The GMAP: A Versatile Tool for Physical Data Independence

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

Computer Sciences Department
University of Wisconsin
Madison, WI 53706
{odysseas,solomon,yannis}@cs.wisc.edu

October 18, 1994

Abstract

Physical data independence is touted as a central feature of modern database systems. It allows users to frame queries in terms of the logical structure of the data, letting a query processor automatically translate them into optimal plans that access physical storage structures. 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. We present an approach that 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.

*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 the National Science Foundation under Grants IRI-9113736, IRI-9224741, and IRI-9157368 (PYI Award) and by grants from DEC, IBM, HP, AT&T, and Informix.

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 [BK89, MS86], field replication [KM92, SC89], 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 depicted as graphs. Throughout this paper we illustrate our approach with an example database describing a university and its personnel (see Figure 1). The textual form of the schema is given in Appendix A using ODL [Cat93].

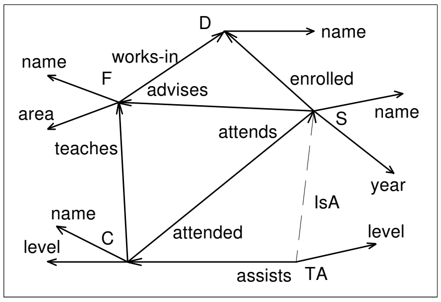

Figure 1: The logical schema

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" as a synonym for "domain" (node in the graph) and the term "base relation" as a synonym for "relationship" (edge). Our algorithms operate on the (binary) relational form of the schema, so 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 predicate `Dept =`

cs_oid is called a *selection*. 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 Figures 2-9). Shaded edges correspond to relationships or IsA associations 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.

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 the above example,

$$Q = \pi_{\mathtt{F,F.area}} \sigma_{\mathtt{D=cs\_oid}} (\mathtt{F.area} \bowtie \mathtt{works\_in}).$$

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. In this paper, we consider only logical queries written in the same language that is used 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 as follows (Figure 2):

```
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
```

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 follows (Figure 3):

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

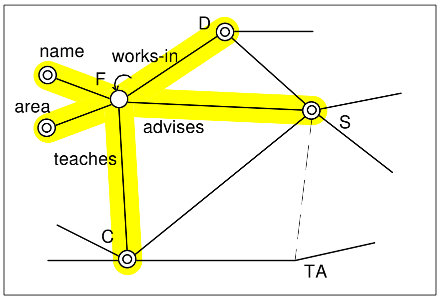

Figure 2: The Faculty class extent

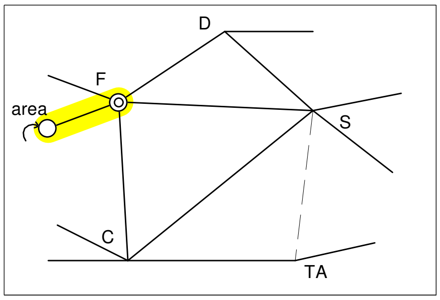

Figure 3: A Faculty index on "area"

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* [KM92, SC89], which has been shown to offer several advantages. The only change necessary is to add "`Dept.name`" to the `select` clause. Only a slight modification of the earlier gmap definition is required (Figure 4):

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

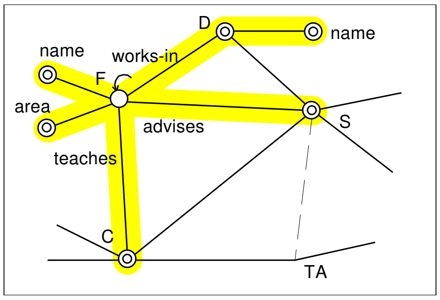

Figure 4: Replication of a nested attribute

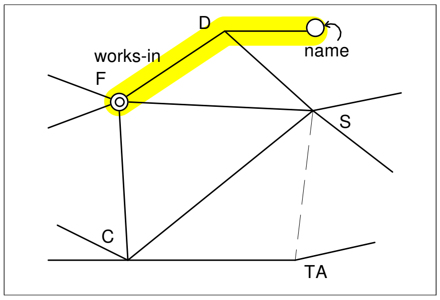

Figure 5: A nested index

Similarly, suppose applications frequently ask for the listing of the faculty of a specific department. In this case we need a fast access path from the department name to the faculty domain. A structure for accelerating such path expressions is called a *nested index* [MS86, BK89], which allows indexing on a nested attribute of a domain. Such an index is easily specified as a gmap (Figure 5):

```
def_gmap faculty_nested_index as btree by
    given Dept.name  select Faculty
    where Faculty works_in Dept.
```

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 [CDV88, MS86, OHMS92] 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. For example, if we would like to modify the previously defined index on faculty area so that it keeps only data for faculty in the computer science department, the definition would be as follows (Figure 6):

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

where cs_oid is the object identifier of the computer science department.

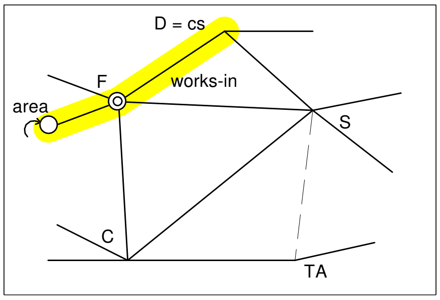

Figure 6: A collection based index

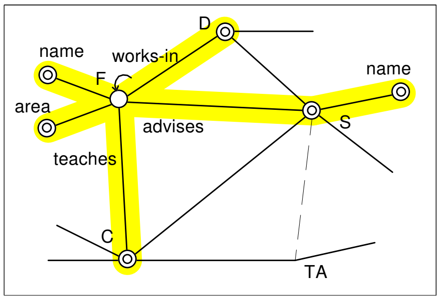

Figure 7: Repl. of a non-functional nested attribute

Field replication and path indexing techniques typically impose restrictions related to the logical schema. For example, in field replication, the nested attribute is required to be a nested *part of* the object in which it is replicated. Furthermore, the object must functionally determine the nested attribute. Similar restrictions hold for path indexing. Our notion of queries enables one to describe any schema subgraph, independently of the edge properties. The extra freedom allows specification of several useful novel structures:

- A storage structure that replicates nested attributes that are non-functionally determined. For example, we can extend the faculty_data structure (Figure 2) by replicating the names of the advised students (Figure 7):

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

- A cross between a path index and an index on a composite key. It allows each component of the key to be supplied by a separate path. For example, the following gmap builds an index that maps area/course-level pairs to faculty in that area who teach such courses (Figure 8):

```
def_gmap faculty_multi_field as btree by
    given Faculty.area, Course.level
    select Faculty
    where Faculty  teaches Course.
```

- An arbitrary decomposition of data in the inheritance hierarchy. For example, the following gmap clusters a teaching assistant's name together with other teaching assistant attributes that do not pertain to arbitrary students (Figure 9):

```
def_gmap ta_with_sname as heap by
    given TA
    select TA.level, Course, Student, Student.name
    where TA isa Student and
          TA assists Course.
```

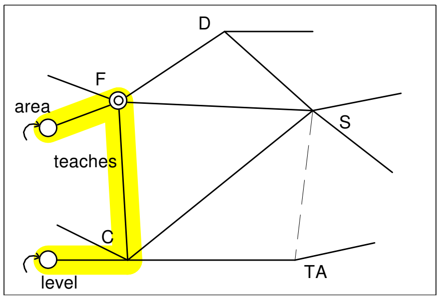

Figure 8: A composite key path index

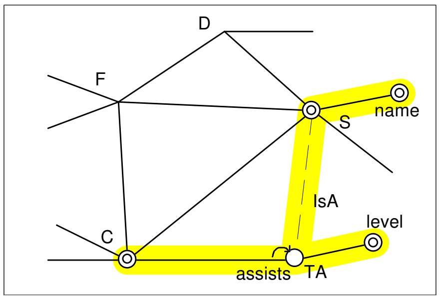

Figure 9: A vertically decomposed subclass extent

A complete taxonomy of existing indexing schemes and other advanced storage structures that can be defined by using gmaps is presented elsewhere [TI94].

3 Query translation

In this section we present the translation algorithm of a logical query into queries over gmaps. 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 represent a logical psj-query $Q = \pi_Q \sigma_Q(R_1 \bowtie \cdots \bowtie R_k)$ as a triple $\langle Q_r, Q_s, Q_p\rangle$ of sets. Q_r is the set of joined binary relations $\{R_1, \ldots, R_k\}$, Q_s is the selection predicate σ_Q represented as a set of simpler predicates (comparisons between attributes and constants), and Q_p is the set of attributes retained by the projection π_Q. We call the set Q_p the *query target*, and its members *target attributes*. Consider for example the following query Q (Figure 6):

```
given Faculty.area select Faculty
where Faculty works_in Dept and
      Dept = cs_oid.
```

Q is a psj-query: if we call R_{ab} and R_{bc} the base relations corresponding to the `Faculty.area` and `works_in` relationships respectively, and a, b, c the attributes corresponding to the domains `Faculty.area`, `Faculty` and `Dept` respectively, then we can rewrite the query as $Q = \pi_{ab}\, \sigma_{c=cs\_oid}(R_{ab} \bowtie R_{bc})$. Using our notation, Q can be represented as the triple: $\langle \{R_{ab}, R_{bc}\}, \{c = \texttt{cs\_oid}\}, \{a, b\}\rangle$. (recall that `Faculty.area` is overloaded to refer both to an attribute and to a relationship).

Given a query Q, we frequently deal with its part that includes only relations in a set $\mathcal{R}$, and its part that includes only relations not in $\mathcal{R}$. These are denoted $Q[rel \in \mathcal{R}]$ and $Q[rel \notin \mathcal{R}]$ respectively. For example, $Q[rel \in \{R_{bc}\}] = \langle \{b\}, \{c = \texttt{cs\_oid}\}, \{R_{bc}\}\rangle = \pi_b\, \sigma_{c=cs\_oid}\, R_{bc}$. Similarly, the subset of Q_s that mentions only attributes in a set $\mathcal{A}$ is denoted $Q_s[attr \in \mathcal{A}]$. Finally, the set of 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 it naturally joins the two query results before any projection has taken place. The projection step is applied after the join and retains all columns that would have been retained by either query. Thus, the same set of columns as in the natural join are produced.

Note that, in general, if we only have the query results in hand, the add-join cannot be calculated since it may require knowledge of some of the missing columns. Consider for example the queries $Q_1 = R_{ab}$ and $Q_2 = \pi_{ac}(R_{ab} \bowtie R_{bc})$. The natural join of Q_1 and Q_2 is $R_{ab} \bowtie \pi_{ac}(R_{ab} \bowtie R_{bc})$, while their add-join is simply $R_{ab} \bowtie R_{bc}$. In general the two expressions will produce different results and the add-join may not even be producible from the results of Q_1 and Q_2 alone (since the b attribute is missing from the result of Q_2).

Let R_{ab}, R_{bc} be two relations with a common attribute b. An *inclusion dependency* from R_{ab} to R_{bc} exists, denoted $R_{ab}.b \subseteq R_{bc}.b$, if every value of b in R_{ab} appears also in R_{bc}.

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 [ASU79, CM77]. 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

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

i.e., when the subquery involving only the relations in $\mathcal{R}$ is not affected by the rest of the query. For example, if $R_{ab} \equiv \pi_{ab}(R_{ab} \bowtie R_{bc})$, then $R_{ab} \bowtie R_{bc}$ covers $\{R_{ab}\}$. In general, the left-hand side of (1) is a superset of the right-hand side. When (1) holds, the part of the query that involves relations not in $\mathcal{R}$ (relation R_{bc} 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 necessary and sufficient conditions for testing coverage is presented elsewhere [TSI94]. The algorithm makes use of the inclusion dependencies of the schema.

3.3.2 Natural join vs. Add-Join

In general, if Q_1 and Q_2 are two psj-queries, $Q_1 \oplus Q_2 \subseteq Q_1 \bowtie Q_2$. However, in the presence of certain integrity constraints, $Q_1 \oplus Q_2 \equiv Q_1 \bowtie Q_2$. For example, consider the queries $Q_1 = R_{ab}$ and $Q_2 = \pi_{ac}(R_{ab} \bowtie R_{bc})$. In general, $Q_1 \bowtie Q_2 = R_{ab} \bowtie \pi_{ac}(R_{ab} \bowtie R_{bc})$ is not equivalent to $Q_1 \oplus Q_2 = R_{ab} \bowtie R_{bc}$. If, however, b is functionally determined by a in R_{ab} (that is, a is a key for R_{ab}) the two joins are equal. Intuitively, the information "lost" by projecting away the b attribute in $\pi_{ac}(R_{ab} \bowtie R_{bc}) \bowtie Q$ can be completely recovered from the remaining attributes (a in this case).

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 [TSI94].

3.4 Query Translation Algorithm

Below we present an algorithm to translate a logical psj-query into queries over gmaps. For the sake of clarity we defer most efficiency considerations to Section 4.

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. $\qquad Q_s[attr \in A(G_r)] = G_s[attr \in A(Q_r)] \cup Q_s[attr \in G_p]$ **and**
3. $\qquad G$ covers $Q_r\}$
4. **for each** subset $\{G_1, ..., G_n\}$ of $\mathcal{H}$ **do**
5. $\quad$ **let** $\mathcal{S} = \{\pi_{A(Q_r)} \sigma_{Q_s} G_1, \ldots, \pi_{A(Q_r)} \sigma_{Q_s} G_n\}$
6. $\quad$ **while** there is $G, H \in \mathcal{S}$ s.t. $G \bowtie H \equiv G \oplus H$
7. $\qquad$ replace G and H in $\mathcal{S}$ by $G \bowtie H$
8. $\quad$ **if** $\mathcal{S} = \{Q'\}$ where $Q \equiv \pi_{Q_p}(Q')$ **then** accept $\{G_1, ..., G_n\}$ 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 that 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 identical to selections that the gmap itself has (line 2); the gmap must cover the relations that it has common 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):

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

The rest of the algorithm tests whether or not this query expression is equivalent to the given logical query. Each join operand in (2) 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 its 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, the set can no longer be reduced, 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.

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

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

Because there are exponentially many subsets of $\mathcal{H}$, Algorithm 1 requires 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 the names and department id's of students attending all 500-level courses:

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

In our formal representation, $Q_r = \{\texttt{attends, enrolled, Course.level}\}$, $Q_s = \{\texttt{Course.level} = 500\}$ and $Q_p = \{\texttt{Student.name, Dept}\}$.

Suppose the database consists of three gmaps: an index G_1 from the names of students to their departments, an index G_2 from the names of students to courses that they attend and the levels of those courses, and an index G_3 from a course-level to courses at that level, together with the departments that supply students to those courses.

Their definitions follow:

```
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). Consider , for example, $G = G_2$. Written as a triple, $G = \langle\{\texttt{Student.name, Course, Course.level}\}, \emptyset, \{\texttt{Student.name, attends}\}\rangle$. Line 1 is satisfied by the attribute `Course` which appears both in G_p and in the relationship `attends` shared by G_r and Q_r. For line 2, note that $G_s = \emptyset$, and $Q_s[attr \in G_p] = Q_s[attr \in A(G_r)] = \{\texttt{Course.level} = 500\}$. To check that G covers Q_r (line 3), note that $G[Q_r]$ is G modified by the removal of `Student.name` (both as an attribute in G_p and a relationship in G_r). If we assume that every `Student` has a `name`, $G[Q_r]$ is equivalent to $\pi_{A(Q_r)}(G)$, which is the result of projecting out the `Student.name` attribute from G.

The algorithm considers subsets of the relevant gmaps G_1, G_2, G_3. Consider, for example, the subset $\{G_1, G_2\}$. The candidate solution corresponding to this combination is the natural join of G_1 and G_2 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 $G_1 \bowtie G_2 \equiv G_1 \oplus G_2$. 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). Assuming that this is the case, line 8 of Algorithm 1 eventually concludes that the candidate solution is a correct one.

Following the same process, the algorithm rejects the subsets $\{G_1, G_3\}$ and $\{G_2, G_3\}$, because the corresponding add-join and natural join are different. However, the combination $\{G_1, G_2, G_3\}$ is a correct solution. During the course of the loop of lines 6 and 7, the algorithm tests 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 fail except $G_1 \oplus G_2$. In the next iteration, the pair $(G_1 \oplus G_2, G_3)$ is considered and the algorithm verifies that $(G_1 \oplus G_2) \bowtie G_3 \equiv (G_1 \oplus G_2) \oplus G_3$.

Interestingly, the solution using all three gmaps is likely to be more efficient than the one that uses only G_1 and G_2, because the index on `Course.level` in G_3 will accelerate the selection in the query. The next section shows how a gmap-equipped 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 [SAC+79], 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" [SAC+79]. 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.

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 pairs of gmaps. If one has neither a greater contribution to the query nor a lower cost than the other, 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 each newly generated solution to all other solutions. If any single gmap or gmap combination has neither a greater contribution to the query nor a lower cost than another, prune it.
c) If the query involves only two relations, stop.	**c)** If there are no partial solutions, stop.
Iteration 3	**Iteration 3**
...	...

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 a 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 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 Algorithm 1 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 [Fin82].

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 this point, there is no partial solution left and the algorithm ends. This example demonstrates that access plans that are pruned in a conventional optimizer are also pruned in its enhanced version. However, since an access plan considered at iteration n in the conventional version may combine more than n gmaps, it may be considered at a later iteration in the enhanced version, thus delaying potential prunings. In general, we expect the performance of the modified optimizer to be similar to the performance of the conventional 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 defining "virtual" relations called *views*, and users express their queries in terms of these 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 of the above. We define the physical structures as functions of the logical schema. Although query translation becomes more complicated, we have shown above that it is still feasible and 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. In addition, updates become much simpler. Translating them into the physical schema turns into the materialized view maintenance problem, which admits simple solutions.

As discussed elsewhere [BCL89], 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 may be replicated in many places.

In this, section we first describe how a *logical update* can be specified, and the restrictions that it must obey. Then we show how the logical update can be translated into a physical update over gmaps, and how this translation can be integrated with the query optimizer to offer a general purpose update propagation mechanism. Finally, we illustrate the update propagation process using an example.

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.
```

Describing the schema of the update using a query, allows us to support updates that do not depend on a specific physical schema. For example, relational databases allow the insertion of entity instances because they assume the existence of an entity extent. Object-oriented systems have also similar restrictions. In our case, by describing the update as a logical query we avoid any physical data dependence. However, it is is not our intention to support updates defined by arbitrary queries, since that could lead into the view update problem. We only want to use the query to define a pattern on the logical schema which represents the "schema" of the update. Thus we need to impose the following restrictions:

- **No selections:** If selections were allowed, they could only supply information that would be redundant to or inconsistent with the update data. For example, the update query

  ```
  def_query define_undergraduate_courses as
      select Course, Course.name, Course.level
      where Course.level < 700,
  ```

  merely states that all tuples in the update data have a level field containing a value less than 700.

- **No projections:** All attributes of the query relations must be included in its target. For example, an insertion request defined by the query

  ```
  def_query add_graduate_student as
      select Student.year, Course
      where Student attends Course,
  ```

  and the update data { [5, cs764] } requests that after the update, there should be an additional 5th-year student taking the course with the indicated id, but gives insufficient information to construct such a student. Such an update has ambiguous semantics and thus should not be allowed.

- **No tuple-generating dependencies:** The relation defined by the update query should not satisfy any tuple generating dependencies [Ull88]. For example, the relation defined by the update query

  ```
  def_query assign_work_to_faculty as
      select Faculty, Student, Course
      where Faculty advises Student
            Faculty teaches Course,
  ```

  satisfies the multivalued dependency `Faculty` $\rightarrow\!\!\!\rightarrow$ `Student`. If the faculty member F was teaching course C_1 before the update, the tuple $[F, S, C_2]$ cannot be added without also adding $[F, S, C_1]$. If the update data included only the first of these tuples, there would be no way to perform the update according to the semantics defined above.

Even with these restrictions, we cannot completely guarantee the validity of an update. For example, if we have not defined any gmap to store `Faculty` data, an insertion of a `Faculty` tuple would be invalid. If we use a single gmap to store the `Faculty` and the `Dept` data, then we will not be able to insert a faculty member who does not work in any department. 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. In particular, the physical schema must have sufficient "information capacity" to hold the inserted data [MIR93]. These issues are not addressed in this paper and are subject of the authors' ongoing research. Furthermore, we do not address the question of whether the update conforms with the integrity constraints of the database. Specific gmap structures or gmap combinations may help accelerate testing the integrity constraints. However, here, we assume that an external mechanism verifies that the updates do not conflict with existing functional or inclusion dependencies.

5.2 Update Translation

Consider an update query U and a gmap query G that have some relations in common, i.e., $U_r \cap G_r \neq \emptyset$. Our goal is to calculate the effect of the update on the data stored in G. Given the restrictions that we imposed on the update query, notably the fact that the update query U may not contain any projections or selections we can easily factor G as

$$G = \pi_{G_p}\sigma_{G_s}(U \bowtie I), \tag{3}$$

where I is the part of the gmap that is irrelevant to the update:

$$I = \langle G_r - U_r, G_s[A(G_r - U_r)], A(G_r - U_r) \cap (G_p \cup A(U_r))\rangle.$$

While complicated, this formula is essentially the same as $G[rel \notin U_r]$; the only difference is that the projection step of I retains additional attributes in the target, to make sure that all needed columns join with U. Note if G is completely irrelevant to U the

Equation (3) allows us to calculate the tuples needed to be inserted or deleted from the gmap G using only the data in the database before the update and the update itself. If we view G and U as the multisets of tuples that are produced by the queries G and U (assume that multiset projection, selection and join operators are used in the queries), respectively, and we use the subscripts *old* and *new* to refer to sets produced before and after the update, then we can rewrite equation (3) as

$$\begin{aligned} G_{new} &= \pi_{G_p}\sigma_{G_s}(U_{new} \bowtie I_{new}) \\ &= \pi_{G_p}\sigma_{G_s}((U_{old} + \Delta U) \bowtie (I_{old} + \Delta I)), \end{aligned}$$

where "+" represents union of multisets if the update is an insertion, and multiset difference if the update is a deletion. Since I is irrelevant to the update, $\Delta I = \emptyset$. Furthermore, multiset projection, selection and join, distributes over multiset union and difference yielding the following:

$$\begin{aligned} G_{new} &= \pi_{G_p}\sigma_{G_s}((U_{old} + \Delta U) \bowtie I_{old}) \\ &= G_{old} + \pi_{G_p}\sigma_{G_s}(\Delta U \bowtie I_{old}). \end{aligned}$$

Finally, since I is a psj-query, it can be translated using Algorithm 1 into a query expression over gmaps. Thus, the change ΔG of the gmap contents can be produced by the query

$$\Delta G = \pi_{G_p}\sigma_{G_s}(\Delta U \bowtie \pi_{I_p}\sigma_{I_s}(\pi_{A(I_r)}G_1 \bowtie \cdots \bowtie \pi_{A(I_r)}G_m)), \tag{4}$$

which only uses existing gmaps and the update data.

Equation (4) can be used for deletions as well as insertions if the gmaps whose target does not functionally determine their other attributes maintain their data as multisets (that is, they records duplicates or multiplicities).

5.3 Update Propagation Algorithm

Below we present the update propagation algorithm which uses the translation mechanism described above. For the sake of clarity we defer most efficiency considerations, to the end of this section.

Algorithm 2 *Given a logical update specified by the update query U and a set of gmaps $\mathcal{G}$ find the update that needs on be performed to each of the gmaps in $\mathcal{G}$.*

1. **for each** $G \in \mathcal{G}$ **do**
2. **let** $I = \langle G_r - U_r, G_s[A(G_r - U_r)], A(G_r - U_r) \cap (G_p \cup A(U_r))\rangle$
3. **if** $I \equiv G$ **then continue** // no update is needed
4. Translate I into a query Q over gmaps in $\mathcal{G}$ using Algorithm 1
5. Execute the query $\pi_{G_p}\sigma_{G_s}(U \bowtie Q)$
6. Insert/Delete all tuples generated by the query into/from G

For each gmap G, the algorithm first identifies the part I that is irrelevant to the update (line 2). If the whole gmap is irrelevant, no update propagation is needed (line 3). Otherwise, we use Algorithm 1 to translate I into a query expression over gmaps (line 4). The update data are joined with the result of this query, producing a multiset of tuples (line 5). If the update is an insertion, the tuples are inserted into G; otherwise they are deleted from G.

There are a few performance issues that we need to address. First, the translation of I is performed through the enhanced query optimizer. The optimizer guarantees that the produced plan will be optimal: any available gmap that can accelerate queries can also be used to accelerate the performance of updates. Second, in many applications, there is a fixed set of logical updates that are used regularly (such as courses being created or students enrolling). These common update queries can be translated in advance and stored as query plans. They only need to be recompiled when the update query or the physical schema changes.

Finally, the algorithm presented above requires that the update gmap holds the leftmost position in the join tree, which may not be the optimal join order. For example, if we are inserting a large number of tuples, it would not be advisable to load the update gmap first. Since, however, we have modeled the update as a gmap, we can let the query optimizer select the appropriate join order for that gmap as well. For this, the query optimizer needs to be extended with the notion of a *required* gmap which should be used in any plan produced by the optimizer. Then, for the update translation, we can flag the update gmap as required and use it together with the remaining database gmaps to compile the gmap query G.

5.4 Update Example

We illustrate the algorithm above with an example. Consider the update query `enroll_student` presented earlier:

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

Assume that our database consists of two gmaps, one that maps faculty members to courses that 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 append to each *(CourseId, StudentId)* pair in the update data the faculty member who teaches the course before it is 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 finding a translation for it. Query 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` cannot 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 3 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.

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 databases 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 physical schema evolution. 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. For example, to run the experiment described in Section 8, we populated our department database with data from a variety of sources. Course enrollments were retrieved from an IMS database maintained by the registrar. Faculty advising data were found in an Ingres database. TA assignments were kept in a hand-maintained ASCII file, and so on. All these data were dumped into Unix files in a simple textual format. We then wrote a set of interface functions that allowed us to view these Unix files as a new gmap structure. The functions allowed iteration through the tuples and statistics inquiries (such as a count of tuples) for the benefit of the query optimizer. During loading, all data were automatically combined (e.g. joined and projected) to match their final formats and then bulk-loaded into the storage structures. The query processing engine guaranteed that all data were processed optimally.

6.2 Database Interoperability

In the previous paragraph, we showed how to create a thin "veneer" over Unix text files to make them behave like gmaps. For purposes of bulk loading, a simple read-only interface sufficed, but we could have added functions for inserting, deleting and associatively accessing tuples in these files. In short, with very little work, a text file can be made to emulate a gmap. 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 has many similarities with the ADMS system [RES93], which uses the concept of view indexing [Rou82] to efficiently represent materialized views, together with a semantic query optimizer to support database interoperability.

6.3 Main Memory Caching

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. The fact that it is not persistent simply means that its data must be replicated in other (persistent) gmaps. Many database applications cache parts of the database, but because of restrictions in data replication, these data cannot be part of the physical schema, and thus have to be accessed and maintained manually.

6.4 Accelerating Complex Structure Updates

Although path indices are useful for accelerating common queries, they are expensive to maintain. Previous proposals for both nested indices and field replication suggest including links along the indexed path to be used during update propagation to accelerate the joins required. In both cases, the data structures and the algorithms for maintaining them are custom-made for the application. 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. In addition, they are much more versatile in that they can be used by arbitrary queries that involve these joins since they are not tied to any specific larger structure.

Suppose, for example, that we use a heap to store course data and also replicate the name and department name of the faculty member who teaches each course:

```
def_gmap course_data as heap by
    given Course
    select Course.name, Course.level, Faculty.name, Dept.name
    where Faculty teaches Course and
          Faculty works_in Dept
```

When a faculty member moves to a different department, we need to update the course data for all of his or her courses. Shekita and Carey [SC89] suggest adding backward links from faculty to their respective courses, in effect, advocating a pointer-based join.

We propose instead to add a gmap from faculty to courses:

```
def_gmap courses_index as hash_table
    given Faculty  select Course
    where Faculty teaches Course
```

Although the performance of this approach may not match that of a pointer join (it has been shown that pointer joins are generally faster than joins accelerated through an external index [CSL+90]), it offers significant advantages. First, the gmaps can be used not only for updating the structure but by any other query as well. Second, if usage patterns change so that maintenance of the accelerator is not beneficial, we can simply remove it. In the hardwired approach, we will always be stuck with the same machinery. Finally, our approach allows the user to place join accelerators at individually chosen points. It is much harder to achieve this flexibility with the hardwired approach.

6.5 Support for Complex Objects

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.

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 [CDF+94]. SHORE is an object-oriented database system under construction at the University of Wisconsin.

We extended SHORE with facilities to support gmaps and translate and execute queries and updates. Logical schema definitions are parsed and stored in a logical-schema catalog. Physical storage structures are created from gmap definitions. Parsed gmap queries are stored persistently in a second physical-schema catalog. The data organization, keys, and record format are also determined by the gmap definitions. Gmaps are created and populated by processing update requests.

We have built a query processor using the algorithms in Sections 3 and 4. For all the examples presented in this paper, query translation adds 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 the gmaps in the physical-schema catalog. Other combinations of arguments support other applications described in Section 6.

We have designed a simple common interface to allow a variety of storage structures to emulate gmaps. This interface includes operations to store and retrieve data and make cost inquiries. We have created implementation for 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 have also modified the SHORE B^+-tree facility to maintain a count of duplicate insertions.

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 Appendix A and 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 in page I/Os 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)*.

To obtain I/O counts, we instantiated queries and updates using random values for the input parameters and the inserted pairs, 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 at the time of experimentation.

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

8.2 Relational Design

The relational configuration follows a textbook translation of the logical schema into relations. We created a relation for each internal node of the schema and one for the many-to-many relationship `attends`. We assumed that the attribute `name` is a primary key for each entity in the schema, thus playing the role of the identity surrogate for the relational configuration. We decomposed the inheritance association vertically, i.e., the TA relation includes TA-specific data (`support_level`) as well as `Student.name` and `Course.name` for the assisted course. To simplify the discussion, we assumed that nested loops is the only available join method. Secondary indices were added when necessary to accelerate joins and selections, and we clustered favorably the records in the heaps.

There was one case where gmaps could not simulate the needed structure. Queries Q6 and Q7 need a secondary index on the `attends` relation on both course name and student name. These indices would return a physical pointer in the `attends` relation. They cannot be expressed as a gmap because gmap data can only be values of nodes in the schema graph, and attends is an edge and not a node. Note, however, that the need for these indices is an artifact of the relational design restrictions: We would be better off with two indices that map courses directly to students and *vice versa*. This construct, called a *join index*, has been shown to offer advantages over the conventional relational approach [Val87]. Thus, we replaced the attends relation with two gmaps that simulate the definition of a join index over the attends relationship. The results showed that the join index performed better than a combination of a relation clustered on one of the two names and a pair of secondary indices. The complete physical schema for the relational configuration is presented in Appendix B.1.

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. The other relationships are stored as part of the domain closest to the edge label in Figure 1. For example, the `advises` relationship is represented by an attribute of the `Faculty` class. We also encountered a case for which we had to alter the planned design to conform with the gmap definition language. Initially, we thought that the Student extent should contain both Students and Teaching Assistants (TAs). However, creating a gmap that contains members of both domains, i.e., members of a class and its subclass, requires a query that allows some form of union, and our restricted notion of psj-query does not support such queries. Instead, we followed the same design as in the relational configuration and decomposed TA vertically. For the given workload, this alteration has no performance effect. The gmap definitions that create the desired physical schema are presented in Appendix B.2. 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. It is worthwhile to estimate the improvement that can be achieved by equipping the object-oriented version of our database, 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

In Appendix B.3 we show the altered definitions of the Faculty, Student, and TA gmaps, with the new complex paths added. 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 [KM92, SC89] 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 were imposed by previous authors 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

Next, we consider the results of fully equipping the database 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. Second, 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

[1]We use the term path indices collectively for what Bertino and Kim [BK89] call path indices, nested indices, and multi-indices.

for our workload since it can eliminate the most expensive step of the medium and large queries Q5, Q7, and Q8. The altered gmap definitions are shown in Appendix B.4. 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^{-1}.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 disk usage of all four approaches. For the gmap configuration, we show the results both with and without replication of student names, in both cases including the enhancements of the configuration with complex paths.

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

8.6 A Change in the Workload

Finally, we consider the problem of adapting the database to a changing workload. Assume that our workload is augmented by a single run of one additional query Q9 that asks for the names of all TAs supported by a department. More precisely, Q9 asks for all TAs who assist a course taught by a faculty member that works in a given department. The simplest way to handle the new query in the initial object-oriented configuration would be to recluster the TA extent so that it is ordered by the supporting department of the TAs, and add an index from course ids to TAs. The cost of the query Q9, however, remains high —164 I/Os—and the total contribution of the query to the load, taking into account the additional update overheads, climbs to 172 I/Os. We use this configuration as the baseline for the new workload.

The problem with the new query is that it requires unclustered access to the large Student file in order to obtain the student name. Reclustering the file does not help since query Q5 depends on that specific clustering for its performance. Keeping two copies of the student file, clustered for Q5 and Q9, respectively, is a bad idea because of the high update rates of the `attends` relationship. Such a solution would also result in a significant increase in the database size without much help in processing cost.

Gmaps offer additional options. We can selectively replicate the `student.name` attribute requested by the new query from the Student extent to the TA extent. Doing so lowers the query cost by 95 I/Os. The net gain after accounting for the additional cost of updates is 79 I/Os. The following table compares the three new configurations. Savings and losses are given relatively to the baseline configuration. Notice that while all options are available to the gmap-equipped database, most conventional databases would only have the first option and a few may support the second.

Description of technique	**Space overhead (MB)**	**Activities affected**	**Saved (Lost)**
Recluster student extent	0	Q5,Q9	(5)
Keep two copies of student extent	40	Q9,U3	(35)
Replicate "Student.name" in TA extent	0.1	Q9,U4	79

Under the new workload the gmap-equipped configuration has a load of 422 I/Os compared to 755 I/Os in the object-oriented system, for total savings of 44%.

9 Related Work

Most existing database systems do not provide true physical 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 [KGBW90], and Zoo [ILH+93]) and every collection in collection-based OO systems (e.g., GemStone [MS86], Extra/Excess [CDV88], and ObjectStore [OHMS92]) 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 [BK89, KBG89, MS86], join indices [Val87], and field replication [KM92, SC89], noting their restrictions and comparing their performance to our scheme. Another approach to decomposing the database is hierarchical join indices [VKC86], 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 [KM90a], 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 [BCL89, YL87]. 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 [2]. Similarly, our handling of functional 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.

[2] In the example of Section 5.4, the translation of the update query for the gmap `CFS` has one solution if there is no inclusion dependency from `teaches` to `attends` and two solutions otherwise. An algorithm that does not check for inclusion dependencies cannot discover the second solution.

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 [BCL89, YL87], 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 [Hal76] or between multiple queries [Sel86], reusing results of previous queries [Fin82], and using integrity constraints for semantic query optimization [Cha90]. Kemper and Moerkotte [KM90b] opt for a unified approach of translation and optimization 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 schemes.

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

[ASU79] A. Aho, Y. Sagiv, and J. Ullman. Equivalences Among Relational Expressions. *SIAM Journal of Computing*, 8(2):218–247, 1979.

[BCL89] J. Blakeley, N. Coburn, and P. Larson. Updating derived relations. *ACM Transactions on Database Systems*, 14(3):369–400, September 1989.

[BK89] 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.

[Cat93] R.G.G. Catel. *The Object Database Standard: ODMG-93*. Morgan-Kaufman, Inc., 1993.

[CDF+94] 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.

[CDV88] 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, June 1988.

[Cha90] U. Chakravarthy. Logic-Based Approach to Semantic Query Optimization. *ACM Transactions on Database Systems*, 15(2):163–207, June 1990.

[CM77] 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.

[CSL+90] M. Carey, E. Shekita, G. Lapis, B. Lindsay, and J. McPherson. An Incremental Join Attachment for Starburst. In *Proc. of the Int. VLDB Conf.*, pages 662–673, 1990.

[Fin82] S. Finkelstein. Common expression analysis in database applications. In *Proc. of the ACM SIGMOD Conf.*, pages 364–374, 1982.

[Hal76] P.V. Hall. Optimization of a single relational expression in a RDBMS. *IBM Journal of Research and Development*, 20(3), May 1976.

[ILH+93] Y. Ioannidis, M. Livny, E. Haber, R. Miller, O. Tsatalos, and J. Wiener. Desktop Experiment Management. *IEEE Data Engineering*, 16(1):19–23, March 1993.

[KBG89] W. Kim, E. Bertino, and J. Garza. Composite Objects Revisited. In *Proc. of the ACM SIGMOD Conf.*, pages 337–347, 1989.

[KGBW90] 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, March 1990.

[KM90a] A. Kemper and G. Moerkotte. Access Support in Object Bases. In *Proc. of the ACM SIGMOD Conf.*, pages 290–301, 1990.

[KM90b] 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.

[KM92] K. Kato and T. Masuda. Persistent Caching. *IEEE Transactions on Software Engineering*, 18(7):631–645, July 1992.

[MIR93] 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.

[MS86] D. Maier and J. Stein. Indexing in an Object-Oriented DBMS. In *2nd Int. Workshop on Object-Oriented Database Systems*, pages 171–182, September 1986.

[OHMS92] J. Orenstein, S. Haradhvala, B. Marguiles, and D. Sakahara. Query Processing in the ObjectStore Database System. In *Proc. of the ACM SIGMOD Conf.*, 1992.

[RES93] 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, October 1993.

[Rou82] N. Roussopoulos. View Indexing in Relational Database. *ACM Transactions on Database Systems*, 7(2):259–290, June 1982.

[SAC+79] P.G. Selinger, M.M. Astrahan, D.D. Chamberlin, R.A. Lorie, and T.G. Price. Access Path Selection in a Relational Database Management System. In *Proc. of the ACM SIGMOD Conf.*, pages 23–34, 1979.

[SC89] 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.

[Sel86] T. Sellis. Global Query Optimization. In *Proc. of the ACM SIGMOD Conf.*, pages 191–205, 1986.

[TI94] 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*, September 1994.

[TSI94] O. Tsatalos, M. Solomon, and Y. Ioannidis. PSJ-Query Translation in the Presence of Logical Schema Constraints. Submitted for publication, October 1994.

[Ull88] J. Ullman. *Principles of Database and Knowledge-base Systems*, volume 1, pages 423–425. Computer Science Press, 1988.

[Val87] P. Valduriez. Join Indices. *ACM Transactions on Database Systems*, 12(2):218–246, June 1987.

[VKC86] P. Valduriez, S. Khoshafian, and G. Copeland. Implementation Techniques of Complex Objects. In *Proc. of the Int. VLDB Conf.*, pages 101–109, 1986.

[YL87] H.Z. Yang and P.A. Larson. Query transformation for PSJ-queries. In *Proc. of the Int. VLDB Conf.*, 1987.

A Logical schema in ODL

```
  interface Dept {
  public:
    attribute   string            name;
  };
  interface Faculty {
    attribute   string            name;
    attribute   string            area
    attribute   ref<Dept>         works_in
    attribute   set<Student>      advises;
    attribute   set<Course>       teaches;
  };
  interface Student {
    attribute   string            name;
    attribute   short             year;
    attribute   ref<Dept>         enrolled;
    attribute   set<Course>       attends;
  };
  interface TA : public Student {
    attribute   float             support_level;
    attribute   ref<Course>     assists;
  };
  interface Course {
    attribute   string            name;
    attribute   short             level;
  };
}
```

B Physical schema

B.1 Relational Physical Schema

```
// Relations
def_gmap DeptRelation as heap by
  given Dept select Dept.name

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

def_gmap CourseRelation as heap by
  given Course select Course.name, Course.level, Faculty.name
  where Faculty teaches Course

def_gmap StudentRelation as heap by
  given Student
  select Student.name, Student.year, Dept.name, Faculty.name
  where Student enrolled Dept and
        Faculty advises Student
```

```
def_gmap TARelation as heap by
  given TA select Student.name, TA.support_level, Course.name
  where TA isa Student and TA assists Course

// Name Indices
def_gmap Dept_name_index as btree by given Dept.name select Dept
def_gmap Faculty_name_index as btree by given Faculty.name select Faculty
def_gmap Course_name_index as btree by given Course.name select Course
def_gmap Student_name_index as btree by given Student.name select Student

// Other Indices
def_gmap Faculty_area_index as btree by
  given Faculty.area select Faculty

def_gmap Course_teaches_index as btree by
  given Faculty.name select Course
  where Faculty teaches Course

def_gmap Student_advisor_index as btree by
  given Faculty.name select Student
  where Faculty advises Student

def_gmap TA_assists_index as btree by
  give Course.name select TA
  where TA assists Course

def_gmap Course2Student_join_index as btree by
  given Course select Student
  where Student attends Course

def_gmap Student2Course_join_index as btree by
  given Student select Course
  where Student attends Course
```

B.2 Object-Oriented Physical Schema

```
// Class Extents
def_gmap DeptExtent as heap by
  given Dept select Dept.name

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

def_gmap CourseExtent as heap by
  given Course select Course.name, Course.level, Student
  where Course attended Student

def_gmap StudentExtent as heap by
  given Student select Student.name, Student.year,
    Dept, Course
```

```
    where Student enrolled Dept and Student attends Course

  def_gmap TAExtent as heap by
    given TA select Student, TA.support_level, Course
    where TA isa Student and TA assists Course

  // Name Indices
  def_gmap Dept_name_index as btree by given Dept.name select Dept
  def_gmap Faculty_name_index as btree by given Faculty.name select Faculty
  def_gmap Course_name_index as btree by given Course.name select Course
  def_gmap Student_name_index as btree by given Student.name select Student

  // Other Indices
  def_gmap Faculty_area_index as btree by
    given Faculty.area select Faculty

  def_gmap Faculty_advises_index as btree by
    given Student select Faculty where Faculty advises Student

  def_gmap Faculty_teaches_index as btree by
    given Course select Faculty where Faculty teaches Course

  def_gmap Faculty_works_index as btree by
    given Dept select Faculty where Faculty works_in Dept
```

B.3 Object-Oriented Schema with Complex Paths

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

  def_gmap StudentExtent as heap by
    given Student select Student.name, Student.year,
      Dept, Dept.name, Course
    where Student enrolled Dept and Student attends Course

  def_gmap TAExtent as heap by
    given TA select Student, TA.support_level, Course, Course.name
    where TA isa Student and TA assists Course
```

B.4 Gmap Schema

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

  def_gmap CourseExtent as heap by
    given Course select Course.name, Course.level,
      Student, Student.name, Faculty.name
```

```
  where Course attended Student and Faculty teaches Course

def_gmap StudentExtent as heap by
  given Student select Student.name, Student.year,
    Dept, Course, Faculty.name
  where Student enrolled Dept and Student attends Course and
    Faculty advises Student
```