Automating Physical Database Design: A Universal Approach and Its Evaluation*

Steve Rozen[†‡]
steve@soi.com

[†]Software Options, Inc.
22 Hilliard Street
Cambridge MA 02138
USA

Dennis Shasha[‡]
shasha@cs.nyu.edu

[‡] Courant Institute of Mathematical Sciences
New York University
251 Mercer Street
New York NY 10012
USA

June 3, 1993

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. Physical database design involves deciding on primary and secondary indexes, deciding how to vertically partition logical tables, deciding which queries should be materialized views, etc. These decisions require a sophisticated understanding of physical design options and query strategies, and involve 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 (e.g. two primary indexes on the same table) conflict. Using features simplifies the incorporation of a wide variety of design options and supports extensibility; 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 AS^{3}AP. Our experiments with the prototype show that it can produce good designs, and that the critical factor limiting their quality is the accuracy of INGRES's query cost estimates. The prototype implementation isolates dependencies on INGRES, permitting retargetability to a wide range of relational, nested-relational, and object-oriented DBMSs.

*©1993 by Steve Rozen and Dennis Shasha. All rights reserved. Submitted for publication.

1 Motivation and Objectives

Modern database management systems encourage a classic "programming by refinement" paradigm, which separates correctness concerns from performance concerns [15]. In this paradigm, developers initially produce correct queries on a schema that cleanly and intuitively represents the real-world entities of interest. Subsequently, the developers focus on improving performance, partly by adjusting the physical organization of the database.[1]

The job of improving performance in database applications is sometimes called "database tuning", of which [69] provides a system-independent overview. Physical database design is an important component of database tuning that seems amenable to a software solution; other such components include load control (transaction admission control) [51] and disk placement [75].

1.1 Rationale for a Physical Database Design Assistant

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 queries can also accommodate 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, [11] shows that a restricted form of secondary index selection is NP-complete.

[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.

- Considerable expertise is needed to understand the performance impact of physical design options offered by a particular database management system (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 [38, 23, 52].)

- 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 suggested by 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 [19]. 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 [10] 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 [62] also suggests that database performance is often a concern, and that a software assistant for physical database design would be a useful adjunct. DBMS tuning is 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 [21, 31] and in [13, 6]. To provide a solution to problem (1), it seems we need to enrich the solution space explored in [21]. 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 [21].

The papers [13, 6] 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 inspection-method-based generation of possible designs and on cost-based searching.

An alternative is to rely primarily on a knowledge-based approach; an important example is the commercial system RdbExpert [18], which we discuss in Section 7. We also discuss [9], which proposes using rules to produce a set of physical designs ranked by a measure of confidence in their quality.

1.3 Formal Statement of the Problem

We now formally define problem (1). Let $\text{Cost}(Q, D)$ denote the cost of computing query Q on a physical design D, and let $\text{Cost}(D)$ 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, such as 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 (and the data statistics), with low weighted Cost:

$$\text{Cost}(W, D) \stackrel{\text{def}}{=} \text{Cost}(D) + \sum_{\langle Q_j, \phi_j \rangle \in W} \phi_j \text{Cost}(Q_j, D). \quad (2)$$

We prefer to use frequencies rather than abstract weights (as [21] 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.

1.4 Example Physical Design Problem

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. 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 [3, 50, 27, 29, 42, 58, 67]. For concreteness, however, we apply our methodology to a specific DBMS, commercial INGRES[71, 60, 14, 1], and use a running INGRES example in describing our framework.

Figure 1 shows a simple logical schema[2] annotated with data 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; for `quotes` the logical keys are {`sno`, `pno`, `minqty`} and {`sno`, `pno`, `maxqty`}.

For each column in the tables, Figure 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 [21], adapted for expositional purposes.

parts #tuples=4000

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

orders #tuples=10000

column	ono	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	sno	pno	minqty	maxqty	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: Example Schema and Data Statistics

Figure 2 shows a four-query workload on the schema of Figure 1.[3] 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.

[3]One might think that the query Q_1 can be computed using the minimum column value supplied with the data statistics in Figure 1. However, INGRES never uses such statistics as the result of a query, because they may be out-of-date.

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 the order (if any) with a particular order number (ono)."

```
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
```

Figure 2: Example Workload

2 A Framework

As discussed above, we want an approach that is applicable both to various relational DBMSs and also to various post-relational models. 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 [1, 69] to the physical database design problem posed by Figures 1 and 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 data 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 [22, 25, 43]) 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.[4]

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

- Use features to represent different designs options (e.g. indexes).
- Use inspection methods[5] 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 Section 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 as a `heap` (an unordered sequence of records without a primary index) without secondary indexes.

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 $idx(1, \tau, t, \bar{c})$ (resp. $idx(2, \tau, t, \bar{c})$), $\tau \in \{$`hash`, `ISAM`, `btree`$\}$.[6] Then a design in which there is a primary `btree` index on

[4]The preceding paragraph is taken from [63], ©VLDB Endowment; reproduction here is permitted under copyright agreement with VLDB Endowment.

[5]"Inspection methods" are "engineering problem-solving methods based on the recognition and use of standard forms" [57]; in our approach, standard forms recognized in a query suggest standard forms of supporting the query with physical design options.

[6]INGRES offers three organizations for primary and secondary indexes: `ISAM`, `hash`, and `btree`. A primary `ISAM` index is a multi-level sparse index, and a primary `hash` index organizes a table by the value of a hash

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 good design for Q_3 mentioned in Section 2.1 (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 paper we assume that all feature sets are compressible.[7] 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

---

function on key values. An ISAM or hash table requires periodic, off-line reorganizations to maintain efficiency as it grows; their frequency depends on the pattern of insertions. A primary btree index is a multi-level dense index that does not usually require off-line reorganization. Secondary indexes are implementationally similar to tables, with the leaf pages of a secondary index being analogous to the data pages of a base table.

[7]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.)

1. $F \cup \{f\}$ is realizable, and

2. $\text{Cost}(Q, F \cup \{f\}) < \text{Cost}(Q, F)$.

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 $\text{cfs}(Q)$.

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 $\text{cfs}(W)$, is defined as $\bigcup_{\langle Q_j, \phi_j\rangle \in W} \text{cfs}(Q_j)$.

We now show a few results (first shown in [63]) 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', $\text{Cost}(W, F) \leq \text{Cost}(W, F')$.

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

Proof. To show a contradiction, suppose not, i.e. that for all ideal feature sets, F, $F - \text{cfs}(W) \neq \emptyset$. Consider a minimal ideal feature set, F'. Let $D = F' - \text{cfs}(W)$. By our assumption, $D \neq \emptyset$, so take any $f \in D$, and let $F'' = F' - \{f\}$. Now $\text{Cost}(W, F'') \not> \text{Cost}(W, F')$ (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 $\text{cfs}(W)$.

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 $S_j \subseteq \text{cfs}(Q_j)$ 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 $\text{cfs}(W)$. Such an F must exist by Theorem 1. If we let each $S_j = \text{cfs}(Q_j) \cap F$ we satisfy the corollary, since by the assumption that all feature sets are compressible, each $F \cap \text{cfs}(Q_j)$ is realizable, and

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

□

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 search algorithm, presented in Section 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 section instantiates this abstract framework for INGRES.

3 Instantiation for INGRES

In order to evaluate the utility of the physical database design framework presented above and in [63], we instantiated and implemented it for the commercial INGRES[8] relational database management system [71, 60, 14, 2]. 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 found 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.

The remainder of this section discusses particulars of the instantiation for INGRES: the Cost function and feature kinds. The following sections 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:

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

- 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 INGRES's plans and estimates as optimal.

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 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 [26]. In other 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 Kinds of Features

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. 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.

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 is a `hash` index and the other has more columns; see [61].)

Vertical Partitionings (Vertical Splitting, Vertical Declustering) A vertical partitioning represents a logical 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 query, Q, that can be computed using only A_1 (or A_2, respectively) can also be computed using only such an index (as discussed in Section 4.5).

Materialized Aggregates We can maintain certain aggregate queries as materialized views. As a simple example, the value of Q_1 in Figure 2 can be materialized in a one-column, one-row table containing the minimum value of `qonhand` in `parts`.

3.2.1 Other Design Parameters

INGRES offers the option of compressing character data by truncating trailing blanks.As a default heuristic, DAD-I uses a compressed organization except on tables subject to `update`s of a variable-length character field. The reason is that (at least in INGRES 5.0 [32]) the query optimizer does not consider the cost of expanding compressed data.[9] 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

[9] 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.)

updates, by default, DAD-I uses effective fill factors[10] of 90% for btree or ISAM and 75% for hash.[11] The database administrator can override DAD-I's defaults for compression and 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 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.

3.2.2 Features for Other DBMSs

One of the advantages of our approach is its generality. All of the feature kinds described above would be applicable to other DBMSs, such as Oracle version 7, though Oracle provides no ISAM or secondary hash indexes [40]. For Oracle we might be able to relax DAD-I's conflict rule for secondary indexes.

Many design options available in other DBMSs could easily be represented as features. For example, Oracle provides the option of co-locating tuples from more than one table on the same page, so that they share the same primary organization. This is called a cluster. A cluster essentially provides a single primary index on several tables, and as a feature would conflict with other primary-index features on the tables belonging to the cluster.

Future relational systems might offer design options such as join indexes [73], which could also be represented as features. Indexes for object-oriented databases (for example path indexes [46] or class-hierarchy indexes [39]) as well as various object-clustering disciplines would also be good candidates for representation as features.

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

[10] 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.

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

so large that searching among *useful*(j)'s subsets becomes intractable. This sections describes feature generation in DAD-I, except for features to support `order by` queries, for which see [61].

4.1 Static Indexes and Sequential Keys

Because `ISAM` and `hash` indexes require reorganization as their table grows they are called "static". (In contrast, `btree` organizations adjust automatically to most update patterns.) Even though it might be necessary to periodically reorganize static indexes, they are important to 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`.[12]

A further issue arises in `ISAM` indexes: insertions with index-key values that tend to monotonically increase or decrease over time are particularly deleterious, because all updates go to the last index page, transforming it into a long overflow chain. An index key with such an update pattern is called a "sequential key".[13] Column updates that have the effect of deleting an index key and always inserting it at an end of the index have the same effect.

Our approach to these issues is to allow the database administrator to supply two pieces of information on every table:

1. whether the database administrator is willing to use a static organization even if the table grows, and

2. which columns in the table that have sequential update patterns.

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 on t, or

- 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).

[12]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.

[13]The terminology "incremental key" or "incrementing key" is also used [1].

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 below and the discussion of `order by` queries in [61].) 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`.

4.2 Indexes for Join Predicates

An *(equality) join predicate* for a pair of correlation names,[14] t, t', in a query, q, is a predicate of the form $t.c = t'.c'$.[15] 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: $t.c$ `in (select` $t'.c'$ `from` $\cdots$). INGRES uses `btree` indexes on join columns to avoid the need for a sort when using a merge-join method, and also uses indexes in an index-join method ("key lookup join" [1]). 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 all the necessary columns for one of the join arguments can be read from the index (see Section 4.5).*

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.*[16]

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

[14] A "correlation name" is a SQL range variable. Rules for generating features are often best formulated in terms of correlation names. As a shorthand we often use a correlation name, t, to refer to the table named (or aliased) by t.

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

[16] This is consistent with INGRES 6.2 documentation, [1] pages 9-25, 9-26, and 9-29.

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

where *static*(t) and *sequential-key*($t.c$) are as defined in Section 4.1.[17] 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.

For the example of Figures 1 and 2, DAD-I generates the following features for join predicates for query Q_3 (the only query with join predicates):

idx(1, `hash`, `parts`, `pno`)
idx(1, `ISAM`, `parts`, `pno`)
idx(1, `btree`, `parts`, `pno`)
idx(1, `btree`, `orders`, `pno`)
idx(1, `hash`, `quotes`, `pno`)
idx(1, `ISAM`, `quotes`, `pno`)
idx(1, `btree`, `quotes`, `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

$$t.c \, \theta \, v_1 \tag{4}$$

[17] 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 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 4.5 below for more discussion.

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.[18] A selection predicate of the form (4) is an *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.*

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. We do this by comparing the number of disk reads required to use the index with the number required for a table scan. For details see [61].

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.

---

[18] 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 [1].
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.

$S(Q, t)$, 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 τ be `hash`; otherwise let τ be `btree`.
      ii. If *selective-enough*(*idx*$(1, \tau, t, \bar{c})$) then include *idx*$(1, \tau, t, \bar{c})$ in X.[19]
      iii. If *selective-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-key*$(t.c)$ then let τ' be `ISAM`; otherwise let τ' be `btree`.
         A. If *selective-enough*(*idx*$(1, \tau', t, \bar{c}{:}r)$) then include *idx*$(1, \tau', t, \bar{c}{:}r)$ in X.
         B. If *selective-enough*(*idx*$(2, \tau', t, \bar{c}{:}r)$) then include *idx*$(2, \tau', t, \bar{c}{:}r)$ in X.

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 example of Figures 1 and 2, DAD-I generates two features for selection predicates in Q_2, (the only query with a selection predicate): *idx*$(1, \texttt{btree}, \texttt{orders}, \texttt{ono})$ and *idx*$(2, \texttt{btree}, \texttt{orders}, \texttt{ono})$.

4.4 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 3.2, 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:

[19] Recall from footnote 17 that whenever DAD-I would generate an index that suffers from overflow DAD-I actually tries to add additional columns to it.

Procedure 2

Input: Q, a query in a workload.

Output: P, a set of vertical partitionings to be added to *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 *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 $vp(R, \{Z, (U - Z) \cup K\})$ 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 $vp(R, \{Z \cup K, (U - Z) \cup K\})$ to P.

Given the schema and workload of Figures 1 and 2, DAD-I adds the vertical partitioning *vp*(`parts`, {{`qonhand`, `pno`}, {`pno`, `descrip`}}) to *useful*(1), and the following to *useful*(3):

vp(`parts`, {{`qonhand`, `pno`}, {`pno`, `descrip`}})
vp(`orders`, {{`ono`, `oprice`, `pno`, `qty`}, {`date`, `ono`, `sno`}})
vp(`quotes`, {{`minqty`, `pno`, `price`, `sno`}, {`maxqty`, `minqty`, `pno`, `remarks`, `sno`}})

Here DAD-I generates *vp*(`parts`, {{`qonhand`, `pno`}, {`pno`, `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[20] all the columns of `orders`.

4.5 Column-Sufficient Indexes

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, t, are drawn from secondary index rather than the base table, we say the index is *column-sufficient* for t in Q. For example, INGRES can compute Q_1 in Figure 2 by

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

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 scanning $idx(2, \texttt{hash}, \texttt{parts}, \texttt{qonhand})$ would involve far fewer pages than scanning 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.

4.5.1 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 a query, Q, DAD-I also attempts to add to $useful(Q)$ the column-sufficient index $idx(2, \tau', R, c_1, \ldots, c_n, c_{n+1}, \ldots, c_{n+m})$, where

1. $\{c_{n+1}, \ldots, c_{n+m}\} = used(t, Q) - \{c_1, \ldots, c_n\}$,
2. $c_{n+1}, \ldots, c_{n+m}$ are sorted by column name (to canonicalize the column-sufficient indexes), and
3. the type of the new index,

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

provided that i) Q does not update R, and ii) INGRES allows a secondary index with $n + m$ columns. At 3, 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 $idx(2, \texttt{hash}, t, c, c')$ cannot. 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 and 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 $idx(2, \texttt{btree}, \texttt{orders}, \texttt{ono}, \texttt{date}, \texttt{oprice}, \texttt{pno}, \texttt{qty}, \texttt{sno})$ 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 ($\rightsquigarrow$ denotes "extends to"):

$$idx(1, \texttt{btree}, \texttt{parts}, \texttt{pno}) \rightsquigarrow idx(2, \texttt{btree}, \texttt{parts}, \texttt{pno,descrip})$$

$$\left.\begin{array}{r} idx(1, \texttt{ISAM}, \texttt{parts}, \texttt{pno}) \\ idx(1, \texttt{hash}, \texttt{parts}, \texttt{pno}) \end{array}\right\} \rightsquigarrow idx(2, \texttt{ISAM}, \texttt{parts}, \texttt{pno,descrip})$$

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

$$idx(1, \texttt{btree}, \texttt{quotes}, \texttt{pno}) \rightsquigarrow idx(2, \texttt{btree}, \texttt{quotes}, \texttt{pno,minqty,price})$$

$$\left.\begin{array}{r} idx(1, \texttt{ISAM}, \texttt{quotes}, \texttt{pno}) \\ idx(1, \texttt{hash}, \texttt{quotes}, \texttt{pno}) \end{array}\right\} \rightsquigarrow idx(2, \texttt{ISAM}, \texttt{quotes}, \texttt{pno,minqty,price})$$

4.5.2 Generating Column-Sufficient Indexes from Scratch

If—after generating column-sufficient indexes by extending previously generated indexes—it turns out that some correlation name, t, in query Q, has no column-sufficient index, and that Q does not write to R, then DAD-I generates an arbitrary index on $used(t, Q)$ (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 $static(t)$, 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 $used(t, C)$ is a sequential key) and $static(t)$, then DAD-I generates a `hash` index. If $\neg static(t)$ DAD-I always generates a `btree` index.

For the example of Figures 1 and 2, INGRES generates a single column-sufficient index from scratch, for Q_1: $idx(2, \texttt{ISAM}, \texttt{parts}, \texttt{qonhand})$.

4.6 Materialized Aggregates

DAD-I is designed to generate a materialized-view feature for queries of the form

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

or

`select` *aggregate-list non-aggregate-list* `from` t C `group by` *c-list*[21]

where

[21] Because of a technical problem with getting cost estimates from INGRES, DAD-I does not in fact handle `group by` queries. See Section 6.1.

1. t is a single table reference,
2. C is an optional `where` clause,
3. every column mentioned in C is also in *c-list*,
4. *aggregate-list* is a list of aggregate-column-function applications—i.e applications of one of `max`, `min`, `count`, `sum`, `avg`, and
5. *non-aggregate-list* is a list of column specifications without aggregate operators.[22]

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 and 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. [7, 70]). Restrictions 1–3 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.

Techniques for maintaining materialized views for larger classes of queries are straightforward [41, 56, 28, 7], though maintaining more complex aggregates is likely to be more expensive, and therefore less useful. Materializing `max` and `min` queries is especially useful in INGRES, which does not use indexes to compute these queries ([1], page 9-26).

4.7 Feature Generation for Other DBMSs

Admittedly, of all aspects of our method, feature generation is most dependent on the particulars of the DBMS for which we are doing physical design. This is unavoidable, because query optimizers differ considerably among DBMSs. Nevertheless, it is constructive to consider which of the feature generation procedures in this section could be used for feature generation for Oracle.

Generation of features to support joins (see Section 4.2) would differ substantially, partly because of the Oracle `cluster` design option (see Section 3.2.2) and partly because of differing join-processing methods. However, with this exception, the procedures above apply to Oracle with little modification. In fact, we believe that Sections 4.4, and 4.6 (partitionings and

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

materialized aggregates) and feature generation for `order by` queries [61] could be used as-is. (The relevant properties observed for INGRES almost certainly hold for Oracle.) Oracle, unlike INGRES, can use `btree` indexes when computing `max` and `min` queries, so we would also need a simple procedure to generate such indexes for these queries.

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 3 shows all the features generated for the example of Figures 1 and 2; each feature is labeled for future reference. Figure 4 shows *useful*(j) for each query in Figure 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. (The idea is similar to a beam search—one that uses the general technique of bounding at each step the number of states to be investigated [45].)

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 2. DAD-I discovers the following costs:

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

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

We formalize "best" as follows.

1. Let $max\text{-}best_I$ be a parameter supplied by the database administrator.
2. Let the objective function for Search I be $\text{Cost}^+(Q_j, F) \stackrel{\text{def}}{=} \text{Cost}(Q_j, F) + \text{Cost}(F)$.[23]

[23] If the objective function for Search I contained no term such as Cost(F), the algorithm of Procedure 3 below would become stalled in connected regions of same-cost states, because many feature sets can share the same Cost for Q_j.

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

Figure 3: Features for Example Workload and Schema

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

Figure 4: $useful(j)$ for Example Workload and Schema

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

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

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

4. Let $\bar{G}$ be the sequence $\left\langle F_k \text{ in } \bar{F} \text{ s.t. } \forall F_{k'}.(F_{k'} \subset F_k \Rightarrow k' \geq k)\right\rangle$. 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

$$best_I(j, \bar{G}, max\text{-}best_I) \stackrel{\text{def}}{=} G_1 \cup \bigcup_{k=2}^{x} G_j \tag{5}$$

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 $best_I(j)$.[24]

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 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 [34].

II+ (Iterative Improvement Plus) DAD-I uses the following algorithm in Search I:

Procedure 3

Input: Q_j, a query.

useful(j), the features generated for Q_j.

$max\text{-}best_I$, as discussed above.

[24] We don't use simply $\bigcup_{k=1}^{x} G_j$ on the right-hand side of equation (5) because if $|G_1| > max\text{-}best_I$ then $x = 0$. In this case we want at least G_1 in $best_I$(j).

num-tries, a parameter supplied by the database administrator.

Output: *lowest*(j), a feature set with the lowest cost found.

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 $\langle S, \text{Cost}(Q_j, S)\rangle$ to *q-table*.
   (d) Add the pair $\langle S, \text{Cost}(S)\rangle$ 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 $\langle S', \text{Cost}(Q_j, S')\rangle$ to *q-table*.
      iii. Add the pair $\langle S', \text{Cost}(S')\rangle$ to *s-table*.
      iv. If $\text{Cost}^+(Q_j, S') < \text{Cost}^+(Q_j, S)$ 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 Cost^+ of Q_j was evaluated, sorted as discussed above.
5. *lowest*(j) is F_1, and *best*$_I(j)$ is *best*$_I(j, \bar{F},$ *max-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 and 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).

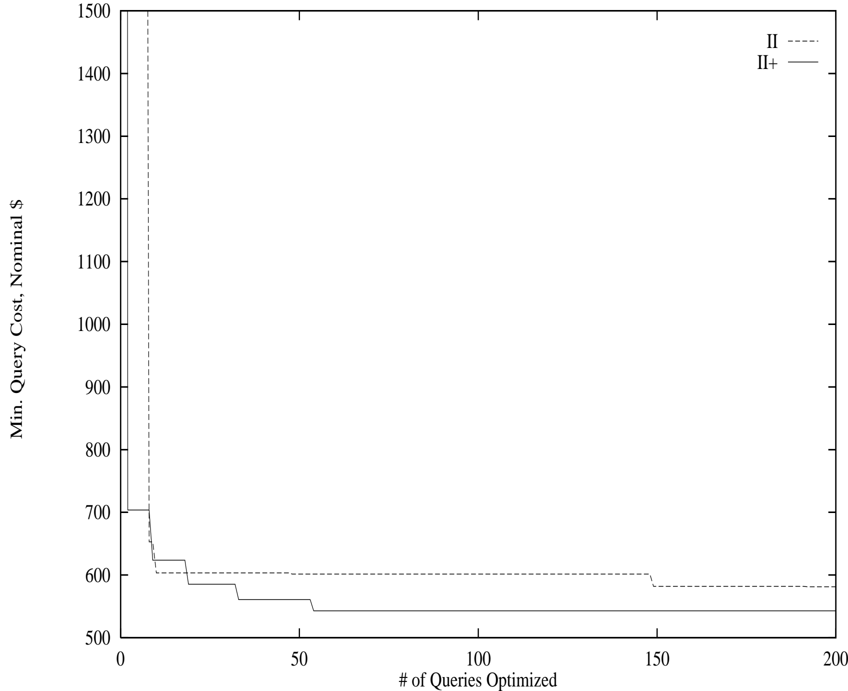


Figure 5: II vs. II+, Query `join_4_ncl`, AS^{3}AP

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 is for the query `join_4_ncl` in the AS^{3}AP workload, and Figure 6 is for query Q_3 from Figure 2. For Figure 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 6, an almost-exhaustive search revealed the same minimum as II+. (In fact, for the run of Figure 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 [34], 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 [34]

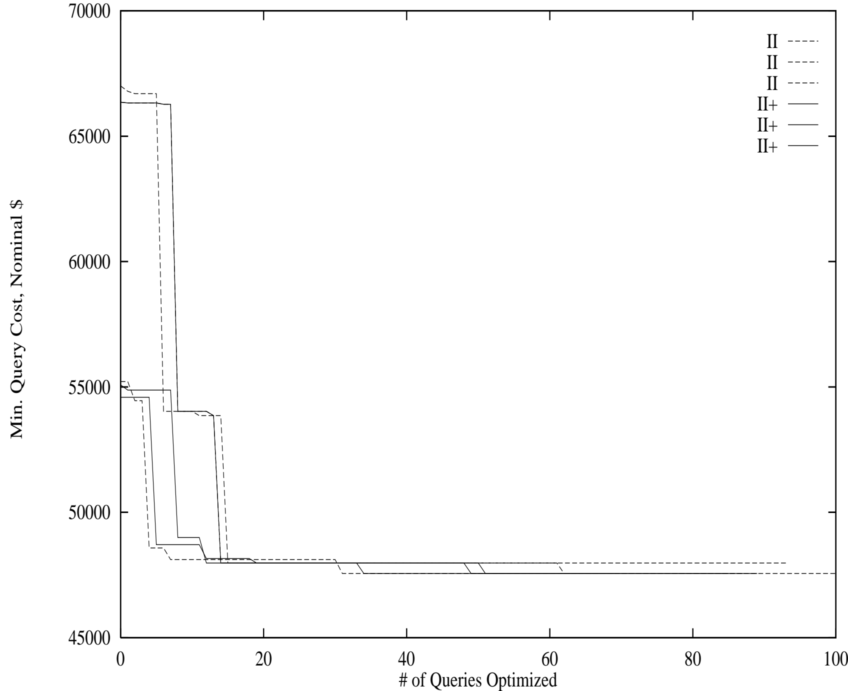


Figure 6: II vs. II+, Query Q_3, Example Workload

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 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 [34]).

Search I Results for Example Given $max\text{-}best_I = 6$, Figure 7 shows the $lowest(j)$ and $best_I(j)$ produced by DAD-I for the example of Figures 1 and 2.

5.2 Search II

Once DAD-I has calculated the $best_I(j)$'s and $lowest(j)$'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

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

Figure 7: Search I Results

sets for which Cost of W (rather than Cost$^+$ of Q_j) has been evaluated, and the cost used 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 4

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

max-$best_{II}$, a parameter supplied by the database administrator, which helps bound the size of S at step 6b.

Output: *lowest* $\subset \bigcup_j best_I(j)$, a feature set minimizing Cost of W among those sets for which Cost of W was evaluated; *lowest* is an approximation to an ideal feature set for W (Definition 6 in Section 2.2).

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$(W, lowest(j))$ is minimal.
3. Let *best* be $best_I(j')$.
4. Let *lowest* be *lowest*(j').
5. Set $X = X - \{j'\}$.
6. While $X \neq \emptyset$ do:
   (a) Let j_{worst} be a $j \in X$ such that ϕ_jCost$(Q_j, lowest)$ is maximal.[25]
   (b) Let S be $best_I(j_{\text{worst}}) \cup best$.

[25] We use j_{worst} as a heuristic because j_{worst} is making the largest contribution to Cost$(W, lowest)$ of any j whose $best_I$ features have not yet been considered in Procedure 4. 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.

(c) Let S' be the result of applying additional feature-generation procedures to S (see Section 5.2.1, below).

(d) Perform on $2^{S'}$ an iterative-improvement search similar to Procedure 3, except that the objective function is Cost of W rather than Cost$^+$ of Q_j.

(e) Let $\bar{F}$ be as in Procedure 3, 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 *lowest* to the first element in $\bar{F}$.

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

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.

For details of point 2 see [61]. Search II feature generation could be used as-is for a DBMS such as Oracle.

5.2.2 Search II Compromise on Example

For our running example from Figures 1 and 2 we get the following workload costs for each $lowest(j)$:

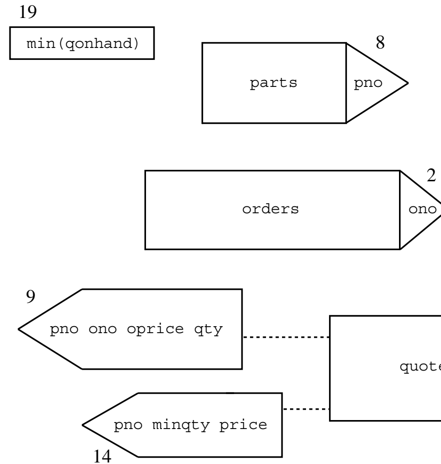
$$\begin{aligned}
\text{Cost}(W, \{\}) &= 833.001 \text{ ``C''},\ 1058.421\,\text{DRA},\ 738.0\,\text{pages} \\
\text{Cost}(W, \{19\}) &= 829.002 \text{ ``C''},\ 954.158\,\text{DRA},\ 739.0\,\text{pages}
\end{aligned}$$

$$\mathrm{Cost}(W,\{3\}) = 823.002\text{ "C"},\ 1023.263\,\textsc{dra},\ 809.0\,\text{pages}$$
$$\mathrm{Cost}(W,\{8,9,14\}) = 502.002\text{ "C"},\ 1450.105\,\textsc{dra},\ 1176.0\,\text{pages}$$

$\mathrm{Cost}(W, \mathit{lowest}(3))$ is lowest, and the maximum $\phi_j\mathrm{Cost}(Q_j, \mathit{lowest}(3))$ is that of Q_1, so $j_{\mathrm{worst}} = 1$. 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 $\mathit{lowest} = \{8, 9, 14, 19\}$ and that with $\mathit{max\text{-}best}_{II} = 8$, $\mathit{best} = \{0, 8, 9, 10, 14, 17, 18, 19\}$. The query with maximal $\phi_j\mathrm{Cost}(Q_j, \mathit{lowest})$ 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 lowest 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 *lowest*. We schematically represent the solution to the example problem as:



The features are

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 3.2). 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 $.

The query plans INGRES selected for $\{2, 8, 9, 14, 19\}$ are shown in [61].

6 Experiments

6.1 Implementation of DAD-I

DAD-I's implementation follows closely the description in the preceding sections.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 appears to be the time used by INGRES to update its catalog tables (data dictionary), and to plan the workload queries.

DAD-I runs the INGRES line-oriented terminal monitor (see [17], 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 catalog tables reflect the feature set to be evaluated. This can require creating or dropping both secondary and primary indexes. It can also involve altering the data statistics associated with 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 catalog tables 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. INGRES does not actually execute the query because DAD-I has issued INGRES's `set noqueryrun` statement. 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.

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[60], 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. As discussed in Sections 3.2.2 and 4.7, some unavoidable dependencies on INGRES arise in the design of features and feature-generation procedures. (A rule-based implementation of feature generation would probably make retargetability easier, but might also be slower.) In addition, the code that com-

municates with INGRES is INGRES-specific. Otherwise, to retarget Search I and II to another relational database would be straightforward provided there is some way to get query costs.

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 and 2).

6.2.1 Test Problem 1

Test Problem 1 is an adaptation of a physical database design problem given in [21]. (The differences are in queries 5, 6, and 9, which are multi-tuple updates in the original workload,[26] and in query 2, which is a `group by` query in the original workload.[27]) The schema and workload for this test are presented in [61].

Some properties of the data, notably selectivities and logical keys, were not reported in [21]. For such situations we invented properties consistent with the characteristics specified in [21].

6.2.2 Test Problems 2 and 3

Test Problem 2 is based on the AS^{3}AP benchmark [72]. We used the AS^{3}AP schema and most of the queries from the AS^{3}AP "single-user" test—a test designed 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.

We use 30 queries and updates from the AS^{3}AP single-user tests, 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.

[26] 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 [65].

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

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 ([61] provides details).

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 the test 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
17	*idx*(1, `hash`, `parts`, `partno`)
24	*idx*(1, `btree`, `orders`, `date`)
30	*idx*(1, `ISAM`, `quotes`, `suppno`,`price`)
32	*idx*(2, `ISAM`, `quotes`, `partno`,`price`,`suppno`)
85	*idx*(2, `btree`, `orders`, `orderno`,`partno`,`suppno`)

with estimated workload cost 1234.175 "C", 9317.0 DRA, 5047.0 pages. 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 *lowest*).

Discussion This is a reasonable physical design, though we know there is a design approximately 2.6% less expensive (reported in [61]).[28] Neither design contains a primary index on

[28]That design differed from the one reported here because of an inadvertent deviation from the specification of Procedure 1. That design's estimated cost is 1234.275 "C", 8389.0 DRA, 5507.0 pages.

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})

Figure 8: Test Problem 2 Solution

a key of orders or quotes. Such indexes are unwarranted based purely on the support they would provide for the queries of the workload. As was the case for the solution to the example problem of Figures 1 and 2, we suspect many database administrators would have overlooked the use of the column-sufficient indexes (32 and 85).

6.3.2 Test Problems 2 and 3

Figure 8 shows the lowest-cost physical design found by DAD-I for Test Problem 2, 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 *lowest*.)

Figure 9 shows the lowest-cost physical design found by DAD-I for Test Problem 3, 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

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	$materialize(Q_{18})$

Figure 9: Test Problem 3 Solution

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 3.2). These errors also led to the anomalous inclusion of a secondary index in one experiment with Test Problem 1 (discussed in [61]). 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 [21], it doesn't make sense to try to second guess the optimizer's plans, but, unlike [21], 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.

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 search among potential designs 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 cost estimates?
- How much human mediation is required and allowed? Can the system complete a partial solution provided by the database administrator?
- 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 [6, 13] 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 ([6], pg. 223)." Like FCDS's designers, we think this is best approached using a balance of knowledge-based and cost-based search components. A key difference between DAD-I and FCDS is that FCDS operates in the context of CODASYL databases and navigational query processing.

FCDS operates as follows. Its inputs are

- "LDS" (Logical Data Structure)—a conceptual-level schema[29] in an E/R-like model [8] (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).
- 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."([13], 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 [6, 13] the authors recognize that the problem is so hard that only a heuristic approach is possible. In [6] 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 [13] 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. [45], pages 329–331) to determine confidence in its conclusions. The report [12] offers a detailed description of the rules and their operation. Additional information on FCDS appears in [47, 5, 48, 49].

7.2 Relational Design Tool

Relational Design Tool for SQL/DS (RDT) [21, 31] is an index-selection tool for SQL/DS [30], and appears to be still available from IBM. In many ways, DAD-I is an attempt to

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

- 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), to existentially useful features (Definition 3 in Section 2.2), 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, and DAD-I's more general search methods mean that it is not possible to minimize database catalog updates as in RDT. In abandoning 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 [18, 16]. In another knowledge-based approach, [9] describes how to use the Dempster-Schafer theory of evidence [68] to assign a measure of "promise" to alternative physical database designs. An even earlier work, [4], apparently also took a knowledge-based approach.

7.3.1 RdbExpert

According to [18], 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 [24].

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. 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 [24] nor [18] reveals 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 partitioning—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 section, most approaches to physical database design involve some mixture of inspection methods and cost-based search. The proposal in [9] is to rely on strong, knowledge-based inspection methods 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 the 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 [74] 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 inspection methods and cost-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 [36]. The problem of physical database design arises as an instance of the problem of satisfying non-functional requirements. For example, [54, 55] discuss the issue of performance as a non-functional requirement, and [53] 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 [20], which describes an implementation technique for the functional database programming language ADABTPL.

Finally, there has been some work on main memory data structure design: [59, 44, 66, 37]. 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.

8 Conclusion and Future Work

8.1 Conclusion

This work has accomplished the following:

- It has successfully generalized the approach pioneered by [21], in which a design assistant uses a query optimizer's cost estimates to find low-cost physical designs.
- It proposes a DBMS-independent framework—consisting of feature sets and the separation of feature generation from search—that allows the use of generic search algorithms.
- It presents 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.
- It evaluates the framework and algorithms by means of an implementation for the INGRES DBMS. For this implementation we developed feature-generation procedures. Experiments showed these to be fast and effective, and also showed that design quality is limited not by the expense of search but by the accuracy of INGRES's query-cost estimates. The implementation isolates dependencies on INGRES, permitting retargetability to other relational or post-relational DBMSs.

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 [33]. 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 [35].

DAD-I's inspection method component, feature generation, is procedural and therefore fast. However, rule-based feature generation could speed up the search phase if the rules excluded some feature sets from consideration or assigned some measure of promise to feature sets. Rule-based feature generation might also ease retargeting to other DBMSs. Therefore we would like to investigate the question of how much to rely on rule-based feature generation as opposed to cost-based search among alternative designs. (We do think that using some cost-based search—as opposed to exclusive reliance on rules—offers advantages in terms of extensibility, retargetability, and output design quality.) We would also like to use our physical design framework in the implementation of a very-high-level persistent programming language such as BULK [64].

ACKNOWLEDGMENTS Professor Edmond Schonberg of New York University read the first author's thesis, and offered many comments that have improved this article. Comments by Professors Benjamin Goldberg, Ernest Davis, and Malcolm Harrison of New York University also were most helpful.

Professor Thomas E. Cheatham of Harvard University (and Software Options) provided much encouragement and support, as did Walter G. Morris, Glenn H, Holloway, Judy G. Townley, and Michael Karr of Software Options. Glenn Holloway also helped me resolve some thorny implementation difficulties, and Walter Morris was a source of continuous moral support.

NYU's Academic Computing Facility provided the hardware and software for running INGRES; we 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.

References

[1] 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*, Apr. 1990.

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

[3] D. 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 Eng.*, 14(11):1711–1730, Nov. 1990.

[4] M. 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, Mar. 1987.

[5] J. V. Carlis and S. T. March. A computer-aided physical database design methodology. *Computer Performance*, 4(4):198–214, Dec. 1983.

[6] J. V. Carlis, S. T. March, and G. W. Dickson. Physical database design: A DSS approach. *Info. & Management*, 6:211–224, 1983.

[7] S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. In *Proc. of the Seventeen Int'l Conf. on Very Large Databases*, pages 577–589, 1991.

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

[9] S. Choenni, H. M. Blanken, and T. Chang. On the automation of physical database design. In *Symposium on Applied Computing*, Feb. 1993.

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

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

[12] C. 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, Aug. 1989.

[13] C. E. Dabrowski, D. K. Jefferson, J. V. Carlis, and S. T. March. Integrating a knowledge-based component into a physical database design system. *Info. & Management*, pages 71–86, 1989.

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

[15] R. B. K. Dewar, A. Grand, S.-C. Liu, and J. T. Schwartz. Programming by refinement, as exemplified by the SETL representation sublanguage. *ACM Trans. Prog. Lang. and Syst.*, 1(1):27–49, July 1979.

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

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

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

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

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

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

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

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

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

[25] G. Graefe and D. J. DeWitt. The EXODUS optimizer generator. In *SIGMOD'87 Proceedings*, pages 160–172, 1987.

[26] J. Gray and F. 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.

[27] L. Haas *et al.* Starburst mid-flight: As the dust clears. *IEEE Trans. Knowledege and Data Eng.*, 2, Mar. 1990.

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

[29] E. N. Hanson, T. M. Harvey, and M. 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, Dec. 1990.

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

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

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

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

[34] Y. E. Ioannidis and Y. C. Kang. Randomized algorithms for optimizing large join queries. In *SIGMOD'90 Proceedings*, pages 312–321, May 1990.

[35] Y. E. Ioannidis, R. T. Ng, K. Shim, and T. K. Sellis. Parametric query optimization. In *Proc. of the Eighteen Int'l Conf. on Very Large Databases*, pages 103–114, Aug. 1992.

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

[37] S. Katz and R. 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.*

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

[39] W. Kim, K.-C. Kim, and A. Dale. Indexing techniques for object-oriented databases. In W. Kim and F. H. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*, pages 371–394. ACM Press (Addison-Wesley Publishing Company), 1989.

[40] G. Koch and R. Muller. *Oracle7: The Complete Reference.* Osborne McGraw-Hill, 1993.

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

[42] 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 *Proc. of the 14 Int'l Conf. on Very Large Databases*, pages 294–304, 1988.

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

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

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

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

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

[48] S. T. March and J. V. Carlis. Physical database design: Techniques for improved database performance. In W. Kim, D. S. Reiner, and D. S. Batory, editors, *Query Processing in Database Systems*, pages 276–296. Springer Verlag, 1985.

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

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

[51] A. Moenkeberg and G. Weikum. Performance evaluation of an adaptive and robust load control method for the avoidance of data-contention thrashing. In *Proc. of the Eighteen Int'l Conf. on Very Large Databases*, pages 432–443, Aug. 1992.

[52] M. Muralikrishna. Improved unnesting algorithms for join aggregate SQL queries. In *Proc. of the Eighteen Int'l Conf. on Very Large Databases*, pages 91–102, 1992.

[53] J. Mylopoulos, L. Chung, and B. Nixon. Representing and using nonfunctional requirements: A process-oriented approach. *IEEE Trans. Software Eng.*, 18(6):483–497, June 1992.

[54] B. 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, Aug. 1991.

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

[56] R. 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.

[57] C. Rich and R. C. Waters. *The Programmer's Apprentice*. Addison-Wesley Publishing Company (ACM Press Frontier Series), 1990.

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

[59] S. Rosenshhein and S. 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.

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

[61] S. Rozen. *Automating Physical Database Design: An Extensible Approach.* PhD thesis, Courant Institute of Mathematical Sciences, New York University, 251 Mercer Street, New York NY 10012-1185, Mar. 1993. Available by anonymous ftp from `cs.nyu.edu` as `/pub/theses/rozen.ps.Z`.

[62] S. Rozen and D. 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, Aug. 1989.

[63] S. Rozen and D. 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, Sept. 1991.

[64] S. Rozen and D. Shasha. Rationale and design of BULK. In *Proceedings of the Third International Workshop on Database Programming Languages (Nafplion).* Morgan Kaufmann, Aug. 1991.

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

[66] E. Schonberg, J. T. Schwartz, and M. Sharir. An automatic technique for selection of data representations in SETL programs. *ACM Trans. Prog. Lang. and Syst.*, 3(2):126–143, Apr. 1981.

[67] 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, Sept. 1986.

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

[69] D. Shasha. *Database Tuning: A Principled Approach.* Prentice Hall, 1992.

[70] M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos. On rules, procedures, caching and views in data base systems. In *SIGMOD'90 Proceedings*, pages 281–290, 1990.

[71] M. Stonebraker, P. Kreps, E. Wong, and G. Held. The design and implementation of INGRES. *ACM Trans. Database Syst.*, 1(3), Sept. 1976.

[72] C. Turbyfill, C. Orji, and D. Bitton. AS^3AP: An ANSI SQL standard scaleable [sic] and portable benchmark for relational database systems. In J. Gray, editor, *The Benchmark Handbook for Database and Transaction Processing Systems*, pages 167–207. Morgan Kaufmann, 1991.

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

[74] 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, Apr. 1990.

[75] G. Weikum, P. Zabback, and P. Scheuermann. Dynamic file allocation in disk arrays. In *SIGMOD'91 Proceedings*, pages 406–415, May 1991.