A Framework for Automating Physical Database Design

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

Abstract

We propose a two-phase algorithm for physical database design. In phase one the algorithm, for each logical query, uses rules to determine characteristics of a physical design (such as indexes) that would be beneficial to the query, and selects a physical design that yields a low cost estimate for that query. In phase two we use a notion of *compromise* between physical database designs. Starting from the physical designs selected in phase one, the algorithm looks for a compromise physical design that minimizes the queries' aggregate frequency-weighted cost. This method is envisioned as a cornerstone in the efficient implementation of a Turing-complete, very-high-level programming language for database applications, but it is also suitable for more conventional relational and ¬1NF database management systems.

1 Introduction

Optimal solutions to sub-problems of physical database design are NP-complete, e.g. secondary index selection [4]. Nevertheless, based on the pragmatic success of query (approximate) "optimization" and (approximate) "optimizations" in programming languages that are also intractable, we believe that physical database design can be automated in the vast majority of cases that arise in practice.

The problem we propose to solve is this: Given a logical database schema together with a set of queries on the schema and their frequencies, find a good physical database design, i.e. one as good as one that a competent human database designer with the same information would produce. This problem is especially important to those many databases where most of the queries are executed from "canned" application programs. We also accommodate ad hoc queries; execution plans for these can be computed relative to a fixed physical design as in current practice.

We see this problem as a key task in a project to implement BULK, a very-high-level language for prototyping and implementing database applications. The full language is behaviorally object-oriented [7] and treats all type constructors, including sets, uniformly (see [2] pg. 7). However, from a query-processing perspective we can think of BULK computations involving persistent data as queries in an extended non-first-normal-form (¬1NF) data model, i.e. a data model providing sets and sequences of tuples wherein tuple elements may themselves be sets or sequences (cf. [20, 14, 13]); this will be our perspective here. Indeed, our approach could be incorporated into a relational or ¬1NF database management system (DBMS) and with modification could be provided as a stand-alone tool for existing DBMSs.

Our approach relies on two key intuitions:

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. E.g. if a query involves only attributes 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 [9, 11, 15]) 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. E.g. suppose one query can be computed efficiently on a relation indexed by attribute a and another can be computed efficiently if the relation is indexed by attribute b, and that neither query can be computed using an index on attribute 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.

We next formalize these two intuitions enough to provide the basis of a practical system.

Once we have described this approach in more detail, the penultimate section of this paper discusses related work, and the last section discusses the advantages and disadvantages of our approach.

2 Describing the Problem and Possible Solutions

We describe the physical database design problem more precisely as follows. The input is a logical description of the database values and a *workload*, a set of pairs $\langle Q_i, \phi_i \rangle$, where Q_i is a query involving the database values and ϕ_i is the frequency, relative to the other queries in the workload, with which Q_i is computed ($\phi_i > 0$). The output is a physical design for which the frequency-weighted aggregate cost of the queries is low. For exposition we restrict our cost measure to disk random accesses (DRAs), and ignore CPU and storage costs. The specific objective function for the optimization does not heavily influence the methodology.

We describe database values and express queries in BULK, an Algol-like language augmented with relational-calculus-like expressions, somewhat in the spirit of Pascal/R [21] or Adaplex [18]. The language admits sets and sequences of dynamically varying size as primitive type constructors, and these type constructors can be non-recursively composed. Furthermore, values of arbitrary type may be shared among programs and persist between program executions, i.e. may be part of a database. For exposition we assume each query constitutes a transaction, and that the sets and sequences referenced in the query are persistent.[1]

Figures 1 and 2 are running examples of two databases and their respective workloads. Database values are described by *annotated schemas* (*schemas* for short), i.e. mappings from names to *value descriptions*. A value description is much like a data type, except that it is annotated with statistical information about a value. E.g. everything after the first (non-bracketed) colon in Figure 1(a) is a value description. Syntactically it resembles a type constructor with value parameters, but semantically the value parameters may be expected average values rather than constraints. Expected values may be gathered by the system as statistics or supplied by the programmer.

Thus in Figure 1(a) S is a set of records, each record consisting of four attributes, a, b, c, and d. S is estimated to contain 100,000 records, with 50,000 different values in the a attribute, 1,000 different values in the b and d attributes, and 10 different values in the c attributes. There are no keys in S.

[1]We emphasize that these are not restrictions in the BULK language, however; transactions may be demarked explicitly, and persistence and type are independent.

```
S :
  set (
   record(a : int,
          b : string(20),
          c : string(5),
          d : string(40)),
   100000,
   {a : 50000, b : 1000, c : 10, d : 1000})
```

(a) Schema 1.

Id	Query	ϕ

$Q_{1.1}$	`v1 <- read();`
	`print(`
	`  {[x.a, x.b]`
	`    : x in S st x.c = v1})`
		1

$Q_{1.2}$	`print([x in S order by x.a, <])`	0.01
$Q_{1.3}$	`v2 <- read(); S <- S ∪ {v2}`	0.05
$Q_{1.4}$	`v3 <- read(); S <- S - {v3}`	0.05

(b) Workload for Schema 1.

Figure 1: Schema 1 and Queries $Q_{1.1}$ through $Q_{1.4}$.

Figure 1(b) shows the workload for Schema 1.

$Q_{1.1}$ "Read a value, v1, and print the set of pairs [x.a, x.b] $\in$ S such that x.c = v1." The notation used is quite similar to that of SETL [22]. The square brackets ([,]) indicate that the the values of enclosed expressions are to be combined in a tuple (sequence). The curly braces ({, }) enclose a set former, the clause x in S indicates that x ranges over the elements of S, and the clause after st ("such that") is the predicate. Assignment is denoted by <-.

$Q_{1.2}$ "Print all elements of S ordered by their a values in ascending order."

$Q_{1.3}$ "Read a value and insert it into S."

$Q_{1.4}$ "Read a value and remove it from S."

The second example, in Figure 2, is an abstraction of the schema and workload of BondDB presented in [19]. Here there are two persistent values. G is a set of around 100 pairs where the second element in the pair is a sequence of, on average, 50 unique integers in the range [0, 100000]. The attribute g-id is a key. G can be thought of as a set of portfolios, and the elements of g-mems are the bonds in the portfolio. B is a set of around 10000 records that represent bonds and their recent quote (i.e. price) history. The attribute b-id

```
G :
  set(
    record(
      g-id   : int(0,200)
      g-mems :
        seq(int(0, 100000), 50, unique)),
    100,
    {g-id : 100, g-mems : 100}, {{g-id}});

B :
  set(
    record(
      b-id   : int(0, 100000),
      b-head : string(100),
      b-hist :
        set(
          record(
            date   : int(0, 366000),
            points : string(40))
          1000, {date : 1000, points : 1000},
          {{date}})),
    10000,
    {b-id : 10000,
     b-head : 10000,
     b-hist : 10000},
    {{b-id}})
```

(a) Schema 2.

Id	Query	ϕ

$Q_{2.1}$	`id <- read(); d <- read();`
	`print(`
	`  {[x.b-id, x.b-head,`
	`    [y in b.b-hist`
	`       st y.date > d`
	`       order by y.date,>] ]`
	`   : x in B st x.b-id = id})`
		0.5

$Q_{2.2}$	`p-id <- read();`
	`print(`
	`  [ [i, B(i).head, x]`
	`    :  i in G(p-id),`
	`       x in B(i).b-hist`
	`    st x.date`
	`       =`
	`       max(domain B(i).b-hist)`
	`  ] )`
		1.0

$Q_{2.3}$	`id <- read(); date <- read();`
	`quote <- read();`
	`B(id).b-hist(date) <- quote`
		0.01

(b) Workload for Schema 2.

Figure 2: Schema 2 and Queries $Q_{2.1}$ through $Q_{2.3}$.

is a user-visible unique identifier for the bond, and is a key; **b-head** contains on average about 100 bytes of additional information that in a real system would be broken up into a number of attributes. The attribute `b-hist` contains on average 1000 points in a time series of quotes i.e. a date, **date**, and some quotes, here abstracted as **points**. The attribute **date** is a key within each value of **b-hist**, and **b-id** is a key for **B**.

The queries on Schema 2 are as follows:

$Q_{2.1}$ "Read an id and a date; print the id, the header information for the bond with that id, and all quotes more recent than the date, in descending order of date." The final > in the **order-by** clause indicates descending order.

$Q_{2.2}$ "Read the id of a portfolio; print each bond in the portfolio and its most recent quote, all in the order the bonds appear in the portfolio." Here we use a convenient notation, called ***map-notation*** for retrieving records from a set by a key. Since **b-id** is the only key for **B** we view **B** as a function from **b-id** to ⟨**b-head**, **b-hist**⟩, and we may write `B(`x`)` to denote the unique ⟨**b-head**, **b-hist**⟩ value associated in **B** with **b-id** x. The operator **domain** applied to a set with a single key yields the set of all key values in the set.

$Q_{2.3}$ "Read a bond id, a date, and a quote; let the quote be the quote for that bond on that date." This query relies heavily on map notation; `B(id).b-hist` is the time series for the bond with id `id`, and

`B(id).b-hist(date) <- quote`

updates the **points** attribute for that quote at **date**, if there is one, and otherwise inserts a new record in `B(id).b-hist`.

2.1 Physical Schemas and Features

From such annotated schemas and workloads the physical design method must produce both physical data structures and physical query execution plans (QEPs). A physical schema involves the layout of files. The *basic (physical) schema* of a set or sequence is simply a file of records. In general, though, a physical schema is a set of optional *features* adjoined to a basic physical schema. Figure 3 shows the basic schema for Schema 1.

The kinds of features needed for the examples are: **v-partition** (vertical partition), **cluster** (store records of a single file with the same value for some attributes close together), **order** (store records in a file in a particular order), **=index** (include an index to support equality predicates, e.g. a hash index), **>index**

```
S : file(
      record(a : int,
             b : string(20),
             c : string(5),
             d : string(40)),
      100000, unique,
      {a : 50000, b : 1000,
       c : 10, d : 1000})
```

Figure 3: Basic Schema for Schema 1.

(include an index to support range queries on a single attribute, e.g. a B^+-tree). This is not intended as an exhaustive list of kinds of features, and could be augmented with others, e.g. co-location of records from two files based on the values of certain attributes or query maintenance.

Clearly not every set of features represents a physical schema. For example, a file with attributes a and b cannot be ordered both by $\langle a, b\rangle$ and by $\langle b, a\rangle$.

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

Definition 2.1 *A feature set that represents a physical schema is a realizable feature set.*

Definition 2.2 *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.[2] One advantage of this is that it simplifies determination of whether a feature set is realizable; in practice a notion of conflict among sets of features seems adequate. E.g. orderings on the same file conflict unless one is a prefix of the other.

We now 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. For a workload, $W = \{\langle Q_1, \phi_1\rangle, \ldots \langle Q_n, \phi_n\rangle\}$ we wish to minimize

$$\mathrm{Cost}(W, F) \overset{\mathrm{def}}{=} \sum_{\langle Q_i, \phi_i\rangle \in W} \phi_i \mathrm{Cost}(Q_i, F). \quad (1)$$

Definition 2.3 *A best feature set for workload W is any realizable feature set, F, such that for every realizable feature set F', $\mathrm{Cost}(W, F) \leq \mathrm{Cost}(W, F')$.*

[2]Given a non-compressible set of features, it is possible to define a, different, compressible set that can express the same physical schemas.

3 Phase One

In phase one each query is initially represented as a *basic plan*, a simple query execution plan (QEP) on the basic schema. A set of rules then inspects each individual query and basic plan in the workload to yield a set of potentially useful features and associated QEPs. From these a good QEP and physical schema are selected for the query, based on cost estimates.

A detailed description of these rules is beyond the scope of this paper, but roughly speaking they are concerned primarily with generating a set of "things to try." For example, if the output of a query is required to be in a particular order, a potentially useful feature would be to store the data in that order. Similarly, if a predicate of the form $x.v = w$ appears in a set former, an index on $x.v$ is potentially useful.

To be more specific about what it means for a feature to be useful to a query, we introduce the following definitions.

Definition 3.1 *A feature f is existentially useful to a query, Q, if there exists a realizable feature set, F such that*

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

Definition 3.2 *The set of all existentially useful features for a query, Q, is termed the complete feature set of that query, denoted* $\mathrm{cfs}(Q)$.

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

Definition 3.3 *The complete feature set of a workload, $W = \{\langle Q_1, \phi_1\rangle, \ldots \langle Q_n, \phi_n\rangle\}$, denoted* $\mathrm{cfs}(W)$, *is defined as*

$$\bigcup_{\langle Q_i, \phi_i\rangle \in W} \mathrm{cfs}(Q_i).$$

Lemma 3.1 *Given a workload, W, there must be some best feature set, F, for W such that $F \subseteq$* $\mathrm{cfs}(W)$.

Proof. To show a contradiction, suppose not. Consider a minimal best feature set, F'. Let $U = F' - \mathrm{cfs}(W)$. By our assumption, $U \neq \emptyset$, so take any $f \in U$, and let $F'' = F' - \{f\}$. Now $\mathrm{Cost}(W, F'') \not> \mathrm{Cost}(W, F')$ (otherwise f would be existentially useful.) Furthermore F'' is a strict subset of F', showing that F' is not minimal. Contradiction. □
This lemma tells us then, that, in searching for a best feature set for W we can confine our attention to subsets of $\mathrm{cfs}(W)$.

Theorem 3.1
Given a workload, $W = \{\langle Q_1, \phi_1\rangle, \ldots \langle Q_n, \phi_n\rangle\}$ there

exists, for each $\langle Q_i, \phi_i \rangle \in W$ *a realizable feature set,* $S_i \subseteq \mathrm{cfs}(Q_i)$*, such that there exists a best schema,* F*, with the property that*

$$F \subseteq \bigcup_{1 \le i \le n} S_i.$$

Proof. Consider a best feature set, F, that is also a subset of cfs(W). Such an F must exist by Lemma 3.1. We can find S_i's satisfying the theorem as follows. Clearly

$$\begin{aligned} F = F \cap \mathrm{cfs}(W) &= F \cap \bigcup_{1 \le i \le n} \mathrm{cfs}(Q_i) \\ &= \bigcup_{1 \le i \le n} (F \cap \mathrm{cfs}(Q_i)). \end{aligned}$$

By the assumption that all feature sets are compressible, each $F \cap \mathrm{cfs}(Q_i)$ is realizable, and therefore $F \cap \mathrm{cfs}(Q_i)$ is an S_i satisfying the theorem. □

Depending on the characteristics of each query, Q_i, in phase one Q_i may be recognized by rules that generate a subset of cfs(Q_i). Phase one then searches this subset for a realizable feature set that minimizes the cost of Q_i. This cost is the *ideal cost* of Q_i, and the associated feature set is an *ideal feature set* for Q_i, representing an *ideal physical schema*.

3.1 Examples

For query $Q_{1.1}$ the basic plan could be represented internally as:

```
print(
 collect-set(
  map(
   filter(
    scan(S),
    (lambda (x) x.c = v)),
   (lambda (x)
     enumerated-sequence(x.a, x.b)))))
```

The notation for QEPs recalls that of [10]. Here **collect-set** has the job of removing duplicates; its precise implementation depends on the eventual use made of the results.

The cost of this plan is estimated based on the statistics in the value description. Space prohibits a discussion of the estimation procedures, but they are similar to those presented in [23]. Because we restrict our attention to disk random access (DRA) as a cost measure, only **scan** and **collect-set** have non-zero cost. The cost for **scan(S)** is bytes(S)/β = 4117, where β = 1676 is the number of usable bytes per page, and for a physical value, ν, bytes(ν) is the number of bytes required to store ν. Based on the assumption that

Table 1: Ideal Feature Sets and Costs for $Q_{1.1}$ through $Q_{1.4}$.

feature	description
f_1	Vertical partition of S with a, b, and c in one file, and d in the other.
f_2	Cluster S by c.
f_3	Index S by c.
f_4	Order S by a.
f_5	Index S by a, b.

query	ideal feature set	ideal cost
$Q_{1.1}$	$\{f_1, f_2, f_3\}$	341
$Q_{1.2}$	$\{f_4\}$	4117
$Q_{1.3}$	$\{f_5\}$	5
$Q_{1.4}$	$\{f_5\}$	5

collect-set would require some paging even if implemented in virtual memory, estimate that the cost for **collect-set**(x), is bytes(x)/β. In this case we estimate the selectivity of the **filter** predicate to be 0.1, so the output of **filter** has 10000 elements, each mapped to a 24-byte tuple. This gives 240,000 bytes for the output and a cost of 143 for **collect-set**. The cost of the basic plan is then 4117 + 143 = 4260.

One pattern of interest in the basic plan is **map** composed with **filter** partially applied to a predicate that is an equality test with an unknown value composed with a record selection, all composed with a **scan**. This suggests indexing and clustering by attribute c, and using an index scan of S. A second rule recognizes that $Q_{1.1}$ never uses the d attribute from elements of S. These rules yield the ideal feature set in Table 1 and the ideal schema in Figure 4. The cost of a QEP using an index on the c attribute of file S1 is then 198, and the total cost for $Q_{1.1}$ is 341. Table 1 also shows the ideal feature sets for $Q_{1.2}$, $Q_{1.3}$, and $Q_{1.4}$.

For query $Q_{1.2}$ we would have the following basic plan.

```
print(
  collect-tuple(
    sort(
      scan(S),
      (lambda (x, y) x.a < y.a))))
```

One rule would take this to:

```
print(collect-tuple(scan(S)))
```

while taking the feature set to that shown in Table 1. A second rule, observing that nothing needs to be done to S to regard it as sequence, would take the plan to simply:

```
S1 : file(
       record(a      : int,
              b      : string(20),
              c      : string(5),
              S2-tid : tid(S2)),
       100000, unique,
       {a      : 50000,
        b      : 1000,
        c      : 10,
        S2-tid : 100000},
       {}, clustered(c));

S-index : =index(S1,{c});

S2 : file(
       record(d      : string(40),
              S1-tid : tid(S1)),
       100000, unique,
       {d : 1000, S1-tid : 100000},
       {{S1-tid}},
       ordered(S1-tid))
```

Figure 4: Ideal Physical Schema for $Q_{1.1}$.

`print(S)`

Depending on the actual implementation of `print` the actual plan would be something like

`map(scan(S), print)`

with DRA cost 4117.

Finally, for queries $Q_{1.3}$ and $Q_{1.4}$ the basic plans are

`insert(S,v2)`

and

`delete(S,v3)`

each with cost $2 + .875\text{bytes}(S)/\beta \approx 3604$.

The applicable rule is to use an index to support membership testing; the index is on a subset of attributes, here {a, b}, providing adequate selectivity. The feature set is as in Table 1 and the plans are

`indexed-insert(S,S-index,v2)`

and

`indexed-delete(S,S-index,v3)`

each with cost 5 DRA.[3]

The rules that lead to the ideal feature sets for Schema 2 are:

[3]We estimate 2 DRA to confirm that the value is not already in the set (resp. to find the value to delete), 2 to update the index and 2 to update the file, and assume that 1/4 of inserts (deletes, resp.) will be of elements already in (not in, resp.) the set.

$Q_{2.1}$:
- The **b-hist** attributes of elements of **B** are ordered by **date** in this query; therefore order them that way in the physical schema (feature g_1).
- A single element of **B** is retrieved by its **b-id** value, and **b-id** is a key. Therefore place an **=index** on **b-id** (g_2). (It is not necessary to cluster **B** by **b-id** because it is a key.)

$Q_{2.2}$:
- A single element of **G** is retrieved by its **g-id** value; same rule as for **b-id** in $Q_{2.1}$ suggests an **=index** on **g-id** (g_3).
- Same rule as for $Q_{2.1}$ (g_2).
- The element of **b-hist** with maximum date is retrieved, therefore place an **>index** on the **date** attribute (g_4).

$Q_{2.3}$:
- Same rule as for $Q_{2.1}$ (g_2).
- An **=index** on **date** for elements of the **b-hist** attribute of **B** (g_5).

4 Phase Two

Given a workload, W, phase two conceptually involves a search through all the feasible subsets of cfs(W). Since the search space is large, a heuristic approach is necessary. No matter what the heuristic used, there are two optimizations that should be employed.

1. When computing Cost(W, F) for a particular feature set, F, according to formula (1), there is clearly no need to continue once the partial sum plus the sum of the ideal costs for the unsummed queries exceeds the minimum value of Cost(W, F') for some F' already considered. I.e. let $W' \subset W$ and let F_i denote the ideal feature set of Q_i. There is no need to continue once

$$\sum_{(Q_i,\phi_i)\in W'} \phi_i \text{Cost}(Q_i, F) + \sum_{(Q_i,\phi_i)\in(W-W')} \phi_i \text{Cost}(Q_i, F_i) \geq \text{Cost}(W, F').$$

   This is important if computing Cost involves estimating the cost of many QEPs.

2. Given a query, Q, a feature set, F, and a feature, f, in many cases any QEP for Q on F is also a query plan for Q on $F \cup \{f\}$ with the same cost, as when e.g. f is an index and Q involves no updates. It is not necessary to estimate the cost of the same QEP for both F and $F \cup \{f\}$.

The following definitions and lemma show us some cases where the second optimization is possible:

Definition 4.1 *We say f is "QEP monotonic" for a query, Q, if for every realizable feature set, F, the following holds: $F \cup \{f\}$ is realizable implies that any query plan for Q on F is also a query plan for Q on $F \cup \{f\}$.*

For example, indexes, orderings, clusterings and maintained queries are QEP monotonic for read-only queries, but vertical partitions are not.

Definition 4.2 *We say f is "cost anti-monotonic" ("cost monotonic", resp.) for Q if for every feasible feature set, F, the following holds: $F \cup \{f\}$ is feasible implies* $\text{Cost}(Q, F \cup \{f\}) \leq \text{Cost}(Q, F)$ *(*$\text{Cost}(Q, F) \leq \text{Cost}(Q, F \cup \{f\})$*, resp.)*

Lemma 4.1 *Let F be a realizable feature set, let f be a feature, and let Q be a query. If (i) the cost of any operation in any QEP for Q is no greater on $F \cup \{f\}$ than on F, and (ii) f is QEP monotonic for Q, then f is cost anti-monotonic for Q.*

Proof. Let X be a QEP for Q on F with minimum cost. By definition of QEP monotonic, X is also a QEP for $F \cup \{f\}$. By assumption, X is no more expensive on $F \cup \{f\}$ than on F, and therefore $\text{Cost}(Q, F \cup \{f\}) \leq \text{Cost}(Q, F)$. □

By Lemma 4.1, given the usual set of QEP operations, indexes, orderings, clusterings, and maintained queries are cost anti-monotonic on read-only queries.

4.1 Binary Compromise

We now propose a specific heuristic, binary compromise, for the search.

Let $W = \{\langle Q_1, \phi_1\rangle, \ldots, \langle Q_n, \phi_n\rangle\}$ be a workload for some logical schema, and for every Q_i in the workload let F_i be its ideal feature set. Let F^* be the set of all ideal feature sets for W. (Several queries may have the same F_i.) For ideal feature set F let $\hat{Q}(F)$ denote the set of queries for which F is the ideal feature set.

1. Begin by finding an $F_k \in F^*$ that minimizes $\text{Cost}(W, F_k)$, i.e. such that for all $F_j \in F^*$ $\text{Cost}(W, F_k) \leq \text{Cost}(W, F_j)$. This feature set is the best overall among the set of ideal feature sets. Call this F_{current}. Let $\mathcal{T}$ be $F^* - \{F_{\text{current}}\}$.

2. Find the $F_k \in \mathcal{T}$ such that the queries for which F_k is the ideal feature set must do a lot of extra work if they use F_{current}. I.e. let the *stand-out cost* of a feature set, S, on F_{current} be

$$\text{SO}(S, F_{\text{current}}) \overset{\text{def}}{=} \sum_{Q_i \in \hat{Q}(S)} \phi_i(\text{Cost}(Q_i, F_{\text{current}}) - \text{Cost}(Q_i, S)).$$

Then find an $F_k \in \mathcal{T}$ maximizing $\text{SO}(F_k, F_{\text{current}})$, i.e. such that for all $F_j \in \mathcal{T}$, $\text{SO}(F_j, F_{\text{current}}) \leq \text{SO}(F_k, F_{\text{current}})$. We call this a *target* (denoted F_{target}). As the algorithm proceeds $\mathcal{T}$ will continue to be the set of possible targets. A target is, intuitively, an ideal feature set for a query such that the query contributes the largest "unnecessary" expense to the aggregate cost when the feature set is F_{current}. The algorithm terminates when $\mathcal{T}$ is empty, or, alternatively, when the sum of stand-out-costs over all feature sets in $\mathcal{T}$ is less than a user-definable fraction of $\text{Cost}(W, F_{\text{current}})$.

3. Compute $\text{Cost}(W, F)$ for some or all of the feature sets, F, in $F_{\text{current}} \oplus F_{\text{target}}$, where $A \oplus B$ denotes $\{F \in 2^{A \cup B} \text{ s.t. } F \text{ is realizable}\}$. Let $\mathcal{C}$ be a heuristically determined subset of $F_{\text{current}} \oplus F_{\text{target}}$. Let F' be a feature set in $\mathcal{C}$ minimizing $\text{Cost}(W, F')$. Let $\mathcal{T}$ be $\mathcal{T} - \{F_{\text{target}}\}$, let F_{current} be F', and go to step 2.

The binary compromise algorithm calculates Cost for $\mathcal{O}(p * 2^q)$ feature sets, where q is the cardinality of the largest pairwise union among the initial feature sets and p is the number of distinct ideal feature sets. A complete search would require us to calculate Cost for $\mathcal{O}(2^n)$ feature sets, where n is the cardinality of the union of all the ideal feature sets. For most realistic problems we expect q to remain small (e.g. ≤ 10), though if necessary, it is always possible to discard some features.

The main point, however, is not just a particular search algorithm, but the uniform framework that can accommodate various kinds of physical design strategies as features.

4.2 Examples

The compromise phase works on the ideal feature sets of Logical Schema 1 as follows. Using the ideal feature sets in Table 1, starting with $\{f_1, f_2, f_3\}$, the algorithm begins by finding the initial F_{current}.

We already know the cost of computing $Q_{1.1}$ on this schema (341 DRA). Suppose the best plan for $Q_{1.2}$ on this feature set is

```
sort(
 tid-merge-join(scan(S2),scan(S1),S1-tid),
 (lambda (x, y) (x.a < x.b)))
```

with associated cost

$$\begin{aligned} & (\log_4 c)c\text{bytes}(\text{elem}(\texttt{S}))/\beta + \\ & \quad [\text{bytes}(\texttt{S1}) + \text{bytes}(\texttt{S2})]/\beta \\ \approx & \quad (8.3 \times 10^5 \times 69 + 77 \times 10^5)/1676 \approx 38785 \end{aligned}$$

where $c = 100000$.[4] The query plan for Query $Q_{1.3}$ on $\{f_1, f_2, f_3\}$ is:

```
if not(
    exists(
     filter(
      tid-merge-join(
       filter(
         scan(S2), (lambda(f2) f2.d = v2.d)),
       filter(
        =index-scan(S1,S-index,v2.c),
        (lambda (f1) f1.tid))),
       S1-tid)))
 then
  let tid be
   indexed-insert(
    S1, S-index,
    make-record(
     record(a:integer, b:string, c:sting),
     v2.a, v2.b, v2.c, null-tid));
  let tid2 be
    insert-in-order(
     S2,
     make-record(
      record(d : string, S1-tid : tid(S1)),
      v2.d, tid));
   tid-update(
     S1,
     tid,
     (lambda (x) (x.S2-tid <- tid2)))
```

A rough estimate of the cost of this plan is

$$\begin{aligned} & .875(.1\text{bytes}(\text{S1}))/\beta + 1 + \\ & \quad .875\text{bytes}(\text{S2})/\beta + .75(4+4) \\ = & \ 288750/1676 + 1 + 3850000/1676 + 6 \\ \approx & \ 2476 \end{aligned}$$

Query $Q_{1.4}$ is similar.

Given the frequencies ($\phi_1 = 1$, $\phi_2 = .01$, $\phi_3 = \phi_4 = .05$), and with W denoting the workload of the first example,

$$\begin{aligned} & \text{Cost}(W, \{f_1, f_2, f_3\}) \\ & \quad = 341 + .01 \times 38785 + .1 \times 2476 \approx 976. \end{aligned}$$

This is the initial F_{current}; the algorithm would also have to calculate Cost$(W, \{f_4\})$ and Cost$(W, \{f_5\})$.

Initially F_{target} is $\{f_5\}$, and the algorithm searches $\{f_1, f_2, f_3\} \oplus \{f_5\} = 2^{\{f_1, f_2, f_3, f_5\}}$. Once more, in the interest of brevity, we only work out the alternative, $\{f_1, f_2, f_3, f_5\}$, that is selected. This represents a vertically partitioned physical schema with an index on each of {c} and {a,b} and clustered by c. Only the costs of queries $Q_{1.3}$ and $Q_{1.4}$ change, to approximately 8 DRA, and the aggregate cost is

$$341 + .01 \times 38785 + .1 \times 8 \approx 729.$$

The feature set finally selected for Schema 2 is $\{g_1, g_2, g_3, g_4\}$.

[4] We assume that the second file of the partition is ordered by tid of the first file, and that records of the first file that occur in the same block have tids with identical prefixes.

5 Related Work

There have been few attempts to build complete practical systems for the physical database problem. Instead much work has focused on more intellectually manageable sub-problems. E.g. recently [3] provides an algorithm for secondary index selection (in databases using tid-intersection) for queries on single relations, and [5] provides an integer linear-programming algorithm for vertical partitioning.

We consider theoretical research on relatively constrained sub-problems as complementary to the pragmatic problem of automating physical database design. This appears similar to the history of query planning, where, as [12] notes, initial work focused on sub-problems. General-purpose methods for solving the practical problem were developed later, and eventually incorporated knowledge about special cases studied previously. Thus we believe that, despite the apparent unmanageability of the full physical database design problem from theoretical perspectives, it can and should be solved pragmatically.

Both [6] (with earlier, related efforts reported in [17, 16]) and [8] address the problem of pragmatic physical database design. We concur with these authors that uncertainties about the expected workload, data statistics, and estimating procedures mean that the reasonable goal is a *good, not optimal* solution to the physical design problem.

The approach discussed in [6] starts from a restricted entity-relationship schema and a workload; the queries are specified navigationally at present, though accommodation of query optimization is a future research objective. The first stage of processing uses a knowledge-based component to apply some 400 rules to the schema and workload, yielding a number of different first-approximations to the physical design in the form of hierarchical record descriptions. Subsequent stages subject these to separate algorithms that (i) further vertically partition the records and (ii) select access paths (orderings and indexes). These algorithms operate on a single file at a time.

Although the input to this system is quite different from what we propose, we note with interest the rough similarity in the use of a rule-based initial stage followed by more algorithmic processing. However, the similarity is only approximate. For example in [6] the initial rule-based processing performs some of the functions our phase two by mediating between conflicting requirements of different queries. Also, by representing physical schemas as feature sets our approach considers both access paths and vertical partitions in the same search rather than as quasi-separable problems.

Another exception to the focus on sub-problems is DBDSGN (commercialized by IBM as Relational Design Tool), an implemented physical design tool for System R [1] described in [8].

In DBDSGN, as in our method, physical design decisions are based on QEP cost estimates. However, DBDSGN cannot take advantage of cost anti-monotone features while estimating query costs, since the query planner is part of an independent DBMS and computes QEPs *de novo* for each candidate physical design.[5]

DBDSGN's search strategy is also coupled to specific characteristics of System R storage structures and QEPs (e.g. join order restrictions). By comparison, the feature set compromise method can accommodate a wider variety of physical design strategies, including vertical partitions and (not elaborated in this paper) co-location of several relations, maintenance of aggregate queries and queries involving compile-time constants, and various kinds of data duplication.

6 A Final Example

As a final example consider the schema and workload of Figures 5 and 6, part of an example used in [8].

In this example we use as relative frequencies the "weights" of [8], and a larger page size. Selectivities, not presented in [8], are supplied by the current authors. As before, we restrict out attention to DRA costs.

We might expect phase one to generate 14 potentially useful indexes, 12 potentially useful orderings, and 3 potentially useful vertical partitions. From these, cost estimation would yield the following ideal organizations and ideal costs:

query	ideal feature set	ideal cost
$Q_{3.1}$	$\{I_1\}$	6
$Q_{3.2}$	$\{I_2, O_1\}$	25
$Q_{3.3}$	$\{I_3, O_2, V_1, I_4, O_3\}$	431
$Q_{3.4}$	$\{I_5, O_4, V_2, I_6, O_5, V_3, I_4, O_3\}$	127

where the features are as follows:

- Vertical Partitions
  - V_1 is a partition of `Quotes` with `suppno`, `partno`, and `price` in one file, and the remaining attributes in the other.

```
Parts : set(
          record(
            partno  : int,
            qonhand : int,
            descrip : str(198),
          ),
        8000,
        {partno  : 8000,
         qonhand : 4000,
         p-info  : 8000},
        {{partno}});

Orders : set(
           record(
             orderno : string(6),
             partno  : int,
             suppno  : string(3),
             date    : date,
             qty     : int,
             o-info  : string(74)),
           24000,
           {orderno : 24000,
            partno  : 8000,
            suppno  : 100,
            date    : 400,
            qty     : 12000,
            o-info  : 24000},
           {{orderno, partno, suppno}});

Quotes :
  set(record(
        suppno : string(3),
        partno : int,
        minqty : int,
        maxqty : int,
        price  : int,
        q-info : string(126)
      ),
      72000,
      {suppno : 100,
       partno : 8000,
       minqty : 4000,
       maxqty : 4000,
       price  : 32000},
      {{suppno, partno, minqty, maxqty}})
```

Figure 5: Schema 3.

[5]Naturally, any attempt to use our method with an independent query planner would likewise have to rely on QEPs selected by the planner.

Id	Query	ϕ
$Q_{3.1}$	`v<-read();` `orders <- orders union v`	20
$Q_{3.2}$	`s <- read();` `orders <- {x in orders st x.suppno != s}`	20
$Q_{3.3}$	`s<-read();` `print(` `{[x.partno, x.descrip, y.price]` `: x in parts, y in quotes st x.partno = y.partno and y.suppno = s})`	10
$Q_{3.4}$	`p<-read(); d<-read();` `print(` `{[y.suppno, y.orderno, x.partno, x.descrip]` `: x in parts, y in orders, z in quotes` `st x.parnto=y.partno and y.suppno=z.suppno and z.price < p and y.date = d})`	2

Figure 6: Queries $Q_{3.1}$ through $Q_{3.4}$ for Schema 3.

- V_2 is a partition of **Orders** with **orderno**, **suppno**, **date**, and **partno** in one file, and the remaining attributes in the other.
- V_3 is a partition of **Quotes** with **suppno** and **price** in one file, and the remaining attributes in the other.

- Indexes

	set	attributes
I_1	Orders	orderno
I_2	Orders	suppno
I_3	Quotes	suppno
I_4	Parts	partno
I_5	Orders	date
I_6	Quotes	suppno, price

- Orderings

	set	attributes
O_1	Orders	suppno
O_2	Quotes	suppno, partno
O_3	Parts	partno
O_4	Orders	date
O_5	Quotes	suppno, price

The first F_{current} is $\{I_5, O_4, V_2, I_6, O_5, V_3, I_4, O_3\}$, with costs as shown in Table 2; the initial F_{target} is $\{I_2, O_1\}$. In the compromise of these two organization we arrive at $\{I_2, O_1, V_2, I_6, O_5, V_3, I_4, O_3\}$, as the second F_{current}. This is compromised with $\{I_3, O_2, V_1, I_4, O_3\}$ to yield the solution, $\{I_2, O_1, V_2, I_6, O_5, V_1, I_4, O_3\}$.

Table 2: Costs for Workload of Schema 3.

query	cost on initial F_{current}	cost on second F_{current}	cost on solution
$Q_{3.1}$	8.5	7.2	7.2
$Q_{3.2}$	2014	25	25
$Q_{3.3}$	439	439	431
$Q_{3.4}$	129	262	262
workload	45098	5558	5478

7 Summary and Future Research

We have presented an extensible method for physical database design. In its first phase, the method relies on rules to generate sets of useful features for each query in the workload. In its second phase, the method relies on a more general notion of feature-set compromise in a search for a design with low aggregate cost. We give conditions under which a best feature set is a subset of the union of useful features for the individual queries. We also specify conditions to avoid redundant estimation of QEP costs for similar feature sets.

We then propose a specific heuristic, binary compromise, for the second phase. In addition, by framing the physical database design problem as a search among subsets of the complete feature set of workload, we hope to have eased the effort of framing other heuristics.

To illustrate our approach we presented three examples, including one that involves a join in a $\neg$1NF schema, and another example that involves two updates, a two-way join, and a three-way join.

Future research will evaluate prototype data structure selection software against more complex schemas and larger workloads. In the longer term, we plan to extend the application of feature-set compromise to data structure selection for distributed data bases, recursive queries, and for main memory data, and include them in a compiled implementation of BULK.

7.0.1 Acknowledgments

This work was supported with funds provided by ONR grants N00014-90-J-1110 and N00014-91-J-1472, by NSF grant IRI-89-01699, and by DARPA under contract N00014-85-C-0710 with ONR. Thanks to the staff of Software Options whose challenging questions helped clarify this material, and to Walter G. Morris and the referees for carefully reading earlier versions of this report and providing much thoughtful criticism.

7.0.2 References

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

[2] F. Bancilhon. Object-oriented database systems. Technical Report 16-88, GIP Altaïr, Jan. 1988.

[3] E. Barcucci, R. Pinzani, and R. Sprugnoli. Optimal selection of secondary indexes. *IEEE TOSE*, 16(1):32-38, Jan. 1990.

[4] D. Comer. The difficulty of optimum index selection. *ACM TODS*, 3(4):440-445, Dec. 1978.

[5] D. W. Cornell and P. S. Yu. An effective approach to vertical partitioning for physical design of relational databases. *IEEE TOSE*, 16(2):248-258, Feb. 1990.

[6] 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.

[7] K. R. Dittrich. Object-oriented database systems: The notion and the issues. In *Proc. Int'l Workshop on Object-Oriented Database Systems*, pages 2-4, Sept. 1986.

[8] S. Finkelstein, M. Schkolnick, and P. Tiberio. Physical database design for relational databases. *ACM TODS*, 13(1):91-128, Mar. 1988.

[9] J. C. Freytag. A rule-based view of query optimization. In *SIGMOD*, pages 173-180, May 1987.

[10] J. C. Freytag and N. Goodman. On the translation of relational queries into iterative programs. *ACM TODS*, 14(1):1-27, Mar. 1989.

[11] G. Graefe and D. J. DeWitt. The EXODUS optimizer generator. In *SIGMOD*, pages 160-172, 1987.

[12] M. Jarke and J. Koch. Query optimization in database systems. *Computing Surveys*, 16(2):111-153, June 1984.

[13] A. Kemper, P. C. Lockemann, and M. Wallrath. An object-oriented database system for engineering applications. In *SIGMOD*, pages 299-310, May 1987.

[14] V. Linnemann *et al.* Design and implementation of an extensible database management system supporting user defined data types and functions. In *VLDB*, pages 294-304, 1988.

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

[16] 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.

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

[18] J. A. Orenstein, S. K. Sarin, and U. Dayal. Managing persistent objects in Ada: Final technical report. Technical Report CCA-86-03, Computer Corporation of America, May 1986.

[19] S. Rozen and D. Shasha. Using a relational system on Wall Street: The good, the bad, the ugly, and the ideal. *CACM*, 32(8):988-994, Aug. 1989.

[20] H.-J. Schek and M. H. Scholl. The relational model with relation-valued attributes. *Information Systems*, 11(2):137-147, 1986.

[21] J. W. Schmidt. Some high level language constructs for data of type relation. *ACM TODS*, 2(3):247-261, Sept. 1977.

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

[23] P. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *SIGMOD*, pages 23-34, 1979.