The GMAP: A Versatile Tool for Physical Data Independence

Odysseas G. Tsatalos* Marvin H. Solomon* Yannis E. Ioannidis†

Computer Sciences Dept., Univ. of Wisconsin-Madison, WI 53706
{odysseas,solomon,yannis}@cs.wisc.edu

Abstract

Physical data independence is touted as a central feature of modern database systems. Both relational and object-oriented systems, however, force users to frame their queries in terms of a logical schema that is directly tied to physical structures. Our approach eliminates this dependence. All storage structures are defined in a declarative language based on relational algebra as functions of a logical schema. We present an algorithm, integrated with a conventional query optimizer, that translates queries over this logical schema into plans that access the storage structures. We also show how to compile update requests into plans that update all relevant storage structures consistently and optimally. Finally, we report on experiments with a prototype implementation of our approach that demonstrate how it allows storage structures to be tuned to the expected or observed workload to achieve significantly better performance than is possible with conventional techniques.

1 Introduction

Physical data independence is usually described as the ability to write queries without being concerned with how the data are actually structured on disk. In current database systems (DBMSs), however, queries are tied to logical constructs such as relations, class extents, or object sets, that closely track the physical organization of data. In a relational database, for example, each relation is usually stored as a file, perhaps with a primary index. The database administrator can improve performance by adding secondary indices or by specifying the clustering of files, but more extensive improvements require modifying the logical schema, for example by de-normalizing tables. Such modifications necessitate rewriting queries and thus physical data independence is lost.

Our goal is to improve physical data independence by decoupling physical decisions such as clustering and replication from the logical data model, so that the physical organization can be altered without changing the logical schema or queries written against it. A more subtle benefit is that it places a wider range of possibilities for data organization at the disposal of the database administrator. For example, the fact that the data are described in a traditional normalized relational schema should not preclude a replicated, nested physical organization, if that organization would achieve better performance for the anticipated mix of queries and updates.

Assume that data are stored in files of records, possibly implemented by an index structure such as a B^+-tree. Instead of requiring a one-to-one correspondence between logical data constructs and physical storage structures (e.g., relation $\leftrightarrow$ file), we allow the contents of each file to be defined as a function of the logical schema, specified by a restricted relational-algebra expression. We call the combination of a file and its definition a *gmap* (pronounced gee-map and an acronym for Generalized Multilevel Access Path.) In the simplest cases, gmaps correspond to traditional storage structures such as an unordered file of the tuples in a relation or an index on that file. Gmaps, however, can also be used to partition the database vertically and horizontally and add multiple access paths, generalizing path indices. Since gmaps are allowed to contain overlapping data, they can also capture redundant storage structures. Gmaps are invisible at the logical layer, so their definitions affect only the performance of queries and not their semantics.

In this paper, we restrict both gmap definitions and queries to project-select-join (psj) expressions over a simple semantic data model. We demonstrate that such expressions are powerful enough to express most conventional storage structures, as well as more "exotic" techniques such as path indices [2, 16], field repli-

*Partially supported by the Advanced Research Project Agency, ARPA order number 018 (formerly 8230), monitored by the U.S. Army Research Laboratory under contract DAAB07-91-C-Q518

†Partially supported by grants from NSF (IRI-9113736, IRI-9224741, and IRI-9157368 (PYI Award)) DEC, IBM, HP, AT&T, and Informix.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

Proceedings of the 20th VLDB Conference
Santiago, Chile, 1994

cation [11, 23], and more. We present an algorithm to translate user queries, expressed as psj-queries over structures in the logical schema, into relational expressions over the gmaps. We also show how this translation can be integrated into a conventional query optimizer.

One of the benefits of our approach is that gmaps may store redundant data to improve the performance of queries. Thus, updates may need to change multiple gmaps in a consistent manner. We show how a simple modification of the query translation algorithm can produce plans to perform these updates. We also demonstrate how this flexibility can be used in several other areas, e.g., acceleration of bulk loading of the database and acceleration of updates of complex objects.

All of the algorithms presented in this paper have been implemented in a prototype system. We report on experiments with a test database that illustrates that for a plausible mix of queries and updates, our techniques allow the physical representation to be tuned to provide better performance than what could be achieved through standard relational or object-oriented methods.

2 The Gmap Definition Language

In this section, we introduce our data model and the corresponding data definition language. The data definition language (DDL) has two parts, the *logical DDL*, which defines the logical schema capturing the conceptual organization of the data, and the *physical DDL*, which defines the storage structures containing the data that instantiate the logical schema. We present the model in two notations, a semantic one (resembling the ER model) and a formal relational one. The two notations are equivalent; the semantic notation is more intuitive as a user interface, but all of our algorithms manipulate the relational forms of schemas.

2.1 The Logical Data Definition Language

In the semantic notation, schemas are displayed as graphs. Throughout this paper we will illustrate our approach with an example database describing a university and its personnel (see Figure 1).

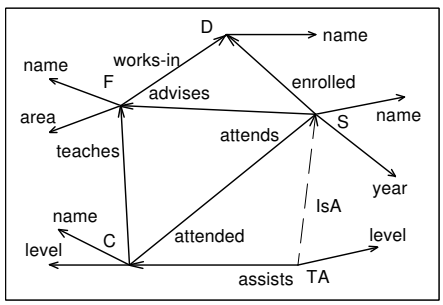

Figure 1: The logical schema

Nodes in this graph represent domains and solid edges represent relationships between them. Leaves represent primitive domains such as integers, character strings, or real numbers. Internal nodes represent domains populated with identity surrogates (tuple or object identifiers). In our example schema, these domains are `Dept` (department), `Faculty`, `Student`, `Course`, and `TA` (teaching assistant). To reduce clutter in the figures, these domain names are abbreviated to their initials. Functional dependencies are indicated by arrow heads. Inclusion dependencies (formally defined in Section 3) can also be expressed but are not shown for simplicity. IsA associations are denoted by dashed arcs pointing to the supertype. For our purposes, they are simply relationships with certain functional and inclusion dependencies implied by default. A name of the form $D.d$ is used to denote both a primitive domain and its relationship to an internal domain. For example, `Course.level` names both a primitive domain of integers and its relationship to the `Course` domain.

In the relational form of the data model, each edge of a schema graph from domain A to domain B is represented as a binary relation with attributes A and B. Because of this correspondence, we often use the term "attribute" to refer to domains (nodes in the graph) and "base relation" to refer to relationships (edges). Because our algorithms operate on the (binary) relational form of the schema, they apply to any semantic model that can be represented by binary relations with functional and inclusion dependencies.

2.2 The Physical Data Definition Language

In our system, all physical storage structures are defined as gmaps. A gmap consists of a set of records (the *gmap data*), a query that indicates the semantic relationships among the attributes of these records (the *gmap query*), and a description of the data structure used to store the records (the *gmap structure*). Although the actual database stores gmap data rather than the base relations, the gmap data may be thought of as the result of running the gmap query on the base relations.

Gmap queries are expressed in a simple SQL-like language. For example, the gmap

```
def_gmap cs_faculty_by_area as btree by
  given Faculty.area
  select Faculty
  where Faculty works_in Dept and
        Dept = cs_oid
```

stores a set of pairs containing `Faculty` identifiers and the corresponding area names. Only faculty in the Computer Sciences department (identified by the constant `cs_oid`) are included. The gmap structure is a B^+-tree indexed by `Faculty.area`. The entire `by` clause defines the gmap query. Attributes following the `given` and `select` keywords are called *input* and *output* attributes, respectively, and the selection `Dept = cs_oid` is called a *restriction*. Input attributes form the search key for gmap structures that allow associative access.

The gmap query can also be expressed graphically as a subgraph of the schema graph called the *query*

graph (see Figure 2). Shaded edges correspond to relationships explicitly mentioned in the `where` clause or implicitly mentioned as part of primitive attribute names. Input attributes are indicated by small arrows, and output attributes are indicated by double circles around nodes. Restrictions are described as annotations on the corresponding nodes.

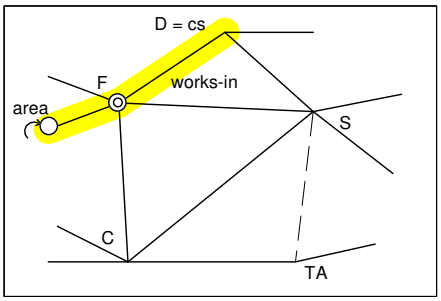

Figure 2: A collection based index

Each query expressible in this language is equivalent to a restricted project-select-join (psj) query on the relational form of the logical schema :

$$Q = \pi_A \sigma_S (R_1 \bowtie R_2 \bowtie \cdots \bowtie R_n).$$

In this example,

$$Q = \pi_{\texttt{F,F.area}} \sigma_{\texttt{D=cs_oid}} (\texttt{F.area} \bowtie \texttt{works_in}).$$

Expressible queries obey three restrictions:

- They are range-restricted, i.e., all attributes in S and A are attributes of the relations R_i,
- selections are conjunctions of comparisons ($=, >, \geq, <, \leq$) between attributes and constants, and
- joins are natural, i.e., only attributes with the same name are joined and all attributes maintain their name in the result.

In the rest of the paper, we use the term *psj-query* to refer to a query that conforms to these restrictions.

2.3 The Query Language

We often use the term *logical query* to refer to queries posed on the logical schema. We currently require logical queries to be written in the same language we use for gmap queries. That is, they must be restricted psj-queries. In addition, each query must be translatable into a psj-query over gmaps or projections of them. Thus, we do not handle cases where logical queries need to be translated into unions or arbitrary sequences of psj-queries. Note that translated logical queries involve relations with arbitrary arity (the gmap data), while gmap queries involve binary relations only (corresponding to relationships).

2.4 Examples

Gmaps can be used to define arbitrary physical representations, including those of a conventional normalized relational database, an object-oriented database, or any combination of the two.

To illustrate the object-oriented approach, suppose we want to cluster together all information about each faculty member. Given the object identifier of a `Faculty` object, we should be able to retrieve personal information as well as the object identifiers of the faculty member's department, advisees, and courses taught. A gmap that meets these specifications may be defined and drawn as in Figure 3.

```
def_gmap faculty_data as heap by
  given Faculty select Student, Dept,
     Course, Faculty.area, Faculty.name
  where Faculty works_in Dept and
     Faculty advises Student and
     Faculty teaches Course
```

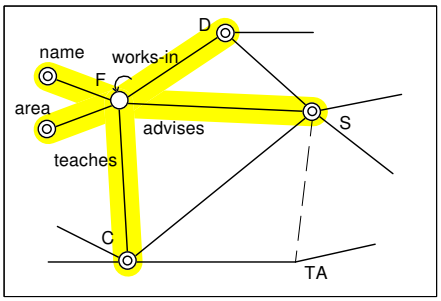

Figure 3: The Faculty class extent

A secondary index in a relational system can also be defined easily in our language. For example, an index on the faculty area is defined as in Figure 4.

```
def_gmap faculty_index_on_area as btree by
  given Faculty.area select Faculty
```

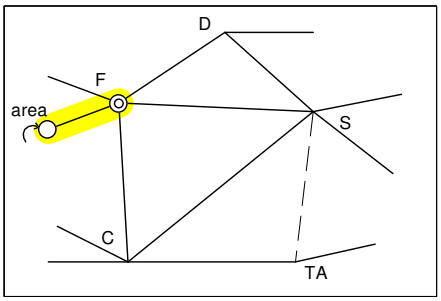

Figure 4: A Faculty index on "area"

Note that the index is not defined in terms of the previous gmap, as would be the case in a relational database, but in terms of the logical schema.

In the `faculty_data` example, it might be desirable to include in a faculty member's record the department name in addition to the department id, because for example, the department name is frequently printed along with the name of the faculty member. The department name is in this case a *nested attribute* of the `Faculty` domain. This essentially implements *field replication* [11, 23], which has been shown to offer several advantages. The only change necessary is to add "`Dept.name`" to the `select` clause.

In the previous examples, the gmap data included all `Faculty` instances. However, there are cases where we frequently access only some instances of a domain. Object-oriented systems that store instances in explicit

collections rather than class extents [5, 16, 18] allow the creation of collection indices, which provide fast access paths only to the subsets of the domains that are included in the collection. Our gmap definition language is powerful enough to express such indices by using restrictions. An example of this technique was shown in Figure 2 above.

Many more indexing schemes can be specified using gmaps, e.g., nested indices, replication of non-functional nested attributes, and indices with composite keys where each key component is a path. A complete taxonomy of existing indexing schemes and other advanced storage structures that can be defined by using gmaps is presented elsewhere [24].

3 Query translation

Before presenting the actual translation algorithm, we first introduce some additional notation and definitions, and also discuss two auxiliary problems that arise as part of query translation.

3.1 Notation

For convenience, we use a triplet $\langle Q_r, Q_s, Q_p \rangle$ as an alternative way of representing a psj-query Q. The set Q_r contains the joined binary relations, the set Q_s contains the selection predicates, and the set Q_p contains the projected attributes. We call the set Q_p the *query target*, and its members *target attributes*. Given a query Q, we frequently deal with its part that includes only relations in a set $\mathcal{R}$, denoted $Q[rel \in \mathcal{R}]$. Similarly, the subset of Q_s that mentions only attributes in a set $\mathcal{A}$ is denoted $Q_s[attr \in \mathcal{A}]$. The set of a attributes in a set of relations $\mathcal{R}$ is denoted $A(\mathcal{R})$.

3.2 Definitions

The *natural join* of two psj-queries P and Q, denoted $P \bowtie Q$, is the natural join of their result relations. The *add-join* of two psj-queries P and Q, denoted $P \oplus Q$, is the psj-query $P \oplus Q = \langle P_r \cup Q_r, P_s \cup Q_s, P_p \cup Q_p \rangle$. The add-join differs from the natural join in that the projections of P and Q are performed *after* all joins of base relations rather than being interleaved with them.

Let $R_1(\alpha, \beta), R_2(\beta, \gamma)$ be two relations with a common attribute β. An *inclusion dependency* from R_1 to R_2 exists, denoted $R_1.\beta \subseteq R_2.\beta$, if every value of β in R_1 appears also in R_2.

Multivalued dependencies are not meaningful in binary relations, but are important in n-ary results of psj-queries, such as the gmap data. Since lossless join decompositions imply multivalued dependencies and gmap data are the result of joining base relations, we can infer certain multivalued dependencies for each gmap. An algorithm that generates a cover of the multivalued dependencies that hold on the gmap relation is described in a longer version of this paper [25]. Multivalued dependencies are important because they help determining the pieces of the gmap relation that can be used to answer a user query.

3.3 Query Equivalence

When translating a logical query into a query over gmaps, we often need to test the equivalence of psj-queries. Two psj-queries Q_1 and Q_2 are equivalent, denoted $Q_1 \equiv Q_2$, if they produce the same result for any valid instance of the database schema. Equivalence testing of arbitrary conjunctive queries, even without taking into account any dependencies, is NP-complete [1, 7]. On the other hand, we can efficiently compare two psj-queries syntactically to see if they are identical (up to trivial differences such as the ordering of the join terms). This is a sufficient condition for equivalence, which we use in our translation algorithm. We are also interested in two special cases of equivalence testing, where psj-queries of specific forms are involved and various types of dependencies are taken into account. These are discussed in the next two subsections, where sufficient conditions for equivalence are provided.

3.3.1 Coverage

A query Q *covers* a set of relations $\mathcal{R}$ if

$$Q[rel \in \mathcal{R}] \equiv \pi_{A(\mathcal{R})}(Q) \tag{1}$$

For example, if $R_1(\alpha, \beta) \equiv \pi_{\alpha,\beta}(R_1(\alpha, \beta) \bowtie R_2(\beta, \gamma))$ then $R_1 \bowtie R_2$ covers $\{R_1\}$. In general, the result of the left-hand side query is a superset of that of the right-hand side query. When (1) holds, the part of the query that involves relations not in $\mathcal{R}$ (relation R_2 in our example) has no effect on $\pi_{A(\mathcal{R})}(Q)$, in the sense that it does not filter out any tuples produced by the rest of the query. An algorithm that implements a sufficient condition for testing coverage is presented elsewhere [25]. The algorithm makes use of the inclusion dependencies of the schema. Its running time is linear in the size of Q and quadratic in the number of inclusion dependencies between relations in Q.

3.3.2 Natural join vs. Add-Join

In general, if P and Q are two psj-queries, $P \oplus Q \subseteq P \bowtie Q$. However, in the presence of certain integrity constraints $P \oplus Q \equiv P \bowtie Q$. For example, suppose $R(\alpha, \beta), S(\beta, \gamma)$ are two relations, P is the query $\pi_{\alpha\beta}(R) = R$, and Q is the query $\pi_{\alpha\gamma}(R \bowtie S)$. Then $P \oplus Q$ is $\pi_{\alpha\beta\gamma}(R \bowtie R \bowtie S) = R \bowtie S$, which is not in general the same as $R \bowtie \pi_{\alpha\gamma}(R \bowtie S)$. If, however, β is functionally determined by α in R (that is, α is a key for R), the two joins are equal. Intuitively, the information "lost" by projecting away the β attribute in $P \bowtie Q$ can be completely recovered from the remaining a attributes.

Detecting when the natural join of two psj-queries is equivalent to a psj-query is very important in our query translation algorithm, since it allows us to rewrite the join of two gmaps (which are psj-queries) as a psj-query. The algorithm iteratively performs such rewritings in order to express the join of several gmaps as a psj-query, which is then checked syntactically for equivalence with the user query. As a sufficient condition for guaranteeing that the natural join of two

queries is a psj-query, we test if it is equivalent to the add-join of the queries in question. An algorithm that implements a sufficient condition for testing this equivalence is presented elsewhere [25]. The algorithm makes use of multivalued dependencies and of query coverage and its running time is quadratic in the size of queries and in the size of multivalued dependencies.

3.4 Query Translation Algorithm

Below we present an algorithm to translate a logical psj-query into a query over gmaps. To simplify the presentation, we omit most considerations of efficiency.

Algorithm 1 *Given a psj-query Q and a set of psj-queries $\mathcal{G}$, find subsets $\{G_1, \ldots, G_n\} \subseteq \mathcal{G}$, s.t.* $Q \equiv \pi_{Q_p}\sigma_{Q_s}(\pi_{A(Q_r)}G_1 \bowtie \cdots \bowtie \pi_{A(Q_r)}G_n)$

1. **let** $\mathcal{H} = \{G \in \mathcal{G}$ s.t. $G_p \cap A(Q_r \cap G_r) \neq \emptyset$ **and**
2. $G_s[attr \in A(Q_r)] \cup Q_s[attr \in G_p] =$
$Q_s[attr \in A(G_r)]$ **and**
3. G covers $Q_r\}$
4. **for each** subset $\{G_1, ..., G_n\}$ of $\mathcal{H}$ **do**
5. **let** $\mathcal{S} = \{\pi_{A(Q_r)}\sigma_{Q_s}G_1, \ldots, \pi_{A(Q_r)}\sigma_{Q_s}G_n\}$
6. **while** there is $G, H \in \mathcal{S}$ s.t. $G \bowtie H \equiv G \oplus H$
7. replace G and H in $\mathcal{S}$ by $G \bowtie H$
8. **if** $\mathcal{S} = \{\mathcal{Q}'\}$ where $\pi_{Q_p}(Q') = Q$ accept current subset of $\mathcal{H}$ as a solution

□

The algorithm first narrows down its search space to gmaps that have something to do with the query (lines 1-3). More specifically, a gmap must have at least one relation in common with the query, with at least one attribute of the relation included in the gmap result (line 1); the query selections on attributes of the gmap relations must be either on the target attributes of the gmap (so that they can be applied on them) or must be identical to selections that the gmap itself has (line 2); and the gmap must cover the common relations with the query (otherwise, the gmap will not have all the information needed by the query) (line 3).

Each possible subset of the relevant gmaps (line 4) gives rise to a single candidate translation (assuming that selections are always pushed through the joins):

$$\pi_{Q_p}(\pi_{A(Q_r)}\sigma_{Q_s}G_1 \bowtie \cdots \bowtie \pi_{A(Q_r)}\sigma_{Q_s}G_n). \quad (2)$$

The rest of the algorithm tests whether or not this query expression is equivalent to the given logical query. Each join operand in the above equation is a projection and a selection on a gmap. Since we verified earlier (line 2) that the query selections can be pushed through the gmap projections, the join operands are psj-queries. The algorithm tries to express their join as a psj-query as well. The join operands are scanned (line 5) and any pair whose natural join is equivalent to their add-join (line 6) is replaced by a single join operand which is again a psj-query (line 7). The set of join operands thus keeps reducing. At some point, we can no longer reduce the set either because there is just one psj-query left or because there is no pair that satisfies the equivalence test (line 6). In the former case, the remaining psj-query is equivalent to the initial join expression (2) after performing one final projection step (π_{Q_p}) and can be syntactically checked for equivalence (line 8) with the logical query. In the latter case, the subset chosen in line 4 is rejected. Algorithm 1 satisfies the following.

Proposition 1 *Given a psj-query Q and a set of psj-queries $\mathcal{G}$, for any subset $\{G_1, \ldots, G_n\} \subseteq \mathcal{G}$ generated by Algorithm 1, the following holds:*

$$Q \equiv \pi_{Q_p}\sigma_{Q_s}(\pi_{A(Q_r)}G_1 \bowtie \cdots \bowtie \pi_{A(Q_r)}G_n)$$

Because there are exponentially many subsets of $\mathcal{H}$, the whole algorithm runs in exponential time. However, checking if a given subset of gmaps can form a solution (lines 5-8) takes polynomial time. In the next section, we show how we can run the algorithm in conjunction with a conventional optimizer to avoid enumerating all subsets.

An example may help illustrate the algorithm. Consider a query `Q` that asks for all 500-level courses, the names and department id's of students attending them:

```
def_query Q by select Student.name, Dept
  where Student attends Course and
      Student enrolled Dept and Course.level = 500.
```

The database consists of three gmaps: an index `G1` from the names of students to their departments, an index `G2` from the names of students to courses they attend and the levels of those courses, and an index `G3` from a course-level to courses at that level, together with the departments that supply students to them.

```
def_gmap G1 as btree by
  given Student.name select Dept
  where Student enrolled Dept
def_gmap G2 as btree by
   given Student.name select Course, Course.level
   where Student attends Course
def_gmap G3 as btree by
  given Course.level select Dept, Course
 where Student attends Course and Student enrolled Dept.
```

All three gmaps are relevant to the query (they pass the tests of lines 1–3). For example `G1` can provide values for two attributes needed by the query (`Dept` and `Student.name`) so it passes line 1, it trivially satisfies the constraint test of line 2, and it covers the relations `enrolled` and `Student.name`, so it passes the test on line 3.

The algorithm considers subsets of the relevant gmaps `G1, G2, G3`. Consider, for example, the subset `{ G1, G2 }`. The candidate solution corresponding to this combination is the natural join of `G1` and `G2` followed by a selection for `Course.level = 500` followed by a projection. The loop of lines 6 and 7 will be executed once to check whether `G1` $\bowtie$ `G2` = `G1` $\oplus$ `G2`. This test will fail unless `Student.name` functionally determines `Student`; otherwise two tuples that join on the `Student.name` need not join on their `Student` id as well. If `Student.name` functionally determines `Student`, then the join on the `Student` id is irrelevant: we can project out that attribute before performing

the join, which implies that the add-join is equivalent to the natural join (line 6). Eventually line 8 of Algorithm 1 will conclude that the candidate solution is a correct one.

Following the same process, the algorithm would reject the subsets { `G1, G3` } and { `G2, G3` }, because the necessary multivalued dependencies do not hold. However, the combination { `G1, G2, G3` } is a correct solution. During the course of the loop of lines 6 and 7, the algorithm will test all pairs of gmaps in this subset to check whether or not their add-join is equivalent to their natural join. As we saw, all the pairs will fail except `G1` $\oplus$ `G2`. In the next iteration, the pair (`G1` $\oplus$ `G2`, `G3`) is considered and it is confirmed that its add-join is equivalent to its natural join.

Interestingly, the solution using all three gmaps is likely to be more efficient than the one that uses only `G1` and `G2`, because the index on `Course.level` in `G3` will accelerate the selection in the query. The next section shows how a gmap-aware optimizer identifies and prunes the inferior plan.

4 Integration with a Query Optimizer

The presentation of Algorithm 1 emphasizes clarity at the expense of efficiency. It implies that all subsets of the gmaps are enumerated in random order and each is tested to see if it provides a solution to the equation. All subsets that pass the test are feasible plans. The version of the algorithm that is actually implemented by our system is considerably more sophisticated. It is integrated with a conventional dynamic-programming query optimizer [21], which controls the order in which subsets are evaluated and uses cost information and intermediate results to prune the search space.

A conventional dynamic-programming optimizer iteratively finds optimal access plans for increasingly larger parts of a query. We follow these steps in more detail, showing at each step what needs to be changed for a gmap-equipped database (Table 1). We then identify the pieces of Algorithm 1 that correspond to these changes. In what follows, for simplicity, we avoid any discussion of "interesting orders" [21]. We also use the term *complete solution* to refer to a gmap access plan (i.e., a specific sequence of joins, together with the method used for each join) that is equivalent to the logical query, and *partial solution* for a gmap access plan that could potentially be enhanced to become a complete solution. A partial solution does not necessarily have to be a psj-query; it may be that no reordering of its joins makes them equivalent to add-joins.

Like a conventional optimizer, the gmap optimizer only attempts to join a partial solution with gmaps that share projected attributes with it, thus avoiding Cartesian products. Each step in the gmap optimizer corresponds to part of Algorithm 1. Step (a1) of the first iteration corresponds to lines 1-3 of Algorithm 1; it finds all gmaps that are relevant to the query. The remaining steps of all iterations represent the rest of the algorithm. Moving from iteration to iteration and step (c) of each iteration corresponds to a specific implementation of line 4, where all subsets of relevant gmaps that are not pruned on the way are regularly explored in increasing size. After the first iteration, step (a1) forms the joins of these subsets (solutions) and step (a2) corresponds to lines 6-8, where these solutions are examined for completeness. Step (a2) can be implemented incrementally taking into account the results of earlier iterations on smaller partial solutions.

Note that step (b) of each iteration has no counterpart in the translation algorithm because it deals only with pruning the search space and not with translation. Implementing this step is not straightforward because it involves not only the cost but also the contribution of solutions to the query. Contributions of partial solutions can be compared on the basis of their pieces that correspond to psj-queries and the set of attributes in their result. When each piece of a partial solution has subsets of the relations and projected attributes of a piece of another partial solution, then the former contributes less and can therefore be removed from further consideration if it also has a higher cost. Query signatures, an encoding of the names of all the relations used by the query, can be used to perform these comparisons efficiently [8].

It is interesting to see how the new algorithm behaves when it is given a set of gmaps that represents a traditional relational physical schema. Assume for example that one gmap is a file containing the extent of the `Faculty` relation with all associated attributes,

```
def_gmap faculty_relation as heap by
  given Faculty select Faculty.name, Faculty.area, Dept
  where Faculty works_in Dept
```

while another gmap is a secondary index on the `Faculty.area` field,

```
def_gmap faculty_index_on_area as btree by
  given Faculty.area select Faculty.
```

Assume that the logical query requests the names of all faculty in the database area. During the first iteration both gmaps are considered. Scanning the relation extent would be far more expensive than accessing the index, but the two solutions are not comparable. Since the index simply returns `Faculty` ids, it is not adequate to answer the query, while the extent is. During the second iteration, the index (the only partial solution left) is considered for a join with the `Faculty` extent. The join would be less expensive than scanning the `Faculty` extent while both plans are equivalent to the logical query. Thus, the solution found during the first iteration is eliminated in the second. At that point, there is no partial solution left and the algorithm ends. This example demonstrates that access plans that are pruned in the conventional optimizer are also pruned in its enhanced version. However, since an access plan considered at iteration n in the old version may combine more than n gmaps, it may be considered at a later iteration in the new version, thus delaying potential prunings. In general, we expect the performance of the modified optimizer to be similar to the

Table 1: Step by step comparison of a conventional optimizer vs. one designed for a gmap-equipped database

Conventional optimizer	**Gmap optimizer**
Iteration 1	**Iteration 1**
For each query relation:	
a) Find all possible access paths.	**a1)** Find all gmaps that are relevant to the query. **a2)** Distinguish between partial and complete solutions among them.
b) Compare their cost and keep the least expensive.	**b)** Compare all gmaps among themselves. If one has neither greater contribution to the query than another nor a lower cost, prune it.
c) If the query involves only one relation, stop.	**c)** If there are no partial solutions, stop.
Iteration 2	**Iteration 2**
For each query join:	
a) Consider joining the relevant access paths found in the previous iteration using all possible join methods.	**a1)** Consider joining all partial solutions found in the previous iteration with another gmap using all possible join methods. **a2)** Distinguish partial and complete solutions among resulting joins.
b) Compare the cost of the resulting join plans and keep the least expensive.	**b)** Compare all generated solutions among themselves and to any earlier solution. If any single gmap or gmap combination has neither a greater contribution to the query than another nor a lower cost, prune it.
c) If the query involves only two relations, stop.	**c)** If there are no partial solutions, stop.
Iteration 3	**Iteration 3**
...	...

performance of the original one. Our experience obtained by using the optimizer for the examples shown in Section 8 supports the prediction.

5 Update propagation

Relational systems mitigate dependencies between the logical and the physical schema through the use of stored queries called *views*, and users express their queries in terms of the views. With this approach, the logical schema becomes a (relational) function of the physical schema. View updates, however, are difficult or impossible to support. The usual solution is to require updates to be expressed in terms of the underlying schema.

Our approach is the inverse. We define the physical structures as functions of the logical schema. Although query translation is more complicated, we have shown above that it is still possible, and it can be integrated with the optimization stage of a conventional system adding little overhead to the preparation of query plans and no overhead to the execution of those plans. Updates, however, are much simpler. Translating them into the physical schema turns into the materialized view maintenance problem, which accepts simple solutions.

As discussed elsewhere [3], propagating updates into materialized views requires the execution of queries over the base relations and the inserted or deleted tuples. However, here we do not necessarily have the base relations stored, and the actual data are replicated in many places. In this section, we illustrate how the query translation algorithm described above can be adapted to translate an update request over the logical schema into the corresponding physical plan. Our algorithm produces optimal update plans, using existing gmaps to accelerate update propagation where possible.

5.1 Specifying Updates

Insertions are specified by supplying a query (the *update query*) and a set of tuples to be inserted (the *update data*), corresponding to the target attributes of the query. The database must be updated in such a way that the change in the results of the query between the original and updated database is precisely the set of tuples in the update data. Deletions are defined similarly, with the roles of "original" and "updated" database reversed. Note the difference from the query used when specifying updates in SQL-like languages, in which the query is used to generate the update tuples. The query here describes only the "schema" of the tuples. Since the update data can be the result of another query, no generality is lost.

For example, students can become enrolled in courses by supplying a set of $(StudentId, CourseId)$ pairs and the update query

```
def_query enroll_student as
  select Student, Course where Student attends Course.
```

Allowing arbitrary queries to be used in update gmaps would re-introduce all the problems of updating through views. Therefore, we disallow projections and explicit selections from update queries and impose a few other restrictions described in more detail elsewhere [25]. Although the semantics of the update query depends only on the logical schema, its validity may depend on the choice of gmaps used to define the physical schema. For example, the physical schema must have sufficient "information capacity" to hold the inserted data [17]. These issues are not addressed in this paper.

5.2 The Algorithm

Given an update query U, a set ΔU of tuples, and a gmap G in the physical schema, we need to find the change ΔG in the value of G corresponding to the change ΔU in the value of U. Suppose we find a collection of psj-queries $H_1, \ldots, H_m$ that are invariant under changes to U such that

$$G = \pi_{G_p}\sigma_{G_s}(U \bowtie H_1 \bowtie \cdots \bowtie H_m).$$

Let G_{prev} be the value of G before the update and G_{new} the value after. Then

$$\begin{aligned} G_{new} &= \pi_{G_p}\sigma_{G_s}((U + \Delta U) \bowtie H_1 \bowtie \cdots \bowtie H_m) \\ &= \pi_{G_p}\sigma_{G_s}(\Delta U \bowtie H_1 \bowtie \cdots \bowtie H_m) + G_{prev} \end{aligned}$$

or equivalently

$$\Delta G = \pi_{G_p}\sigma_{G_s}(\Delta U \bowtie H_1 \bowtie \cdots \bowtie H_m). \qquad (3)$$

In other words, the updates to G can be found by evaluating the right-hand side of (3). As shown elsewhere [25], we can use Algorithm 1 to find $H_1, \ldots, H_m$. Here, we illustrate the algorithm with an example.

Consider the update query `enroll_student` presented earlier

```
def_query enroll_student as select Student, Course
  where Student attends Course.
```

Assume that our database consists of two gmaps, one that maps faculty members to the courses they teach,

```
def_gmap FC as btree by given Faculty select Course
  where Faculty teaches Course,
```

and one that records the students and teacher of each course,

```
def_gmap CFS as btree by
  given Course select Faculty, Student
  where Faculty teaches Course and
     Student attends Course.
```

To propagate the update to the database, we consider each database gmap separately. The gmap `FC` is not affected by the update since it has no common relations with the update query. The updates to gmap `CFS` depend both on the update data and on the existing contents of `FC`. We need to enhance the *(CourseId, StudentId)* pairs in the update data with the faculty members who teach the courses before they are added to `CFS`. The algorithm constructs the tuples to be inserted by considering the part of the gmap that is not affected by the update, i.e., the query

```
def_query Q by select Course, Faculty
  where Faculty teaches Course,
```

and trying to find a translation for it. `Q` can definitely be answered by using gmap `FC`, and thus the tuples to be inserted into `CFS` are found by joining the update gmap `enroll_student` and the gmap `FC`. Gmap `CFS` may not be used as the source of the needed information because `CFS` will not contain *(CourseId, FacultyId)* pairs for courses that do not yet have any students. Line 2 of Algorithm 1 tests whether `CFS` covers the relation `teaches`. As long as there is no inclusion dependency from `teaches` to `attends`, the `CFS` gmap will be rejected.

Equation (3) can be used for deletions as well as insertions, provided each gmap whose target does not functionally determine its other attributes maintains its data as a multiset (that is, it records duplicate insertions or a count of them).

6 Applications

In Section 2, we demonstrated how gmaps subsume the facilities of primary and secondary storage structures in conventional database systems. In this section, we outline a variety of other applications that use the integrated query translation-optimization engine for queries and updates.

6.1 Database Loading

Existing DBMSs often provide special features to support bulk loading of data. These facilities tend to be *ad hoc* and impose restrictions on the format of the imported data. The user must manually translate all imported data into files that match the primary storage structures and then load each one individually. If the imported data can be described as a psj-query on the logical schema, initial loading can be viewed as a special case of updates. The imported data files can be viewed as inefficient gmap structures. The "real" gmaps used to store the data permanently are loaded by running their queries against the imported files.

6.2 Accelerating Complex Structure Updates

Although path indices are useful for accelerating common queries, they are expensive to maintain. Previous proposals for the maintenance of complex access paths suggest using *ad hoc* techniques to accelerate expensive updates, such as internal links between instances in nested indices and field replication [23, 2] or links from instances to their collections [16, 18]. We can achieve the same effect by using simple gmaps along the paths that need to be traversed during the update propagation. Gmaps placed at points where expensive joins are performed act like join accelerators in the same way that internal links accelerate joins. These gmaps can accelerate not only updates to the structure concerned, but any other query or update to which they apply. If usage patterns change so that the cost of maintaining the accelerators exceeds their benefit, we can simply remove them. In the hardwired approach, we will always be stuck with the same machinery. Furthermore, our approach allows the user to place join accelerators at specific points. It is much harder to achieve this flexibility with the hardwired approach.

6.3 Other applications

Using update queries to describe the schema of the modified data facilitates the handling of complex objects. A complex object insertion can be described

as an update gmap whose query describes the complex object schema. If the inserted object consists of several tuples of data, the tuples are not inserted one at a time. Instead, the query plan treats the update gmap like any other physical data container and performs bulk operations. The result of the query plan execution is a set of tuples to be inserted into each physical storage structure. Any available bulk-loading interfaces to these structures can be exploited.

In Section 6.1, we discussed how to create a thin "veneer" over imported text files to make them behave like gmaps. This idea can be extended to support heterogeneous storage organizations by hiding their differences under the gmap abstraction. As long as the data contents of all these distinct sources can be described by psj-queries over a single logical schema, the query optimizer can translate logical queries into access plans over them. This strategy is similar to the ADMS handling of database interoperability [19, 20].

Another application of gmaps is to support cached data in transient main-memory data structures such as arrays or hash tables. If the contents of the structure can be described by a psj-query on the logical schema, it can be treated like a gmap.

7 Implementation

To verify the applicability and practicality of our algorithms and obtain a feeling for their performance, we built a prototype implementation of our system on top of SHORE [4]. SHORE is an object-oriented database system under construction at the University of Wisconsin. Logical schema definitions are parsed and stored in a logical-schema catalog. Physical storage structures are created from gmap definitions. The parsed gmap query is stored persistently in a second physical-schema catalog. The data organization, keys, and record format are also determined by the gmap definition. The gmap is created and populated by processing an update request.

We built a query processor using the algorithms in Sections 3 and 4. For all the examples presented in this paper, query translation added only a negligible overhead to the overall query cost. The query processor also contains hooks to support the update processor, as outlined in Section 5. The update processor accepts three lists of gmaps: update gmaps, target gmaps to be updated, and database gmaps that may be used to supply data for the updates. For simple updates, the first list contains just one gmap and the target and database lists each contain all of the gmaps in the physical-schema catalog. Other combinations of arguments support other applications described in Section 6.

We designed a simple common interface for storage structures, including operations to store and retrieve data and to make cost inquiries. We implemented this interface on top of existing SHORE facilities (B^+-trees and heaps) as well as Unix files (for importing and exporting data) and main-memory structures (for update gmaps). To support the use of the algorithm in Section 5 for deletions, we also modified the SHORE B^+-tree facility to maintain a count of duplicate insertions rather than rejecting them.

8 A Performance Demonstration

In this section, we describe experiments with a test database that illustrate that for a plausible mix of queries and updates, our techniques can provide better performance than either relational or object-oriented databases. As we observed in Section 2, gmaps can be used to describe the relations and the primary and secondary indices of relational databases, as well as the class extents, object sets, and path indices of object-oriented databases. Thus, we are able to use our system to simulate two "conventional" configurations, one based on a normalized relational design and one following a typical object-oriented database design. All of our results are reported in terms of counts of I/O operations, since absolute performance in "real" databases would be affected by a variety of implementation-dependent features that are beyond our control.

In the experiment, we used an extended version of the university database presented earlier. We populated one department with actual data describing the Computer Sciences Department at the University of Wisconsin—Madison and generated synthetic data for 99 more departments to create a database of reasonable size. While the actual database used for the experiment includes additional fields, for simplicity we only discuss the logical schema as presented in Figure 1. A few interesting parameters of the data are presented in Table 2.

Table 2: Parameters of the database

	Faculty	Students	Courses	TAs	Depts
Instances	5000	50000	10000	2000	100
KB/inst	1	0.8	1	0.85	3

8.1 The Workload

The workload contains multiple runs of eight queries and five updates. The actual number of runs was determined by various factors. We tried to maintain a balance between expensive queries and simple ones, hold the update load at about a third of the total load for most of the configurations, and make the relative frequencies as realistic for a university environment as possible. Tables 3 and 4 briefly describe each query and update. For example, the workload contains three runs of query Q6. The last two columns contain the total cost attributed to all runs of query Q6 when it is processed on two different configurations of the database: a relational one (Section 8.2) and an object-oriented one (Section 8.3). Note that each update inserts a single pair of values. For example, when a student adds a course (U3), the inserted pair would be *(student id, course id)*.

Table 3: Queries in the workload

No	Mix	Given	Find	Rel	OO
Q1	10	faculty name	field, dept	30	30
Q2	10	faculty name	courses taught	30	30
Q3	10	student name	year, dept, advisor	50	50
Q4	10	TA name	support level, course	40	40
Q5	3	faculty area	students advised	57	57
Q6	3	student name	courses, teachers	39	42
Q7	2	course name	students attending	84	80
Q8	1	dept name	courses taught	64	59

Table 4: Updates in the workload

No	Mix	Update Description	Rel	OO
U1	1	Faculty teaches a new course	5	7
U2	2	Faculty starts advising a student	10	14
U3	20	Student adds a course	160	120
U4	4	TA is assigned to a course	20	20
U5	6	Student enrolls in a department	24	24

To obtain I/O counts, we instantiated queries and updates using random values for the input parameters and the inserted pairs. Then, we used the actual plans obtained by our system to perform the query or update and recorded the size of the result as well as the sizes of all intermediate results used in joins. From these data, we calculated I/O counts using a simple model of the B^+-tree and the heap storage structures based on the assumptions that the page size is 8KB and that 4MB are used for buffers. The analytical approach was chosen in favor of measuring actual response times, since the underlying SHORE system was still under development.

8.2 Relational Design

The relational configuration follows a textbook translation of the logical schema into relations. We added secondary indices as needed, to improve joins and selections, and we clustered favorably the records in the heaps. The total database size for this configuration is 74MB. Running the full workload on that database costs 613 page I/Os, 64% caused by queries and 36% by updates. The exact contribution of all runs of each query and update in the total cost is shown in Tables 3 and 4.

8.3 Object-Oriented Design

The physical schema for the object-oriented configuration includes one extent file for each internal domain in the logical schema. The relationship `attends` is stored both as part of student objects and as part of the course objects, i.e., students contain pointers to all courses they attend, and courses contain pointers to the students attending them. This duplication allows efficient execution of queries Q6 and Q7. Each other relationship is stored as part of the domain closest to the edge label in Figure 1. The total database size for this configuration is 75MB. Running the full workload on the database costs 583 page I/Os, 68% caused by queries and 32% by updates. The last column of Tables 3 and 4 show the costs for all runs of each query and update.

8.4 Object-Oriented Design with Complex Access Paths

The previous two configurations do not use any complex access path such as path indices[1] or field replication. In this section, we consider the case of an object-oriented DBMS equipped with such access paths. For the given workload, field replication can be applied in three cases, as shown in Table 5. Each row of the table describes what is replicated, the additional space required in MB, the affected queries (no update is affected by these additions), and the savings that can be attributed to the replication for all runs of the affected query.

Table 5: Conventional complex path techniques

Replicate	In	Space	No	Saved
"Faculty.work_in.name"	Faculty	0.2	Q1	10
"Student.enrolled.name"	Student	2	Q3	10
"TA.assists.name"	TA	0.1	Q4	20
			Tot. saved	40

The rest of the physical schema is identical to the earlier object-oriented schema. The data replication adds 2MB of additional space bringing the total database size to 77MB. The new database offers a 7% performance gain over the previous one. It is interesting to note that the technique of field replication as originally described [11, 23] cannot be applied to any other field in our database because it is restricted to edges from classes to single-valued attributes. Similar restrictions also prohibit any useful application of path indices in our database. The restrictions are there for a reason: the implementation of these access paths and the task of propagating updates would be far more complex without them.

8.5 A Configuration with Gmaps

In this section, we consider our approach, i.e., a database fully equipped with gmaps. Instead of designing a physical organization from scratch, we show how we can make incremental changes on the object-oriented physical schema to improve its performance.

First, we replicate attributes over paths that traverse relationships both in the inverse direction (from attribute to class), and in the forward direction (from class to attribute). Such a replication brings cost savings in queries Q3 and Q7. Then, we consider replicating attributes over paths that are not functional by associating multiple values with each instance of the root of the path. Such replication is very beneficial for our workload since it can eliminate the most expensive

[1] We use the term path indices collectively for what Bertino and Kim [2] call path indices, nested indices, and multi-indices.

step of the medium and large queries Q5, Q7, and Q8. The exact cost savings and overheads that are caused by each additional access path are shown in Table 6.

Table 6: Applications of gmaps

Replicate	In	Space	No	Saved
"Student.advises^{-1}.name"	Student	2	Q3,U2	18
"Course.teaches^{-1}.name"	Course	0.4	Q6,U1	9
"Faculty.advises.name"	Faculty	0.6	Q5	51
"Course.attends.name"	Course	12	Q7	76
"Faculty.teaches.name"	faculty	0.4	Q8	50
			Tot. saved	204

These modifications offer a significant total gain of 204 I/Os, or 35% over the cost of the initial object-oriented approach. If we add to these savings those of the previous section (with gmaps simulating field replication) the total gain climbs to 42%. It is interesting to note the low update overhead of all these replications. The reason is that we replicated attributes, like `student.name`, that are static, i.e., do not get updated. As a result, the only updates that are affected are those on the intermediate links of the path. The `attends` relationship, for example, is the most volatile relationship in the schema, so we expect that the slightest update overhead would overwhelm any savings gained. However, since the attends relationship is bi-directional (stored as an attribute of both the student and the course), both extents need to be updated anyway. Reading and writing one more attribute, `student.name`, does not add any overhead.

The space overhead is also low in all but one case: replicating the student names in each course more than doubles the size of the course extent. Whether or not the space-time trade-off is worthwhile depends on the application. The total increase in space usage is 15.4MB, bringing the total database size to 92.4MB. Table 7 compares the query cost, update cost, and

Table 7: Summary costs

	Space (MB)	Query cost	Update cost	Tot. cost
Relational	74	394	219	613
Object-oriented	75	398	185	583
OO + extensions	77	358	185	543
Gmaps (1)	80.4	227	188	415
Gmaps (2)	92.4	151	188	339

disk usage of all four approaches. For the gmap configuration, we show the results both with and without replication of student names, always including the enhancements of the configuration with complex paths.

9 Related Work

Most existing database systems do not provide true data independence, since every construct of the logical schema corresponds directly to a primary physical structure. For example, every relation in most relational systems, every class extent in extent-based OO systems (e.g., Orion [15] and Zoo [10]) and every collection in collection-based OO systems (e.g., GemStone [16], Extra/Excess [5], and ObjectStore [18]) is stored in a separate file. The main flexibility at the physical level comes from secondary access paths to these files.

Several extensions of both the primary physical structure and the secondary access paths have been recently proposed in the literature that allow storing together data from more than one logical construct. In Sections 2 and 8, we discussed path indices [2, 14, 16], join indices [26], and field replication [11, 23], noting their restrictions and comparing their performance to our scheme. Another approach to decomposing the database is hierarchical join indices [27], a generalization of join indices that allows one to build an index over identity surrogates that populate trees of the logical schema graph. Access Support Relations (ASR) offer a different generalization of join indices [12], which allows the definition of indices over the instances of arbitrary chains of logical schema nodes. This scheme offers a higher degree of flexibility and allows the definition of indices that store both complete and partial instances of each chain. Except for the last feature, the contents of both hierarchical join indices and ASRs can be represented as psj-queries, and can thus be defined as a gmap. However, since gmap queries do not support unions, they cannot represent outer joins, and therefore cannot store incomplete instances of chains.

With respect to the translation algorithm, our work most closely resembles research at the University of Waterloo on materialized views [3, 28]. Our algorithm supports a more restricted query language, but uses information about inclusion and functional dependencies as well as "topological" information implicit in a graph-based logical schema. This information allows us to identify solutions that would be missed by the more general algorithm. Section 5 contains an example of a solution that can only be found when inclusion dependencies are taken into account. Similarly, our handling of functional and multivalued dependencies is more general than that of the algorithm of Yang and Larson, which simply uses the primary key information for each relation. Unless all non-trivial dependencies are generated by superkeys (i.e., unless all relations are in at least 4th Normal Form), our scheme will find more solutions.

With respect to the integration with the rest of the query optimizer, most earlier efforts use a two-stage approach, where the queries are first translated into queries over physical structures, and the resulting queries are then optimized one-by-one by a conventional optimizer. In addition to the work on materialized views [3, 28], such efforts include research whose goal was not physical data independence but simply processing efficiency. Examples include research on reusing common subexpressions within a query [9] or between multiple queries [22], reusing results of previous queries [8], and using integrity constraints for semantic query optimization [6]. Kemper and Moerkotte [13] opt for a unified approach of translation and op-

timization for the ASRs by extending a rule based optimizer to include appropriate rewriting rules. Our approach of enhancing a conventional optimizer with the necessary translation steps takes advantage of full cost information available to the optimizer to perform early pruning of inferior solutions, while keeping the overall optimization cost low.

10 Conclusions

We have presented a new approach to physical schema design that uses a declarative language to describe the contents of storage structures. Carefully restricting the language allows efficient algorithms to translate queries over the logical schema into access plans using the physical data structures. We have shown how to integrate the query translation algorithm into a conventional query optimizer. A simple modification of the query translation algorithm supports propagation of updates to the database. A prototype system that incorporates the major aspects of this approach is currently operational. We have used it to demonstrate in a realistic environment how our approach can achieve significant performance gains over more conventional ones.

In the future, we plan to extend the translation algorithm to take into account additional integrity constraints and also permit the translation of queries into unions of other queries. We would also like to incorporate storage structures that offer a hierarchical interface, and study the feasibility of using the query optimizer to exploit in-memory data storage structures. Finally, the increase of available choices in the physical schema design puts the burden on the database administrator to make the correct choices. We need to design tools that guide the administrator in choosing the appropriate combination of storage structures.

References

[1] A. Aho, Y. Sagiv, and J. Ullman. Equivalences Among Relational Expressions. *SIAM Journal of Computing*, 8(2):218–247, 1979.

[2] E. Bertino and W. Kim. Indexing Techniques for Queries on Nested Objects. *IEEE Transactions on Knowledge and Data Engineering*, 1(2):196–214, June 1989.

[3] J. Blakeley, N. Coburn, and P. Larson. Updating derived relations. *ACM Transactions on Database Systems*, 14(3):369–400, Sept. 1989.

[4] M. Carey, D. DeWitt, M. Franklin, N. Hall, M. McAuliffe, J. Naughton, D. Schuh, M. Solomon, C. Tan, O. Tsatalos, S. White, and M. Zwilling. Shoring Up Persistent Applications. In *Proc. of the ACM SIGMOD Conf.*, May 1994.

[5] M. Carey, D. DeWitt, and S. Vandenberg. A Data Model and Query Language for Exodus. In *Proc. of the ACM SIGMOD Conf.*, pages 413–423, Chicago, IL, June 1988.

[6] U. Chakravarthy. Logic-Based Approach to Semantic Query Optimization. *ACM Transactions on Database Systems*, 15(2):163–207, June 1990.

[7] A. Chandra and P. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *Proc. of Annual ACM Symposium on Theory of Computing*, pages 77–90, May 1977.

[8] S. Finkelstein. Common expression analysis in database applications. In *Proc. of the ACM SIGMOD Conf.*, pages 364–374, 1982.

[9] P. Hall. Optimization of a single relational expression in a RDBMS. *IBM Journal of Research and Development*, 20(3), May 1976.

[10] Y. Ioannidis, M. Livny, E. Haber, R. Miller, O. Tsatalos, and J. Wiener. Desktop Experiment Management. *IEEE Data Engineering*, 16(1):19–23, Mar. 1993.

[11] K. Kato and T. Masuda. Persistent Caching. *IEEE Transactions on Software Engineering*, 18(7):631–645, July 1992.

[12] A. Kemper and G. Moerkotte. Access Support in Object Bases. In *Proc. of the ACM SIGMOD Conf.*, pages 290–301, 1990.

[13] A. Kemper and G. Moerkotte. Advanced Query Processing in Object Bases Using Access Support Relations. In *Proc. of the Int. VLDB Conf.*, pages 290–301, 1990.

[14] W. Kim, E. Bertino, and J. Garza. Composite Objects Revisited. In *Proc. of the ACM SIGMOD Conf.*, pages 337–347, 1989.

[15] W. Kim, J. F. Garza, N. Ballou, and D. Woelk. Architecture of the ORION Next-generation Database System. *IEEE Transactions on Knowledge and Data Engineering*, 2:109–124, Mar. 1990.

[16] D. Maier and J. Stein. Indexing in an Object-Oriented DBMS. In *2nd Int. Workshop on Object-Oriented Database Systems*, pages 171–182, Asilomar, CA, Sept. 1986.

[17] R. Miller, Y. Ioannidis, and R. Ramakrishnan. The Use of Information Capacity in Schema Integration and Translation. In *Proc. of the Int. VLDB Conf.*, pages 120–133, 1993.

[18] J. Orenstein, S. Haradhvala, B. Marguiles, and D. Sakahara. Query Processing in the ObjectStore Database System. In *Proc. of the ACM SIGMOD Conf.*, 1992.

[19] N. Roussopoulos. View Indexing in Relational Database. *ACM Transactions on Database Systems*, 7(2):259–290, June 1982.

[20] N. Roussopoulos, N. Economou, and A. Stamenas. ADMS: A Testbed for Incremental Access Methods. *IEEE Transactions on Knowledge and Data Engineering*, 5(5):762–773, Oct. 1993.

[21] P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, and T. Price. Access Path Selection in a Relational Database Management System. In *Proc. of the ACM SIGMOD Conf.*, pages 23–34, 1979.

[22] T. Sellis. Global Query Optimization. In *Proc. of the ACM SIGMOD Conf.*, pages 191–205, 1986.

[23] E. Shekita and M. Carey. Performance Enhancement Through Replication in an Object-Oriented DBMS. In *Proc. of the ACM SIGMOD Conf.*, pages 325–336, 1989.

[24] O. Tsatalos and Y. Ioannidis. A Unified Framework for Indexing in Database Systems. In *Proc. of the Int. Conf. on Database and Expert System Applications*, Sept. 1994.

[25] O. Tsatalos, M. Solomon, and Y. Ioannidis. Enhanced Storage Structures for OODBMSs. Unpublished manuscript, available from the authors, Feb. 1994.

[26] P. Valduriez. Join Indices. *ACM Transactions on Database Systems*, 12(2):218–246, June 1987.

[27] P. Valduriez, S. Khoshafian, and G. Copeland. Implementation Techniques of Complex Objects. In *Proc. of the Int. VLDB Conf.*, pages 101–109, Kyoto, Japan, 1986.

[28] H. Yang and P. Larson. Query transformation for PSJ-queries. In *Proc. of the Int. VLDB Conf.*, 1987.