The Update of Index Structures in Object-Oriented DBMS

Andreas Henrich
Universität Siegen, Praktische Informatik, D-57068 Siegen, Germany
henrich@informatik.uni-siegen.de
http://www.informatik.uni-siegen.de/pi/user/henrich

Abstract

Index structures in object-oriented database management systems should support selections not only with respect to physical object attributes, but also with respect to derived attributes. A simple example arises, if we assume the object types *Company*, *Division*, and *Employee*, with the relationships *has_division* from *Company* to *Division*, and *employs* from *Division* to *Employee*. In this case the index structure should allow to support queries for companies specifying the number of employees of the company.

Unfortunately, there is one main problem with index structures addressing derived attributes, namely the question: Which entries in which index structures have to be updated after a certain update in the object base? In the example above, the creation of a new employee must trigger an index update for the corresponding company. In the present paper we propose a practical solution for this problem. This solution is based on so-called *index update definitions* which comprise (1) an *event description* for the event causing the need for the index update, (2) a *reference* for the affected index structure, (3) a *query* determining the elements for which the respective index entries have to be updated, and (4) the corresponding *update operation*.

Using these *index update definitions* we can handle complex derived attributes defined e.g. employing regular path expressions. The paper first describes the environment for our considerations originating from the fields of software engineering environments and information retrieval. Thereafter the use of our approach is demonstrated by the help of a comprehensive example. Finally we sketch the implementation of our approach based on a multi-thread architecture.

1 Introduction

Index structures supporting selections or similarity queries are an indispensable part of modern object-oriented database management systems (ooDBMS). Without such index structures query languages such as OQL would be useless for large object bases, because the query processing times would be unacceptablely high. As long as the selection predicates address direct object attributes, the situation remains very much the same as with relational databases, and hence, does not cause major problems. However, in query languages for ooDBMS selection predicates can as well address properties which may be characterized as *derived attributes*. Such derived attributes can be defined in different ways:

1. Query language inherent operators can be used to calculate derived attributes. Let us for example assume a schema with an object type *Company*, an object type *Division*, and an object type *Employee*. Furthermore let us assume a relationship *has_division* from *Company* to *Division* and a relationship *employs* from *Division* to *Employee*. In this case we can select companies with more than 1000 employees with a query in an OQL-oriented syntax as follows:

   ```
   select x.name
   from Company x
   where 1000 < count(x.has_division.employs)
   ```

2. Another way to calculate derived attributes is to employ user-defined methods. Assume that we have defined a method *calc_liquidity* for companies. In this case we could ask for companies with a liquidity of at least 1.000.000 US$ using the following query:

   ```
   select x.name
   from Company x
   where 1000000 <= x.calc_liquidity
   ```

To efficiently support selections addressing derived attributes, it should be possible, to define index structures for derived attributes in the same way as for physical attributes. At first glance, this might seem a simple extension. On the other hand, the maintenance of the index structure becomes much more difficult with this extension. To understand the problems arising from derived attributes with the maintenance of the index structure, let us consider the examples given above. If we built an index structure for companies addressing the derived attribute `'count(x.has_division.employs)'`, the entry for a company in this index structure has to be updated whenever an employee working for a division of the company is created in or removed from the object base. As a consequence, we have to introduce some type of meta information for the object type *Employee* which defines the index update operation which has to be performed for the corresponding company whenever an *Employee* is created or removed.

The situation becomes even more complicated when user-defined methods are applied. As with our example method *calc_liquidity* such a method will usually gather some information from related objects and condense this information. The origin of the base information is hidden in the method. However, if we want to address derived attributes calculated by user-defined methods in an index structure, we need detailed information about the origin of the input data of the method in order to update the corresponding index entry when part of the input data is changed.

In the present paper we describe our solution for the depicted problem. This solution has been developed in the context of a project concerned with the implementatiuon of a specialized ooDBMS for software engineering environments. With respect to the index update problem this ooDBMS comprises three main components: (1) H-PCTE, an implementation of PCTE, the ISO and ECMA standard for a public tool interface for an open repository. This component is responsible for object creation, object update operations and the navigational access to the object base. (2) P-OQL, an OQL-oriented query language for PCTE, which is responsible for the set-oriented declarative access to the objects. Furthermore this query language includes content-based information retrieval facilities. (3) Index structures for standard attributes and document description vectors. The aspects of this environment which are essential for this paper will be described in more detail in section 2.

The basic principle behind our approach is to extend the meta information for the object types by so called *index update definitions* (IUDs). For each object type there is one IUD for each index structure which might be affected by the creation, deletion or update of an object of this type. Roughly spoken an IUD contains the information necessary to perform the update of the index structure. More precisely each IUD is a quadruple with the following components: (1) A description of the event, which causes the necessity of the index update. (2) A reference to the (potentially) affected index structure, for which one or more entries have to be updated. (3) A query determining the instances for which the index entries have to be updated, and (4) the necessary index actualication (creation, deletion or update of the entry). The IUDs for an index structure can be automatically created when the index structure is defined. They allow for an efficient processing of update operations in the database.

It has to be mentioned that our work has been influenced by approaches in the areas of view maintenance and active databases. The *view maintenance problem* is concerned with the determination of changes to a view based on changes to the base relations. Various approaches dealing with different view definition languages, different amounts of available information for the view maintenance, and different types of modifications on the base relations have been published (see e.g. [GM95, GL93, GMS93]). In the area of active databases [DHW95, DGG95] so-called event-condition-action rules are applied to react when a certain event occurs. In relation to the mentioned approaches the main contribution of the present paper is threefold: (1) We transfer and adapt the basic ideas from both fields (view maintenance and active databases) to the index update problem. (2) We extend the approaches to complex regular path expresions, which can be employed in our approach to define derived attributes. (3) We describe our approach for a concrete application scenario and sketch the implementation of the approach.

The rest of the paper is organized as follows: In section 2 we explain the aspects of PCTE and P-OQL which are essential for this paper. Thereafter the proposed index update definitions are described (cf. section 3). Since the PCTE data model covers only structural aspects, we will first focus on these aspects. Thereafter in section 4 we will sketch how our approach can be extended to cover user-defined methods as well. Section 5 describes the implementation of our approach. Finally section 6 concludes the paper.

2 Example Environment

2.1 PCTE

As mentioned PCTE (*Portable Common Tool Environment*) is the ISO and ECMA standard for a public tool interface for an open repository [PCT93, PCT94, WJ93]. As one of its major components PCTE contains a structurally object-oriented object management system (OMS).

H-PCTE [Kel92] is a main-memory-oriented high-performance implementation of the OMS of PCTE especially designed to meet the performance requirements arising from fine-grained data modeling.

The data model of the PCTE OMS can be seen as an extension of the binary Entity-Relationship Model. The objectbase contains objects and relationships. Relationships are normally bi-directional. Each relationship is realized by a pair of directed links, which are reverse links of each other, i.e. point into opposite directions.

The type of an object is given by its name, a set of applied attribute types, and a set of allowed outgoing link types. New object types are defined by inheritance.

A link type is given by a name, an ordered set of key attribute types, a set of (non-key) attribute types, a set of allowed destination object types, and a category. PCTE offers five link categories: *composition* (defining the destination object as a component of the origin object), *existence* (keeping the destination object in existence), *reference* (assuring referential integrity and representing a property of the origin object), *implicit* (assuring referential integrity) and *designation* (without referential integrity).

Throughout this paper we will use the simplified schema for OOA-documents given in figure 1 as the basis for our examples. It consists of the object types *Document*, *OOA_Diagram*, *Class*, *Attribute* and *Method*. The attribute types applied to each object type are given in the ovals at the upper left corner of the rectangles representing the object types.

The link types are indicated by arrows. A double arrowhead at the end of a link indicates that the link has cardinality *many*. Links with cardinality many must have a key attribute. In the example the numeric attribute *no* is used for this purpose. For example the link type *contains* has such a key attribute and is hence described as *"no.contains"*. Therefore an instance of this link type can be addressed by its link name which consists of the concrete value for the key attribute and the type name separated by a dot – e.g. *"3.contains"*. A *'C'* or *'R'* in the triangles at the center of the line representing a pair of links, indicates that the link has category *composition* or *reference*.

Finally the schema contains a subtype relationship between the object types *Document* and *OOA_Diagram* which is indicated by a broad shaded arrow.

2.2 P-OQL

P-OQL [Hen95] is an OQL-oriented query language for PCTE, and OQL (*Object Query Language*) [Cat93] is the ODMG (*Object Database Management Group*) proposal for a query language for ooDBMS. The main differences between P-OQL and standard OQL are due to the adaptation

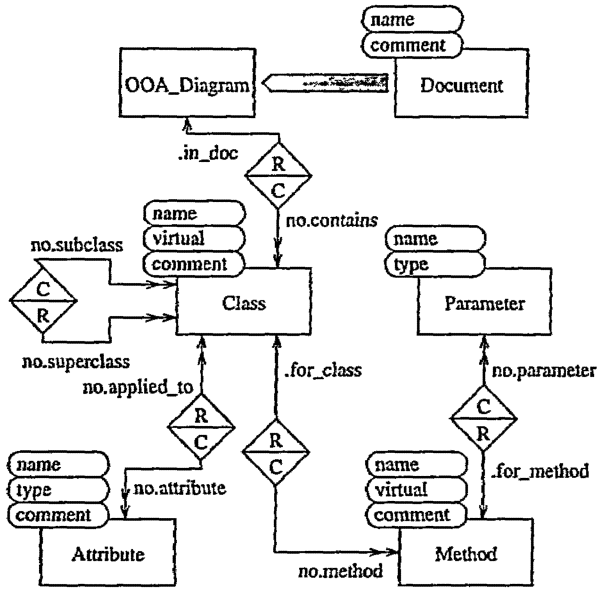

Figure 1: Example schema

to the data model of PCTE. Hence, especially the treatment of links is specific to P-OQL. Furthermore P-OQL includes information retrieval facilities [Hen96b].

A query in P-OQL is either a select-statement, or the application of an operator (like *sum*).

Assume that we search for the name and the applicable methods of all classes in the OOA-diagram named *"Clearing"*. The following query in P-OQL yields this information:

```
select C:name,
       normalize C:[_.superclass]*/_.method/->name
from D in OOA_Diagram,
     C in (D:_.contains/->.)
where D:name = "Clearing"
```

In the from-clause of this query two base sets are defined: Base set `D` addressing all OOA-diagrams in the objectbase and base set `C` addressing all objects which can be reached from the actual object of base set `D` via a path matching the regular path expression `'D:_.contains/->.'`. In this path expression the prefix `'D:'` means that the actual element of base set `D` is used as the starting point of the definition. `'_.contains'` means that exactly one link of type *contains* must be traversed. The underscore (`'_'`) is used as a wildcard for numerical key attributes denoting that arbitrary key values are allowed. `'/->'` is used to address the destination object of the path. In addition `'->'` can be used to address the last link of the path. The `'.'` at the end of the regular path expression means that the object under concern is addressed. It is also possible, to address an attribute – which is for example the case in the select-clause of the above query – or a tuple of values.

Due to the definition of an *independent* base set (base set `D`) and a *dependent* base set (base set `C`), each OOA-diagram is combined with each class, which can be reached from the object representing the OOA-diagram via a link of type *contains*.

Summarizing, a base set can be defined in P-OQL in five different ways: (1) giving an object type name or an object type name suffixed by a `'^'` meaning that objects of all subtypes are addressed as well; (2) using a link type name to address all links of a given type; (3) defining a set of objects or links using a regular path expression as with base set `C` in our example; (4) defining a set of objects or links via a sub-select; or (5) passing a set of objects or links via the API when submitting a query.

In the where-clause of our example query the considered combinations of OOA-diagrams and classes defined in the from-clause are restricted to those for which the *name* attribute of the OOA-diagram has the value *"Clearing"*. Besides such simple conditions P-OQL for example allows the use of quantifiers and subqueries in the where-clause.

The select-clause of the example query states that a multiset of pairs is requested. Each pair consists of the name of the class and a set containing the names of the methods which can be applied to instances of the class. Since methods defined for superclasses of the class under concern can also be applied to instances of the class, we have to address these methods as well. To this end, the regular path expression `'C:[_.superclass]*/_.method/->name'` is used in the query. The starting point of this expression is the actual element of base set `C`. The meaning of `'[_.superclass]*'` is that zero or more links matching the link definition `'_.superclass'` have to be traversed. Alternatively P-OQL knows the iteration facilities [*path_definition*]+ to indicate that a path matching *path_definition* has to be traversed at least once and [*path_definition*] to indicate that a path matching *path_definition* is optional [1]. After traversing an arbitrary number of *superclass* links, exactly one link of type *method* has to be traversed, to reach a method. This is stated in the query using the link definition `'_.method'`. Finally the *name* attribute of this method is addressed using `'/->name'`. Summarizing, the regular path expression `'C:[_.superclass]*/_.method/->name'` determines a multiset containing the names of all methods which can be reached from the class object under concern traversing an arbitrary number of *superclass* links and exactly one link of type *method*. This multiset is transformed into a set using the unary operator `normalize`.

In addition to the link definitions used in the above example, which have been based on a given link type name and a definition of the allowed values for the key attributes using wildcards or intervals, P-OQL allows the specification of a set of link categories instead, with the meaning that all links having one of the given categories fulfill this link definition. E.g. the expression `'[{c, e}]+/->.'` addresses all objects which can be reached via a path consisting only of links with category *composition* or *existence*. Furthermore, not only the category of the link itself, but also the category of its reverse link can be specified using `'@'` as a prefix for the category definition. This feature is for example useful if the *"document"*, containing the actual object as a component, has to be determined. Finally an additional shield-option can be used to prevent traversing a link of a given type or a link with a destination object of a given type. For example the link definition `'{c shield subclass}'` allows to traverse arbitrary *composition* links except those of type *subclass* and the link definition `'{c shield Class}'` allows to traverse all *composition* links except those which have a destination object of type *Class*.

As mentioned at the beginning of this section P-OQL contains additional information retrieval facilities to support content-based similarity searches. These facilities im-

[1] The termination of [...]* and [...]+ is guaranteed by a corresponding fix point semantics.

plement the vector space model [SWY75]. This model assumes that an available term set is used to identify both, maintained documents and information requests. Queries and documents are represented as *term vectors* of the form $D_i = (a_{i1}, a_{i2}, \ldots, a_{it})$ and $Q_j = (q_{j1}, q_{j2}, \ldots, q_{jt})$ where t is the number of the terms in the term set and where the coefficients a_{ik} and q_{jk} represent the relevance of document D_i or query Q_j, respectively, with respect to term k. In the literature various term-weighting formulas have been proposed to calculate the a_{ik} and q_{jk}. In P-OQL we employ the formulas presented in [SB88] which have been proved to be competitive e.g. in [Har95].

Let N denote the number of documents, n_k denote the number of documents containing term k, and tf_{ik} denote the *term frequency* of term k in document D_i [2]. Then the components of the document description vector $D_i = (a_{i1}, a_{i2}, \ldots, a_{it})$ can be calculated as follows:

$$a_{ik} = \frac{tf_{ik} \cdot \log \frac{N}{n_k}}{\sqrt{\sum_{l=1}^{t} \left(tf_{il} \cdot \log \frac{N}{n_l} \right)^2}} \qquad (1)$$

In this formula the term frequency tf_{ik} representing the relevance of document D_i with respect to term k is multiplied by the inverse collection frequency $\log \frac{N}{n_k}$ representing the selectivity of term k in the document collection. Finally the result is normalized to eliminate the influence of the length of the document.

If the query itself is given as a query text Q_j for which similar documents are searched, the following formula proposed in [SB88] can be used to calculate the query description vector $Q_j = (q_{j1}, q_{j2}, \ldots, q_{jt})$:

$$q_{jk} = \begin{cases} \left(0.5 + \frac{0.5 \cdot tf_{jk}}{\max\limits_{1 \le l \le t} tf_{jl}} \right) \cdot \log \frac{N}{n_k} & \text{if} \quad tf_{jk} > 0 \\ 0 & \text{if} \quad tf_{jk} = 0 \end{cases} \qquad (2)$$

Now the similarity between the query and a document in the object base can be calculated using for example the conventional vector product formula:

$$\text{similarity}(Q_j, D_i) = \sum_{k=1}^{t} a_{ik} \cdot q_{jk} \qquad (3)$$

To integrate these formulas into P-OQL, we use three operators. First there are the unary operators `D_vector` and `Q_vector` which determine a document or query description vector, respectively. The third operator is the binary operator `sim` applying formula (3) to document or query description vectors.

The operator `D_vector` can be applied to *document definitions*. A *document definition* can e.g. be (1) a string attribute, (2) an object (in this case all string attributes of the object are considered), or (3) a collection (set, multiset, list, or structure) of the before-mentioned entities. Such a collection can e.g. be defined by a regular path expression.

If we are for example looking for non-virtual class definitions similar to a given textual class description, it would be appropriate to consider each *Class* object together with the associated attribute and method definitions as a *"class definition document"*. This is done in the following query applying the `D_vector` operator to a set of objects determined by the regular path expression `'[{c shield subclass}]*/->.'`. The query determines the 10 classes most similar to the textual class description:

```
head[10]
 sort-
  (select (Q_vector "<textual class description>"
           sim
           D_vector D:[{c shield subclass}]*/->.),
          D:name
   from D in Document^
   where D:virtual = FALSE)
```

In this query the query text is given as a string constant. The `Q_vector` operator is applied to this query text to create the query description vector. The document description vectors are determined applying the `D_vector` operator to the *"class definition documents"* defined by the regular path expression `'D:[{c shield subclass}]*/->.'`. This means that the union of all string attributes of all objects contained in the set determined by this path expression is considered as the text of the *"class definition document"* of the class under concern.

The select-statement yields a multiset of pairs, where the first component contains the similarity value for the actual *"document"* in base set D, and the second component contains the value of the *name* attribute of the document. Since we use `Document^` as the base set for the documents, we are not only concerned with OOA-diagrams, but with all types of documents stored in the object base. The `sort-` operator is applied to the result of the select-statement yielding a list with the pairs sorted descendingly according to their similarity values. Then the head operator is used to extract the 10 elements at the beginning of this list.

With respect to our index update problem the operator `D_vector` is an extremely interesting example for a derived attribute. If we want to support a similarity search like the one described in the query above by an index structure, the index structure has to address the derived attribute `'D_vector D:[{c shield subclass}]*/->.'`. Obviously this derived attribute of a class changes every time, a string attribute of an attribute or a method of this class is changed and even when a string attribute of a parameter is changed.

3 Index Update Definitions

For a closer examination of the index update problem, we consider four example index structures defined for the object type *Class* in our example schema. For each index structure the considered *"attribute"* and the type of the index structure are specified in the following index definition:

```
Index structures for object type Class:
1. name                                        [B-tree]
2. count _.method->.                           [B-tree]
3. normalize _.method/_.parameter/->name       [B-tree]
4. D_vector [{c shield subclass}]*/->.         [LSD-tree]
```

The first index structure is a B-tree [Com79] for the names of the classes. The second index structure is a B-tree for the number of the methods of each class. This index structure could e.g. be used to support queries like

```
select name from Class where 3 < count _.method->.
```

The third index structure addresses the names of the parameters of the methods of each class. The expression `'normalize _.method/_.parameter/->name'` defines a set

[2] It has to be mentioned that the term frequency is usually calculated after performing stop word elimination and stemming.

containing these names. Since each class will usually have multiple associated parameter names, there will be as much entries for each class in this index structure as there are parameter names. With this index structure queries like

```
select name
from Class
where "element" in
        normalize _.method/_.parameter/->name
```

can be supported.

Finally the fourth index structure is an LSD-tree for document description vectors — *LSD* stands for *local split decision* in this context. This index structure presented in [Hen96a] supports similarity searches based on term vectors. It can e.g. be used to support the query described at the end of section 2.2 determining the 10 classes most similar to a given textual class description.

The question we want to consider now is: When do we have to update which index structure? Obviously the answer to this question is extremely simple for the first index structure. An entry for a class has to be created when the class itself is created, it has to be updated every time the value of the *name* attribute of the class is changed, and it has to be removed from the index structure when the class is deleted from the object base. Unfortunately the situation is much more complicated for the other index structures. For example, we have to change the entry of a class in index structure 4 when the value of a string attribute of a parameter of a method of this class is changed.

To control the required index update operations, we introduce so called *index update definitions* (IUDs). Every time a new index structure is defined, the object and link types are determined for which changes of instances of the type can cause the necessity for an update in the index structure. For each determined type one or more IUDs are created for the events which might cause the necessity for an index update. Each IUD contains the information needed to perform the update when necessary. To this end, an IUD comprises the following information:

1. *event description*

   Here the event causing the necessity of the index update operation defined in the three other components of the IUD is described. The following types of events have to be distinguished: For an object we have to consider the creation and the deletion. Furthermore we have to consider the change of the value of an associated attribute and the creation and the deletion of links originating from the object. The creation and the deletion of a link is considered here as an update of the origin object, because each outgoing link represents a property of the object itself. On the other hand, we consider the change of a link attribute as an event for the link, because the attribute value represents a property of the link itself. Hence, such an event has to be considered in an IUD for the corresponding link type.

2. *reference to the index structure*

   The index structure which has to be updated is defined in this component of the IUD.

3. *the affected instances*

   One problem with the update of an index structure is to determine the entries of the index structure which have to be changed. In our case the starting point for the determination of these entries must be the object or link X which is actually changed. To determine the indexed instances affected by the change of object or link X, we exploit the P-OQL facility to address objects or links passed via the API as a base set. We define a query based on such a *passed base set* which determines the objects for which the corresponding index entries have to be changed. Roughly spoken this query uses the changed object or link X as its starting point and inverts the path expression used to define the index structure. In the remainder of this section we will give various examples for such queries.

4. *the update operation*

   In this component of the IUD the update operation is defined. We distinguish the operations *CREATE* (= insert a new entry into the index structure), *DELETE* (= remove an entry from the index structure) and *UPDATE* (= update an entry in the index structure).

Using this information a concrete update operation in the object base can be performed as follows: First of all, the event components of the IUDs for the type of the changed object are considered to check if — and in case, which — index structures are affected by the update operation. Then the affected index structures are updated as follows:

1. For necessary deletions or changes in the index structure (defined in component 4 of the IUDs as *DELETE* or *UPDATE*), the objects for which index entries have to be deleted or changed are determined applying the query defined in the third component of the respective IUD. Then the old values for the indexed attributes of these objects are calculated and the corresponding index entries are determined by exact match queries. These entries are deleted from the index structures.
2. Now the intrinsic update operation in the object base can be performed.
3. For changes (cf. component 4 of the IUDs) thereafter new index entries are created for the objects, for which the corresponding index entries have been removed before. Furthermore necessary insertions into the index structures are performed now.

In the remainder of this section we demonstrate the creation and the use of the IUDs for our example index definitions given at the beginning of section 3.

- *Considerations for all four index structures*

  Since all four index structures are defined for the object type *Class* update operations are obviously necessary whenever a *Class* object is created or deleted.

  To guarantee these index updates when a class is created, the following IUD is maintained with the object type *Class* for all four index structures:

event:	creation of an object
structure:	*<id of the index structure>*
instances:	—
action:	*CREATE*

  In these IUDs there is no P-OQL query in the component *instances*, because there are no corresponding entries in the index structures before the creation of the object. Since the *action* is defined as *CREATE*, it is nevertheless clear that a new index entry has to be

inserted into the index structure for the created object itself.

To initiate corresponding index update operations for the deletion of classes as well, we additionally attach the following IUD to the meta information for the object type *Class* for all four index structures:

event:	deletion of an object
structure:	*<id of the index structure>*
instances:	`select . from PASSED_OBJECTS`
action:	*DELETE*

With these IUDs the trivial P-OQL query `select . from PASSED_OBJECTS` is used to determine the objects for which index entries have to be deleted. This query uses the keyword `PASSED_OBJECTS` to define that the objects passed in a list via the API should be used as base set for the query. In our case this list contains only the object which has to be deleted. The dot in the select-clause of the P-OQL means that an object reference for the object has to be returned. *Object references* are the PCTE means to access objects.

- *Index structure 1* (`name`)

  Since this index structure addresses an attribute of the indexed objects themselves, changes in the index structure are only necessary when the value of this attribute is changed for a class. Hence, the following IUD is attached to the meta information for the object type *Class* for this index structure:

event:	changed value for attribute *name*
structure:	*<id of index structure 1>*
instances:	`select . from PASSED_OBJECTS`
action:	*UPDATE*

- *Index structure 2* (`count _.method->.`)

  If a derived attribute addressed in an index structure is defined by a regular path expression, each modification of a link which could be part of a path matching the regular path expression can affect the value of the derived attribute for an indexed object. Furthermore, modifications at the destination objects can affect the value of the derived attribute for an indexed object if the destination objects are addressed in the path expression. Since `'_.method->.'` addresses only the link itself, the destination objects need not be considered for this index structure. Hence, we only have to control the creation and the deletion of links of the type *method*.

  For the deletion of a link of type *method* we attach the following IUD to the meta information for the origin object type *Class*:

event:	deletion of a *method* link
structure:	*<id of index structure 2>*
instances:	`select . from PASSED_OBJECTS`
action:	*UPDATE*

Note that the *action* in this IUD is *UPDATE*, because the deletion of a *method* link will decrease the value of the derived attribute `'count _.method->.'` for the origin object of the link.

For the creation of *method* links the following IUD is generated for the origin object type *Class*:

event:	creation of a *method* link
structure:	*<id of index structure 2>*
instances:	`select . from PASSED_OBJECTS`
action:	*UPDATE*

Since changes of link attributes do not affect the value of the derived attribute `'count _.method->.'` the presented IUDs suffice to guarantee the updates for the second index structure.

- *Index structure 3*
  (`normalize _.method/_.parameter/->name`)

  Since multiple link definitions occur in the regular path expression used to define the derived attribute addressed in this index structure, we have to consider the path expression step by step.

  First, we have to react when a *method* link is created or deleted. To this end, IUDs corresponding to the IUDs generated for the second index structure are attached to the meta information for the object type *Class*. Since these IUDs are completely analogously to the IUDs defined for the second index structure, we refer to the above description for the details.

  Second, the creation and the deletion of *parameter* links has to be considered. For that the following IUDs are generated for the object type *Method*:

event:	creation of a *parameter* link
structure:	*<id of index structure 3>*
instances:	`select .for_class/->.` `from PASSED_OBJECTS`
action:	*UPDATE*

event:	deletion of a *parameter* link
structure:	*<id of index structure 3>*
instances:	`select .for_class/->.` `from PASSED_OBJECTS`
action:	*UPDATE*

The regular path expression `'.for_class/->.'` used in the P-OQL queries which determine the *Class* objects for which index entries have to be updated, inverts the first part of the regular path expression `'_.method/_.parameter/->name'`. It should be obvious that this inversion can be performed automatically.

Third, we have to react when the *name* attribute of a *Parameter* object is changed. To this end the following IUD is attached to the meta information of the object type *Parameter*:

event:	changed value of the attribute *name*
structure:	*<id of index structure 3>*
instances:	`select .for_method/.for_class/->.` `from PASSED_OBJECTS`
action:	*UPDATE*

Here the original path expression is completely inverted in the P-OQL query determining the *Class* objects for which the corresponding index entries have to be updated.

There is no need to add further IUDs to react, when an object of type *Parameter* is created or deleted, because in the considered index definition *Parameter* objects

are always reached via corresponding links; and IUDs for the creation and the deletion of these links have already been created.

- *Index structure 4*

```
(D_vector [{c shield subclass}]*/->.)
```

With this index definition the iteration `[...]*` as well as the category definition together with the `shield` option induce some problems when deriving the corresponding IUDs. Technically spoken, we have to employ the PCTE schema operations to recursively determine all object types which may occur at the end of a path matching the expression `'[{c shield subclass}]*/->.'`. To this end, the link types for which instances can be traversed because of the expression must be determined together with their destination object types.

We start with the object type *Class*. *Class* objects themselves can occur at the end of a path matching the above path expression, because the iteration allows to traverse no link at all. Hence, we have to react whenever a string attribute of a class is changed. Therefore, we add the object type *Class* to an initially empty auxiliary set for the object types, which may occur at the end of a path matching the expression `'[{c shield subclass}]*/->.'`.

Starting from a class, we can traverse arbitrary *composition* links except for links of type *subclass*. This corresponds to the link types *attribute* and *method*. Hence, we have to add the destination object types *Attribute* and *Method* to our set of potential path destination objects. Starting from attributes and methods, we can traverse *parameter* links. Therefore, we have to add the destination object type *Parameter* to our auxiliary set. Now this set is complete, because all link types which fulfill the given definition and start from an object type in our auxiliary set have already been considered.

For each object type in our auxiliary set (*Class*, *Attribute*, *Method* and *Parameter*) we have to maintain the following IUD:

event:	changed value of a string attribute
structure:	*<id of index structure 4>*
instances:	

```
select B:.
from A in PASSED_OBJECTS,
     B in (A:[@c shield superclass]*/->.)
where B:. is of type Class
```

action:	*UPDATE*

With respect to the query given in the above IUD for the determination of the affected instances it has to be mentioned that this query represents the most general choice. Obviously the full exploitation of the schema information would allow for more restrictive queries for the object types.

In addition to changes in the string attributes of the destination objects, we have to react, when a link which can be part of a path addressed by the regular path expression `'[{c shield subclass}]*/->.'` is created or deleted. To this end, we add IUDs for the object type *Class* which control the creation and the deletion of *attribute* and *method* links. For the object type *Method* we add IUDs controlling the creation and the deletion of *parameter* links. Here we state only the IUDs for the object type *method*, because the IUDs for the object type *class* are completely analogous:

event:	creation of a *parameter* link
structure:	*<id of index structure 4>*
instances:	

```
select B:.
from A in PASSED_OBJECTS,
     B in (A:[@c shield superclass]*/->.)
where B:. is of type Class
```

action:	*UPDATE*

event:	deletion of a *parameter* link
structure:	*<id of index structure 4>*
instances:	

```
select B:.
from A in PASSED_OBJECTS,
     B in (A:[@c shield superclass]*/->.)
where B:. is of type Class
```

action:	*UPDATE*

Two important points have to be mentioned with respect to the inversion of regular path expressions.

First, the invertability of regular path expressions is by no means a special feature of P-OQL and the PCTE data model. In the ODMG data model underlying OQL for example inverse relationships are also common. Furthermore, since the ODMG data model does not provide link key attributes or link categories, the inversion of a regular path expression becomes even simpler for OQL than for P-OQL. However — and this is the second important point — neither in P-OQL nor in OQL it is possible to invert all conceivable path expressions. A simple example occurs in PCTE when links with the category *designation* are addressed in a path expression. Since designation links do not have a reverse link, such an expression cannot be inverted. However, this does not seem to be a major restriction. If we want to use a link type in an index definition, we have to use one of the categories *composition*, *existence*, or *reference*.

Summarizing, the derivation of the IUDs for a given index definition represents a non-trivial problem. This holds especially for the queries which are needed to determine the affected instances. Further problems arise from the fact that we allow to address an object type together with all its subtypes in an index definition.

A comprehensive set of rules for the "inversion" of the index definitions is an open topic at the moment. Our implementation uses only a restricted set of rules which cover the most usual language facilities. This seems to be admissible, because most of the extremely complicated cases are either relatively unlikely in practical applications or can be replaced by simpler constructs. One example in this respect is the `shield` option addressing object types. There is no general way to invert path expression using this facility. However, a `shield` option addressing object types can always be replaced by a `shield` option addressing link types, which can be inverted relatively easily.

Finally it has to be mentioned that the described update procedure can cause significant efficiency problems, especially when locking aspects are taken into consideration. However, improvements in various directions seem possible. One opportunity could be to defer index updates to times with a low system load. This is an approach which seems to be well suited especially in the context of software engineering environments. Here updates usually have a high degree

of locality. In this case the index structure can be used to speed up the access to the rest of the database whereas the small part of the database which is actually subject to change is scanned completely.

4 Integration of User-defined Methods

The integration of user-defined methods would require that *inverse methods* are available for the methods used in index definitions. Such an *inverse method* must exist for each object (and link) type, for which changes to the instances can affect the result value of the method. In case, the inverse methods can be used to define the query determining the instances for which index entries have to be updated.

In fact, the need for inverse methods is a hard restriction for the applicability of our approach with respect to user-defined methods. However, there is no more general approach for the integration of user-defined methods, because a method may access arbitrary information for the calculation of its result.

5 Implementation Aspects

Our implementation of the described approach exploits the sophisticated schema management operations of PCTE to derive the IUDs. These schema management facilities allow to address the complete meta information about object types, link types and attribute types. Using these facilities the inversion of regular path expressions can be implemented with moderate effort.

In order to react when objects or links are created, deleted or changed, we employ the fine-grained notification mechanism of H-PCTE. This thread-based notification mechanism allows to define up-call procedures, which are invoked when certain change operations are performed for an object or link. Each up-call procedure has parameters defining the changed object or link. These parameters can be exploited to determine and restrict the IUDs which have to be checked in response to the change causing the up-call.

The complete update operation (change in the object base and all respective up-calls) are performed under the protection of the PCTE transaction mechanism. In the future we will try to implement less restrictive approaches which allow for a higher degree of concurrency.

6 Conclusion

We have presented a concrete approach for the index update problem arising from the consideration of derived attributes. The basic principle behind our approach is to extend the meta information for the object types by so called *index update definitions* (IUDs). For each object type there is one IUD for each index structure which might be affected by the creation, deletion or update of an object of this type. Each IUD contains the information necessary to perform the update of the index structure. More precisely each IUD is a quadruple containing: (1) A description of the event, which causes the necessity of the index update. (2) A reference to the potentially affected index structure. (3) A query determining the instances for which the index entries have to be updated, and (4) the necessary index actualization operation. The IUDs for an index structure can be automatically created when the index structure is defined. They allow for an efficient processing of update operations in the database.

Future work will be necessary primarily with respect to a comprehensive rule set for the creation of the IUDs covering as much facilities of the query language as possible. Furthermore bulk update techniques as sketched at the end of section 3 represent an interesting research topic with respect to our approach.

References

[Cat93] R. Cattell, editor. *The Object Database Standard: ODMG-93.* Morgan Kaufmann, San Mateo, Cal., USA, 1993.

[Com79] D. Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, June 1979.

[DGG95] K.R. Dittrich, S. Gatziu, and A. Geppert. The active database management system manifesto: A rulebase of ADBMS features. In *Proc. 2nd Int. Workshop on Rules in Database Systems*, volume 985 of *LNiCS*, pages 3–20, Glyfada, Athens, Greece, 1995.

[DHW95] U. Dayal, E.N. Hanson, and J. Widom. Active database systems. In W. Kim, editor, *Modern Database Systems: The Object Model, Interoperability, and Beyond*, pages 434–456. ACM Press and Addison-Wesley, 1995.

[GL93] T. Griffin and L. Libkin. Incremental maintenance of views with duplicates. In *Proc. 1995 ACM SIGMOD Int. Conf. on Management of Data*, pages 328–339, San Jose, Cal., USA, 1993.

[GM95] A. Gupta and I.S. Mumick. Maintenance of materialized views: Problems, techniques, and applications. *Data Engineering Bulletin*, 18(2):3–18, 1995.

[GMS93] A. Gupta, I.S. Mumick, and Subrahmanian. Maintaining views incrementally. In *Proc. 1993 ACM SIGMOD Int. Conf. on Management of Data*, pages 157–166, Washington, DC, USA, 1993.

[Har95] D. Harman. Overview of the second text retrieval conf. (TREC-2). *Information Processing and Management*, 31(3):271–289, 1995.

[Hen95] A. Henrich. P-OQL: an OQL-oriented query language for PCTE. In *Proc. 7th Conf. on Software Engineering Environments*, pages 48–60, Noordwijkerhout, Niederlande, 1995. IEEE Computer Society Press.

[Hen96a] A. Henrich. Adapting a spatial access structure for document representations in vector space. In *Proc. 5th Int. Conf. on Information and Knowledge Management*, pages 19–26, Rockville, Md., USA, 1996.

[Hen96b] A. Henrich. Document retrieval facilities for repository-based system development environments. In *Proc. 19th Annual Int. ACM SIGIR Conf. on Research and Development in Information Retrieval*, pages 101–109, Zürich, 1996.

[Kel92] U. Kelter. H-PCTE: A high-performance object management system for system development environments. In *Proc. 16th Annual Int. Computer Software and Applications Conf.*, pages 45–50, Chicago, Ill., USA, September 1992.

[PCT93] Portable Common Tool Environment - Abstract Specification / C Bindings / Ada Bindings. Standards ECMA-149/-158/-165, 3rd edition, 1993.

[PCT94] Portable Common Tool Environment - Abstract Specification / C Bindings / Ada Bindings. ISO IS 13719-1/-2/-3, 1994.

[SB88] G. Salton and C. Buckley. Term-weighting approaches in automatic text retrieval. *Information Processing & Management*, 24(5):513–523, 1988.

[SWY75] G. Salton, A. Wong, and C.S. Yang. A vector space model for automatic indexing. *Communications of the ACM*, 18(11):613–620, November 1975.

[WJ93] L. Wakeman and J. Jowett. *PCTE - The standard for open repositories.* Prentice Hall, Hemel Hempstead, Hertfordshire, UK, 1993.