Prototype System for Method Materialisation and Maintenance in Object–Oriented Databases

Juliusz Jezierski  
Poznań University of Technology  
Institute of Computing Science  
Piotrowo 3A, 60-965 Poznań, Poland  
+48 61 6652127  
Juliusz.Jezierski@cs.put.poznan.pl

Mariusz Masewicz  
Poznań University of Technology  
Institute of Computing Science  
Piotrowo 3A, 60-965 Poznań, Poland  
+48 61 6652808  
Mariusz.Masewicz@cs.put.poznan.pl

Robert Wrembel  
Poznań University of Technology  
Institute of Computing Science  
Piotrowo 3A, 60-965 Poznań, Poland  
+48 61 6652529  
Robert.Wrembel@cs.put.poznan.pl

ABSTRACT
The efficient execution of a method has a great impact on a system response time. Optimising access to data returned by methods is difficult as methods are written in high–level programming languages. Moreover, estimating a method’s execution cost is another serious problem because of the complexity of a method’s code. A promising technique to tackle the problem of optimising execution of methods is based on method materialisation. Within our project we developed, so called, hierarchical method materialisation technique. In this paper we present a prototype system for hierarchical materialisation of methods and for the management of their results in an object–oriented database.

Categories and Subject Descriptors
H.2. [Database Management]: Physical Design – access methods.

General Terms

Keywords
object–oriented database, method materialisation, storage structure.

1. INTRODUCTION
Applications for object–oriented databases can be written in a procedural language or/and in an object query language (OQL), and they usually call object methods. A method can be a very complex program, whose computation may last long and therefore the efficient execution of a method has a great impact on a query response time. Optimising access to data returned by methods is difficult as methods are written in high–level programming languages. Moreover, the estimation of methods execution costs is also a problem. A promising technique, called method materialisation (precomputation) may be used for reducing access time to data. The materialisation of a method consists in: (1) computing the result of a method once, (2) store it persistently in a database, and then (3) use the persistent value when the method is invoked, rather than computing the result every time the method is invoked. Not all methods, however, are good candidates for materialisation. Some examples of such methods are as follows: methods comparing or looking for similarities between two pieces of information, methods having many input arguments with large domains. On the one hand, the materialisation of a method reduces the time necessary to access the method's result. But on the other hand, when the result of a method has been materialised it has to be kept up to date when data used for computing this result change.

The areas of applying methods materialisation include: (1) typical object–oriented databases with stored complex methods, (2) object–relational data warehouse systems [6, 5], (3) materialised object–oriented views [10, 3, 1].

1.1 Related Work
Method materialisation was proposed in [8, 2, 9] in the context of indexing techniques and query optimisation. The work of [8] sets up the analytical framework for estimating costs of caching complex objects. In the approach of [2], the results of materialised methods are stored in an index structure based on B–tree, called method–index. The application of method materialisation proposed in [2] is limited to methods that: (1) do not have input arguments, (2) use only atomic type attributes to compute their values, and (3) do not modify values of objects. Otherwise, a method is left non–materialised. The concept of [9] uses the so called Reverse Reference Relation, which contains the tuples that store an information about a method being materialised, object used to compute the value of a method and its input argument values. In order to maintain materialised methods, every object has appended the set those method identifiers that used the object. One drawback of this solution is that, the set of method identifiers that must be appended to every object impacts design phase of the

1 This work is partially supported by the grant no. 4 T11C 019 23 from the Polish State Committee for Scientific Research (KBN), Poland.
2. HIERARCHICAL MATERIALISATION OF METHODS

The idea behind the hierarchical materialisation [7] is as follows. When hierarchical materialisation is applied to method \( m_i \), then the result of \( m_i \) is stored persistently and additionally, the results of all methods called from \( m_i \) are also stored persistently. Having selected method \( m_i \) for the materialisation, the result of the first invocation of \( m_i \) for a given object \( o_i \) is stored persistently. Each subsequent invocation of \( m_i \) for the same object \( o_i \) uses the already materialised value. When an object \( o_i \) used to materialise the result of method \( m_i \), \( m_i \) is updated or deleted, then \( m_i \) has to be recomputed. This recomputation can use unaffected intermediate materialised results, thus reducing the recomputation time overhead.

Hierarchical materialisation technique was developed for reducing access time to complex objects (via methods) composed of many component objects stored on a disk.

**Example 1.** In order to illustrate the idea behind the *hierarchical materialisation* let us consider a simple database describing geometric figures, as shown in Figure 1. Every complex figure is composed of triangles and circles. Triangles, in turn, are composed of segments. A segment connects two points. The Point class has methods \( \text{getX} \) and \( \text{getY} \) that, for a given input scale value, return values of X and Y coordinates of a point. The \( \text{length} \) method in the Segment class returns a length of a segment in a given scale. The \( \text{area} \) methods in the Triangle as well as in the Circle class return an area of a triangle and circle in a given scale, respectively. Finally, Figure/area returns an area of the whole figure in a given scale including areas of its triangles and circles. In order to compute this value, Figure/area calls Triangle/area and Circle/area. Triangle/area calls Segment/length, which in turn, calls Point/getX and Point/getY.

Figure 2 presents collaboration diagram between the instances of classes from Figure 1. Let us assume that object identified by \( c_f \) is composed of objects \( t_f \) and \( c_f \). \( t_f \) is further composed of: \( s_1, s_2, \) and \( s_3, s_4 \) is composed of two points: \( p_1 \) and \( p_2 \). Similarly, \( s_2 \) is composed of \( p_2 \) and \( p_3 \), whereas \( s_3 \) is composed of \( p_1 \) and \( p_3 \). Let us further assume that the \( \text{area} \) method in Complex Figure was marked as materialised.

![Collaboration Diagram](image)

Let us assume that the \( \text{area} \) method was invoked for \( c_f \) with the input argument, i.e., scale value of 5. In our example, the hierarchical materialisation mechanism results in materialising values of the following methods: Triangle/\( \text{area}(5) \) for \( t_f \); Segment/length(5) for \( s_1, s_2, \) and \( s_3 \); Point/\( \text{getX}(5) \) and Point/\( \text{getY}(5) \) for \( p_1, p_2, p_3, p_4 \). Circle/\( \text{area}(5) \) for \( c_f \).

3. DATA STRUCTURES

In order to materialise methods in a class and maintain the materialised results, four additional data dictionary structures have been developed. These structures, which are outlined below, are called Materialised Methods Dictionary, Materialised Method Results Structure, Graph of Method Calls, and Inverse References Index [7].

3.1 Materialised Methods Dictionary

*Materialised Methods Dictionary (MMD)* makes available the data dictionary information about all methods. It stores among others the following information about every defined method: the name of a method, the name of a class the method was defined in, the type of a value returned by a method, the implementation of a method, a flag indicating whether a method was set as...
materialised or not, the array of sensitive attributes for a method, the array of input arguments of a method.

3.2 Materialised Method Results Structure
Materialised Method Results Structure (MMRS) stores the following information about every materialised method: the identifier of a method, an object identifier of the object which the method was invoked for, the array of input argument values a method was invoked with, the value returned by a method while executed for a given object and for a given array of input argument values.

When method \( m_i \) is invoked for a given object \( o_i \) and this method has been previously set as materialised, then \( \text{MMRS} \) is searched in order to get the result of \( m_i \) invoked for \( o_i \). If it is not found then, \( m_i \) is computed for \( o_i \) and stored in \( \text{MMRS} \). Otherwise, the materialised result of \( m_i \) is read instead of executing \( m_i \). When an object used to compute the materialised value of \( m_i \) is updated or deleted, then the materialised value becomes invalid and is removed from \( \text{MMRS} \).

In order to be able to apply the method materialisation technique to method \( m_i \) the original code of \( m_i \) has to be extended with a section that checks if appropriate results of \( m_i \) have been materialised.

3.3 Graph of Method Calls
A method defined in one class can invoke other methods defined in other classes. The chain of method dependencies, where one method calls another, is called Graph of Method Calls (GMC). GMC is used by the procedure that maintains the materialised results of methods. When materialised method \( m_i \) becomes invalid all the materialised methods that use the value of \( m_i \) also become invalid. In order to invalidate those methods the content of GMC is used. GMC stores pairs of values: the identifier of a calling method and the identifier of a method being called.

3.4 Inverse References Index
When a sensitive attribute value of an object changes, and if that object was used to compute a value of materialised method \( m_i \), then \( m_i \) and all methods in GMC that used \( m_i \) have to be invalidated. For example, if object \( c_f \) (cf. Figure 2) changes its radius, then the value of \( \text{cf}. \text{area()} \) becomes invalid. In order to invalidate dependent methods the system is able to find inverse references in object composition hierarchy. In order to ease the traversal of a composition hierarchy in an inverse direction we use so called inverse references for each object. An inverse reference for object \( o_j \) is the reference from \( o_i \) to other objects that reference \( o_j \). For example, the inverse reference for object \( p_2 \) (cf. Figure 2) contains two object identifiers \( s_1 \) and \( s_2 \).

The Inverse References Index (IRI) has the following features:

- In case of modifying a reference from object \( o_i \) to object \( o_j \) only one object - \( o_i \) has to be locked, whereas the appropriate inverse reference from \( o_j \) to \( o_i \) is modified in IRI.

At the implementation level, a separated IRI is created for every, but a root class in the composition hierarchy.

3.5 Maintenance of Materialised Methods
When the materialised value of method \( m_i \) becomes obsolete it is removed from an appropriate MMRS. This removal causes that the results of methods that called \( m_i \) also become invalid and have to be removed from MMRS. The removal of materialised results from MMRS is recursively executed up to the root of GMC by a dedicated procedure. The procedure traverses GMC and aggregation relationships in an inverse direction, i.e. from the bottom to the top. The procedure has two input arguments: the identifier of a method being invalidated and the object identifier for which the method was materialized. The number of recursive calls to this procedure depends on the number of levels in GMC. Let \( h \) be the number of levels including leaves and the root. The number of times the MMRS_Propagate_Remove_Result procedure is executed is expressed by the following formula:

\[ h^{h-1} \times (n_{CM} \times n_{VO}) \]

where \( n_{CM} \) is the number of methods calling a given method at a given level of GMC, and \( n_{VO} \) is the number of objects in inverse relationships at a given level of composition hierarchy.

4. PROTOTYPE SYSTEM
The proposed hierarchical materialisation technique is currently being developed. The prototype system is being implemented in Java, on top of the FastObjects 17 object-oriented database system. The concept of hierarchical materialisation was successfully evaluated by experiments and their results are available in [7]. Whereas this section overview the functionality of our prototype, that includes:

- the management of materialised method results (MMRS) and indexing of these results (IRI),
- the automatic construction the GMC and the automatic extraction of sensitive attributes,
- a graphical tool for the management and visualisation of the data dictionary content.

4.1 Tool for automatic construction of GMC
The screen shot of the application for the automatic construction of the GMC is shown in Figure 3. The application parses codes of compiled Java classes in order to find dependencies between calling and called methods.

The results of parsing are stored in the GMC. In Figure 3, the list named “Choose classes” allows to select one or more classes for parsing. Scrolled lists named “Fields” and “Methods” present the structure and behaviour of a selected class, respectively. List called “method info” shows signatures of methods being called from a selected method in the “Methods” list. The presented parsing tool also allows to find sensitive attributes (cf. Section 3). In the current implementation, an attribute of a class is marked as sensitive if it appears on the right hand side of arithmetic expressions or if its value is passed to a method as an input value.
In the current implementation, the automatic construction of the **GMC** is limited to Java programs only.

### 4.2 Tool for data dictionary management and visualisation

Method parsing results may be stored in a database, in the data dictionary **MMD** and **GMC**. Their content as well as the content of the **MMRS** are visualised and managed by another tool, shown in Figure 6. The left hand side panel of this tool contains the list of all methods stored in the **MMD**. A selected method is then fully described in the right hand side panel. This panel displays the method’s input arguments (“Arguments”), the information about methods calling the selected method (“Calling by”) as well as methods being called from it, the set of sensitive attributes, the content of the method’s body, and materialised results of this method.

### 4.3 Experimental results

In this section we are presenting the experimental results that comprise measuring time overhead for the management of materialised methods under updates to objects. In the experiments we used the database of geometric figures composed of 100 instances of the **Complex_Figure** class. Every **Complex_Figure** object was composed of 10 triangles. Every triangle was composed of 3 segments, each of which was composed of 2 points. The size of objects in the experiments equalled to 300kB, resulting in the total size of our test database of 2.1GB. The experiments were run on a PC with 1GHz processor, 256MB of RAM.

The structure of **GMC** used in the experiment is as follows. Method $F_{1}.area$ calls $T_{1}.area$, $T_{2}.area$, ..., $T_{10}.area$. Each area method of a given triangle calls method $length$ of its segments. Each segment calls methods $getX$ and $getY$ of its points.
materialization reduces method computation time provided that invalidation of less than 6 branches in a complex figure takes place. When the number of invalidated branches is 6 and more the hierarchical materialization slows down method executions.

![Fig. 5. Maintenance time characteristics of materialised methods for objects of size 300kB](image)

We have run the experiment for objects of 13 different sizes ranging from 100B to 300kB. In each case, the crossing point between the two lines was between 50% and 60%.

Notice that the rematerialisation and maintenance times of materialised methods depend on: (1) the "shape" of GMC, (2) time spent on executing non-materialised methods, i.e. the computation complexity of a method being materialised, and (3) the number of GMC branches being invalidated. The invalidation and rematerialisation time overhead linearly increases with the number of branches that must be invalidated.

5. SUMMARY, CONCLUSIONS, AND FUTURE WORK

Materialisation of methods in object-oriented databases is a promising technique increasing system's performance. In this paper we presented the prototype system for the management of materialised methods. The prototype, implemented in Java, allows to automatically construct the graph of method calls, extract sensitive attributes, visualise and manage data dictionary information, and manage materialised results of methods.

The current implementation of hierarchical materialisation and the prototype has a few following limitations.

1. A method for the materialisation is explicitly selected by a database administrator/designer during a system tuning activity.

2. The body of method $m_i$ being materialised may not contain OQL commands as it would cause difficulties in registering the used object identifiers and values of the method in MMRS.

3. Materialised methods must not modify the content of a database.

Currently we are working on:

1. Incorporating into our prototype the support for methods having objects as input arguments and returned values.

2. Analysis of method bodies in order to allow materialisation of methods using OQL statements in their bodies.

Future work will concentrate on the development of a technique that will allow to select automatically or semi–automatically the right method for materialisation. To this end, a cost model describing the complexity of a method needs to be developed.

6. REFERENCES


