Physical Object Management

Alfons Kemper

Fakultät für Mathematik und Informatik
Universität Passau
D–94030 Passau, F. R. G.
kemper@db.fmi.uni-passau.de

Guido Moerkotte

Fakultät für Informatik
Universität Karlsruhe
D–76128 Karlsruhe, F. R. G.
moer@ira.uka.de

In this chapter we will discuss a few physical object management techniques that are rooted in the relational context but were thoroughly adapted to the needs and requirements of the object-oriented model(s). After introducing a sample object-oriented database schema (and an extension thereof) we discuss *Access Support Relations*, an indexing technique for path expressions. Then, we overview techniques for indexing within type hierarchies and describe the materialization of functions in an object-oriented context. We also provide an overview of techniques for deriving good object clustering and finally sketch a few proposals for storage models for objects.

1 The Running Example

The discussion of this chapter is based on an object-oriented data model, called GOM (Generic Object Model) [KM93], that unites the most salient features of many recently proposed models in one coherent framework. In this respect, the objective of GOM can be seen as providing a syntactical framework of the essential object-oriented features. Independently, but with the same intention, Zdonik and Maier proposed the so-called Reference Model in [ZM89]. The features that GOM provides are relatively *generic* (and basic) such that the results derived for this particular data model can be applied to a variety of other object-oriented models.

Let us now introduce an example object base, called *Company*. This database will be used throughout the chapter to illustrate the physical object management techniques. The (mostly self-explanatory) type definitions are shown in Figure 1. The type *Manager* is defined as a subtype of *Emp* which has two consequences: (1) *Manager* instances inherit the structural and behavioral specification of *Emp* instances and (2) a *Manager* instance can occur wherever an *Emp* object is required. The latter is a consequence of the *substitutability* property. The type *Emp* (and also *Manager*) contains a set-valued[1] attribute *jobHistory* referring to an arbitrary number of *Task* instances.

A (small) object base extension for this schema is shown in Figure 2. An object may be viewed as a triple (*OID*, *Type*, *rep*) where *rep* corresponds to the internal state (i.e., everything within the box). The labels id_i for $i \in \{1, 2, 3, \ldots\}$ denote the system-wide unique object identifiers (*OIDs*). References via attributes are maintained uni-directionally in GOM—as in almost all other object models. For example, in an extension of the above schema there exists a reference in the form

[1]Note, that there is a major difference between a (non-sharable) set value and a set object.

```
type Emp is
  body
    [name: string;
     worksIn: Dept;
     salary: int;
     sex: char;
     jobHistory: {Task};]
  operations
    declare skill: → int;
  implementation
    define skill is  !! derived from the jobHistory
      begin
      var totalC, totalD: float := 0.0;
      foreach (t in self.jobHistory)
        begin
        totalC := totalC + t.competence * t.duration;
        totalD := totalD + t.duration;
        end foreach
      return round(totalC/totalD);
    end define skill;
end type Emp;
```

```
type Task is
  body
    [competence: int;
     duration: float;]
end type Task

type Dept is
  body
    [name: string;
     secretary: Emp;
     mgr: Manager;]
end type Dept;

type Manager
  supertype Emp is
  body
    [backUp: Emp;]
end type Manager;
```

Figure 1: Type Definitions of the Running Example *Company*

Task

	id_0	id_1	id_2	id_3
duration	12	9	22	13
competence	10	7	8	3

Emp

	id_4	id_5	id_6	id_7
name	"Sander"	"Versace"	"Hinault"	"LeMond"
worksIn	id_8	id_8	id_9	id_9
salary	90000	100000	260000	100000
sex	'F'	'M'	'M'	'M'
jobHistory	$\{id_0, id_1\}$	$\{id_2\}$	$\{id_3\}$	$\{\ldots\}$

Dept

	id_8	id_9	id_{10}
name	"Clothes"	"Bicycles"	"Shoes"
secretary	id_4	id_6	*NULL*
mgr	id_{11}	id_{12}	id_{13}

Manager

	id_{11}	id_{12}	id_{13}
name	"Boss"	"Chief"	"Master"
worksIn	id_8	id_9	*NULL*
salary	150000	*NULL*	900000
sex	'M'	'M'	'M'
jobHistory	$\{\ldots\}$	$\{\ldots\}$	$\{\ldots\}$
backUp	id_5	id_7	*NULL*

Figure 2: Example Extension of the Object Base *Company*

of a stored *OID* from an *Emp*loyee to his or her *Dept* via the *worksIn* attribute, but not vice versa. These references are maintained by storing the unique *OID* of the referenced object in the referencing attribute or variable.

Let us sketch a few example queries based on the *Company* object base—formulated in the query language GOMql [KMP93]:

Q_1: **range** e : Emp	Q_2: **range** e : Emp	Q_3: **range** e : Emp
retrieve e	**retrieve** e.worksIn.mgr.salary	**retrieve** e
where e.worksIn.mgr.salary < 200000	**where** e.name = "Sander"	**where** e.skill = 10

In query Q_1 those *Emp*loyees are retrieved whose *Manager* earns less than 200000. This query involves a path expression (i.e., *e.worksIn.mgr.salary*), which is supported by the indexing structure discussed in Section 2. With respect to this particular path expression, the query Q_1 is called a backward query because it logically involves traversing the path backwards from the qualifying *Manager* instances—i.e., those earning less than 200000—to the *Emp* instances. Query Q_2, on the other hand, is a forward query since it involves traversing the same path expression in forward direction. The query Q_3 contains a user-defined function in the selection predicate. This query requires the evaluation of the function *skill* for all *Emp* instances. Section 4 describes an optimization technique for expediting the evaluation of such a query.

It should be obvious that, in particular, queries Q_1 and Q_3 are costly to evaluate—in the absence of suitable index structures.

2 Access Support Relations: Indexing Path Expressions

In the context of associative search, one of the most performance-critical operations in relational databases is the *join* of two or more relations. Much research has been done into expediting joins, for example, access structures to support join, *sort-merge* join and *hash-join* algorithms have been developed. Recently, the binary join index structure [Val87], building on links [Här78], has been proposed.

In object-oriented database systems with object references the join based on matching attribute values plays a less predominant role. More important are object accesses along reference chains leading from one object instance to another. This kind of object traversal is also called *functional* or *implicit join*.

In this section we present a general indexing structure, called *Access Support Relations (ASRs)*, which is designed to support functional join along arbitrarily long attribute chains which may even contain collection-valued attributes. In this respect access support relations constitute materializations of frequently traversed reference chains.

2.1 Definitions

A path expression [KKS92] has the form

$$o.A_1.\cdots.A_n$$

where o is a tuple structured object containing the attribute A_1 and $o.A_1.\cdots.A_i$ refers to an object or a set of objects, all of which have an attribute A_{i+1}. The result of the path expression is the set of objects (or values) of type t_n that can be reached from o via the specified attribute chain.

Formally, a path expression or attribute chain is defined as follows:

Definition 2.1 (Path Expression) *Let $t_0, \ldots, t_n$ be (not necessarily distinct) types. A path expression on t_0 is an expression $t_0.A_1.\cdots.A_n$ iff for each $1 \leq i \leq n$ one of the following conditions holds:*

- *The type t_{i-1} is defined as* **type** t_{i-1} **is** $[\ldots, A_i : t_i, \ldots]$, *i.e., t_{i-1} is a tuple with an attribute A_i of type t_i*[2]*.*

- *The type t_{i-1} is defined as* **type** t_{i-1} **is** $[\ldots, A_i : \{t_i\}, \ldots]$, *i.e., the attribute A_i is set-structured. In this case we speak of a set occurrence at A_i in the path $t_0.A_1.\cdots.A_n$.*

For simplicity of presentation we assume that the types involved are not being defined as subtypes of some other type. This, of course, is generally possible; it would only make the definition a bit more complex.

The second part of the definition is useful to support access paths through sets. If it does not apply for a given path the path is called *linear*. A path expression that contains at least one set-valued attribute is called *set-valued*.

Since an access path can be seen as a relation, we will use relation extensions to represent materialized path expressions. The next definition maps a given path expression to the underlying access support relation declaration.

Definition 2.2 (Access Support Relation (ASR)) *Let $t_0, \ldots, t_n$ be types, $t_0.A_1.\cdots.A_n$ be a path expression. Then the access support relation $[\![t_0.A_1.\cdots.A_n]\!]$ is of arity $n+1$ and has the following form:*

$$[\![t_0.A_1.\cdots.A_n]\!] : [S_0, \ldots, S_n]$$

The domain of the attribute S_i is the set of identifiers (OIDs) of objects of type t_i for $(0 \leq i \leq n)$. If t_n is an atomic type then the domain of S_n is t_n, i.e., values are directly stored in the access support relation.

We distinguish several possibilities for the extension of such relations. To define them for a path expression $t_0.A_1.\cdots.A_n$ we need n temporary relations $[\![t_0.A_1]\!]$, ..., $[\![t_{n-1}.A_n]\!]$.

Definition 2.3 (Temporary Binary Relations) *For each i $(1 \leq i \leq n)$—that is, for each attribute in the path expression—we construct the temporary binary relation $[\![t_{i-1}.A_i]\!]$. The relation $[\![t_{i-1}.A_i]\!]$ contains the tuples $(id(o_{i-1}), id(o_i))$ for every object o_{i-1} of type t_{i-1} and o_i of type t_i such that*

- *$o_{i-1}.A_i = o_i$ if A_i is a single-valued attribute.*

- *$o_i \in o_{i-1}.A_i$ if A_i is a set-valued attribute.*

If t_n is an atomic type then $id(o_n)$ corresponds to the value $o_{n-1}.A_n$. Note, however, that only the last type t_n in a path expression can possibly be an atomic type.

Let us reconsider the path expression of our schema *Company*, for which we now indicate the type constraints with the underbraces:

[2]This means that the attribute A_i can be associated with objects of type t_i or any subtype thereof.

$$P \equiv \underbrace{\underbrace{\underbrace{Emp.worksIn}_{Dept}.mgr}_{Manager}.salary}_{int}$$

When considering the update problem, it should be obvious that strong typing is vital to indexing over path expressions. Therefore, models with a more relaxed typing paradigm have to impose user-specified and dynamically controlled type constraints on attributes and/or paths that are indexed.

For the path expression specified above, the temporary binary relations have the following extensions:

⟦Emp.worksIn⟧	
OID_{Emp}	OID_{Dept}
id_4	id_8
id_5	id_8
id_6	id_9
id_7	id_9
id_{11}	id_8
...	...

⟦Dept.mgr⟧	
OID_{Dept}	$OID_{Manager}$
id_8	id_{11}
id_9	id_{12}
id_{10}	id_{13}

⟦Manager.salary⟧	
$OID_{Manager}$	int
id_{11}	150000
id_{13}	900000

2.2 Extensions of Access Support Relations

We now introduce different possible extensions of the ASR $[\![t_0.A_1.\cdots.A_n]\!]$. We distinguish four extensions:

1. The *canonical* extension, denoted $[\![t_0.A_1.\cdots.A_n]\!]_{can}$ contains only information about complete paths, that is, paths originating in t_0 and leading (all the way) to t_n. Therefore, it can only be used to evaluate queries that originate in an object of type t_0 and "go all the way" to t_n.

2. The *left-complete* extension $[\![t_0.A_1.\cdots.A_n]\!]_{left}$ contains all paths originating in t_0 but not necessarily leading to t_n, but possibly ending in a *NULL*.

3. The *right-complete* extension $[\![t_0.A_1.\cdots.A_n]\!]_{right}$, analogously, contains paths leading to t_n, but possibly originating in some object o_j of type t_j which is not referenced by any object of type t_{j-1} via the A_j attribute.

4. Finally, the *full* extension $[\![t_0.A_1.\cdots.A_n]\!]_{full}$ contains all partial paths, even if they do not originate in t_0 or do end in a *NULL*.

Definition 2.4 (Extensions) *Let* $\bowtie$ ($⟗$, $⟕$, $⟖$) *denote the natural (outer, left outer, right outer) join on the last column of the first relation and the first column of the second relation. Then the different extensions are obtained as follows:*

$$\begin{array}{rcl}
[\![t_0.A_1.\cdots.A_n]\!]_{can} & := & [\![t_0.A_1]\!] \bowtie \cdots \bowtie [\![t_{n-1}.A_n]\!] \\
[\![t_0.A_1.\cdots.A_n]\!]_{full} & := & [\![t_0.A_1]\!] ⟗ \cdots ⟗ [\![t_{n-1}.A_n]\!] \\
[\![t_0.A_1.\cdots.A_n]\!]_{left} & := & \left(\cdots\left([\![t_0.A_1]\!] ⟕ [\![t_1.A_2]\!]\right)\cdots ⟕ [\![t_{n-1}.A_n]\!]\right) \\
[\![t_0.A_1.\cdots.A_n]\!]_{right} & := & \left([\![t_0.A_1]\!] ⟖ \cdots\left([\![t_{n-2}.A_{n-1}]\!] ⟖ [\![t_{n-1}.A_n]\!]\right)\cdots\right)
\end{array}$$

For our example path expression, the full extension $[\![Emp.worksIn.mgr.salary]\!]_{full}$ looks as follows:

$[\![$**Emp.worksIn.mgr.salary**$]\!]_{\mathbf{full}}$			
$S_0 : OID_{Emp}$	$S_1 : OID_{Dept}$	$S_2 : OID_{Manager}$	$S_3 : int$
id_4	id_8	id_{11}	150000
id_5	id_8	id_{11}	150000
id_{11}	id_8	id_{11}	150000
id_6	id_9	id_{12}	—
—	id_{10}	id_{13}	900000
...	...	...	...

This extension contains all paths and subpaths corresponding to the underlying path expression. The first three tuples actually constitute complete paths which would be present in the canonical extension as well; however the fourth and the fifth paths would be omitted in the canonical extension. In the left-complete extension the first four tuples would be present, whereas the fifth one would be omitted since it does not originate in *Emp*. Analogously, the right-complete extension would contain the first three and the fifth tuple and omit the fourth tuple since it does not end in an *int* value representing the *salary* of some *Manager*.

It should be obvious that the full extension of an ASR contains more information than the left- or right-complete extensions which, in turn, contain more information than the canonical extension. The right- and left-complete extensions are incomparable.

The difference in information contents has implications on the applicability of an ASR in evaluating a path expression originating in an object of type s:

Definition 2.5 (Applicability) *An access support relation* $[\![t_0.A_1.\cdots.A_n]\!]_X$ *under extension* X *is applicable for a path* $s.A_i.\cdots.A_j$ *where* s *is a subtype*[3] *of* t_{i-1} *under the following condition, depending on the extension* X*:*

$$Applicable([\![t_0.A_1.\cdots.A_n]\!]_X, s.A_i.\cdots.A_j) = \begin{cases} X = full & \wedge \quad 1 \le i \le j \le n \\ X = left & \wedge \quad 1 = i \le j \le n \\ X = right & \wedge \quad 1 \le i \le j = n \\ X = can & \wedge \quad 1 = i \le j = n \end{cases}$$

The *full* extension can be utilized to evaluate any subpath, the *left-* and *right-complete* extensions can only be applied to evaluate prefix- and suffix-paths, respectively. The *canonical* extension can only be applied to evaluate a complete path expression (i.e., one that spans the entire access support relation).

2.3 Storage Structure

The storage structure of access support relations is borrowed from the binary join index proposal [Val87]. Each ASR is redundantly stored in two index structures: the first being keyed on the left-most attribute and the second on the right-most attribute. Suitable index structures are hash tables or B^+-trees. The hash table is particularly suitable for keys consisting of OID attributes

[3]Note that every type is a subtype of itself.

since only exact match queries have to be supported. On the other hand, B^+-trees are advantageous for attributes that allow range queries, for example, *int* and *float* values. Note that these attributes can only occur at the right-most column of an ASR; an example is the right-most column of the ASR $[\![Emp.worksIn.mgr.salary]\!]$ which represents the *salary* attribute of *Manager* instances. The following discussion is solely based on B^+-trees; however, it can easily be adapted to hash tables.

Graphically, the redundant storage scheme consisting of two B^+-trees for each ASR is visualized for the canonical ASR $[\![Emp.worksIn.mgr.salary]\!]_{can}$ as follows:

B^+

$[\![$**Emp.worksIn.mgr.salary**$]\!]_{\mathbf{can}}$			
$S_0 : OID_{Emp}$	$S_1 : OID_{Dept}$	$S_2 : OID_{Manager}$	$S_3 : int$
id_4	id_8	id_{11}	150000
id_5	id_8	id_{11}	150000
id_{11}	id_8	id_{11}	150000
...	...	...	...

B^+

We will call the left B^+-tree the "forward clustered" tree and, analogously, the right one the "backward clustered" tree. The left B^+-tree supports the evaluation of a forward query (e.g., retrieving the salary of id_4's *Manager*). The right B^+-tree supports the evaluation of backward queries—with respect to the underlying path expression. For our example, an entry point for finding the *Emp* instances whose *Manager*'s *salary* is below 200000 is provided by the backward clustered B^+-tree.

This storage scheme is also well suited for traversing paths from left-to-right (forward) as well as from right-to-left (backward) even if they span over several access support relations. Again, let us graphically visualize the situation:

B^+

$[\![$**Emp.worksIn.mgr**$]\!]_{\mathbf{can}}$		
$S_0 : OID_{Emp}$	$S_1 : OID_{Dept}$	$S_2 : OID_{Manager}$
id_4	id_8	id_{11}
id_5	id_8	id_{11}
id_{11}	id_8	id_{11}
...	...	...

B^+

B^+

$[\![$**Manager.salary**$]\!]_{\mathbf{can}}$	
$S_0 : OID_{Manager}$	$S_1 : int$
id_{11}	150000
id_{13}	900000
...	...

B^+

The above example illustrates the virtues of the redundant storage model for ASRs. The right B^+-tree of the ASR $[\![Manager.salary]\!]_{can}$ directly supports the lookup of those *Manager* instances whose *salary* is below 200000 (i.e., the *Manager* with OID id_{11} in our example). Then, the right B^+-tree of the ASR $[\![Emp.worksIn.mgr]\!]_{can}$ supports the traversal to the corresponding *Emp* instances to obtain the result $\{id_4, id_5, id_{11}\}$.

Thus, the backward traversal constitutes a "right-to-left" semi-join across ASRs:

$$\pi_{S_0}\left([\![Emp.worksIn.mgr]\!]_{can} \ltimes \left(\sigma_{S_1<200000}\, [\![Manager.salary]\!]_{can}\right)\right)$$

Analogously, the "forward clustered" B^+-tree supports the semi-join from left to right, such that, for instance, the *salary* of id_4's *Manager* can be retrieved efficiently. This corresponds to the "left-to-right" semi-join across ASRs:

$$\pi_{S_3}\left(\left(\sigma_{S_0=id_4}\,[\![Emp.worksIn.mgr]\!]_{can}\right) \rtimes [\![Manager.salary]\!]_{can}\right)$$

2.4 Related Work

Access support relations were developed by the authors [KM90]. A more detailed description including a comprehensive quantitative evaluation can be found in [KM92]. Access support relations constitute a generalization of two relational techniques: the *links* developed in [Här78] and the binary *join indices* proposed in [Val87]. Rather than relating only two relations (or object types), our technique allows support of access paths ranging over many types. The ASR scheme subsumes and extends several previously proposed strategies for access optimization in object bases. The index paths in [MS86] are restricted to chains that contain only single-valued attributes and their representation is limited to binary partitions of the access path. Similarly, the object-oriented access techniques described for the Orion model [BK89] are contained as a special case in our framework.

Our technique differs in three major aspects from the aforementioned approaches:

- access support relations allow collection-valued attributes within the attribute chain
- access support relations may be maintained in four different *extensions*. The extension determines the amount of (reference) information that is kept in the index structure.
- The paths over which ASRs are defined may be decomposed into partitions (sub-paths) of arbitrary lengths. This allows the database designer to choose the best extension and path partitioning according to the particular application characteristics.

Also the (separate) replications of object values as proposed for the Extra object model [SC89] and for the POSTGRES model [SAH87, Sel88] are largely subsumed by ASRs.

3 Indexing over Type Hierarchies

For the discussion of this section consider the type hierarchy shown in Figure 3. Based on this type hierarchy, we can phrase the following three queries, the meaning of which should be obvious:

Q_4: **range** e : Emp	Q_5: **range** c : CEO	Q_6: **range** p : Person
retrieve e	**retrieve** c	**retrieve** p
where e.salary > 200000	**where** c.salary > 200000	**where** p.age > 60

In query Q_4 we want to retrieve *Emp* instances (including *Manager* and *CEO* instances) whose *salary* exceeds 200000. In query Q_5, on the other hand, we are only interested in the *CEO* instances with such a high *salary*; because of substitutability the result of Q_5 is a subset of the result obtained in query Q_4. In query Q_6 all *Person* instances whose *age* exceeds 60 are retrieved (i.e., including all *Emp*, *Manager*, *CEO* and *Student* instances).

The evaluation of such queries can (and should) be supported by indexing. Indexing can be viewed as a special case of access support relations, except that mostly the backward clustered B^+-tree is relevant and, therefore, the forward clustered tree may be omitted. Consider, for example,

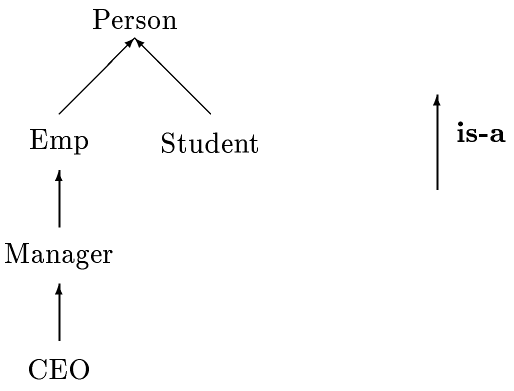


Figure 3: Sample Type Hierarchy

the index for *Emp.salary* which is represented as the ASR $[\![Emp.salary]\!]$ as follows—assuming that id_{77} and id_{88} identify *CEO* instances:

$[\![$Emp.salary$]\!]$	
$S_0 : OID_{Emp}$	$S_1 : int$
id_4	90000
id_5	100000
id_7	100000
id_{11}	150000
id_6	260000
id_{13}	900000
id_{77}	1500000
id_{88}	2000000
...	...

B^+

Note that this index, because of substitutability, implicitly includes all *Manager* and *CEO* instances, for which the *salary* is "known". Therefore, the evaluation of query Q_4 is well supported since it involves a lookup in a single B$^+$-tree only. However, query Q_5 is not as well supported since it involves retrieving the OIDs from the index $[\![Emp.salary]\!]$ for which the salary attribute exceeds 200000. The resulting set, however, contains *Emp* and *Manager* instances as well. So, in a (costly) second phase, the *CEO* instances have to be extracted from this set.

3.1 Single Type Indexing

The idea of single type indexing is to incorporate only the *direct* instances of a particular type in the index. Let us denote the set of direct instances of a type T as $\overline{T}$. Then, for our example we could create three separate indexes: $[\![\overline{Emp}.salary]\!]$, $[\![\overline{Manager}.salary]\!]$, and $[\![\overline{CEO}.salary]\!]$.

These indexes have the following form:

$[\![\overline{\text{Emp}}$.salary$]\!]$	
$S_0 : OID_{Emp}$	$S_1 : int$
id_4	90000
id_5	100000
id_7	100000
id_6	260000
...	...

B^+

$[\![\overline{\text{Manager}}$.salary$]\!]$	
$S_0 : OID_{Manager}$	$S_1 : int$
id_{11}	150000
id_{13}	900000
...	...

B^+

$[\![\overline{\text{CEO}}$.salary$]\!]$	
$S_0 : OID_{CEO}$	$S_1 : int$
id_{77}	1500000
id_{88}	2000000
...	...

B^+

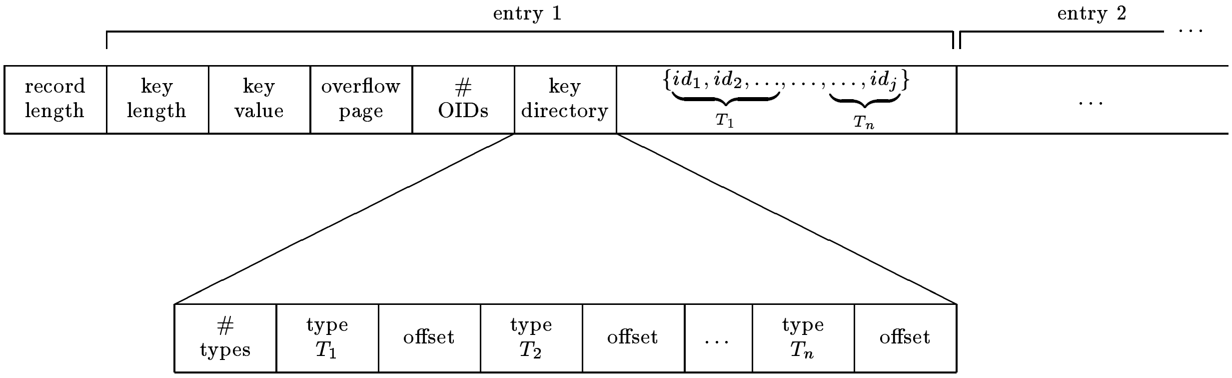
Now, evaluating query Q_5 is well supported because it involves only a lookup in the index $\llbracket \overline{CEO}.salary \rrbracket$. However, the evaluation of query Q_4 now involves a lookup in three separate B$^+$-trees and unioning the results:

$$\sigma_{S_1>200000} \llbracket \overline{Emp}.salary \rrbracket \cup \sigma_{S_1>200000} \llbracket \overline{Manager}.salary \rrbracket \cup \sigma_{S_1>200000} \llbracket \overline{CEO}.salary \rrbracket$$

This problem appears even more severe when considering query Q_6 under the assumption of separate single type indexing on the *age* attribute.

3.2 Type Hierarchy Indexing

Because of the above discussed disadvantages of separate single type indexing, [KKD89] proposed the use of a type hierarchy index. The type hierarchy index consists of a single B$^+$ tree which comprises all direct and indirect instances of the indexed type. This is basically what we adhered to in the ASR definitions; however [KKD89] developed a special structure of the leaf nodes which is sketched as follows:



This layout of the leaf nodes provides support for extracting the (OIDs of) objects of a particular type by jumping to the corresponding offset which is maintained in the key directory.

[LOL92] developed a sophisticated indexing scheme, called H-trees, for combining type hierarchy indexing with single type indexing. The basic idea consists of nesting B$^+$-trees, that is, nesting the index tree of a subtype within the tree of the supertype. For our example this is graphically shown in Figure 4, where the three H-trees, $H_{\overline{Emp}}$ for direct *Emp* instances, $H_{\overline{Manager}}$ for direct *Manager* instances, and $H_{\overline{CEO}}$ for *CEO* instances, are sketched.

The nesting is achieved by incorporating so-called L pointers which refer from the supertype index to the subtype index tree. There are two essential conditions for a valid H-tree nesting:

1. The range of the subtree referenced by an L-pointer must be contained in the range of the referencing node of the supertype tree. In terms of our example, the range of the subtree H_2 must be contained in the range of H_1, and the range of H_4 must be contained in the range of H_3.

2. All leaf nodes of a subtype tree have to be covered by the supertype tree. This means that all leaf nodes have to be reachable by following L-pointers emanating from the supertype tree.

A single type lookup on H-trees is carried out by searching in the corresponding H-tree and simply ignoring the L-pointers. A type hierarchy lookup is carried out by searching in the H-tree of the root type (over which the query is stated) and traversing the L-pointers to subtype trees.

Of course, the maintenance of the H-trees imposes a severe overhead on update operations; the exact penalty of which still has to be investigated more thoroughly.

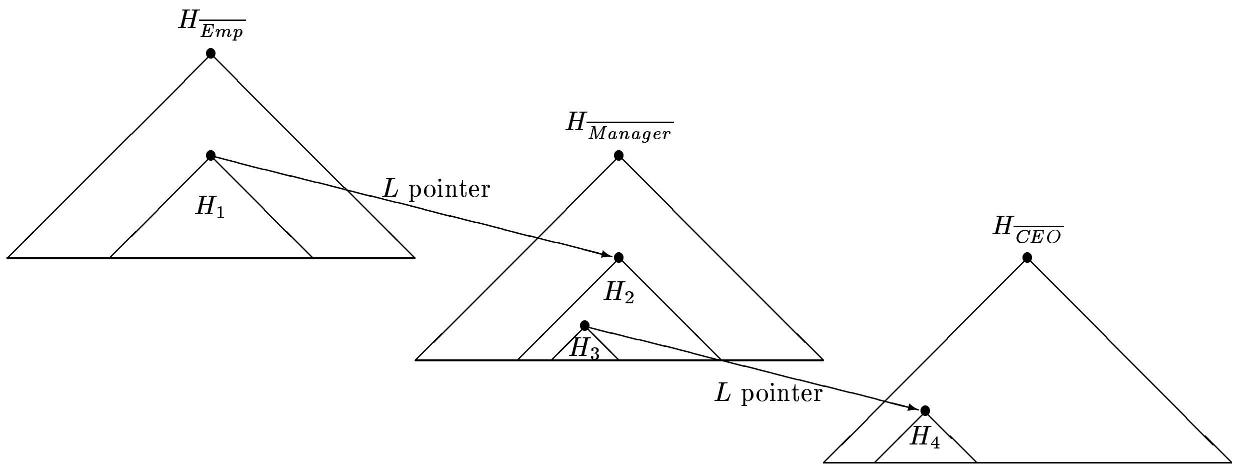


Figure 4: Three Nested H-Trees

4 Function Materialization

Let us now discuss an optimization technique that is devised to expedite the evaluation of queries containing function invocations, such as our example query Q_3. The technique is rooted in view materialization, which is a well-known technique in relational database systems. Here, we present the basics of *function materialization*, a similar, yet more powerful, optimization concept for object-oriented data models. Exploiting the object-oriented paradigm—namely *classification*, *object identity*, and *encapsulation*—facilitates a rather easy incorporation of function materialization into (existing) object-oriented systems. Only those types (classes) whose instances are involved in some materialization are appropriately modified and recompiled—thus leaving the remainder of the object system invariant. Furthermore, the exploitation of encapsulation (information hiding) and object identity provides for additional performance tuning measures which drastically decrease the invalidation and rematerialization overhead incurred by updates in the object base.

4.1 Storing Materialized Results

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

Definition 4.1 (Generalized Materialization Relation, GMR)
Let $t_1, \ldots, t_n, t_{n+1}, \ldots, t_{n+m}$ be types and let $f_1, \ldots, f_m$ be side-effect free functions with $f_j : t_1, \ldots, t_n \rightarrow t_{n+j}$ for $1 \leq j \leq m$. Then the generalized materialization relation $\langle\!\langle f_1, \ldots, f_m \rangle\!\rangle$ for the functions $f_1, \ldots, f_m$ is of arity $n + 2 * m$ and has the following form:

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

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

An extension of the GMR $\langle\langle f_1, \ldots, f_m \rangle\rangle$ is consistent if a *true* validity indicator implies that the associated materialized result is currently valid, that is,

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

In the remainder of this paper we consider only consistent GMR extensions.

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

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

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

In this presentation we will discuss only the materialization of functions having complex argument types. As can easily be seen it is not practical to materialize a function for all values of an atomic argument type (e.g., *float*). Therefore, in [KKM94] we proposed *restricted GMRs* for materializing functions for selected arguments only.

An example of a GMR comprising only a single materialized function (*Emp.skill*) is shown below:

⟨⟨Emp.skill⟩⟩		
$O_1 : OID_{Emp}$	$skill : int$	$V_{skill} : bool$
id_4	9	*true*
id_5	8	*true*
id_6	3	*true*
...	...	...
id_{12}	3	*true*
id_{13}	10	*true*

It should now be obvious that the example query Q_3 can be evaluated as

$$\pi_{O_1} \left(\sigma_{skill=10} \ \langle\langle Emp.skill \rangle\rangle \right)$$

as long as the GMR $\langle\langle Emp.skill \rangle\rangle$ contains the valid *skill* result for *all Emp* instances.

4.2 Storage Representation of GMRs

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

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

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

Unfortunately, the (currently existing) multi-dimensional storage structures, such as the Grid-File [NHS84], are not well-suited to support more than three or four dimensions. Therefore, more conventional indexing schemes have to be utilized to expedite access on GMRs of higher arity. The index structures are chosen according to the expected query mix, the number of argument fields in the GMR, and the number of functions in the GMR. A good proposal for multi-dimensional indexing relying on conventional B-trees is given in [Lum70].

4.3 Invalidation and Rematerialization of Function Results

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

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

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

Definition 4.2 (Reverse Reference Relation)
The Reverse Reference Relation RRR is a set of tuples of the form

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

For each tuple $r \in RRR$ the following condition holds: The object (with the identifier) $r.O$ has been accessed during the materialization of the function $r.F$ with the argument list $r.A$ (the angle brackets $\langle\ldots\rangle$ denote the list constructor in GOM syntax).

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

For our example object base *Company*, a part of the RRR that controls the invalidation of precomputed results in the GMR $\langle\langle Emp.skill\rangle\rangle$ is shown in Figure 5. Each time an object is updated in the object base, the RRR is inspected to determine which materialized results have to be invalidated (lazy rematerialization) or recomputed (immediate rematerialization). [KKM94] describes ways to detect object updates by schema modification and efficient algorithms for maintaining the RRR—which, of course, changes as a result of object base updates.

RRR		
O	F	A
id_0	Emp.skill	$\langle id_4 \rangle$
id_1	Emp.skill	$\langle id_4 \rangle$
id_4	Emp.skill	$\langle id_4 \rangle$
id_2	Emp.skill	$\langle id_5 \rangle$
id_3	Emp.skill	$\langle id_6 \rangle$
...	...	...

$\langle\langle$**Emp.skill**$\rangle\rangle$		
$O_1 : OID_{Emp}$	$skill : int$	$V_{skill} : bool$
id_4	9	*true*
id_5	8	*true*
id_6	3	*true*
...	...	...
id_{12}	3	*true*
id_{13}	10	*true*

Figure 5: The Data Structures of the GMR Manager

4.4 Strategies to Reduce the Invalidation Overhead

The invalidation mechanism outlined so far is (still) rather unsophisticated and, therefore, induces unnecessarily high update penalties upon object modifications. In [KKM94], four dual techniques to reduce the update penalty—consisting of invalidation and rematerialization—by better exploiting the potential of the object-oriented paradigm are developed. The techniques described there are based on the following ideas:

1. *isolation of relevant object properties:* Materialized results typically depend on only a small fraction of the state of the objects visited in the course of materialization. For example, the materialized *skill* certainly does not depend on the *sex* and *name* attributes of an *Emp*.

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

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

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

4.5 Related Work

The details of function materialization including a quantitative assessment are presented in [KKM94]. Function materialization is similar to materialization of views in the relational context. The most important work is reported in [BCL89]. Further work in precomputing queries and database procedures was done in POSTGRES. Here, the "QUEL as a Datatype" attributes are precomputed and cached in separate data structures. The control concepts are discussed in [SAH87, Jhi88, Sel88].

5 Object Placement

The idea of *clustering* is to place objects which are logically related (i.e., frequently used together) physically close to each other. The effectiveness of clustering is measured in terms of the number of page faults occurring during an application. The lower the number of page faults the better. The interconnectivity of objects in object bases demands new clustering techniques.

There are two different approaches to clustering: *sequence-based clustering* and *partition-based clustering*. The former relies on algorithms producing an object sequence. The latter relies on partitioning algorithms which partition the object net into partitions such that each one fits into a single page. We discuss each of these approaches. Then we provide a discussion.

5.1 Sequence-Based Clustering

The sequence-based approach to clustering stores objects in a specific sequence onto disk. All known sequence-based clustering strategies can be seen as instances of a generic sequence-based clustering strategy consisting of two phases: (1) the *presort* stage which is followed by (2) the *traversal* phase. During the first stage, all objects to be clustered are sorted. In the second stage, the sorted objects serve as roots for the traversal algorithm. All objects reachable from the root object are appended to the clustering sequence—mostly directly on disk—according to their visiting order. Traversals may end for two reasons: no more objects can be reached from the current root object, and no more objects fit into the current page. In both cases, a new traversal starts with the next (not yet placed) object in the presorted object sequence as the root. Of course, special care has to be taken in order to prevent objects from being stored twice.

Obviously, the presort order must depend on some information associated with each object. This information has to be taken from an ordered domain. A first source of information is the object's time of creation (*toc*). Depending on whether we sort by increasing or decreasing time of creation, the two presort orders *toc* and toc^{-1} are derived. Another common set of presort orders is derived from—often user specified—type orders. This results in a partial order only, since objects which are instances of the same type remain unordered. The user should be allowed to specify the presort order. The most common example are presort orders which are based on attribute values and typically sort objects within type extents.

The most popular traversal algorithms are *depth first*, *breadth first*, and *best first* traversals. The *best first* traversal is successively applied to (not already stored) objects of the presort sequence until all objects are placed. The traversal depends on the information which links to follow and, if there exist several possible links, which one to choose. Link weights between 0 and 1 for each link provide sufficient information (the *best first* algorithm follows the one with the highest weight).

If we require that the weights take the values 0 and 1 only and all objects visited during one traversal fit into a single page, the traversal algorithm can be specified by a *placement tree*. A *placement tree* is a tree with nodes being labeled by types and edges being labeled by attribute names. For the *Company* extension shown in Figure 2, the example placement tree storing departments together with their managers and secretaries (see Figure 6) results in the following object placement (we assume that three objects fit into a single page):

$$[id_8, id_{11}, id_4], [id_9, id_{12}, id_6], [id_{10}, id_{13}, —]$$

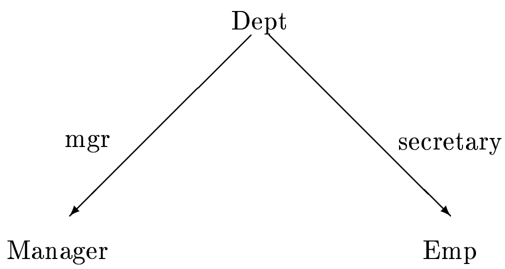


Figure 6: Sample Placement Tree

5.2 Partition-Based Clustering

The clustering problem is closely related to the partitioning problem in which a graph is to be partitioned into several disjunct subgraphs (partitions). The optimization problem consists of finding partitions such that the sum of the weights of all outgoing references (i.e., references to vertices contained in other partitions) is minimal. In partition-based clustering, the weights for the references can be determined in the same way as for sequence-based clustering. Given the weights, partitioning algorithms are applied to determine a partitioning under the constraint that the objects contained in each partition fit into a single page.

The first partitioning algorithm applied to clustering was the Kernighan&Lin graph partitioning heuristics [KL70]. It starts with an arbitrary partitioning and then iterates over each pair of partitions. During each iteration, objects are swapped between the partitions, if this results in a decrease in the total weight of all outgoing references. Problems occur, if the objects cannot be swapped due to their sizes and the space left in the pages of the partitions. Another disadvantage of this algorithm is its high asymptotic complexity of $O(n^{2.4})$.

These disadvantages led to the development of fast greedy graph partitioning algorithms based on Kruskal's algorithm for computing minimum-weight spanning trees. The basic version of such a greedy graph partitioning algorithm—called *ggp*—is outlined as follows:

define ggp(objectNet) **is**
    partitions := $\langle\,\rangle$;
    place each object in objectNet into a different page;
    insert these pages into *partitions*;
    Let *edgeList* be a list of tuples of the form (o_1, o_2, w_{o_1,o_2})
        where w_{o_1,o_2} is the total weight of all edges between o_1 and o_2;
    Sort *edgeList* by descending weights;
    foreach $((o_1, o_2, w_{o_1,o_2})$ **in** *edgeList*) **do**
    begin
      Let P_1, P_2 be the partitions containing objects o_1, o_2;
      if ($P_1 \neq P_2$ **and** the total size of all objects of P_1 and P_2 is less than the page size)
      begin
        Move all objects from P_2 to P_1;
        Remove P_2 from *partitions*;
      end if;
    end foreach;
    return partitions;
end ggp;

The first step of this algorithms consists of assigning each object to a new (separate) partition. All these partitions are maintained in the variable *partitions*. Then, for all pairs of objects (o_1, o_2) in the object net with at least one reference between them, a tuple (o_1, o_2, w_{o_1,o_2}) is inserted into the list *edgeList* where w_{o_1,o_2} is the accumulated weight of all references between o_1 and o_2. All tuples of *edgeList* are visited in the order of descending weights. Let (o_1, o_2, w_{o_1,o_2}) be the current tuple. Let P_1, P_2 be the partitions to which the objects o_1 and o_2 are assigned. If $P_1 \neq P_2$ and the total size of all objects assigned to P_1 and P_2 is less than the page size, the two partitions are joined.[4] Otherwise, the edge is merely discarded—and the partitions remain invariant. For e being the number of edges in the clustering graph, the complexity of the greedy graph partitioning algorithm is $O(e \log e)$. For further improvements of this heuristic see [GKKM93].

5.3 Discussion

There exist a few possibilities for the time of object placement:

- objects are stored at their creation time,
- objects are stored at the end of the creating transaction, or
- reclustering occurs at an arbitrary point in time.

If an object is stored immediately at its creation time, an object can only be stored (clustered) with objects that have already been created prior to it. Further, there exist no references from other objects to it. Hence, the clustering strategies described in this section do not apply. In order to have some control over object placement, user hints are implemented, that is, the user may specify an (existing) object close to which the newly created object is to be placed. Closeness is measured as being in the same page or not. Of course, this approach puts the burden of good object placement on the shoulders of the programmer. It has been implemented in, for example, Exodus [CDRS86], ObjectStore [LLOW91], and ORION [Kim90].

If objects are placed at the end of a creating transaction, objects cannot only be stored together with objects created before them, but also with objects created later within the same transaction. Hence, possibly a whole set of objects, most likely referencing each other, is to be placed at the end of a transaction. Further, it is also likely that the creating transaction will not only create new objects but also assign values to their attributes and establish references between the newly created and existing objects and vice versa. In this case, any of the clustering strategies can either be restricted to the newly created objects, or involve already created objects also. The former case is easier to handle. As soon as already created objects come into play, decisions have to be made which of these objects are taken into consideration for (re-) clustering. Thus, the problems of reclustering (see below) have to be faced (at each commit time). This might impose an unacceptable run time penalty. The only practical choice is to apply a clustering strategy of low run-time complexity to only the newly created objects. One choice is to apply *placement trees* as implemented in O_2 [BD90].

Reclustering is mainly a question of cost. Since reclustering should take place only if all the objects to be considered for the new object placement exist, any clustering strategy can be applied. It could be up to the database administrator (DBA) to select the best time for reclustering and the best strategy for the given application. The DBA should consider the clustering performance and the run-time overhead of the different clustering strategies. But there is another crucial point:

[4] Partitions are represented as binary trees to accelerate the join operation.

if the system relies on logical object identifiers (i.e., object identifiers which are independent of the actual physical location of the objects), there is no need to readjust the references to objects moved during reclustering. This is not true for physical object identifiers, where the actual address of the object is encoded in the identifier. Here, all references to the objects that are re-placed must be adjusted. This can be prohibitively expensive. In the worst case, the whole object base has to be searched for references to be updated. In case backward references exist, this might be affordable.

Sequence-based clustering strategies are used in several object base systems such as Cactis [HK89], Encore [HZ87], and ORION [BKKG88]. To our knowledge, partition-based clustering strategies have not (yet) been incorporated into any system. Partition-based clustering was first introduced in [TN91]. There the Kernighan&Lin graph partitioning algorithm is applied [KL70]. Further investigations can be found in [TN92]. Several greedy graph partitioning algorithms were introduced in [GKKM93] and compared with other variants of graph partitioning algorithms.

6 Object Storage Models: Decomposition and Replication

In this section we will sketch different storage models for object bases. The goal, of course, is to expedite access to the relevant parts of an object base while, at the same time, keeping the overhead for updates tolerable. The subsequent discussion of three alternative storage representations is largely based on [VKC86]. A quantitative evaluation of some of the proposed storage schemes for one (benchmark) application can be found in [TRSB93].

6.1 NSM: N-ary Storage Model

This storage model was assumed in the graphical representation of our example object base *Company* in Figure 2. In this storage model, objects are considered as N-ary tuples where the tuples comprise *all* the attributes constituting the structural representation of the object. Note that this includes the structural representation of inheritance—as illustrated for objects of type *Manager* in Figure 2.

It may be tempting to map complete type extensions into a single, dedicated file. However, this may violate the clustering objective.

6.2 DSM: Decomposition Storage Model

While the NSM scheme "bundles" entire objects, the DSM model [CK85] decomposes an object such that each attribute value is stored separately. Thus, the storage model consists of triples

$$(OID, attribute, value)$$

where the value is either an atomic value or the OID of the referenced object. In terms of our example object base extension, (parts of) the storage representation looks as follows:

$$
\begin{array}{l}
(id_0, \textit{duration}, 12) \\
(id_0, \textit{competence}, 10) \\
\quad \vdots \\
(id_4, \textit{name}, \text{“\textit{Sander}”}) \\
(id_4, \textit{worksIn}, id_8) \\
(id_4, \textit{salary}, 90000) \\
(id_4, \textit{sex}, \text{‘\textit{F}’}) \\
(id_4, \textit{jobHistory}, id_0) \\
(id_4, \textit{jobHistory}, id_1) \\
\quad \vdots
\end{array}
$$

In this normalized representation, even the set-valued attribute *jobHistory* is flattened by "unnesting" the set containing (references to) *Task* objects.

Except for distributed database applications this representation scheme appears to be impractical. Therefore, a hybrid of the two aforementioned representations is described in the following section.

6.3 Fragmentation and Replication

In [VKC86] the P-DSM (partial DSM) storage scheme is proposed as a hybrid between the decomposition (DSM) and the N-ary (NSM) storage schemes. Under P-DSM, objects are (vertically) fragmented based on attrribute affinities, such that attributes which are frequently used together are stored in the same fragment. In the relational context there exists a large body of work on deriving these affinities from the work load (e.g., [HS75, NCWJ84]).

The P-DSM model naturally leads to replication when attributes are incorporated in more than one fragment of an object. Of course, replication incurs an additional update penalty.

A sample fragmentation with selective replication for some objects of our *Company* object base is shown below:

$$
\boxed{\begin{array}{l}
(id_4, \textit{name}, \text{“\textit{Sander}”}) \\
(id_4, \textit{worksIn}, id_8) \\
(id_4, \textit{salary}, 90000) \\
(id_4, \textit{jobHistory}, \{id_0, id_1\})
\end{array}}
\quad
\boxed{\begin{array}{l}
(id_4, \textit{name}, \text{“\textit{Sander}”}) \\
(id_4, \textit{salary}, 90000) \\
(id_4, \textit{sex}, \text{‘\textit{F}’})
\end{array}}
$$

$$
\boxed{\begin{array}{l}
(id_{11}, \textit{name}, \text{“\textit{Boss}”}) \\
(id_{11}, \textit{worksIn}, id_8) \\
(id_{11}, \textit{salary}, 150000) \\
(id_{11}, \textit{jobHistory}, \{\ldots\})
\end{array}}
\quad
\boxed{\begin{array}{l}
(id_{11}, \textit{name}, \text{“\textit{Boss}”}) \\
(id_{11}, \textit{salary}, 150000) \\
(id_{11}, \textit{sex}, \text{‘\textit{M}’})
\end{array}}
\quad
\boxed{\begin{array}{l}
(id_{11}, \textit{name}, \text{“\textit{Boss}”}) \\
(id_{11}, \textit{backUp}, id_5)
\end{array}}
$$

Here, the *Emp* object id_4 is fragmented into two (partially overlapping) fragments while the *Manager* object id_{11} is split into three overlapping fragments. The fragments are presumably derived according to some attribute affinities obtained from the object base usage profile.

A natural fragmentation boundary is given by the subtype/supertype hierarchy. One could bundle all attributes defined in one type of the inheritance hierarchy within a separate fragment. Then, a CEO instance would be modeled by four fragments: one comprising the attributes defined in *Person*, one for *Emp*, one for *Manager*, and one for attributes specific to a *CEO* object.

7 Future Research

In this chapter we described several physical object management techniques that have been proposed to optimize object base applications. While each one of these techniques is fairly well understood, their interaction in one system still requires research. For example, indexes and clustering are affected by updates. Either they are totally invalidated (i.e., indexes and materialized functions) or they exhibit a lower effectiveness (i.e., clustering). Nevertheless, upt to now each technique has its own dedicated way to detect and handle updates. A better way would be to factor out the detection/invalidation mechanism and develop a general mechanism which can then be directly supported by lower levels of the object base system—hopefully resulting in better performance.

From the discussion in this chapter it should be clear that each of the techniques requires tuning; the task of tuning a physical object base when these techniques are combined is much more difficult. Therefore, tools need to be developed for aiding the database administrator in the task of deriving an effective physical object base design for a given usage profile. One of the prerequisites for such a tool is the ability to characterize the application behavior by monitoring, sampling or (static) application analysis. Another requirement is a comprehensive cost model which takes the interdependencies of the various physical object management techniques into account.

Given the fact that many advanced applications do not rely on ad-hoc usage but, instead, access and manipulate the database by predefined programs, one might infer that the profile of these applications can be very specific. As an example consider production in manufacturing where the production process is well-organized, deterministic and imposes real-time requirements. Given its deterministic nature the profile is very specificic. Given the real-time requirements the database system must be very efficient for the given profile. Since performance is highly influenced by the physical organization of the data, the physical organization should be very specifically tailored for the profile. Hence, for a given logical schema and profile it should be possible to explicitly define a fine grained physical schema. This has several implications: First, the physical data structures must be much more specifically tailored for specific application profiles than in traditional applications. Being provocative one could state that today's storage structures and indexes are broad-band techniques applicable to a broad range of applications. Developing finer grained techniques implies the necessity of refining benchmarking techniques. Second, languages are needed that allow to specify the physical schema at a grain required by the application and supported by the storage structures. Third, if for example the production phase is reorganized, the physical database design must be reorganized as well. Hence, reorganization issues must be investigated more thoroughly. The requirements induced by reorganization needs become much harder. Considering manufacturing, the production phase runs 24 hours a day 7 days a week. For these non-stop applications with real-time constraints, incremental reorganization becomes a hard issue.

Acknowledgement

This work was supported in part by the German Research Council (DFG) under contracts Ke 401/6-1 and SFB 346.

References

[BCL89] J. A. Blakeley, N. Coburn, and P. A. Larson. Updating derived relations: Detecting irrelevant and autonomously computable updates. *ACM Trans. on Database Systems*, 14(3):369–400, Sep 89.

[BD90] V. Benzaken and C. Delobel. Enhancing performance in a persistent object store: Clustering strategies in O_2. In A. Dearle, G. Shaw, and S. Zdonik, editors, *Implementing Persistent Object Bases*, pages 403–412, Martha's Vineyard, Sep 1990. Morgan Kaufmann.

[BK89] E. Bertino and W. Kim. Indexing techniques for queries on nested objects. *IEEE Trans. Knowledge and Data Engineering*, 1(2):196–214, Jun 1989.

[BKKG88] J. Banerjee, W. Kim, S. J. Kim, and J. F. Garza. Clustering a DAG for CAD databases. *IEEE Trans. Software Eng.*, 14(11):1684–1699, Nov 1988.

[CDRS86] M. Carey, D. DeWitt, J. Richardson, and E. Shekita. Object and file management in the EXODUS extensible database system. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 91–100, Kyoto, Japan, Aug 86.

[CK85] G. Copeland and S. Khoshafian. A decomposition storage model. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 268–279, Austin, TX, May 1985.

[GKKM93] C. Gerlhof, A. Kemper, C. Kilger, and G. Moerkotte. Partition-based clustering in object bases: From theory to practice. In *Proc. of the Intl. Conf. on Foundations of Data Organization and Algorithms (FODO)*, Chicago, Illinois, Oct 1993.

[Här78] T. Härder. Implementing a generalized access path structure for a relational database system. *ACM Trans. on Database Systems*, 3(3):285–298, Sep 1978.

[HK89] S. E. Hudson and R. King. Cactis: A self-adaptive, concurrent implementation of an object-oriented database management system. *ACM Trans. on Database Systems*, 14(3):291–321, Sep 1989.

[HS75] J. A. Hoffer and D. G. Severance. The use of cluster analysis in physical database design. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, 1975.

[HZ87] M. Hornick and S. Zdonik. A shared, segmented memory system for an object-oriented database. *ACM Trans. Office Inf. Syst.*, 5(1):70–95, Jan 1987.

[Jhi88] A. Jhingran. A performance study of query optimization algorithms on a database system supporting procedures. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 88–99, L.A., CA, Sep 1988.

[Kim90] W. Kim. *Introduction to Object-Oriented Databases.* The MIT Press, Cambridge, MA, 1990.

[KKD89] W. Kim, K. C. Kim, and A. Dale. Indexing techniques for object-oriented databases. In W. Kim and F. H. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*, pages 371–394, Reading, MA, 1989. Addison Wesley.

[KKM94] A. Kemper, C. Kilger, and G. Moerkotte. Materialization of Functions in Object Bases: Design, Implementation and Assessment. *IEEE Trans. Knowledge and Data Engineering*, 1994 (in print; an excerpt appeared in: *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 258–268, May 1991)

[KKS92] M. Kifer, W. Kim, and Y. Sagiv. Querying object-oriented databases. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 393–402. acmpress, Jun 1992.

[KL70] B. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal*, 49(2):291–307, Feb 1970.

[KM90] A. Kemper and G. Moerkotte. Access support in object bases. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 364–374, Atlantic City, USA, May 1990.

[KM92] A. Kemper and G. Moerkotte. Access Support Relations: an indexing method for object bases. *Information Systems*, 17(2):117–146, 1992.

[KM93] A. Kemper and G. Moerkotte. *Object-Oriented Database Management: Applications in Engineering and Computer Science.* Prentice Hall, Englewood Cliffs, N.J., 1993. (in print).

[KMP93] A. Kemper, G. Moerkotte, and K. Peithner. A blackboard architecture for query optimization in object bases. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, Dublin, Ireland, Aug 1993.

[LLOW91] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The ObjectStore database system. *Communications of the ACM*, 34(10):50–63, 1991.

[LOL92] C. C. Low, B. C. Ooi, and H. Lu. H-trees: A dynamic associative search index for OODB. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 134–143, Jun 1992.

[Lum70] V. Y. Lum. Multi-attribute retrieval with combined indexes. *Communications of the ACM*, 13:660–665, 1970.

[MS86] D. Maier and J. Stein. Indexing in an object-oriented DBMS. In K. R. Dittrich and U. Dayal, editors, *Proc. IEEE Intl. Workshop on Object-Oriented Database Systems, Asilomar, Pacific Grove, CA*, pages 171–182. IEEE Computer Society Press, Sep 1986.

[NCWJ84] S. Navathe, S. Ceri, G. Wiederhold, and D. Jinglie. Vertical partitioning algorithms for database design. *ACM Trans. on Database Systems*, 9(4):680–710, Dec 1984.

[NHS84] J. Nievergelt, H. Hinterberger, and K. C. Sevcik. The grid file: An adaptable, symmetric multikey file structure. *ACM Trans. on Database Systems*, 9(1):38–71, 1984.

[SAH87] M. Stonebraker, J. Anton, and E. Hanson. Extending a database system with procedures. *ACM Trans. Database Systems*, 12(3):350–376, Sep 1987.

[SC89] E. J. Shekita and M. J. Carey. Performance enhancement through replication in an object-oriented DBMS. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 325–336, Portland, OR, May 1989.

[Sel88] T. K. Sellis. Intelligent caching and indexing techniques for relational database systems. *Information Systems*, 13(2):175–186, 1988.

[TN91] M. M. Tsangaris and J. F. Naughton. A stochastic approach for clustering in object bases. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 12–21, Denver, CO, May 1991.

[TN92] M. M. Tsangaris and J. F. Naughton. On the performance of object clustering techniques. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 144–153, San Diego, CA, Jun 1992.

[TRSB93] W. B. Teeuw, C. Rich, M. H. Scholl, and H. M. Blanken. An evaluation of physical disk I/Os for complex object processing. In *Proc. IEEE Conf. on Data Engineering*, pages 363–372, Vienna, Austria, Apr 1993.

[Val87] P. Valduriez. Join indices. *ACM Trans. on Database Systems*, 12(2):218–246, Jun 1987.

[VKC86] P. Valduriez, S. Khoshafian, and G. Copeland. Implementation techniques of complex objects. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 101–110, Kyoto, Japan, Aug 1986.

[ZM89] S. Zdonik and D. Maier. Fundamentals of object-oriented databases. In S. Zdonik and D. Maier, editors, *Readings in Object-Oriented Databases*, pages 1–32. Morgan-Kaufmann Publ. Co., 1989.