Function Materialization in Object Bases

Alfons Kemper *Christoph Kilger* *Guido Moerkotte*

Universität Karlsruhe
Fakultät für Informatik
D-7500 Karlsruhe, F. R. G.
Netmail: [kemper|kilger|moer]@ira.uka.de

Abstract

We describe *function materialization* as an optimization concept in object-oriented databases. Exploiting the object-oriented paradigm—namely *classification*, *object identity*, and *encapsulation*—facilitates a rather easy incorporation of function materialization into (existing) object-oriented systems. Furthermore, the exploitation of encapsulation (information hiding) and object identity provides for additional performance tuning measures which drastically decrease the rematerialization overhead incurred by updates in the object base. The paper concludes with a quantitative analysis of function materialization based on a sample performance benchmark obtained from our experimental object base system GOM.

1 Introduction

Once the initial "hype" associated with 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. Of course, the large body of knowledge of optimization techniques that was gathered over the last 15 years in, e.g., the relational area provides a good starting point. But lastly, only those optimization techniques that are specifically tailored for the object-oriented model 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 [12]: 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* [10, 11], which constitute materializations of heavily traversed path expressions that relate objects along attribute chains.

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 upon update operations. In this respect function materialization is related to relational view materialization [1, 3, 4]. Analogous work exists for the POSTGRES data model to optimize the evaluation of queries accessing (virtual) POSTQUEL attributes, e.g., [6, 7, 8, 13, 14, 15].

The above cited work is similar to ours with respect to the general idea of precomputing (caching) results However, the exploitation of object-oriented features, especially the classification of objects into *types*, *object identity*, and the principle of *encapsulation* facilitates a much finer-grained control over the rematerialization requirements of precomputed results in our approach than is possible in relational view materialization and extended relational caching: First, 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. Second, 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. Third, 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 transformation that could possibly invalidate a precomputed *volume* result while *rotate* and

translate leave the materialized *volume* invariant.

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

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 concepts even into existing object base systems: Only very few system modules have to be modified while the kernel system remains largely unchanged.

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 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 provide a (first) quantitative analysis of function materialization based on some simple benchmarks derived from computer geometry applications. Section 7 concludes this paper.

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. 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 following sample type definition introduces a new tuple structured type *Vertex*:

```
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;
        ...
end type Vertex;
```

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* to write the attribute are implicitly provided.[2] It is the type designer's choice whether these operations are made public by including them in the **public**-clause.

```
type Cuboid supertype ANY is
  public V1, set_V1, ..., V8, set_V8, ... length,
         width, height, volume, weight, rotate, scale, ...
  body [V1,...,V8: Vertex; Mat: Material; Value: float;]
  operations
    declare length: — float;
    declare width: — float;
    declare height: — float;
    declare volume: — float;
    declare weight: — float;
    declare translate: Vertex — void;
    declare scale: Vertex — void;
    declare rotate: char, float — void;
    declare distance: Robot — float;
  implementation
    define length is
      return self.V1.dist(self.V2);
    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);
        ...
        self.V8.translate(t);
      end define translate;
    ...
end type Cuboid;
```

Figure 1: Skeleton of the Type Definition *Cuboid*

Figure 1 shows the definition of the type *Cuboid* which serves as the running example throughout the remainder of this paper. In this definition, we have intentionally made all parts of the structure of a *Cuboid* visible (**public**), e.g., all boundary *Vertex* objects $V1, \ldots V8$ are accessible and directly modifiable. This is needed to demonstrate our function materialization approach in its full generality. 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 definition of *Cuboid* makes use of two auxiliary types *Material* and *Robot* that are not outlined here. A sample database is shown in Figure 2. The object identifiers are denoted by id_1, id_2, etc.

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

[2] Actually, in GOM a more elegant mechanism to realize value returning and value receiving operations is provided (see [12]).

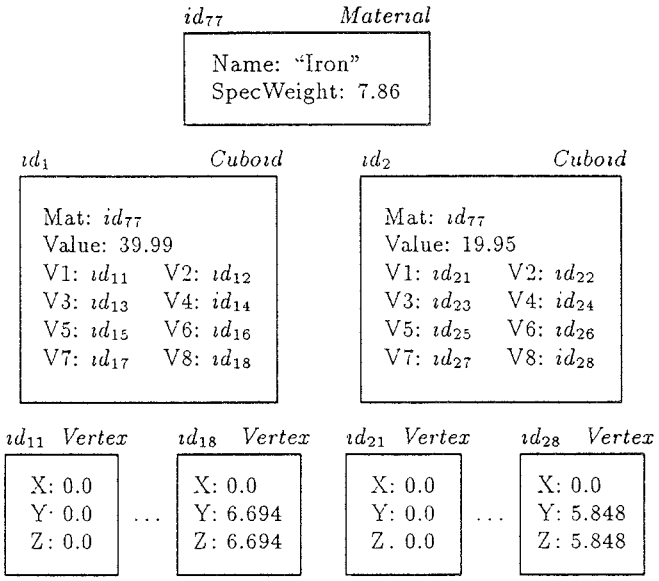


Figure 2: Database Extension of Example Schema

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
```

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

The evaluation of queries which reference the *volume* and/or the *weight* of *Cuboid*-instances can exploit the precomputed results instead of invoking the functions *volume* or *weight*, respectively.

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:

First, if the materialized function $f: t_1, \ldots, t_n \rightarrow t'$ has more than one argument 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.

Second, clustering of (precomputed) 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 [8] where separate caching of precomputed POSTQUEL attributes proved to be almost always superior to caching within the tuples.

If several functions are materialized which share all argument types the results of these functions may be stored in the same data structure. This provides for more efficiency when evaluating queries accessing results of several functions and avoids to store arguments redundantly.

These thoughts lead us to the following definition:

Definition 3.1 (Generalized Materializat. Relat.)

Let $t_1, \ldots, t_n, t'_1, \ldots, t'_m$ be types and $f_1, \ldots, f_m$ side-effect free functions with $f_j : t_1, \ldots, t_n \rightarrow t'_j$ for $1 \leq j \leq m$. 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:

$[O_1: t_1, \ldots, O_n: t_n, f_1: t'_1, V_1: bool, \ldots, f_m: t'_m, V_m: bool]$ □

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 type is complex, references to the result objects. The attributes $V_1, \ldots, V_m$ (standing for *validity*) indicate whether the stored results are currently valid. In this paper we restrict our discussion to functions having complex argument types and atomic result types. However, our concepts scale up to arbitrary functions as detailed in [9].

For each tuple τ of a GMR extension $\langle\langle f_1, \ldots, f_m \rangle\rangle$ over types $t_1, \ldots, t_n$ the following condition must hold:

$$\pi_{O_1, \ldots, O_n}\, \langle\langle f_1, \ldots, f_m \rangle\rangle = ext(t_1) \times \ldots \times ext(t_n) \;\wedge$$
$$\tau.V_j = true \;\Rightarrow\; \tau.f_j = f_j(\tau.O_1, \ldots, \tau.O_n)$$

Thus, upon an update to a database object that invalidates a materialized function result we have two choices.
Immediate rematerialization: The invalidated function result is immediately recomputed as soon as the invalidation occurs.
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.

Example: Consider the database extension 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*

◇

4 Dynamic Aspects of Function Materialization

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

The modification of an object is reported to the GMR manager. Then, the GMR manager invalidates or rematerializes all results affected by the update. Therefore, the GMR manager maintains the *Reverse Reference Relation* (*RRR*), which contains tuples of the 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 to 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.

Definition 4.1 (Reverse Reference Relation)
The Reverse Reference Relation RRR is a set of tuples of the form [*O*: *OID*, *F*: *FunctionId*, *A*: ⟨*OID*⟩].

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 entries 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.

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*	$\langle id_1, id_5 \rangle$
⋮	⋮	⋮
id_5	*distance*	$\langle id_1, id_5 \rangle$
id_5	*distance*	$\langle id_2, id_5 \rangle$
⋮	⋮	⋮
id_{11}	*volume*	$\langle id_1 \rangle$
id_{11}	*weight*	$\langle id_1 \rangle$
id_{11}	*distance*	$\langle id_1, id_4 \rangle$
⋮	⋮	⋮
id_{77}	*weight*	$\langle id_1 \rangle$
id_{77}	*weight*	$\langle id_2 \rangle$

$\langle\langle volume, weight \rangle\rangle$				
O_1	*volume*	V_1	*weight*	V_2
id_1	300.0	*true*	2358.0	*true*
id_2	200.0	*true*	1572.0	*true*

$\langle\langle distance \rangle\rangle$			
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*

Figure 3: The Data Structures of the GMR Manager

Example: Consider the GMRs $\langle\langle volume, weight \rangle\rangle$ and $\langle\langle distance \rangle\rangle$ (this example is based on 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 statement *GMR_Manager.invalidate*(o).

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

lazy(o) ≡ **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. Upon 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) ≡

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. 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.

The easiest way to realize the notification of the GMR manager about updates is to modify all update operations such that every invocation of an update operation triggers the invocation of *GMR_Manager.invalidate*. Figure 4 in Section 5 shows the modification of the update operation *set_X*. Another possible approach, the adaptation of the object manager, is discussed in [9].

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 three 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:
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*.
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.
Exploitation of strict encapsulation: By strictly encapsulating the representation of objects used by a materialized function the number of update operations that need to 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.

Due to space limitations we will restrict the discussion here to materialized functions that access only tuple-structured types. However, the concepts can easily be extended to set- and list-structured types (see [9]).

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 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:

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

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 given in

[9]. 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 $t.A \in RelAttr(f)$.[3] 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*

$$SchemaDepFct(t.set\_A) = \{f | f \text{ is a material. function and } t.A \in RelAttr(f)\}$$ □

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 $o.set\_A(\ldots)$ of an object o of type t triggers the invocation of the GMR manager as follows:

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

$SchemaDepFct(t.set\_A)$ is inserted as a set-valued constant into the body of the modified update operation $t.set\_A$—thus, the expression $SchemaDepFct(t.set\_A)$ needs not be evaluated each time $o.set\_A(\ldots)$ is invoked. However, the materialization of a further function for which $t.A$ is relevant requires a recompilation of $t.set\_A$ with a modified set-valued constant.

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.[4] 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 $o.set\_A(\ldots)$ 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

```
id111.set_X(2.5);   !! Vertex id111 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 $o.ObjDepFct \cap SchemaDepFct(t.set\_A)$. 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.

Example: Recall the database extension shown in Figure 2. Suppose that the two GMRs $\langle\langle volume, weight \rangle\rangle$ and $\langle\langle distance \rangle\rangle$ were introduced.

Consider the invocation $id_{11}.set\_X(3.0)$ which modifies the X coordinate of Vertex id_{11}. Figure 4 shows the modification of the update operation $Vertex.set\_X$. The set of materialized functions that is dependent upon the update $id_{11}.set\_X(3.0)$ is then given by the intersection of the sets $SchemaDepFct(Vertex.set\_X)$ and $id_{11}.ObjDepFct$.

$$SchemaDepFct(Vertex.set\_X) = \{volume, weight, distance\}$$
$$id_{31}.ObjDepFct = \{volume, weight\}$$

In this case, the intersection coincides with the set $id_{31}.ObjDepFct$. However, in general this is not the case. ◇

[3] Of course, also *create* and *delete* operations on the argument types affect materialized results and, therefore, have to be modified (see [9]).

[4] 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.

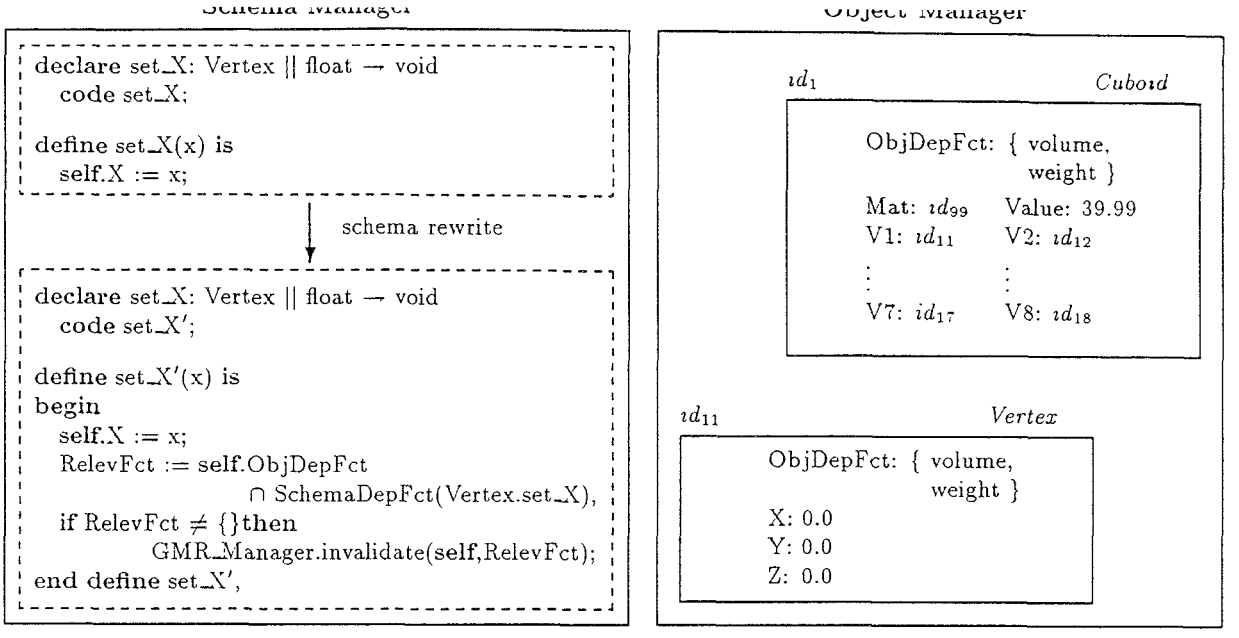


Figure 4: Interaction between Schema and Object Manager

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 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 (!) rematerializations 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

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

We assume that the set $InvalidatedFct(t.u)$ for each operation $t.u$ is determined by the database programmer. Then all update operations u for which $InvalidatedFct(t.u) \neq \{\}$, 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;                                    ◇
```

6 Benchmark Results

This section sketches the results of a first quantitative analysis of function materialization. The benchmarks were run on our experimental object base system GOM that is built on top of the EXODUS storage manager [5]. The analysis is based on the *Cuboid* example (see Section 2). All subsequent results were measured on a database containing 8000 *Cuboid* instances, each *Cuboid* referencing 8 *Vertex* instances and one *Material* instance. The database was stored on a 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 database is rather small we decided to use a correspondingly small database buffer of 600 kBytes to compensate for the small database volume.

6.1 Application Profile

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 *random* are randomly chosen and ε is a small constant:

$Q_{bw} \equiv$	$Q_{fw} \equiv$
range c: Cuboid	**range** c: Cuboid
retrieve c	**retrieve** c.volume
where c.volume = $r \pm \varepsilon$	**where** c.CuboidID[5] = random[6]

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*,[6] 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 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 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.

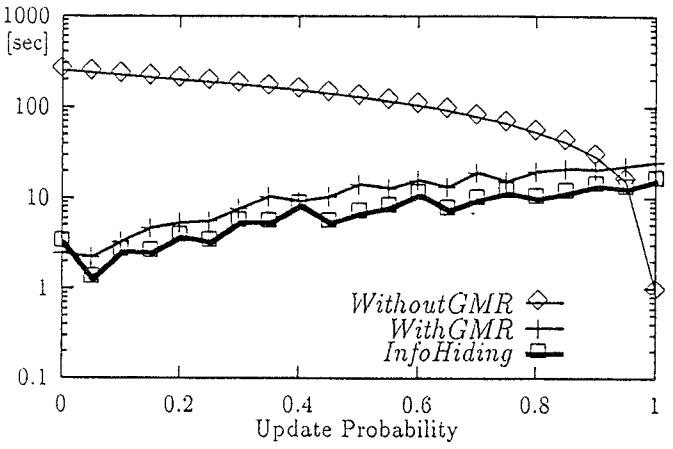
6.2 Results

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 5. 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: the GMR $\langle\langle volume \rangle\rangle$ is maintained under *immediate* rematerialization and utilized to evaluate the

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

queries.

InfoHiding: in this version the GMR ⟨⟨*volume*⟩⟩ is maintained under information hiding to reduce the invalidation and rematerialization overhead.



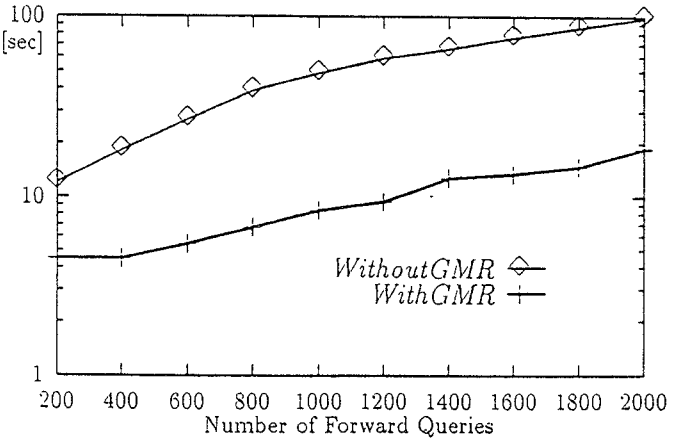
#Op	Q_{mix}	U_{mix}	P_{up}
40	$Q_{bw} = 0.5$ $Q_{fw} = 0.5$	$I = 0.5$ $S = 0.5$	0.0 ... 1.0

Figure 5: Varying Update Probabilities

From the plot in Figure 5 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 $P_{up} = 0.95$.

From this first benchmark 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 6. We observe that the exploitation of the GMR ⟨⟨*volume*⟩⟩ 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.

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 7. 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 ⟨⟨*volume*⟩⟩ 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 emtpy with respect to ⟨⟨*volume*⟩⟩. 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 7 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.



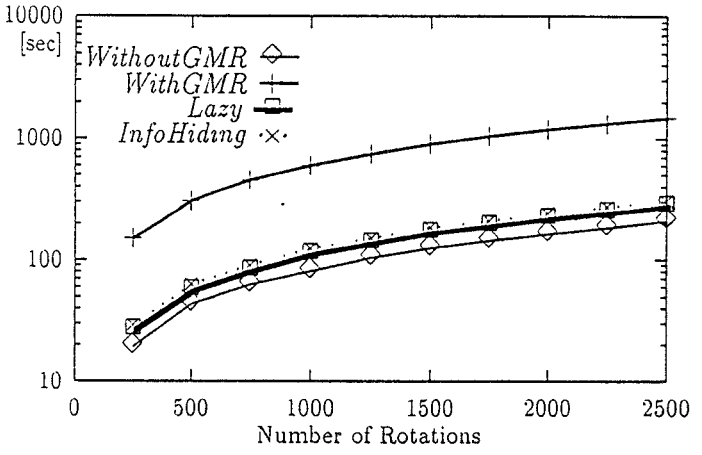
#Op	Q_{mix}	U_{mix}	P_{up}
200 ... 2000	$Q_{fw} = 1.0$	—	0.0

Figure 6: Cost of Forward Queries

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.

7 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



#Op	Q_{mix}	U_{mix}	P_{up}
250 ... 2500	—	$R = 1.0$	1.0

Figure 7: Invalidation Overhead

is largely based on rewriting the schema. 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. 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. 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 (reported on in this paper) and one from a more traditional administrative application (see [9]) 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 [10] 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.

References

[1] M. E. Adiba and B. G. Lindsay. Database snapshots. In *Proc. of The Conf. on Very Large Data Bases*, 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. of the DOOD Conf.*, 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 TODS*, 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.*, pages 61–71, Washington, D.C., 1986.

[5] M. Carey et al. Objects and file management in the EXODUS extensible database system. In *Proc. of The Conf. on Very Large Data Bases*, Kyoto, Japan, Aug 86.

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

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

[8] 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*, pages 88–99, L.A., CA, Sep 1988.

[9] A. Kemper, C. Kilger, and G. Moerkotte. Materialization of functions in object bases. Technical Report 28/90, Fakultät für Informatik, Universität Karlsruhe, D-7500 Karlsruhe, Nov. 1990.

[10] 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*, pages 290–301, Brisbane, Australia, Aug 1990.

[11] A. Kemper and G. Moerkotte. Access support in object bases. In *Proc. of the ACM SIGMOD Conf.*, pages 364–374, Atlantic City, NJ, May 90.

[12] A. Kemper, G. Moerkotte, H.-D. Walter, and A. Zachmann. GOM: a strongly typed, persistent object model with polymorphism. In *Proc. of BTW*, pages 198–217, Kaiserslautern, Mar 1991. Springer-Verlag.

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

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

[15] 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.*, pages 281–290, Atlantic City, NJ, May 90.