RT2R2
A Survey of Indexing Techniques

Pastel
Working Group R2

July 22, 1998
Contents

1 Indexing in Object-Oriented Databases .......................................................... 3
  1.1 Access Support Relations: Indexing Path Expressions ................................. 3
    1.1.1 Auxiliary Definitions ............................................................................ 3
    1.1.2 Extensions of Access Support Relations .............................................. 6
    1.1.3 Storage Structure .................................................................................. 7
    1.1.4 Join Index Hierarchies ......................................................................... 9
    1.1.5 Related Work ....................................................................................... 11
  1.2 Function Materialization ............................................................................. 12
    1.2.1 Storing Materialized Results .................................................................. 12
    1.2.2 Retrieval of Materialized Results ......................................................... 14
    1.2.3 Storage Representation of GMRs ............................................................ 15
    1.2.4 Invalidation and Rematerialization of Function Results ......................... 15
    1.2.5 Strategies to Reduce the Invalidation Overhead ..................................... 16
    1.2.6 Related Work ....................................................................................... 17
  1.3 Indexing over Type Hierarchies ................................................................... 18
    1.3.1 Single Type Indexing ............................................................................ 19
    1.3.2 Type Hierarchy Indexing ...................................................................... 19
    1.3.3 The CG Tree ......................................................................................... 20
  1.4 Navigation Index for Object Bases ............................................................... 22

2 Access Methods for Multimedia Databases ......................................................... 27
  2.1 Current Trends in the Similarity Retrieval Indexing .................................... 27
    2.1.1 Adjusting New Problems to Fit Existing Tools ....................................... 28
    2.1.2 Developing New Techniques .................................................................. 29
  2.2 Current Research Activities by PASTEL Partners ........................................ 32
    2.2.1 Vector-Approximation File: VA-File ..................................................... 32
    2.2.2 Metric Tree: M-tree .............................................................................. 35

3 Generalized Indexing: Adding Value to Heterogeneous, Distributed Resource Collections ................................................................................................. 39
  3.1 A Generalized Approach to Indexing ............................................................. 40
  3.2 A Model of Generalized Indexing .................................................................. 40
  3.3 A Formal View on Generalized Indexing ...................................................... 42
  3.4 Specializing the Generic Model .................................................................... 43
    3.4.1 BibTeX Files ......................................................................................... 43
    3.4.2 The PI-Index ....................................................................................... 43
  3.5 A Service Framework for Generalized Indexing .......................................... 44
Chapter 1

Indexing in Object-Oriented Databases

In this and the next section we will use the object base shown in Figure 1.1 for illustration for the illustration of new indexing techniques in object-oriented database systems.

1.1 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. A lot of research effort has been spent on expediting the join, e.g., access structures to support the join, the sort-merge join, and the hash-join algorithm were developed. Recently, the binary join index structure [1] building on links [2] was designed as another optimization method for this operation.

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 join [3] or implicit join [4].

In Section ?? we discussed techniques for evaluating such functional joins. In this section we present a very general indexing structure, called Access Support Relations (ASRs), which is designed to support the functional join along arbitrary long attribute chains where the chain may even contain collection-valued attributes. The ASRs allow to avoid the actual evaluation of the functional joins by materializing frequently traversed reference chains.

1.1.1 Auxiliary Definitions

A path expression has the form

\[ o.A_1, \ldots, A_n \]

where \( o \) is a tuple structured object containing the attribute \( A_1 \) and \( o.A_1, \ldots, A_i \) 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 $R_n$, which is recursively defined as follows:

$$
R_0 := \{o\}
$$

$$
R_i := \bigcup_{v \in R_{i-1}} v.A_i \quad \text{for } 1 \leq i \leq n
$$

Thus, $R_n$ is a set of OIDs of objects of type $t_n$ or a set of atomic values of type $t_n$ if $t_n$ is an atomic data type, such as $\text{int}$.

It is also possible that the path expression originates in a collection $C$ of tuple-structured objects, i.e., $C.A_1.\cdots.A_n$. Then the definition of the set $R_0$ has to be revised to: $R_0 := C$.

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

**Definition 1.1.1 (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 $t_i$.

\[1\] This means that the attribute $A_i$ can be associated with objects of type $t_i$ or any subtype thereof.
• The type \( t_{i-1} \) is defined as type \( t_{i-1} \) is \([\ldots, \{A_i\}, \ldots]\), 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 the presentation we assumed that the involved types are not being defined as a subtype of some other type. This, of course, is generally possible; it would only make the definition a bit more complex to read.

The second part of the definition is useful to support access paths through sets\(^2\). 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 1.1.2 (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 \( \llbracket t_0.A_1.\cdots.A_n \rrbracket \) is of arity \( n+1 \) and has the following form:

\[
\llbracket t_0.A_1.\cdots.A_n \rrbracket : [S_0, \ldots, S_n]
\]

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 \( \llbracket t_0.A_1 \rrbracket, \ldots, \llbracket t_{n-1}.A_n \rrbracket \).

**Definition 1.1.3 (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 \( \llbracket t_{i-1}.A_i \rrbracket \). The relation \( \llbracket t_{i-1}.A_i \rrbracket \) contains the tuples \((\text{id}(o_{i-1}), \text{id}(o_i))\) 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 \( \text{id}(o_n) \) 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 illustrate this on an example of our University schema:

\[
P \equiv \text{Students}.\text{takenExams}.\text{givenBy}.\text{Name}
\]

\[\begin{array}{c}
\text{Exams} \\
\text{Professors} \\
\text{string}
\end{array}\]

\[^2\text{Note, however, that we do not permit powersets.}\]
The type constraints of the path expression are emphasized with the underbraces. When considering the update problem, it should be obvious that strong typing—as enforced by the ODMG model—is vital to indexing over path expressions. Therefore, models with a more relaxed typing paradigm such as, e.g., GemStone, which is based on the dynamically typed Smalltalk, 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:


table

<table>
<thead>
<tr>
<th>Students.takenExams</th>
<th>Exams.givenBy</th>
</tr>
</thead>
<tbody>
<tr>
<td>OID_{Students}</td>
<td>OID_{Exams}</td>
</tr>
<tr>
<td>id_{35}</td>
<td>id_{21}</td>
</tr>
<tr>
<td>id_{35}</td>
<td>id_{23}</td>
</tr>
<tr>
<td>id_{37}</td>
<td>id_{22}</td>
</tr>
<tr>
<td>id_{37}</td>
<td>id_{27}</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>[Professors.Name]</th>
</tr>
</thead>
<tbody>
<tr>
<td>OID_{Professors}</td>
</tr>
<tr>
<td>id_{1}</td>
</tr>
<tr>
<td>id_{5}</td>
</tr>
<tr>
<td>id_{6}</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

1.1.2 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, i.e., 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 1.1.4 (Extensions)** Let \(\Join (\Join, \Join, \Join)\) denote the natural (outer, left outer, right outer) join on the last column of the first relation and the first column

6
of the second relation. Then the different extensions are obtained as follows:

\[
\begin{align*}
[t_0.A_1 \ldots A_n]_{can} & := [t_0.A_1] \times \cdots \times [t_{n-1}.A_n] \\
[t_0.A_1 \ldots A_n]_{ful} & := [t_0.A_1] \times \cdots \times [t_{n-1}.A_n] \\
[t_0.A_1 \ldots A_n]_{left} & := (\cdots ([t_0.A_1] \times [t_1.A_2]) \cdots \times [t_{n-1}.A_n]) \\
[t_0.A_1 \ldots A_n]_{right} & := ([t_0.A_1] \times \cdots ([t_{n-2}.A_{n-1}] \times [t_{n-1}.A_n]) \cdots )
\end{align*}
\]

The full extension of \([Students.takenExams.givenBy.Name]_{ful}\) looks as follows:

<table>
<thead>
<tr>
<th>(S_0 : OID_{Students})</th>
<th>(S_1 : OID_{Exams})</th>
<th>(S_2 : OID_{Professors})</th>
<th>(S_3 : string)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(id_{35})</td>
<td>(id_{21})</td>
<td>(id_1)</td>
<td>“Knuth”</td>
</tr>
<tr>
<td>(id_{35})</td>
<td>(id_{23})</td>
<td>(id_6)</td>
<td>“Babbage”</td>
</tr>
<tr>
<td>(id_{37})</td>
<td>(id_{22})</td>
<td>(id_1)</td>
<td>“Knuth”</td>
</tr>
<tr>
<td>(id_{37})</td>
<td>(id_{27})</td>
<td>(id_6)</td>
<td>“Babbage”</td>
</tr>
<tr>
<td>—</td>
<td>—</td>
<td>(id_5)</td>
<td>“Turing”</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

This extension contains all paths and subpaths corresponding to the underlying path expression. The first four tuples actually constitute complete paths which would be present in the canonical extension as well; however the fifth path would be omitted in the canonical extension. In the left-complete extension the only he first four tuples would be present, whereas the fifth tuple would also be present in the the right-complete extension.

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 next definition states under what conditions an existing access support relation can be utilized to evaluate a path expression that originates in an object (or a set of objects) of type \(s\).

**Definition 1.1.5 (Applicability)** An access support relation \([t_0.A_1 \ldots A_n]_X\) under extension \(X\) is applicable for a path \(s.A_i \ldots A_j\) where \(s\) is a subtype\(^3\) of \(t_{i-1}\) under the following condition, depending on the extension \(X\):

\[
\text{Applicable}([t_0.A_1 \ldots A_n]_X, s.A_i \ldots A_j) = \begin{cases} 
X = \text{full} & \land 1 \leq i \leq j \leq n \\
X = \text{left} & \land 1 = i \leq j \leq n \\
X = \text{right} & \land 1 \leq i \leq j = n \\
X = \text{can} & \land 1 = i \leq j = n
\end{cases}
\]

### 1.1.3 Storage Structure

The storage structure of access support relations is borrowed from the binary join index proposal by Valduriez [1]. Each ASR is redundantly stored in two index

\(^3\)Note, that every type is a subtype of itself.
structures: the first being keyed on the left-most attribute and the second being keyed 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 since only exact match queries have to be supported. On the other hand, B+-trees are advantageous for attributes that allow range queries, e.g., int and float values. Note, that these attributes can only occur at the right-most column of an ASR. 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 \([\text{Students.takenExams.givenBy.Name}]_{\text{can}}\) as follows:

\[
\begin{array}{|c|c|c|c|}
\hline
S_0 : \text{OID}_{\text{Students}} & S_1 : \text{OID}_{\text{Exams}} & S_2 : \text{OID}_{\text{Professors}} & S_3 : \text{string} \\
\hline
\text{id}_{35} & \text{id}_{21} & \text{id}_1 & \text{“Knuth”} \\
\text{id}_{35} & \text{id}_{23} & \text{id}_6 & \text{“Babbage”} \\
\text{id}_{37} & \text{id}_{22} & \text{id}_1 & \text{“Knuth”} \\
\text{id}_{37} & \text{id}_{27} & \text{id}_6 & \text{“Babbage”} \\
\ldots & \ldots & \ldots & \ldots \\
\hline
\end{array}
\]

We will call the left B+-tree the “forward clustered” tree, and, analogously, the right one the “backward clustered” tree. The left-hand B+-tree supports the evaluation of a forward query, e.g., retrieving the professor’s name who has examined the student identified by \text{id}_{35}. The left-hand B+-tree supports the evaluation of backward queries—with respect to the underlying path expression. For our example, an entry point for finding the Students who have taken exams from “Knuth” 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 “connecting” access support relations. Again, let us graphically visualize the situation:

\[
\begin{array}{|c|c|c|}
\hline
S_0 : \text{OID}_{\text{Students}} & S_1 : \text{OID}_{\text{Exams}} & S_2 : \text{OID}_{\text{Professors}} \\
\hline
\text{id}_{35} & \text{id}_{21} & \text{id}_1 \\
\text{id}_{35} & \text{id}_{23} & \text{id}_6 \\
\text{id}_{37} & \text{id}_{22} & \text{id}_1 \\
\text{id}_{37} & \text{id}_{27} & \text{id}_6 \\
\ldots & \ldots & \ldots \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|}
\hline
S_0 : \text{OID}_{\text{Professors}} & S_1 : \text{string} \\
\hline
\text{id}_6 & \text{“Babbage”} \\
\text{id}_1 & \text{“Knuth”} \\
\text{id}_6 & \text{“Turing”} \\
\ldots & \ldots \\
\hline
\end{array}
\]

8
The above example illustrates the virtues of the redundant storage model for ASRs. The right B⁺-tree of the ASR \([\text{Professors.Name}]_{\text{can}}\) directly supports the lookup of those \textit{Professors} whose \textit{Name} is \textit{Knuth}, i.e., the one with OID \textit{id} in our example. Then, the right B⁺-tree of the ASR \([\text{Students.takenExams.givenBy}]_{\text{can}}\) supports the traversal to the corresponding \textit{Students} to obtain the result \{\textit{id}_{35}, \textit{id}_{37}\}.

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

\[
\Pi_{s_0} \left( [\text{Students.takenExams.givenBy}]_{\text{can}} \times \left( \sigma_{s_{1}=\text{Knuth}} [\text{Professors.Name}]_{\text{can}} \right) \right)
\]

Analogously, the “forward clustered” B⁺-tree supports the semi-join from left to right, such that, for instance, the \textit{Names} of \textit{Professors} who have examined student \textit{id}_{35} can be retrieved efficiently. This corresponds to the “left-to-right” semi-join across ASRs:

\[
\Pi_{s_3} \left( \left( \sigma_{s_{0}=\text{id}_{35}} [\text{Students.takenExams.givenBy}]_{\text{can}} \right) \times [\text{Professors.Name}]_{\text{can}} \right)
\]

### 1.1.4 Join Index Hierarchies

Recently, [5] have adapted the ASR scheme to a so-called join index hierarchy. Their key idea is to omit the intermediate objects in the join index and merely store the OIDs of the start and the target object. In addition, the number of possible paths between the start and target object is counted and stored.

Their approach is still based on the binary access support relations \([t_0.A_1], \ldots, [t_{n-1}.A_n]\). A join indexing covering the sub-path from \(t_i\) to \(t_j\)—denoted \(\text{JI}(t_i.A_{i+1} \cdots A_j)\)—is obtained from the binary ASRs as follows:

\[
\text{JI}(t_i.A_{i+1} \cdots A_j) := [t_i.A_{i+1}] \bowtie_c [t_{i+1}.A_{i+2}] \bowtie_c \ldots \bowtie_c [t_{j-1}.A_j]
\]

The \(\bowtie_c\) operation is similar to a conventional natural join except that it projects on the OIDs of the start and target objects and it properly derives the counts of the number of different paths leading from start to target object and eliminates duplicates with the same start and target by aggregating the count values.⁴

<table>
<thead>
<tr>
<th>(\text{JI}(t_i.A_{i+1} \cdots A_k))</th>
<th>(\text{JI}(t_k.A_{k+1} \cdots A_p))</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{OID}_{t_i})</td>
<td>(\text{OID}_{t_k})</td>
</tr>
<tr>
<td>(id_{57})</td>
<td>(id_{67})</td>
</tr>
<tr>
<td>(id_{57})</td>
<td>(id_{78})</td>
</tr>
<tr>
<td>(id_{59})</td>
<td>(id_{78})</td>
</tr>
<tr>
<td>(id_{57})</td>
<td>(id_{88})</td>
</tr>
<tr>
<td>(id_{59})</td>
<td>(id_{88})</td>
</tr>
</tbody>
</table>

Let us briefly explain the derivation of the first tuple \([id_{57}, id_{88}, 17]\). There are 3 paths connecting \(id_{57}\) with \(id_{67}\) and 3 paths connecting \(id_{67}\) with \(id_{88}\). Therefore, there are \(3 \times 3 = 9\) different ways to traverse from \(id_{57}\) to \(id_{88}\) via \(id_{67}\). Likewise,

---

⁴For simplicity we assume that the binary ASRs are augmented with a count attribute which is set to 1 in all tuples.
there are $2 \times 4 = 8$ different ways to traverse from $id_{57}$ to $id_{88}$ via $id_{78}$. This amounts to $9 + 8 = 17$ different paths between $id_{57}$ and $id_{88}$.

For our university database the join index $JI(Students.takenExams.givenBy.Name)$ looks as follows:

![Diagram](image)

Figure 1.2: The Complete Join Index Hierarchy for a Path of Length 4

Note that join indices are always ternary relations—no matter how long a path expression they cover, because intermediate objects are omitted. In our case, the join index does not contain less tuples than the canonical ASR because none of the students in our example database has taken two (or more) exams from the same professor.

Maintaining just one join index covering the entire path expression is usually not sufficient because it allows to evaluate only those queries that span the entire path. The other extreme is to materialize all the possible join indices that cover any one of the subpaths. This results in precomputing (and maintaining) the so-called complete join index hierarchy. For the abstract path expression $t_0.A_1.A_2.A_3.A_4$ the complete join index hierarchy is shown in Figure 1.2.

The disadvantage of the complete join index hierarchy is that materializing all the possible join indices leads to high storage and high update costs. This is especially disadvantageous if some of the join indices are rarely/never used for query processing.

Assuming that most queries traverse from $t_0$ to $t_4$ (or vice versa), from $t_1$ to $t_4$, from $t_2$ to $t_4$, or from $t_0$ to $t_2$ then the partial join index hierarchy shown in Figure 1.3 is most appropriate.

**Updating a Join Index Hierarchy** Updates on the object base have to be properly propagated to all materialized join indices that span the corresponding type. Consider the partial join index hierarchy of Figure 1.3: An update of attribute $A_2$ of an object $o_1 \in t_1$ has to be propagated to the binary ASR (join index) $[t_1.A_2]$ and to the three join indices $JI(t_0.A_1.A_2)$, $JI(t_1.A_2.A_3.A_4)$, and $JI(t_0.A_1.A_2.A_3.A_4)$. Such updates are propagated step-wise from the bottom to the top of the join index.
hierarchy. For this purpose we observe the following rule. Let $\Delta \mathbf{JI}(t_i.A_{i+1},\ldots.A_k)$ denote a set of tuples inserted into join index $\mathbf{JI}(t_i.A_{i+1},\ldots.A_k)$. Then, the join index $\mathbf{JI}(t_i.A_{i+1},\ldots.A_k,\ldots.A_p)$ is properly updated by inserting the tuples $\Delta \mathbf{JI}(t_i.A_{i+1},\ldots.A_k) \cup_c \mathbf{JI}(t_i.A_{k+1},\ldots.A_p)$.

An analogous rule holds if the tuples were inserted into the join index spanning a suffix (instead of a prefix) of the encompassing path; e.g., if tuples were inserted into $\mathbf{JI}(t_k.A_{k+1},\ldots.A_p)$. Inserting tuples into a join index has to be done with care: If a tuple $[id_i, id_j, n]$ exists in the relation and another tuple $[id_i, id_j, m]$ representing $m$ additional paths between $id_i$ and $id_j$ is inserted, the two should be combined to the one tuple $[id_i, id_j, (n + m)]$.

Let us illustrate this bottom-up update propagation on our partial join index hierarchy of Figure 1.3. Inserting the additional tuple(s) $\Delta [t_1.A_2]$ into $[t_1.A_2]$ is propagated to the other join indices as follows:

1. $\mathbf{JI}(t_0.A_1.A_2)$ is updated by inserting the tuples $\Delta \mathbf{JI}(t_0.A_1.A_2) := [t_0.A_1] \cup_c \Delta [t_1.A_2]$.

2. $\mathbf{JI}(t_1.A_2.A_3.A_4)$ is updated by inserting the tuples $\Delta [t_1.A_2] \cup_c \mathbf{JI}(t_2.A_3.A_4)$.

3. In updating $\mathbf{JI}(t_0.A_1.A_2.A_3.A_4)$ the set of new tuples $\Delta \mathbf{JI}(t_0.A_1.A_2)$ for join index $\mathbf{JI}(t_0.A_1.A_2)$ that was computed in step 1 is reused. $\mathbf{JI}(t_0.A_1.A_2.A_3.A_4)$ is updated by inserting the tuples $\Delta \mathbf{JI}(t_0.A_1.A_2) \cup_c \mathbf{JI}(t_2.A_3.A_4)$.

1.1.5 Related Work

Access support relations were developed in [6]. A more detailed description can be found in [7]. Access support relations constitute a generalization of two relational techniques: the links developed by Härder [2] and the binary join indices proposed by Valduriez [1]. Rather than relating only two relations (or object types) our technique allows to support 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 GemStone [8] 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 [9] are contained as a special case in our framework. [10] reports on an indexing technique for hierarchical object structures, i.e., nested relations, which is related to our access support relations.

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) replication of object values as proposed for the Extra object model [11] and for the POSTGRES model [12, 13] are largely subsumed by ASRs. The join index hierarchies were proposed by [5].

1.2 Function Materialization

Let us now discuss an optimization technique that is devised to expedite the evaluation of queries containing function invocations, such the following example query:

```
select s
from s in AllStudents
where s.gpa() > 3.0
```

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

1.2.1 Storing Materialized Results

There are two obvious locations where materialized results could possibly be stored: in or near the argument objects of the materialized function or in a separate data structure. Storing the results near the argument objects means that the argument and the function result are stored within the same page such that the access from the argument to the appropriate result requires no additional page access. In general, storing results near the argument objects has several disadvantages:
• If the materialized function \( f : t_1, \ldots, t_n \rightarrow t_{n+1} \) has more than one argument \((n > 1)\) one of the argument types must be designated to hold the materialized result. But this argument has to maintain the results of all argument combinations—which, in general, won’t fit on one page.

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

Therefore, [14] chose to store materialized results in a separate data structure dis-associated from the argument objects. If several functions are materialized which share all argument types, the results of these functions may be stored within the same data structure. This provides for more efficiency when evaluating queries that access results of several of these functions and, further, avoids to store the arguments redundantly. These thoughts lead to the following definition:

**Definition 1.2.1 (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 \( \langle f_1, \ldots, f_m \rangle \) for the functions \( f_1, \ldots, f_m \) is of arity \( n + 2 \times m \) and has the following form:

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

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 \( \langle f_1, \ldots, f_m \rangle \) is consistent if a \textit{true} validity indicator implies that the associated materialized result is currently valid, i.e.:

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

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

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

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

2. **Lazy rematerialization:** The invalidated function result is only marked as being invalid by setting the corresponding \( V_i \) attribute to \textit{false}. The rematerialization of invalidated results is carried out as soon as the load of the object base management system falls below a predetermined threshold or—at the latest—at the next time the function result is needed.
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 [14] we proposed restricted GMRs for materializing functions for selected parameters only.

An example of a GMR comprising only a single materialized function, i.e., Students.gpa is shown below:

<table>
<thead>
<tr>
<th>Students.gpa</th>
</tr>
</thead>
<tbody>
<tr>
<td>O₁ : OIDStudents</td>
</tr>
<tr>
<td>id₃₅</td>
</tr>
<tr>
<td>id₃₇</td>
</tr>
<tr>
<td>id₁₅</td>
</tr>
</tbody>
</table>

1.2.2 Retrieval of Materialized Results

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

Note that invalid or missing results need not necessarily be (re-)computed upon the evaluation of some query. For example, if any student having a gpa greater than 3.0 is to be retrieved (e.g., to serve as a tutor), and if such a student can be found by inspecting the (incomplete) GMR no invalidated or missing results need be (re-)computed.

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

**Definition 1.2.2 (Valid Extension)**

A consistent extension of the GMR \( \langle f₁, \ldots, f_m \rangle \) is called \( f_j \)-valid iff

\[
\pi_{V_j} \langle f₁, \ldots, f_m \rangle = \{ \text{true} \}
\]

**Definition 1.2.3 (Complete extension)**

A consistent extension of the GMR \( \langle f₁, \ldots, f_m \rangle \) is a complete extension iff

\[
\pi_{O₁, \ldots, O_n} \langle f₁, \ldots, f_m \rangle = \text{ext}(t₁) \times \ldots \times \text{ext}(t_n)
\]

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

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

$$\pi_{O_1} \left( \sigma_{gpa>3.0} \langle Students.gpa \rangle \right)$$

as long as the GMR $\langle Students.gpa \rangle$ is $gpa$-valid and complete.

1.2.3 Storage Representation of GMRs

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

$$MDS \begin{bmatrix} O_1 & \ldots & O_n & f_1 & \ldots & f_m & V_1 & \ldots & V_m \end{bmatrix}$$

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 at-
tributes of the records being stored in the MDS.

Instead of using multi-dimensional storage structures, such as the Grid-File [15],
one could also utilize more conventional indexing schemes to expedite the 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 based on regular B-
trees is given in an early paper by V. Lum [16].

1.2.4 Invalidation and Rematerialization of Function Re-

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 general references are maintained only uni-directionally. In this case, 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
every 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:

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

15
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 \( o \) need not be one of the arguments \( o_1, \ldots, o_n; \) it could be some object related 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.\(^5\)

**Definition 1.2.4 (Reverse Reference Relation)**

The Reverse Reference Relation RRR is a set of tuples of the form

\[
[O : OID, F : FunctionId, A : \langle OID \rangle]
\]

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 \( r.A \). Remember, that the angle brackets \( \langle \ldots \rangle \) denote the list constructor.

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 University object base a part of the RRR that controls the invalidation of precomputed results in the GMR \( \langle Students.gpa \rangle \) is shown in Figure 1.4. Each time an object is updated in the object base, the RRR is inspected to find out which materialized results have to be invalidated (lazy rematerialization) or recomputed (immediate rematerialization). Reference [14] describes ways to detect object updates by schema modification and efficient algorithms for maintaining the RRR—which, of course, changes under object base updates.

### 1.2.5 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.

\(^5\)This holds only if no global variables are used by the materialized function. Otherwise, the RRR contains reverse references not only to the argument objects but also to the accessed global variables.
In [14], four complementary techniques to reduce the update penalty—consisting of invalidation and rematerialization—by better exploiting the potential of the object-oriented paradigm were 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 \textit{gpa} certainly does not depend on the \textit{Semester} and \textit{Name} attributes of a \textit{Student}.

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

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

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

\subsection*{1.2.6 Related Work}

The function materialization discussed here is—in its basic ideas—similar to materialization of views in the relational context. The most important work is reported in [17] and [18]. Lindsay proposed so-called relational \textit{snapshots} [19]—which, however, are not guaranteed consistent with the actual state of the database. The snapshots are only periodically recomputed and, thereby, brought back into a consistent state. Thus, a snapshot can only be used for certain applications, e.g., browsing, that do not require a completely consistent information contents.

Further work in precomputing queries and database procedures was done in the extended relational database project \textsc{POSTGRES} [20]. Here, the so-called “QUEL as a Datatype” attributes are precomputed and cached in separate data structures. The control concepts are discussed in [21, 22, 23, 13, 12, 24].

The function materialization concepts discussed in this chapter were developed in [25]; a more detailed discussion is given in [26].
1.3 Indexing over Type Hierarchies

For the discussion of this section consider the type hierarchy shown in Figure 1.5. Based on this type hierarchy we can phrase the following three queries, the meaning of which should be obvious:

\[
Q_1: \text{select } c \\
\text{from Emp } e \\
\text{where } e.\text{salary} > 200000
\]

\[
Q_2: \text{select } c \\
\text{from CEO } c \\
\text{where } c.\text{salary} > 200000
\]

\[
Q_3: \text{select } p \\
\text{from Person } p \\
\text{where } p.\text{age} > 60
\]

In query \(Q_1\) we want to retrieve \(\text{Emps}\) (including \(\text{Managers}\) and \(\text{CEOs}\)) whose \textit{salary} exceeds 200000. In query \(Q_2\), on the other hand, we are only interested in the \(\text{CEOs}\) with such a high \textit{salary}; because of substitutability the result of \(Q_2\) is a subset of the result obtained in query \(Q_1\). In query \(Q_3\) all \(\text{Persons}\) whose \textit{age} exceeds 60 are retrieved, i.e., including all \(\text{Emps}\), \(\text{Managers}\), \(\text{CEOs}\) and \(\text{Students}\).

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, the index for \(\textit{Emp.salary}\) which is represented as the ASR \(\llbracket \textit{Emp.salary} \rrbracket\) as follows—assuming that \(id_4, id_5, id_6,\) and \(id_7\) are \(\text{Emp}\) objects, \(id_{11}\) and \(id_{13}\) are \(\text{Managers}\) and \(id_{77}\) and \(id_{88}\) identify \(\text{CEOs}\):

\[
\begin{array}{|c|c|}
\hline
\text{Emp.salary} & \text{B}^+ \\
\hline
S_0 : OID_{Emp} & S_1 : \text{int} \\
\hline
id_4 & 90000 \\
id_5 & 100000 \\
id_7 & 100000 \\
id_{11} & 150000 \\
id_6 & 260000 \\
id_{13} & 900000 \\
id_{77} & 1500000 \\
id_{88} & 2000000 \\
\ldots & \ldots \\
\hline
\end{array}
\]

Note, that this index, because of substitutability, implicitly includes all \(\text{Managers}\) and all \(\text{CEOs}\), for which the \textit{salary} is “known”. Therefore, the evaluation of query \(Q_1\) is very well supported since it involves a lookup in a single \(B^+\)-tree only. However, query \(Q_2\) is not as well supported since it involves retrieving the OIDs from the
index \([\text{Emp.salary}]\) for which the salary attribute exceeds 200000. The resulting set, however, contains \textit{Emps} and \textit{Managers} as well. So, in a (costly) second phase, the \textit{CEOs} have to be extracted from this set.

### 1.3.1 Single Type Indexing

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

These indexes have the following form:

\[
\begin{array}{|c|c|}
\hline
S_0 : \text{OID}_\text{Emp} & S_1 : \text{int} \\
\hline
id_4 & 90000 \ \\
\hline
id_5 & 100000 \ \\
\hline
id_7 & 100000 \ \\
\hline
id_6 & 260000 \ \\
\hline
\end{array}
\quad
\begin{array}{|c|c|}
\hline
S_0 : \text{OID}_\text{Manager} & S_1 : \text{int} \\
\hline
id_{11} & 150000 \ \\
\hline
id_{13} & 900000 \ \\
\hline
... & ... \ \\
\hline
\end{array}
\quad
\begin{array}{|c|c|}
\hline
S_0 : \text{OID}_\text{CEO} & S_1 : \text{int} \\
\hline
id_{77} & 1500000 \ \\
\hline
id_{88} & 2000000 \ \\
\hline
... & ... \ \\
\hline
\end{array}
\]

Now, evaluating query \(Q_2\) is very well supported because it involves only a lookup in the index \([\text{CEO.salary}]\).

However, the evaluation of query \(Q_1\) now involves a lookup in three separate \(B^+\)-trees and unioning the results, i.e.:

\[
\sigma_{s_1>200000} [\text{Emp.salary}] \cup \sigma_{s_1>200000} [\text{Manager.salary}] \cup \sigma_{s_1>200000} [\text{CEO.salary}]
\]

This problem appears even more severe when considering query \(Q_3\) under the assumption of separate single type indexing on the \textit{age} attribute.

### 1.3.2 Type Hierarchy Indexing

Because of the above discussed disadvantages of separate single type indexing \cite{27} propose 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, as well; except that \cite{27} developed a special structure of the leaf nodes which is sketched as follows:

\[
\begin{array}{|c|c|c|c|c|c|c|c|c|}
\hline
\text{record length} & \text{key length} & \text{key value} & \text{overflow page} & \# \text{OIDs} & \text{key directory} & \{id_{1}, id_{2}, \ldots, id_{j}\} & r_1 & \ldots & r_n \\\n\hline
\end{array}
\]

\[
\begin{array}{|c|c|}
\hline
\# \text{types} & \text{type } T_1 \ \\
\hline
\end{array}
\quad
\begin{array}{|c|c|}
\hline
\text{offset} & \text{type } T_2 \ \\
\hline
\end{array}
\quad
\begin{array}{|c|c|}
\hline
\ldots & \text{type } T_n \ \\
\hline
\end{array}
\quad
\begin{array}{|c|c|}
\hline
\text{offset} & \ldots \ \\
\hline
\end{array}
\]

19
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.

[28] developed an indexing scheme, called $H$-trees, for combining type hierarchy indexing with single type indexing. The basic idea consists of nesting $B^+$-trees, i.e., nesting the index tree of a subtype within the tree of the super type. For our example this is graphically visualized in Figure 1.6, where the three $H$-trees $H_{\text{Emp}}$ for direct $\textit{Emp}$ instances, $H_{\text{Manager}}$ for direct $\textit{Manager}$ instances, and $H_{\text{CEO}}$ for $\textit{CEO}$ 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 $T_2$ must be contained in the range of $T_1$, and the range of $T_4$ must be contained in the range of $T_3$.

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.

### 1.3.3 The CG Tree

[29] pointed out the principal difference between a key grouping index—such as the CH-tree—and a set-grouping index—such as the H-tree. Figure 1.7 sketches
the relative performance of these two indexing schemes for exact match and range queries. The key grouping scheme has very good performance (i.e., low numbers of pages have to be read) for exact match queries whereas the set grouping scheme degenerates if many sets (i.e., many levels of a type hierarchy) have to be processed. This is due to the fact that basically every type extent is covered by a separate B-tree. On the other hand, the key grouping scheme shows poor performance for range queries because it has to process a large number of leaf pages. It cannot draw profit from a restriction on the number of sets (type extents) that should be processed because all sets’ objects are intermixed on the leaf pages.

Observing this principal difference, [29] designed the so-called CG-tree which combines the advantages of both schemes. The idea is to replace the leaf pages of a B+-tree by several linked lists, one for each set (type extent) being indexed. This basic idea is illustrated in Figure 1.8 for two sets (type extents) $s_1$ and $s_2$ only.

The linked lists of leaf pages are considered to be at level 1 of the tree. Then, at level 2 of the tree particularly structured so-called directory pages are needed that reference the pages at level 1. The directory pages have the following structure—for $n$ indexed sets $s_1, \ldots, s_n$:

<table>
<thead>
<tr>
<th>number of records</th>
<th>prev</th>
<th>next</th>
<th>$K_1$</th>
<th>$R_1$</th>
<th>$K_2$</th>
<th>$R_2$</th>
<th>$R_3$</th>
<th>$\ldots$</th>
<th>$K_m$</th>
<th>$R_m$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>$R_1$</td>
<td>$K_1$</td>
<td>$R_2$</td>
<td>$K_2$</td>
<td>$R_3$</td>
<td>$\ldots$</td>
<td>$K_m$</td>
<td>$R_m$</td>
</tr>
<tr>
<td>$R_1$</td>
<td>$s_1$</td>
<td>$\cdots$</td>
<td>$s_n$</td>
<td>$R_1$</td>
<td>$s_1$</td>
<td>$\cdots$</td>
<td>$s_n$</td>
<td>$R_2$</td>
<td>$s_1$</td>
<td>$\cdots$</td>
</tr>
</tbody>
</table>

The directory page contains $m$ search keys $K_1, \ldots, K_m$. Thereby, the $m$ ranges $R_1, \ldots, R_m$ are defined. For each range, the directory contains $n$ pointers to level
1 pages. The pointer $R_{i,s_j}$ refers to the page of set $s_j$-objects whose keys are in the range $R_i$, i.e., whose keys are in the interval $[K_i, K_{i+1})$. If the set $s_i$ does not contain any such elements, $R_{i,s_j}$ is null.

The higher-up nodes of the CG-tree are regular B+-tree nodes—having just one emanating node pointer per range.

The cardinality of the indexed sets and their distribution of key values may be non-uniform. In our example, one can expect that higher salaries are typically found for CEO objects whereas the lower salaries are usually paid to “regular” Emp objects. To compensate for this skew in attribute value distribution, leaf nodes may be shared by several neighbored directory entries. Such a situation is shown in Figure 1.9. Assuming that an underflow occurs in the leaf pages $L_1$ and/or $L_2$ of the tree shown on the left-hand side. This underflow is compensated by merging the two leaves into a single leaf page $L_{12}$—as shown on the right-hand side. This combined leaf page is now referenced by two neighboring directory entries via the pointers $R_1.s_1$ and $R_2.s_1$.

Unfortunately, space limitations prohibit a more detailed description of the CG-tree in this presentation; more details can be found in [29].

### 1.4 Navigation Index for Object Bases

Navigation along various types of references among objects has become one of the performance bottle-neck since the traditional techniques used for navigation are by no means sufficient. With a new organization, called Navigation Index, the navigation is performed by computations based on the theory of simple continued fractions.

In this index, traversing is performed by using two simple data structures; expander and linker. The efficiency of the technique is based on the fact that some moves in the navigation can be done just by computation in the CPU. The idea is to append objects with codes which not only uniquely identify the objects, but also reflect relationships with other objects in the object store. The important feature of the coding technique should be that, once an object code is given, codes of all the objects referencing the given object can be provided. We suggest applying the theory of simple continued fractions, SICFs [59]. The justification for this choice is provided in the next section. This approach of coding was first used in [42] as a
method for hierarchy processing in relational systems. However, our application to
the object store navigation is much more general since it must manage navigations
in oriented graphs.

The Idea

As originally presented in [42] and also explained in [66], SICFs may be used as a
fixed-length encoding method for representing ordered tree structures. Positions of
nodes of ordered trees can be, in fact, determined (identified, coded), by a sequence
of branch position numbers, one for every level preceding to the node from its root
node. A disadvantage of this "natural" coding technique is that resulting codes have
variable lengths depending on the position of a specific node in the tree. On the
other hand, such codes represent by their values an ordering system on trees, which
is very important for data manipulation operations on tree structures. Actually, the
coding technique by SICFs, can provide the same capabilities and, at the same time,
does not have the disadvantage of the variable-length codes. As we will see later,
it is a fixed-length coding method, but SICF codes can be easily expanded into the
natural codes and also any natural code can be transformed into its SICF code. In
this way we can find out the position of a node in the tree and also determine the
codes of all the ancestor nodes. Because of the limited space of this paper we are
not able to explain all the details here and we will only concentrate on the most
essential facts relevant to our research. Let a simple continued fraction code, C, be a
rational number maintained as a nominator, N, and a denominator, D. The general
rules which assign codes to objects in trees are the following:

- the root object Om is assigned the code \( C_m = \frac{N_m}{D_m} = \frac{1}{S} \), where \( S > 1 \) is
called the seed of the tree,

- references starting from a given object are labeled from left to right with
natural numbers, branch sequence numbers, \( q_i \), \( i = 1, 2, ... \),

- provided the code of a starting object in a reference is known, each terminal
object in a reference is assigned a rational number according to the following
formula:

\[
\text{code(terminal object)} = \frac{1}{\text{(branch sequence number + code(starting object))}}
\]

The interesting and very important feature of the coding method is that, once
a specific code is given by its nominator and denominator, we can compute codes
of objects referencing the given object. Furthermore, we can also compute branch
sequence numbers of references leading to the object, and the seed of the tree root.
To provide the information, we just apply the following formulas:

\[
\begin{align*}
D_i + 1 &= N_i \\
N_i + 1 &= D_i \mod N_i \\
q_i &= D_i \div N_i
\end{align*}
\]
Figure 1.10: Coding trees by simple continued fractions

Such mathematical formalism is also important for defining and computing Code Key, $K(C)$, the natural code mentioned earlier. It is a composite key consisting of a seed and branch sequence numbers leading to a specific object with code $C$, namely:

$$K(C) = qm, qm - 1 \ldots, q2, q1$$

Notice that $qm$ is the seed of the tree. The importance of the key is that it provides for a topological ordering (i.e. the top-down, left-right ordering of codes in a linear array). We will use this feature of the codes in Section 3.5 for the design of the implementation of the navigation index. Most of the facts discussed so far are also illustrated in Figure 1.10. Let us summarize what the coding technique by SICFs offers for an object store forming a tree structure:

- the position of every object in a tree is uniquely coded by a fixed-length code,
- every code can be expanded into a sequence of codes representing ancestor objects in the tree hierarchy,
- for every expanded code we can compute the branch sequence number,
- the seed of a tree can be computed from any code of a specific tree,
- codes of dependent objects can be computed, provided their branch sequence numbers are known.

**Navigation Index Decomposition Paradigm**

Now we return to the more realistic view of an object store, as it was defined in the previous section. We will try to investigate if and how the coding technique of SICFs could be applied to it. According to the definition, object store is not a tree. Parts of the object store can form trees, but these trees can be appended to other trees because of the shared objects. Some references can even form cycles.
The basic idea of how to deal with this kind of structure is decomposition. The network of objects is decomposed into smaller parts, i.e. graphs and trees, which can be coded by SICFs. Then, the objects and their codes are stored in a structure called expander. Another index structure, linker, is kept to maintain relationships between the decomposed trees. The decomposition of object store can be defined as follows.

**Definition 1.4.1** Graph \( G \) is a set of objects from \( O \) which are related (or transitively related) and a set of references from \( R \) containing these objects.

**Corollary 1.4.1** In this way, the object store can be decomposed into disjoint subsets of objects and references called graphs. Each object and reference appears in exactly one graph which implies that graphs are not connected. The graphs are general oriented graphs including shared nodes (objects) and cycles.

**Definition 1.4.2** Tree graph, \( T \), is a sub-graph of \( G \) which can be obtained by the following decomposition steps:

1. Every isolated, entry, or shared object is a root of tree,
2. Tree is the enclosure of the root, not containing other roots and their enclosures, plus associated references.

**Corollary 1.4.2** The tree decomposition of a graph is again a disjoint decomposition of objects and references. However, references connecting different trees, more precisely, references pointing to roots, are not included in this decomposition.

We illustrate the decomposition process by using a running sample of the object store shown in Figure 1.11. There we have two graphs:

\[
G_1 = \{O_0\}
\]

\[
G_2 = \{O_1, O_2, ..., O_{10}, R_1, R_2, ..., R_{10}\}
\]
Since there is only one object in the first graph G1, this object is also the only tree of the graph, T1,1 = O0. The second graph offers more possibilities for illustration. According to Definition 2, the roots of these graph trees are O1, O2, and O6. Once the roots are defined, the tree decomposition is obvious:

\[
\begin{align*}
T2, 1 & = O1, O3, O4; R2, R3 \\
T2, 2 & = O2, O5, O7, O8; R4, R7, R8 \\
T2, 3 & = O6, O9, O10; R9, R10
\end{align*}
\]

It has been shown in [66], that this object store decomposition paradigm is general enough to manage also cycles and multiple links between objects.
Chapter 2

Access Methods for Multimedia Databases

2.1 Current Trends in the Similarity Retrieval Indexing

The extension of data types in multimedia databases has resulted in an important change of the content query paradigm – it significantly differs from that of traditional database and information retrieval systems.

First, such queries cannot be as logically rigorous, because objects, such as images or videos, are not atomic symbols, so the object equality is not a particularly realistic predicate. Instead, searches tend to be based on similarity, so resemblance or proximity is more important than the perfect matching. Second, all that is similar is not necessarily correct – this is not entirely new concept, but its significance is increasing. In other words, the paradigm rejects the idea that queries may be expressed in terms of necessary and sufficient conditions able to determine exactly which objects one wishes to retrieve. Instead, a query acts more like an information filter, defined in terms of specific properties (features, concepts), which reduces the user’s search task by providing only a small number of candidates to be examined – it is more important that candidates which are likely to be of interest are not excluded than it is that possibly irrelevant candidates be included.

Second, depending on application, different types of similarity queries are required, however, the most frequently used similarity types of queries are known as the range and the nearest neighbor queries.

**Range query:** find all objects that are within a specific distance from the desirable query object;

**Nearest neighbor query:** find the first k closest objects to the given query object.

Some systems also require queries, such as ”find all similar sets of objects”, which are usually called the all-pair queries or the similarity joins. The aim is to find all pairs of objects that satisfy the constraints of a similarity selection, where the pair either comes from the same set or two sets of objects – alternatively, this operation could find all-triples, etc.
According to [64], the similarity sampling is also a relevant type of query for many applications. This query returns objects, representing samples of the database relative to a reference. This might be useful for browsing operations, such as zoom in and zoom out in large data sets.

Dealing with the “similarity” notion in a single and unified approach is, in its general form, quite a difficult challenge, due to the following major difficulties which immediately arise:

**Generality** The concept of similarity is usually type and application dependent, so a unified general view is very difficult to define. A recent remarkable proposal in [53] introduces a linguistic framework to specify ad hoc similarity-based query languages. This approach is interesting but, because of its generality, unable to give any practical insight on the development of a specific application problem.

**Complexity** Computational complexity is another major source of difficulties. The fact is that as soon as one tries to gain in generality, un-tractable or undesirable problems arise. This is unfortunately true even for apparently simple tasks, as [53] demonstrates for the case of string similarity.

**Performance** Indexing and storage structures aim at efficiency in processing large databases. The basic requirement on indexing in a database is that it should provide sufficient discrimination to prevent retrieval of a large fraction of the database – it does not need to produce the exact result. This is especially true when dealing with a similarity match rather than with exact match queries. Reasonable solutions have only been found for specific, “very well” defined, problems.

**Semantic content** Automatic extraction of semantic descriptors (features) from a MM object is a formidable challenge – think of, say, image [41] and/or text understanding techniques. Furthermore, the presence of multiple context-dependent interpretations further complicates the problem.

The current research activities try: (1) to transform the new problems into other problems for which efficient implementations are already known, or (2) to develop new structures, able to deal with some of the new problems directly.

### 2.1.1 Adjusting New Problems to Fit Existing Tools

**Object transformation** As an example, objects with $n$ vertices in a $k$-dimensional space can be mapped into points in a $nk$-dimensional space. In particular, a 2-dimensional rectangle described by coordinates $(x_1, y_1, x_2, y_2)$ is equally represented in a 4-dimensional space, where each attribute resides in a different dimension. Naturally, after such transformation, points can be stored directly in existing multidimensional point indexes, such as the grid files.

A different approach is to map objects from a $k$-dimensional space into 1-dimensional objects embedded in the original $k$-dimensional space such that they can be ordered linearly. These 1-dimensional objects can be indexed using conventional indexes such as B-trees.
In specific cases, more sophisticated transformations can be used. For instance, consider the QBIC system [46] which allows for fast retrieval of images described by shape, color, and texture features. The “trick” to deal with color queries is to use a distance-preserving transformation (such as the Discrete Fourier Transform - DFT) that allows for false-drops but avoids false dismissals. The same approach is used in [37] for the retrieval of approximate temporal sequences. In both these cases, the transformed space can be searched via a spatial access method (SAM), such as the R-tree or one of its variants [50, 60].

**Object duplication** A k-dimensional data space can be partitioned into pair-wise disjoined sub-spaces. An object identifier is duplicated and stored in all sub-spaces it intersects; that is, an object identifier may be stored in multiple pages.

This technique is best explained through an example. The sub-sequence matching problem aims to identify sequences which contain a sub-sequence similar to a given one. In [48], sequences are first “broken down” by using a sliding window, each obtained sub-sequence is transformed by using DFT, and the resulting set of points (trail) in the chosen feature space is approximated by a set of rectangles, which are then organized by a SAM. A conceptually similar approach has been undertaken in [63] to search for specific patterns on a database of raster images.

**Distance measure transformation** The problem of selecting a suitable set of features to accurately represent objects is not always an easy task. An alternative approach, investigated in [47], suggests to start from an expert-defined similarity (distance) matrix, and assumes that each object is a point in a “virtual” high-dimensional space. By means of a heuristic transformation technique, aiming to preserve pair-wise distances as much as possible, objects are mapped to points in a lower k-dimensional space (k being user-defined), where SAMs can be used.

Presumably, any approximated distance measure must preserve the distance between objects as much as possible. It has been shown in [48] that no dismissals occur, if the actual distance is lower-bounded by the actual distance in the distance space, more precisely:

\[
    d_{\text{approximated}}(O_i, O_j) \leq d_{\text{actual}}(O_i, O_j),
\]

for any pair of objects from a given database.

### 2.1.2 Developing New Techniques

**TV-tree** The idea to use dynamically only the necessary (typically smaller) number of object dimensions for addressing is proposed in [55]. The idea is to use a variable number of dimensions to build a tree index, adapting to the number of objects to be indexed and the level of the tree. Thus, for nodes that are close to the root, only few dimensions are used, resulting in a high fanout which is
believed to be good for searching. Descending the tree, more dimensions are considered to obtain a higher discriminating power. Given that the feature vectors contract and extend dynamically, resembling a telescope, this tree is called the Telescopic-Vector tree, or TV-tree. Simulation results in [55] report superiority of the TV-tree over the R*-tree with up to 80% savings in disk accesses.

**X-tree** A new method for indexing large amounts of point and spatial data in high-dimensional space is proposed in [38]. The development is motivated by the well-known observation that index structures, such as the R*-tree, are not adequate for indexing high-dimensional data sets. The major problem of these R-tree-based index structures is the overlap of the bounding boxes in their directory nodes – it is increasing with growing dimensions. To avoid this problem, X-trees introduce a new organization of the directory which uses a split algorithm minimizing overlap by utilizing a new concept of super-nodes. The basic idea is keeping the directory as hierarchical as possible, and at the same time avoiding the directory node splits that would result in high overlaps. Experimental results in [38] demonstrate that the X-tree can outperform R*-tree and the TV-tree by up to two orders of magnitude.

**SS-tree** Another type of indexing which is suitable for multi-dimensional point data, called the similarity indexing, has been proposed in [64]. The solution is provided in the form of an index structure called SS-tree. It is a dynamic indexing structure with splitting and balancing strategies similar to those defined for R-trees. As a fixed distance measure, a weighted Euclidean distance was chosen. From some other perspective, SS-tree can also be seen as a hierarchical clustering algorithm, because similar objects (points in a high-dimensional vector space) are stored in leaf nodes while centroids of the corresponding subtree objects are maintained in the index nodes. As the experiments published in [64] demonstrates, SS-tree is superior for similarity indexing applications to the R*-tree.

**Metric trees** Most commonly used distance and similarity measures between objects satisfy the definition of a metric distance function, \(d(x, y)\), i.e.:

\[
\begin{align*}
(i) & \quad d(x, y) = d(y, x) \\
(ii) & \quad 0 < d(x, y) < \infty, x \neq y \text{ and } d(x, x) = 0 \\
(iii) & \quad d(x, y) \leq d(x, z) + d(z, y)
\end{align*}
\]

(i) symmetry;
(ii) positivity;
(iii) triangle inequality.

Approaches based on so-called metric trees directly use pair-wise distances between objects to recursively partition the search space without considering positions of objects in a multidimensional space – dimensionality of space needs not even to be known. Several techniques, such as GNAT [40] and VP-trees [44], which basically differ in the criterion used for partitioning objects, have recently been proposed, with the common objective to design a data structure that would exploit local features of the metric for solving similarity (proximity) queries [62].

30
VP-trees The basic construction of a VP-tree is to break the space up using spherical cuts. To build it, pick a point in the data set – this is called the *vantage point*, hence the name VP-trees. Now, consider the *median* sphere centered at the vantage point with a radius such that half the remaining points fall inside it and half fall outside. For every other point, put it in one branch if it is inside the sphere and in another branch if it is outside the sphere. The lower level branches are constructed recursively. This idea has been extended in [44] to the *n*-ary case and implemented as an indexing method for content-based image retrieval.

GNAT’s The primary goal of the Geometric Near-neighbor Access Trees is to have a data structure that would reflect the intrinsic geometry of the underlying data. More specifically, the top node of the tree gives a brief summary of the data in a given metric space, and when going down, one gets a more and more accurate sense of this data geometry. The aim is achieved as a hierarchical *Dirichlet domain* (or *Voronoi diagram*) based structure – given a number of points, \( x_1, x_2, \ldots, x_k \), the Dirichlet domain of \( x_i \) consists of all possible points in the space which are closer to \( x_i \) than to any other point \( x_j \) (\( j \neq i \)). In order to minimize the number of distance computations while executing queries, a number of pre-computed distances is stored in the tree. However, the tree is built top-down, so the resulting structure is not very convenient for dynamic files.

Clustering The problem of finding similar, or in a broader sense *useful*, data patterns in large databases is closely related to the problem of clustering, i.e. identification of densely populated regions in multi-dimensional data sets.

The *probability-based* approaches, e.g. [49], typically make the assumption that probability distributions on separate attributes are statistically independent of each other. However, this is far from true, because a strong correlation between data file attributes exist and, in fact, form natural clusters which should be identified. The *distance-based* approaches, e.g. [54], assume that all data objects are given in advance and that scanning of such data sets is not expensive – large files are quite expensive to read. In another clustering method, called CLARENS [58], a data cluster is represented by its *mediod*, or the most centrally located data point in the cluster. The clustering process is formalized as a searching a graph in which each node is a \( k \)-partition represented by a set of \( K \) mediods, and two nodes are neighbors if they only differ by one mediod.

Probably the most successful clustering method for large data files is the BIRCH (Balanced Iterative Reducing and Clustering using Hierarchies) [67]. This method tries, incrementally and dynamically, to cluster incoming multi-dimensional metric data points with the available (limited) memory resources. Its I/O cost is linear in the size of the data set. A single scan of the data set typically yields a good clustering level, and one or more additional passes can improve the quality even further.
2.2 Current Research Activities by PASTEL Partners

Respecting the specific needs of multimedia data and the current "state of the art" in this field, the choice of research contributions was, in general, motivated by the following argumentation.

Generality Dependency on specific domain and/or application. Since different, often ad hoc, techniques are used for retrieval based on color, shape, position, texture, etc., a variety of access structures would be required to support different search modalities. Work on generalized access structures is therefore needed to avoid explosion in the number of access methods that a MM database system should implement [51].

Vector space limitation Most approaches are based on multidimensional or spatial access methods. They assume that all objects have numerical-valued features and can be represented as objects in a specific multidimensional space. However, not all multimedia object features are numeric and the number of feature values can vary from object to object, e.g., features can also be expressed as sets of categorical values. Thus, investigation on new access methods able to deal with this more complex scenario is needed.

Dimensional curse Another problems with the usage of the multidimensional and spatial access methods arise when the space dimensionality is high. Besides worsening of performance, this approach breaks down when indexing $N$ points from a space whose number of dimensions exceeds $O(\log N)$ [62].

Dynamic files Most available solutions assume static data sets, i.e. where neither the size nor the volume of the database change over time. This is especially true for existing designs based on metric trees. Since metric trees are built using a top-down decomposition strategy, they are typically not balanced. This has negative effects not only on query performance but also on updates, which are complex to manage and/or can lead to a global structural reorganization.

Specifically, we have concentrated on developing access structures for high-dimensional vector and generic metric spaces, as well as search mechanisms for genomic databases.

2.2.1 Vector-Approximation File: VA-File

Content based similarity search in multimedia databases often relies on features which are represented by points in a high-dimensional vector space. The number of dimensions in such feature vectors varies between moderate, from 4–8 in [30] or 45 in [31], and large, such as 315 in a recently-proposed color indexing method [32], or over 900 in some astronomical indexes [33]. To provide a sufficient retrieval quality, several features are combined to a large vector. The conventional approach to index such feature vectors and to support similarity search is to utilize a space- or
Figure 2.1: (a) Building the VA-File. (b) Vector selectivity for the VA-File as a function of the database size. $b_i = 6$ for all experiments.

data-partitioning method, e.g. a gridfile [34] or an X-tree [35]. However, space- and data-partitioning methods suffer from the so-called *dimensional curse*. Recently, it has been shown analytically and experimentally in [36] that such methods are unsuitable if the number of dimensions becomes large. Tree methods exhibit a linear search complexity at high dimensionality, and they are outperformed by a simple sequential scan.

In this situation, we have come up with an alternative organization based on *approximations* to make the unavoidable sequential scan as fast as possible. The *vector approximation file* (VA-File) divides the data space into $2^b$ rectangular cells where $b$ denotes a user specified number of bits. Instead of hierarchically organizing these cells like in grid-files or X-trees, the VA-File allocates a unique bit-string of length $b$ for each cell, and approximates data points that fall into a cell by that bit-string. The VA-File itself is simply an array of these compact, geometric approximations. Nearest neighbor queries are performed by *scanning* the entire approximation file, and by excluding the vast majority of vectors from the search (filtering step) based only on these approximations.

**Compressing Vector Data:** For each dimension $i$, a small number of bits ($b_i$) is assigned ($b_i$ is typically between 4 and 6), and $2^{b_i}$ slices along the dimension $i$ are determined in such a way that all slices are equally full. These slices are numbered $0, \ldots, 2^{b_i} - 1$ and are kept constant while inserting, deleting and updating data points. Let $b$ be the sum of all $b_i$, i.e. $b = \sum_{i=1}^{d} b_i$. Then, the data space is divided into $2^b$ hyper-rectangular cells, each of which can be represented by a unique bit-string of length $b$. Each data point is approximated by the bit-string of the cell into which it falls. Figure 2.1(a) illustrates this for four sample points. In addition to the basic vector data and the approximations, only the boundary points along each dimension must be stored. Depending upon the accuracy of the data points and the number of bits chosen, the approximation file is 4 to 8 times smaller than the vector file. Thus, storage overhead ratio is very small, on the order of 0.125 to 0.25.

**The Filtering Step:** When searching for the nearest neighbor, the entire approx-
Figure 2.2: (a) Block selectivity of real data. (b) Wall-clock time for the image database. $b_i = 8$ for all experiments.

imation file is scanned and upper and lower bounds on the distance to the query can easily be determined based on the rectangular cell represented by the approximation. Assume $\delta$ is the smallest upper bound found so far. If an approximation is encountered such that its lower bound exceeds $\delta$, the corresponding object can be eliminated since at least one better candidate exists. Analogously, we can define a filtering step when the $k$ nearest neighbor must be retrieved. A critical factor of the search performance is the selectivity of this filtering step since the remaining data objects are accessed in the vector file and random IO operations occur. If too many objects remain, the performance gain due to the reduced volume of approximations is lost. The selectivity experiments in Figure 2.1 shows improved vector selectivity as the number of data points increases ($d = 50$, $k = 10$, $b_i = 6$, uniformly distributed data). At $N = 500'000$, less than 0.1% (=500) of the vectors remain after this filtering step. Note that in this figure, the range of the y-axis is from 0 to 0.2%.

**Accessing the Vectors:** After the filtering step, a small set of candidates remains. These candidates are then visited in increasing order of their lower bound on the distance to the query point $Q$, and the accurate distance to $Q$ is determined. However, not all candidates must be accessed. Rather, if a lower bound is encountered that exceeds the $(k$-th) nearest distance seen so far, the VA-file method stops. The resulting number of accesses to the vector file is shown in Figure 2.1(b) for 10th nearest neighbor searches in a 50-dimensional, uniformly distributed data set ($b_i = 6$). At $N = 50'000$, only 19 vectors are visited, while at $N = 500'000$ only 20 vectors are accessed. Hence, apart of the answer set (10) only a small number of additional vectors are visited (9–10).

**Performance Comparison:** In order to demonstrate that the VA-File methods is a viable alternative, we performed many evaluations based on synthetic as well as real data sets. In what follows, we present a representative nearest neighbor search evaluation comparing the VA-File with the R*-tree, the X-tree and a simple sequential scan. The data set was obtained by extracting 45-dimensional feature
vectors from an image database containing more than 50'000 images\textsuperscript{1}. The number of nearest neighbor to search was always 10 (i.e. \( k = 10 \)). All experiments were performed on a Sun SPARCstation 4 with 64 MBytes of main memory and all data was stored on its local disk.

Figure 2.2(a) depicts the percentage of blocks visited as a function of dimensionality. As mentioned earlier, the tree-methods degenerate to a scan through all leaf nodes. On the other hand, the VA-File improves with dimensionality and outperforms the tree-methods beyond a dimensionality of 6. We further performed timing experiments based on this data set: In Figure 2.2(b), the elapsed time for 10th nearest neighbor searches with varying dimensionality is plotted. Notice that the scale of the \( y \)-axis is logarithmic in this figure. In low-dimensional data spaces, the sequential scan (\( 5 \leq d \leq 6 \)) and the X-tree (\( d < 5 \)) produce least disk operation and execute the nearest neighbor search fastest. In high-dimensional data spaces, that is \( d \geq 6 \), the VA-File outperforms all other methods.

\textbf{Discussion:} Our experimental results suggest that the VA-File—or a method similar to it—is generally the preferred method for nearest-neighbor similarity search over moderate and large data sets with dimensionality in excess of around 6. Further, the simple and flat structure of the VA-File also offers a number of important advantages such as parallelism, distribution, concurrency and recovery, all of which are non-trivial for hierarchical methods. Moreover, the VA-File also supports weighted search, thereby allowing relevance feedback to be incorporated. Relevance feedback can have a significant impact on search effectiveness.

\subsection{2.2.2 Metric Tree: M-tree}

Indexing dynamic and large collections of persistent data in order to support efficient execution of similarity queries is the general objective of M-trees. In fact, this kind of structure can be applied whenever a \textit{metric distance function} exists to measure the level of (dis-)similarity between pairs of indexed objects. In this respect, it can be seen as a generalized index structure able to substitute search functions of most of the traditional methods, including the multidimensional access structures, even though the performance of specialized structures in proper applications can typically be expected better than the performance of corresponding M-trees. This implies that the primary application area should be seen among cases where either the traditional methods fail or when no other information (such as the total ordering of the set of indexed objects) but a distance measure for pairs of objects exists, which quite often appear in multimedia applications.

More precisely, indexed objects are assumed to belong to a \textit{metric space} \( \mathcal{U} \), the \textit{universe}, i.e. a space for which a distance function, \( d : \mathcal{U}^2 \rightarrow \mathbb{R}^+ \cup \{0\} \), satisfying the properties of \textit{symmetry}, \textit{positivity}, and \textit{triangle inequality} for any triple of objects

\footnotesize{\textsuperscript{1}The color similarity measure described by Stricker and Orendo [31] generates 9-dimensional feature vectors for each image. A newer approach treats five overlapping parts of images separately, and generates a 45-dimensional feature vector for each image. Note: this method is not based on color histograms.}
$O_x, O_y, O_z \in U$.

In order to design an efficient secondary storage organization, the following requirements are also considered:

**Paging** The tree should be paged, i.e. consisting of fixed-size nodes. This is motivated by the need of optimizing the design for trees which reside on external memory devices, a basic requirement for large databases.

**Balancing** The tree has to be balanced, with paths from the root to leaves all having the same length – strict balancing might be relaxed, but the tree should never be allowed to degenerate into a list.

**Dynamism** The tree has to be dynamic, i.e. it must be able to deal with insertions and/or deletions without degrading search performance and storage utilization, and completely avoiding global (costly) reorganizations.

To the best of our knowledge, no existing index structure can solve similarity queries on a generic metric space and, at the same time, satisfy the above requirements. The basic idea and motivations are presented in [43], a specification of the M-tree can be found in [65] – see also Appendix TR7 of this deliverable.

**The concepts**

In M-trees, database objects are recursively organized by considering their distances from selected (reference) objects, called the *routing* objects. Routing objects are those database objects which have acquired routing roles according to specific, called *promotion*, algorithms.

Each routing object $O_j$ provides access to a sub-tree, $T(O_j)$, through a root pointer, $\text{ptr}(T(O_j))$, which is a part of the $O_j$'s entry in an M-tree node. It is also true to say that objects in the root node of $T(O_j)$ are univocally "associated" with their *parent* object, i.e., the routing object $O_j$. The union of $\{O_j\}$ and the set of objects in $T(O_j)$ is denoted as $\mathcal{T}(O_j)$ and called the *covering tree* of $O_j$. Note that this also applies to ground objects, for which $\mathcal{T}(O_j) = \{O_j\}$.

For each object $O_j$, its *covering region*, $\mathcal{R}(O_j)$, is defined as:

$$\mathcal{R}(O_j) = \{O_i \in T | d(O_j, O_i) \leq r(O_j)\}$$

where $r(O_j)$ is the *covering radius* of $O_j$. For ground objects we take $r(O_j) = 0$, which implies that $\mathcal{R}(O_j) = \{O_j\}$.

The semantics of the covering radius is that the distance from $O_j$ to any object in $\mathcal{T}(O_j)$ does not exceed $r(O_j)$, that is:

$$r(O_j) \geq \max\{d(O_j, O_i) | O_i \in \mathcal{T}(O_j)\}$$  \hspace{1cm} (2.1)

\textsuperscript{2}For ease in notation, $T$ designates the complete tree and $\mathcal{T}$ its corresponding set of objects.
As any other balanced tree, M-tree grows in a bottom-up fashion. Every object, when inserted, starts as a ground object and only later on can be promoted to become a routing object. In general, an object can be promoted several times, thus moving up in the tree to a node at level \( l (l > 0) \), leaf nodes are assumed to be at level 0). For each level \( k \in [1, l] \), the object still keeps its routing role, thus provides access to a sub-tree constrained by an appropriate covering radius.

**The operations**

**Insertion** The algorithm for inserting an entry for a new object \( O_n \), first descends the M-tree to locate the most suitable leaf node for accommodating \( O_n \). If insertion into the chosen node would cause the overflow, a node split is triggered and updates are propagated up to higher level(s) of the tree.

**Range query** Given a query object \( Q \) and a search radius \( r(Q) \), the search algorithm starts in the root and follows all paths that cannot be excluded from leading to relevant objects. Provided a single node at a time is processed, the algorithm maintains a queue of pending requests, \( PR \), containing pointers to nodes still waiting for examination; initially \( PR = \{ ptr(T) \} \).

**Nearest-neighbor** Given a query object \( Q \), the algorithm starts by considering:
1. any database object as the current nearest neighbor of \( Q \), and
2. the distance from this object to \( Q \) as the search radius. Then, the search continues in a similar way as for the range queries, but whenever an object more closer to \( Q \) is found, the current nearest neighbor is replaced and the search radius reduced.

**Implementation Issues**

Respecting the M-tree principles, nodes for inserting new objects as well as the choices of promoted objects can be determined arbitrarily and the M-tree always performs, though possibly slowly but correctly, accessing probably all nodes in a given tree. It is also obvious that among those many possible object distributions in an M-tree nodes, some arrangements of the same object set might presumably result in (much) better search efficiencies than the others. That means that though a specific object distribution in an M-tree is not a condition for a successful search, it is an important performance issue to be considered.

The dominant cost component of traditional storage structure operations is formed by the number of I/O accesses needed to execute these operations. Provided a single disk environment is used, the maximum number of such accesses is typically upper-bound by 50 accesses per second. Since the CPU costs are in such organizations prevalently reduced to only a not very high number of key comparisons, analytic formulas or quantitative measurements obtained from simulations or transactions run over some real-life databases usually neglect the CPU costs - such quantities are important to characterize performance of storage organizations.
In addition to the number of I/O accesses, the performance of M-trees is also influenced by the number of pair-wise distance, or (dis-)similarity, computations which are needed for executing transactions. Since a specific dis-similarity measure is one of the M-tree’s parameters, it is very difficult to take any fix assumption about the actual CPU costs – different application environments have reported required computation times ranging from microseconds to seconds or more. That means that in general, the performance of M-trees is not only I/O, but also CPU bound. However, the significance of these bounds depends heavily on an application for which specific M-tree is built.
Chapter 3

Generalized Indexing: Adding Value to Heterogeneous, Distributed Resource Collections

Claudia Niederée, Ulrike Steffens, Hans-Werner Sehring, Florian Matthes, Joachim W. Schmidt
Software Systems Institute,
Technical University Hamburg-Harburg, Hamburg, Germany
{c.niederee,ul.steffens,hw.sehring,f.matthes,j.w.schmidt}@tu-harburg.de

This section of the survey reports about current work on generalized indexing at the Software Systems Institute of the Technical University Hamburg-Harburg. A more complete description of this work may be found in the article On Indexing in Digital Libraries: Cooperation, Personalization and Evolution by Niederée, Steffens, Sehring, Matthes, and Schmidt. The work is embedded into two projects WEL and Kolibri in the context of digital libraries. Motivated by the requirements of distributed and heterogeneous collections like digital libraries a generalized approach to indexing is proposed in this paper. The main contribution of the generalization described is to introduce an additional intermediate layer between the navigational structure of the index and the information resources. This layer is populated by so-called index elements providing value-adding decision support and access information concerning the underlying information resources.

Starting from this model, services based on criteria like usage modes have been identified and classified. Special focus was on index evolution, cooperation based on indexes and personalization aspects.
3.1 A Generalized Approach to Indexing

In databases, information retrieval, and traditional paper documents, an index is often tightly coupled with the information resources being indexed degrading indexes to mere auxiliary components. The situation is somewhat different in digital libraries, where large collections of heterogeneous information resources from distributed information providers have to be structured by an index.

- Index and information resources may reside at different places making the actual step from the index to the information resource a potentially time-consuming and/or expensive one.
- There is no longer a single information provider for the entire indexed collection of resources.
- An index in a distributed environment often has to integrate selected material from several different resource collections.

These characteristics lead to a looser coupling between indexes and information resources. As a consequence, indexes may be provided by third parties as a value-adding service, which structures and annotates the existing material according to the needs of a specific user group or application domain.

In the generalized approach, loosening the coupling is achieved by introducing an additional, intermediate layer populated by surrogates for the information resources. These surrogates are called index elements and are first-class objects with useful operations in addition to a simple dereference operation.

3.2 A Model of Generalized Indexing

In the model it is assumed that a possibly distributed and heterogeneous resource space containing the collection of the indexed information resources is underlying the index. The information resources are equipped with handles enabling operations on the resources. The index is an additional information structure providing meta information structuring the underlying collection of information resources according to some syntactic or semantic ordering principle and improving access. This improvement of access may be in speed, in quality, or by providing new entry points to the information.

Taking a more abstract view on indexing, two major building blocks for generalized indexes are identified:

- A navigational structure that organizes the information resources into subcollections and steers navigation through the index

- Index elements that populate an intermediate, linking layer between the navigational structure and the information resources. The index elements describe the indexed information resources.
The information resources themselves are not part of the index. The index elements only provide references to these resources. The components of a generalized index model and their relationships are described in the following sections and summarized in the UML class diagram depicted in Figure 3.1.

The index elements are referenced by the nodes of the navigational structure and point to the information resource they represent. In order to meet their role as surrogates of the information resources, they contain information for two purposes: *decision* and *access*.

**Decision Information:** A main purpose of index elements is to provide enough information for the user to decide about the usability and potential profit of accessing the information resource. This includes preview information like thumbnail images, titles and abstracts of text documents, snapshots, keys of a data record, quality judgements like ratings, annotations and other kinds of third party meta information (or links to it) as proposed, for example, in [74] and classification information like keywords, subjects, or categories grouping and classifying the underlying information resources.

**Access Information:** Access information varies from simple memory addresses of local objects through references to remote objects to complex queries for remote information resources like databases or other content providers (digital libraries). This information may be completed by access modalities like access costs or the
availability of the information resources and administrative information like sizes and formats of the resources.

3.3 A Formal View on Generalized Indexing

A formalization of the index elements and their internal structure is developed with intent to clarify the concepts and to provide a precise basis for later modeling and implementation of the index elements. It is based on an abstract view on the underlying information resources called resource space. Three components are identified for an index element:

- **partial information** reflecting selected information from the information resources in a possibly condensed form;

- **access information** describing the location of the resource, access modalities and access methods;

- **value-adding information** resulting from evaluating the resource in the context of other documents, personal experiences, projects etc.

The information for decision support is aggregated in two different components of an index element, one containing the partial information that can be directly derived from the information resource and the other encompassing value-adding information that results from evaluating the resource in its context. The context of an information resource is defined in [73] as all the information apart from the
resource itself influencing the understanding and use of this information resource. This includes factors as different as the collection the considered information resource belongs to, classification schemes, and background knowledge and experience of the user.

The components of the index element and their relationship to the information resource is illustrated in Figure 3.2 an are described by predicates in [73].

3.4 Specializing the Generic Model

For a better understanding of generalized indexing, the introduced concepts are illustrated with different kinds of indexes. The summarized description of different example indexes in table 3.1 illustrates the wide variety of indexes included in the generalized indexing concepts.

3.4.1 BibTeX Files

Bib-files used as a basis for the BibTeX System are a good example of generalized indexes where the index is used for a wide variety of operations as a unit of its own independent of the underlying documents. The entries of a Bib-File can be easily identified as index elements surrogating the publications they actually reference.

A variety of partial information, which directly emerges by projection or condensation of the underlying publications, is contained within the entry fields, as for example author, title or pages. In addition, fields like annotate or price incorporate value-adding information, which is dependant on the context in which the entry was created. Access information is not as easy to detect within a BibTeX entry. In the general case, a share of the partial information like author, title, ISBN number serves as access information, too. Equipped with this information, the user might find the desired publication in a library or a bookstore. Beyond, some BibTeX fields may also include direct access information, like a hint to the owner of a document copy in the local environment or a field url.

As part of the Kolibri project [76], Bib-files are modeled as generalized indexes and an infrastructure is being implemented that realizes several index services.

3.4.2 The PI-Index

In the Warburg Electronic Library (WEL) project ([77], see also [75]), an interdisciplinary project of our department with the Art History department of the University of Hamburg, we are developing a digital library for art history research. It is part of the WEL project to transfer the PI-Index (PI = Political Iconography) into an adequate electronic counterpart. Political Iconography (PI) is the area of art history which examines political messages conveyed in in paintings, monuments and buildings.
<table>
<thead>
<tr>
<th>Index</th>
<th>Index Element</th>
<th>Partial Information</th>
<th>Value-Adding Information</th>
<th>Access Information</th>
<th>Information Resource</th>
</tr>
</thead>
<tbody>
<tr>
<td>Book Index</td>
<td>entry</td>
<td>keyword</td>
<td>none</td>
<td>page number</td>
<td>paragraph containing keyword</td>
</tr>
<tr>
<td>City Map</td>
<td>entry</td>
<td>street name</td>
<td>none</td>
<td>grid square</td>
<td>map sector</td>
</tr>
<tr>
<td>Database Index</td>
<td>index entry</td>
<td>primary or secondary key</td>
<td>none</td>
<td>memory address</td>
<td>data record</td>
</tr>
<tr>
<td>WWW Bookmark Collection</td>
<td>bookmark</td>
<td>document title or descriptor</td>
<td>annotations</td>
<td>URL</td>
<td>WWW resource</td>
</tr>
<tr>
<td>Inverted File (IR)</td>
<td>entry</td>
<td>index term, frequencies, etc.</td>
<td>frequencies relative to document collection, etc.</td>
<td>document reference</td>
<td>document</td>
</tr>
<tr>
<td>PI-Index</td>
<td>index card</td>
<td>thumbnails</td>
<td>classification, comments</td>
<td>image source or location of original</td>
<td>multimedia document</td>
</tr>
<tr>
<td>BibTeX File</td>
<td>bibliographic entry</td>
<td>author, title, etc.</td>
<td>comments, keywords</td>
<td>URL or derived from partial information</td>
<td>electronic or paper document</td>
</tr>
</tbody>
</table>

Table 3.1: Generalized Indexing: Examples

The index consists of an elaborated, hierarchically structured ontology of concepts referring to politics, political acts, and social phenomena and of 250,000 paper cards classified according to these concepts. The ontology provides the navigational structure of the index. The cards are index elements: Each card contains information about the underlying art historic artifacts (information resource) like a small image and references to the artifact. In addition, the cards also contain value-adding information including their classification and comments. The cards are also artifacts of their own that are subject to art historic research without accessing the underlying resources.

3.5 A Service Framework for Generalized Indexing

The presented model of generalized indexing is intended as a basis for the development of a framework for generic, value-adding services for the use and management of indexes and information resources.

For this purpose the numerous services expected from a generalized index are categorized systematically. This categorization leads to a service framework, which offers two major advantages. On the one hand, it promotes the development of new services for each category. On the other hand, it serves as the basis for the con-
struction of a generic software architecture that is able to deal with a broad range of
different indexes, different information resources and different application scenarios.
Our study is based on the model developed in Section 3.1 and distinguishes indexing
services along three dimensions:

1. Categorization based on the **index components** involved;

2. Categorization based on **user role**;

3. Categorization based on **service complexity**;

Since indexes structure information resources according to the particular require-
ments of individuals or of a user group, they constitute an important tool in per-
sonalized workspaces as well as an artifact for cooperative work in project teams. Therefore, the discussion of services pays special attention to services supporting cooperation and personalization.

Furthermore, indexes are subject to a permanent evolution resulting from two dif-
ferent sources. On the one hand, the underlying collection of information resources
changes rapidly. On the other hand, the active usage of an index leads to an evolu-
tion taking place inside the index itself, where the changing personal background
and experience of the index users are mirrored. To reflect the evolution of an index,
special services have to be provided.

### 3.6 Related Work

The described approach unites different ideas of supporting and exploiting surro-
gates of information resources as proposed in the literature within a general model,
emphasizing the existing commonalities. In the Dublin Core Project [79] the usefulness
of such surrogates for resource discovery described by the Dublin Core element
set is stressed. In [72] the lack of surrogates is identified as one of the key restrictions
to the retrieval capabilities of WWW indexers. The system GERHARD (German
Automated Retrieval and Directory) [69], built in the context of a DFG project at
the University of Oldenburg, is an example of a concrete application of a generalized
index containing a navigational structure and index elements.

The importance of meta information as provided by index elements is stressed by
various projects focusing on different aspects like architectural issues [68], classifi-
cation schemes [70, 71], or models for their description [78, 79].
Bibliography


