Controlled Redundancy in Object Bases: Optimization Methods for Organizational Data Modelling

Alfons Kemper* Guido Moerkotte+ Michael Steinbrunn*

*Lehrstuhl für Informatik III
RWTH Aachen
Ahornstraße 55
W-5100 Aachen, Germany
[*kemper* | *steinbrunn*]*@informatik.rwth-aachen.de*

+Fakultät für Informatik
Universität Karlsruhe
Am Fasanengarten
W-7500 Karlsruhe, Germany
moer@ira.uka.de

Abstract

We review the definition and then analyse two optimization techniques that were developed for object-oriented databases. Both of these approaches are based on incorporating redundancy into the object base. However, the redundancy is entirely controlled by the system, rather than burdening the user with this task. One technique, called Access Support Relations, is based on redundantly maintaining frequently traversed object paths. The other, Function Materialization, is founded on precomputing particularly performance critical functions. In both cases, the redundantly maintained information can be viewed as an index to speed up object base queries. We discuss the exploitation of these optimization approaches on two sample, yet representative applications from the area of organizational data modelling.

1 Introduction

Object-oriented data models are emerging as the new generation database technology—especially for advanced, non-standard database applications. However, despite their flexibility and modelling power, it is widely agreed upon that the acceptance of object-oriented database technology will ultimately depend on its performance in comparison to established database systems—e.g., the relational systems.

In this paper we analyse two optimization concepts which are both founded on incorporating (system-) controlled redundancy in the object base. These optimization approaches which are tailored for object-oriented data models are:

1. Access Support Relations (ASRs), which are used to materialize frequently traversed reference chains and maintain them in separate storage structures, and
2. Function Materialization, which is based on precomputing performance critical functions.

Both approaches induce redundancy into the object base schema. However, the redundancy is implicitly controlled by the system, rather than burdening the user with the task of keeping redundant information consistent.

Furthermore, we will illustrate the two indexing methods on sample—yet, as we believe, representative applicational examples of organizational data modelling: the maintenance of (redundant) bidirectional references—e.g., in a part explosion application and the redundant materialization of complex virtual objects—e.g., the precomputation of a Department/Project/Employee matrix.

The remainder of this paper is organized as follows. In Section 2 we outline the two optimization concepts—details on theses indexing methods can be found in the cited literature. In Section 3 we analyse object-oriented representation of relationships and the virtues of Access Support Relations in optimizing the representation. Section 4 is devoted to the investigation of materialized functions in optimizing a representative example application with complex virtual objects. Section 5 concludes the paper.

2 GOM's Optimization Concepts Revisited

In this section, we shall briefly outline the concepts of Access Support Relations and Function Materialization to provide the basis for the discussion in the subsequent sections. Both concepts are part of our experimental object model GOM [1], and both hinge on storing redundant information in order to speed up user queries. Theses queries may either be formulated using the declarative, QUEL-like language GOMql, or

the procedural, C-like language GOMpl.

2.1 Access Support Relations

An Access Support Relation (ASR) is an index structure that supports the evaluation of so-called *path expressions*. We provide a short review of the main principles; details can be derived from [2, 3]. Access Support Relations constitute a generalization of binary join indices originally proposed for the relational model [4] and later extended for object models [5, 6]. Binary join indices are an extension of links, developed by Härder [7]. The binary links were also applied to object models in [8, 9].

For illustrating the main idea of access support relations let us introduce a sample object base schema modelling a department store—in the sequel we will call the schema *DeptStore*:

type EMP **is**
[Name: STRING;
WorksIn: DEPT;
Salary: INT;]

type MANAGER
supertype EMP **is**
[Cars: {CAR};]

type DEPT **is**
[Name: STRING;
Mgr: MANAGER;
Profit: INT;]

type CAR **is**
[License: STRING;
Make: STRING;
HorsePower: INT;]

A small extension of this object base schema is shown in Figure 1. The schema *DeptStore* contains the following path expression:

$$P \equiv \underbrace{\underbrace{\underbrace{\underbrace{EMP.WorksIn}_{DEPT}.Mgr}_{MANAGER}.Cars}_{CAR}.Make}_{STRING}$$

The underbraces indicate the type of the respective sub-path. Graphically, this path can be visualized as follows (single arrows correspond to single valued attributes, double arrows represent set valued attributes along the path):

$$EMP \xrightarrow{WorksIn} DEPT \xrightarrow{Mgr} MANAGER \xrightarrow{Cars} CAR \xrightarrow{Make} STRING$$

Let us now formulate a query that utilizes the above described path expression. Assume we want to retrieve the *EMP*loyees whose *Manager* drives (among others) a *Car* made by "Jaguar". The following QUEL-like query will determine this set of *EMP*loyees:

range e: EMP
retrieve e
where "Jaguar" **in** e.WorksIn.Mgr.Cars.Make

For a given *EMP*-binding of the range variable e the *STRING* values that can be reached via the path $e.WorksIn.Mgr.Cars.Name$ are inspected. That is, the database system determines—by traversing along the indicated reference chain—all strings and checks whether one of the strings equals "Jaguar". For example, for the binding $e = id_1$ the system determines $e.worksIn.mgr.cars.name = \{$"Jaguar", "BMW"$\}$.

With respect to our example object base in Figure 1 this query yields the result set $\{id_1, id_2, id_8\}$. Note, that the *MANAGER* instance identified by id_8 is contained in the result set—because of the substitutability of *MANAGER*s for *EMP*s.

The processing of this sample query causes tremendous costs in run time as our following analysis indicates. If the object base has no supplementary index structures that can be exploited for the evaluation of this query, the query processer has to inspect all objects of the types *CAR*, *MANAGER*, *DEPT* and *EMP* at least once. Thus, the run time of processing the query is (at least) proportional to:

$$\#(EMP) + \#(DEPT) + \#(MANAGER) + \#(CAR)$$

where $\#(t)$ denotes the cardinality of the type extension $ext(t)$.[1]

To support this query, we materialize the path in a separate storage structure, the so-called *Access Support Relation*. In particular, Figure 2 shows a part of the canonical extension of the ASR that corresponds to our sample object base *DeptStore*. Each row in the ASR corresponds to a path (reference chain) in the object base. If an attribute of a path is set-typed, the ASR contains as many rows as the cardinality of the set for this path. Objects are represented by their (unique) identifiers, whereas atomic values appear directly in the ASR (such as *STRING* in Figure 2).

This kind of ASR extension contains only complete paths that originate in objects of the leftmost column's type and lead "all the way" to objects of the rightmost column's type. Apart from this extension, in [2] we propose a total of four different extensions, where some may also include partial paths.

2.2 Function Materialization

To illustrate the basic idea of function materialization, let us assume that we have defined several object

[1] Actually, in this analysis we assumed that each object of type *CAR* is referenced by at least one object of type *MANAGER*, each *MANAGER* is referenced by at least one *DEPT* via the attribute *Mgr*, and each *DEPT* is referenced by at least one *EMP* via the attribute *WorksIn*. Otherwise, the processing time could be (a bit) lower.

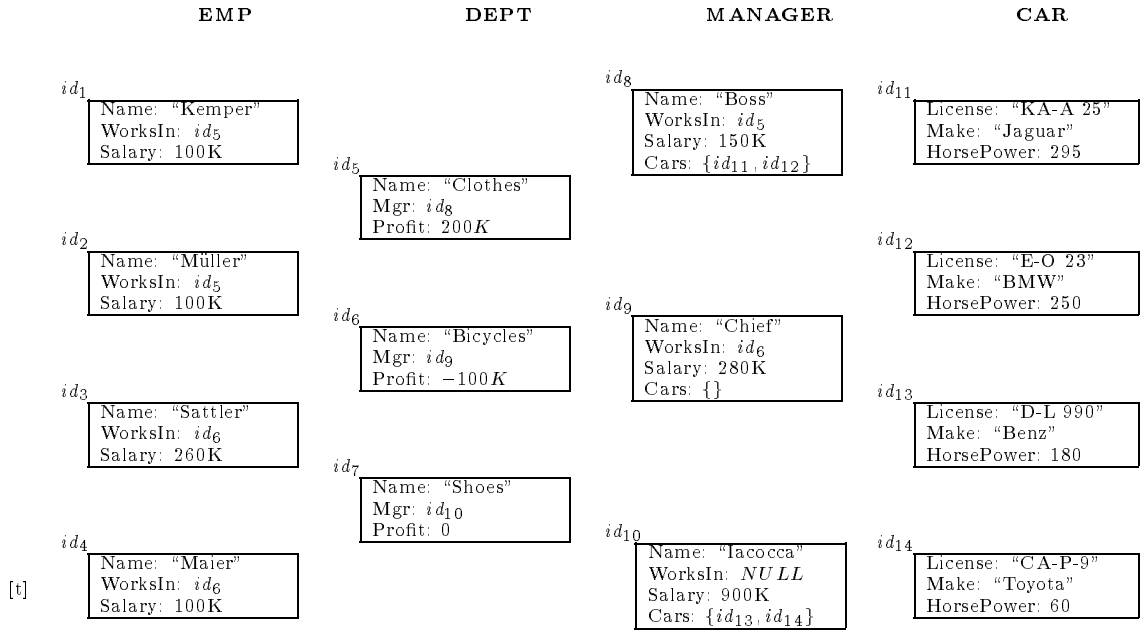


Figure 1: Sample extension of object base *DeptStore*

$[\![EMP.WorksIn.Mgr.Cars.Make]\!]_{can}$

S_0: OID_{EMP}	S_1: OID_{DEPT}	S_2: $OID_{MANAGER}$	S_3: OID_{CAR}	S_4: $STRING$
...	...	...	...	...
id_1	id_5	id_8	id_{11}	"Jaguar"
id_2	id_5	id_8	id_{11}	"Jaguar"
id_2	id_5	id_8	id_{12}	"BMW"
...	...	...	...	...

Figure 2: Canonical extension of the ASR for the object base *DeptStore*

types like *Cuboid*, *Vertex* and *Material* (cf. Figure 3); *volume* and *weight* are *Cuboid* methods with obvious purpose. Consider the query

range c: Cuboid
retrieve c
where c.volume > 20.0 **and** c.weight > 100.0

In order to execute this query, every *Cuboid* object has to be accessed and its *volume* and *weight* computed—there is no way to speed up the evaluation with an index. On the other hand, if *volume* and *weight* were "ordinary" attributes, we could provide indices to enhance efficiency. Function materialization, analogous to view materialization in relational database systems [10, 11], makes it possible to retrieve the precomputed result with much lesser cost, exactly like any other attribute [12]. It is also possible to define indices on the materialized results to support queries like the one above. Of course, there are additional efforts required to keep the materialized results up to date. In the following paragraphs, we shall summarize the basic concepts that ensure the integrity of materialized function results. It must be stressed that there exist several refinements in order to keep the rematerialization expenses to a minimum; details can be derived from [13], [14] and [15].

The *Generalized Materialization Relation* (GMR for short) is the data structure that holds materialized function results. In its simplest form, it contains the result for all possible parameter combinations. Therefore, only non-atomic arguments (i.e., object types) are permitted in this case to keep the size of the GMR within reasonable limits. As an example, consider the objects in Figure 3. Suppose that both *volume* and *weight* are methods of the object type *Cuboid* (objects id_1, id_2, id_3). Their implementation is not shown, but

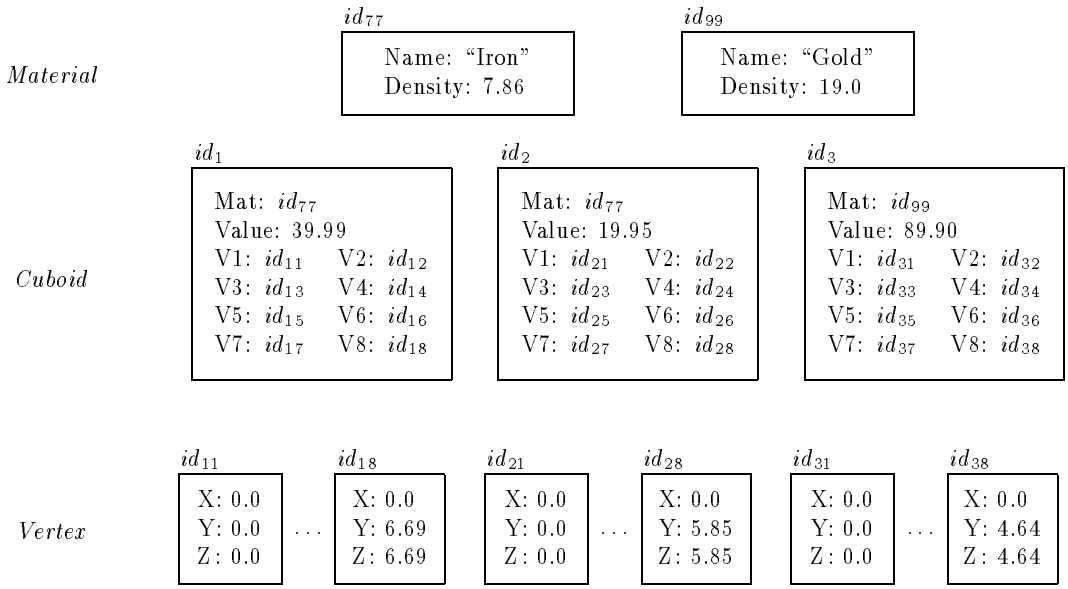


Figure 3: Sample Objects

$\langle\langle volume, weight \rangle\rangle$				
par	*volume*: *int*	V_1: *bool*	*weight*: *int*	V_2: *bool*
id_1	300.0	*true*	2358.0	*true*
id_2	200.0	*true*	1572.0	*true*
id_3	100.0	*true*	1900.0	*true*

Figure 4: GMR for materialized functions *volume* and *weight*

it is obvious that the function *volume* has to access each of the *Cuboid*'s *Vertex* attributes (V_1, ..., V_8) in order to compute the length of the edges, and thus the volume. The function *weight* would call *volume* and it would access the *Material* object in order to determine its density. As both of these functions do not have side effects, they are possible candidates for materialization.

Figure 4 shows the GMR for *volume* and *weight*; because they share the same argument, only one GMR is necessary. The first column *par* of the GMR consists of all possible arguments for the materialized functions; in other words, the projection π_{par}(GMR) is the cartesian product of all the parameter types' extensions. In our example, there is only one (implicit) argument, *Cuboid*, and the extension of *Cuboid* is $\{id_1, id_2, id_3\}$. Thus, the GMR contains exactly three rows with two materialized results for each one. The subsequent GMR columns contain the materialized results for the functions *volume* and *weight*, respectively. In addition, a validity flag (V_1 resp. V_2) is used to indicate whether the result is currently valid, i.e., consistent with the state of the object base.

Modifications that are made to the object base may cause one or more of the materialized results to become invalid. For instance, the value of $id_1.volume$ depends on the object id_1 and the *Vertex* instances id_{11}, ..., id_{18}; $id_1.weight$ depends on id_1, id_{11}, ..., id_{18} and id_{77}. The modification of one of these objects invalidates the appropriate materialized function result. Such modifications are called *relevant updates*, the objects itself *relevant objects* (for a certain materialized result).

To detect relevant updates, we have to mark the relevant objects for each materialized function result. Because the number of marks for a given object might be quite high, this information is maintained in a separate data structure, the so-called *Reverse Reference Relation (RRR)*. For references like the value id_{77} of the *Mat* attribute in the *Cuboid* object id_1, the RRR provides the reverse direction, hence its name. For example, when object id_{18} is modified, the materialized results for $id_1.volume$ and $id_1.weight$ have to be recomputed. As an alternative to the immediate update of the materialized result, it is possible to set the appropriate validity flag V to *false* in order to keep the GMR consistent—this is called *lazy rematerialization*.

Sometimes it is not necessary to recompute a result "from scratch." Consider, for example, the materialization of a method *cardinality* for set objects,

```
type T_left is                type T_right is              type T_R
body                          body                         with extension is
[ ...                         [ ...                        body
  R: T_right;                   R^-1: T_left;              [ left: T_left;
  ...]                          ...]                         right: T_right; ]
end type T_left;              end type T_right;            end type T_left;
```

(a) (b) (c)

Figure 5: Possible Modelling for a $1:1$ Relationship

```
type T_left is                type T_right is              type T_R
body                          body                         with extension is
[ ...                         [ ...                        body
  R: T_right;                   R^-1: {T_left};            [ left: T_left;
  ...]                          ...]                         right: T_right; ]
end type T_left;              end type T_right;            end type T_left;
```

(a) (b) (c)

Figure 6: Possible Modelling for a $1:N$ Relationship

```
type T_left is                type {T_right} is            type T_R
body                          body                         with extension is
[ ...                         [ ...                        body
  R: {T_right};                 R^-1: {T_left};            [ left: T_left;
  ...]                          ...]                         right: T_right; ]
end type T_left;              end type T_right;            end type T_left;
```

(a) (b) (c)

Figure 7: Possible Modelling for a $N:M$ Relationship

that is computed by counting all elements in the set. The insertion and deletion of set elements invalidates the materialized cardinality; however, it is not necessary to count all elements again if, say, one of them is deleted from the set.

This is the basic idea for so-called *compensating actions*, that the user may define for selected relevant updates (like insert/delete in the set example above). Compensating actions can greatly enhance the efficience of the materialization scheme; we shall return to this idea in Section 4.2.

3 Modelling Relationships in Object Bases

3.1 Representation Alternatives

The object-oriented data model offers several alternatives for modelling relationships. In particular, a binary relation can be modelled in three different ways: Figures 5–7 show the possibilities for $1:1$, $1:N$ and $N:M$ relationships between the two types T_{left} and T_{right}, respectively. In each figure, the three alternatives are shown: For instance, in Figure 5a objects of type T_{right} are referenced via the attribute R from objects of type T_{left}. In Figure 5b, T_{left} is referenced by T_{right}, and in Figure 5c objects of both types are referenced by a dedicated relationship object.

Each of these implementations favours a particular type of query: alternative (a) is advantageous if the traversal takes place in the direction T_{left} to T_{right}; the opposite is true for (b).

The third implementation, (c), is very well suited for determining which objects of type T_{left} and T_{right} are related. A realization like this resembles a class "relationship" that was proposed in [16]. All the operations and features that are attributed to such a class, such as indices for the object references, can be incorporated into this implementation. Also if the relation comprises attributes of its own, representation (c) is usually the best choice.

Often it is desirable to get the advantages of more than one of the realizations, e.g., if the number of traversals between the two participating objects is distributed evenly for each of the directions. This can be achieved by combining two implementations, e.g., (a) and (b), thus creating bidirectional references. But this means that information about the relationship is stored redundantly. The redundancy increases the efficiency of certain query operations, but it also jeopardizes the integrity of the database: The user has to make sure that in case of modifications all redundant information is updated in an atomic operation. If, instead, the database system itself would maintain this information, the risk of an integrity violation by (possibly buggy) application programs could be avoided.

Using Access Support Relations, this explicit utilization of redundant information in order to enhance the efficiency of query processing is not necessary. The user can implement the realization that is most adequate for modelling his application; if the performance is not satisfactory, the system can be tuned according to the requirements. The redundant information thus introduced is taken care of by the ASR management—the user need not worry about preserving integrity. This strategy is superior to the relationship class in [16], because ASRs need not be maintained by the application programmer—only the actually chosen representation of the relationship has to be considered, which is not necessarily the alternative (c). We shall illustrate the advantages of this approach by means of an often-encountered application example.

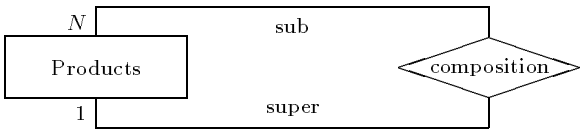


Figure 8: ER Representation of *composition*

3.2 An Example Application: Part Explosion

An example that demonstrates the benefits and hazards of redundancy is the creation of bidirectional references. In most object base systems references may only be traversed in one direction. If the user requires bidirectional references, he has to implement the necessary schema additions and management algorithms himself.

As we expounded in Section 2.1, ASRs can be used as a "shortcut" for path expressions. But ASRs can also be used to traverse a path expression from the right-hand side. A possible application is the modelling of a relationship *composition* that relates *Products* as sub- and super-parts. The ER-schema of this application is depicted in Figure 8. There exist several possibilities for this kind of relationship: If the application mainly requires to determine the constituent parts of a given *Product*, the realization on the right-hand side of Figure 9 is adequate. On the other hand, if the possible composite *Products* for a given *Product* is of chief interest, the left-hand side is appropriate. Because each *Product* can (in this modelling) only be used in one composite *Product*, there is just a single attribute (*super*) necessary to relate a *Product* to its composite *Product*. If, on the other hand, a composite *Product* has to refer to its parts, we need a set-valued attribute (*sub*) that contains all constituents.

Each of these alternatives favours one query type, whereas the other one would be very expensive (it required repeated iteration over the entire extension of *Products*). To introduce both reference directions, we could combine the two realizations in Figure 9, i.e., we maintain the pointer to the *super* part as well as the set of *sub* parts; but this would also mean that we place redundant information into the database.

Thus, this approach would put the integrity of the database at risk in case of modifications. Every application that changes something has to make sure that the references are updated properly, which also means that this task has to be coded anew for each of these applications. It would be more advantageous if the database system itself could take care of the maintenance of integrity. In GOM, there are two ways to achieve this:

```
type Product is
      ⋮
operations
  refine insert: Product → void
         code ExtendedInsert;

      ⋮
implementation
  define ExtendedInsert(subpart) is
  begin
    subpart.super := self;
    self.insert(subpart);
  end define ExtendedInsert;
      ⋮
```

Figure 10: Refinement of the *insert* method

1. The appropriate update methods are refined in order to make the necessary checks (in this case, assignment of the *super* attribute and insertion and deletion of the *sub* set).

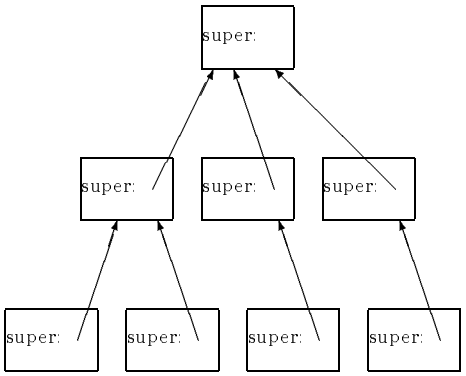
2. One of the representations in Figure 9 is chosen to realize one of the reference directions explicitly. Traversal in the opposite direction is supported by an ASR.

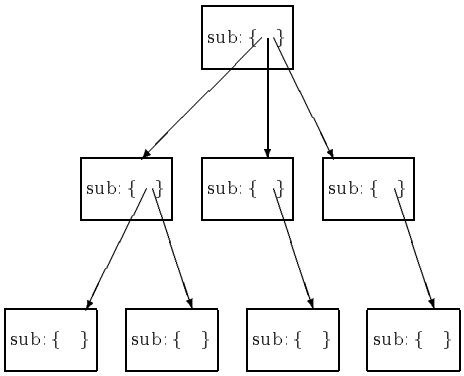
The first possibility, refinement of update methods, can be used to keep any information consistent that depends on the particular item. Therefore, if properly written, inconsistencies are virtually impossible. A refinement of the *insert* method for the *sub* set could be implemented as in Figure 10; for the *delete* method it would look quite similar. However, this approach suffers from several disadvantages:

- The user has to determine all those methods that have to be refined.

- Because the information about an integrity constraint is scattered over the various methods, maintenance tends to become difficult, especially in case of more than one constraint.

- It is not possible to defer the update of dependent information until this information is actually needed (e.g., deferring the update of the *super* attribute until it is accessed).






```
type Product is
body
[
        ...
        super: Product;
        ...
]
end type Product;
```

```
type Product is
body
[
        ...
        sub: {Product};
                !! Set of Product
        ...
]
end type Product;
```

Figure 9: Alternative Representations of *composition*

- A method of one object type (e.g., *ExtendedInsert*) has to know about an attribute of a second object type (e.g., *super*) and has to be granted write access, which makes proper encapsulation and information hiding impossible.

The second possibility avoids these problems, because the backward references are maintained automatically by the ASR management in the database system itself. The only thing the user has to do is defining the necessary ASR. In our case, we need an ASR for the paths between two *Products* (via the *sub* attribute). Figure 11 depicts a sample *Product* tree and Figure 12 the canonical extension of the ASR that materializes all *Product/Product* paths. Without ASR, finding the *Product* that has a spoke as one of its parts would require an exhaustive search of the set of all *Products*.

With the ASR, we can use the index on the right-hand column (cf. Figure 12) to identify the boxed tuple that indicates that the spoke is a part of the wheel.

Computing the transitive hull, e.g., determining *all* subparts of a given superpart, can be performed by repeated consultation of the ASR—without ever having to access the actual objects. In addition, the ASR needs much less storage space than the objects itself, so the paging behaviour is likely to be much better than it would be if we traversed the paths in the ob-

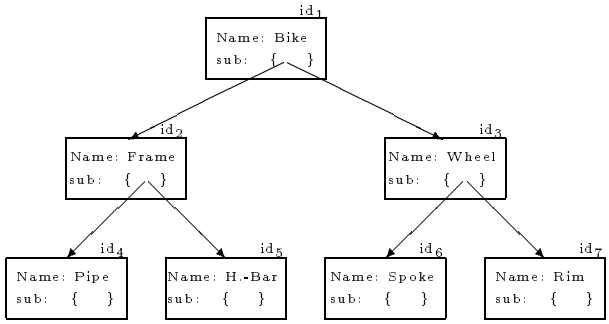


Figure 11: Sample *Product* tree

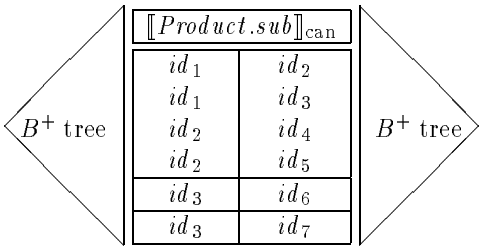


Figure 12: ASR for the sample *Product* tree

$[\![Product.sub^*]\!]_{\text{can}}$	
id_1	id_2
id_1	id_3
id_1	id_4
id_1	id_5
id_1	id_6
id_1	id_7
id_2	id_4
id_2	id_5
id_3	id_6
id_3	id_7

Figure 13: Transitive Hull ASR for the sample *Product* tree

ject base. This is also true for the opposite direction: If we are interested in all the base parts that make up a bicycle, we can deduce this information entirely from the ASR. It is even possible to define a "transitive hull" ASR $[\![Product.sub^*]\!]$ that contains all transitive relations between superpart and subpart of every *Product*/Sub-*Product* chain (Figure 13). The problem of finding the base parts for our bicycle is thus reduced to a single consultation of the hull ASR, namely selecting all tuples with id_1 as their first attribute.

Thus, the redundant information that is kept in the ASR is capable of increasing query efficiency tremendously, whereas the burden of maintaining this information and keeping it consistent with the object base is completely removed from the user's shoulders. The ASR management in the database system takes care of ASR updating whenever necessary.

4 Virtual Objects

Another application of controlled redundancy is the materialization of function results. Again, the built-in management of redundant information avoids the sacrifice of integrity in order to achieve efficiency.

4.1 Illustrative Example

We shall illustrate the benefits of function materialization with the help of the example in Figure 14. It models a *Company* that consists of a set of *Departments*, which in turn employ *Employees*. On the other hand, the *Company* carries out *Projects*, where *Programmers*, i.e., *Employees*, work on.

An application program may find it desirable to determine which *Employees* in which *Departments* work on which *Projects*. In other words, a matrix

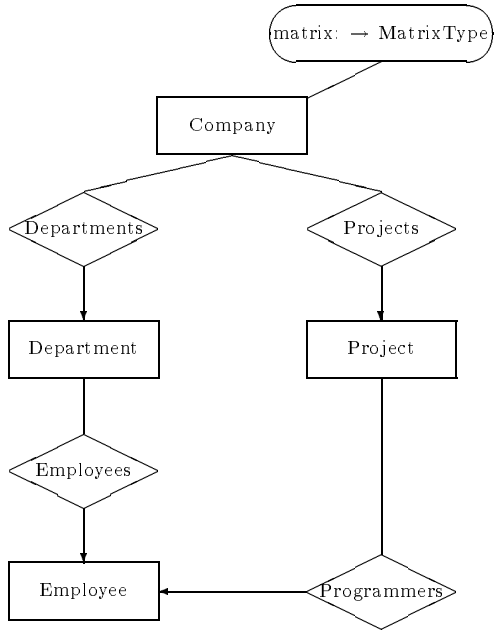


Figure 14: ER representation for *Company*

		Departments			
		D_1	D_2	...	D_n
Projects	P_1	$\{e_1, e_2\}$			
	P_2	$\{e_2\}$			$\{e_5, e_6\}$
	⋮				
	P_m		$\{e_3, e_4\}$		$\{e_6, e_7, e_8\}$

Figure 15: *Department*/*Project*/*Employee* Matrix

(*Department*, *Project*) with sets of *Employees* as its components is needed (cf. Figure 15).

All the information that is required to compute the matrix is already present in the database—but the cost for frequent invocation is likely to be intolerable as the complexity is $O(\#(Company.Departments) \cdot \#(Company.Projects))$. The user might be tempted to cache the result within his application, so it is possible that the program bases its computations on outdated information. If the matrix were included in the schema (e.g., as an attribute for *Company*), a similar problem would arise: each application would have to make sure that the matrix is updated whenever a relevant modification is performed. The resulting risk for the database's integrity has been pointed out already in Section 3.2; one of the possible solutions, the refinement of update methods, could also be used in

this context. However, the drawbacks mentioned there apply here as well, so we propose another approach.

The object type *Company* is supplied with the method *matrix* that computes the *Department/Project* matrix. The method *matrix* is materialized, thus the expenses for updating the resulting *matrix* object are kept to a minimum. Although the matrix object constitutes a redundancy, the integrity of the database is preserved by the GMR management. The user who invokes the *matrix* method can rely on the validity and consistence of the result; in addition, he may discard it immediately after use without performance penalty for a repeated invocation in the future.

Depending on the access profile and real-time requirements of the application, the GMR mechanism permits tuning of the rematerialization behaviour according to the user's needs. If real-time capabilities of the application that makes use of the materialized function are requested, a stored result that has become invalid due to some modifications in the object base must be recomputed at once (immediate rematerialization) to avoid delays upon calling the method.

If, on the other hand, overall performance is the chief interest, a relevant modification just has to set the GMR's validity flag V to *false*; the rematerialization of an invalidated result is delayed until the next invocation of *matrix*. This mode is especially useful for applications with high update probability, because two subsequent relevant modifications do not cause two recomputations of the result.

Materialization of complex function results can be characterized as *virtual objects*: Although the user can handle these objects as usual, they are not permanent members of the database. It is not guaranteed that subsequent invocations of a certain method with complex result type yield the same object, i.e., an object with the same identifier. This situation resembles virtual memory management: The same virtual address is not necessarily always mapped to the same physical address, it is not even guaranteed that a certain virtual address can be mapped to a physical address at any given time. It is, however, guaranteed that upon access a mapping can be established—exactly like a (lazy) rematerialized result, that is only guaranteed to be valid upon access.

4.2 Optimization Approaches

The solution outlined in the previous section enables the user to introduce redundant information controlled by the database system, that can be used to speed up query processing. When modifications are made, possibly a materialized function result (i.e. the redundant information) has to be updated. If this happens, the function in question has to be called in order to recompute the result that has become invalid. However, not always is it necessary to recompute the result by means of invoking the code of the materialized function. In many cases, it is possible to compensate the impact of the modification on the materialized result with much less effort. An example of such a so-called *compensation action* would be the incrementation of a materialized set cardinality whenever an element is inserted. Compensating actions can be defined for all of a materialized function's relevant modification operations. Although the user needs detailed knowledge of the materialized function's working principles, compensating actions can increase the speed of the rematerialization procedure considerably.

```
declare compensate_remove_programmers: → MatrixType;
                    ⋮
define compensate_remove_programmers(company, oldmatrix,
       programmer_to_remove) is
begin
  foreach (dept in company.departments) do
    oldmatrix[self.number, dept.number]
                          .remove(programmer_to_remove);
  return(oldmatrix);
end define compensate_remove_programmers;
```

Figure 16: Compensating action for *prgrammers.remove*

In our *Company* application, there exist several modification operations that affect the result of the materialized matrix. These include, among others, insertion and deletion of elements in *Departments* and *Projects* as well as the *Departments' Employees* and the *Projects' Programmers*. For example, a compensating action for the operation *Programmers.remove* can be implemented as in Figure 16. The GMR management calls the appropriate compensation action(s) before the actual modification takes place, so the compensating action can make use of the old state in order to compute the new, updated result that represents the new state. The arguments for the compensating action are:

- the materialized function's arguments,
- the old materialized result,
- the modification operation's arguments.

The return value is the new result. The set iteration in the program fragment in Figure 16 is necessary because the *Programmer*'s *Department* cannot be deduced from the arguments.

As a refinement of this approach, there is also a possibility to compensate the effects of higher-level modifications. Under the assumption that the *Company* thus modelled is completely encapsulated and permits access only via a set of (*Company*-associated) operations, we could compensate a modification like "reassign-programmer". This operation would assign new duties to a programmer by transferring him from one of the *Projects* to another—the compensating action can do exactly the same in the matrix object.

5 Conclusion

In this paper we first reviewed two previously proposed optimization methods for object bases. We then analysed the applicability of Access Support Realtions and Materialized Functions in order to facilitate the task of solving certain often-encountered problems in object-oriented modelling. Both of these concepts introduce redundant information, but the redundancy is carefully controlled by the database system itself. The user does not have to sacrifice a clear, straightforward schema by introducing obscure elements for the sake of performance gain. The presence of additional information is not directly visible at the user interface. The decision whether to make use of Access Support Relations or Materialized Functions is completely transparent to the application programmer, so it is possible to tune the system performance without having to modify existing software. We illustrated by means of two, as we deem for object bases typical examples how the utilization of system-controlled, materialized information can be applied. Because it results in designs that are simpler and therefore easier to understand and maintain, the benefits are not merely on the performance side.

References

[1] A. Kemper, G. Moerkotte, H.-D. Walter, and A. Zachmann, "GOM: a strongly typed, persistent object model with polymorphism," in *Proc. der GI-Fachtagung Datenbanken in Büro, Technik und Wissenschaft (BTW)*, (Kaiserslautern), pp. 198–217, Springer-Verlag, Informatik-Fachberichte Nr. 270, Mar 1991.

[2] A. Kemper and G. Moerkotte, "Access support in object bases," in *Proc. of the ACM SIGMOD Conf. on Management of Data*, (Atlantic City, NJ), pp. 364–374, May 1990.

[3] A. Kemper and G. Moerkotte, "Access Support Relations: an indexing method for object bases," *Information Systems*, vol. 17, no. 2, 1992.

[4] P. Valduriez, "Join indices," *ACM Trans. on Database Systems*, vol. 12, pp. 218–246, Jun 1987.

[5] G. Copeland and S. Khoshafian, "A decomposition storage model," in *Proc. of the ACM SIGMOD Conf. on Management of Data*, (Austin, TX), pp. 268–279, May 1985.

[6] P. Valduriez, S. Khoshafian, and G. Copeland, "Implementation techniques of complex objects," in *Proc. of The Conf. on Very Large Data Bases (VLDB)*, (Kyoto, Japan), pp. 101–110, Aug 1986.

[7] T. Härder, "Implementing a generalized access path structure for a relational database system," *ACM Trans. on Database Systems*, vol. 3, pp. 285–298, Sep 1978.

[8] D. Maier and J. Stein, "Indexing in an object-oriented DBMS," in *Proc. IEEE Intl. Workshop on Object-Oriented Database Systems, Asilomar, Pacific Grove, CA* (K. R. Dittrich and U. Dayal, eds.), pp. 171–182, IEEE Computer Society Press, Sep 1986.

[9] E. Bertino and W. Kim, "Indexing techniques for queries on nested objects," *IEEE Trans. Knowledge and Data Engineering*, vol. 1, pp. 196–214, Jun 1989.

[10] J. A. Blakeley, P. A. Larson, and F. W. Tompa, "Efficiently updating materialized views," in *Proc. of the ACM SIGMOD Conf. on Management of Data*, (Washington, D.C.), pp. 61–71, 1986.

[11] E. Hanson, "A performance analysis of view materialization strategies," in *Proc. of the ACM SIGMOD Conf. on Management of Data*, (San Francisco, CA), pp. 440–453, May 87.

[12] M. Stonebraker, J. Anton, and E. Hanson, "Extending a database system with procedures," *ACM Trans. Database Systems*, vol. 12, pp. 350–376, Sep 1987.

[13] A. Kemper, C. Kilger, and G. Moerkotte, "Materialization of Functions in Object Bases: Design, Implementation and Assessment," *IEEE Trans. Knowledge and Data Engineering*, 1992. (to appear).

[14] A. Kemper, C. Kilger, and G. Moerkotte, "Function materialization in object bases," in *Proc. of the ACM SIGMOD International Conference on Management of Data*, pp. 258–268, 1991.

[15] M. Steinbrunn, "Entwurf und Realisierung der Funktionenmaterialisierung in GOM," Master's thesis, Universität Karlsruhe, Fakultät für Informatik, D-7500 Karlsruhe, Feb 1991.

[16] J. Rumbaugh, "Relations as semantic constructs in an object-oriented language," in *Proc. of the ACM Conf. on Object-Orientd Programming Systems and Languages (OOPSLA)*, (Orlando, Fl), pp. 466–481, Oct 1987.