parts}, \texttt{qonhand})$
1	$idx(2, \texttt{btree}, \texttt{orders}, \texttt{ono,date,oprice,pno,qty,sno})$
2	$idx(1, \texttt{btree}, \texttt{orders}, \texttt{ono})$
3	$idx(2, \texttt{btree}, \texttt{orders}, \texttt{ono})$
4	$idx(2, \texttt{ISAM}, \texttt{parts}, \texttt{pno,descrip})$
5	$idx(1, \texttt{hash}, \texttt{parts}, \texttt{pno})$
6	$idx(1, \texttt{ISAM}, \texttt{parts}, \texttt{pno})$
7	$idx(2, \texttt{btree}, \texttt{parts}, \texttt{pno,descrip})$
8	$idx(1, \texttt{btree}, \texttt{parts}, \texttt{pno})$
9	$idx(2, \texttt{btree}, \texttt{orders}, \texttt{pno,ono,oprice,qty})$
10	$idx(1, \texttt{btree}, \texttt{orders}, \texttt{pno})$
11	$idx(2, \texttt{ISAM}, \texttt{quotes}, \texttt{pno,minqty,price})$
12	$idx(1, \texttt{hash}, \texttt{quotes}, \texttt{pno})$
13	$idx(1, \texttt{ISAM}, \texttt{quotes}, \texttt{pno})$
14	$idx(2, \texttt{btree}, \texttt{quotes}, \texttt{pno,minqty,price})$
15	$idx(1, \texttt{btree}, \texttt{quotes}, \texttt{pno})$
16	$vp(\texttt{quotes}, \{\{\texttt{minqty}, \texttt{pno}, \texttt{price}, \texttt{sno}\}, \{\texttt{maxqty}, \texttt{minqty}, \texttt{pno}, \texttt{remarks}, \texttt{sno}\}\})$
17	$vp(\texttt{parts}, \{\{\texttt{qonhand}, \texttt{pno}\}, \{\texttt{pno}, \texttt{descrip}\}\})$
18	$vp(\texttt{orders}, \{\{\texttt{ono}, \texttt{oprice}, \texttt{pno}, \texttt{qty}\}, \{\texttt{date}, \texttt{ono}, \texttt{sno}\}\})$
19	$materialize(Q_1)$

Figure 5.3: Features for Example Workload and Schema

$$\begin{aligned}
\textit{useful}(0) &= \{\} \\
\textit{useful}(1) &= \{0, 17, 19\} \\
\textit{useful}(2) &= \{1, 2, 3\} \\
\textit{useful}(3) &= \{4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18\}
\end{aligned}$$

Figure 5.4: *useful*(j) for Example Workload and Schema

An important refinement of this basic idea is to avoid including in the best set those features that do not reduce the cost of Q_j (see step 4, below). For example, consider Q_3 of Figure 1.2. DAD-I discovers the following costs:

$$\begin{aligned}
\text{Cost}(Q_3, \{8, 9, 14\}) &= 480\,\text{“C”},\ 848\,\textsc{dra},\ 1176\,\text{pages} \\
\text{Cost}(Q_3, \{8, 9, 14, 17\}) &= 480\,\text{“C”},\ 848\,\textsc{dra},\ 1260\,\text{pages}
\end{aligned}$$

Feature 17, *vp*(`parts`, $\{\{$`qonhand`, `pno`$\}$, $\{$`pno`, `descrip`$\}\}$), contributes nothing, and should be excluded from $\textit{best}_I(j)$.

We formalize “best” as follows.

1. Let $\textit{max-best}_I$ be a parameter supplied by the database administrator.

2. Let $\text{Cost}^+(Q_j, F) \overset{\text{def}}{=} \text{Cost}(Q_j, F) + \text{Cost}(F)$.[1]

3. Let $\bar{F} = F_1, F_2, \ldots$ be the sequence of all the unique feature sets for which DAD-I evaluated Cost^+ during the search for *useful*(j), ordered so that

    (a) $\text{Cost}^+(Q_j, F_1) \leq \text{Cost}^+(Q_j, F_2) \leq \cdots$, and

[1]We include storage cost (Cost(F)) to provide some pressure toward smaller features sets during the search for the best features in *useful*(j). Without including some term such as this in the objective function for Search I, the iterative improvement algorithm that we present in Procedure 4 below would become stalled in connected regions of same-cost states.

(b) for all F_k, $k > 1$, if $(\mathrm{Cost}^+(Q_j, F_{k-1}) = \mathrm{Cost}^+(Q_j, F_k))$ then $|F_{k-1}| \leq |F_k|$.

In other words, $\bar{F}$ is in ascending order by $\mathrm{Cost}^+(Q_j, F_k)$ and cardinality of F_k.

4. Let $\bar{G}$ be the sequence $\left\langle F_k \text{ in } \bar{F} \text{ s.t. } F_{k'} \subset F_k \Rightarrow k \leq k' \right\rangle$. Intuitively, $\bar{G}$ is $\bar{F}$ after removing feature sets that really are not an improvement over some subset to the left within $\bar{F}$.

We then define

$$best_I(j, \bar{G}, max\text{-}best_I) \stackrel{\text{def}}{=} G_1 \cup \bigcup_{k=2}^{x} G_j \tag{9}$$

where x is the maximal $x \geq 0$ such that $|\bigcup_{k=1}^{x} G_k| \leq max\text{-}best_I$. When $\bar{G}$ and $max\text{-}best_I$ are understood, we write $best_I(j)$.[2]

Complexity of Search I

Clearly, the size of *useful*(j) can be at least as large as the number of selection columns in Q_j. Since DAD-I generates a unique secondary index for each selection column, and since these indexes do not conflict, the number of conflict-free subsets of *useful*(j) can be exponential in the number of selection columns in Q_j. This suggests that an exhaustive search will not be acceptable for Search I for every query, and our experience with the test workloads discussed in Section 6.2 bears out this assessment. DAD-I's solution is to allow the database administrator to supply as a parameter the size of the largest *useful*(j) on which to perform

[2]We don't use simply $\bigcup_{k=1}^{x} G_j$ on the right-hand side of equation (9) because if $|G_1| > max\text{-}best_I$ then $x = 0$. In this case we want at least G_1 in $best_I$(j).

an exhaustive search—the default value for this parameters is 10. If the size of *useful*(j) exceeds this parameter, then DAD-I uses an iterative-improvement randomized search algorithm similar to that in [IK90]. For smaller feature sets an exhaustive search is feasible, and hence preferable, because an exhaustive search can perform fewer data-dictionary updates than II+ to get cost estimates for the same number feature sets.

II+ (Iterative Improvement Plus)

DAD-I uses the following algorithm in Search I:

Procedure 4

Input:	Q_j, a query.
	useful(j), the features generated for Q_j.
	$\textit{max-best}_I$, as discussed above.
	num-tries, a parameter supplied by the database administrator.
Output:	$\textit{ideal}(j)$, a feature set with the lowest cost found.
	$\textit{best}_I(j)$.

1. Let *q-table* be a partial mapping from feature set to query cost, initially $\emptyset$.

2. Let *s-table* be a partial mapping from feature set to storage cost, initially $\emptyset$.

3. Do *num-tries* times:

    (a) Let S be a "random" conflict-free subset of *useful*(j).

```
    (b) Set $X = X - \{S'\}$.
    (c) Add the pair $\langle S, \mathrm{Cost}(Q_j, S) \rangle$ to *q-table*.
    (d) Add the pair $\langle S, \mathrm{Cost}(S) \rangle$ to *s-table*.
    (e) Let $X$ be the set of neighbors of $S$, where "neighbors" is defined below.
    (f) While $X \neq \emptyset$ do:
        i. Let $S'$ be a random element of $X$.
        ii. Add the pair $\langle S', \mathrm{Cost}(Q_j, S') \rangle$ to *q-table*.
        iii. Add the pair $\langle S', \mathrm{Cost}(S') \rangle$ to *s-table*.
        iv. If $\mathrm{Cost}^{+}(Q_j, S') < \mathrm{Cost}^{+}(Q_j, S)$ then set $S = S'$ and go to step 3e.
        v. If $X = \emptyset$, then add more distant neighbors of $S$ (which have not yet
           been considered) to $X$, where "more distant neighbors" is defined
           below.
4. Using *q-table* and *s-table*, compute $\bar{F}$, the sequence of feature sets for which
   $\mathrm{Cost}^{+}$ of $Q_j$ was evaluated, sorted as discussed above.
5. *ideal*$(j)$ is $F_1$, and $best_I(j)$ is $best_I(j, \bar{F}, max\text{-}best_I)$
```

The initial value for S at step 3a is chosen from among subsets of 2^F that do not include both a vertical partitioning and a column-sufficient index on the same table. The set, X, of neighbors of S at step 3e is taken to be the set of immediate super- and subsets of S that are conflict-free. At step 3(f)v, X is re-initialized to contain the neighbors that differ from S by the *replacement* of a feature from S. For example, a "more distant neighbor" of $\{1, 4, 5\}$ might be $\{4, 5, 8\}$ (8 replaces 1). If this step is omitted we have a generic iterative-improvement (II) algorithm.

Figures 5.5 and 5.6 show the effect of step 3(f)v. These graphs plot the minimum query cost found (on the y axis) as a function of the number of different feature sets tried (on the x axis). Minimum query cost is measured in nominal \$, our name for the common currency to which we convert CPU, disk access, and storage costs (using the cost coefficients in Section 6.2.3). Figure 5.5 is for the query `join_4_ncl` in the AS^{3}AP workload (query 14 in our numbering in Appendix B.2), and Figure 5.6 is for query Q_3 from Figure 1.2. For Figure 5.5 we suspect that the observed minimum (for II+) is in fact the minimum, though we did not perform an exhaustive search. For the query of Figure 5.6, an almost-exhaustive search revealed the same minimum as II+. (In fact, for the run of Figure 5.5 each iteration of the loop at step 3 discovered the lowest-cost feature set.) Both graphs cover more than one iteration of the loop at step 3.

These graphs are typical of the of tests we did on the alternative versions of iterative improvement for Search I. In these graphs the effect of step 3(f)v is to make it more likely that the algorithm will find a minimum within a given number of query optimizations.

In [IK90], a related strategy, two-phase optimization (2PO), is found to be superior to II for optimizing large join queries. 2PO begins with an II optimization, and then runs a simulated annealing optimization starting at an optimal result taken from among results produced by II. This means that the search can move from one local minimum to another, lower-cost, local minimum, provided that the intervening states are not too expensive. The authors of [IK90] conclude that the reason is that the state space for optimizing large joins forms a "cup". One of the characteristics of a "cup" is that there is a large region of low-cost states containing many local minima. Once a state in the low-cost region is

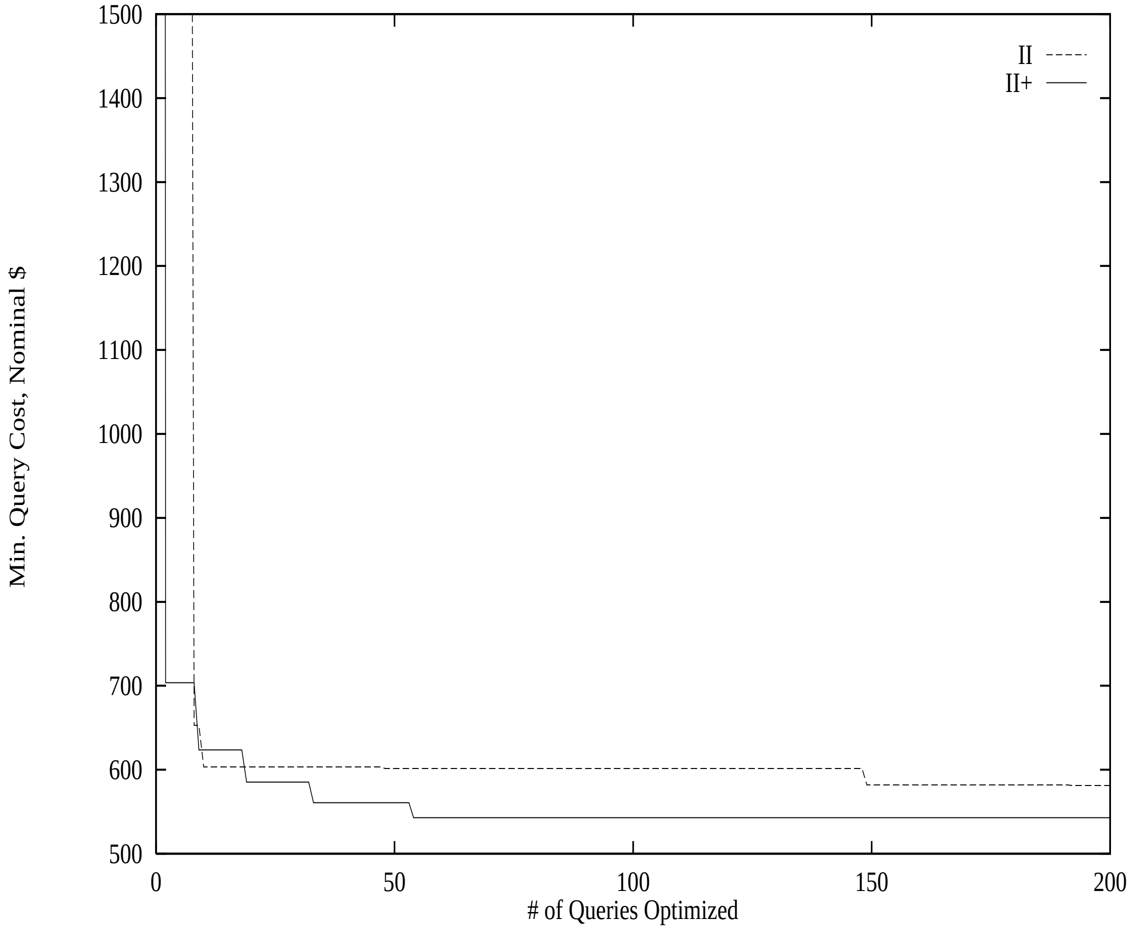

Figure 5.5: II vs. II+, Query `join_4_ncl`, AS^{3}AP

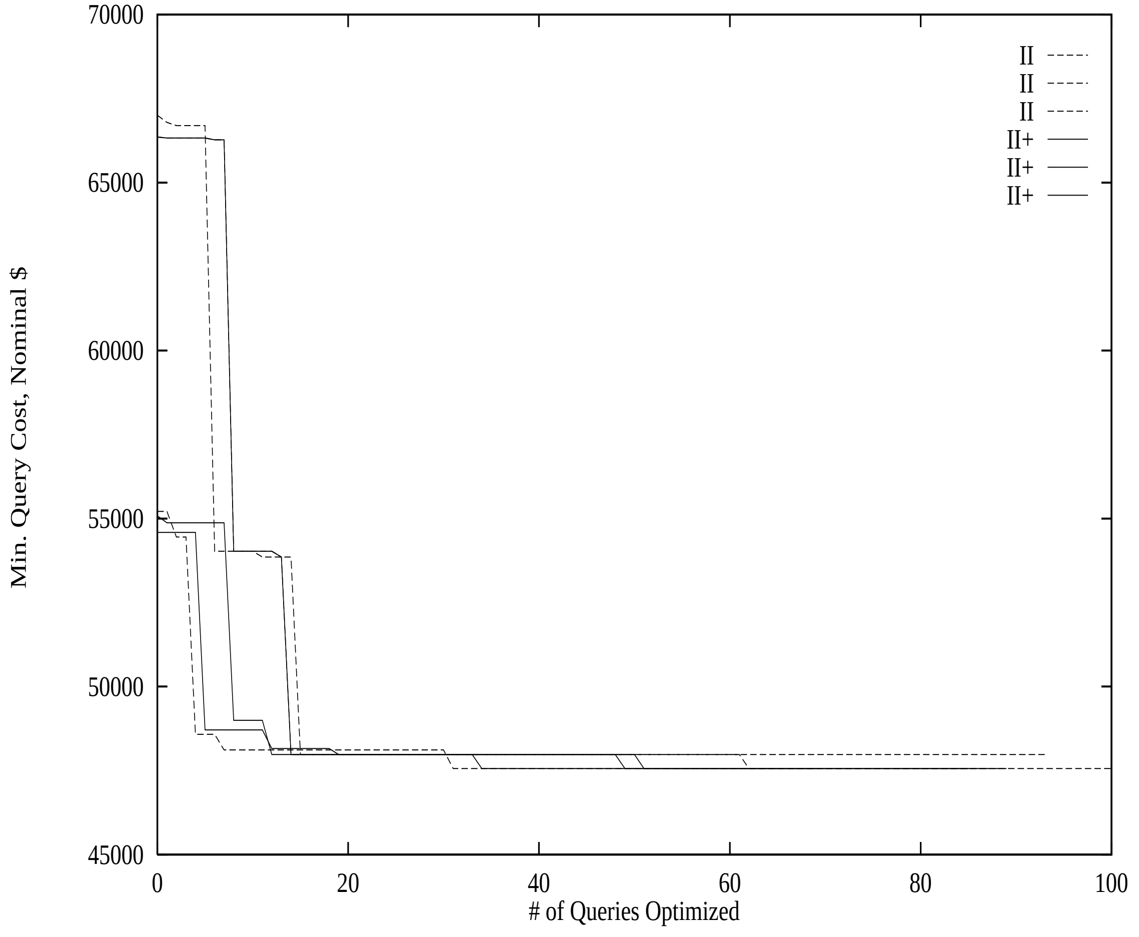

Figure 5.6: II vs. II+, Query Q_3, Example Workload

found, simulated annealing can visit a number of local minima in the region, and hopefully find the best. Widening the neighborhood in II+ seems to have a similar effect; it allows the algorithm to visit nearby low-cost states that would not ordinarily be considered neighbors.

II+ and II were not the only search methods we tried for Search I. We chose II+ because it seems to yield a low-cost feature set quickly. It would be quite easy to attempt other methods, (e.g. simulated annealing or two-phase optimization as discussed in [IK90]).

Search I Results for Example

Given $max\text{-}best_I = 6$, DAD-I produces the following $ideal(j)$ and $best_I(j)$ for the example of Figures 1.1 and 1.2:

query	*ideal*	$best_I$
Q_0	$\{\}$	$\{\}$
Q_1	$\{19\}$	$\{0, 17, 19\}$
Q_2	$\{3\}$	$\{1, 2, 3\}$
Q_3	$\{8, 9, 14\}$	$\{8, 9, 10, 14, 18\}$

5.2 Search II

Once DAD-I has calculated the $best_I(j)$'s and $ideal(j)$'s, it begins Search II. The algorithm depends on $best_{II}(W, \bar{F}, max\text{-}best_{II})$, which is similar to $best_I$, except that $\bar{F}$ contains feature sets for which Cost of W (rather than Cost$^+$ of Q_j) has been evaluated, and the interesting cost for ordering $\bar{F}$ is Cost of W rather than Cost$^+$ of Q_j.

Given this definition of $best_{II}$, the Search II algorithm is:

Procedure 5

Input: $ideal(j)$ and $best_I(j)$ for every query in a workload, W.

$max\text{-}best_{II}$, a parameter supplied by the database administrator.

Output: $ideal \subset \bigcup_j best_I(j)$ such that Cost(W, *ideal*) is low.

```
1. Let $X$ be the set $\{j\}_{\langle \phi_j, Q_j \rangle \in W}$.
2. Let $j'$ be a $j \in X$ such that Cost($W, ideal(j)$) is minimal.
3. Let $best$ be $best_I(j')$.
4. Let $ideal$ be $ideal(j')$.
5. Set $X = X - \{j'\}$.
6. While $X \neq \emptyset$ do:
    (a) Let $j_{\text{worst}}$ be a $j \in X$ such that $\phi_j$Cost($Q_j$, $ideal$) is maximal.[3]
    (b) Let $S$ be $best_I(j_{\text{worst}}) \cup best$.
    (c) Let $S'$ be the result of applying additional feature-generation procedures to $S$ (see Section 5.2.1, below).
    (d) Perform an iterative-improvement search on $2^{S'}$ similar to Procedure 4, except that the objective function is Cost of $W$ rather than Cost$^+$ of $Q_j$.
```

[3]We use j_{worst} as a heuristic because j_{worst} is making the largest contribution to Cost(W, *ideal*) of any j whose $best_I$ features have not yet been considered in Procedure 5. The hope, then, is that searching feature sets that include elements of $best_I(j_{\text{worst}})$ will result in the largest reduction in the Cost of W.

   (e) Let $\bar{F}$ be as in Procedure 4, step 4, except that we use Cost of W rather than Cost$^+$ of Q_j.

   (f) Set $best = best_{II}(W, \bar{F}, max\text{-}best_{II})$.

   (g) Set *ideal* to the first element in $\bar{F}$.

   (h) Set $X = X - \{j_{\text{worst}}\}$.

7. *ideal* is a feature set minimizing Cost of W, among those sets for which Cost of W was evaluated.

The parameter $max\text{-}best_{II}$ helps bound the size of S at step 6b.

5.2.1 Search II Feature Generation

During Search-II, DAD-I generates features that

1. help join two halves of a vertically partitioned table, or

2. combine pairs of existing vertical partitionings into new vertical partitionings.

Point 1 is straightforward: To reconstitute a table that has been vertically partitioned into two tables with column sets A and A', DAD-I creates a view that joins on $A \cap A'$ (which must be non-empty—see Procedure 2). Indexes for rejoining the split table are not necessary for the queries which originally motivated the vertical partitioning, since these queries don't need to reconstitute the original table. However, in Search II the entire workload is under consideration, so indexes on the columns in $A \cap A'$ might help. Therefore, DAD-I generates indexes on these columns, in essentially the same way that it generates indexes for joins that are explicit in the workload (as discussed in Section 4.2)—the main difference is that DAD-I does not use these indexes as the basis for generating additional column-sufficient indexes. We call indexes generated for reconstituting vertically partitioned tables *rejoin* indexes.

Point 2 requires more explanation. The idea is to combine a vertical partitioning good for a query, Q, and one good for another query, Q', into a vertical partitioning good for *both* Q and Q', as discussed on page 50. We define $fatten(i, i')$, where i and i' are vertical-partitioning features on the same table; $fatten(i, i')$ produces a set of features as follows:

1. Let i be the vertical partitioning $vp(R, \{A, A'\})$, and let i' be the vertical partitioning $vp(R, \{B, B'\})$.

2. Let it be that some Q_j can be computed using only the columns in A and that some Q'_j can be computed using only the columns in B. (Other cases are symmetrical.)

3. Perform step 1c of Procedure 2 with $Z = A \cup B$ to produce, if possible, a vertical partitioning.

We can now fill in the details of step 6c in Procedure 5:

1. Let S be as at step 6b of Procedure 5.

2. Let S'' be the subset of S containing only vertical partitionings.

3. S' at step 6c of Procedure 5 is formed by adding to S

   (a) the rejoin indexes for the vertical partitionings in S'', and

   (b) the results of $\mathit{fatten}(i, i')$ for every pair, (i, i'), of

   vertical partitionings in S''.

5.2.2 Search II Compromise on Example

For our running example from Figures 1.1 and 1.2 we get the following workload costs for each $ideal(j)$:

$$\begin{aligned}
\mathrm{Cost}(W, \{\}) &= 833.001 \text{ “C”},\ 1058.421\,\textsc{dra},\ 738.0\,\text{pages} \\
\mathrm{Cost}(W, \{19\}) &= 829.002 \text{ “C”},\ 954.158\,\textsc{dra},\ 739.0\,\text{pages} \\
\mathrm{Cost}(W, \{3\}) &= 823.002 \text{ “C”},\ 1023.263\,\textsc{dra},\ 809.0\,\text{pages} \\
\mathrm{Cost}(W, \{8, 9, 14\}) &= 502.002 \text{ “C”},\ 1450.105\,\textsc{dra},\ 1176.0\,\text{pages}
\end{aligned}$$

$\mathrm{Cost}(W, ideal(3))$ is lowest, and the maximum $\phi_j \mathrm{Cost}(Q_j, ideal(3))$ is that of Q_1, so $j_{\text{worst}} = 1$. Applying step 6c to

$$\mathit{best}_I(3) \cup \mathit{best}_I(1) = \{0, 8, 9, 10, 14, 17, 18, 19\}$$

yields no new features.

The result of the search among conflict-free subsets of S' at step 6d reveals that

$$\mathit{ideal} = \{8, 9, 14, 19\}$$

and that with $\mathit{max\text{-}best}_{II} = 8$,

$$\mathit{best} = \{0, 8, 9, 10, 14, 17, 18, 19\}.$$

The query with maximal $\phi_j \mathrm{Cost}(Q_j, \mathit{ideal})$ is Q_2, so j_{worst} becomes 2. Since $\mathit{best}_I(2) = \{1, 2, 3\}$, we have $S = \{0, 1, 2, 3, 8, 9, 10, 14, 17, 18, 19\}$, to which again no new features are added. The search at step 6d updates *ideal* to $\{2, 8, 9, 14, 19\}$ and *best* to $\{0, 2, 8, 9, 14, 19\}$.

At this point j_{worst} becomes 0 (with $\mathit{best}_I(0) = \emptyset$). The final application of the feature generation rules at step 6c yields no new features, and step 6d applied to *best* yields no lower-cost set than the current value of *ideal*, so the solution to the example problem is

Label	Feature
2	*idx*(1, `btree`, `orders`, `ono`)
8	*idx*(1, `btree`, `parts`, `pno`)
9	*idx*(2, `btree`, `orders`, `pno,ono,oprice,qty`)
14	*idx*(2, `btree`, `quotes`, `pno,minqty,price`)
19	*materialize*(Q_1)

with $\mathrm{Cost}(W, \{2, 8, 9, 14, 19\}) = 480.007$ "C", 856.0 DRA, 1371.0 pages.

This result is plausible, though for Q_3 many human designers might use primary indexes on `pno` (features 10 and 15) rather than the column-sufficient indexes 9 and 14. This choice would certainly make Q_3 more expensive; DAD-I estimates that Q_3 on $\{8, 10, 15, 16\}$ requires 1109 DRA as opposed to 848 DRA for $\{8, 9, 14\}$. The vertical partitioning of `quotes` (16) does reduce the number of disk accesses over what they would be on the full table, but using a primary `btree` on `pno` to read the data pages requires more disk accesses than the using secondary index 14, because the secondary index (considered apart from its base table) has a sparse organization, whereas the primary key (in conjunction with its base table) has a dense organization.

In fact, there are organizations with lower cost estimates for this example workload, but that were excluded because of the stringent rule of secondary index conflict (see Section 3.3). For example, $\{3, 8, 9, 14, 19\}$ uses less space (and slightly less CPU) than {2, 8, 9, 14, 19}: 1248 pages as opposed to 1371 pages. (Feature 3 is $idx(2, \texttt{btree}, \texttt{orders}, \texttt{ono})$, and feature 2 is $idx(1, \texttt{btree}, \texttt{orders}, \texttt{ono})$.) The reason is that there is less unused space on data pages in a primary `heap` organization for `orders` than in a primary `btree` organization. Nonetheless, DAD-I's result appears to be quite good, considering the approximate nature of the data statistics, query frequencies, and query-plan cost estimates used. Over a five-year system lifetime, this difference amounts to less than 5 nominal \$ (our name for the common currency to which we convert CPU, disk access, and storage costs). This cost is trivial compared to the estimated total cost of running the workload for five years, which is about 47,600 nominal \$.

Appendix A.2 shows the query plans INGRES selected for $\{2, 8, 9, 14, 19\}$.

Chapter 6

Experiments

6.1 Implementation of DAD-I

DAD-I's implementation follows closely the description in the preceding chapters. The entire system is coded in about 13,000 lines of Common LISP. We chose Common LISP for portability and ease of experimentation. The LISP part of DAD-I was not a performance bottleneck; the salient bottleneck appeared to be the time used by INGRES to update its data dictionary (catalog tables), and to plan the workload queries.

DAD-I runs the INGRES line-oriented terminal monitor (see [ULT90], Section 3) as a separate process. For our tests, DAD-I and INGRES were on separate machines. To get INGRES's cost estimates for queries on a feature set, DAD-I first ensures that INGRES's data dictionary (catalog tables) reflect the feature set to be evaluated. This can require creating or dropping both secondary and primary indexes.[1] It can also involve altering the data statistics associated with

[1]Primary indexes are created and dropped with the INGRES `modify` statement.

tables or secondary indexes, because different organizations of the same logical data have different storage requirements.

For example, a `heap` organization uses little space above that needed for the tuples themselves—they are packed as tightly as possible in each page. On the other hand, a `btree` organization requires much more space—for the internal nodes in the index, for the leaf pages (it is dense), and for free space in the data pages. These estimates of storage utilization are crucial not because of the storage cost per se, but because they determine INGRES's estimates of the number of disk accesses needed for various query plans.

Once DAD-I has arranged for the data dictionary to correspond to the feature set to be evaluated, it sends queries, one by one, to INGRES and reads back, for each query, INGRES's query plan and cost estimate. For some queries and feature sets DAD-I must transform the query. For example, if a feature set contains a vertical partitioning, queries on the partitioned table may involve a join of the two halves of the vertical partitioning.

After receiving a query from DAD-I, INGRES generates the query plan and cost estimate (because DAD-I has executed INGRES's `set qep` statement). INGRES does not actually execute the query because DAD-I has executed INGRES's `set noqueryrun` statement.

INGRES's interface for obtaining query plans and cost estimates is really designed to be used by database administrators, not by physical design software. As a result, it would be too difficult to make DAD-I capable of understanding the query plans INGRES presents. But it is usually simple to extract INGRES's cost estimate. However, there are two situations in which it is not possible to get INGRES's cost estimate.

1. When the query is not a join *and* there is no secondary index that INGRES can possibly use in computing the query:

   In this case INGRES does not use its so-called JOINOP query processor [RS86], and consequently produces no cost estimate or query plan. DAD-I recognizes situations in which INGRES does not use JOINOP, and in these situations DAD-I produces the cost estimate. (The query plan in these cases is obvious.)

2. When INGRES, in a stereotypical way, executes a query by performing several suboperations, for each of which the query optimizer produces a plan:

   Queries with a `group by` clause are an example. If we were to use `set noqueryrun` to prevent INGRES from executing the workload queries, INGRES would produce a plan only for the first suboperation. If INGRES did execute the query, a crash would likely result, because DAD-I has updated the catalog tables in ways that don't correspond to the actual state of the database. For this reason, DAD-I cannot work with queries for which INGRES produces multiple plans.

In addition, INGRES's cost estimates to not include the cost of database writes in `update`, `insert`, and `delete` queries; DAD-I estimates these.

We constructed DAD-I to support extension and retargetability. Dependencies on INGRES arise in the design of features, in the feature-generation procedures, and, of course, in the code that communicates with INGRES. It would be fairly easy to add new kinds of features, especially if they didn't involve query transformations. To retarget to another relational database also would be straightforward, provided there is some way to get query costs; the feature-generation procedures could be easily adapted. A rule-based implementation of feature generation would probably make retargetability easier, but might also be slower.

6.2 Description of the Tests

We used DAD-I to determine physical database designs for three physical design problems (besides the example of Figures 1.1 and 1.2).

6.2.1 Test Problem 1

Test Problem 1 is an adaptation of a physical database design problem given in [FST88]. (The differences are in queries 5, 6, and 9, which are multi-tuple updates in the original workload,[2] and in query 2, which is a `group by` query in the original workload.[3]) The schema and workload for this test are shown in Appendix B.1.

Some properties of the data, notably selectivities and logical keys, were not reported in [FST88]. For such situations we invented properties consistent with the characteristics specified in [FST88].

6.2.2 Test Problems 2 and 3

Test Problem 2 is based on the AS^{3}AP benchmark [TOB91]. We used the AS^{3}AP schema and most of the queries from the AS^{3}AP "single-user" test—a test designed

[2] In the case of the multi-tuple updates, we wanted to avoid having to implement the cost estimating procedures for multi-tuple updates in INGRES. They could be implemented along the lines described in [ST85].

[3]Section 6.1 describes the difficulty with `group by` queries.

to measure how well a DBMS deals with individual queries, as opposed to how well it deals with contention for data among concurrent transactions. We chose these queries as a workload because they provide a reasonably rich and complete set of operations on a reasonably complex logical schema.

From the AS^{3}AP single-user tests we omitted only

- queries designed to test "operational issues", such as bulk loading and index building,
- queries designed to test the relative efficiency of output to screen, file, and table,
- updates designed to test a DBMS's ability to detect violation of uniqueness and inclusion (foreign key) dependencies,
- multi-tuple updates (see footnote 2), and
- two group-by queries (for the reason discussed in Section 6.1).

This left 30 queries and updates, comprised of

- eight single-table selections,
- eight joins,
- two projections that, by using `select distinct`, yield far fewer tuples than contained in the input table,
- four aggregates, and
- eight updates involving single tuples.

In Test Problem 2, we arbitrarily assigned a frequency of one to all 30 queries. In Test Problem 3, frequencies of queries that involve writes are increased (Appendix B.3 provides the frequencies.)

6.2.3 Cost Coefficients for Tests

Based on comparisons of optimizer estimates with actual CPU resources used in queries, we estimate that each unit of CPU cost in INGRES (a "C") corresponds roughly to 100ms of CPU time on a 6MIPS machine. New York University's Academic Computing Facility (NYU ACF) charges \$75.00 for one hour of CPU on the test machine, and assuming a 5-year system horizon, this makes the cost of one "C" per hour \$91.25. NYU ACF charges \$.0001 per DRA, making the cost of 1 DRA per hour \$4.380.

Finally, although NYU ACF does not charge directly for storage space (but does limit the amount available to each user), we estimate the cost per 2kbyte page by assuming that we can buy and maintain for 5 years a 1Gbyte disk at a cost of \$20,000. This yields a cost per page of approximately \$.03815. Of course, as discussed in Section 3.1, DAD-I is parameterized so that, for a particular system under design, database administrators can specify the coefficients for "C" per hour, DRA per hour, and storage pages.

6.3 Experimental Results and Discussion

6.3.1 Test Problem 1

The lowest-cost physical design found by DAD-I consists of the following features (the integer feature labels are those generated by DAD-I)

Label	Feature
1	*idx*(1, `hash`, `quotes`, `partno`)
10	*idx*(2, `ISAM`, `quotes`, `maxqty`)
17	*idx*(1, `hash`, `parts`, `partno`)
23	*idx*(2, `btree`, `orders`, `date,orderno,partno,qty,suppno`)
29	*idx*(2, `ISAM`, `quotes`, `suppno,partno,price`)
33	*idx*(1, `btree`, `orders`, `orderno`)

with estimated workload cost 1234.275 "C", 8389.0 DRA, 5507.0 pages. The associated query plans are in Appendix A.3. The values for parameters supplied by the database administrator are: $max\text{-}best_I = max\text{-}best_{II} = 8$, $num\text{-}tries = 3$ for Search I, and $num\text{-}tries = 2$ for Search II (with the initial try starting at *ideal*).

Discussion

Judging from the query plans, this design is good, except for the inclusion of *idx*(2, `ISAM`, `quotes`, `maxqty`). This anomaly is due to the fact that (i) INGRES produces a lower cost estimate for query 0 than DAD-I and (ii) if this index is present, INGRES supplies the estimate. As discussed in Section 6.3.3, estimate accuracy it the critical limiting factor in the quality of DAD-I's designs; Section 8.2 also discusses how estimate accuracy could be improved.

6.3.2 Test Problems 2 and 3

The lowest-cost physical design found by DAD-I for Test Problem 2 is

Label	Feature
1	idx(1, `btree`, `updates`, `key`)
5	idx(2, `btree`, `updates`, `code`)
7	idx(1, `ISAM`, `uniques`, `key`)
11	idx(2, `btree`, `updates`, `int`)
17	idx(2, `ISAM`, `tenpct`, `signed`)
23	idx(1, `hash`, `hundred`, `key`)
48	idx(2, `hash`, `uniques`, `address`)
55	idx(1, `ISAM`, `tenpct`, `key`)
58	idx(2, `ISAM`, `uniques`, `code,date,signed`)
60	idx(2, `ISAM`, `hundred`, `code,date,signed`)
78	idx(2, `ISAM`, `tenpct`, `code,date`)
98	idx(2, `ISAM`, `tenpct`, `name,int`)
105	idx(2, `btree`, `updates`, `decim`)
198	vp(`updates`, {{`address`, `double`, `fill`, `float`, `key`, `name`}, {`code`, `date`, `decim`, `int`, `key`, `signed`}})
213	$materialize(Q_{18})$

with cost 484.030 "C", 16652.421 DRA, 22521.0 pages. (The values for parameters supplied by the database administrator in Test Problems 2 and 3 are as for Test Problem 1, with the exception that *num-tries* = 1 for Search II, with the single try begun from the current *ideal*.)

The lowest-cost physical design found by DAD-I for Test Problem 3 is

Label	Feature
0	*idx*(2, `btree`, `updates`, `key,code,double,int,name,signed`)
7	*idx*(1, `ISAM`, `uniques`, `key`)
24	*idx*(1, `ISAM`, `hundred`, `key`)
55	*idx*(1, `ISAM`, `tenpct`, `key`)
58	*idx*(2, `ISAM`, `uniques`, `code,date,signed`)
60	*idx*(2, `ISAM`, `hundred`, `code,date,signed`)
78	*idx*(2, `ISAM`, `tenpct`, `code,date`)
105	*idx*(2, `btree`, `updates`, `decim`)
213	*materialize*(Q_{18})

with cost 694.245 "C", 31653.789 DRA, 17219.0 pages.

Discussion

The large number of secondary indexes in the design for Test Problem 2 may seem suspect to experienced database designers. However, we believe their inclusion results from the fact that all queries have the same frequency in Test Problem 2. By contrast, in the "typical" application, complex queries (multi-table joins and decision-support queries) are much less frequent than point updates and queries involving one or two tables. As can be seen in the results of Test Problem 3, increasing the frequency of updates (and of a few two-table joins) results in a design with far fewer secondary indexes on `updates`, and with no vertical partitioning of `updates`.

6.3.3 Estimate Accuracy

Errors in the query optimizer's cost estimates appear to be the critical limiting factor in the quality of DAD-I's results. These errors required us to force some secondary indexes to conflict when they otherwise would not have had to (see Section 3.3). These errors also led to the anomalous inclusion of a secondary index in the result for Test Problem 1. Furthermore, in one query that we tested, INGRES's estimate of disk-accesses was 405, and the actual number was 16; the optimizer's estimate was off by a factor of 25. Query plan stability may also be an issue: for large join queries the query optimizer might not always produce the same plan.

Even though DAD-I is a competent assistant, human oversight is still required. On the other hand, design using such an assistant would be much easier than manual design. In a system where plans, in addition to costs, could be obtained from the query optimizer, a system like DAD-I could provide its own estimates where needed. As observed in [FST88], it doesn't make sense to try to second guess the optimizer's plans, but, unlike [FST88], we think it might make sense to second guess the optimizer's cost estimates. Section 8.2 discusses how, in the longer term, estimate accuracy could be improved.

Chapter 7

Related Work

We categorize software systems for physical database design in a number of dimensions:

- What "linguistic levels" does the design span? Does it include aspects of logical as well as physical design?
- How big is the solution space?
- How much does the system rely on inspection methods to generate the solution space?
- Does the system rely on explicit search among a solution space using a cost-based objective function? If not, how are various criteria applied to produce a design? If so, does the query optimizer produce the plans and estimates?
- How much human mediation is required and allowed? Can the database administrator constrain the designer to a partial solution?

- Is the design process limited to achieving performance goals, or are other possible design objectives represented?

We next discuss several systems in detail.

7.1 FCDS

The authors of [CMD83, DJCM89] do not name their system but we will call it FCDS (for **F**orm, **C**onvert, **D**esign, and **S**elect, the four main phases of the design process in their system). We share FCDS's broad objectives: FCDS is "a tool to support a DBA [database administrator] in the task of physical database design. [It] facilitates the explicit specification of the design problem and greatly expands the number of design alternatives which can be considered for a particular design problem ([CMD83], pg. 223)." Like FCDS's designers, we think this is best approached using a balance of knowledge- and search-based components.

A key difference between DAD-I and FCDS is that FCDS was begun in the context of CODASYL databases, with navigational query processing. Although [DJCM89] mentions the possibility of adapting FCDS to systems with relational-style query optimization, apparently this has not been done.

FCDS operates as follows. Its inputs are

- "LDS" (Logical Data Structure)—a conceptual-level schema[1] in an E/R-like model [Che76] (all relations are binary and 1-1 or 1-n; no attributes are allowed on relations). The LDS also provides entity cardinality, relationship degree, and domain cardinality (of attributes).

[1]This "Logical Data Structure", corresponds to what would more commonly by called "conceptual" in the context of logical design for relational databases.

- Workload—this is navigational, though expressed in an SQL-like notation, and also contains query frequency and selectivity information.

- Hardware Environment Description—"secondary-memory access time (random and sequential), data transfer rate, maximum blocksize, cost of CPU and retrieval time, and cost of storage space."([DJCM89], pg. 74) These parameters are used to develop the objective function.

In addition, there is a degree of retargetability in terms of the solution space, which can be adapted to model the implementation of a particular DBMS.

The outputs of FCDS are

- physical record structures (including vertical-partitionings, data item duplication, and horizontal partitions), and

- primary and secondary access paths (in the CODASYL sense, so this includes heap, hashed, ISAM, pointer, and repeating-field structures.)

In [CMD83, DJCM89] the authors recognize that the problem is so hard that only a heuristic approach is possible. In [CMD83] the authors take a "DSS" (Decisions Support System) approach to the problem, and rely heavily on human guidance—that is, the authors conceive of FCDS as a decision support system for the database administrator's physical design decisions. A contribution of [DJCM89] is to replace much of that human guidance with a rule-based system. This system employs backward-chaining rules, with classical "certainty factors" (see e.g. [LS93], pages 329–331) to determine confidence in its conclusions. The report [Dab89]) offers a detailed description of the rules and their operation.

FCDS uses four phases to produce a physical database design:

1. FORM uses the rule-base system discussed in [DJCM89] to enumerate some of the possible representations of the LDS in terms of record types and relationship representations, (e.g. direct pointers or nesting).

2. CONVERT transforms operations on LDS to operations on the record types and relationship representations produced by FORM.

3. DESIGN selects a file organization for each record type, using

   (a) a detailed cost model, and

   (b) algorithms for vertical partitioning and for selecting access paths, given a record type and a (fixed) access pattern.

   The algorithms can use classical mathematical programming techniques, such as integer linear programming, because there is no query optimization here, and the system is dealing with accesses on single files. The algorithms for vertical partitioning and access path selection are separate, even though the problems are *not* separable. Therefore DESIGN iterates once. The database administrator can also specify partial solutions.

4. SELECT chooses the lowest-cost set of file organizations implementing the particular set of record types. It can also display the set of solutions whose cost is within a specified percentage of the lowest cost.

Besides the fact that FCDS works in the context of the CODASYL model, a salient difference between DAD-I and FCDS is that FCDS operates from a conceptual design. Given that there are many systems and widely accepted methodologies to aid in logical design for relational databases, we decided that DAD-I's input should be a logical design. To some extent, of course, logical design can affect performance, and in some cases DAD-I changes the logical design if the consequence is a lower-cost physical design. Philosophically, however, we take the view that DAD-I does this simply to improve an incomplete separation of logical from physical design—we think that, for example, view materialization would be a good addition to the physical design options of relational DBMS.

Additional information on FCDS appears in [Mar83, CM83, MC85, MC87].

7.2 Relational Design Tool

Relational Design Tool for SQL/DS (RDT) is a system for index selection [FST88, IBM85] for IBM's SQL/DS [IBM], and appears to be still available from IBM. RDT is based on a prototype, DBDSGN, for System R [Ast76]. In many ways, DAD-I is an attempt to

- enlarge the solution space available in RDT (which covers index selection—single-column indexes plus those multi-columns indexes that the database administrator explicitly adds to the solution space),
- generalize from RDT's notion of "plausible columns" (discussed below), and
- construct a more-easily retargetable architecture.

The inputs to RDT are a schema and workload with statistics, much as for DAD-I. RDT presents the workload queries to the SQL/DS optimizer, and using the `EXPLAIN REFERENCE` statement, obtains for each query a set of "plausible columns", i.e. columns that the optimizer considers "plausible for indexing". Then RDT computes a set of "atomic configurations", a kind of basis set of indexing designs, from which the cost of all other designs can be computed. The ability to use atomic configurations to calculate the cost of all indexing designs relies on

1. the optimizer actually finding the lowest-cost plan, and

2. the fact that the optimizer uses at most one index for each correlation name in a query (because SQL/DS does not use TID intersection).

The advantage of using atomic configurations is that, as a function of the number of plausible columns, there are asymptotically fewer atomic configurations than indexing designs in general, for workloads seen in practice.

RDT then gets from the SQL/DS query optimizer the cost for each atomic configuration using the `EXPLAIN COST` command. At this point, RDT, at the database administrator's option, may use heuristics to discard some possible indexes. Finally, RDT searches among indexing designs involving the remaining indexes. At the database administrator's option, RDT can use additional heuristics to avoid considering all such designs. In addition, if so instructed, RDT, instead of having SQL/DS evaluate the cost of all atomic configurations, can have SQL/DS evaluate query costs as needed during the search.

In the process of creating a more general design system than RDT we have lost the ability to use atomic configurations: INGRES, unlike SQL/DS, performs TID intersection, so such an approach using atomic configurations is invalid for INGRES. Also, DAD-I's more general search methods mean that it is not possible to minimize database catalog updates as in RDT. In abandoning reliance on atomic configurations we had little choice (because INGRES *does* use TID intersection), and we are confident that the larger solution space and more retargetable architecture of DAD-I are worth the attendant reduction in efficiency.

7.3 Knowledge-Based Approaches

RdbExpert is a DEC product that uses a knowledge-based approach to create physical designs for VAX Rdb/VMS databases [DEC92, DEC]. In another knowledge-based approach, [CBC93] describes how to use the Dempster-Schafer theory of evidence [Sha76] to assign a measure of "promise" to alternative physical database designs. An even earlier work, [BN87], apparently also took a knowledge-based approach.

7.3.1 RdbExpert

According to [DEC92], RdbExpert's core is a "knowledge base (KB) of Rdb/VMS physical design expertise" (pg. 1-3) that RdbExpert uses—given an application-execution environment, workload, schema, and statistics—to produce a physical database design. This knowledge base was originally prototyped in OPS5, and later reimplemented using C and SQL [Gio91].

In addition, RdbExpert helps database administrators organize the physical and logical design process (it interfaces with logical design software), and produces the necessary procedures to create the generated design. Input information can come from an implemented database or from SQL statements; database administrators can create data statistics (for not-yet-implemented systems), or can have them gathered from running applications by trace utilities. RdbExpert also offers a language for specifying physical database design problems.

For each logical schema, workload, and associated statistics, RdbExpert produces a physical design. This can include a summary or detailed rationale for the design.

Neither [Gio91] nor [DEC92] reveals any concrete specifics of RdbExpert's internals. However, one difference between DAD-I and RdbExpert is that, unlike RdbExpert, DAD-I considers design strategies—aggregate materialization and vertical partitionings—that involve changes to the logical schema. Also, it is reportedly difficult to extend RdbExpert to take into account new features, and it is tied to Rdb/VMS.

7.3.2 Using Dempster-Schafer Theory

As we suggest at the beginning of this chapter, most approaches to physical database design involve some mixture of a knowledge-based component and a search component. One way to make the search cheaper is to rely more heavily on the knowledge-based component, and [CBC93] proposes a very strong knowledge-based component to do much of the work done in DAD-I's search phase.

The approach is to use the Dempster-Schafer theory of evidence to assign a *belief* value to each design, the magnitude of which is, intuitively, a measure of confidence that that design is good. For example, if there is a query, Q, with an equality predicate on column c of table R and a query, Q', with a range predicate on column c of table R, then there might be one rule that proposes (for Q) a `hash` index on R, and two rules (one for Q and one for Q') that propose an `ISAM` index on R. (An `ISAM` index would be proposed for each query, because `ISAM` indexes are good for both range and equality predicates—though inferior to `hash` indexes for equality predicates.) Using Dempster-Schafer theory, these indexing proposals can be combined to say that either a `hash` index or an `ISAM` index is a good idea, with a belief value for each possibility determined by a weighting of the original rules. So in this example, the belief value for an organization with the `ISAM` index might be higher, assuming all of the original rules had equal weight.

After designs and associated belief values have been generated, the tool would request (from the query optimizer) the cost estimates for the workload on some of the physical designs with the highest belief values. The report [Wal90] offers a set of rules that might be used in this approach for physical database design for INGRES (though without belief values).

There is continuum in the reliance on knowledge-based and search-based approaches. Probably only experimentation with actual physical design tools will tell us where on this continuum a physical design tool performs best both in terms of efficiency and in terms of the quality of the output design.

7.4 Other Related Work

Other work takes a knowledge-based approach to the larger problem of information system development, including the translation of a conceptual schema to a logical schema [JMSV92]. The problem of physical database design arises as an instance of the problem of satisfying non-functional requirements. For example, [Nix91, Nix93] discuss the issue of performance as a non-functional requirement, and [MCN92] describes the general framework of goal "satisficing" used to develop implementations that are likely to satisfy their non-functional requirements.

Physical database design also arises in transformational approaches to implementing database programming languages, as in [Feg91], which describes an implementation technique for the functional database programming language ADABTPL.

Finally, there has been some work on main memory data structure design: [RK77, Low78, SSS81, KZG81]. Some of this work is similar to DAD-I, in that it involves the selection of representation of logical operations on an abstract data type (e.g. set or list) from a fixed repertoire of implementations, based on uses of an instance of the abstract data type.

Chapter 8

Conclusion and Future Work

8.1 Conclusion

This thesis has accomplished the following:

- We successfully generalized the approach pioneered by [FST88], in which a design assistant uses a query optimizer's cost estimates to find low-cost physical designs.

- We devised a framework—consisting of feature-sets and the separation of feature generation from search—that allows us to use generic search algorithms, and that supports retargetability of design assistants to different DBMSs, and even to $\neg$1NF or object-oriented DBMS.

- We designed and implemented a heuristic search algorithm that involves (i) finding the best features for individual queries, and then (ii) combining the best features for individual queries to find a feature set with a low cost for the workload as a whole. This heuristic can manage the complexity of realistically sized physical design problems, and still produce low-cost designs.

- We designed and implemented procedures to generate features for the INGRES relational DBMS. These procedures are applicable or easily adaptable to other relational DBMSs. Our experiments with INGRES show that these procedures are fast and effective.

- The implementation of our prototype design assistant, DAD-I, isolates dependencies on the specifics of INGRES. This substantiates the claim that our framework could accommodate many relational and post-relational DBMSs.

- We experimented on several physical design problems using DAD-I. The experimental results support the conjecture that a design assistant based on our framework can produce good designs, and that design quality is limited not by the expense of search but by the accuracy of the query optimizer's estimates. In Section 8.2 we propose several approaches to improving estimate accuracy.

8.2 Future Work

A closer integration between the design assistant and query optimizer could lead to a major improvement in the quality of the output designs and to faster search:

- When producing designs for an existing, off-the-shelf DBMS, the assistant could produce better designs if it could examine queries and override the optimizer's cost estimate when necessary.

- In a (hypothetical) DBMS in which the design assistant and query optimizer are tightly integrated, the database administrator could provide additional information to the optimizer/designer to allow it to improve the accuracy of query cost estimates. For example, the cardinality of intermediate join results might be useful, given the difficulty of estimating these [IC91]. The optimizer could then spend more time on compile-time optimization of queries when the database administrator has supplied additional statistics.

- If the design assistant and query optimizer were tightly coupled it might be possible to pass query plan information sideways among different physical designs, in a way analogous to sideways information passing in parametric query optimization [INSS92].

We think a design assistant that (like DAD-I) relies on both knowledge- and search-based approaches can produce designs superior to those that can be produced by an assistant without an explicit, cost-based, search. DAD-I's knowledge-based component, feature generation, is implemented procedurally, with the result that feature generation is fast. However, we might be able to reduce the complexity of the search with a rule-based implementation of feature generation. The rules might be able to exclude some features or feature sets from consideration, or even assign some measure of promise to feature sets (as proposed in [CBC93]). Using rule-based feature generation might also ease the task of retargeting to other DBMSs. We do not know how much a physical design assistant should rely on a smart, precise feature generation as opposed to explicit search. We hope to investigate this question in a successor to DAD-I.

Finally, we would also like to see if the feature-set framework could be used to select main-memory data structures for very-high-level languages such as SETL [SDDS86], Griffin [New93], or Bulk [RS91b].

Appendix A

Query Plans

The following query plans were produced by INGRES, and have been edited to improve readability.

A.1 A Secondary Index Join With Excessively Low Estimate

This is a query plan for query 8 in Test Problem 1 (see Appendix B.1):

```
                            Join(suppno)(C0)
                            Sorted(partno)
                            Pages 6 Tups 58
                            D18 C11
                           /          \
                    Join(partno)      iddx47
                    Sorted(partno)    Isam(suppno)
                    Pages 1 Tups 1    Pages 444 Tups 72000
                    D10 C1
                   /             \
            Sort(partno)          iddt63_1
            Pages 1 Tups          Hashed(partno)
            D9 C1                 Pages 1000 Tups 8000
               |
         Join(date)(C0)
         Sorted(date)
         Pages 1 Tups 1
         D9 C1
        /             \
Proj-rest              Proj-rest
Sorted(date)           Sorted(date)
Pages 1 Tups 60        Pages 1 Tups 60
D2 C0                  D1 C0
   |                       |
iddx23                 iddx54
B-Tree(date)           B-Tree(date)
Pages 419 Tups 24000   Pages 354 Tups 24000
```

The low estimate is the number of tuples at the node headed `Join(date)(C0)`; the estimate should be 60 rather than 1, because the tuples selected from `iddx23` and `iddx54` have the same value in the `date` column. (Each of `iddx23` and `iddx54` is a secondary index whose first column is `orders.date`.)

A.2 Example Problem

Costs and plans for all queries except Q_3 are provided by DAD-I.

Q_0 The query plan is trivial; it is only necessary to insert the new tuple into `orders`. (This would involve updating and the primary and secondary indexes on `orders`, as well as updating a data page.)

Q_1 The query plan is a scan of the one-tuple table containing the materialized aggregate query.

Q_2 The query plan is to use feature 2, $idx(1, \texttt{btree}, \texttt{orders}, \texttt{ono})$ to find any tuple satisfying the equality predicate on `ono`.

Q_3

```
                      Join(pno)

                      Sorted(pno)

                      Pages 8334 Tups 83333

                      D848 C480

                     /              \

           Join(pno)                  Proj-rest

           Sorted(pno)                Sorted(pno)

           Pages 1112 Tups 10000      Pages 121 Tups 10000

           D657 C180                  D186 C0

          /         \                      |

Proj-rest            Proj-rest            iddx9

Sorted(pno)          Sorted(pno)          B-Tree(pno)

Pages 400 Tups 4000  Pages 91 Tups 10000  Pages 186 Tups 10000

D511 C0              D141 C0

     |                    |

parts                iddx14

B-Tree(pno)          B-Tree(pno)

Pages 511 Tups 4000  Pages 141 Tups 10000
```

A.3 Test Problem 1

Plans produced by INGRES, edited to improve readability.

Q_0

```
Proj-rest
Sorted(partno)
Pages 1 Tups 0
D0 C0
   |
quotes
Hashed(partno)
Pages 1637 Tups 72000
```

Q_1

```
            Join(partno)
            Sorted(partno)
            Pages 1 Tups 1
            D3 C0
           /           \
Sort(partno)            parts
Pages 1 Tups 1          Hashed(partno)
D2 C0                   Pages 1143 Tups 8000
   |
Proj-rest
Sorted(date)
Pages 1 Tups 1
D2 C0
   |
iddx23
B-Tree(date)
Pages 419 Tups 24000
```

Q_2
```
Proj-rest
Sorted(partno)
Pages 1 Tups 0
D0 C0
   |
quotes
Hashed(partno)
Pages 1637 Tups 72000
```

Q_3 (Estimate produced by DAD-I.)

Q_4
```
Sort()
Pages 1 Tups 2
D1 C1
   |
Proj-rest
Sorted(orderno)
Pages 1 Tups 2
D1 C0
   |
orders
B-Tree(orderno)
Pages 1683 Tups 24000
```

Q_5
```
Proj-rest
Sorted(partno)
Pages 1 Tups 0
D0 C0
   |
quotes
Hashed(partno)
Pages 1637 Tups 72000
```

Q_6 (Estimate produced by DAD-I.)

Q_7
```
        Join(partno)
        Sorted(suppno)
        Pages 72 Tups 720
        D727 C57
      /                \
Proj-rest                parts
Sorted(suppno)           Hashed(partno)
Pages 4 Tups 720         Pages 1143 Tups 8000
D7 C0
   |
iddx29
Isam(suppno)
Pages 606 Tups 72000
```

Q_8

```
                          Join(suppno)(C0)
                          Sorted(suppno)
                          Pages 187 Tups 1860
                          D377 C327
                         /            \
            Sort(suppno)            iddx29
            Pages 6 Tups 60         Isam(suppno)
            D62 C5                  Pages 606 Tups 72000
                 |
            Join(partno)
            Sorted(date)
            Pages 6 Tups 60
            D62 C4
           /             \
Proj-rest                 parts
Sorted(date)              Hashed(partno)
Pages 1 Tups 60           Pages 1143 Tups 8000
D2 C0
   |
iddx23
B-Tree(date)
Pages 419 Tups 24000
```

Q_9

```
                    Join(partno)(C0)

                    Sorted(partno)

                    Pages 1 Tups 1

                    D2 C0

                    /            \

Sort(partno)                 parts

Pages 1 Tups 1               Hashed(partno)

D1 C0                        Pages 1143 Tups 8000

   |

Proj-rest

Sorted(orderno)

Pages 1 Tups 1

D1 C0

   |

orders

B-Tree(orderno)

Pages 1683 Tups 24000
```

Appendix B

Test Problems

B.1 Test Problem 1: Schema and Workload

B.1.1 Test Problem 1: Schema

This schema is similar to that in Figure 1.1, but there are some important differences in data statistics, and in schema of `orders`. The logical schema, but not the key information and data statistics, are from [FST88].

`parts` #tuples=8000

column	<u>partno</u>	qonhand	descrip
type	integer	integer	char(184)
# values	8000	4000	8000
min, max	1, 8000	1, 8000	`"0"`, `"Z..."`

`orders` #tuples=24000

column	<u>orderno</u>	<u>partno</u>	<u>suppno</u>	date	qty	oinfo
type	char(6)	integer	char(3)	integer	integer	char(71)
# values	12000	8000	100	400	12000	24000
min	`"0"`	1	`"AAA"`	19850101	1	`"0"`
max	`"Z..."`	8000	`"ZZZ"`	19930101	1000000	`"Z..."`

`quotes` #tuples=72000

column	<u>suppno</u>	<u>partno</u>	<u>minqty</u>	<u>maxqty</u>	price	remarks
type	char(3)	integer	integer	integer	money	char(15)
# values	100	8000	4000	4000	32000	72000
min	`"0"`	1	1	1	0.10	`"0"`
max	`"ZZZ"`	80000	1000000	1000000	1000.00	`"Z..."`

The logical keys of `quotes` are

$$\{\texttt{suppno}, \texttt{partno}, \texttt{minqty}\}$$

and

$$\{\texttt{suppno}, \texttt{partno}, \texttt{maxqty}\}$$

Both `parts` and `orders` have one key each, comprising the underlined columns.

B.1.2 Test Problem 1: Workload

Test Problem 1: Workload

j	ϕ_j $(/hour)$	Q_j

0	5	`select quotes.suppno,quotes.price from quotes`
		`where quotes.partno=:pno`
		`      and quotes.minqty<1000`
		`      and quotes.maxqty>2000`

1	5	`select orders.orderno,orders.partno,`
		`       parts.descrip,orders.date,orders.qty`
		`from orders,parts`
		`where orders.date<=831216`
		`      and orders.date>=830000`
		`      and orders.suppno=:sno`
		`      and orders.partno=parts.partno`

2	5	`select min(quotes.price),max(quotes.price)`
		`from quotes`
		`where quotes.partno=:pno and quotes.suppno=:suppno`

3	20	`insert`
		`into orders (orderno,partno,suppno,date,qty,oinfo)`
		`values (:ono,:pno,:sno,:date,:qty,:oinfo)`

4	10	`select distinct orders.partno,orders.qty from orders`
		`where orders.orderno=:ono`
		`order by qty`

Test Problem 1: Workload (cont.)

j	ϕ_j $(/hour)$	Q_j
5	20	`update quotes set price=:price where quotes.suppno=:sno and quotes.partno=:partno and quotes.minqty=:minqty`
6	20	`delete from orders where orders.suppno=:sno and orders.orderno=:orderno and orders.partno=:partno`
7	10	`select parts.partno,parts.descrip,quotes.price from parts,quotes where quotes.suppno=:sno and parts.partno=quotes.partno`
8	2	`select orders.suppno,orders.orderno, parts.partno,parts.descrip from parts,orders,quotes where orders.date=:date and quotes.price<:price and orders.suppno=quotes.suppno and parts.partno=orders.partno`

Test Problem 1: Workload (cont.)

j	ϕ_j $(/hour)$	Q_j
9	5	`delete from orders` `where orders.date>:date` `and orders.orderno=:orderno` `and orders.suppno=:suppno` `and orders.partno` `in (select parts.partno from parts` `where parts.qonhand>:qoh)`

Within this workload, each of queries 0, 1, 3, 4, 7, and 8 is semantically equivalent to the corresponding query in [FST88].

B.2 Test Problem 2: Schema and Workload

We used the AS^{3}AP schema and data statistics as presented in [TOB91], Table 4.3 (pg. 178), modified only to make all data distributions uniform. This we did because

- DAD-I cannot provide non-uniform data statistics to INGRES version 5.0, and
- only three columns in the workload (`uniques.float`, `uniques.double`, and `updates.float`) have non-uniform distributions, and none of these is a selection or join column in the tests.

In the workload we included from [TOB91], Appendix 2 (pages 196–202), the queries with the following identifiers:

0. `sel_1_cl`
1. `sel_1_ncl`
2. `sel_10pct_cl`
3. `sel_100_cl`
4. `sel_100_ncl`
5. `sel_10pct_ncl`
6. `variable_select` (first)
7. `variable_select` (second)
8. `join_2_cl`
9. `join_2_ncl`

10. `join_2`
11. `join_3_cl`
12. `join_3_ncl`
13. `join_4_cl`
14. `join_4_ncl`
15. `join_1_1_pct`
16. `proj_100`
17. `proj_10pct`
18. `scal_agg`
19. `info_retrieval`
20. `simple_report`
21. `total_report`
22. `app_t_mid`
23. `mod_t_mid`
24. `del_t_mid`
25. `app_t_end`
26. `mod_t_end`
27. `del_t_end`
28. `mod_t_int`
29. `mod_t_cod`

B.3 Test Problem 3: Frequencies

The queries with frequency $\neq$ 1/hour are:

AS^{3}AP Query	Frequency (/hour)
`join_2_cl`	2
`join_2_ncl`	2
`join_2`	2
`app_t_mid`	20
`mod_t_mid`	20
`del_t_mid`	20
`del_t_end`	20
`mod_t_int`	20
`mod_t_cod`	20

Bibliography

[Ast76] M. M. Astrahan *et al.* System R: Relational approach to database management. *ACM Trans. Database Syst.*, 1(2), June 1976.

[BBG$^+$90] Don S. Batory, J. R. Barnett, J. F. Garza, K. P. Smith, K. Tsukuda, B. C. Twichell, and T. E. Wise. GENESIS: An extensible database management system. *IEEE Trans. Software Engineering*, 14(11):1711–1730, November 1990.

[BN87] Michael L. Brodie and S. Nesson. Physical design advisor (PDA): An expert system design aid for the physical design of Model 204 databases. Technical report, Computer Corporation of America, March 1987.

[Car75] Alfonso F. Cárdenas. Analysis and performance of inverted data base structures. *Communications of the ACM*, 18(5), May 1975.

[CBC93] Sunil Choenni, Henk M. Blanken, and Thiel Chang. On the automation of physical database design. In *Symposium on Applied Computing*, February 1993.

[Cha76] D. D. Chamberlin *et al.* SEQUEL 2: a unified approach to data definition, manipulation, and control. *IBM J. Research and Development*, 20(6):560–575, 1976.

[Che76] P. P. Chen. The entity-relationship model: Toward a unified view of data. *ACM Trans. Database Syst.*, 1(1):1–36, January 1976.

[CM83] John V. Carlis and Salvatore T. March. A computer-aided physical database design methodology. *Computer Performance*, 4(4):198–214, December 1983.

[CMD83] John V. Carlis, Salvatore T. March, and Gary W. Dickson. Physical database design: A DSS approach. *Info. & Management*, 6:211–224, 1983.

[COD71] CODASYL Data Base Task Group April 71 Report. ACM, New York, 1971.

[Com78] D. Comer. The difficulty of optimum index selection. *ACM Trans. Database Syst.*, 3(4):440–445, December 1978.

[CW91] Stefano Ceri and Jennifer Widom. Deriving production rules for incremental view maintenance. In *Proceedings of the Seventeenth International Conference on Very Large Databases*, pages 577–589, 1991.

[Dab89] Christopher E. Dabrowski. A detailed description of the knowledge-based system for physical database design. Technical Report NISTIR

89-4139 (volumes 1 and 2), National Institute of Standards and Technology, National Computer Systems Laboratory, Information Systems Engineering Division, Gaithersburg MD 20899, August 1989.

[Dat87] C. J. Date. *A Guide to INGRES.* Addison-Wesley, 1987.

[DEC] Digital Equipment Corporation. *VAX Rdb/VMS Reference Manual.*

[DEC92] Digital Equipment Corporation. *DEC RdbExpert for VMS*, April 1992. Manual Number AA-LE46B-TE for DEC RdbExpert for VMS Version 2.0.

[DGLS79] Robert B. K. Dewar, Arthur Grand, Ssu-Cheng Liu, and Jacob T. Schwartz. Programming by refinement, as exemplified by the SETL representation sublanguage. *ACM Trans. Prog. Lang. and Syst.*, 1(1):27–49, July 1979.

[DJCM89] Christopher E. Dabrowski, David K. Jefferson, John V. Carlis, and Salvatore T. March. Integrating a knowledge-based component into a physical database design system. *Info. & Management*, pages 71–86, 1989.

[EHR80] W. Effelsberg, T. Härder, and A. Reuter. An experiment in learning DBTG database administration. *Information Systems*, 5(2):137–147, 1980.

[Feg91] Leonidas Fegaras. Using type transformation in database system implementation. In *Proceedings of the Third International Workshop* *on Database Programming Languages*, pages 337–353. Morgan Kaufmann, August 1991.

[Fre87] Johann Christoph Freytag. A rule-based view of query optimization. In *SIGMOD'87 Proceedings*, pages 173–180, May 1987.

[FST88] S. Finkelstein, M. Schkolnick, and P. Tiberio. Physical database design for relational databases. *ACM Trans. Database Syst.*, 13(1):91–128, March 1988.

[GD87] Goetz Graefe and David J. DeWitt. The EXODUS optimizer generator. In *SIGMOD'87 Proceedings*, pages 160–172, 1987.

[Gio91] Mike Gioielli. Developing an expert system for database design. *AI Expert*, 6(10):42–46, October 1991.

[GP87] Jim Gray and Franco Putzolu. The 5 minute rule for trading memory for disc accesses and the 10 byte rule for trading memory for CPU time. In *SIGMOD'87 Proceedings*, pages 395–398, 1987.

[GW87] Richard A. Ganski and Harry K. T. Wong. Optimization of nested SQL queries revisited. In *SIGMOD'87 Proceedings*, pages 23–33, May 1987.

[Haa90] Laura Haas *et al.* Starburst mid-flight: As the dust clears. *IEEE Trans. Knowledege and Data Engineering*, 2, March 1990.

[Han87] Eric N. Hanson. *Efficient Support for Rules and Derived Objects in Relational Database Systems.* PhD thesis, University of California at Berkeley, August 1987.

[HHR90] Eric N. Hanson, Tina M. Harvey, and Mark A. Roth. Experiences in DBMS implementation using an object-oriented persistent programming language and a database toolkit. Technical Report AFIT/EN-TR-90-8, Air Force Institute of Technology, December 1990.

[IBM] IBM. *SQL/Data System for VSE: A Relational Data System for Application Development.* Manual Number G320-6590.

[IBM85] IBM. *Relational Design Tool—Structured Query Language/Data System*, 1985. Manual Number SH20-6451-1.

[IC91] Yannis E. Ioannidis and Stavros Christodoulakis. On the propagation of errors in the size of join results. In *SIGMOD'91 Proceedings*, pages 268–277, June 1991.

[IK90] Yannis E. Ioannidis and Younkyung Cha Kang. Randomized algorithms for optimizing large join queries. In *SIGMOD'90 Proceedings*, pages 312–321, May 1990.

[INGa] ASK Computer Systems, Inc., INGRES Products Division, 1080 Marina Village Parkway, Alameda CA 94501-4026. INGRES Documentation.

[INGb] INGRES technical note `note013.all`. Distributed with INGRES software.

[ING90] ASK Computer Systems, Inc., INGRES Products Division, 1080 Marina Village Parkway, Alameda CA 94501-4026. *INGRES Database*

Administrator's Guide for the UNIX Operating System, Release 6.2, April 1990.

[INSS92] Yannis E. Ioannidis, Raymond T. Ng, Kyuseok Shim, and Timos K. Sellis. Parametric query optimization. In *Proceedings of the Eighteenth International Conference on Very Large Databases*, pages 103–114, August 1992.

[JMSV92] M. Jarke, J. Mylopoulos, J. W. Schmidt, and Y. Vassiliou. DAIDA: An environment for evolving information systems. *ACM Trans. Information Syst.*, 10(1), January 1992.

[Kim82] Won Kim. On optimizing an SQL-like nested query. *ACM Trans. Database Syst.*, 7(3):443–469, September 1982.

[KKD89] Won Kim, Kyung-Chang Kim, and Alfred Dale. Indexing techniques for object-oriented databases. In Won Kim and Frederick H. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*, pages 371–394. ACM Press (Addison-Wesley Publishing Company), 1989.

[Koe81] Shaye Koenig. *A Transformational Framework for Automatic Derived Data Control and Its Application in an Entity-Relationship Data Model.* PhD thesis, New York University, 1981.

[KZG81] Shmuel Katz and Ruth Zimmermann-Gal. An advisory system for developing data representations. In *Proceedings of IJCAI (International Joint Conference on Artificial Intelligence), Vancouver, Canada*, 1981.

Revised version accepted for publication by *The Science of Computer Programming*.

[LKD$^+$88] V. Linnemann, K. Küspert, P. Dadam, P. Pistor, R. Erbe, A. Kemper, N. Südkamp, G. Walch, and M. Wallrath. Design and implementation of an extensible database management system supporting user defined data types and functions. In *Proceedings of the Fourteenth International Conference on Very Large Databases*, pages 294–304, 1988.

[Loh88] Guy M. Lohman. Grammar-like functional rules for representing query optimization alternatives. In *SIGMOD'88 Proceedings*, pages 18–27, 1988.

[Low78] James R. Low. Automatic data structure selection: An example and overview. *Communications of the ACM*, 21(5):376–385, May 1978.

[LS93] George F. Luger and William A. Stubblefield. *Artificial Intelligence Structures and Strategies for Complex Problem Solving, second edition.* The Benjamin/Cummings Publishing Company, Inc., 1993.

[Mar83] Salvatore T. March. A mathematical programming approach to the selection of access paths for large multiuser databases. *Decision Sciences*, December 1983.

[MC85] Salvatore T. March and John V. Carlis. Physical database design: Techniques for improved database performance. In Won Kim, David S. Reiner, and Don S. Batory, editors, *Query Processing in Database Systems*, pages 276–296. Springer Verlag, 1985.

[MC87] Salvatore T. March and John V. Carlis. On the interdependencies between record structure and access path design. *Journal of MIS*, 4(2), 1987.

[McG89] David McGoveran. Secrets of relational performance tuning. *Database Programming & Design*, 2(7):26–35, July 1989.

[MCN92] John Mylopoulos, Lawrence Chung, and Brian Nixon. Representing and using nonfunctional requirements: A process-oriented approach. *IEEE Trans. Software Engineering*, 18(6):483–497, June 1992.

[MJC88] Scott L. Vandenberg Michael J. Carey, David J. DeWitt. A data model and query language for EXODUS. In *SIGMOD'88 Proceedings*, pages 413–423, June 1988.

[MS86] David Maier and Jacob Stein. Indexing in an object-oriented DBMS. In *Proc. Int'l Workshop on Object-Oriented Database Systems*, pages 171–182. IEEE Computer Society Press, September 1986.

[Mur92] M. Muralikrishna. Improved unnesting algorithms for join aggregate SQL queries. In *Proceedings of the Eighteenth International Conference on Very Large Databases*, pages 91–102, 1992.

[New93] New York University Programming Langauges Group. *Griffin Reference Manual*, 1993.

[Nix91] Brian Nixon. Implementation of information system design specifications: A performance perspective. In *Proceedings of the Third*

International Workshop on Database Programming Languages, pages 149–168. Morgan Kaufmann, August 1991.

[Nix93] Brian A. Nixon. Dealing with performance requirements during the development of information systems. In *RE '93, IEEE International Symposium on Requirements Engineering, San Diego, CA*, January 1993.

[O'N91] Patrick E. O'Neil. The set query benchmark. In Jim Gray, editor, *The Benchmark Handbook for Database and Transaction Processing Systems*, pages 209–245. Morgan Kaufmann, 1991.

[ORA88] Oracle Corporation, 20 Davis Drive, Belmont CA 94002. *ORACLE RDBMS Database Administrator's Guide, Version 6.0*, November 1988.

[Pai84] Robert Paige. Applications of finite differencing to database integrity control and query/transaction optimizations. In Gallaire, Minker, and Nicholas, editors, *Advances in Database Theory, Volume 2*, pages 171–209. Plenum Press, 1984.

[RC87] J. E. Richardson and M. J. Carey. Programming constructs for database system implementation in EXODUS. In *SIGMOD'87 Proceedings*, 1987.

[RK77] Stan Rosenshhein and Shmuel Katz. Selection of representations for data structures, in Proceedings of a Symposium on Artificial Intelligence and Programming Languages. *SIGPLAN Notices*, 12(8):147–154, 1977.

[RS86] Lawrence A. Rowe and Michael Stonebraker. The commercial INGRES epilogue. In Michael Stonebraker, editor, *The INGRES Papers: Anatomy of a Relational Database System*, pages 63–82. Addison-Wesley, 1986.

[RS89] Steve Rozen and Dennis Shasha. Using a relational system on Wall Street: The good, the bad, the ugly, and the ideal. *Communications of the ACM*, 32(8):988–994, August 1989.

[RS91a] Steve Rozen and Dennis Shasha. A framework for automating physical database design. In *Proceedings of the 17th International Conference on Very Large Data Bases (Barcelona)*, pages 401–411. Morgan Kaufmann, September 1991.

[RS91b] Steve Rozen and Dennis Shasha. Rationale and design of BULK. In *Proceedings of the Third International Workshop on Database Programming Languages (Nafplion)*. Morgan Kaufmann, August 1991.

[SCF$^+$86] P. Schwartz, W. Chang, J. C. Freytag, G. Lohman, J. McPherson, C. Mohan, and H. Pirahesh. Extensibility in the Starburst database system. In *Proc. Int'l Workshop on Object-Oriented Database Systems*, pages 85–92. IEEE Computer Society Press, September 1986.

[SDDS86] J. T. Schwartz, R. B. K. Dewar, E. Dubinsky, and E. Schonberg. *Programming With Sets*. Springer-Verlag, 1986.

[Sha76] G. Shafer. *A Mathematical Theory of Evidence*. Princeton University Press, 1976.

[Sha92] Dennis Shasha. *Database Tuning: A Principled Approach.* Prentice Hall, 1992.

[SJGP90] Michael Stonebraker, Anant Jhingran, Jeffrey Goh, and Spyros Potamianos. On rules, procedures, caching and views in data base systems. In *SIGMOD'90 Proceedings*, pages 281–290, 1990.

[SKWH76] Michael Stonebraker, Peter Kreps, Eugene Wong, and Gerald Held. The design and implementation of INGRES. *ACM Trans. Database Syst.*, 1(3), September 1976.

[SSS81] Edmond Schonberg, Jacob T. Schwartz, and Micha Sharir. An automatic technique for selection of data representations in SETL programs. *ACM Trans. Prog. Lang. and Syst.*, 3(2):126–143, April 1981.

[ST85] M. Schkolnick and P. Tiberio. Estimating the cost of updates in a relational database. *ACM Trans. Database Syst.*, 10(2):163–179, June 1985.

[TOB91] Carolyn Turbyfill, Cyril Orji, and Dina Bitton. AS^{3}AP: An ANSI SQL standard scaleable [sic] and portable benchmark for relational database systems. In Jim Gray, editor, *The Benchmark Handbook for Database and Transaction Processing Systems*, pages 167–207. Morgan Kaufmann, 1991.

[ULT90] Digital Equipment Corporation. *ULTRIX/SQL Reference Manual*, June 1990. Manual Number AA-PBZ6A-TE (for INGRES).

[Val87] Patrick Valduriez. Join indices. *ACM Trans. Database Syst.*, 12(2):218–246, June 1987.

[Wal90] H. G. Walraven. KOFDO kennissyteem voor ondersteuning van het fysiek database ontwerp. Technical report, Gemeenshappelijk Administratiekantoor (GAK), Staalmeesterslaan 410, Amsterdam, The Netherlands, PO Box 8300, 1005 CA Amsterdam, April 1990.