A Unified Framework for Indexing in Database Systems

Odysseas G. Tsatalos* and Yannis E. Ioannidis**

Computer Sciences Department
University of Wisconsin
Madison, WI 53706

Abstract. Several types of data organizations have been proposed in the literature for object-oriented and relational databases. In studying these organizations, there appears to be no underlying common basis on which they can be compared. Many of these organizations impose restrictions on their applicability that seem unnecessary or ad-hoc. We present a unified framework for physical database storage. We show that most existing data organizations follow naturally as special cases of this framework. Furthermore, this framework permits the specification of new types of data organization that have not been proposed or that cannot be described in existing systems, yet appear to be useful.

1 Introduction

One of the most prominent characteristics attributed to modern database systems (DBMSs) is *physical data independence*. In an effort to increase the physical data independence in object-oriented and relational DBMSs, researchers have proposed numerous types of data organizations in the past few years. However, studying these data organizations reveals no underlying common basis on which they can be compared and many restrictions on their applicability seem unnecessary and rather ad-hoc. We have devised a mechanism that disassociates completely a logical schema from its physical representation, thus achieving genuine physical data independence [12]. In this paper, we briefly present the salient features of this mechanism and then use it as a conceptual framework to study various physical data models of DBMSs. We show that the vast majority of existing data organization types follow naturally as special cases of this framework, when restrictions are applied on the framework's properties. This helps in understanding the differences of these organizations better. More importantly, the developed framework permits the specification of new data organization types that cannot be described in existing systems, yet appear to be useful.

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

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

2 The Gmap Mechanism for Physical Data Independence

In current DBMSs, the process of designing the physical schema is not clearly distinguished from the logical schema design. For example, each logical construct definition implies the creation of a corresponding physical data structure. In contrast, we propose using a completely separate physical data definition language (DDL) for the physical schema design. This language allows the definition of each physical data structure as a function of, possibly many, logical constructs. The function is a restricted query expression over the logical schema. User queries are expressed in terms of the logical schema and are not aware of the physical one. We have devised an efficient algorithm for translating such queries into access plans on the physical data structures. A prototype system [12] that incorporates the major aspects of this approach including update propagation and integration with the query optimizer is currently operational.

2.1 Logical Data Definition Language

We use a simple object-oriented data model in which schemas are displayed as graphs. Throughout this paper we illustrate our approach with an example schema describing a university with departments (D), faculty (F), students (S), courses (C) and teaching assistants (TA) (see Figure 1). Nodes in this graph

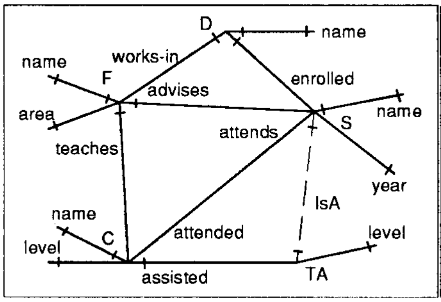

Fig. 1: The logical schema

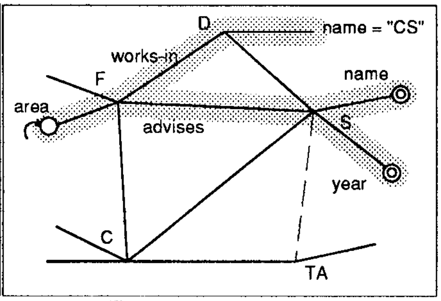

Fig. 2: Query "simple"

represent domains (classes) and solid edges represent associations between them. Classes are divided into *primitive classes*, e.g., integers or character strings, and *entity classes*, whose members are identity surrogates (oids). There are two *kinds* of associations: relationships and inheritance associations. Associations have a *type* and a *cardinality ratio*. The type of relationships captures the intention of the user to view one of the two classes as an attribute of the other. The cardinality ratio of the relationship determines how many instances of one class may be related to an instance of the other class. When the cardinality of the relationship is "to-one" a mark is added in that direction of the edge. Finally, an inheritance association between two entity classes, classifies one class as a subclass of the other. Additional details can be found elsewhere [12].

2.2 Logical Data Manipulation Language

The query language serves the dual role of a simple language to be used in our examples and the language used by the physical DDL to define data structures.

As an example, the query that retrieves the name and year of all students advised by faculty working in the computer sciences department in a given area is

```
def_query simple by
  given Faculty.area select Student.name, Student.year
  where Faculty advises Student and
        Faculty works_in Dept and Dept.name = "CS".
```

The classes following the **given** and **select** keywords are called *input* and *output* classes, respectively. Input classes play the role of query parameters, i.e., they correspond to variables that can be bound to objects from those classes when the query is issued. Output classes indicate the types of objects included in the query result. The remaining parts of the query should be self-explanatory.

Each query can also be expressed graphically as a subgraph of the schema graph called *query graph*. Figure 2 shows the query graph of the example query. Shaded edges correspond to associations explicitly mentioned in the where clause or implicitly mentioned as part of primitive class names. Input classes are indicated by small arrows, and output classes are indicated by double circles.

2.3 Physical Data Definition Language

In our system, each physical storage structure is defined as a *gmap* (Generalized Multilevel Access Path). A gmap consists of a set of records (*gmap data*), a query that indicates the semantic relationships among their fields (*gmap query*), and a description of the data structure used to store the records (*gmap structure*). Issuing the gmap query on a database always produces the gmap data as a result.

For example, suppose we want to cluster together information about each student. Given the object identifier of a Student object, we should be able to retrieve the student name, the student year and the department the student is enrolled in. A gmap that meets these specifications may be defined as

```
def_gmap students_file as heap by
  given Student select Student.name, Student.year, Dept
  where Student enrolled Dept.
```

The statement defines the equivalent of a relation file, i.e., a heap structure directly accessible by the Student oid and containing all student attributes.

3 A Unifying Framework

We identify four aspects of the graph representation of a gmap query. Specifically, such graphs are characterized by their shape, the flavor of their input and output nodes, and their having any selection constraints. Figure 3 shows the possible restrictions that may be placed on each one of them. It then identifies the most important combinations of them (roughly from most to least restrictive), and for each one names the resulting indexing scheme category, which is discussed in a separate section below. For that part of the table, an empty entry implies that no restriction is placed on the corresponding graph characteristic.

A gmap query (without cycles) imposes a directionality on the edges it includes, from its input to its output nodes, essentially capturing how the edges are conceptually traversed when querying. We use the term *arc* to refer to an

Graph Characteristics Restrictions				
Graph characteristic	**Shape**	**Flavor of output nodes**	**Flavor of input nodes**	**Selection constraints**
Possible restrictions	*single arc* *linear path*	*endnodes*	*one endnode* *one node*	*disallowed*

Data Organization Categories				
single edge	single arc	endnodes	one endnode	disallowed
linear path	linear path	endnodes	one endnode	disallowed
multi-output	linear path		one endnode	disallowed
graph			one node	disallowed
multi-input				disallowed
partial				

Figure 3. Gmap categorization based on graph characteristics

edge coupled with such a direction. In many of the categories of Figure 3, existing types of data organizations impose additional restrictions on some characteristics of the arcs that may be part of a gmap query graph. These are the arcs' *kind*, *type*, and *cardinality ratio*, defined in Section 2.1. We use *IsA* and *rel* as abbreviations of inheritance associations and relationships, respectively. For inheritance associations, the arc type may be from class to subclass ($C \rightarrow Sub$) or from subclass to class ($Sub \rightarrow C$), while the cardinality ratio is always $1 \rightarrow 1$. For relationships, the arc type may be from class to attribute ($C \rightarrow A$) or from attribute to class ($A \rightarrow C$) and the cardinality ratio may take any value.

4 Single Arc Indexing

Every edge on the schema graph connects a pair of related classes. It is conceivable that for every such edge there exist a query that requires traversing it. Thus, it is desirable to be able to provide efficient traversal of every edge of the logical schema in both directions, independently of its kind, type, or cardinality ratio. Providing the needed efficiency is straightforward via gmap definitions.

4.1 Previous Techniques

• *Secondary indices in relational databases.* This category includes the conventional, single-key indices of relational databases. The arc properties of these secondary indices are [kind = *rel*, type = $A \rightarrow C$, card = $1 \rightarrow 1$ or $1 \rightarrow N$].
• *Secondary indices on set attributes.* In contrast to relational systems, object-oriented and nested relational systems allow multivalued attributes. This category includes all indices on such attributes. Examples of systems that allow the definition of such indices are ObjectStore [7] and Orion [3]. The arc properties of these indices are [kind = *rel*, type = $A \rightarrow C$].
• *Join indices* [13]. They have been proposed to enhance the performance of joins over many-to-many relationships. The related pairs are stored in two indices, each one ordered according to the two classes participating in the relationship. The two indices offer two different clusterings of the relationship data as well as associative access from both classes. The technique applies to any arc with properties [kind = *rel*] that connects entity classes. An example follows:

```
def_gmap join_index_part1  as btree by
  given Student select Course where Student attends Course
def_gmap join_index_part2  as btree by
  given Course select Student where Student attends Course.
```

• *Multi-indices.* These also use multiple single arc indices. Since their introduction [3, 8], multi-indices have been implemented in at least one commercial product [7]. They allow efficient traversal of a path on the schema graph, by adding indices along the arcs of the path. The important property that these indices should have is that the output of one should be usable as input of the next. This property allows chaining of indices, essentially piping the output of one to the next without accessing any other storage structure. Gmaps have this property, since they use a common object id representation for both output and input objects. For example, the following two gmap definitions create a multi-index that returns the oids of faculty members that work in a department specified by a given name:

```
def_gmap multiindex_part1 as btree by
  given Dept.name select Dept
def_gmap multiindex_part2 as btree by
  given Dept select Faculty where Faculty works Dept.
```

Both multi-index proposals [3, 9] impose the following restrictions on the index arc properties: [kind = *rel*, type = $A \rightarrow C$]. In addition, the proposal by Maier and Stein enforces the restriction [card = $1 \rightarrow 1$ or $1 \rightarrow N$].

4.2 New Applications

• *Class to attribute indices.* Indices that return the value of an attribute given an object id have never been considered, mainly because most systems do include efficient structures to access an object given its id. However, it may be the case that an object is very large, and therefore, the values of an attribute for many objects may span a very large space. For such cases, single arc indices of type $C \rightarrow A$ provide a better clustering. For example, for a query that requests the departments oids of several students, the gmap

```
def_gmap DeptStudent as btree by
  given Student select Dept where Student enrolled Dept
```

includes the needed data in fewer pages and potentially in a more convenient order than the gmap in Section 2.3.

• *Indices on inheritance associations.* *IsA* arcs do not capture relationships between distinct objects, but essentially between two different manifestations of the same object. For example, the information about a `TA` may be stored in two distinct objects, one for the attributes of the `TA` and the other for the generic `Student` attributes. In that case, an index may be valuable to find the oid of one of the objects given the oid of the other.

• *Extended multi-indices.* Any chain of gmaps such that the output nodes of one are the input nodes of the other forms an extended multi-index. For example, the following multi-index

```
def_gmap multiindex_part1 as btree by given Student.year select Student
def_gmap multiindex_part2 as btree by
  given Student select TA where  TA IsA Student
```

```
def_gmap multiindex_part3 as btree by given TA select TA.level
```

includes in its path both relationships and inheritance associations, both $C \dashrightarrow A$ and $A \dashrightarrow C$ arcs, and arcs of many cardinality ratios.

5 Linear Path Indexing

In the previous section, the input and the output classes of gmaps were connected with a single edge. We generalize this structure by allowing the input and output classes to be connected via an arbitrarily long chain of edges.

5.1 Previous Techniques

• *Nested indices.* Consider a linear path with all classes, starting from the input class, being attributes of the next class in the chain. Then, the input class is called a *nested attribute* of the output class. An index that maps objects of the nested attribute class to objects of the nesting class is called a *nested index* [3, 8]. The following statement defines a nested index over the same path we used to define a multi-index in Section 4.1 (Figure 4).

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

Clearly, the chain of indices is replaced by a single index that performs the end-to-end mapping. This results in increased performance since a single index traversal is needed, but also implies reduced index usability and more expensive updates [2]. Originally, nested indices were proposed with the same restrictions on the arc properties as the multi-indices, i.e., [kind = *rel*, type = $A \dashrightarrow C$] [3] or [kind = *rel*, type = $A \dashrightarrow C$, card = $1 \dashrightarrow 1$ or $1 \dashrightarrow N$] [8].

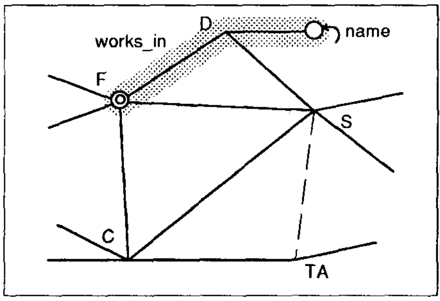

Fig. 4: Gmap "nested_index"

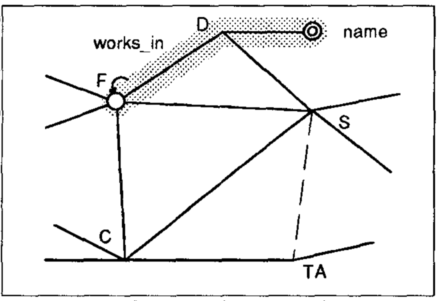

Fig. 5: Gmap "field_replication

• *Field replication.* Consider the exactly opposite case from above, where all classes starting from the output class are attributes of the next class in the chain. Then, it is the output class that is a nested attribute of the input class. Mapping objects of the nesting class to objects of the nested attribute is called *field replication* [4, 11]. By changing the role of the input and output attributes in the previous example, we derive an example of field replication (Figure 5):

```
def_gmap field_replication as heap by
  given Faculty select Dept.name where Faculty works_in Dept.
```

As in the case of class-to-attribute indices in Section 4.2, field replication allows an alternative clustering of the data, which can be important for many applications. Originally, field replication was proposed with the following restrictions on the arc properties: [kind = rel, type = $C \rightarrow A$, card = $1 \rightarrow 1$ or $N \rightarrow 1$].

5.2 New Applications

• *Extended nested indices.* We can generalize both previous techniques by removing all restrictions on the type and kind of the arcs. Note that by chaining nested indices together we obtain a generalization of the hybrid scheme originally proposed for chaining ordinary nested indices [3].

6 Multi-Output Indexing

While arbitrary chains of arcs can be described using the previous types of organizations, the output classes are always endnodes of a path. In this section, we study the opportunities that arise by relaxing this restriction.

6.1 Previous Techniques

• *Path indices.* When the whole path is included in the index output, the resulting data organization is called a *path index* [3]. As an example, consider the path that connects department names with courses taken by students enrolled in the department. The following gmap defines a path index on that path :

```
def_gmap path_index as btree by
  given Dept.name select Dept, Student, Course
  where Student enrolled Dept and Course attended Student.
```

By recording the whole path, the index is useful to more queries than the corresponding nested index, and often allows more efficient updates. Originally, path indices were proposed with the same restrictions on the arc properties as nested indices, i.e., [kind = rel, type = $A \rightarrow C$].

• *Access Support Relations (ASRs)* [5, 6]. These are also full materializations of chains of edges. ASRs are more flexible than path indices: they allow each arc to be either $C \rightarrow A$ or $A \rightarrow C$, but the type must be the same for all the arcs.

6.2 New Applications

• *Extended path indices.* Gmaps do not need the restrictions on arc properties originally imposed on path indices and ASRs. Furthermore, there is no reason to require that every single node of the path be included in the output. For some applications, it may be convenient to include in the gmap data only some of the classes in the path. For example, in the last example (Section 6.1), we may not be interested in `Dept` and `Student` oids but only in `Course` oid and `Course name`. A path index that stores exactly what is needed follows:

```
def_gmap extended_path_index as btree by
  given Dept.name select Course, Course.name
  where Student enrolled Dept and Course attended Student.
```

By eliminating path nodes from the output, the size of the gmap data may decrease significantly. Thus, the resulting gmap offers the needed data in a more compact form, which implies better performance.

7 Graph Indexing

In this section, we relax the restriction that one can only index along linear paths. In general, the gmap graph can be a tree or even have cycles. Although such shapes may not look familiar in the context of indexing, many of the most commonly used storage organizations belong in this category.

7.1 Previous Techniques

• *Relations.* As mentioned earlier (Section 2.3), relations can easily be described as gmaps. As a result, gmaps can be used to achieve a relational physical representation for any given logical schema. Gmaps that correspond to relation files imply the restrictions [kind = *rel*, type = $C \rightarrow A$, card = $1 \rightarrow 1$or $N \rightarrow 1$].
• *Class Extents and Nested Relations.* Class extents are similar to relations. One difference is that they may imply a different heap implementation to support a logical oid access. Another is that they may include multi-valued attributes. Thus, the only implied restrictions on arc properties are [kind = *rel*, type = $C \rightarrow A$]. Some systems allow storing in a class extent members of its subclasses [1]. Describing such organizations requires gmap queries with the union operator; hence, these variants cannot be represented within our framework.
• *Hierarchical Join Indices* [14]. These are a generalization of join indices that also requires a nonlinear query graph for its description. An HJI captures the structure of a complex object by recording the surrogates of the nested objects included in each complex object. Since complex objects have in general a hierarchical structure, the query graph for the index is a tree. For example, the HJI for the `Faculty` complex object can be defined as

```
def_gmap hierarchical_join_index as btree by
  given Faculty select Dept, Student, Course, TA
  where Faculty works_in Dept and Faculty advises Student and
      Faculty teaches Course and Course assisted TA.
```

8 Multi-Input Indexing

8.1 Previous Techniques

• *Indices with composite keys.* A relational index with a composite key, i.e., a key consisting of the concatenation of multiple relation fields, is the most common example of an organization whose query graph includes multiple inputs.

8.2 New Applications

• *Indices with composite key-paths.* Nested indices allow the input class to be many edges away from the output class; conventional indices with composite keys allow multiple input classes that are one edge away from the output class. By combining these, we have an organization with a composite key, such that every component of the key may potentially belong in a separate path. An example is shown in Figure 6. The index maps area/course-level pairs to faculty in that area who teach such courses:

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

The order of the input classes is important and is captured as part of the gmap definition, although for simplicity it is not shown in the graphical representation.

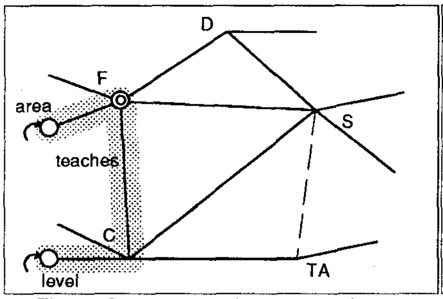

Fig. 6: Gmap "extended_composite"

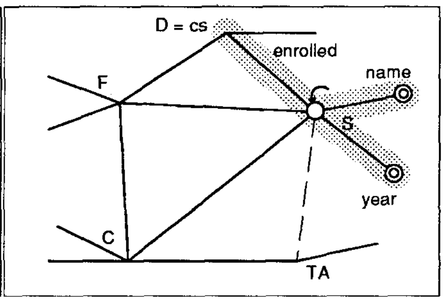

Fig. 7: Gmap "cs_collection"

9 Partial Indexing

9.1 Previous Techniques

• *Collections of Objects.* Until now we assumed an extent-based system: queries were expressed in terms of classes. However, many objects-oriented DBMSs maintain only user-defined object collections, which include only a subset of the class members. The following gmap uses a restriction to define a collection containing only computer science students (Figure 7):

```
def_gmap cs_collection as heap by
  given Student select Student.name, Student.year
  where Student enrolled Dept and Dept = cs_oid.
```

In the above, cs_oid is the oid of the computer science department.

• *Indices on Collections of Objects.* Systems that store instances in explicit collections rather than class extents also allow the creation of indices on top of these collections [8, 10]. These indices provide a fast access path only to the class members that are included in the collection. For example, an index on student **year** for students in the computer science department can be defined as

```
def_gmap cs_collection_index as btree by
   given Student.year select Student
   where Student enrolled Dept and Dept = cs_oid.
```

9.2 New Applications

• *Intentionally defined collections.* In many cases, an alternative way to describe the contents of a set is through its intention, i.e., by using a query that defines the characteristics of the object in the set. Our approach permits the creation of such intentionally defined collections using arbitrary selection conditions in gmap queries, thus allowing arbitrary horizontal decompositions of the database.

10 Conclusions

Exploration of the space of alternative physical data organization types has been the focus of this paper. We have taken a mechanism that achieves genuine physical data independence and have used it as a conceptual, unifying, framework for describing such organizations. Most types of data organizations proposed earlier in the context of relational or object-oriented systems follow naturally as special cases of this framework. Also, the framework has permitted the specification of new useful data organizations that have not been proposed earlier. Given that the entire space of these alternatives is realizable within the existing implementation of gmaps by using different gmap queries [12], one may immediately apply our work to physical design problems.

References

1. R. Agrawal and N. H. Gehani. ODE : The Language and the Data Model. In *Proc. of the ACM SIGMOD Conf.*, pages 36–45, 1989.
2. E. Bertino. Optimization of Queries using Nested Indices. In *Proc. Int. Conf. on Extending Database Technology*, pages 44–59. Springer-Verlag, Mar. 1990.
3. 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.
4. K. Kato and T. Masuda. Persistent Caching. *IEEE Transactions on Software Engineering*, 18(7):631–645, July 1992.
5. A. Kemper and G. Moerkotte. Access Support in Object Bases. In *Proc. of the ACM SIGMOD Conf.*, pages 290–301, 1990.
6. 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.
7. C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The ObjectStore database system. *Communications of the ACM*, 34(10), Oct. 1991.
8. D. Maier and J. Stein. Indexing in an Object-Oriented DBMS. In *2nd Int. Workshop on Object-Oriented Database Systems*, pages 171–182, Sept. 1986.
9. D. Maier, J. Stein, A. Otis, and A. Purdy. Development of an Object Oriented DBMS. In *Proc. the Int. Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pages 472–482, Portland, Oregon, Sept. 1986.
10. J. Orenstein, S. Haradhvala, B. Marguiles, and D. Sakahara. Query Processing in the ObjectStore Database System. In *Proc. of the ACM SIGMOD Conf.*, 1992.
11. E. Shekita and M. Carey. Performance Enhancement Through Replication in an Object-Oriented DBMS. In *Proc. of the ACM SIGMOD Conf.*, 1989.
12. O. Tsatalos, M. Solomon, and Y. Ioannidis. The GMAP: A Versatile Tool for Physical Data Independence. In *Proc. of the Int. VLDB Conf.*, Sept. 1994.
13. P. Valduriez. Join Indices. *ACM Transactions on Database Systems*, 12(2):218–246, June 1987.
14. P. Valduriez, S. Khoshafian, and G. Copeland. Implementation Techniques of Complex Objects. In *Proc. of the Int. VLDB Conf.*, pages 101–109, 1986.