Function Materialization in Object Bases: Design, Realization and Evaluation*

Alfons Kemper† *Christoph Kilger*‡ *Guido Moerkotte*‡

†RWTH Aachen
Lehrstuhl für Informatik III
5100 Aachen, Germany
kemper@informatik.rwth-aachen.de

‡Universität Karlsruhe
Fakultät für Informatik
7500 Karlsruhe, Germany
[*kilger*|*moer*]@ira.uka.de

*This work was partially supported by the German Research Council DFG under contract number SFB 346 and Ke 401/6-1. An excerpt presenting selected issues appeared in: *Proc. ACM SIGMOD Conf. on Management of Data, Denver, CO, May 1991* [10].

Abstract

View materialization is a well-known optimization technique of relational database systems. In this work we present a similar, yet more powerful optimization concept for object-oriented data models: *function materialization*. Exploiting the object-oriented paradigm—namely *classification*, *object identity*, and *encapsulation*—facilitates a rather easy incorporation of function materialization into (existing) object-oriented systems. Only those types (classes) whose instances are involved in some materialization are appropriately modified and recompiled—thus leaving the remainder of the object system invariant. Furthermore, the exploitation of encapsulation (information hiding) and object identity provides for additional performance tuning measures which drastically decrease the invalidation and rematerialization overhead incurred by updates in the object base. First, it allows to cleanly separate the object instances that are irrelevant for the materialized functions from those that are involved in the materialization of some function result and, thus, to penalize only those involved objects upon update. Second, the principle of information hiding facilitates fine-grained control over the invalidation of precomputed results. Based on specifications given by the data type implementor the system can exploit operational semantics to better distinguish between update operations that invalidate a materialized result and those that require no rematerialization. The paper concludes with a quantitative analysis of function materialization based on two sample performance benchmarks obtained from our experimental object base system GOM.

Index Terms Access methods, Query processing, Indexing methods, Database-programming languages, Object-oriented languages.

1 Introduction

Once the initial "hype" associated with the object-oriented database systems settles the prospective users—especially those from the engineering application domains—will evaluate this new database technology especially on the basis of performance. Our experience with engineering database applications indicates that no engineer is willing to trade performance for functionality. The fate of object-oriented database systems will, therefore, largely depend on their performance relative to currently employed database technology, e.g., the relational database systems. It is not sufficient to merely rely on existing optimization techniques developed in the relational context and to adapt these to the needs of object-oriented systems. Of course, the large body of knowledge of optimization techniques that was gathered over the last 15 years in the relational area provides a good starting point. But lastly, only those optimization techniques that are specifically tailored for the object-oriented model and, thus, exploit the full potential of the object-oriented paradigm will yield—the much-needed—drastic performance improvements.

In this paper we describe one (further) piece in the mosaic of performance enhancement techniques that we incorporated in our experimental object base system GOM [14]: the *materialization of functions*, i.e., the precomputation of function results. Materialization—just like indexing—is based on the assumption that the precomputed results are eventually utilized in the evaluation of some associative data access. Function materialization is a dual approach to our previously discussed indexing structures, called *Access Support Relation* [12, 11] which constitute materializations of heavily traversed path expressions that relate objects along attribute chains.

Exploiting the potentials of the object-oriented paradigm, especially the classification of objects into *types*, *object identity*, and the principle of *encapsulation* in conjunction with *information hiding* facilitates a very modular design of function materialization. Our approach is based on the modification and—subsequent—recompilation of those type schemes whose instances are involved in the materialization of a function result; thus leaving the remainder of the object system invariant. This makes it easy to incorporate our approach even into existing object base systems since only very few system modules have to be modified while the kernel system remains largely unchanged.

Similarly to indexing, function materialization induces an overhead on update operations. The primary challenge in the design of function materialization is the reduction of the invalidation and rematerialization overhead. In this respect function materialization is related to relational view materialization, as proposed in [3, 4]. In this work algorithms are given to detect the relevance or irrelevance of updates to materialized views, and to rematerialize a view without having to re-evaluate the expression defining the view. In the work on snapshots [1] a technique is discussed to materialize and periodically update the result of relational expressions. Further work exists for extended relational models, in particular the POSTGRES data model [21]. In POSTGRES an attribute of type POSTQUEL may consist of a relational query that has to be evaluated upon referencing. Much work has been spent on optimizing the evaluation of queries accessing such POSTQUEL attributes, e.g., [6, 7, 9, 18, 19, 20], by caching the results.

The above cited work is similar to ours with respect to the general idea of precomputing results. However, exploiting object-oriented features facilitates a much finer-grained control over rematerialization requirements of precomputed results in our approach than is possible in relational view materialization and extended relational caching:

1. We can cleanly separate those object instances that are involved in the materialization of a function result from non-involved objects. Thus, the penalty incurred by the need to rematerialize a result can be restricted to the involved objects.

2. Within those objects that are involved in some materialization, we can decide in which function materialization they have been involved and which attributes are relevant for the respective function materialization.

3. Utilizing information hiding we can exploit operational semantics in order to reduce the rematerialization overhead even further. For example, in geometric modeling the data type implementor could provide the knowledge that *scale* is the only geometric transformation that could possibly invalidate a precomputed *volume* result while *rotate* and *translate* leave the materialized *volume* invariant.

4. By providing specialized (compensating) functions that rematerialize results at lower costs than invocations of the materialized function, update costs can further be decreased.

These tuning measures suggest that we can provide function materialization at a much lower update penalty than relational view materialization techniques can possibly achieve—which is backed by our quantitative analysis. This makes function materialization even feasible for rather update-intensive applications.

The remainder of this paper is organized as follows. In the next section we briefly review our object model GOM. Then, in Section 3 the static aspects of function materialization are presented. In Section 4 we deal with the dynamic aspects of function materialization: the mechanisms to keep materialized results up to date while the state of the object base is being modified. Reducing the update overhead is the subject of Section 5. In Section 6 we introduce the restricted materialization which facilitates the materialization of functions having atomic argument types. In Section 7 we provide a quantitative analysis of function materialization based on two sample benchmarks, one derived from computer geometry and the other based on a more traditional administrative application. Section 8 concludes this paper with a summary and an outlook into future work. In the Appendix a formal method for analyzing materialized functions is presented that forms the basis for our invalidation and rematerialization mechanism.

2 GOM: Our Object-Oriented Data Model

In essence, GOM provides all the compulsory features identified in the "Manifesto"[1] [2] in one orthogonal syntactical framework. GOM supports single *inheritance* coupled with *subtyping* and *substitutability* under *strong typing*: a subtype instance is always substitutable for a supertype instance. To enforce strong typing all database components, e.g., attributes, variables, set- and list-elements, are constrained to a particular type or a subtype thereof. GOM supports *object identity* in such a way that the OID of an object is guaranteed to remain invariant throughout its lifetime. Objects are referenced via their object identifier; referencing and dereferencing is implicit in GOM.

The *structural* description of a new object type can be either a *tuple*, a *set*, or a *list*. A *tuple* consists of a collection of typed attributes. The tuple constructor is denoted as

[1] Albeit the design of GOM was carried out before the "Manifesto" was written.

$[a_1 : t_1, \ldots, a_n : t_n]$ for unique attribute names a_i and, not necessarily different, type names t_i. *Set* and *list* types are denoted as $\{t\}$ and $\langle t \rangle$, respectively, where t is a type name. Objects of type $\{t\}$ or $\langle t \rangle$ may only contain elements of type t or subtypes thereof. Lists are analogous to sets except that an order is imposed upon the elements and duplicate elements are possible. A new type is introduced using the *type definition frame* which is illustrated on the example types *Vertex* and *Material* below:

```
type Vertex is
    public X, set_X, Y, set_Y, Z, set_Z,
           translate, scale, rotate, dist
    body [X, Y, Z: float;]
    operations
       declare translate: Vertex → void;
       declare scale: Vertex → void;
       declare rotate: float, char → void;
       declare dist: Vertex → float;
    implementation
       ...
end type Vertex;
```

```
type Material is
    public SpecWeight, set_SpecWeight,
           Name, set_Name
    body [Name: string;
          SpecWeight: float;]
end type Material;
```

The **public** clause lists all the type-associated operations that constitute the interface of the newly defined type. GOM enforces information hiding by object encapsulation, i.e., only the operations that are explicitly made **public** can be invoked on instances of the type. However, for each attribute A two built-in operations named A to read the attribute and *set_A*[2] to write the attribute are implicitly provided. It is the type designer's choice whether these operations are made public by including them in the **public**-clause. We trust that the essential parts of the type definitions *Vertex* and *Material* are self-explanatory. *Material* is a tuple-structured type that has no explicitly defined behavior; it merely provides for access to and modifying of the two attributes *Name* and *SpecWeight*.

Based on these auxiliary type definitions we can now define the type *Cuboid* (see Figure 1) which serves as the running example throughout the remainder of this paper. We have intentionally made all parts of the structure of a *Cuboid* visible (**public**) to the clients of the type, e.g., all the boundary *Vertex* objects, *V1*, ..., *V8* are accessible and, therefore, directly modifiable. This is needed to demonstrate our function materialization approach in its full generality. Later on (in Section 5), we will refine the definition of *Cuboid* by hiding many of the structural details of the *Cuboid* representation—and, thus, drastically decrease the invalidation penalty of many update operations.

Note that the function *distance* depends on the object types *Cuboid* and *Robot*—the type definition of *Robot* is not outlined here. A sample database is shown in Figure 2. The object identifiers are denoted by id_1, id_2, ... In this diagram we also show two set-structured instances id_{59} of type *Workpieces* and id_{60} of type *Valuables* whose types were not introduced. Sets of type *Workpieces* contain *Cuboids*, that are used in some manufacturing process, whereas sets of type *Valuables* contain *Cuboids*, which are interesting because of their value, e.g., gold ingots. Interesting functions for *Workpieces* instances are: *total_volume* and *total_weight*; *Valuables* instances respond to the function *total_value*.

[2]Actually, in GOM a more elegant mechanism to realize value returning and value receiving operations is provided (see [14]).

```
type Cuboid supertype ANY is   !! ANY is the implicit supertype of all types
   public length, width, height, volume, weight, rotate, scale, translate, distance
            V1, set_V1, ..., V8, set_V8, Value, set_Value, Mat, set_Mat
   body [V1, V2, V3, V4, V5, V6, V7, V8: Vertex;
           Mat: Material; Value: decimal;]
   operations
      declare length: → float;             !! V1.dist(V2)
      declare width: → float;              !! V1.dist(V4)
      declare height: → float;             !! V1.dist(V5)
      declare volume: → float;
      declare weight: → float;
      declare translate: Vertex → void;
      declare scale: Vertex → void;
      declare rotate: char, float → void;
      declare distance: Robot → float;    !! Robot is defined elsewhere
   implementation
      define length is
         return self.V1.dist(self.V2);   !! delegate the computation to Vertex V1
      ...
      define volume is
         return self.length * self.width * self.height;
      define weight is
         return self.volume * self.Mat.SpecWeight;
      define translate(t) is
         begin
            self.V1.translate(t);   !! delegate translate to Vertex instance V1
            ...
            self.V8.translate(t);   !! delegate translate to Vertex instance V8
         end define translate;
      ...
end type Cuboid;
```

Figure 1: Skeleton of the Type Definition *Cuboid*

A more detailled description of GOM can be found in [13]—here, we only reviewed those features essential for function materialization.

3 Static Aspects of Function Materialization

Consider the above definition of the type *Cuboid* with the associated functions *volume* and *weight*. Assume that the following query, which is phrased in the QUEL-like query language GOMql, is to be evaluated:

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

For the evaluation of this query the selection predicate has to be evaluated by invoking the functions *volume* and *weight* for each *Cuboid* instance. To expedite the evaluation of this query the results of *volume* and *weight* can be precomputed: we call this the

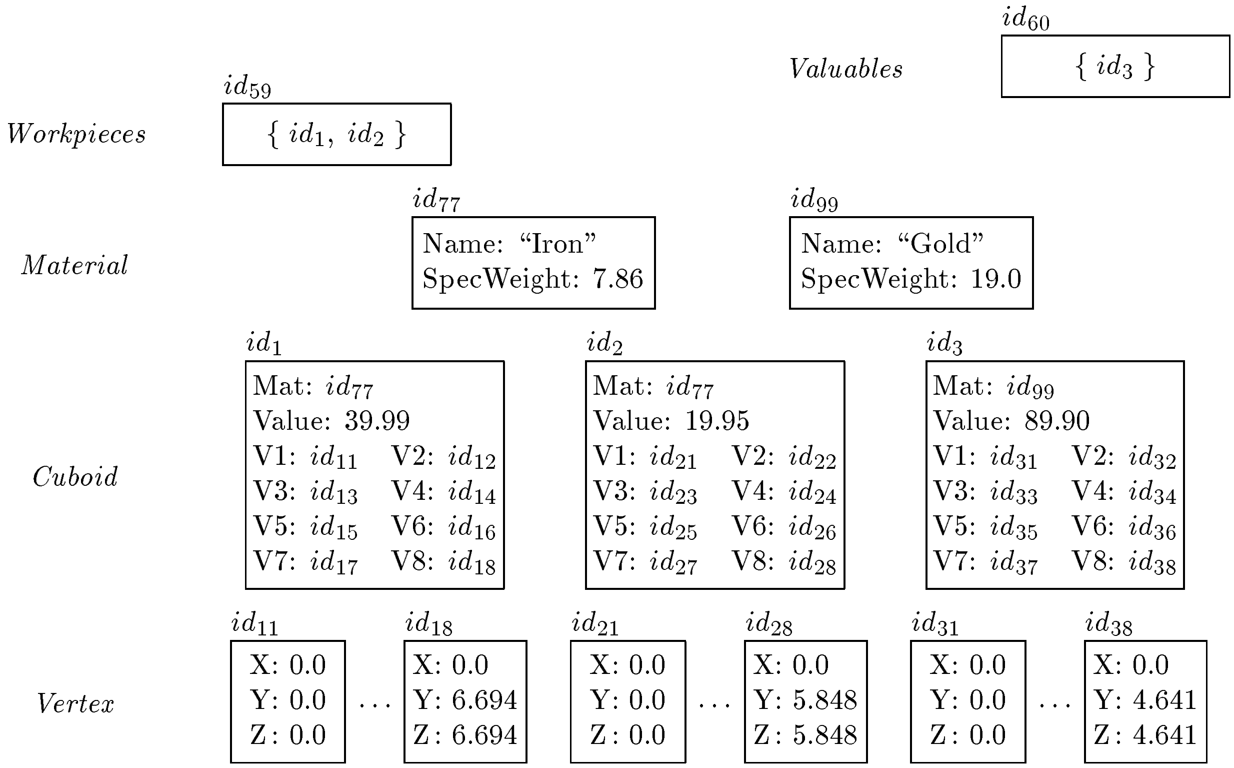


Figure 2: Database Extension of Example Schema

materialization of the functions *volume* and *weight*. In GOMql, the materialization of the functions *volume* and *weight* is initiated by the following statement:

range c: Cuboid
materialize c.volume, c.weight

In this statement the variable *c* is bound to the extension of type *Cuboid*, i.e., the set of instances of type *Cuboid*. Thus, the results of *volume* and *weight* are precomputed for each *Cuboid* instance. Of course, the variable *c* could also be bound to some set- or list-structured object with elements of type *Cuboid*. However, in the subsequent definitions we consider only variables bound to type extensions but these definitions could easily be extended to include variables bound to sets or lists.

After the materialization of *volume* and *weight* the above query can use the precomputed results instead of invoking the functions *volume* and *weight* for each *Cuboid* instance. This query is called a *backward query* as it uses a predicate on the function results to select the corresponding argument objects. As the evaluation of this query needs the *volume* and *weight* of all *Cuboid* instances it would be advantageous to have these functions precomputed for all argument objects.

Another kind of queries that can be supported by function materialization are *forward queries* which are characterized by the retrieval of function results of particular objects. For example, consider the forward query

range c: MyValuableCuboids
retrieve sum(c.weight)

where *MyValuableCuboids* is a variable of type *Valuables*. This query could exploit precomputed results of the function *weight*. However, in this case, it is not necessary to have the function *weight* precomputed for all *Cuboid* instances but only for those present in the set *MyValuableCuboids*.

Besides forward and backward queries there is a large variety of queries that are able to exploit precomputed function results. In Section 3.2 we discuss the support of these queries by materialized functions.

3.1 Storing Materialized Results

There are two obvious locations where materialized results could possibly be stored: in or near the argument objects of the materialized function or in a separate data structure. Storing the results near the argument objects means that the argument and the function result are stored within the same page such that the access from the argument to the appropriate result requires no additional page access. In general, storing results near the argument objects has several disadvantages:

- If the materialized function $f : t_1, \ldots, t_n \rightarrow t_{n+1}$ has more than one argument ($n > 1$) one of the argument types must be designated to hold the materialized result. But this argument has to maintain the results of all argument combinations—which, in general, won't fit on one page.

- Clustering of function results would be beneficial to support selective queries on the results. But this is not possible if the location of the materialized results is determined by the location of the argument objects.

Therefore, we chose to store materialized results in a separate data structure disassociated from the argument objects. This decision is also backed by a quantitative analysis undertaken in the extended relational system POSTGRES by A. Jhingran [9] where separate caching (CS) of precomputed POSTQUEL attributes proved to be almost always superior to caching within the tuples (CT).

If several functions are materialized which share all argument types, the results of these functions may be stored within the same data structure. This provides for more efficiency when evaluating queries that access results of several of these functions and, further, avoids to store the arguments redundantly. These thoughts lead to the following definition:

Definition 3.1 (Generalized Materialization Relation, GMR)
Let $t_1, \ldots, t_n, t_{n+1}, \ldots, t_{n+m}$ be types and let $f_1, \ldots, f_m$ be side-effect free functions with $f_j : t_1, \ldots, t_n \rightarrow t_{n+j}$ for $1 \leq j \leq m$. Then the generalized materialization relation $\langle\!\langle f_1, \ldots, f_m \rangle\!\rangle$ for the functions $f_1, \ldots, f_m$ is of arity $n + 2 * m$ and has the following form:

$$\langle\!\langle f_1, \ldots, f_m \rangle\!\rangle \; : \; [O_1 : t_1, \ldots, O_n : t_n, f_1 : t_{n+1}, V_1 : bool, \ldots, f_m : t_{n+m}, V_m : bool] \qquad \square$$

Intuitively, the attributes $O_1, \ldots, O_n$ store the arguments (i.e., values if the argument type is atomic or references to objects if the argument type is complex); the attributes $f_1, \ldots, f_m$ store the results or—if the result is of complex type—references to the result objects of the invocations of the functions $f_1, \ldots, f_m$; and the attributes $V_1, \ldots, V_m$ (standing for *validity*) indicate whether the stored results are currently valid.

In the beginning we will discuss only the materialization of functions having complex argument types. As can easily be seen it is not practical to materialize a function for all values of an atomic argument type, e.g., *float*. Therefore, we postpone the discussion of materialized functions with atomic argument types to Section 6 where *restricted GMRs* are introduced.

Example: Consider the database extension shown in Figure 2. The extension of the GMR $\langle\langle volume, weight\rangle\rangle$ with all results valid is depicted below.

$\langle\langle volume, weight\rangle\rangle$				
O_1 : *Cuboid*	*volume* : *float*	V_1 : *bool*	*weight* : *float*	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*

◇

In the above definition of GMRs nothing is stated about the relationship between the columns f_j and the correspondig validity flags V_j. Nevertheless, it is convenient to require that each stored result is either invalidated, i.e., the corresponding validity flag is set to *false*, or the result is correct with respect to the current state of the object base. This idea is reflected by the following definition of *consistent* GMR extensions.

Definition 3.2 (Consistent Extension)
An extension of the GMR $\langle\langle f_1, \ldots, f_m\rangle\rangle$ is a consistent extension iff

$$\forall \tau \in \langle\langle f_1, \ldots, f_m\rangle\rangle : \ \tau.V_j = true \ \Rightarrow \ \tau.f_j = f_j(\tau.O_1, \ldots, \tau.O_n)$$

□

In the remainder of this paper we consider only consistent GMR extensions. However, consistency is only a minimal requirement on GMR extensions. Further requirements like completeness and validity are introduced in the next subsection where the retrieval of materialized results is discussed.

The above definition of consistency provides for some tuning measure with respect to the invalidation and rematerialization of results. Upon an update to a database object that invalidates a materialized function result we have two choices:

1. *immediate rematerialization:* The invalidated function result is immediately recomputed as soon as the invalidation occurs.

2. *lazy rematerialization:* The invalidated function result is only marked as being invalid by setting the corresponding V_i attribute to *false*. The rematerialization of invalidated results is carried out as soon as the load of the object base management system falls below a predetermined threshold or—at the latest—at the next time the function result is needed.

3.2 Retrieval of Materialized Results

All GMR extensions are maintained by the *GMR manager*. The GMR manager offers retrieval operations to access argument objects and materialized results. Retrieval operations on GMRs can be represented in a tabular way—similarly to the relational query language *Query By Example (QBE)*.[3] The table below represents two abstract retrieval operations on a GMR $\langle\langle f_1, \ldots, f_m\rangle\rangle$ with complex argument types $t_1, \ldots, t_n$ and atomic result types:

[3] The tabular notation improves clarity of presentation in this subsection. However, in general, GOMql is used to denote queries.

$O_1 : t_1$	$O_2 : t_2$	$\ldots$	$O_n : t_n$	f_1	f_2	$\ldots$	f_m
id_{i_1}	id_{i_2}	$\ldots$	id_{i_n}	?	?	$\ldots$	?
?	?	$\ldots$	?	$[lb_1, ub_1]$	$[lb_2, ub_2]$	$\ldots$	$[lb_m, ub_m]$

The first retrieval operation is a *forward query*, where all argument objects are specified and the corresponding function values are obtained from the GMR. The second retrieval denotes a *backward range query*, where a range is specified for each function value and the corresponding argument objects are obtained. In general, the columns of a GMR query can contain any combination of constants (or ranges), '?'-signs and "don't-care"-signs (denoted by '–') facilitating a large variety of possible retrieval operations on GMRs.

The GMR retrieval operations can be used to support queries containing invocations of materialized functions. For example, if all results of the materialized function f_j are requested by some query, e.g., to perform some aggregate operation on the results, all results currently being valid can be obtained from the GMR—invalid results have to be (re-)computed. These (re-)computed results are also used by the GMR manager to update the GMR. Further, if the GMR is not complete, i.e., it does not contain an entry for each argument combination, the results of missing argument combinations have to be computed as well. Missing GMR entries whose results are computed during the evaluation of some query may be inserted into the GMR (for a discussion of incomplete GMR extensions see below).

Note that invalidated or missing results need not necessarily be (re-)computed upon the evaluation of some query. For example, if any *Cuboid* having a *weight* greater than 100.00 is to be retrieved (e.g., to serve as a counterweight), and if such a *Cuboid* can be found by inspecting the (incomplete) GMR no invalidated or missing results need be (re-)computed.

However, if a GMR extension is complete, i.e., it contains one entry for each argument combination, and the results of all functions occuring in the respective query are valid, the query can be evaluated on the GMR without having to (re-)compute any result. Subsequently, we formalize the notions of valid and complete GMR extensions.

Definition 3.3 (Valid Extension)
A consistent extension of the GMR $\langle\langle f_1, \ldots, f_m \rangle\rangle$ is called f_j-valid iff

$$\pi_{V_j} \; \langle\langle f_1, \ldots, f_m \rangle\rangle = \{true\}$$ □

Definition 3.4 (Complete extension)
A consistent extension of the GMR $\langle\langle f_1, \ldots, f_m \rangle\rangle$ is a complete extension iff

$$\pi_{O_1,\ldots,O_n} \; \langle\langle f_1, \ldots, f_m \rangle\rangle \; = \; ext(t_1) \times \ldots \times ext(t_n)$$

where $ext(t_i)$ denotes the extension of type t_i, i.e., the set of all instances of type t_i. □

Upon the creation of a new GMR the database programmer can choose whether the GMR extension has to be complete or whether the extension may be set up incrementally (starting with an empty GMR extension). Incrementally set up GMR extensions can be used as a cache for function results that were computed during the evaluation of queries. If the number of entries is limited (due to space restrictions) specialized replacement strategies for the GMR entries can be applied. Note that GMRs must be set up incrementally if they contain at least one partial function.

Invocations of materialized functions can not only occur in declarative queries but also in some function or operation definition. In this case every invocation of a materialized function is mapped to a forward query that will be evaluated by the GMR manager. For example, an invocation $f_i(o_1, \ldots, o_n)$ with $i \in \{1, \ldots, m\}$ would be transformed to the following algebraic expression if the GMR $\langle\langle f_1, \ldots, f_m \rangle\rangle$ is present:

$$\pi_{f_i}\ \sigma_{O_1 = o_1 \wedge \ldots O_n = o_n}\ \langle\langle f_1, \ldots, f_m \rangle\rangle$$

Of course, it has to be determined that the particular entry is valid.

3.3 Storage Representation of GMRs

The flexible retrieval operations on the GMRs require appropriate index structures to avoid the exhaustive search of GMR extensions. For that, well-known indexing techniques from relational database technology can be utilized. The easiest way to support the flexible and efficient access to any combination of GMR fields would be a single multi-dimensional index structure, denoted *MDS*, over the fields O_1, ..., O_n, f_1, ..., f_m:

MDS	O_1	...	O_n	f_1	...	f_m	V_1	...	V_m

Here, the first $n + m$ columns constitute the $(n + m)$-dimensional keys of the multi-dimensional storage structure. The m validity bits V_1, ..., V_m are additional attributes of the records being stored in the MDS.

Unfortunately, the (currently existing) multi-dimensional storage structures, such as the Grid-File [16], are not well-suited to support more than three or four dimensions. Therefore, our GMR manager has to utilize more conventional indexing schemes to expedite access on GMRs of higher arity. The index structures are chosen according to the expected query mix, the number of argument fields in the GMR, and the number of functions in the GMR. A good proposal for multi-dimensional indexing is given in an early paper by V. Lum [15].

In the remainder of this paper we discuss techniques to keep GMR extensions consistent. These techniques are independent of both the storage structure of a GMR and its completeness and validity.

4 Dynamic Aspects of Function Materialization

In this section we will investigate the algorithms that are needed to keep GMRs in a consistent state (according to Definition 3.2) while the object base is being modified.

We describe how materialized results are being invalidated if an update is reported to the GMR manager, and present the *lazy* and *immediate* rematerialization algorithms. Further, we discuss the adaptation of GMR extensions in case of the creation or deletion of argument objects of materialized functions. The last subsection of this section presents the update notification mechanism that is responsible for reporting updates to the GMR manager.

4.1 Invalidation and Rematerialization of Function Results

When the modification of an object o is reported to the GMR manager the GMR manager must find all materialized results that become invalid. This task is equivalent to determining all materialized functions f and all argument combinations $o_1, \ldots, o_n$ such that the modified object o has been accessed during the materialization of $f(o_1, \ldots, o_n)$. Note that in GOM—as in most other object bases—references are maintained only uni-directional; there is no efficient way to determine from an object o the set of objects that reference o via a particular path. Therefore, the GMR manager maintains reverse references from all objects that have been used in some materialization to the appropriate argument objects in a relation called *Reverse Reference Relation* (*RRR*). The RRR contains tuples of the following form:

$$[o,\, f,\, \langle o_1, \ldots, o_n \rangle]$$

Herein, o is a reference to an object utilized during the materialization of the result $f(o_1, \ldots, o_n)$. Note that o needs not be one of the arguments $o_1, \ldots, o_n$; it could be some object related (via attributes) to one of the arguments. Thus, each tuple of the RRR constitutes a reference from an object o influencing a materialized result to the tuple of the appropriate GMR in which the result is stored. We call this a *reverse reference* as there exists a reference chain in the opposite direction in the object base.[4]

Definition 4.1 (Reverse Reference Relation)
The Reverse Reference Relation RRR is a set of tuples of the form

$$[O : OID,\, F : \mathit{FunctionId},\, A : \langle OID \rangle]$$

For each tuple $r \in RRR$ the following condition holds: The object (with the identifier) $r.O$ has been accessed during the materialization of the function $r.F$ with the argument list $r.A$. □

The reverse references are inserted into the RRR during the materialization process. Therefore, each materialized function f and all functions invoked by f are modified—the modified versions are extended by statements that inform the GMR manager about the set of accessed objects. During a (re-)materialization of some result the modified versions of these functions are invoked.

Example: Let the GMRs $\langle\langle \mathit{volume}, \mathit{weight} \rangle\rangle$ and $\langle\langle \mathit{distance} \rangle\rangle$ be defined (again, this example is based on the extension shown in Figure 2). The extensions of the RRR and the two GMRs are shown in Figure 3. Note that two *Robots* with the identifiers id_4 and id_5 are assumed to exist in the object base. ◇

Based on the RRR we can now outline the algorithms for invalidating or rematerializing a stored function result, i.e., the computations that have to be performed by the GMR manager when an object o has been updated. The GMR manager is notified about an update by the following statement:

GMR_Manager.invalidate(o);

[4]This holds only if no global variables are used by the materialized function. Otherwise, the RRR contains reverse references not only to the argument objects but also to the accessed global variables.

RRR

O	F	A
id_1	*volume*	$\langle id_1 \rangle$
id_1	*weight*	$\langle id_1 \rangle$
id_1	*distance*	$\langle id_1, id_4 \rangle$
id_1	*distance*	$\langle id_1, id_5 \rangle$
$\vdots$	$\vdots$	$\vdots$
id_5	*distance*	$\langle id_1, id_5 \rangle$
id_5	*distance*	$\langle id_2, id_5 \rangle$
id_5	*distance*	$\langle id_3, id_5 \rangle$
$\vdots$	$\vdots$	$\vdots$
id_{35}	*volume*	$\langle id_3 \rangle$
id_{35}	*weight*	$\langle id_3 \rangle$
id_{38}	*distance*	$\langle id_3, id_4 \rangle$
id_{38}	*distance*	$\langle id_3, id_5 \rangle$
id_{77}	*weight*	$\langle id_1 \rangle$
id_{77}	*weight*	$\langle id_2 \rangle$
id_{99}	*weight*	$\langle id_3 \rangle$

$\langle\langle volume, weight \rangle\rangle$

O_1	*volume*	V_1	*weight*	V_2
id_1	300.0	*true*	2358.0	*true*
id_2	200.0	*true*	1572.0	*true*
id_3	100.0	*true*	1900.0	*true*

Storage Structure:

MDS

O_1	*volume*	*weight*	V_1	V_2

$\langle\langle distance \rangle\rangle$

O_1	O_2	*distance*	V_1
id_1	id_4	10.2	*true*
id_1	id_5	213.0	*true*
id_2	id_4	85.2	*true*
id_2	id_5	5.0	*true*
id_3	id_4	0.9	*true*
id_3	id_5	220.0	*true*

Storage Structure:

MDS

O_1	O_2	*distance*	V_1

Figure 3: The Data Structures of the GMR Manager

The algorithms below reflect the two different possibilities of *lazy rematerialization* and *immediate rematerialization*.

The *lazy rematerialization* algorithm only invalidates the affected GMR entries:

lazy(o) $\equiv$ **foreach** triple $[o, f_i, \langle o_1, \ldots, o_n \rangle]$ **in** *RRR* **do**
(1) **set** $V_i := false$ of the appropriate tuple **in** $\langle\langle f_1, \ldots, f_i, \ldots, f_m \rangle\rangle$
(2) **remove** $[o, f_i, \langle o_1, \ldots, o_n \rangle]$ **from** *RRR*

Step 2 of the algorithm—i.e., the removal of the RRR entry—ensures that for the same, repeatedly performed object update the invalidation is done only once. Subsequent invalidations due to updates of o will be blocked at the beginning of $lazy(o)$ by not finding the RRR entry which was removed upon the first invalidation—thus the unnecessary penalty of accessing the tuple in the GMR to re-invalidate an already invalidated result is avoided. By the next rematerialization of $f(o_1, \ldots, o_n)$ all relevant RRR entries are (re-)inserted into the RRR—analogously to the immediate rematerialization algorithm shown below.

Under the *immediate rematerialization* strategy we have to recompute the affected function results:

immediate(o) $\equiv$ **foreach** triple $[o, f_i, \langle o_1, \ldots, o_n \rangle]$ **in** *RRR* **do**
(1) **remove** $[o, f_i, \langle o_1, \ldots, o_n \rangle]$ **from** *RRR*
(2) **recompute** $f_i(o_1, \ldots, o_n)$ and
$*$ **remember** all accessed objects $\{o'_1, \ldots, o'_p\}$
$*$ **replace** the old value of $f_i(o_1, \ldots, o_n)$ **in** $\langle\langle f_1, \ldots, f_i, \ldots, f_m \rangle\rangle$

(3) **foreach** v **in** $\{o'_1, \ldots, o'_p\}$ **do**

  $*$ **insert** the triple $[v, f_i, \langle o_1, \ldots, o_n \rangle]$ **into** RRR (if not present)

We will explain step 1 of this algorithm last. In step 2 we recompute the function result $f_i(o_1, \ldots, o_n)$ and remember all objects visited in this process in order to insert them into the RRR in step 3. However, it cannot be guaranteed that the RRR does not contain any obsolete entries which constitute "leftovers" from the previous materialization(s) of $f_i(o_1, \ldots, o_n)$—this happens whenever two subsequent materializations of $f_i(o_1, \ldots, o_n)$ visit different sets of objects. Let $[w, f_i, \langle o_1, \ldots, o_n \rangle]$ be such a leftover entry meaning that in an earlier materialization of $f_i(o_1, \ldots, o_n)$ the object w was visited; but the current materialized result of $f_i(o_1, \ldots, o_n)$ is not dependent on the state of w. Then the next (seemingly relevant) update on w will remove the triple $[w, f_i, \langle o_1, \ldots, o_n \rangle]$ from the RRR by step 1 of the above outlined algorithm while steps 2 and 3 do not inject any new information that is not already present in the GMR and the RRR.

In most cases an object will be re-used after an update—thus, the same RRR entry that has been removed in step 1 of the above algorithm will be re-inserted into the RRR. This situation could be remedied by a *second chance algorithm*, that is based on marking RRR entries instead of removing them in step 1.

With respect to removing left-over entries our RRR maintenance algorithm can be termed *lazy* because left-over entries are removed only when the corresponding object is updated. An alternative to this strategy would be a periodic reorganization of the RRR.

The rematerialization of function results that constitute complex objects may lead to the creation of new objects. Invalidated result objects cannot be deleted by the GMR manager as they may be referenced in other contexts independently of the materialization of the function. Thus, to minimize the number of unreferenced but undeleted result objects GMRs with complex result types should be maintained under lazy materialization. A garbage collection mechanism can be employed to remove unreferenced objects.

4.2 The Creation and Deletion of Argument Objects

If a new object of an argument type of some materialized function—or a subtype thereof—is created, new GMR entries have to be inserted into the appropriate GMRs in order to keep the extensions of those GMRs consistent. Therefore, the following statement is invoked with o being the new object and t being the type of o:

GMR_Manager.new_object(o, t);

For each GMR affected by the creation of the new object function results are computed for all argument combinations containing the new object o; the appropriate tuples are inserted into the GMR.

The deletion of an object o that has been used as an argument during the materialization of any result initiates the invocation of the following statement:

GMR_Manager.forget_object(o);

Here, o is the identifier of the object that is to be deleted. The following algorithm is performed by an invocation of *GMR_Manager.forget_object*(o):

foreach tuple $[o, f, \langle o_1, \ldots, o_n \rangle]$ **in** RRR
(1) **if** $o \in \{o_1, \ldots, o_n\}$ **and** $\langle\!\langle f_1, \ldots, f_m \rangle\!\rangle$ exists with $f \in \{f_1, \ldots, f_m\}$ **then**
remove $[o_1, \ldots, o_n, F_1, V_1, \ldots, F_m, V_m]$ **from** $\langle\!\langle f_1, \ldots, f_m \rangle\!\rangle$
(2) **remove** $[o, f, \langle o_1, \ldots, o_n \rangle]$ **from** RRR

All RRR entries that contain a reference to the deleted object in their first attribute are visited. Step 1 of this algorithm tests, whether the deleted object is an argument object of the materialized result referenced by the RRR entry $[o, f, \langle o_1, \ldots, o_n \rangle]$. If the test is positive, i.e., $o \in \{o_1, \ldots, o_n\}$, the appropriate GMR entry is removed. In step 2 of the algorithm, the RRR entry is removed. Note that there may exist further entries in the RRR that contain a reference to the deleted object in their argument list. Those entries can only be found by exhaustively searching the RRR or by using a supplementary index on the RRR. However, we will utilize a lazy maintenance algorithm by leaving those so-called *blind references* in the RRR. When a RRR entry containing a blind reference is accessed the blind reference is detected by not finding the referenced argument combination in the appropriate GMR—the entry will then be removed analogously to the left-over entries discussed in the previous RRR maintenance algorithms.

4.3 The Update Notification Mechanism

The operations of the GMR manager to keep the GMR extensions up to date (*invalidate*, *new_object*, *forget_object*) could be invoked either by the object manager or by the updating operation.

The object manager can inform the GMR manager about relevant object modifications when the updated object is stored in the object base. This approach makes the adaptation of the object manager necessary which may be prohibitively difficult in existing systems. Further, the adaptation of the object manager has the following shortcomings:

- Every user of the object base will be penalized by the materialization of functions—even if only those parts of the object base are accessed that are not involved in any materialization.
- As applications may first modify some objects and then access materialized results being affected by the former updates, all updates must immediately be propagated to the object manager in order to keep the GMR extensions consistent. This need to store updated objects immediately prevents optimization strategies based on deferring the storage of modified objects.

Therefore, in GOM we chose the schema rewrite approach which is based on analyzing the materialized functions and modifying the relevant parts of the object base schema, i.e., those update operations that affect materialized results. In GOM, the state of the object base can only be modified by the elementary update operations *t.create* and *t.delete* for any type t, *t.set_A* for any tuple-structured type t and any attribute A of t, and *t.insert* and *t.remove* for any set-structured type t. Every elementary update operation associated with some type t involved in the materialization of any function is modified and recompiled, such that each time the update operation is invoked, the invocation of one of the functions *invalidate*, *new_object*, and *forget_object* will be triggered.

The approach of injecting the notification mechanism into the primitive updating operations has the advantage that no adaptation of the object manager is needed. Instead it requires a modification of the updating functions and their recompilation. Further, the schema rewrite approach guarantees that the GMR manager is immediately informed when an update occurs—by that, the extensions of the GMRs will remain consistent.

Figure 4 shows the modified versions of the update operation *t.set_A* for a tuple-structured type t with attribute A and of the delete operation *t.delete*. The operation *t.set_A* is extended by the statement

GMR_Manager.invalidate(**self**);

such that the invalidation occurs *after* the value of attribute A has been updated. If the materialized results are invalidated before the update the immediate rematerialization strategy would lead to wrong results. The *delete* operation is extended by the statement

GMR_Manager.forget_object(**self**);

that is invoked *before* the object is deleted.

In the next section we show how the set of update operations that have to be modified can be drastically reduced.

```
declare set_A: t || t' → void
   code set_A';
define set_A' (x) is
   begin
      self.A := x;
      GMR_Manager.invalidate(self);
   end define set_A';
```

```
declare delete: t || → void
   code delete';
define delete' is
   begin
      GMR_Manager.forget_object(self);
      self.system_delete;
   end define delete';
```

Figure 4: Modification of Primitive Update Operations (Version 1)

5 Strategies to Reduce the Invalidation Overhead

The invalidation mechanism described so far is (still) rather unsophisticated and, therefore, induces unnecessarily high update penalties upon object modifications. In the following we will describe four dual techniques to reduce the update penalty—consisting of invalidation and rematerialization—by better exploiting the potential of the object-oriented paradigm. The techniques described in this section are based on the following ideas:

1. *isolation of relevant object properties:* Materialized results typically depend on only a small fraction of the state of the objects visited in the course of materialization. For example, the materialized *volume* certainly does not depend on the *Value* and *Mat* attributes of a *Cuboid*.

2. *reduction of RRR lookups:* The unsophisticated version of the invalidation process has to check the *RRR* each time any object o is being updated. This leads to many unnecessary table lookups which can be avoided by maintaining more information within the objects being involved in some materialization—and thus restricting the lookup penalty to only these objects.

3. *exploitation of strict encapsulation:* By strictly encapsulating the representation of objects used by a materialized function the number of update operations that need be modified can be reduced significantly. Since internal subobjects of a strictly encapsulated object cannot be updated separately—without invoking an outer-level operation of the strictly encapsulated object—we can drastically reduce the number of invalidations by triggering the invalidation only by the outer-level operation.

4. *compensating updates:* Instead of invoking the materialized function to recompute an invalidated result, specialized compensating actions can be invoked that use the old result and the parameters of the update operation to recompute the result in a more efficient way.

5.1 Isolation of Relevant Object Properties

Suppose that *volume* and *weight* have been materialized. Then these two materialized functions surely don't depend on the attribute *Value*. Nevertheless, under the unsophisticated invalidation strategy the operation invocation

id_1.set_Value(123.50);

does lead to the invalidation of $id_1.volume$ and $id_1.weight$, both of which are unnecessary. Likewise, the operation invocation

id_1.set_Mat(Copper); !! *Copper* being a variable of type *Material*

leads to the necessary invalidation of $id_1.weight$, but also to the unnecessary invalidation of $id_1.volume$. In order to avoid such unnecessary invalidations the system has to separate the relevant properties of the objects visited during a particular materialization from the irrelevant ones. Then invalidations should only be initiated if a relevant property of an object is modified.

Definition 5.1 (Relevant Attributes)
Let $f : t_1, \ldots, t_n \rightarrow t_{n+1}$ be a materialized function. Then the set $RelAttr(f)$ is defined as:

$$RelAttr(f) = \{t.A \mid \text{there exist } o_1, \ldots, o_n \text{ of type } t_1, \ldots, t_n \text{ and } o \text{ of tuple type } t \text{ such that } o.A \text{ is accessed to materialize } f(o_1, \ldots, o_n)\} \quad \square$$

The relevant properties of a materialized function f are automatically extracted from the implementation of the function f—of course, also inspecting all functions invoked by f. The mechanism for extracting the set *RelAttr* from the implementation of a function is outlined in the Appendix of this paper.

A materialized function result $f(o_1, \ldots, o_n)$ can only be invalidated by an invocation $o.set\_A(\ldots)$ on some object o of type t and $t.A \in RelAttr(f)$. The following is the key definition for avoiding unnecessary GMR invalidations:

Definition 5.2 (Schema Dependent Functions)
Let t be a tuple type and let A be any attribute of t. We define the set of (materialized) functions which depend on the update operation $t.set\_A$ as

$$SchemaDepFct(t.set\_A) = \{f \mid f \text{ is a materialized function and } t.A \in RelAttr(f)\} \quad \square$$

Now, the invalidation overhead can be reduced by (1) modifying only those update operations *t.set_A* that are relevant to some materialized function, i.e., only those operations *t.set_A* where $SchemaDepFct(t.set\_A) \neq \{\}$, and (2) informing the GMR manager not only about the updated object, but also about the set of materialized functions potentially affected by the update. Thus, the modification *o.set_A*(...) of an object *o* of type *t* triggers the invocation of the GMR manager as follows:

GMR_Manager.invalidate(o, SchemaDepFct(t.set_A));

The set *SchemaDepFct*(*t.set_A*) is inserted as a set-valued constant into the body of the modified update operation *o.set_A*—thus, the expression *SchemaDepFct*(*t.set_A*) need not be evaluated each time *o.set_A*(...) is invoked.

Example: The relevant properties for the function *volume* are given below:

$$RelAttr(volume)=\{Cuboid.V1, Cuboid.V2, Cuboid.V4, Cuboid.V5, Vertex.X, Vertex.Y, Vertex.Z\}$$

From this it follows that the stored results of the function *volume* can only be invalidated by the update operations *set_V1*, *set_V2*, *set_V4* and *set_V5* associated with type *Cuboid*, and by the *set_X*, *set_Y* and *set_Z* operations of type *Vertex*. ◇

5.2 Marking "Used" Objects to Reduce RRR Lookup

The improvement of the invalidation process developed in the preceding subsection ensures that no more unnecessary invalidations occur.[5] However, one problem still remains: the GMR manager is invoked more often than necessary to check within the RRR whether an invalidation has to take place. Suppose object *o* of type *t* is updated by operation *o.set_A*(...) and all functions which have used *o* for materialization are *not* contained in *SchemaDepFct*(*t.set_A*). In this case there cannot be a materialized value that must be invalidated due to the update *o.set_A*. Consider, for example, the update

id_{111}.set_X(2.5); !! the Vertex id_{111} **not** being a boundary *Vertex* of any *Cuboid*

of the *Vertex* instance id_{111} that is not referenced by any *Cuboid*. Since the functions *volume* and *weight* are contained in the set *SchemaDepFct*(*Vertex.set_X*) the GMR manager is being invoked—only to find out by a RRR-lookup that no invalidation has to be performed. This imposes a (terrible) penalty upon geometric transformations of "innocent" objects, e.g., *Cylinders* and *Pyramids*, if the *volume* of *Cuboid* has been materialized—due to the fact that all three types are clients of the same type *Vertex*.

Our goal is to invoke *GMR_Manager.invalidate* only when an invalidation has to take place. Therefore, we append to each object *o* the set-valued attribute *ObjDepFct* that contains the identifiers of all materialized functions that have used *o* during their materialization. Now, the set of functions whose results are invalidated by the update *o.set_A* can be determined exactly by

$$o.ObjDepFct \cap SchemaDepFct(t.set\_A)$$

[5] Actually, under the unlikely condition that the same object type is utilized in the same materialization in different contexts there may still be an unnecessary invalidation.

The set-valued attributes *ObjDepFct* are maintained in the same way as the entries of the RRR: if an entry $[o, f_i, \langle o_1, \ldots, o_n \rangle]$ is inserted into (removed from) the RRR, f_i is inserted into (removed from) $o.ObjDepFct$.

Note that conceptually it would be possible to migrate all RRR information into the individual objects—avoiding the RRR and all RRR lookups altogether. But since a single object is usually involved in numerous materializations of different functions and different argument combinations, this requires too much storage space within the objects and, thus, destroys any kind of object clustering.

Figure 5 shows the modified versions of the update operations that exploit the attribute *ObjDepFct*.

```
declare set_A: t || t' → void
   code set_A';

define set_A' (x) is
   begin
      self.A := x;
      RelevFct := self.ObjDepFct ∩
         SchemaDepFct(t.set_A);
      if RelevFct ≠ {} then
         GMR_Manager.invalidate(self,RelevFct);
   end define set_A';
```

```
declare delete: t || → void
      code delete';

define delete' is
   begin
      if self.ObjDepFct ≠ {} then
         GMR_Manager.forget_object(self);
      self.system_delete;
   end define delete';
```

Figure 5: Modification of Primitive Update Operations (Version 2)

Example: Recall the database extension shown in Figure 2. Suppose that the following GMRs were introduced: $\langle\langle total\_volume, total\_weight \rangle\rangle$ for the type *Workpieces*, $\langle\langle total\_value \rangle\rangle$ for the type *Valuables*, and $\langle\langle volume, weight \rangle\rangle$ for the type *Cuboid*.

Consider the invocation $id_{31}.set\_X(\ldots)$ which modifies the X coordinate of Vertex id_{31}. Figure 6 shows the modification of the update operation *Vertex.set_X*. The set of materialized functions that is dependent upon the update $id_{31}.set\_X(\ldots)$ is then given by the intersection of the sets $SchemaDepFct(Vertex.set\_X)$ and $id_{31}.ObjDepFct$.

$$\begin{aligned} SchemaDepFct(Vertex.set\_X) &= \{volume, weight, total\_volume, total\_weight\} \\ id_{31}.ObjDepFct &= \{volume, weight\} \end{aligned}$$

In this case, the intersection coincides with the set $id_{31}.ObjDepFct$. However, in general this is not the case, e.g., for the operation $Cuboid.set\_V1$ and the update $id_3.set\_V1$. $\diamond$

5.3 Information Hiding

Despite the improvements of the invalidation mechanism outlined in the previous two subsections three problems that can be avoided by exploiting information hiding remain.

First, the improvements incorporated so far do not totally prevent the penalization of operations on objects not involved in any materialization. For example, update operations defined on other geometric objects, e.g., *Pyramids*, are penalized by the materialization of *Cuboid.volume*, if the type *Vertex* is utilized in the definition of both types. This is a

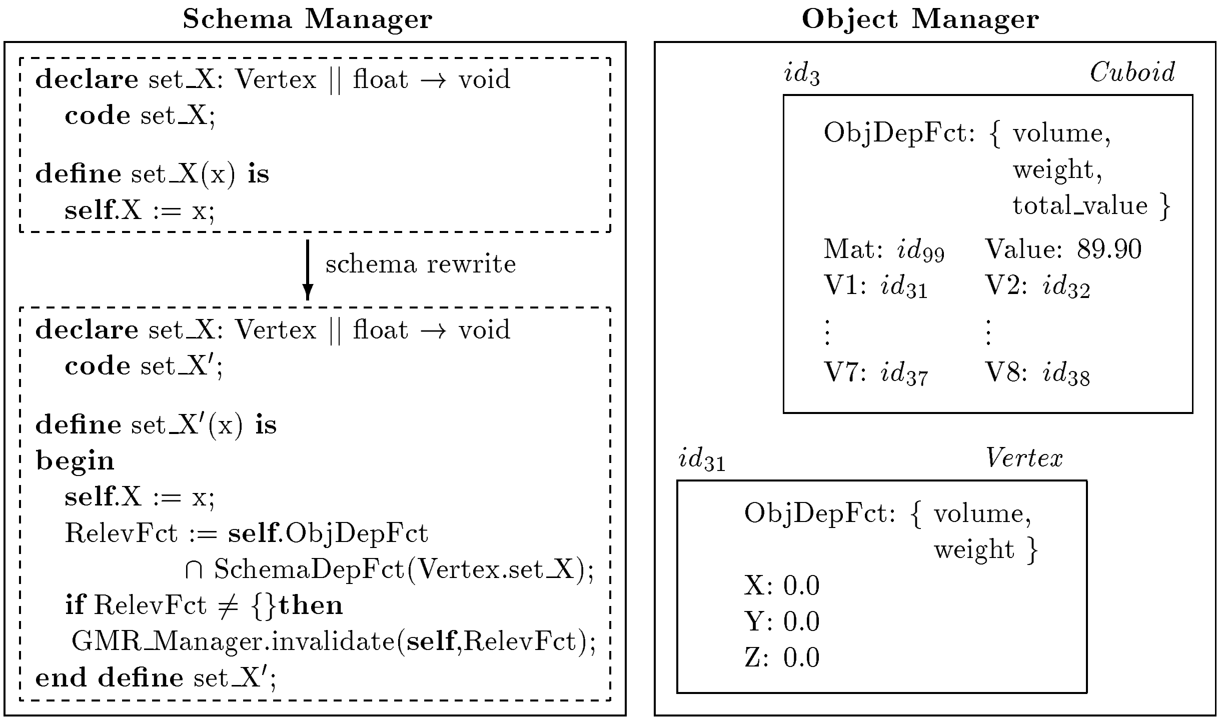


Figure 6: Interaction between Schema and Object Manager

consequence of modifying the update operations of the lower-level types, e.g., *Vertex.set_X* which is then invoked on *every* update of attribute X of type *Vertex*.

Second, a single update operation consisting of a sequence of lower-level operations may trigger many subsequent rematerializations of the same precomputed result. For example, one single invocation of $id_1.scale(\ldots)$ triggers 12 (!) invalidations of $id_1.volume$ initiated by the *set_X*, *set_Y* and *set_Z* operations of type *Vertex*. Obviously, one invalidation should be enough.

Third, our algorithms detailed so far cannot detect the irrelevance of an update operation sequentially invoking lower-level operations which neutralize each other with respect to a precomputed result. For example, the invocation of $id_1.rotate$ performs 12 invalidations of $id_1.volume$ despite the fact that *no* invalidation is required since the volume stays invariant under rotation.

We can exploit information hiding to avoid the unnecessary overhead incurred by the three above mentioned problems. Analogous to information hiding in traditional software design we call an object *strictly encapsulated* if the direct access to the representation of this object—including all its subobjects—is prohibited; manipulations may only be possible by invoking public operations defined on the type of that object. These operations constitute the *object interface*. In GOM strict encapsulation is realized (1) by disclosing all access operations for attributes from the **public** clause, (2) by creating all subobjects of an encapsulated complex object during the initialization of that object, and (3) by enforcing that no public operation returns references to subobjects. Thus, no undesired access to subobjects via, e.g., object sharing is possible.

By enforcing strict encapsulation only updating interface operations have to be modified to perform invalidations. Further, the number of invalidations due to the invocation of an update operation is reduced to one. Last not least, update operations leaving the result of a materialized function invariant need not be modified. Thus by specifying and exploiting a set of *Invalidated Functions* for each invalidating public operation the above mentioned problems can easily be eliminated.

Definition 5.3 (Invalidated Functions)
Let t be a strictly encapsulated type and u be a public operation associated with that type. We define the set of invalidated (materialized) functions of $t.u$ as

$$\mathit{InvalidatedFct}(t.u) \ := \ \{f \mid f \text{ is a materialized function and } t.u \text{ affects results of } f\} \qquad \Box$$

We assume that the set $\mathit{InvalidatedFct}(t.u)$ for each operation $t.u$ is determined by the database programmer. Then all update operations u for which $\mathit{InvalidatedFct}(t.u) \neq \{\}$, are extended by statements to inform the GMR manager—analogously to the modification of elementary update operations.

As outlined in Section 4 the materialized function f and all functions invoked by f are modified to mark all used objects. If for the materialization of a function f a strictly encapsulated object is used, only this object, but none of its subobjects, have to be marked. Public functions of strictly encapsulated types are regarded to be *atomic*—thus, functions invoked by public functions may remain unchanged.

Example: Consider the type definition of *Cuboid* as presented in Figure 1. Now assume that the **public** clause reads as follows:

```
persistent type Cuboid supertype ANY is
   public rotate, scale, translate, volume, weight
   ...
end type Cuboid;
```

From this type definition it can be deduced—by a close observation of the operational semantics—that the only operation that affects a materialized *volume* is the operation *scale*. All other operations do not invalidate the precomputed *volume*. With respect to the materialization of *volume*, *scale* has to be modified as follows:

```
declare scale: Cuboid || Vertex → void code scale′;

define scale′ (v) is
begin
   ...          !! Statements to scale the cuboid !!
   RelevFct := self.ObjDepFct ∩ InvalidatedFct(Cuboid.scale);
   if RelevFct ≠ {} then GMR_Manager.invalidate(self, RelevFct);
end define scale′;
```

$\diamond$

5.4 Compensating Actions

A materialized result that has been invalidated by an update can be recomputed either by an invocation of the materialized function or by a specialized function compensating the update. For example, consider the GMR $\langle\!\langle$*total_volume*$\rangle\!\rangle$.[6] When a new *Cuboid* instance is inserted into a set of type *Workpieces* the result of *total_volume* can be recomputed by

[6]The function *total_volume* is associated with the type *Workpieces*.

adding the *volume* of the inserted *Cuboid* to the old result of *total_volume*—instead of having to recompute *volume* for all members of the set. For this the database programmer has to specify a *compensating action*, i.e., a function that compensates for the insertion of a new *Cuboid* into a set of type *Workpieces*:

declare increase_total: Workpieces || Cuboid, float → float;

define increase_total (new_cuboid, old_total) **is**
  return old_total + new_cuboid.volume;

Compensating actions may only be specified for update operations associated with argument types of materialized functions. It is not allowed to specify a compensating action for an update operation associated with a non-argument type, as this may lead to inconsistent GMR extensions. If, for example, a compensating action is specified for the materialized function *total_volume* and the update operation *Cuboid.scale* the GMR ⟨⟨*total_volume*⟩⟩ could become inconsistent by an invocation of *Cuboid.scale*.

Based on Figure 2, assume *Cuboid* id_1 to be a member of the set id_{59} of type *Workpieces*. The invocation $id_{59}.remove(id_1)$ removes id_1 from the set id_{59}—due to our RRR maintenance algorithm, id_1 remains marked. Thus, if *Cuboid* id_1 is scaled subsequently, the invocation of the specified compensating action is triggered—leading to a wrong result, as id_1 is no longer a member of id_{59}.

Further, compensating actions can only be specified for *modified* update operations, i.e., update operations that are extended by statements to inform the GMR manager about updates. If an argument type t is strictly encapsulated only public update operations of t are modified. Otherwise, if t is not strictly encapsulated, the elementary update operations $t.set\_A$ (if t is tuple-structured) or $t.insert$ and $t.remove$ (if t is set-structured) are modified.

The following definition formalizes the concept of compensating actions.

Definition 5.4 (Compensating Action)
Let $f : t_1, \ldots, t_n \rightarrow t_{n+1}$ *be a materialized function, and let* $u : t_i || t'_1, \ldots, t'_k \rightarrow \textbf{void}$ *be an update operation with* $1 \leq i \leq n$ *and* $f \in SchemaDepFct(t_i.u)$ *or* $f \in InvalidatedFct(t_i.u)$. *A function*

$$c : t_i || t'_1, \ldots, t'_k, t_{n+1} \rightarrow t_{n+1}$$

is called a compensating action *for function* f *and update operation* $t_i.u$, *if the following two sequences of statements are equivalent for any objects* o_i *of type* t_i $(1 \leq i \leq n)$, o'_j *of type* t'_j $(1 \leq j \leq k)$ *and any variable* x *of type* t_{n+1}*:*

$$\left[\begin{array}{l} o_i.u(o'_1, \ldots, o'_k); \\ x := f(o_1, \ldots, o_n); \end{array}\right] \equiv \left[\begin{array}{l} x := o_i.c(o'_1, \ldots, o'_k, f(o_1, \ldots, o_n)); \\ o_i.u(o'_1, \ldots, o'_k); \end{array}\right] \qquad \square$$

Compensating actions have to be supplied by the database programmer. As the equivalence of the two sequences of statements in the above definition is not decidable it is the database programmer's responsibility to guarantee the correctness of the compensating action.

The GMR manager maintains compensating actions in a table named *CA*. *CA* contains one triple $[u, f, c]$ for every compensating action c (and no other entries) with c being associated with the update operation u and the materialized function f. If the compensating action *increase_total* is specified for *Workpieces.insert* and *total_volume* *CA* would contain the following entry:

		Upd_Op	Mat_Fct	Comp_Act
CA	$=$	$Workpieces.insert$	$total\_volume$	$increase\_total$
		$\vdots$	$\vdots$	$\vdots$

Based on CA, we define the set $CompensatedFct(u)$ for each update operation u:

Definition 5.5 (Compensated Functions)
Let $t.u$ be an update operation associated with type t. We define the set of compensated (materialized) functions of $t.u$ as

$$CompensatedFct(t.u) := \pi_{Mat\_Fct}\ \sigma_{Upd\_Op=t.u}\ CA \qquad \square$$

If for an update operation $t.u$ and a materialized function f a compensating action is specified, i.e., $f \in CompensatedFct(t.u)$, $GMR\_manager.compensate$ is invoked instead of $GMR\_manager.invalidate$. Then, an invocation

$o.u(o'_1, \ldots, o'_k)$;

of the update operation $t.u$ with arguments $o, o'_1, \ldots, o'_k$ triggers the invocation of the operation $GMR\_manager.compensate$ as follows:

GMR_Manager.compensate($\langle o, o'_1, \ldots, o'_k \rangle$, $t.u$, $RelevFct$);

with $RelevFct = o.ObjDepFct \cap CompensatedFct(t.u)$. For each function $f \in RelevFct$ the compensating action c for u and f is retrieved from CA. The new values of the results of f invalidated by the invocation of u are computed by invocations of the form

$o.c(o'_1, \ldots, o'_k, old)$;

where *old* is the old GMR value. The update operation u is modified to initiate the triggering of *GMR_Manager.compensate*. Contrary to *GMR_Manager.invalidate*, the operation *GMR_Manager.compensate* is invoked *before* the update is executed. Thus, all compensating actions can access the state of the object base before being modified. The following example illustrates the modification of update operations in case of compensating actions.

Example: Assume that the compensating action *increase_total* (as defined above) is specified for the function *total_volume* and the update operation *Workpieces.insert*, i.e., $total\_volume \in CompensatedFct(Workpieces.insert)$. Then, *Workpieces.insert* is modified as follows:

```
declare insert: Workpieces || Cuboid → void
   code insert';

define insert' (cub) is
begin
   RelevFct := self.ObjDepFct ∩ CompensatedFct(Workpieces.insert);
   if RelevFct ≠ {} then
      GMR_Manager.compensate(⟨self, cub⟩, insert, RelevFct);
   self.system_insert(cub);
   RelevFct := self.ObjDepFct ∩ SchemaDepFct(Workpieces.insert) \ RelevFct;
   if RelevFct ≠ {} then
      GMR_Manager.invalidate(self, RelevFct);
end define insert';
```
◇

6 Restricted GMRs

Suppose we are interested only in *Cuboids* made of iron and do not consider *Cuboids* made of gold. In this case, we need to materialize *volume* and *weight* only for *Cuboid* instances c that satisfy the predicate $p \equiv c.Mat.Name =$ *"Iron"*. In GOMql, the *p-restricted* materialization of *volume* and *weight* is initiated by the following statement:

range c: Cuboid
materialize c.volume, c.weight
where c.Mat.Name = "Iron"

The following definition introduces restricted GMRs:

Definition 6.1 (Restricted GMRs)
Let $p : t_1, \ldots, t_n \rightarrow bool$ be a predicate over the argument types $t_1, \ldots, t_n$ of the GMR $\langle\langle f_1, \ldots, f_m \rangle\rangle$. We denote the p-restricted GMR for functions $f_1, \ldots, f_m$ as $\langle\langle f_1, \ldots, f_m \rangle\rangle_p$.
An extension of $\langle\langle f_1, \ldots, f_m \rangle\rangle_p$ is complete iff

$$\pi_{O_1,\ldots,O_n} \langle\langle f_1, \ldots, f_m \rangle\rangle_p = \{[o_1, \ldots, o_n] \mid [o_1, \ldots, o_n] \in ext(t_1) \times \ldots \times ext(t_n) \ \wedge \ p(o_1, \ldots, o_n)\} \qquad \square$$

Analogously to Definition 3.4 the extension of a restricted GMR is complete if it contains one entry for each argument combination that satisfies the restriction predicate p (and no other entry).

Restricted GMRs can be used to evaluate forward queries in the same way as unrestricted GMRs—when a function result is needed that is not materialized as its arguments do not fulfill the restriction predicate, the result will be computed using the "normal" function.

Further, each GMR provides an access path to the results of the materialized functions that can be used to evaluate backward queries. Thus, a *restricted GMR* can be taken as a *partial index* on the function results as examined in [18]. Consider the following prototypical backward query:

range $o_1 : t_1, \ldots, o_n : t_n$
retrieve $o_1, \ldots, o_n$
where Φ

First, the selection predicate Φ is transformed into disjunctive normal form. As we have a *backward* query, Φ must contain an invocation $f(o_{i_1}, \ldots, o_{i_k})$ ($\{i_1, \ldots, i_k\} \subseteq \{1, \ldots, n\}$) of some materialized function f. Those conjuncts of Φ containing variables of the set $\{o_{i_1}, \ldots, o_{i_k}\}$ are called the *relevant* parts of the selection predicate Φ and are denoted by Φ'.

Contrary to forward queries, a p-restricted GMR can be used to evaluate a backward query only if the restriction predicate p covers the relevant parts of the selection predicate, i.e., $\Phi' \Rightarrow p$ is valid for *any* instance of the database [18]. This is, in general, an undecidable problem. However, for a simple but in practice useful subclass of predicates that is presented in [17] the equivalent problem of deciding the satisfyability of $\neg p \wedge \Phi'$ is decidable.

In [17] three types of comparisons are considered:

Comparison with a constant:	$x\ \theta\ c$	(Type 1),
Comparison between variables:	$x\ \theta\ y$	(Type 2),
Comparison with an offset:	$x\ \theta\ y+c$	(Type 3).

Here, θ is one of $\{=, \neq, <, >, \leq, \geq\}$, x and y are variables and c is a constant. A predicate p belongs to the considered subclass if (1) p is a Boolean combination of comparisons of the above types and (2) the disjunctive normal form of p after eliminating all negations does not contain the comparison operator $\neq$ in any comparison of Type 2 or 3. For this subclass of predicates, a polynominal algorithm to decide satisfyability is given in [17]. Rosenkrantz and Hunt [17] show further, that the problem of deciding the satisfyability of predicates becomes NP-hard if the comparison operator $\neq$ is included.

Let Φ' be the relevant part of the selection predicate of a backward query Q_{bw}. The p-restricted GMR $\langle\langle f_1, \ldots, f_m \rangle\rangle_p$ is applicable to evaluate Q_{bw}, if the following conditions hold:

1. $\neg p$ belongs to the above defined class of predicates (p does not contain comparisons of the form $x = y$ or $x = y + c$).

2. Φ' belongs to the above defined class of predicates (Φ' does not contain comparisons of the form $x \neq y$ or $x \neq y + c$).

3. $\neg p \wedge \Phi'$ is *not* satisfiable.

The validity of the above conditions can be decided in $O(k^3)$ time [17] where k is the number of variables in $\neg p \wedge \Phi'$.

Example: Assume the function *distance* to be declared as follows:

declare distance: Cuboid, Cuboid $\rightarrow$ float;

As $distance(c_1, c_1) = 0.0$ and $distance(c_1, c_2) = distance(c_2, c_1)$ for any *Cuboid* instances c_1 and c_2, it is convenient to restrict the materialization of *distance* by the following predicate:

$$p(c_1, c_2) \equiv (c_1 \neq c_2) \wedge (c_1.V1.X \leq c_2.V1.X)$$

The backward query

range c: Cuboid
retrieve c
where (distance(c, id_{99}) $<$ 100.0 $\wedge$ c $\neq$ id_{99} $\wedge$ c.V1.X $\leq$ id_{99}.V1.X) $\vee$
(distance(id_{99}, c) $<$ 100.0 $\wedge$ c $\neq$ id_{99} $\wedge$ id_{99}.V1.X $\leq$ c.V1.X)

can be evaluated using the GMR $\langle\langle distance \rangle\rangle_p$. $\diamond$

In the subsequent subsections we describe techniques to keep restricted GMRs consistent and we point out the impact of materialized functions with atomic argument types on the restriction predicate.

6.1 Keeping Restricted GMRs Consistent

In Section 4 we have outlined the algorithms to keep (unrestricted) GMRs consistent while the object base is being modified. These techniques apply also to restricted GMRs. But restricted GMRs must further be adapted if an update of the object base affects the restriction predicate.

Consider, for example, the GMR $\langle\!\langle volume,weight \rangle\!\rangle_p$ with p defined as $O_1.Mat.Name =$ *"Iron"* (the attribute O_1 of the GMR $\langle\!\langle volume,weight \rangle\!\rangle_p$ is of type *Cuboid*). In the database extension depicted in Figure 2 the *Cuboid* instance id_3 is made of gold and, therefore, its *volume* and *weight* are not contained in $\langle\!\langle volume,weight \rangle\!\rangle_p$. If the material of id_3 is changed from gold to iron the GMR $\langle\!\langle volume,weight \rangle\!\rangle_p$ has to be adapted.

Let $f_1, \ldots, f_m$ be functions with the argument types $t_1, \ldots, t_n$. A restriction predicate p of the GMR $\langle\!\langle f_1, \ldots, f_m \rangle\!\rangle_p$ can be viewed as a materialized function declared as

$$p : t_1, \ldots, t_n \rightarrow bool$$

Obviously, p is already materialized within $\langle\!\langle f_1, \ldots, f_m \rangle\!\rangle_p$ since the following equation holds:

$$p(o_1, \ldots, o_n) \quad \equiv \quad \Big((o_1, \ldots, o_n) \in \pi_{O_1,\ldots,O_n} \; \langle\!\langle f_1, \ldots, f_m \rangle\!\rangle_p\Big)$$

The materialization of a restriction predicate is invalidated by updates of the object base in the same way as "normal" materialized functions. Thus, to keep the "materialization" of a restriction predicate up to date, the algorithms outlined in Section 4 are employed. Every function invoked by the evaluation of the restriction predicate p is modified. During the materialization process entries are inserted into the RRR which represent objects accessed during the evaluation of p.

If an update of some object is conveyed to the GMR manager that affects the evaluation of a restriction predicate, the following algorithm is performed by *GMR_Manager.invalidate*:

predicate(o) $\equiv$ **foreach** triple $[o, p, \langle o_1, \ldots, o_n \rangle]$ **in** *RRR* **do**
- (1) **remove** $[o, p, \langle o_1, \ldots, o_n \rangle]$ **from** *RRR*
- (2) **recompute** $p(o_1, \ldots, o_n)$ and
  - $*$ **remember** all accessed objects $\{o'_1, \ldots, o'_p\}$
  - $*$ **if** $p(o_1, \ldots, o_n) =$ true
    then
    insert $[o_1, \ldots, o_n, f_1(o_1, \ldots, o_n), true, \ldots, f_m(o_1, \ldots, o_n), true]$
    into $\langle\!\langle f_1, \ldots, f_m \rangle\!\rangle_p$ (if not present)
    else
    remove $[o_1, \ldots, o_n, F_1, V_1, \ldots, F_m, V_m]$
    from $\langle\!\langle f_1, \ldots, f_m \rangle\!\rangle_p$ (if present)
- (3) **foreach** v **in** $\{o'_1, \ldots, o'_p\}$ **do**
  - $*$ **insert** the triple $[v, p, \langle o_1, \ldots, o_n \rangle]$ **into** *RRR* (if not present)

Steps 1 and 3 of this algorithm are similar to steps 1 and 3 of the immediate rematerialization algorithm described in Subsection 4.1. In step 2 the predicate result is recomputed. If the result is *true* the appropriate function results are materialized. Otherwise, the appropriate GMR entries are removed (if present).

6.2 Materialized Functions with Atomic Argument Types

A function with an atomic argument type, e.g., *float* or *int*, cannot be materialized for all argument combinations—a restriction predicate must be specified to determine the set of argument values for which results are materialized. For example, consider the function *Cuboid.weight* that is now modified to compute the weight of a *Cuboid* instance with respect to the gravitational force of the site where the cuboid resides:

> **declare** weight: Cuboid || float → float;
>
> **define** weight (gravitation) **is**
>   **return self.**volume ∗ **self.**Mat.SpecWeight ∗ gravitation / 9.81;

If we want to materialize the *weight* of all *Cuboid* instances for all planets of our solar system, we can do that by introducing the restricted GMR $\langle\langle weight \rangle\rangle_p$ with the restriction predicate p defined as

$$p \equiv (gravitation = 9.81 \vee \ldots \vee gravitation = 22.01)$$

There are two ways to restrict arguments of atomic types: An argument x of an atomic type is called *range-restricted*, if it is restricted by a predicate of the form $lb \leq x \leq ub$. x is called *value-restricted* if its restriction predicate is of the form $x = v_1 \vee \ldots \vee x = v_k$. To value-restrict an argument x the database programmer can build a set-object containing the values $v_1, \ldots, v_{k_x}$—the restriction predicate for x is then given as $x \in \{v_1, \ldots, v_{k_x}\}$.

float-valued arguments of a materialized function must always be value-restricted, whereas *int*-valued arguments may be value- or range-restricted.

7 Benchmarking two Example Applications

This section sketches the results of a first quantitative analysis of the function materialization concept, using two example applications. The first application is based on the *Cuboid* example that has been used in the previous sections to illustrate our algorithms and data structures. The second application is derived from a more traditional database domain: the personnel and project administration in a large company.

The benchmarks were run on our experimental object base system GOM that is built on top of the EXODUS storage manager [5]. The database was stored on a DEC disk (with 25 ms average transfer time) directly connected to a DEC Station 3100 with 16 MByte main memory running under the Ultrix operating system. The reported times correspond to the user times, i.e., the actual times a user has to wait to obtain the result. The benchmark was run in single user mode, thus eliminating interaction by concurrent users. Since the described applications are rather small we decided to use a correspondingly small database buffer of 600 kBytes to compensate for the small database volume.

7.1 Benchmarking the Cuboid Example

Specification of the Application Profiles

This analysis is based on the *Cuboid* example that has been used in the previous sections to illustrate our algorithms and data structures. All subsequent results were measured on a database containing 8000 *Cuboid* instances, each *Cuboid* referencing 8 *Vertex* instances and one *Material* instance.

The operation mix is described as a quadruple

$$M = (Q_{mix}, U_{mix}, P_{up}, \#ops)$$

Here, the query mix Q_{mix} is the set of (two) weighted queries of the form $Q_{mix} = \{(w_1, Q_{bw}), (w_2, Q_{fw})\}$ where $w_1 + w_2 = 1$. The two queries—Q_{bw} (backward query) and Q_{fw} (forward query)—are outlined as follows, where r and *randomID* are randomly chosen and ε is a small constant:

$Q_{bw} \equiv$ **range** c: Cuboid
retrieve c
where $r - \varepsilon <$ c.volume $< r + \varepsilon$

$Q_{fw} \equiv$ **range** c: Cuboid
retrieve c.volume
where c.CuboidID[7]= randomID[8]

The update mix $U_{mix} = \{(w'_1, D), (w'_2, I), (w'_3, S), (w'_4, R), (w'_5, T)\}$ consists of weighted update operations. The letters represent the following updates: D denotes the deletion of a randomly chosen *Cuboid*,[8] I denotes the creation of a new *Cuboid* of randomly chosen dimensions, and S, R and T represent scalation, rotation and translation of a randomly chosen *Cuboid*, respectively. Again the sum of all weights has to be 1, i.e., $\sum_{i=1}^{5} w'_i = 1$.

The weights indicate the relative probability that one particular update (query) is chosen from the set of possible updates (queries). For example, if a query is to be performed it will be the backward query Q_{bw} with probability w_1.

The update probability P_{up} determines the ratio between updates and queries in the benchmarked application. For example, a value $P_{up} = 0.1$ determines that—on the average—out of 100 operations we will encounter 10 updates which are randomly chosen from the set U_{mix}—according to the weights $w'_1, \ldots, w'_5$—and 90 queries which are randomly chosen from the set Q_{mix}—according to the weights w_1 and w_2.

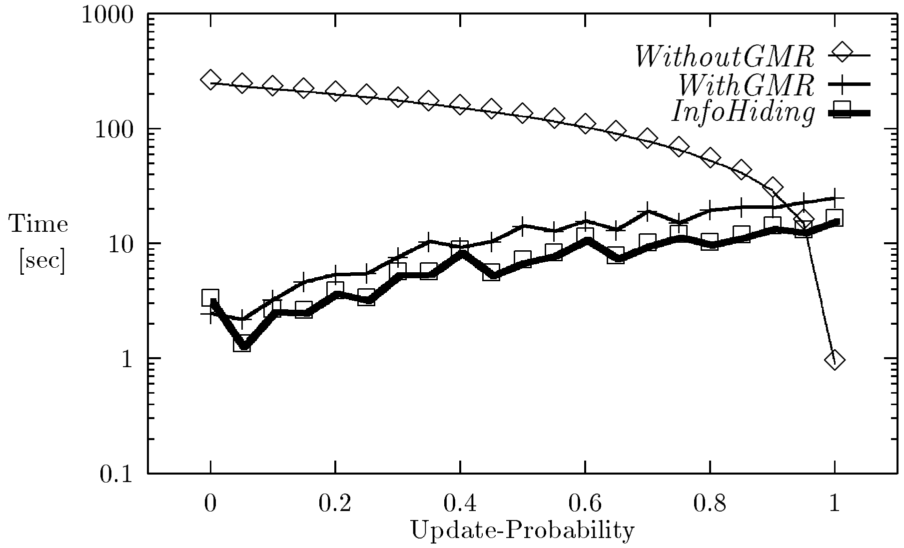
The variable $\#ops$ denotes the total number of operations performed in the described benchmark.

Benchmark Results

The first benchmark determines the performance of function materialization for an application profile under varying update probabilities. The update probability was varied from 0 to 1 with increments of 0.05. For each update probability 40 operations were executed on the object base. The application profile and the performance measurements are graphically visualized in Figure 7—operations whose associated weight is 0 are omitted from the operation mix specification. The update probability is plotted against the x-axis and the time to perform the 40 operations is plotted against the logarithmically scaled y-axis. We measured three different program versions:

- *WithoutGMR*: the "normal" program without any function materialization.
- *WithGMR*: in this configuration the GMR $\langle\langle volume \rangle\rangle$ is maintained under *immediate* rematerialization and utilized to evaluate the queries.
- *InfoHiding*: in this version the GMR $\langle\langle volume \rangle\rangle$ is maintained under information hiding—as described in Section 5.3—to reduce the invalidation and rematerialization overhead.

[7] *CuboidID* is an additional user-supplied identifier to uniquely select a particular Cuboid.
[8] Finding the qualifying *Cuboid* was supported by an index.

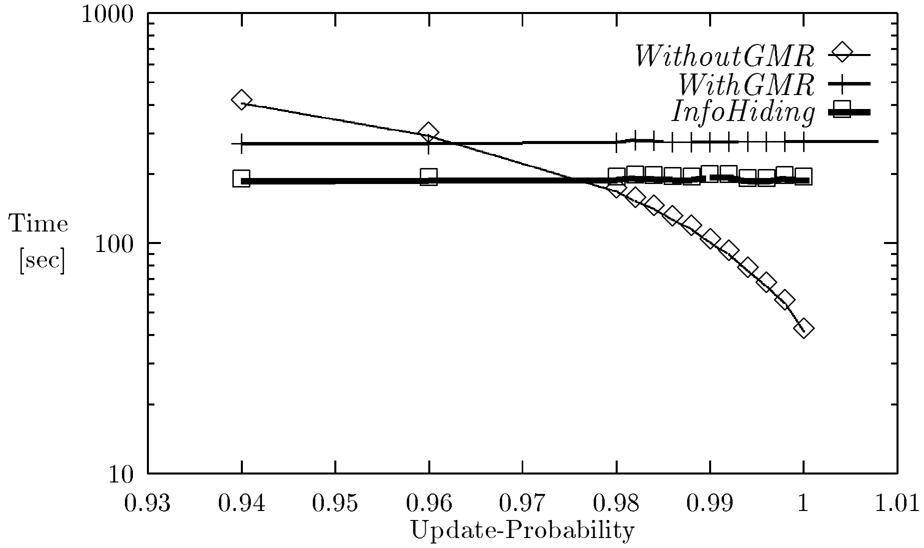


#ops	40	
Q_{mix}	Q_{bw}	Q_{fw}
	0.5	0.5
U_{mix}	I	S
	0.5	0.5
P_{up}	0 **step** .05 **to** 1	

Figure 7: Performance of GMR under Varying Update Probabilities

From the plot in Figure 7 we can deduce that up to an update probability of about 0.9 the GMR-version outperforms the non-supported version. Exploiting information hiding in the GMR maintenance moves the break even point to about $P_{up} = 0.95$.

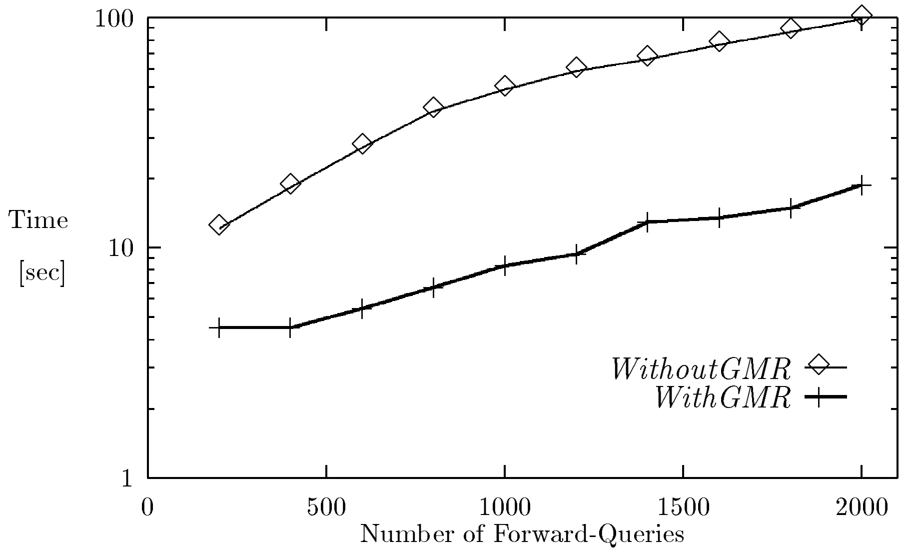
In the next benchmark a slightly different application profile was utilized: now we perform 500 operations where each operation is either a backward query or a *scale*—the relative number depending on the update probability. The update probability is varied between 0.94 and 1.0 with the first two increments being 0.02 and the remaining increments being 0.002. The results of this benchmark are visualized in Figure 8. Again we compare the three program versions *WithoutGMR*, *WithGMR*, and *InfoHiding*. In this example application the break even point of the *WithGMR* version versus the non-supported *WithoutGMR* version is around 0.96, and the break-even point between the unsupported program and GMR with information hiding (*InfoHiding*) is at an update probability of about 0.975.



#ops	500
Q_{mix}	Q_{bw}
	1.0
U_{mix}	S
	1.0
P_{up}	0.94 **to** 1.0

Figure 8: Determining the Break-Even Point of Function Materialization

From these two first benchmarks we can conclude that materialization achieves a tremendous performance gain for backward queries. In the next benchmark we want to investigate the costs of forward queries for which the gain due to materialization is less dramatic but—as it turns out—still significant. In this benchmark we steadily increase the number of forward queries, the only operation performed in this benchmark. The results are shown in Figure 9. We observe that the exploitation of the GMR $\langle\langle$*volume*$\rangle\rangle$ constitutes a performance gain of about a factor 4 to 5. The reader should notice, however, that in this benchmark only queries and no updates were performed.



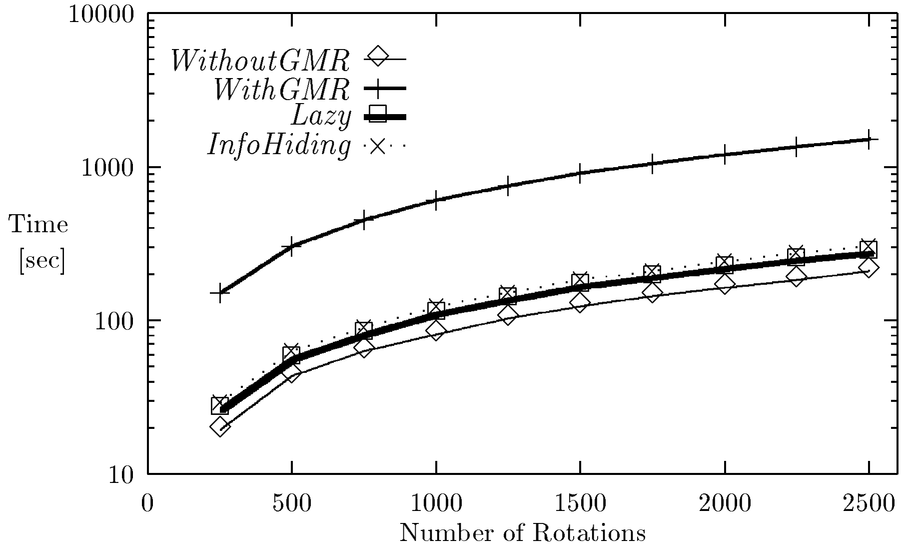
#ops	200 **to** 2000
Q_{mix}	Q_{fw} 1.0
U_{mix}	—
P_{up}	0.0

Figure 9: Cost of Forward Queries

The subsequent benchmark was designed to investigate the overhead of invalidation and rematerialization incurred by function materialization. For this purpose we used an application profile that consists of only *rotate* operations, the number of which is steadily increased. The results are visualized in Figure 10. Aside from the three previously introduced program versions *WithGMR*, *WithoutGMR* and *InfoHiding*, we incorporated into this benchmark a fourth system configuration, called *Lazy*. In this configuration we maintained the GMR $\langle\langle$*volume*$\rangle\rangle$ under *lazy rematerialization*. Under *Lazy* all materialized *volume* results had been invalidated before the benchmark was started—this causes the RRR and the sets *ObjDepFct* to be empty with respect to $\langle\langle$*volume*$\rangle\rangle$. Nevertheless, this configuration still imposes a penalty on performing a geometric transformation due to the checks that have to be made within objects of type *Vertex*—to determine that the set *ObjDepFct* is empty. From Figure 10 we conclude that this penalty is, however, rather low since the curves *WithoutGMR* and *Lazy* run very close. This means that switching from *immediate* rematerialization to *lazy* rematerialization drastically decreases the update penalty. This makes our materialization concept even viable for application domains where occasional "bursts of updates" are followed by prolonged periods of a rather static behavior, e.g., the life cycle of an engineering artifact.

The *InfoHiding* version induces an overhead that is similar to *Lazy*—remember that we only perform *rotate* operations which, under information hiding, do not require an invalidation. However, if the benchmark consisted of *scale* operations the *InfoHiding* configuration would have much higher overhead than the *Lazy* version.

We remember that the "normal" *WithGMR* version cannot detect that *rotate* is irrelevant for materialized *volume* results. Therefore, a substantial penalty is incurred due to the invalidation and rematerialization. The penalty constitutes almost a factor 10 as compared to the unsupported version.



#ops	250 **to** 2500
Q_{mix}	—
U_{mix}	R 1.0
P_{up}	1.0

Figure 10: Invalidation Overhead Incurred by Materialized *volume*

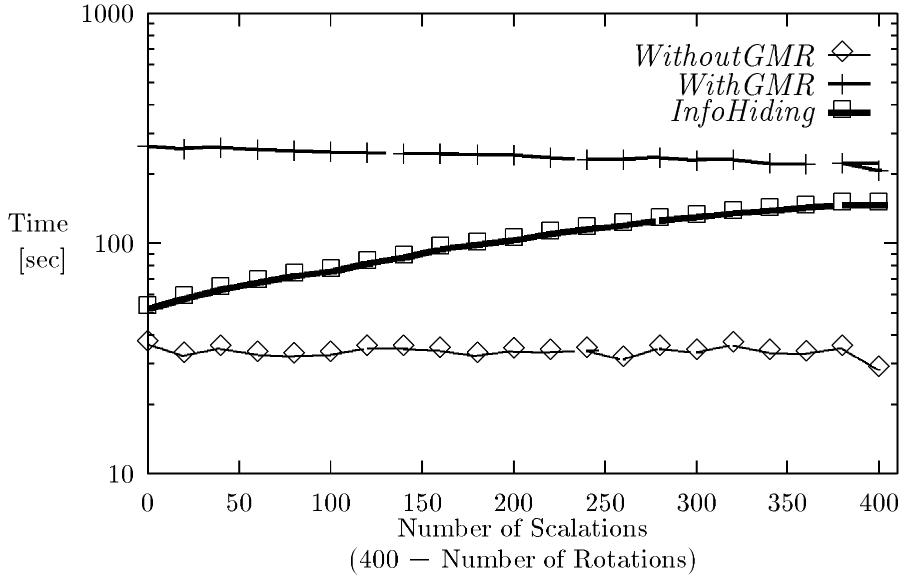
In the last benchmark we investigate the benefits of information hiding with respect to reducing the invalidation overhead. For this purpose we perform 400 operations, each being an update consisting of *scale* and *rotate* operations. We simultaneously increase the probability that a *scale* (S) is chosen from 0 to 1 and decrease the probability that a *rotate* (R) operation is performed from 1 to 0—increments and decrements being 0.05. The results are plotted in Figure 11. It turns out—as expected—that the costs for *WithoutGMR* and *WithGMR* are almost invariant to the varying ratio of *scale* to *rotate* operations. The *InfoHiding* version benefits from the operation mix that predominantly contains *rotate* operations because the *InfoHiding* version "detects" that *rotates* are irrelevant for materialized *volume* results. Therefore, the *InfoHiding* curve starts close to the *WithoutGMR* curve and steadily climbs towards the *WithGMR* cost curve as the number of *scale* operations in the operation mix is increased. But the overhead of *InfoHiding* remains well below the overhead of *WithGMR* because under *InfoHiding* each *scale* operation induces only one invalidate whereas the "normal" GMR-maintenance triggers 12 invalidations.

7.2 Benchmarking the Company Example

Description

This example is based on the matrix organization of a company and the ranking of employees. For lack of space we do not describe the GOM types involved in this example in full detail.

An object of type *Company* consists of the name of the company, all departments of the company and the set of projects that are carried out within the company. A *Department* is described by its name and the set of its employees. *Projects* are modeled by the project name, the status of the project, the size of the project and the set of programmers that are involved in the project. We consider only software projects. Thus, the size of a project is modeled by the lines of code it comprises. The status of a project is given by a decimal value that ranges between −1000 and 1000. A negative status denotes a delay and a loss that is caused by the project, a positive status denotes that the project is profitable.



#ops	400
Q_{mix}	—
U_{mix}	S 0 **step** .05 **to** 1
	R 1 **step** −.05 **to** 0
P_{up}	1.0

Figure 11: The Benefits of Information Hiding

The type *Employee* is a subtype of the type *Person*. Each *Employee* has a unique employee number, a salary and a history of jobs, which is modeled as a set of jobs. A *Job* describes the part of a *Project* that has been delegated to a particular *Employee*. Therefore, each *Job* contains a reference to the *Project*, the number of lines of code that have been written by the *Employee* and two Boolean values that denote the status of the *Employee*. The attributes of the type *Job* are used to compute an assessment value—the *ranking* of an *Employee* is then given by the average of the assessment values of all jobs in the employee's job history. The function *ranking* is associated with type *Employee*. This function is materialized for the subsequent benchmark.

The second function whose materialization has been benchmarked is the function *matrix* associated with the type *Company*. This function computes the department project matrix for a company. A department project matrix is defined as a set of tuples of the type *MatrixLine*, defined as

$$[Dep : Department, Proj : Project, Emps : SetofEmployee]$$

One tuple τ of type *MatrixLine* states that the employees contained in the set $\tau.Emps$ are employed in department $\tau.Dep$ and work in project $\tau.Proj$. The matrix contains only *MatrixLine* instances τ with $\tau.Emps \neq \{\}$.

The reference graph depicted in Figure 12 elucidates the types involved in the benchmark. An arrow leading from type t_1 to type t_2 means that objects of type t_1 contain an attribute of type t_2. Double pointed arrows denote that objects of the first type contain set-valued attributes with elements of the second type. Arrows leading from t_1 to t_2 are labeled with the name of the appropriate attribute.

Specification of the Application Profiles

The database contains one *Company* instance and 20 *Departments*, each of which has 100 *Employees*. 1000 *Projects* are stored in the database. In the following we assume that the considered company is referenced by the variable *comp*. On the average every employee has been involved in 10 projects.

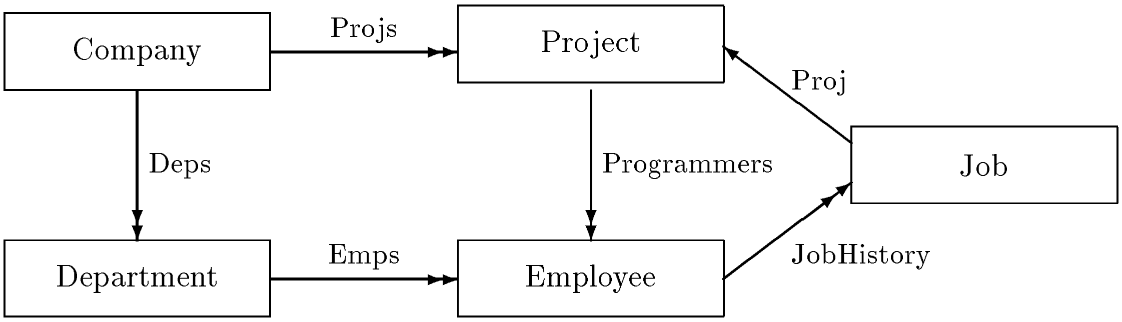


Figure 12: Reference Graph of Types in the Company Example

Again, the operation mix is described as a quadruple

$$M = (Q_{mix}, U_{mix}, P_{up}, \#ops)$$

The query mix consists of forward queries and backward queries on the (materialized) function *ranking*, and of selections on the result of the function *matrix*. It is specified by the set of three weighted queries $Q_{mix} = \{(w_1, Q_{fw,r}), (w_2, Q_{bw,r}), (w_3, Q_{sel,m})\}$ where $w_1 + w_2 + w_3 = 1$. $Q_{fw,r}$ and $Q_{bw,r}$ denote forward and backward queries on *ranking*, respectively. $Q_{fw,r}$ and $Q_{bw,r}$ are specified as follows, with *randomNo* and r randomly chosen and ε being some small constant:

$Q_{fw,r} \equiv$ **range** e: Employee
retrieve e.ranking
where e.EmpNo = randomNo

$Q_{bw,r} \equiv$ **range** e: Employee
retrieve e
where r $-\ \varepsilon <$ e.ranking $<$ r $+\ \varepsilon$

The result of an invocation *comp.matrix* is a set of tuples of type *MatrixLine*, i.e., tuples of the type [*Dep*, *Proj*, *Emps*]. The query $Q_{sel,m}$ selects all tuples of *comp.matrix* that contain a randomly chosen *Department*, and retrieves the appropriate *Proj* field:

$Q_{sel,m} \equiv$ **range** d: Department, l: MatrixLine
retrieve l.Proj
where l **in** comp.matrix **and**
d **in** l.Dep **and**
d.DepNo = randomNo

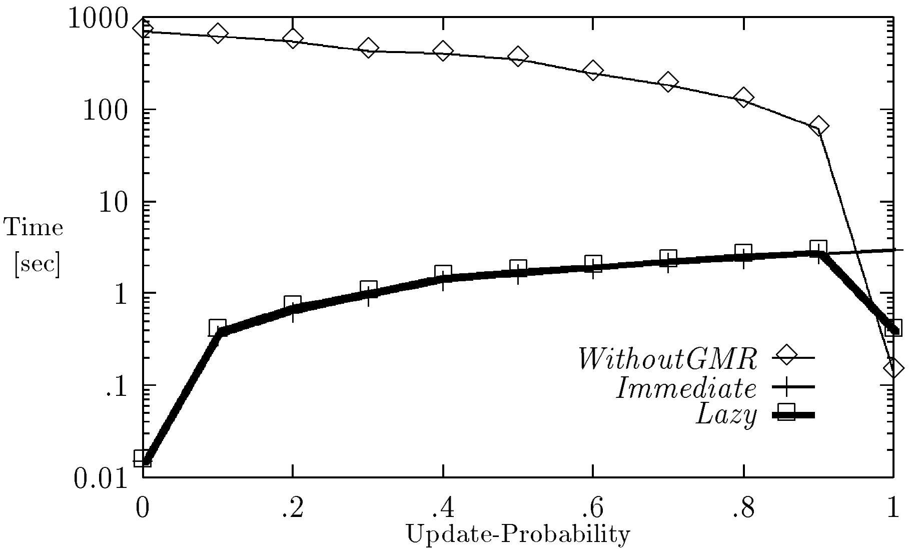
The update mix is specified by the set $U_{mix} = \{(w'_1, N), (w'_2, P)\}$ with $w'_1 + w'_2 = 1$. N denotes the insertion of a new instance of type *Employee*. P denotes the promotion or degradation of a randomly chosen employee affecting his or her status.

Analogous to the previous benchmark P_{up} denotes the probability that one operation is an update rather than a query, and *#ops* denotes the number of operations performed in the described benchmarks.

Benchmark Results

We measured the following program versions to evaluate the materialization of *ranking*:

- *WithoutGMR*: the "normal" program without any function materialization.



#ops	10
Q_{mix}	$Q_{bw,r}$ 1.0
U_{mix}	P 1.0
P_{up}	0 **to** 1 **step** 0.1

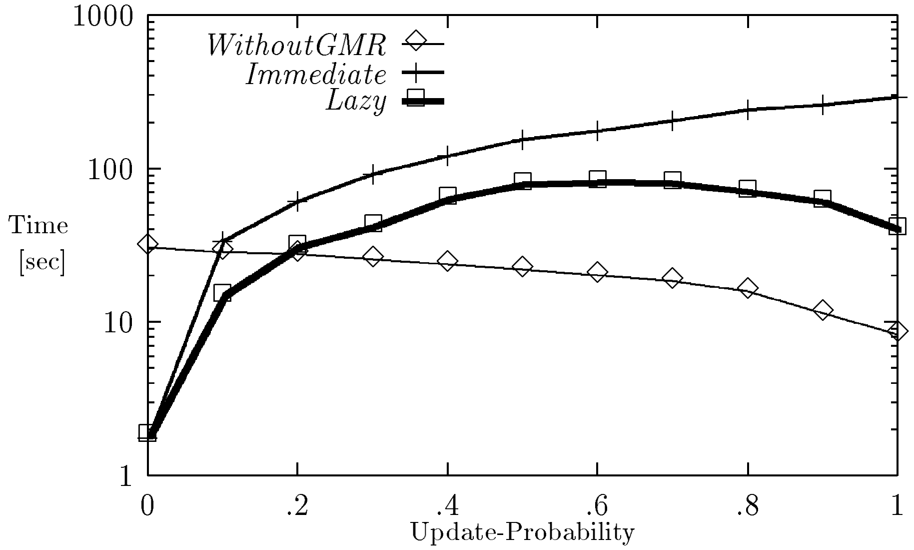
Figure 13: Cost of Backward Queries

- *Immediate*: the program version which maintains the GMR $\langle\langle ranking \rangle\rangle$ under immediate rematerialization
- *Lazy*: the program version which maintains the GMR $\langle\langle ranking \rangle\rangle$ under lazy rematerialization

The first benchmark illustrates the performance gain for backward queries obtained from the materialization of *ranking*. The update probability varies between 0.0 and 1.0; for each update probability 10 operations are performed. The operation mix consists of backward queries and update operations, i.e., the promotion of employees. The benchmark results are shown in Figure 13. For update probabilities below 0.95 both versions using the GMR outperform the non-supported version. The *Lazy* and *Immediate* rematerialization strategies show no performance difference in this benchmark except for $P_{up} = 1.0$. This is due to the fact that for backward queries all materialized results have to be valid.

In the next benchmark depicted in Figure 14 we investigated the costs of an operation mix consisting of forward queries and update operations, i.e., the promotion of employees. For each update probability, 1000 operations have been performed. Here, *Lazy* rematerialization achieves a performance gain of about a factor 2 to 12 with respect to *Immediate* rematerialization, as—under *Lazy* rematerialization—invalidated results of *ranking* are only recomputed if they are accessed. The costs for the *Lazy* strategy decreases for P_{up} between 0.6 and 1.0 as the probability to access an invalidated function result decreases for higher update probabilities. In this benchmark the break even point of the program version *WithoutGMR* with respect to the supported versions lies at $P_{up} = 0.1$ for the immediate rematerialization strategy and at $P_{up} = 0.2$ for the lazy rematerialization strategy.

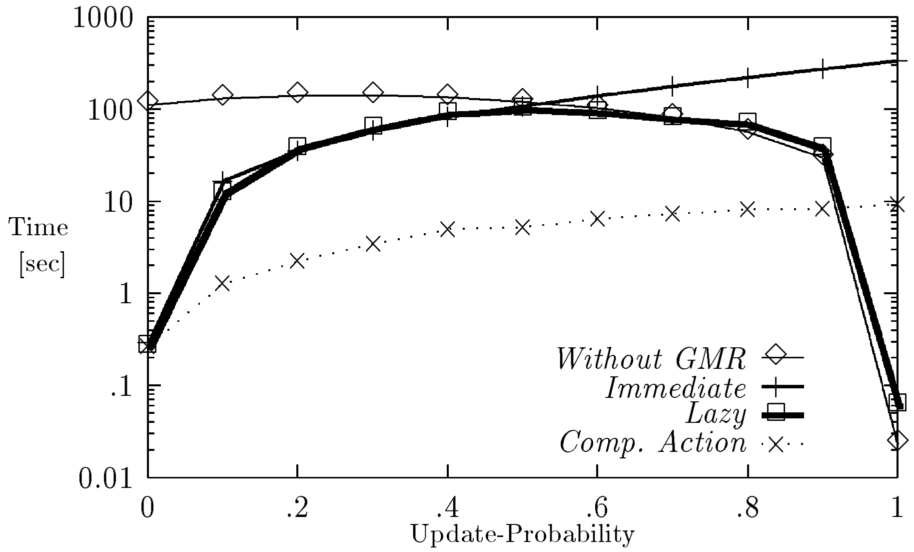
The previous two benchmarks showed that the break-even point for the materialization of rather simple functions accessing only a small part of the object base is dependent on the number of backward queries contained in the query mix. In the next benchmark we investigate the materialization of a more costly function, the computation of the project department matrix of a company. To benchmark the function *matrix*, the size of the company referenced by *comp* has been decreased to 5 departments and 100 projects.



#ops	1000
Q_{mix}	$Q_{fw,r}$ 1.0
U_{mix}	P 1.0
P_{up}	0 **to** 1 **step** 0.1

Figure 14: Cost of Forward Queries

Each department has 10 employees, and 5 employees are involved in one project. The number of jobs per employee remains invariant (10). As we have only one company in our object base the GMR $\langle\langle matrix \rangle\rangle$ contains just one materialized result.



#ops	10
Q_{mix}	$Q_{sel,m}$ 1.0
U_{mix}	N 1.0
P_{up}	0 **to** 1 **step** 0.1

Figure 15: The Benefits of Compensating Actions

The operation mix used in this benchmark consists of selections on the project department matrix ($Q_{sel,m}$) and of insert operations that introduce new projects (N). Four versions of the program were considered: *Without* GMR, with the GMR $\langle\langle matrix \rangle\rangle$ under *Lazy* and under *Immediate* rematerialization and with a compensating action that compensates for the insertion of a new project.

The update probability ranges from 0.0 to 1.0. For each update probability 10 operations were performed. As there exists just *one* materialized result every update operation leads to the invalidation of this result. Under the program versions using the *Immediate* strategy or the compensating action every update leads to the recomputation of the matrix. The results of this benchmark are shown in Figure 15.

For P_{up} below 0.5 the curves *Immediate* and *Lazy* are very close, as every update eventually leads to the recomputation of *comp.matrix* in both versions (under lazy rematerialization the recomputation is triggered by the access to the invalidated matrix after an update). With increasing update probability the probability of two or more subsequent update operations increases, leading to a tremendous performance gain of *Lazy* rematerialization with respect to *Immediate* rematerialization. In fact, the supported version under *Lazy* rematerialization performs as well as the unsupported version for $0.5 < P_{up} \leq 0.9$.

For update probabilities between 0.0 and 0.9 the program version using a compensating action outperforms the other three versions. This is due to the fact that using a compensating action an update does not lead to the recomputation of the whole matrix. For high update probabilities ($P_{up} > 0.9$), the *Lazy* version becomes superior to the version using the compensating action, as under the *Lazy* strategy subsequent updates do not lead to a rematerialization.

From this benchmark two conclusions can be drawn:

1. Lazy rematerialization is always superior to immediate rematerialization if all materialized objects are affected by updates and if several updates are performed subsequently.

2. Compensating actions may drastically reduce the update costs and thus decrease the penalty paid by the rematerialization mechanism.

8 Conclusion

In this paper we developed an architecture and efficient algorithms for the maintenance of materialized functions in object-oriented databases. Our architecture provides for easy incorporation of function materialization into existing object base systems because it is largely based on rewriting the schema. In the design of the maintenance algorithms we placed particular emphasis on reducing the invalidation and rematerialization overhead. By exploiting the object-oriented paradigm—namely object typing, object identity, and encapsulation—we were able to achieve fine-grained control over the invalidation requirements and, thus, to lower the invalidation and rematerialization penalty incurred by update operations. The recomputation overhead for invalidated results can be further reduced by providing compensating actions.

In addition, one can tune the system by switching between *immediate* and *lazy* rematerialization. The latter strategy can be used to decrease the penalty during update-intensive phases even further—for example, in a database to store engineering artifacts where periodically some of the objects are extensively being modified (design phase) whereas the remaining time the object base remains mostly static.

On an experimental basis we incorporated function materialization—currently limited to single function GMRs—in our object base management system GOM. The first quantitative analyses gathered from two benchmark sets, one from the computer geometry domain and one from a more traditional administrative application are very promising. Especially when functions are utilized in search predicates—our so-called backward queries—the materialization constitutes a tremendous performance gain, even for rather high update probabilities.

Currently, we are extending our rule-based query optimizer [12] to generate query evaluation plans that utilize materialized values instead of recomputing them.

Acknowledgements

Peter C. Lockemann's continuous support of our research is gratefully acknowledged. Andreas Horder carried out the computer geometry benchmark; Michael Steinbrunn participated in the design and prototypical realization of the concepts. We further thank the anonymous referees for their many valuable comments.

References

[1] M. E. Adiba and B. G. Lindsay. Database snapshots. In *Proc. of The Conf. on Very Large Data Bases (VLDB)*, pages 86–91, Montreal, Canada, Aug 80.

[2] M. Atkinson, F. Bancilhon, D. J. DeWitt, K. R. Dittrich, D. Maier, and S. Zdonik. The object-oriented database system manifesto. In *Proc. Int. Conf on Deductive and Object-Oriented Databases*, pages 40–57, Kyoto, Japan, Dec 1989.

[3] J. Blakeley, N. Coburn, and P. Larson. Updating derived relations: Detecting irrelevant and autonomously computable updates. *ACM Trans. on Database Systems*, 14(3):369–400, Sep 89.

[4] J. Blakeley, P. Larson, and F. Tompa. Efficiently updating materialized views. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 61–71, Washington, D.C., 1986.

[5] M. J. Carey, D. J. DeWitt, J. E. Richardson, and E. J. Shekita. Object and file management in the EXODUS extensible database system. In *Proc. of The Conf. on Very Large Data Bases (VLDB)*, pages 91–100, Kyoto. Japan, Aug 1986.

[6] E. Hanson. A performance analysis of view materialization strategies. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 440–453, San Francisco, CA, May 87.

[7] E. Hanson. Processing queries against database procedures. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, Chicago, May 88.

[8] G. Huet. Confluent reductions: Abstract properties and applications of term rewriting systems. *Journal of the ACM*, 27(4):797–821, 1980.

[9] A. Jhingran. A performance study of query optimization algorithms on a database system supporting procedures. In *Proc. of The Conf. on Very Large Data Bases (VLDB)*, pages 88–99, L.A., CA, Sep 1988.

[10] A. Kemper, C. Kilger, and G. Moerkotte. Function materialization in object bases. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 258–267, Denver, CO, May 1991.

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

[12] A. Kemper and G. Moerkotte. Advanced query processing in object bases using access support relations. In *Proc. of The Conf. on Very Large Data Bases (VLDB)*, pages 290–301, Brisbane, Australia, Aug 1990.

[13] A. Kemper and G. Moerkotte. *Object-Oriented Information Management for Engineering Applications*. Prentice Hall, Englewood Cliffs, N.J., 1992 (forthcoming).

[14] A. Kemper, G. Moerkotte, H.-D. Walter, and A. Zachmann. GOM: a strongly typed, persistent object model with polymorphism. In *Proc. of the German Conf. on Databases in Office, Engineering and Science (BTW)*, pages 198–217, Kaiserslautern, Mar 1991. Springer-Verlag, Informatik Fachberichte Nr. 270.

[15] V. Lum. Multi-attribute retrieval with combined indexes. *Communications of the ACM*, 13:660–665, 1970.

[16] J. Nievergelt, H. Hinterberger, and K. C. Sevcik. The grid file: An adaptable, symmetric multikey file structure. *ACM Trans. on Database Systems*, 9(1):38–71, 1984.

[17] D. Rosenkrantz and H. Hunt. Processing conjunctive predicates and queries. In *Proc. of The Conf. on Very Large Data Bases (VLDB)*, pages 64–72, Montreal, 1980.

[18] T. K. Sellis. Intelligent caching and indexing techniques for relational database systems. *Information Systems*, 13(2):175–186, 1988.

[19] M. Stonebraker, J. Anton, and E. Hanson. Extending a database system with procedures. *ACM Trans. on Database Systems*, 12(3):350–376, Sep 1987.

[20] M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos. On rules, procedures, caching and views in data base systems. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 281–290, Atlantic City, NJ, May 90.

[21] M. Stonebraker and L. Rowe. The design of POSTGRES. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 340–355, Washington, D.C., 1986.

Appendix: Extracting the Relevant Path Expressions

Subsequently we present a formal method to determine the set *RelAttr(f)* for a materialized function $f : t_1, \ldots, t_n \to t_{n+1}$. This method is based on the extraction of relevant path expressions from f. A path expression $t_i.A_1.\cdots.A_k$ $(1 \le i \le n,\ k \ge 1)$ is relevant to f if f uses the value of $v.A_1.\cdots.A_k$ for some variable v of type t_i to compute its result. For example, the path expression $Cuboid.V1.X$ is relevant to the function $Cuboid.volume$, as the X coordinate of Vertex $V1$ of a cuboid is used to determine the volume of the cuboid.

To determine the relevant path expressions of a function f we assign a *path extraction structure* $\mathcal{E}(\mathcal{S}) = (\mathcal{P}_\mathcal{S}, \mathcal{R}_\mathcal{S})$ to each syntactic structure $\mathcal{S}$, i.e., expression, statement or function, that appears in the body of f. Herein, $\mathcal{P}_\mathcal{S}$ is the set of path expressions which can be extracted from $\mathcal{S}$. $\mathcal{R}_\mathcal{S}$ is a term rewriting system, as examined by Huet in [8], containing rules of the form $v \to p$ where v is a variable and p is a path expression. The application of a rule $v \to p$ to a path expression $v.A_1.\cdots.A_k$ replaces v by p. The occurrence of a rule $v \to p$ in $\mathcal{R}_\mathcal{S}$ indicates that an assignment of the form $v := e$ occurs in $\mathcal{S}$ or in any function that is called by $\mathcal{S}$. Here, e is an expression and p can be extracted from e.

The relevant path expressions of a sequence of statements $s_1; \ldots; s_n$ are extracted by combining the path extraction structures $\mathcal{E}(s_1), \ldots, \mathcal{E}(s_n)$ using the overloaded operator $\circ$, which is introduced by Definition 8.1. $\mathcal{E}(s_1; \ldots; s_n)$ is then given as $\mathcal{E}(s_1)\circ\ldots\circ\mathcal{E}(s_n)$—the first component of $\mathcal{E}(s_1; \ldots; s_n)$ contains the set of relevant path expressions of $s_1; \ldots; s_n$.

Definition 8.1 *Let $\mathcal{P}$, $\mathcal{P}_1$ and $\mathcal{P}_2$ be sets of path expressions, $\mathcal{R}$, $\mathcal{R}_1$ and $\mathcal{R}_2$ rewrite systems and $\mathcal{E}_1 = (\mathcal{P}_1, \mathcal{R}_1)$ and $\mathcal{E}_2 = (\mathcal{P}_2, \mathcal{R}_2)$ path extraction structures.*

$$\begin{aligned}
\mathcal{R}_1 \circ \mathcal{R}_2 &:= \{x \to z' \mid x \to_{\mathcal{R}_1} z, z \to_{\mathcal{R}_2} z'\} \cup \{x \to z \in \mathcal{R}_1 \mid \forall z' :\ z \not\to_{\mathcal{R}_2} z'\} \\
\mathcal{P} \circ \mathcal{R} &:= \{z' \mid z \in \mathcal{P}, z \to_{\mathcal{R}} z'\} \cup \{z \in \mathcal{P} \mid \forall z' :\ z \not\to_{\mathcal{R}} z'\} \\
\mathcal{E}_1 \circ \mathcal{E}_2 &:= ((\mathcal{P}_2 \circ \mathcal{R}_1) \cup \mathcal{P}_1,\ (\mathcal{R}_2 \circ \mathcal{R}_1) \cup (\mathcal{R}_1 \setminus \{x \to z \in \mathcal{R}_1 \mid \exists z' :\ x \to_{\mathcal{R}_2} z'\}))
\end{aligned}$$

$\circ$ is left associative. Thus, a sequence $\mathcal{E}_1 \circ \mathcal{E}_2 \circ \ldots \circ \mathcal{E}_n$ is evaluated from left to right. □

The meaning of the terms $\mathcal{R}_1 \circ \mathcal{R}_2$ and $\mathcal{P} \circ \mathcal{R}$ is obvious. To understand the meaning of $\mathcal{E}_1 \circ \mathcal{E}_2$, remember that $\mathcal{E}_1 \circ \mathcal{E}_2$ yields the path extraction structure of a sequence of two statements, say s_1 and s_2. Note that s_1 is executed before s_2. The first component of $\mathcal{E}_1 \circ \mathcal{E}_2$ contains the path expressions of s_2 after being rewritten by $\mathcal{R}_1$ and all path expressions of s_1. The path expressions of s_2 have to be rewritten as they may start with a variable that is mapped to another path expression by a rule in $\mathcal{R}_1$. The second component of $\mathcal{E}_1 \circ \mathcal{E}_2$ contains the rules of $\mathcal{R}_2$ after rewriting by the rules of $\mathcal{R}_1$. Further, all rules of $\mathcal{R}_1$ are contained in the rewriting system of $\mathcal{E}_1 \circ \mathcal{E}_2$ except for those rules, that have the same left hand side as any rule in $\mathcal{R}_2$. To understand this, remember that very rule $v \to p$ indicates the occurrence of an assignment of some expression to the variable v. Thus, if s_2 re-assigns a variable v that has already been used before, then all rules in $\mathcal{R}_1$ that reflect the old value of v have to be abandoned.

As mentioned before, the path extraction structure of an assignment $v := e$ is given by $\mathcal{E}(v := e) := (\mathcal{P}(e),\ \{v \to z | z \in \mathcal{P}(e)\})$. The definition of path extraction structures for all expressions, statements, and function declarations is given in Figure 16. For simplicity we assume that none of the used functions is recursive and that none of the used functions has any side-effects. In general, $\mathcal{P}(f)$ is a superset of the set of path expressions which are evaluated during an invocation of f.

The set of path expressions relevant to a materialized function f is used to determine the set *RelAttr(f)*. Thus, we are only interested in path expressions of the form $t.A$. Therefore, in a last step, we rewrite all variables v_i in $\mathcal{P}(f)$ by t_i if t_i is the type of v_i. Further, all paths of the form $t.A_1.\cdots.A_k$ in $\mathcal{P}(f)$ are cut into path expressions with a maximal length of two.

[9] *op* can be any binary operator, e.g. $+$, $-$, $*$, $<$, $=$, $\ldots$

Let e be an expression. If e is	
$v.A_1.\cdots.A_k$	$\mathcal{E}(e) := (\{v.A_1.\cdots.A_k\}, \emptyset)$
e_1 *op* e_2[9]	$\mathcal{E}(e) := (\mathcal{P}(e_1) \cup \mathcal{P}(e_2), \emptyset))$
$f(e_1, \ldots, e_n)$	$\mathcal{E}(e) := (\mathcal{P}(f) \circ \mathcal{R}(v_1 := e_1) \circ \ldots \circ \mathcal{R}(v_n := e_n), \emptyset)$
$e'.f(e_1, \ldots, e_n)$	$\mathcal{E}(e) := (\mathcal{P}(f) \circ \mathcal{R}(v_1 := e_1) \circ \ldots \circ \mathcal{R}(v_n := e_n)$ $\circ \mathcal{R}(\textbf{self} := e'), \emptyset)$
with f defined as **define** $f(v_1, \ldots, v_n)$ body;	
Let s be a statement. If s is	
$s_1; \ldots; s_n;$	$\mathcal{E}(s) := \mathcal{E}(s_1) \circ \ldots \circ \mathcal{E}(s_n)$
$v := e;$	$\mathcal{E}(s) := ((\mathcal{P}(e), \{v \rightarrow z \mid z \in \mathcal{P}(e)\})$
foreach v **in** e **do** s';	$\mathcal{E}(s) := (\mathcal{P}(e), \{v \rightarrow z \mid z \in \mathcal{P}(e)\}) \circ \mathcal{E}(s')$
if e **then** s_1 **else** s_2;	$\mathcal{E}(s) := \mathcal{E}(e) \circ (\mathcal{P}(s_1) \cup \mathcal{P}(s_2), \mathcal{R}(s_1) \cup \mathcal{R}(s_2))$
while e **do** s';	$\mathcal{E}(s) := \mathcal{E}(e) \circ \mathcal{E}(s') \circ \mathcal{E}(e)$
return e;	$\mathcal{E}(s) := \mathcal{E}(e)$
Let f be a function defined as	
define $f(v_1, \ldots, v_n)$ body;	$\mathcal{E}(f) := \mathcal{E}(\text{body})$

Figure 16: Definition of path extraction structures

Example: This example illustrates the extraction of the relevant path expressions from the function *Cuboid.length* that invokes *Vertex.dist*. The statements of both functions are numbered. To each statement the appropriate path extraction structure is assigned. Note that all identifiers must be unique—thus, $\textbf{self}_c$ and $\textbf{self}_v$ are introduced.

#	Statement	Path Extraction Structure
	define length **is**	
1	**return self**.V1.dist(**self**.V2);	$\mathcal{E}_1 := (\mathcal{P}(\text{dist}) \circ \mathcal{R}(\text{vertex2} := \textbf{self}_c.\text{V2}) \circ \mathcal{R}(\textbf{self}_v := \textbf{self}_c.\text{V1}), \emptyset)$
	define dist(vertex2) **is**	
	var dx, dy, dz: float;	
	begin	
2	dx := **self**.X − vertex2.X;	$\mathcal{E}_2 := (\{\textbf{self}_v.\text{X}, \text{vertex2.X}\}, \{\text{dx} \rightarrow \textbf{self}_v.\text{X}, \text{dx} \rightarrow \text{vertex2.X}\})$
3	dy := **self**.Y − vertex2.Y;	$\mathcal{E}_3 := (\{\textbf{self}_v.\text{Y}, \text{vertex2.Y}\}, \{\text{dy} \rightarrow \textbf{self}_v.\text{Y}, \text{dy} \rightarrow \text{vertex2.Y}\})$
4	dz := **self**.Z − vertex2.Z;	$\mathcal{E}_4 := (\{\textbf{self}_v.\text{Z}, \text{vertex2.Z}\}, \{\text{dz} \rightarrow \textbf{self}_v.\text{Z}, \text{dz} \rightarrow \text{vertex2.Z}\})$
5	**return** sqrt($\text{dx}^2 + \text{dy}^2 + \text{dz}^2$);	$\mathcal{E}_5 := (\{\text{dx, dy, dz}\}, \emptyset)$
	end define dist;	

Now, $\mathcal{E}(\textit{length})$ is given as $\mathcal{E}_1$, and $\mathcal{E}(\textit{dist})$ is given as $\mathcal{E}_2 \circ \mathcal{E}_3 \circ \mathcal{E}_4 \circ \mathcal{E}_5$:

$$\mathcal{E}_2 \circ \mathcal{E}_3 \circ \mathcal{E}_4 \circ \mathcal{E}_5 = (\{\textbf{self}_v.\text{X}, \textbf{self}_v.\text{Y}, \textbf{self}_v.\text{Z}, \text{vertex2.X}, \text{vertex2.Y}, \text{vertex2.Z}\}, \ldots)$$

$$\mathcal{E}_1 = (\{\textbf{self}_v.\text{X}, \textbf{self}_v.\text{Y}, \textbf{self}_v.\text{Z}, \text{vertex2.X}, \text{vertex2.Y}, \text{vertex2.Z}\} \circ \{\text{vertex2} \rightarrow \textbf{self}_c.\text{V2}\} \circ \{\textbf{self}_v \rightarrow \textbf{self}_c.\text{V1}\}, \emptyset)$$

$$= (\{\textbf{self}_c.\text{V1.X}, \textbf{self}_c.\text{V1.Y}, \textbf{self}_c.\text{V1.Z}, \textbf{self}_c.\text{V2.X}, \textbf{self}_c.\text{V2.Y}, \textbf{self}_c.\text{V2.Z}\}, \emptyset)$$

◇