Optimal Type Hierarchy Linearization for Queries in OODB

Thomas A. Mueck
Abteilung Data Engineering
Universität Wien
Vienna, Austria
mueck@ifs.univie.ac.at

Martin L. Polaschek
Abteilung Data Engineering
Universität Wien
Vienna, Austria
polaschek@ifs.univie.ac.at

Abstract

Selection criteria in OODB refer to object properties as well as to type membership. The latter stimulated research on type hierarchy indexing, i.e., fast access to object sets based on type membership criteria.

A recent proposal in this context is the multikey type index. It is based on a symmetrical multikey search structure with \( n \) dimensions for \( n \) indexed object properties and one additional dimension for type membership. The domain of this type dimension is the type set of the indexed object. However, an important prerequisite for an efficient implementation of a multikey type index is an optimal linearization of the type set.

In this paper we show in how far the linearization affects the performance of the multikey type index and present the construction of optimal linearizations which is straightforward only in case of single inheritance. In case of multiple inheritance a recursive divide-and-conquer solution produces all existing optimal linearizations for a given type hierarchy.

The main contribution of this paper is a detailed outline of the corresponding algorithm for multiple inheritance type hierarchies. By using this algorithm, the multikey type index is an interesting alternative to the traditional approaches for type hierarchy indexing, which are almost exclusively based on B+-tree derivates.

Keywords Object-oriented databases, indexing, multi-dimensional search structures, multiple inheritance, type/class hierarchies

1 Introduction

The multikey type (MT) index as presented in [10, 11] maps the type hierarchy of an OODB scheme to a multidimensional search structure. The hierarchy is represented by an additional domain in the search data structure which is called type domain. The result is a multikey type index.

interface Person: 
    extent persons;
    attribute Integer id_card_no;
    attribute String name;
    attribute Integer income;
    attribute Float weight;

interface Student: Person { extent students; }

interface FacultyMember: Person { extent facultyMembers; }

interface Instructor: FacultyMember, Student { extent instructors; }

interface AssistantProfessor: FacultyMember { extent assistantProfessors; }

interface AssociateProfessor: FacultyMember { extent associateProfessors; }

interface FullProfessor: FacultyMember { extent fullProfessors; }

Figure 3: Schema definition

Q1: select x from x in persons
    where x.income < 10000

Q2: select x from x in facultyMembers
    where x.income <= 30000
    and x.income > 20000

Q3: select x from x in associateProfessors
    where x.income < 45000

Figure 4: Query definitions

namely set membership. Technically speaking both approaches extend B*-trees with multiple lists to organize OIDs with respect to their set membership. The Nested Inherited Index [1] and the Generalized Nested Inherited Index [14] are B*-tree based hybrid approaches supporting type membership as well as path expressions.

Multikey index structures have been discussed as an alternative to traditional, B*-tree based search structures in the context of type hierarchy indexing (see [13], [4], [5] and [10]).

In the rest of the paper Section 2 contains the problem statement and a definition of the term optimal linearization. Section 3 is the main section of the paper with a detailed presentation of the linearization algorithm. In Section 4 the proposal is put to work, an outline of a possible implementation framework is given. At the end of the paper, there are conclusions and references.

2 Hierarchy Mapping - The Problem Statement

Considering the ODMG-93 Object Database Standard [2], query specifications always refer object collections. To illustrate this query concept we use the database schema and the example queries (shown in Figures 3 and 4 in ODMG-93 notation). Since Q2 refers to instances of FacultyMember as well as to instances of its subtypes, it contains an implicit predicate on the object type, i.e., x.type ≤ FacultyMember, where ≤ denotes the partial order relation of the type hierarchy. In a multikey type index, such a predicate corresponds to a range of the type domain. The rationale for the following algorithm is that the query performance of a multikey type index is largely determined by the choice of the actual type hierarchy linearization. Figures 6 and 7 show the differences in the size of the actual query volumes based on different linearizations.

Assuming an arbitrary linearization, the query range in the type domain may also contain types not qualifying for the query request (see Figure 6). Since the resource consumption of a range query is positively correlated with the size of the respective range, we aim at minimal ranges for all extents (see Figure 7 for the extent of FacultyMember), i.e., at a linearization in such a way that exactly one interval contains all types which are part of one subhierarchy.
In geometrical terms, an optimal linearization yields for each possible type in a query a subspace not containing any object identifiers not belonging to the query result (see Figure 8 containing the queries given in Section 1). Consequently, a type domain setup (linearization) resulting in minimal query subspace volumes for all possible query requests is called optimal. More specifically, an ordering \( \subseteq \) is optimal for \((T, \leq)\), if \( \subseteq \) is a total ordering (see (1) in definition below), and for each subhierarchy of \((T, \leq)\) (with \(T_{\leq t}\) denoting the subhierarchy rooted at \(t\)), there is a closed interval \([u, v]\) in \((T, \subseteq)\), containing the same elements (i.e., types) as \(T_{\leq t}\) (see (2) in definition below).

**Definition 1 (Optimal linearization)**

Let \((T, \leq)\) be a type hierarchy and \(T_{\leq t}\) be the subhierarchy rooted at \(t\) (i.e., the interval \([u, v]\) in \((T, \subseteq)\), containing the same elements (i.e., types) as \(T_{\leq t}\) (see (2) in definition below).

\[\forall t, u \in T : t \subseteq u \lor u \subseteq t \quad \text{and} \quad \forall t \in T : \exists u, v \in T_{\leq t} \text{ in such a way that } [u, v]_{(T, \subseteq)} = T_{\leq t}\]

Obviously there are type hierarchies without any optimal linearization. Figure 9 shows the smallest hierarchy without linearization. Although stating a necessary and sufficient condition for the existence of an optimal linearization of a type hierarchy is not totally trivial, a closer look at the above definition yields at least one necessary and one sufficient condition (\(\text{super}(t)\) denoting the set of direct supertypes of \(t\)):

- An optimal linearization exists if each type has at most one supertype, i.e., single inheritance (\(\forall t \in T : |\text{super}(t)| \leq 1\) is sufficient).

- An optimal linearization does not exist if any type has more than two supertypes (\(\forall t \in T : |\text{super}(t)| \leq 2\) is necessary).

In the case of single inheritance the computation of the optimal linearization is straightforward. For example, a standard depth-first traversal of the hierarchy will do. The respective traversal has been proposed in [13].

![Figure 9: Hierarchy without optimal linearization](image)

**Figure 9: Hierarchy without optimal linearization**

In the multiple inheritance case Figure 10 illustrates that the second existence condition is only necessary. For both hierarchies depicted in this figure, the condition holds. However, a closer look at the two type hierarchies reveals that hierarchy (a) has an optimal linearization whereas hierarchy (b) has none. Informally, this result can be obtained by the isolation of all non-trivial subhierarchies, in particular \{AE\}, \{CF\}, \{DEF\}, \{BCDEF\} for (a) and \{AE\}, \{CF\}, \{DEF\}, \{BDEF\} for (b). The goal is a ‘flattening’ of the hierarchy such that the set of type identifiers forms a string and each subhierarchy is represented by a substring of this string. Drawing the corresponding set diagrams for the two hierarchies (see Figure 11) we observe that, in the first case, the diagram can be flattened in this way whereas in the second case this is not possible, since one of the subhierarchies cannot be represented by a substring (in Figure 11 this is \{BDEF\}).

![Figure 11: Set diagrams for type hierarchies of Figure 10](image)

**Figure 11: Set diagrams for type hierarchies of Figure 10**

in this way whereas in the second case this is not possible, since one of the subhierarchies cannot be represented by a substring (in Figure 11 this is \{BDEF\}).

In the following section, we present an algorithm which finds all optimal linearizations for a given hierarchy \((T, \leq)\).

### 3 The Mapping Algorithm

Prior to an in-depth presentation of the mapping algorithm we use the example hierarchies of the previous section (see Figure 10) for an informal presentation of the linearization task. Let \((S, \leq)\) denote the hierarchy to be processed.
1. The (unmarked) maximal elements of \((S, \leq)\) together with the subhierarchies rooted at these elements are determined and marked. For the running example the results for the two hierarchies are given as:

![Diagram](image)

In what follows some notational conventions for the necessary data structures (abstract representations) hold:

- squares represent single types
- marked types are shaded
- solid shapes represent sets
- dashed shapes symbolize lists

2. Non-disjoint subhierarchies are concatenated (operator \(\&\), see example Figure 12, exact definition below). If there are non-disjoint subhierarchies which cannot be concatenated (i.e., the intersecting parts are not located on either end of the lists) no linearization exists.

![Diagram](image)

With respect to hierarchy (a) there is only one concatenation step (intersection contains \(E\)). The processing of hierarchy (b) involves two concatenation operations, one for an intersection containing \(E\), the other one for an intersection containing \(F\).

![Diagram](image)

3. The lists resulting from the previous step are refined (operator \(*\), see example Figure 13. It should be noted that \(B \setminus \bigcup A_k = \emptyset\). Exact definition of \(*\) below), more specifically, for each subhierarchy rooted at an unmarked maximum it is checked, whether or not the subhierarchy has a nonempty intersection with more than one list element. In this case a refinement attempt is made. If there is any such subhierarchy without a possible refinement no linearization exists.

Considering hierarchy (a) there are two candidates (subhierarchies) for refinement: \(\{CF\}\) and \(\{DEF\}\). However, \(\{CF\}\) has a nonempty intersection with only one list element, i.e., \(\{BCDF\}\). Consequently, no refinement is done. \(\{DEF\}\) has nonempty intersections with both \(\{BCDF\}\) and \(\{E\}\), the result of the refinement is given below.

![Diagram](image)

In case of hierarchy (b) the only refinement candidate is \(\{DEF\}\). This subhierarchy has a nonempty intersection with consecutive list elements. However, the refinement fails, because the interior list element \(\{BD\}\) is not a subset of \(\{DEF\}\). At this point the linearization algorithm terminates for hierarchy (b). There is no optimal linearization for this hierarchy.

The next iteration for hierarchy (a) yields the subhierarchies \(\{E\}\) and \(\{CF\}\) as candidates. \(\{CF\}\) is a relevant candidate having nonempty intersections with \(\{DF\}\) and \(\{BC\}\). We end up with a configuration like:

![Diagram](image)

4. Using the \(\lor\) and \(*\) operators steps 1–3 produce lists of sets. The same processing scheme is applied to each element of these lists recursively. The results of the recursive descents are collected in the overall result set.

The final recursive calls for the list elements do not yield any modifications in the context.
of our running example. The final result is 

\[(A,E,D,F,C,B)\] and 

\[(B,C,F,D,E,A).\]

\[
\begin{align*}
A & \rightarrow B \\
(A_{\alpha_1}, A_{\alpha_2}) & \rightarrow (A_{\beta_1}, A_{\beta_2}, B_1, B_2)
\end{align*}
\]

A * B

\[
\begin{align*}
(A_{\alpha_1}, A_{\alpha_2}) & \rightarrow (A_{\beta_1}, A_{\beta_2}, B_1, B_2) \\
A_{\alpha_1} & \rightarrow (A_{\beta_1}, A_{\beta_2} B_1, B_2)
\end{align*}
\]

Figure 12: Concatenation operation (informally)

The main part of the proposed algorithm is a recursive function \(\text{order}\). Another integral part of this algorithm is a structured set \(S'\) constructed during the traversal of \((T, \leq)\). Elements of \(S'\) are either atoms (i.e., type identifiers) or structured lists. Elements of structured lists are in turn again structured sets. The recursive definition of this data structure allows arbitrary nestings. In the sequel, two special cases are used: flat sets, i.e., structured sets containing merely atoms, and flat lists, i.e., structured lists containing merely flat sets.

The following notational conventions for variables hold: flat sets are denoted by \(A, B, C, \ldots\), structured sets by \(A, B, C, \ldots\), flat list by \(A, B, C, \ldots\) and atoms by \(a, b, c, \ldots\). There are no variables used for structured lists.

Function \(\text{order}\) is invoked by the wrapping function depicted in Figure 15. After termination of the algorithm, a postprocessing step on \(T'\) produces all optimal linearizations (see below).

\[
\begin{align*}
D & \leftarrow \emptyset \\
T' & \leftarrow \text{order}(T, \leq)
\end{align*}
\]

Figure 15: Wrapper for function \(\text{order}\)

Prior to the formal definition of the function \(\text{order}\) in Figure 16 an informal pseudocode representation is given in Figure 14. However, the actual execution of \(\text{order}\) is illustrated by an example given below. At this point we have to define the exact meaning of all operators used in \(\text{order}\). The following operations and symbols are used:

- \{\}, () and \(\emptyset\) denote the set constructor, the list constructor, and the empty set, respectively.
- Set operators defined on flat sets are union (\(\cup\)), difference (\(\setminus\)), cardinality (\(|\cdot|\)), intersection (\(\cap\)) and set membership \(\in\).
- The operators \(\cup, \setminus\) and \(\in\) are also defined for the top level of structured sets.
- \(|A|\) denotes the number of flat sets contained in \(A\), denoted by \(A_1, A_2, \ldots, A_{|A|}\).

\[
\begin{align*}
1. & \begin{align*}
\text{begin} & \text{order}(S, \leq) \\
2. & \text{if } |S| < 3 \text{ then return } S \text{ end}
\end{align*} \\
3. & M \leftarrow \max(S \setminus D, \leq) \\
4. & L \leftarrow \bigcup_{m \in M} \{(S_{\leq m})\} \\
5. & D \leftarrow D \cup M \\
6. & S' \leftarrow S \setminus \bigcup_{m \in M} S_{\leq m} \\
7. & \text{while } \exists A \in L \text{ do} \\
8. & \quad L \leftarrow L \setminus \{A\} \\
9. & \quad \text{if } \exists B \in L : \bigcup A \cup B \neq \emptyset \text{ then} \\
10. & \quad \quad \text{if } A \circ B \text{ is defined then} \\
11. & \quad \quad \quad L \leftarrow L \setminus \{B\} \cup \{A \circ B\} \\
12. & \quad \text{else abort} \\
13. & \text{end} \\
14. & \text{else} \\
15. & \quad \text{while } \exists x \in \max(\bigcup A_i \setminus D, \leq) : \\
16. & \quad \quad |\{A_i \mid A_i \cap S_{\leq x} \neq \emptyset\}| > 1 \text{ do} \\
17. & \quad \quad \quad A \leftarrow A \circ S_{\leq x} \\
18. & \quad \quad \quad D \leftarrow D \cup \{x\} \\
19. & \quad \text{else abort} \\
20. & \text{end} \\
21. & S' \leftarrow S' \cup \{(\text{order}(A_1, \leq), \text{order}(A_2, \leq), \ldots, \text{order}(A_{|A|}, \leq))\} \\
22. & \text{end} \\
23. & \text{end order}
\end{align*}
\]

Figure 16: Recursive construction of all optimal linearizations

- \(\max\) yields a subset of a partially ordered set \(A\) in such a way that all elements in the subset are maximal elements of \(A\) and none of them are minimal elements of \(A\), i.e.,

\[
\max(A, \leq) \rightarrow \{a \in A \mid \exists a' \in A : a < a' \land \exists a'' \in A : a'' < a\}
\]

- \(\circ\) concatenates two overlapping flat lists, i.e., flat lists with common types in their respective sets. More precisely, two flat lists \(A\) and \(B\) overlap if and only if \(\bigcup A \cap \bigcup B \neq \emptyset\). All sets in such a list have to be nonempty and pairwise disjoint.

It should be noted that \(\circ\) is defined if and only if \(\exists (i, j) : A_i \cap B_j \neq \emptyset, i \in \{1, |A|\}, j \in \{1, |B|\}\). Informally, each of the two sets containing the common types has to be at one end of its enclosing list to enable concatenation. If this holds, there are \(4\) cases as depicted in Figure 17 (empty sets are removed from the concatenation result). Example: \((\{B\}, \{CD\}, \{EF\}) \circ ((FG), \{H\})\) yields \((\{B\}, \{CD\}, \{E\}, \{F\}, \{G\}, \{H\})\), whereas \((\{B\}, \{CD\}, \{EF\}) \circ ((DG), \{H\})\) is undefined, since \(\{CD\} \cap \{DG\} \neq \emptyset\) and \(\{CD\}\) is not placed at either end of its enclosing list.

- \(\ast\) represents refinement. \(A \ast B\) is defined if and only if \(A\) denotes a flat list of pairwise
disjoint and nonempty sets, \( B \) denotes a flat
set, \( \exists (i,j) : i < j \) and
\[
\forall k, 1 \leq k \leq |A| : \\
\begin{cases}
A_k \cap B = \emptyset & \text{for } k < i \\
A_k \cap B \neq \emptyset & \text{for } k = i \\
A_k \cap B \neq \emptyset & \text{for } i < k < j \\
A_k \cap B = \emptyset & \text{for } k > j \\
A_k \cap B = \emptyset & \text{for } i < k 
\end{cases}
\]
If \( A \ast B \) is defined, the result is given by (again,
empty sets are removed from the result):
\[
A \ast B \mapsto (A_1, \ldots, A_{i-1}, A_i \setminus B, A_i \cap B, A_{i+1}, \ldots, A_{j-1}, A_j \cap B, A_j \setminus B,
A_{j+1}, \ldots, A_{|A|})
\]
Example: \( (\{B\}, \{CDE\}, \{FG\}, \{H\}) \ast \{DEF\} \)
yields \((\{B\}, \{C\}, \{DE\}, \{F\}, \{G\}, \{H\})\).
\((\{B\}, \{CDE\}, \{FG\}, \{H\}) \ast \{DEGH\} \)
is undefined, since \((\{CDF\} \cap \{DEGH\}) \neq \emptyset\)
and \(\{H\} \cap \{DEGH\} \neq \emptyset\) and \(\{FG\} \not\subset \{DEGH\}\).

In the wrapping procedure (see Figure 15) set \( D \)
is initialized as empty set. Its purpose is to hold
already processed type identifiers. The wrapping
procedure passes \( T \) as actual parameter to the
initial call of function \( \text{order} \).

For a given type hierarchy \((T, \leq)\), e.g.,
\( T = \{ABCDEFG\} \) with \( \leq \) given in Figure 18,
after termination of order the result contained in \( S' \)

![Figure 18: Example type hierarchy](image)
can be used to construct all optimal orderings (the
actual execution of the function order is illustrated
in the Appendix). For the above example the
value of \( S' \) is \((\{(B\{E\F\}\} \cap \{H\} \{D\} \{A\C\})\)).

The set of all optimal optimizations is constructed in
the following way. Each set in the result can
be represented by an arbitrary permutation of
its elements, whereas each list yields only two
correct representations (i.e., forward or backward
sequence). In particular, sets \( \{EF\}, \{DG\}, \)
and \( \{AC\} \) can be represented by \(2!\) permutations each.
The same is true for the set \( \{B\{E\F\}\} \) containing
one atomic element \( B \) and a list \((\{E\F\})\) as second
element. It should be noted that this list contains
only one element, namely set \( \{EF\} \). The list
containing four elements, i.e., \( \{B\{E\F\}\}, \{H\}, \)
\( \{DG\}, \) and \( \{AC\} \) has only \(2!\) correct representations.
All in all a simple postprocessing traversal
yields \(2! \cdot 2! \cdot 2! \cdot 2! = 32\) optimal linearizations
for \( T \), e.g., \( B \cdot E \cdot F \cdot H \cdot D \cdot G \cdot A \cdot C\), \( B \cdot F \cdot E \cdot H \cdot D \cdot G \cdot A \cdot C\),
\( E \cdot F \cdot B \cdot H \cdot D \cdot G \cdot A \cdot C\), etc.

Applying the algorithm to the hierarchy of
Figure 5 results in \( \{\text{Person}, \{\text{FacultyMember,}
\text{AssistantProfessor, FullProfessor, AssociateProfessor}\}
\{\text{Instructor}\} \{\text{Student}\})\) thus giving \(2! \cdot 2! \cdot 4! = 96\) optimal linearizations.

4 Implementation Issues

In this section we apply the linearization algorithm
for the purpose of type hierarchy indexing. In
particular, the implementation of a multikey type
index with the help of optimal linearizations is
outlined.

In general, a multikey type index can be built
using any multikey search data structure. Con-
sequently, this section is not focussed on any
particular data structure like, e.g., the BV-tree
[3] or the hB-tree [7]. The only data structure
requirement is a non-degenerating behavior in case
of data skew.

Multikey search data structures interpret
n-tuples as elements of an n-dimensional geo-
mnetrical space. Physically stored tuples have to
be enclosed by an n-dimensional hyperrectangle
(called data space in the sequel) defined by totally
ordered attribute domains. Initially, the data space
is mapped to a single disk page. When the storage
space of this disk page is exhausted, the data
space has to be partitioned into two subspaces,
mapped to one disk page each. In any case of
page overflow this pattern is repeated. Thus,
the dataspace is successively partitioned into an
increasing number of subspaces as the number of
stored tuples increases.
1. begin order($S$, $\leq$) 
   $S$: set of type identifiers 
   $\leq$: partial ordering 
2. if $S$ contains less than 3 elements then return $S$ 
3. assign the set of all unmarked maximal elements to $M$ 
4. assign a set of lists to $L$ in such a way that each list contains one set 
   corresponding to a subhierarchy rooted at an element of $M$ 
5. mark all elements of $M$ 
6. assign all elements of $S$ to $S'$ which are not member of any subhierarchy 
7. foreach element of $A$ of $L$ do 
   remove $A$ from $L$ 
8. if there exists an element $B$ of $L$ such that 
   there is a $B_j$ of $B$ and an $A_i$ of $A$ with nonempty intersection then 
9. if $A \circ B$ is defined then 
   replace $B$ by $A \circ B$ in $L$ 
10. else there is no solution 
11. end 
12. else 
13. while there exists an unmarked maximal element $z$ such that 
   there are at least 2 elements of $A$ having a nonempty intersection 
   with the subhierarchy rooted at $z$ do 
14. if $A \ast S \leq z$ is defined then 
15. refine $A$ with $S \leq z$, i.e., with the subhierarchy rooted at $z$ 
16. mark $z$ 
17. else there is no solution 
18. end 
19. end 
20. end 
21. call order (recursively) for each element of $A$ and add the list of all results to $S'$ 
22. end 
23. end 
24. end 
25. return $S'$ 
26. end order 

Figure 14: Pseudocode for function order

In most cases an exact match query will qualify one subspace and therefore one disk page, whereas 
a range query will qualify a set of buddy subspaces 
corresponding to a set of disk pages. This concept 
of data space partitioning is theoretically appealing 
as it allows to implement symmetric index 
structures without any distinction between one 
clustering and $n-1$ non-clustering data structures 
for $n$ indexed object properties. However, from a 
technical point of view, subspace boundary values 
have to be stored and maintained in order to 
reconstruct the data space partitioning in case of 
query or update operations.

This means that any multikey search structure 
used to implement database indices has to contain 
two parts, namely one storage structure for boundary 
values (i.e. the partitioning information) and a 
second storage structure for tuples containing the 
object identifiers, the types, and the actual values 
of the indexed attributes. In what follows, the 
terms boundary structure and value structure will 
be used to refer to these parts, respectively.

One possible data structure setup could look 
like this: in any disk page of the value structure, 
the key values component contains the values of the 
indexed attributes. This component is followed by 
a list of object identifiers such that each identifier 
refers to an object having the attribute values 
given in the key values component. It should be 
noted that, in this context, the type identifier 
can be handled like any other attribute value, i.e., as 
part of the key values component. The optimal 
linearization algorithm guarantees minimum length 
query intervals in this type dimension.

Recalling the type hierarchy example from 
Section 1, a multikey type index on attribute income 
is shown in Figure 19 depicting the boundary 
structure and the value structure for this example.

The execution of query requests with the help 
of a multikey type index involves two phases:

- a traversal of the boundary structure (either 
  kept in main memory or on mass storage) 
  collecting the set of relevant disk page addresses

- a processing of the corresponding set of mass 
  storage transfer operations (checking all tuples 
  stored in the fetched disk pages and discarding 
  all tuples not qualifying for the query request)

Reconsidering Qz, the query execution is supported 
by this index in the following way (with respect to
attribute `income`). In a first step, the boundary structure is used to determine relevant parts of the corresponding value structure, \( X \) and \( Z \) in the example. In a subsequent step, these parts of the value structure are used to retrieve a set of object identifiers, \( \omega_3, \omega_4, \omega_5, \) and \( \omega_6 \) in this example.

An important advantage of this kind of indexing framework is that exactly the same search structure technology could be applied to maintain one index structure for all relevant attributes of `Person`, e.g. `income`, `weight`, `name`, and so on. Considering for example

\[
\text{Q4: select } x \text{ from } x \text{ in facultyMembers where } x.\text{income} < 50000 \\
\text{and } x.\text{name} = "Doe"
\]

the execution needs associative access to attribute `income` as well as to `name`. In single key approaches, the OODBMS is forced to maintain two distinct search data structures, thus spending considerably more storage space for index maintenance and considerably more index scan time.

5 Conclusions and Work In Progress

Optimal type hierarchy linearizations are motivated in the context of multikey type indices. An important prerequisite for this recent approach to type hierarchy indexing is an algorithm which is able to find such linearizations. Since we are aiming at minimum length intervals in the type domain, a linearization is called optimal if and only if the resulting type domain contains for each subhierarchy an interval with exactly the types of this subhierarchy. Using the linearization algorithm, any state-of-the-art multikey search data structure can be used to set up efficient type hierarchy indices. The main advantage of this approach is that, in case of more than one indexed attribute, the amount of index storage space overhead can be significantly reduced compared to a set of single key indices.

An interesting challenge is posed by hierarchies for which an optimal linearization in the definition of this paper does not exist. In these cases, at least two approaches have to be considered: allowing redundancy (i.e., duplicating types in the type domain) or allowing a controlled suboptimal setup of the type domain. These questions are subject to further work.

Acknowledgements

The authors thank the reviewers of a previous version of this paper and especially H.P. Kriegel for valuable comments and suggestions.

References


[12] Beng Chin Ooi, Jiawei Han, Hongjun Lu and Kian Leong Tan. Index nesting – an efficient approach to indexing in object-oriented databases. VLDB Journal, Volume 5, Number 3, pages 215–228, August 1996.


Appendix: Execution Example

The tables below provide snapshot information for selected variables and expressions. In each table, the first column refers to the line numbers given in Figures 14 and 16. In particular, the values in each table row correspond to the values of the traced expressions after the execution of the referenced line of code. Undefined expressions are denoted by "-". For notational convenience, the innermost set

<table>
<thead>
<tr>
<th>D</th>
<th>L</th>
<th>S'</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>AB {(ACDGH)(BEFH)}</td>
<td>∅</td>
<td>AB</td>
</tr>
</tbody>
</table>

After the call with \{ABCDEF\} and the initialization steps, \(L\) contains the subhierarchies of the maximal elements of \(S\) in separate lists. In the following illustrations, all processed types (i.e., types in \(D\)) are shaded.

<table>
<thead>
<tr>
<th>D</th>
<th>L</th>
<th>S'</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>AB {(ACDGH)(BEFH)}</td>
<td>∅</td>
<td>(BEFH)</td>
<td>-</td>
</tr>
<tr>
<td>8</td>
<td>AB {(ACDGH)}</td>
<td>∅</td>
<td>(BEFH)</td>
<td>-</td>
</tr>
<tr>
<td>9</td>
<td>AB {(ACDGH)}</td>
<td>∅</td>
<td>(BEFH)</td>
<td>(ACDGH)</td>
</tr>
<tr>
<td>11</td>
<td>AB {(BEF H ACDG)}</td>
<td>∅</td>
<td>(BEFH)</td>
<td>(ACDGH)</td>
</tr>
</tbody>
</table>

After a first concatenation operation, the situation is as depicted in the figure (the dashed lines connect buddies in a list). At this point, there is no further concatenation operation possible, so a refinement attempt is made for each list in \(L\).

<table>
<thead>
<tr>
<th>D</th>
<th>L</th>
<th>S'</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>AB {(BEF H ACDG)}</td>
<td>∅</td>
<td>(BEF H ACDG)</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>AB ∅</td>
<td>∅</td>
<td>(BEF H ACDG)</td>
<td></td>
</tr>
</tbody>
</table>

Refinement candidates are the maximal elements of \(\bigcup A_i \setminus D\) (see right hand side figure), in this case only D and E for \{(BEF \{H\} {ACDG})\}, since C is a leaf. The subhierarchy of D is \{DGH\}. It has a nonempty intersection with both, \{H\} and \{ACDG\}. So it is a possible operand for \(*\),

<table>
<thead>
<tr>
<th>D</th>
<th>L</th>
<th>S'</th>
<th>A</th>
<th>max(...)</th>
<th>x</th>
<th>S' ≤x</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>AB ∅</td>
<td>∅</td>
<td>(BEF H ACDG)</td>
<td>DE</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>15</td>
<td>AB ∅</td>
<td>∅</td>
<td>(BEF H ACDG)</td>
<td>DE</td>
<td>D</td>
<td>DGH</td>
</tr>
<tr>
<td>17</td>
<td>AB ∅</td>
<td>∅</td>
<td>(BEF H DG AC)</td>
<td>DE</td>
<td>D</td>
<td>DGH</td>
</tr>
<tr>
<td>18</td>
<td>ABD ∅</td>
<td>∅</td>
<td>(BEF H DG AC)</td>
<td>E</td>
<td>D</td>
<td>DGH</td>
</tr>
</tbody>
</table>

After refinement, A contains the sets \{BEF\}, \{H\}, \{DG\} and \{AC\} (shown right hand side, larger figure) which are processed by subsequent invocations of the recursive function. The only nontrivial invocation is for \{BEF\}, the result \{B(\{EF\})\} is depicted in the smaller figure at the right hand side. After termination of all four recursive decents, the resulting situation is given in the following table:

<table>
<thead>
<tr>
<th>D</th>
<th>L</th>
<th>S'</th>
<th>A</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>ABDE ∅</td>
<td>{({B({EF}) H DG AC})</td>
<td>({B({EF}) H DG AC})</td>
<td></td>
</tr>
</tbody>
</table>

Result: \{({B(\{EF\}) H DG AC})\}