The VLDB Journal
© Springer-Verlag 1997

# A configurable type hierarchy index for OODB

**Thomas A. Mueck, Martin L. Polaschek**

Abteilung für Data Engineering, Universität Wien, Rathausstr. 19/4, A-1010 Wien, Austria; e-mail: {mueck, polaschek}@ifs.univie.ac.at

Edited by E. Bertino. Received October 7, 1996 / Accepted March 28, 1997

**Abstract.** With respect to the specific requirements of advanced OODB applications, index data structures for type hierarchies in OODBMS have to provide efficient support for multiattribute queries and have to allow index optimization for a particular query profile.

We describe the *multikey type index* and an efficient implementation of this indexing scheme. It meets both requirements: in addition to its multiattribute query capabilities it is designed as a mediator between two standard design alternatives, key-grouping and type-grouping.

A prerequisite for the multikey type index is a linearization algorithm which maps type hierarchies to linearly ordered attribute domains in such a way that each subhierarchy is represented by an interval of this domain. The algorithm extends previous results with respect to multiple inheritance.

The subsequent evaluation of our proposal focuses on storage space overhead as well as on the number of disk I/O operations needed for query execution. The analytical results for the multikey type index are compared to previously published figures for well-known single-key search structures.

The comparison clearly shows the superiority of the multikey type index for a large class of query profiles.

**Key words:** OODB – Access methods – Indexing – Type hierarchies – Multiple inheritance

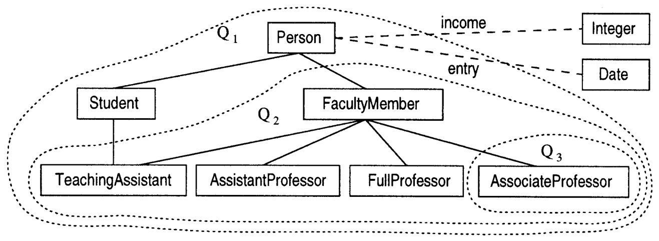
## 1 Introduction

The specific performance requirements of advanced OODB applications (e.g., CIM systems, geographical information systems) yield additional challenges for type hierarchy indices. In this context, multiattribute search structures are a recently discussed alternative (see [21], [11], [12], [19], [17], and [5]) to previously used B$^+$-tree structures.

We propose the so called *multikey type index* (MT-index) which supports multiattribute queries as well as data structure configuration based on query profiles. At the data structure level, the MT-index uses the concept of multiattribute search structures and provides a seamless integration of the type hierarchy by means of a linearization algorithm. Aiming at state-of-the-art OODBMS supporting multiple inheritance the algorithm is designed to handle this specific feature. Although our proposal does not rely on a *particular* multiattribute search structure, we provide an application example based on the hB-tree [14]. Other data structures like the BV-tree [7] or the hB$^{\Pi}$-tree [6] could be used without any modification of the framework. We justify our approach by a comparison of the corresponding performance figures to earlier published results ([16, 20]).

Previous work on type hierarchy indexing uses B$^+$-tree structures. In [13], the indexing performance of one B$^+$-tree per indexed type (called *Single Class Index*) is compared to the *Class Hierarchy Index* using one B$^+$-tree as a storage structure for all types (further enhanced by leaf node directories). The *Class Division* approach presented in [21] can be viewed as a mediator between these two alternatives. Using a replication scheme for OIDs, the class division approach trades storage space and update performance for query performance. So called *H-trees* are introduced in [16], the central idea is to represent the type hierarchy in the index structure by nested single-type B$^+$-trees. *CG-trees* [12] and *hcC-trees* [24] focus on a more general problem, namely set membership. Technically speaking, both approaches augment B$^+$-trees by multiple lists to group OIDs with respect to set membership. The *Nested Inherited Index* [2] and the *Generalized Nested Inherited Index* [23] are B$^+$-tree-based hybrid approaches supporting type membership as well as path expressions. Multi-attribute index structures as an alternative to B$^+$-tree structures in the context of type hierarchy indexing are discussed in [21], [11], [12], [19] and [17]. A summary of various indexing approaches in the realm of OODB can be found in [3] and [18].

The paper is structured as follows: the rest of this introductory section contains a detailed problem statement and an outline of the solution described later on. Section 2 deals with a selection of related work considered most relevant for the following presentation. Section 3 contains the formal definition of the term *optimal linearization*, Section 4 describes the hierarchy linearization algorithm as a prerequisite for the MT-index. Implementation issues are discussed in Sect. 5, in particular, a multiattribute search structure is used to demonstrate the potential of our approach. An an-



$Q_1$ select x from x in person where x.income $>$ 10000 and x.entry $<$ ”01-01-1990”

$Q_2$ select x from x in facultyMember where x.income $>=$ 20000 and x.income $<$ 30000

$Q_3$ select x from x in associateProfessor where x.income $<$ 45000 and x.entry = ”08-07-1992”

**Fig. 1.** Type hierarchy and example queries

|  | *type* | income | weight |
|---|---|---|---|
| $\omega 1$ | Student | 20000 | 70 |
| $\omega 2$ | TeachingAssistant | 30000 | 91 |
| $\omega 3$ | Student | 20000 | 65 |
| $\omega 4$ | AssistantProfessor | 30000 | 72 |
| $\omega 5$ | FullProfessor | 30000 | 85 |
| $\omega 6$ | AssistantProfessor | 29000 | 93 |
| $\omega 7$ | AssistantProfessor | 30000 | 105 |

**Fig. 2.** Database instances

alytical evaluation is presented in Sect. 6. This evaluation contains material on the index size as well as query performance in case of exact match and (partial) range queries. At the end of the paper, there are conclusions and references.

### 1.1 Problem statement

We address two independent technical issues in the context of object set indexing:

1. queries referring to type subhierarchies and
2. adaptive index organizations with respect to query profiles.

**ad 1**: Like in SQL, any query in an OODB may refer either to a single attribute or to several attributes at once (in the latter case, it is called *multiattribute* query). Additionally, any query in an OODB may refer either to one type or, implicitly, to a subhierarchy and therefore to a set of types. The type hierarchy and the queries of Fig. 1 illustrate the concepts of single-attribute and multiattribute queries as well as the implicit qualification of a subhierarchy by a type identifier. The first query is a single-attribute range query implicitly referring to all types of the hierarchy (person represents all objects of type Person *and* all objects of subtypes of type Person). The second query is a multiattribute range query, the referenced hierarchy is a subhierarchy, thus less object types than in the previous example qualify for the query. In the third query, a single type (i.e., AssociateProfessor) is addressed. Being a multiattribute query, a range is specified for attribute income, whereas an exact match is given for attribute entry.

As the example queries show, the indexing component of an OODBMS has to deal with multiattribute queries as well as with type hierarchy predicates.

**ad 2**: Focusing on the data structure implementation of a type hierarchy index, there are two design alternatives: key-grouping versus type-grouping approaches (see also [12] where the more general term *set grouping* was used for type-grouping):

– data structures based on *key-grouping* maintain a first-level data structure organization for key values. All object identifiers with the same key value (e.g., all the people with an income of 10,000) may or may not be further organized with respect to their types (second-level organization);
– *type-grouping* structures also maintain an asymmetric data organization, however, in this case the first-level order criterion is the object type (e.g., all objects of type Person are stored in one search data structure) and second-level search structures support key value access.

This design decision has a major influence on the resulting I/O performance in case of exact match and range queries, independent of the actual data structure implementation. In general, a key-grouping type hierarchy index supports exact match queries better than range queries, whereas for type-grouping structures, the reverse is true.

Following from this, an indexing component should incorporate the advantages of both key-grouping and type-grouping structures, depending on concrete query profiles.

Summing up, the technical problem tackled in this paper is the design of an efficient index data structure supporting single-attribute as well as multiattribute queries in type hierarchies. Additionally, the proposed index data structure has to be configurable with respect to key-grouping versus type-grouping. The first requirement is straightforward with respect to advanced OODB applications. The second requirement aims at an adaptable indexing framework in the sense that the importance of type-grouping increases with the fraction of range queries in the profile.

### 1.2 Proposed solution

The MT-index incorporates the type hierarchy structure of a given database scheme into a standard multiattribute search structure in such a way that the hierarchy is mapped to one of the attribute domains (called *type domain* in the sequel). The result is an index with $k + 1$ keys corresponding to the $k$ indexed object properties and to one additional key representing type membership. The principle is shown in Fig. 3. Looking, for example, at query $Q_2$, it qualifies instances of FacultyMember as well as instances of the subtypes of FacultyMember if these instances fulfill the query predicate. Consequently, this query contains an implicit predicate on the object type, i.e, x.type $\leq$ FacultyMember, where $\leq$ denotes the partial order relation of the type hierarchy. Using an MT-index, implicit type predicates are mapped to ranges of the type domain. An additional benefit of the type domain is the full control over the index data structure with

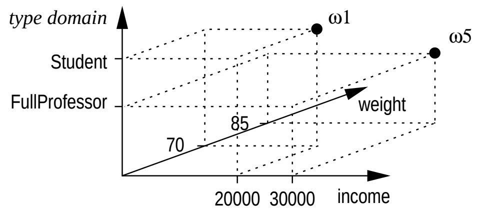


**Fig. 3.** 3D data space of an MT-index for two indexed properties (*income* and *weight*)

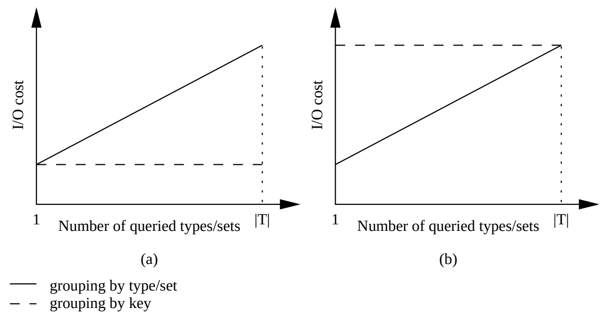


**Fig. 4a,b.** Efficiency of **a** exact match queries and **b** range queries [12]

respect to its type-grouping behavior. Basically, the partitioning strategy for the type domain immediately determines the "degree" of type-grouping. Inhibiting any partitioning favors key-grouping, whereas a forced total partitioning (i.e., one partition per type domain value) favors type-grouping. Any compromise between these two alternatives is possible.

An obvious prerequisite for this mapping is a linearization of the type hierarchy (see Fig. 3). However, since there are $|T|!$ linearizations for a particular type set $T$, one may ask if one linearization should be preferred to another. In what follows we

- show that some linearizations of a given $T$ are better than others with respect to the resulting query performance,
- present an algorithm which is able to find all optimal linearizations for a given type hierarchy,
- discuss an implementation using the well-known hB-tree, and
- justify the MT-index approach by an analytical performance comparison to $B^+$-tree-based index structures.

Summing up, the proposed solution is shown to be an interesting alternative in the context of type hierarchy indexing, especially when considering the indexing requirements of non-standard OODB applications.

## 2 Key grouping versus type grouping

Figure 4 shows the relationship between key-grouping and type-grouping on the one hand and exact match and range queries on the other hand (performance measured in number of I/O operations). Using key-grouping, the number of qualified types does not influence the I/O cost of a particular query, since the number of I/O operations is determined only by the range specified for the indexed attribute. On the contrary, in case of type-grouping, the number of qualified types is the main performance factor. For each qualified type, usually a type-specific data structure has to be scanned for matching key values. The relationship between key-grouping and type-grouping performance is as follows:

- Considering a particular exact match query, the I/O cost for any key-grouping structure will be constant and smaller or equal than the I/O cost for any type-grouping approach.
- On the contrary, for a particular range query, the I/O cost for any key-grouping structure will still be constant, however, larger or equal than the I/O cost for any type-grouping approach.

In the remainder of this section we present prominent examples of type-grouping and key-grouping index implementations. To justify our own proposal, we subsequently relate the performance of the MT-index (implemented by means of an hB-tree) to these $B^+$-tree approaches (cf. the figures in [16]).

### 2.1 Key grouping – the CH-index

A CH-index (*class-hierarchy index*) [13] maintains one search structure for all types of the indexed hierarchy and therefore implements a pure key-grouping strategy. The CH-index permits efficient single-scan access to the instances of all types of the indexed hierarchy. It outperforms any type-grouping index if a query qualifies the indexed type and all its subtypes or at least a major subset of the indexed hierarchy. However, if only a few types of the indexed hierarchy are qualified by a query, a type-grouping index (see below) performs better, because only a few (small) indices have to be scanned, in particular one for each type in the query scope.

$B^+$-trees are chosen as underlying data structure (see Fig. 5). An index record of a CH leaf node consists of record length, key length (for variable-length keys), key value, overflow page pointer, and the list of object identifiers of objects holding the key value in the indexed attribute. The object identifiers in the list are grouped by type. The key directory contains the offset for each type having objects in the list of object identifiers, thus speeding up intra-node lookup.

If a query refers to a large number of types, a small fraction of the OIDs found during the index traversal have to be discarded and vice versa. There are two extreme cases: if all types are qualified by the query, no OIDs will be out of query scope, whereas if a query refers only to one type, probably a large number of OIDs are fetched in vain.

Using this search structure, the whole $B^+$-tree has to be scanned in *all* cases, even if most of the entries in the leaf nodes have to be discarded in a subsequent processing step. In case of a query over the full type hierarchy, the CH-index is an attractive proposal for both range queries and exact match queries. If the query refers to a subhierarchy or to single types, exact match queries still perform well, whereas range query performance degrades drastically.

### 2.2 Type grouping – the SC-index and the H-tree

An early technique used in the ORION system is called *single class* index (SC-index) in [13]. Using single-class in-

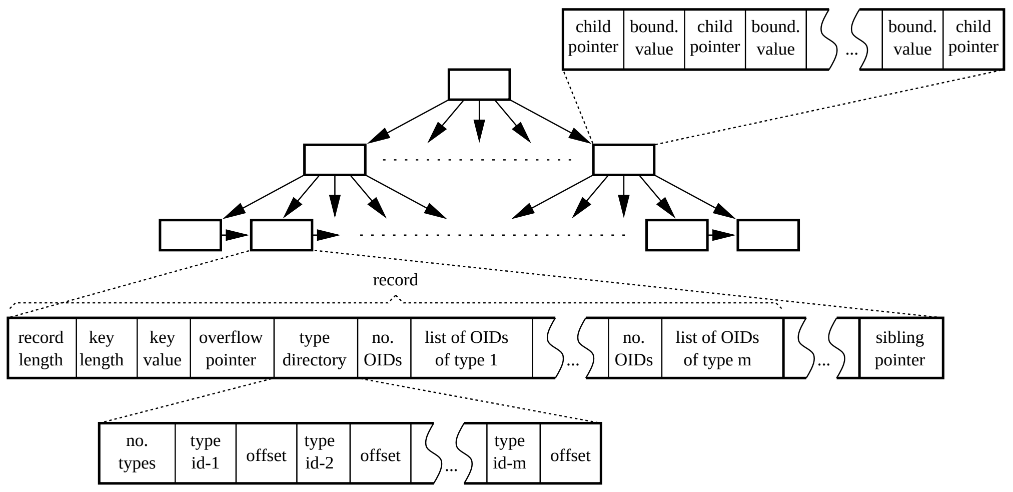


**Fig. 5.** CH-index structure

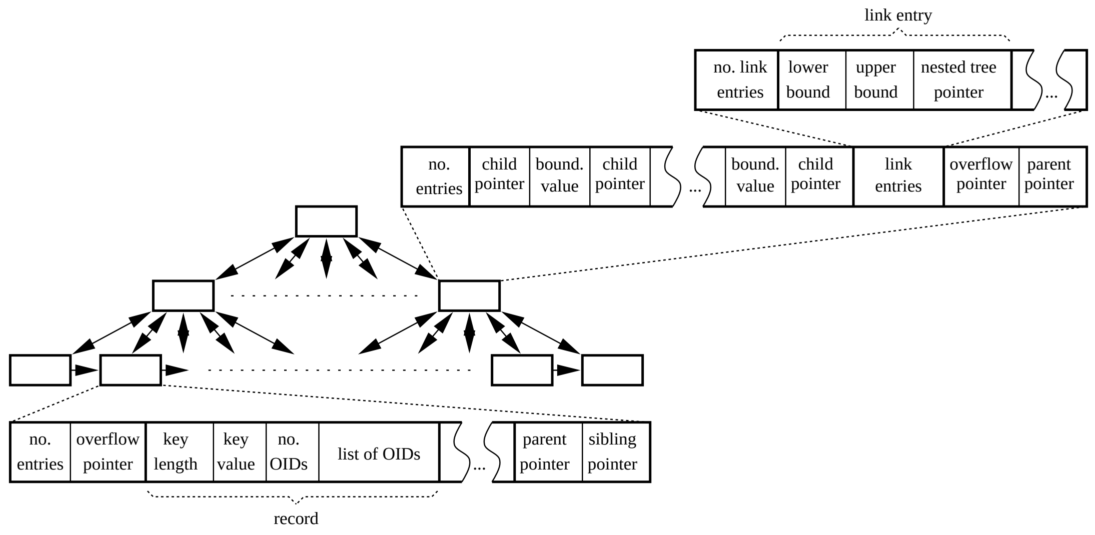


**Fig. 6.** H-tree structure

dexing, the index creation for an object property requires the construction of *one* B$^+$-tree *for each type* in the indexed hierarchy, thus implementing pure type-grouping.

Queries implicitly referring to large subhierarchies scan a large number of B$^+$-trees. Although several trees are traversed, the positive aspect is that all retrieved OIDs qualify with regard to the object type. A favorable case is a range query over one type. Exact match queries over more than one type are not favorable.

The H-tree [16, 15, 20] can be viewed as a direct successor of the SC-index. The main difference between the SC-index and the H-tree is that the former maintains a set of isolated type-specific B$^+$-trees, whereas the latter uses a nesting structure for these B$^+$-trees. The nesting reflects the structure of the indexed type hierarchy.

In particular, the H-tree component (i.e., the B$^+$-tree) of the indexed type is *nested* (see below) with the H-trees of the immediate subtypes of the indexed type, the H-trees of these types are nested with the H-trees of their respective subtypes and so forth. Thus, an H-tree index for an attribute in an inheritance subgraph is a hierarchy of H-trees nested according to the supertype-subtype relationship. The references used to establish the nesting are used for traversal shortcuts during query execution.

The motivation behind index nesting is to avoid *full* scans of *each* H-tree component when a number of types in the indexed hierarchy is queried. In particular, when scanning the trees of a type and its subtypes, one needs to perform a full search in the H-tree component of the supertype (called outer component) and only partial searches in the H-trees of the subtypes (called inner components). This restriction of the traversal process is the major advantage of the H-tree compared to the SC-index.

Figure 6 shows the organization of an H-tree component. H-tree leaf nodes closely resemble SC-index leaf nodes. Data item *key value* holds one value of the indexed object property. Inner nodes contain interval boundary values and pointers to successor nodes (child pointers) like in standard B$^+$-trees. The link entries implement the nesting feature. Each link entry contains a pointer to a subtree of the H-tree component of a subtype. Additional parts of the link entry are the boundary values of this nested subtree. As the number of

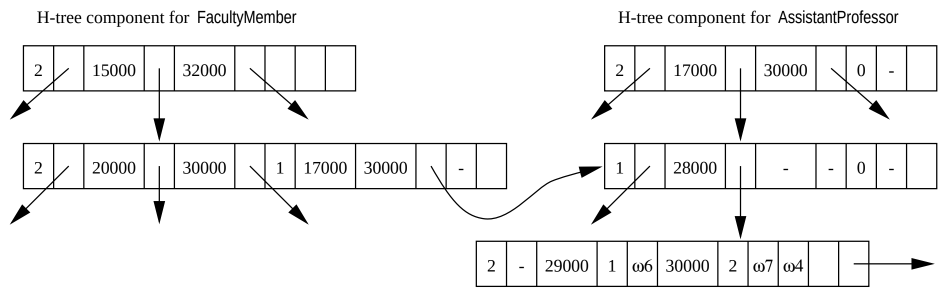

H-tree component for FacultyMember
H-tree component for AssistantProfessor

**Fig. 7.** H-tree component for AssistantProfessor

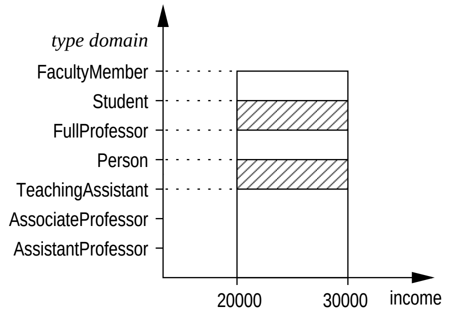


**Fig. 8.** Query volume for $Q_2$ under suboptimal linearization

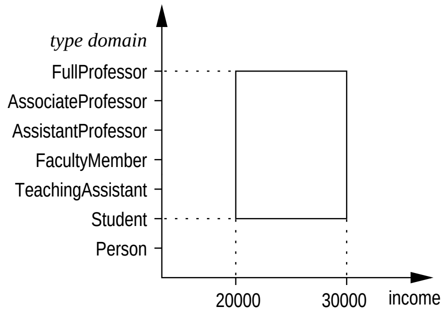


**Fig. 9.** Query volume for $Q_2$ under optimal linearization

link entries per node is not restricted, overflow pages (organized with the help of overflow pointers) may be necessary to store additional pointers. Access to the predecessor of a particular node is supported by parent pointer entries in the H-tree nodes.

Figure 7 shows parts of an H-tree created for attribute income of type Person and its subtypes. The figure shows part of the H-tree component for type *FacultyMember* with one link entry with lower boundary = $17,000$ and upper boundary = $30,000$. The nested tree pointer refers to a node of the H-tree component for type AssistantProfessor. The positive aspect of the H-tree approach is the exclusion of a number of inner tree nodes from the tree traversal during query processing. The problems are a decreased node fanout due to the space requirements of the link entries on the one hand and complex query and update algorithms on the other hand.

## 3 Type hierarchy mapping

As already mentioned in Sect. 1, queries contain implicit predicates on the object type. In an MT-index such a predicate corresponds to a range of the type domain. The reason for the following considerations is that the query performance of an MT-index is largely determined by the choice of the actual type hierarchy linearization. Figures 8 and 9 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 Fig. 8). 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 Fig. 9 for the extent of FacultyMember). This means a

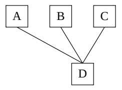


**Fig. 10.** Hierarchy without optimal linearization

linearization in such a way that exactly one interval contains all types which are part of one subhierarchy.

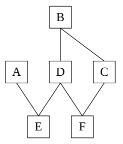
This yields for each possible type in a query a subspace which does not contain any object identifiers not belonging to the query result. Consequently, a type domain setup (linearization) resulting in minimal query subspace volumes for all possible query requests is called *optimal*. More specifically, an ordering $\sqsubseteq$ is optimal for $(T, \leq)$, if $\sqsubseteq$ 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 $(T, \sqsubseteq)$, 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 $]\infty, t]$ in $(T, \leq)$). An ordering $\sqsubseteq$ is called* optimal linearization *for $(T, \leq)$, if*

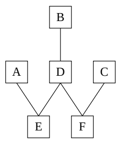
$$\forall t, u \in T: \ t \sqsubseteq u \ \vee \ u \sqsubseteq t \quad \textit{and} \tag{1}$$

$$\forall t \in T: \ \exists u, v \in \ T_{\leq t} \ \textit{such}$$
$$\textit{that} \ \ [u, v]_{(T, \sqsubseteq)} = T_{\leq t}\,. \tag{2}$$

There are type hierarchies without optimal linearization. Figure 10 shows the smallest hierarchy for which such a linearization does not exist. Although stating a necessary and sufficient condition for the existence of an optimal linearization for a type hierarchy is not totally trivial, a closer look at the above definition yields at least one necessary and one sufficient condition ($super(t)$ denoting the set of direct supertypes of $t$):

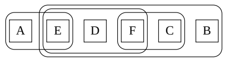


(a)

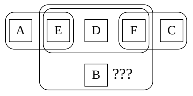


(b)

**Fig. 11a,b.** Type hierarchies (a) with and (b) without optimal linearization



(a)



(b)

**Fig. 12a,b.** Set diagrams for type hierarchies of Fig. 11

– An optimal linearization exists if each type has at most one supertype, i.e., in the case of single inheritance ($\forall t \in T : |super(t)| \leq 1$ is sufficient).
– An optimal linearization does not exist if any type has more than two supertypes ($\forall t \in T : |super(t)| \leq 2$ 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 [21]. Additional work dealing with depth-first linearization can be found in [8].

In the multiple inheritance case, Fig. 11 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 Fig. 12) 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 Fig. 12 this is {BDEF}).

In the following section, we present an algorithm which produces all optimal linearizations for a given hierarchy $(T, \leq)$. The novel feature of this algorithm is its ability to cope with multiple inheritance hierarchies.

## 4 The mapping algorithm

Readers mainly interested in the performance comparison may skip this entire section.

### *4.1 Notation, operators and linearization function*

#### 4.1.1 Notation

The main part of the proposed algorithm is a recursive function *order*. Another integral part of this algorithm is a *structured set* $\boldsymbol{S}'$ constructed during the traversal of $(T, \leq)$. Elements of $\boldsymbol{S}'$ are either atoms (i.e., type identifiers) or *structured lists*. Elements of structured lists are 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.

```
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 $\boldsymbol{L}$ 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 $\boldsymbol{S}'$ which are not member
      of any subhierarchy
7.    foreach element of $\mathcal{A}$ of $\boldsymbol{L}$ do
8.       remove $\mathcal{A}$ from $\boldsymbol{L}$
9.       if there exists an element $\mathcal{B}$ of $\boldsymbol{L}$ such that
         there is a $B_j$ of $\mathcal{B}$ and an $A_i$ of $\mathcal{A}$
         with non-empty intersection then
10.         if $\mathcal{A} \circ \mathcal{B}$ is defined then
11.            replace $\mathcal{B}$ by $\mathcal{A} \circ \mathcal{B}$ in $\boldsymbol{L}$
12.         else there is no solution
13.         end
14.      else
15.         while there exists an unmarked maximal element $x$ such that
            there are at least 2 elements of $\mathcal{A}$ having
            a non-empty intersection
            with the subhierarchy rooted at $x$ do
16.            if $\mathcal{A} * S_{\leq x}$ is defined then
17.               refine $\mathcal{A}$ with $S_{\leq x}$, i.e., with the subhierarchy rooted
                     at $x$
18.               mark $x$
19.            else there is no solution
20.            end
21.         end
22.         call order (recursively) for each element of $\mathcal{A}$ and add the list
            of all results to $\boldsymbol{S}'$
23.      end
24.   end
25.   return $\boldsymbol{S}'$
26. end order
```

**Fig. 13.** Pseudocode for function *order*

The following notational conventions for variables hold: variables for flat sets like {ABCD} are denoted by $A, B, C, \ldots$, for structured sets like {({AB}{C})D} by $\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}, \ldots$, for flat lists like ({AB}{C}{D}) by $\mathcal{A}, \mathcal{B}, \mathcal{C}, \ldots$ and for atoms by $a, b, c, \ldots$. There are no variables used for structured lists.

#### 4.1.2 Operations

Operations and symbols subsequently used are:

– $\{\}$, () 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 ($||$), intersection ($\cap$) and set membership $\in$.

| $i$ | $j$ | $A \circ B$ |
|---|---|---|
| $\lvert A \rvert$ | 1 | $(A_1, \cdots, A_{\lvert A \rvert-1}, A_{\lvert A \rvert} \setminus B_1, A_{\lvert A \rvert} \cap B_1, B_1 \setminus A_{\lvert A \rvert}, B_2, \cdots, B_{\lvert B \rvert})$ |
| $\lvert A \rvert$ | $\lvert B \rvert$ | $(A_1, \cdots, A_{\lvert A \rvert-1}, A_{\lvert A \rvert} \setminus B_{\lvert B \rvert}, A_{\lvert A \rvert} \cap B_{\lvert B \rvert}, B_{\lvert B \rvert} \setminus A_{\lvert A \rvert}, B_{\lvert B \rvert-1}, \cdots, B_1)$ |
| 1 | 1 | $(A_{\lvert A \rvert}, \cdots, A_2, A_1 \setminus B_1, A_1 \cap B_1, B_1 \setminus A_1, B_2, \cdots, B_{\lvert B \rvert})$ |
| 1 | $\lvert B \rvert$ | $(A_{\lvert A \rvert}, \cdots, A_2, A_1 \setminus B_{\lvert B \rvert}, A_1 \cap B_{\lvert B \rvert}, B_{\lvert B \rvert} \setminus A_1, B_{\lvert B \rvert-1}, \cdots, B_1)$ |

**Fig. 14.** Concatenation operator (formal definition)

– The operators $\cup$, $\setminus$ and $\in$ are also defined for the top level of structured sets.
– $\lvert A \rvert$ denotes the number of flat sets contained in the flat list $A$, which are denoted by $A_1, A_2, \ldots, A_{\lvert A \rvert}$.
– *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.,

$$\max(A, \leq) \mapsto \{a \in A \mid \nexists a' \in A : a < a' \\ \wedge\ \exists a'' \in A : a'' < a\}\,.$$

– $\circ$ 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 $\bigcup A_i \cap \bigcup B_i \neq \emptyset$. All sets in such a list have to be non-empty and pairwise disjoint.
It should be noted that $\circ$ is defined if and only if $!\exists(i,j) : A_i \cap B_j \neq \emptyset, i \in \{1, \lvert A \rvert\}, j \in \{1, \lvert B \rvert\}$. 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 four cases as depicted in Fig. 14 (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.
– $*$ represents refinement. $A * B$ is defined if and only if $A$ denotes a flat list of pairwise disjoint and non-empty sets, $B$ denotes a flat set, $!\exists(i,j) : i < j$ and

$$\forall k, 1 \leq k \leq \lvert A \rvert : \begin{cases} A_k \cap B = \emptyset & \text{for } k < i \\ A_k \cap B \neq \emptyset & \text{for } k = i \\ A_k \subset B & \text{for } i < k < j \\ A_k \cap B \neq \emptyset & \text{for } k = j \\ A_k \cap B = \emptyset & \text{for } k > j \end{cases}$$

If $A * B$ is defined, the result is given by (again, empty sets are removed from the result):

$$A * B \mapsto (A_1, \cdots, A_{i-1}, A_i \setminus B, A_i \cap B, \\ A_{i+1}, \cdots, A_{j-1}, A_j \cap B, A_j \setminus B, \\ A_{j+1}, \cdots, A_{\lvert A \rvert})$$

Example: ({B} {CDE} {FG} {H}) $*$ {DEF} yields ({B} {C} {DE} {F} {G} {H}). ({B} {CDE} {FG} {H}) $*$ {DEGH} is undefined, since {CDE}$\cap${DEGH}$\neq \emptyset$ and {H}$\cap${DEGH}$\neq \emptyset$ and {FG}$\not\subset${DEGH}.

### 4.1.3 Function *order*

A pseudocode representation of function *order* is given in Fig. 13. The exact definition of this function together with an execution trace for an example hierarchy can be found in the Appendix. After termination of the algorithm, a postprocessing step on the result produces all optimal linearizations (see below).

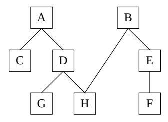


**Fig. 15.** Example type hierarchy

In a wrapping procedure, a set $D$ is initialized as empty set. Its purpose is to hold already processed (marked) type identifiers. The same wrapping procedure passes $T$ as actual parameter to the initial call to function *order*.

For a given type hierarchy $(T, \leq)$, e.g., $T$ = {ABCDEFG} with $\leq$ shown in Fig. 15, after termination of *order* the result contained in $\boldsymbol{S}'$ can be used to construct all optimal orderings. For the above example, the value of $\boldsymbol{S}'$ is {({B({EF})}{H}{DG}{AC})}. The set of all optimal linearizations 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({EF})} containing one atomic element B and a list ({EF}) as second element. It should be noted that this list contains only one element, namely set {EF}, so there is only one correct representations of this list. The list containing four elements, i.e., {B({EF})}, {H}, {DG} and {AC}, has two correct representations. All in all a simple postprocessing traversal yields 2! · 2! · 2 · 2! · 2! = 32 optimal linearizations for $T$, e.g., B-E-F-H-D-G-A-C, B-F-E-H-D-G-A-C, E-F-B-H-D-G-A-C, etc.

Applying the algorithm to the hierarchy of Fig. 1 results in {Person, ({FacultyMember, AssistantProfessor, FullProfessor, AssociateProfessor} {TeachingAssistant} {Student})} thus giving 2! · 2 · 4! = 96 optimal linearizations.

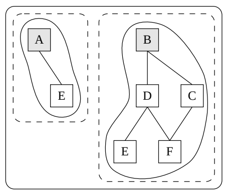
Like in the case of most other graph algorithms, the runtime function of the linearization algorithm depends not only on $\lvert S \rvert$ but also on the number of edges in the inheritance hierarchy. Additionally, the number of invocations of $\circ$ and $*$ depends heavily on the specific structure of the hierarchy (e.g., in case of a single inheritance hierarchy, neither the refinement operator nor the concatenation operator are invoked at all). Although a formal derivation of the runtime function is beyond the scope of the paper, empirical experiments show that the resource consumption for practically relevant hierarchy sizes is almost negligible. For example, a C++ implementation of the algorithm needed 0.013 s CPU time for a 52-node 3-way tree on an outdated 133-MHz Pentium® running Solaris™. The linearization of a 52-node multiple inheritance hierarchy including various invokations of $\circ$ and $*$ needed 0.024 s CPU time.[1]

[1] The results were obtained by calling the linearization algorithm 10,000 times and dividing the CPU time figures accordingly.

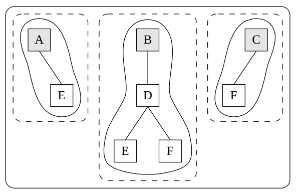
### *4.2 Informal linearization example*

The example hierarchies of the previous section (see Fig. 11) are used 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:



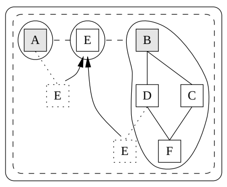
Hierarchy (a)



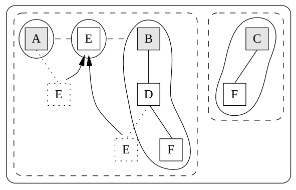
Hierarchy (b)

   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 $\circ$, see example Fig. 16). 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.

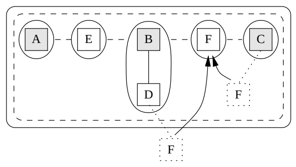


Hierarchy (a) Step 1



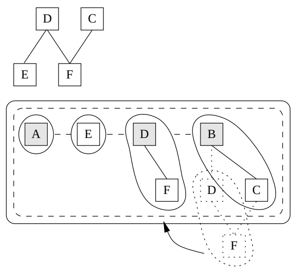
Hierarchy (b) Step 1

   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.

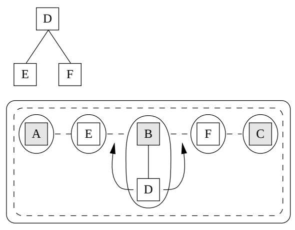


Hierarchy (b) Step 2

3. The lists resulting from the previous step are refined (operator $*$, see example Fig. 5.1), more specifically, for each subhierarchy rooted at an unmarked maximum, it is checked, whether or not the subhierarchy has a non-empty 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.
   The reason for such a refinement is obvious when looking at hierarchy (a) after the application of the concatenation operator. It is easy to show that an immediate recursive descent with the rightmost list element {BCDF} would produce incorrect results.
   Considering hierarchy (a), there are two candidates (subhierarchies) for refinement: {CF} and {DEF}. However, {CF} has a non-empty intersection with only one list element, i.e., {BCDF}. Consequently, no refinement is done with {CF}. {DEF} has non-empty intersections with both {BCDF} and {E}, the result of the refinement is given below.

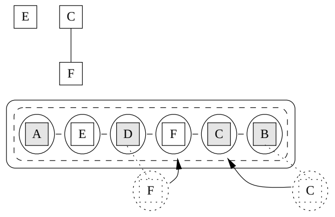


Hierarchy (a)



Hierarchy (b)

   In case of hierarchy (b) the only refinement candidate is {DEF}. This subhierarchy has a non-empty 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 non-empty intersections with {DF} and {BC}. We arrive at a configuration like:



Hierarchy (a)

4. Using the $\circ$ 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 cause 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.

## 5 Implementation issues

In this section we apply the linearization algorithm for the purpose of type hierarchy indexing. In particular, the im-

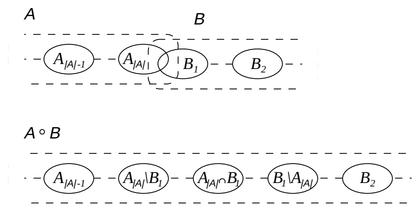


**Fig. 16.** Concatenation operation (informally)

plementation of an MT-index with the help of optimal linearizations is outlined.

### 5.1 Boundary structure and value structure

In general, an MT-index can be built using *any* multiattribute search structure. Consequently, this section is not focused on any particular data structure like, e.g., the BV-tree [7], hB-tree [14], or the hB$^{\Pi}$-tree [6]. The only data structure requirement is a non-degenerating behavior in case of data skew.

Multiattribute search data structures interpret $m$-tuples as elements of an $m$-dimensional geometrical space. Physically stored tuples have to be enclosed by an $m$-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 data space is successively partitioned into an increasing number of subspaces as the number of stored tuples increases.

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 *symmetrical* index structures without any distinction between one clustering and $m - 1$ non-clustering data structures for $m$ 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 multiattribute 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, in our application context, 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 a particular value combination of the indexed object properties. 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.

The execution of query requests with the help of an MT-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 and
– 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.

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,

```
Q4  select x from x in facultyMembers
    where x.income < 50 000 and x.entry < "04-10-1989"
```

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

The splitting strategy for the type domain can be adapted to a concrete query profile. The domain split potential is determined by the data volume: the minimum data page occupancy (e.g., 0.66) and the given raw data volume determine the number of data pages, which in turn determines the number of domain splits (see Sect. 6.2). Consequently, the only tunable parameter is the relative number of splits allocated to each domain. A larger number of splits in one domain implies a smaller number of splits in one or more of the other domains.

With respect to the limited total number of domain splits, an MT-index could increase the number of splits in the type domain (thus increasing the degree of type-grouping)

– if the fraction of queries referring to subhierarchies is large compared to the fraction of queries referring to the entire indexed hierarchy and
– if the number of types in the qualified subhierarchies is typically small compared to the total number of types in the indexed hierarchy.

### 5.2 The hB-Tree as MT-index

The hB-tree [14] is a multiattribute search structure with guaranteed index and data node occupancy for arbitrary raw data distributions. Figure 18 shows an hB-tree. The three rectangles represent internal hB-tree nodes. The leaf nodes (denoted by capital letters) are not shown in this figure. The boundary values contained in internal hB-tree nodes are organized as k-d-trees [1]. With respect to leaf node organization (value structure), we present two alternatives which are orthogonal to possible leaf node data structures like linked lists or k-d-trees as proposed in [14]: Figure 19a shows a leaf node organization using a directory. For each occurring combination of attribute values and type identifier the

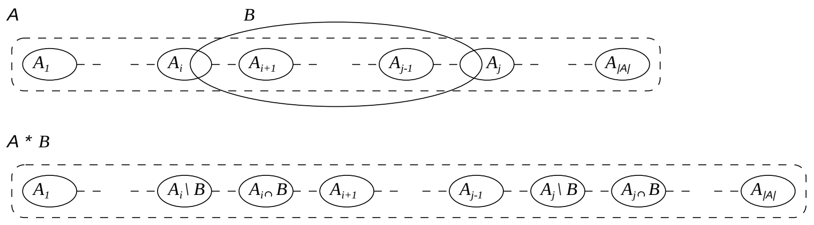


**Fig. 17.** Refinement operation (informally)

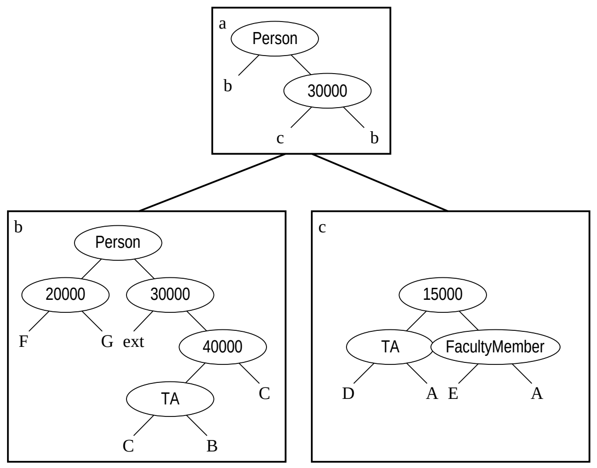


**Fig. 18.** The hB-tree used as MT-index

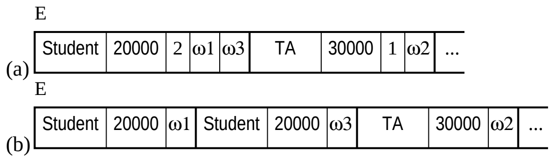


**Fig. 19.** Leaf node organizations

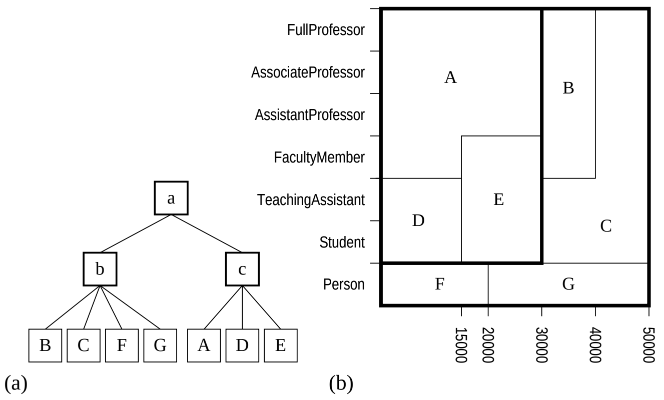


**Fig. 20.** Visualization and data structure representation of the hB-tree

**Table 1.** Model parameters

| *Param.* | *Meaning* |
|---|---|
| $T$ | Set of indexed types |
| $T^Q \subset T$ | Set of types referenced by a query $Q$ |
| $k$ | Number of indexed attributes |
| $d^Q \leq d$ | Number of attribute values qualified by query $Q$ |
| $n$ | Number of objects |
| $n_i$ | Number of objects of type $t_i \in T$ |
| $f$ | fanout: number of successors of an internal node |
| size($x$) | Size of $x$ in bytes |
| $e$ | Number of records in an index leaf node |
| $n_{vt}$ | Number of objects per value per type |
| $b_i^L$ | Number of index leaf nodes for $t_i$ |
| $b_i^I$ | Number of internal index nodes for $t_i$ |
| $b_i$ | Total number of index nodes for $t_i$ |

list of corresponding OIDs is stored. The counter is used to calculate the offset to the next values/type combination. Figure 19b shows a leaf node organization without directory. In this case, leaf nodes contain one record per indexed object with OID, type, and $k$ attribute values. Informally, the advantage of a leaf node directory decreases with increasing $k$ and increasing attribute domain cardinalities, since particular combinations become less likely, i.e., lists of OIDs become shorter.

Continuing the example of Fig. 18, Fig. 20 depicts the resulting data space partitioning with the corresponding high-level representation of the hB-tree. Additionally, a closer look at this figure leads to another technical issue: temporarily sacrificing the minimum node occupancy, one could split the type domain to the full extent, i.e., one region boundary per type *in advance*. Clearly, such a pre-splitting scheme allows the omission of the type identifiers from the leaf node records, thus saving index space.

## 6 Performance analysis and comparison

Index size and query performance of the MT-index are compared to the analytical performance results of the CH-index and the H-tree. As usual, there is no dominating approach for all hierarchy structures and query profiles; however, we are able to provide a few rules of thumb for index selection.

As in [16] and [13] and most other analytical approaches, our estimations are based on *best case* assumptions, e.g., maximum index page occupancy. The corresponding average case figures do not yield significantly different results (see [14] and [20]). A central parameter of the following models (see [13] and [16]) is denoted by $nt$ and describes the *attribute configuration*: each value of the indexed attribute occurs in objects of $nt$ types. Let $d$ and $d_i$ denote the number of different values stored in the indexed attribute and the number of different values stored in objects of type $t_i$, respectively. The relationship between these three parameters is described by $d_i = d \cdot \frac{nt}{|T|}$ with $1 \leq nt \leq |T|$. We interpret the two extreme values of $nt$: $nt = 1 \Rightarrow d = \sum_{|T|} d_i$ and $nt = |T| \Rightarrow d = d_i$. In the first case, each attribute value occurs in objects of exactly one type, whereas in the second case each attribute value occurs in objects of all types. The former situation is referred to as *no overlap*, the latter as *full overlap*, and any $nt$ value between 1 and $|T|$ as *partial overlap*.

All other parameters are described in Table 1. Storage space assumptions made for particular data items are given in

**Table 2.** Size of data items

| *Data item* | *Size* | *Meaning* |
|---|---|---|
| *Node* | 4096 | Index node (internal or leaf) |
| *Counter* | 2 | Counter used for OIDs, node entries, ... |
| *Att* | 4 | Indexed attribute |
| *OID* | 12 | Object identifier |
| *Offset* | 2 | Offset within a node |
| *TypeId* | 2 | Type identifier |
| *NodeId* | 8 | Node identifier |

Table 2. *Rec* and *Dir* refer to leaf node records and internal directories maintained in leaf nodes, respectively. In what follows, the analytical models for the single-key structures are based on slightly modified material presented in [16].

## 6.1 Storage space requirements

The results of two experiments focusing on the index size are presented in Sects. 6.1.2 and 6.1.3. The evaluation results on the analytical model are given in Sect. 6.1.1. In both experiments the number of indexed types is varied. The difference between the two experimental settings is that, in the first case, a constant number of objects per type is assumed, whereas in the second case a constant total number of objects is distributed over a variable number of types.

Both experiments are made for $k = 1$ and $k = 2$ (i.e., one indexed attribute in contrast to two indexed attributes), as well as for $nt = 1$ (no overlap), $nt = \frac{|T|}{2}$ (partial overlap), and $nt = |T|$ (full overlap). For $k = 1$, a leaf node organization (grouping by value and type) is used for the hB-tree, whereas for $k > 1$ this additional organization is omitted.

### 6.1.1 Analytical model

The CH-index is a balanced multiway tree featuring a sophisticated leaf node organization: in a leaf node, object IDs are grouped by attribute value and object type. In contrast to this approach, the nesting of an H-tree corresponds to the structure of the type hierarchy, therefore, a type-oriented leaf node organization is obsolete. Due to the structure of the CH-index (in contrast to the H-tree), the parameters $b$, $b^L$, and $b^I$ are used without subscript $i$.

$$\begin{aligned}\text{size}(\mathit{Dir}) &= \text{size}(\mathit{Counter}) + nt \cdot (\text{size}(\mathit{TypeId}) \\ &\quad + \text{size}(\mathit{Offset})) \\ \text{size}(\mathit{Rec}^{(CH)}) &= \text{size}(\mathit{Att}) + \text{size}(\mathit{Dir}) \\ &\quad + nt \cdot (\text{size}(\mathit{Counter}) + n_{vt} \cdot \text{size}(\mathit{OID}))\end{aligned}$$

$$\begin{aligned}e^{(CH)} &= \left\lfloor \frac{\text{size}(\mathit{Node}) - \text{size}(\mathit{Counter})}{\text{size}(\mathit{Rec}^{(CH)})} \right\rfloor \\ b^L &= \left\lceil \frac{d}{e^{(CH)}} \right\rceil \\ b^I &= \left\lceil \frac{b^L}{f^{(CH)}} \right\rceil + \left\lceil \frac{\left\lceil \frac{b^L}{f^{(CH)}} \right\rceil}{f^{(CH)}} \right\rceil + \cdots + 1\end{aligned}$$

with

$$f^{(CH)} = \left\lfloor \frac{\text{size}(\mathit{Node})}{\text{size}(\mathit{Att}) + \text{size}(\mathit{NodeId})} \right\rfloor$$

Maintaining one CH-index for each indexed attribute[2] yields a total storage space consumption for a CH-index implementation of

$$b^{(CH)} = k \cdot (b^L + b^I)\,.$$

In the case of the H-tree, we have to calculate the size by means of the overall sum for all $|T|$ nested type-specific multiway trees. Consequently, we obtain for this data structure

$$\begin{aligned}\text{size}(Rec^{(H)}) &= \text{size}(\mathit{Att}) + \text{size}(\mathit{Counter}) \\ &\quad + n_{vt} \cdot \text{size}(\mathit{OID})\end{aligned}$$

$$\begin{aligned}e^{(H)} &= \left\lfloor \frac{\text{size}(\mathit{Node}) - \text{size}(\mathit{Counter})}{\text{size}(\mathit{Rec}^{(H)})} \right\rfloor \\ b_i^L &= \left\lceil \frac{d_i}{e^{(H)}} \right\rceil \\ b_i^I &= \left\lceil \frac{b_i^L}{f^{(H)}} \right\rceil + \left\lceil \frac{\left\lceil \frac{b_i^L}{f^{(H)}} \right\rceil}{f^{(H)}} \right\rceil + \cdots + 1 \\ b_i &= b_i^L + b_i^I\end{aligned}$$

with

$$f^{(H)} = \left\lfloor \frac{2}{3} \cdot \frac{\text{size}(\mathit{Node})}{\text{size}(\mathit{Att}) + \text{size}(\mathit{NodeId})} \right\rfloor$$

The reduced fanout is caused by the link entries. These tuples (consuming one third of the internal nodes) are used to implement the nesting, see above. Again, we assume one H-tree per indexed attribute, and consequently estimate a storage space consumption of

$$b^{(H)} = k \cdot \sum_{t_i \in T} b_i\,.$$

In the case of an hB-tree used to implement an MT-index we have to distinguish between the above-mentioned leaf node organizations (see Sect. 5.2, excluding the pre-splitting alternatives):

– grouping by attribute value and type:

$$\begin{aligned}\text{size}(\mathit{Rec}^{(hB)}) &= k \cdot \text{size}(\mathit{Att}) + \text{size}(\mathit{TypeId}) \\ &\quad + \text{size}(\mathit{Counter}) + n_{vt} \cdot \text{size}(\mathit{OID})\end{aligned}$$

$$\begin{aligned}e^{(hB)} &= \left\lfloor \frac{\text{size}(\mathit{Node}) - \text{size}(\mathit{Counter})}{\text{size}(\mathit{Rec}^{(hB)})} \right\rfloor \\ b^L &= \left\lceil \frac{n}{e^{(hB)} \cdot n_{vt}} \right\rceil\end{aligned}$$

[2] In this model, we assume that the execution of a $k$-attribute query results in the traversal of $k$ single-key indices. A possible alternative is the traversal of *one* index structure and the subsequent fetching of the referenced objects. However, objects not qualifying for the query (i.e., objects with non-matching values in the other attributes) have to be discarded afterwards. The problem in this context is the significant waste of I/O bandwidth caused by the transfer of non-qualifying objects.

– no grouping:

$$\text{size}(Rec^{(hB)}) = k \cdot \text{size}(Att) + \text{size}(TypeId) + \text{size}(OID)$$

$$e^{(hB)} = \left\lfloor \frac{\text{size}(Node) - \text{size}(Counter)}{\text{size}(Rec^{(hB)})} \right\rfloor$$

$$b^L = \left\lceil \frac{n}{e^{(hB)}} \right\rceil$$

In the following evaluation, the first variant is chosen for the case of one indexed attribute and the second variant in case of more than one indexed attribute. At this point, a closer look at the fanout of hB-tree nodes is needed. The internal nodes are organized as k-d-trees [1]. Each k-d-tree node contains one attribute value, two node identifiers of successor nodes, and two additional bytes of maintenance information. The resulting fanout

$$f^{(hB)} = \left\lfloor \frac{\text{size}(Node)}{(\text{size}(Att) + 2 \cdot \text{size}(NodeId) + 2) \cdot r} \right\rfloor$$

contains a reduction factor $r$ quantifying the overhead of the k-d-tree organization. In particular, a number of references in the k-d-tree again reference k-d-tree nodes belonging to the same hB-tree node. These references do not contribute to the fanout of the hB-tree node. In [14], an evaluation yields $1 \leq r \leq 1.5$, in our case variations of $r$ do not yield significantly different results. Based on this fanout, we obtain

$$b^I = \left\lceil \frac{b^L}{f^{(hB)}} \right\rceil + \left\lceil \frac{\left\lceil \frac{b^L}{f^{(hB)}} \right\rceil}{f^{(hB)}} \right\rceil + \cdots + 1\,,$$

$$b^{(hB)} = b^L + b^I\,.$$

### 6.1.2 Experiment 1: constant number of objects per type

For fixed $n_i$ and $d$, $|T|$ is increased. In other words, for a constant number of objects per type new types are added. The underlying assumption is that the creation of a new type more or less implies the insertion of new database objects of this type. Actual experimental values are $n_i = 10,000$, $d = 10,000$ and $1 \leq |T| \leq 50$. Under these assumptions, the database population $n$ is increased from $10,000$ to $500,000$. The resulting total number of index blocks for the CH-index, the H-tree, and the MT-index are depicted in Fig. 21. In the one-dimensional case, the storage space consumption of the three approaches is more or less the same, for overlapping attribute configurations, the H-tree dominates the MT-index by about 5% in the case of partial overlap and by about 8% in the case of full overlap.

Unsurprisingly, for $k > 1$, the MT-index outperforms the single-key approaches with respect to index size. The space overhead reduction is most appealing for the case of full overlap and less impressive in the case of a non-overlapping attribute configuration.

### 6.1.3 Experiment 2: constant total number of objects

Although the setup for this experiment seems almost the same as in Experiment 1, the application context is totally different: For fixed $n$ and $d$, $|T|$ is increased, i.e., for a constant number of objects the number of types is increased. This corresponds to the creation of new types and a subsequent migration of existing objects to the newly created types. Actual experimental values are $n = 500,000$, $d = 10,000$ and $1 \leq |T| \leq 50$. Results are presented in Fig. 22.

In the case of increasing $|T|$ with fixed $n$, the influence of $nt$ is even stronger than in the case of increasing $n$ and $|T|$. Again, discussing the single-attribute case first, the OIDs in hB-tree nodes are grouped by combinations of attribute value and type. Consequently, in the non-overlapping case, the size of the hB-tree is constant, because existing attribute values and the corresponding OIDs are assigned to the newly created types, i.e., $n_{vt}$ is constant. The same holds for the CH-index, but not for the H-tree, where the creation of new index trees causes a slightly increased storage overhead.

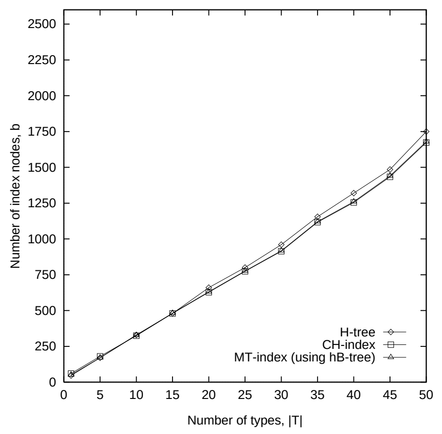
In case of partial and full overlap, the ranking between CH-index and H-tree changes, the MT-index is somewhere in between. In case of partial overlap, the advantage of the H-tree over the MT-index is upper bound by about 5% (9% in case of full overlap). The result for the CH-index is caused by the comparatively small number of records per leaf node (6 records for $|T| \leq 12$, 5 for $12 < |T| \leq 35$ and 4 for $|T| > 35$).

The results for two indexed attributes (see Fig. 22) are almost self-explanatory. The overall index size yielded by the single-key approaches is simply $k$ times the index size of the single-attribute case. Due to the robust directory organization of the hB-tree, the index size does not increase if the number of stored OIDs does not increase. As there is no additional hB-tree leaf node organization for $k > 1$, neither the attribute configuration nor $|T|$ has any impact on the index size.

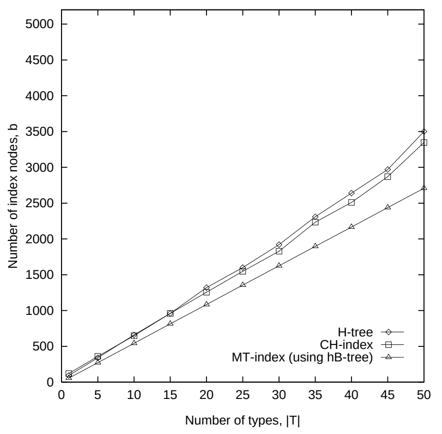
## *6.2 Query performance*

Four experimental settings are used to evaluate the query performance of the MT-index. Each of these experiments focuses on one parameter of possible query profiles, in particular:

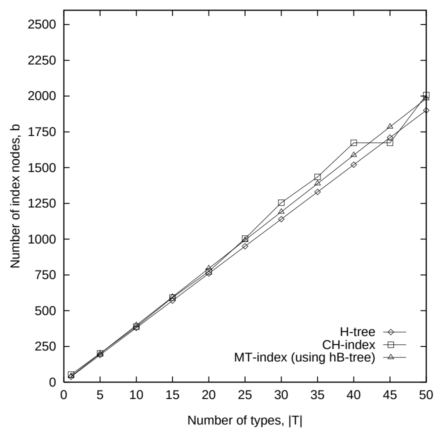
– In Experiment 1, the size of the query interval in the type domain is varied. The experiment is done for one and for two indexed attributes, as well as for exact match queries and range queries, respectively.
– In Experiment 2, the focus is on the variation of the query intervals in all other attributes with a fixed query scope in the type domain. Using a hierarchy with ten types, evaluation results are given for range queries against the full hierarchy as well as for range queries against a sub-hierarchy with five types. Again, the one-attribute case is compared to the two-attribute case.
– In Experiment 3, the impact of the hierarchy linearization on the query performance is determined. For the two-attribute case, the performance of three MT-index scenarios is compared to the H-tree and the CH-index. The first scenario is based on an optimal linearization as described in Sect. 3, the second scenario uses multiple index scans, one index scan per queried type, whereas the third scenario is based on minimum single scan with



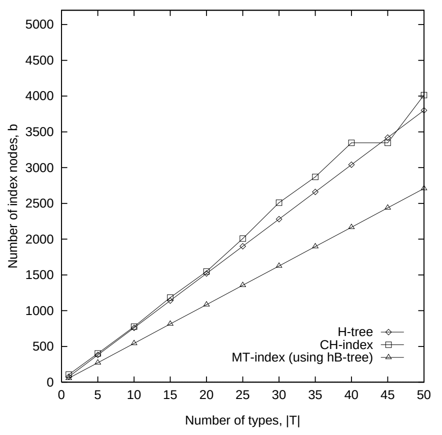
(a) $k$ = 1, $nt$ = 1



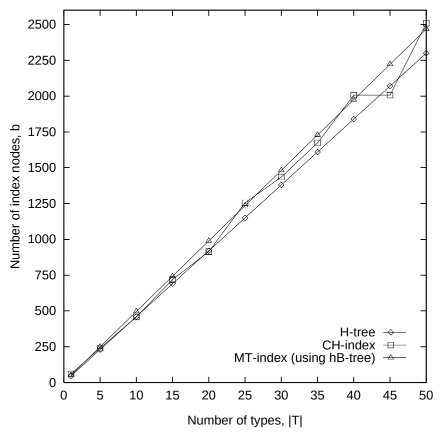
(b) $k$ = 2, $nt$ = 1



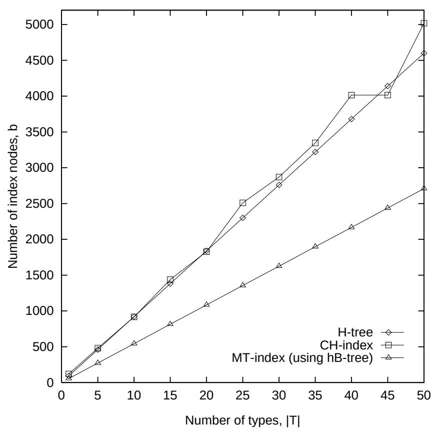
(c) $k$ = 1, $nt = \frac{|T|}{2}$



(d) $k$ = 2, $nt = \frac{|T|}{2}$



(e) $k$ = 1, $nt = |T|$



(f) $k$ = 2, $nt = |T|$

**Fig. 21a–f.** Index size: results of Experiment 1

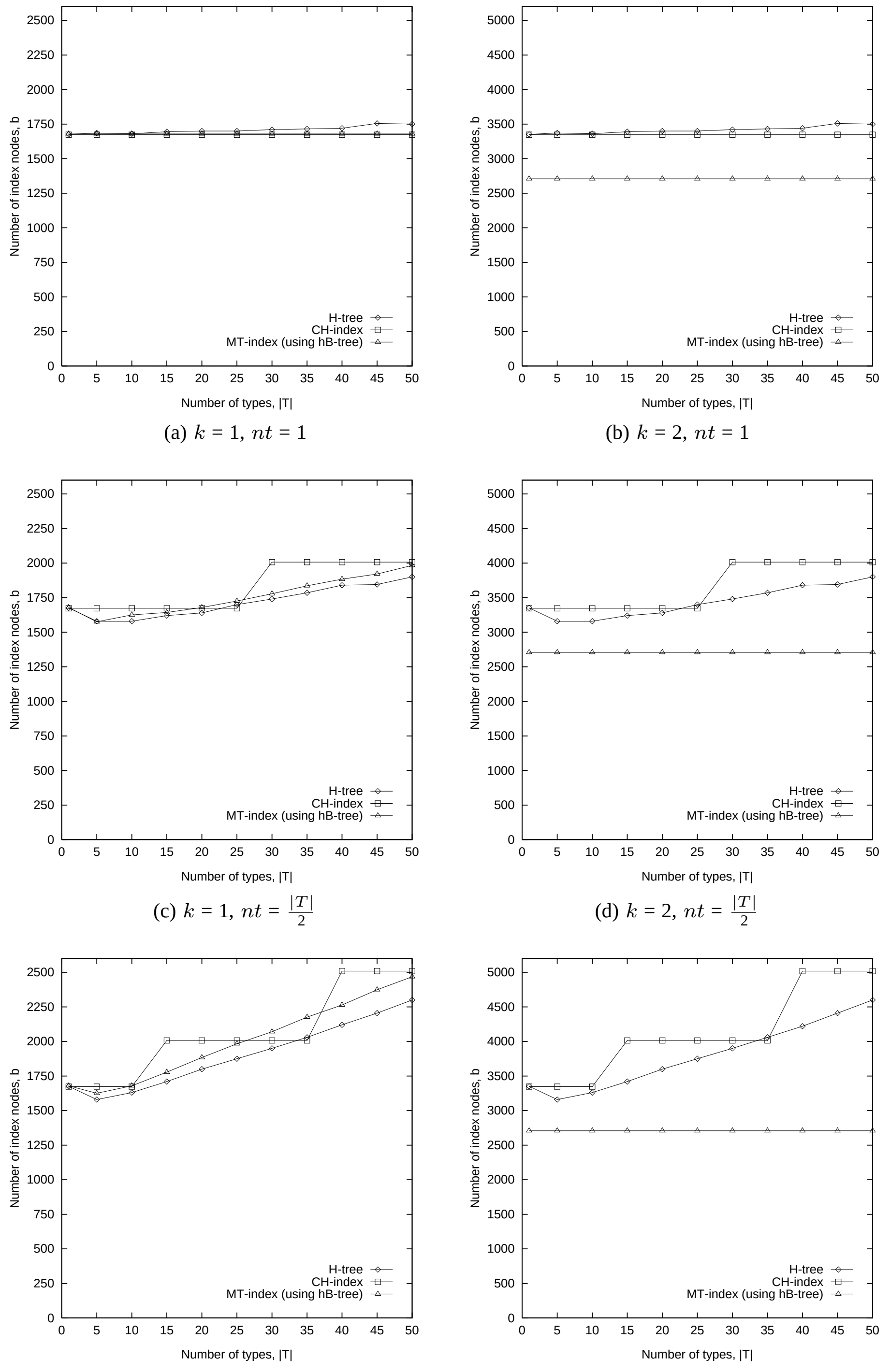
a subsequent elimination of objects belonging to non-matching types. Similar to the previous setting, the size of the query intervals in the indexed attributes is varied.

– The fourth experimental setting addresses a case which is less favorable for the MT-index, namely queries referring only to a subset of the set of indexed attributes. Range queries referring to a varying number of types as well as range queries against a fixed number of types with a increasing size of the query interval are investigated.

For all query performance evaluations, a fully overlapping attribute configuration is assumed ($nt = |T|$). Prior to the evaluation experiments we describe the underlying analytical model for the three data structures in question.

### 6.2.1 Analytical model

Assuming a traversal of $k$ single-key indices for a $k$-attribute query, the estimated number of disk-I/O operations for a query qualifying $d^Q$ attribute values is given by



(a) $k = 1,\ nt = 1$ (b) $k = 2,\ nt = 1$ (c) $k = 1,\ nt = \frac{|T|}{2}$ (d) $k = 2,\ nt = \frac{|T|}{2}$ (e) $k = 1,\ nt = |T|$ (f) $k = 2,\ nt = |T|$

**Fig. 22a–f.** Index size: results of Experiment 2

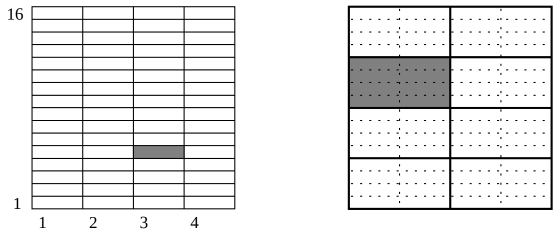
$$k \cdot \left( height^{(CH)} + \left\lceil \frac{d^Q}{e^{(CH)}} \right\rceil \right)$$

for the CH-index and by

$$k \cdot \sum_{t_i \in T^Q} \left( height_i^{(H)} + \left\lceil \frac{\frac{d_i}{d} \cdot d^Q}{e^{(H)}} \right\rceil \right)$$

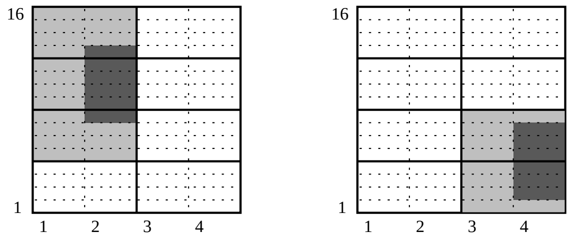
for the H-tree (see [16]). $height^{(CH)}$ ($height_i^{(H)}$) denotes the height of the CH-index (the H-tree for type $t_i$).

An estimation of the number of disk-I/O operations in an $m$-dimensional multiattribute search structure like the hB-tree is less straightforward. We use a probabilistic model outlined in [22]. In particular, the number of I/O operations for a multiattribute range query is calculated with the help of a random variable $V(S, Q, R)$. This variable represents the fetched data volume in a data space which has been normalized to $[0, 1[^m$. The random variable $V$ is expressed in terms of attribute domain structure $S(s_1, \cdots, s_m)$, query profile $Q(q_1, \cdots, q_m)$, and data space partitioning $R(r_1, \cdots, r_m)$.



(a) Attribute domains (b) Data space partitioning

**Fig. 23a,b.** An example for a 2-dimensional data space $S = (\frac{1}{4}, \frac{1}{16})$ and its current partitioning $R = (\frac{1}{2}, \frac{1}{4})$



(a) 3 I/O operations (b) 2 I/O operations

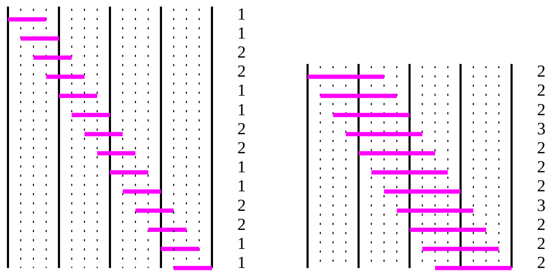
**Fig. 24a,b.** Query template $Q = (\frac{1}{4}, \frac{6}{16})$, query requests (*dark shaded*) $Q_1 : A_1 = 2 \wedge A_2 \in [8, 13]$ and $Q_2 : A_1 = 4 \wedge A_2 \in [2, 7]$ and resulting I/O operations (*light shaded*)

The dimension-specific values $\frac{1}{s_i}, \frac{1}{r_i}$ and $q_i$ represent for dimension $i$ the domain cardinality, the number of subintervals yielded by index split operations and the length of the query interval (see Figs. 23 and 24).

Two model assumptions are made: (a) the domain cardinality equals $2^j$, with $j$ being a positive integer, and (b) all subintervals are of the same length. The latter assumption is based on the usual data distribution models, basically uniformly distributed and uncorrelated raw data, which yield a regular partitioning grid (see example below) in most search structures.

An example illustrated in Fig. 23 contains a particular $S$ (left-hand side) and $R$ (right-hand side). The query template, two possible positions of the query window and the resulting number of I/O operations are shown in Fig. 24 for the same example.

In this model representation, one tuple corresponds to a minimal rectangle in the data space (see shaded area (3, 5) in Fig. 23a). The data space partitioning $R(\frac{1}{2}, \frac{1}{4})$ yields 8 mass storage transfer units (see Fig. 23b, one of the transfer units is shaded). In Fig. 24, the same query template, i.e., $Q(\frac{1}{4}, \frac{6}{16})$ causes either two or three I/O operations, depending on the actual placement of the query window in the data space.



(a) s=1/16, r=1/4, q=3/16 (b) s=1/16, r=1/4, q=6/16

**Fig. 25a,b.** Position of the query window in dimension $j$ and number of I/O operations as a function of $s_j, r_j, q_j$

More precisely, let $W_j(s_j, q_j, r_j)$ denote the random variable for the resulting number of I/O operations in a specific dimension $j$. A closer look at Fig. 25 and a careful analysis of the corresponding I/O pattern yields two values for $W_j$, in particular

$$w'_j = \left\lceil \frac{q_j}{r_j} \right\rceil r_j \quad \text{and} \quad w''_j = \left\lceil \frac{q_j}{r_j} \right\rceil r_j + r_j .$$

Using $b_j$ as an abbreviation for $\left\lceil \frac{q_j}{r_j} \right\rceil r_j$ and analyzing Fig. 25, the probabilistic density function of $W_j$ can be described as

$$\mathrm{P}(W_j = b_j) = \frac{(b_j - q_j + s_j)(1 - b_j + r_j)}{r_j(1 - q_j + s_j)}, \tag{3}$$

$$\mathrm{P}(W_j = b_j + r_j) = \frac{(1 - b_j)(q_j - b_j + r_j - s_j)}{r_j(1 - q_j + s_j)} . \tag{4}$$

Recalling the two-point distribution of $W_j$, probability $\mathrm{P}(W_j = b_j + r_j)$ equals $1 - \mathrm{P}(W_j = b_j)$. Equations 3 and 4 yield an expectation value for $W_j$

$$\mathrm{E}(W_j) = b_j + \frac{(1 - b_j)(q_j + r_j - s_j - b_j)}{1 - q_j + s_j}, \tag{5}$$

which corresponds to the estimated interval length in dimension $j$ as a function of $(s_j, r_j, q_j)$.

Under the assumption of uniformly distributed and uncorrelated raw data and stating a similar assumption for the random positions of the query window, we derive $V(S, R, Q)$ as the product over all $W_j$, that is

$$V = \prod_{1 \le j \le m} W_j , \tag{6}$$

and the corresponding expectation value as

$$\mathrm{E}(V) = \prod_{1 \le j \le m} \left( b_j + \frac{(1 - b_j)(q_j + r_j - s_j - b_j)}{1 - q_j + s_j} \right) . \tag{7}$$

In what follows, simple approximations for Eqs. 5 and 7 are used:

$$\widetilde{\mathrm{E}}(W_j) = \begin{cases} q_j + r_j - s_j & \text{if } q_j \le 1 - r_j \\ 1 & \text{if } q_j > 1 - r_j \end{cases}$$

and

$$\widetilde{\mathrm{E}}(V) = \prod_{1 \le l \le n} \widetilde{\mathrm{E}}(W_j) .$$

In our application context, $m = k + 1$ holds for $k$ indexed attributes, i.e., one additional dimension representing the type domain. The model parameters for the type domain are calculated as follows: The domain structure and the query range for the type dimension are given by

$$s_1 = \frac{1}{|T|} \quad \text{and} \quad q_1 = \frac{|T^Q|}{|T|} ,$$

the partitioning of the type domain is given by

$$r_1 = \begin{cases} \frac{1}{\sqrt[k+1]{b^L}} & \text{if } |T|^{k+1} \leq b^L \\ \frac{1}{|T|} & \text{otherwise} \end{cases} \quad (8)$$

In other words, the standard splitting strategy is also applied to the type domain[3]. Considering a maximum split potential of $|T|$ subintervals in the type dimension and the overall split potential given by the number of data nodes $b^L$ we obtain the two cases described in Eq. 8.

The model parameters for all other dimensions are given by $s_j = \frac{1}{d}$, $q_j = \frac{d^Q}{d}$, and $r_j = \frac{1}{\sqrt[k]{r_1 \cdot b^L}}$. Recalling that $\widetilde{\mathrm{E}}(V)$ is an approximation for the expected volume in $[0, 1[^m$, we obtain $\left\lceil b^L \cdot \widetilde{\mathrm{E}}(V) \right\rceil$ I/O operations for data nodes. The number of index node I/O operations ranges from $height^{(hB)}$ nodes in the case of a point query to $b^I$ nodes for a range query, assuming the very unlikely case, that all internal nodes have to be read during query execution. For the worst case, we therefore obtain $b^I + \left\lceil b^L \cdot \widetilde{\mathrm{E}}(V) \right\rceil$ disk-I/O operations for range query execution.

### 6.2.2 Experiment 1: varying number of queried types

In this setting, a range query with a fixed query range as well as an exact match query are considered. In both cases, the number of qualified types is varied and a fully overlapping attribute configuration is used.

In the case of the range query, 20% of the attribute domain is qualified by the query request, in terms of the analytical model, this means $\frac{d^Q}{d} = 0.2$. For $n = 500,000$, $d = 10,000$, and $|T| = 10$ the number of I/O operations is calculated for different sets of qualified types $T^Q$ with $1 \leq |T^Q| \leq 10$. Results are given in Fig. 26a and b.

Also, in the case of the exact match query evaluation the number of qualified types is varied. For $n = 500,000$, $d = 10,000$, $d^Q = 1$, and $|T| = 10$ the number of I/O operations is determined. As in the previous case, the cardinality of the sets of qualified types $T^Q$ is increased from 1 to 10. Results are visualized in Fig. 26c and d.

The range query setup produces significantly different results for the CH-index on the one hand and for the H-tree and the MT-index on the other hand. The CH-index is characterized by a constant number of I/O operations. The reason is that OIDs of *all types* are stored in one data structure without sufficient support for type-specific access. As a result, all leaf nodes containing qualified attribute values have to be read. Subsequently, OIDs of non-qualified types have to be discarded. In this context, the H-tree has a clear advantage, because for each type a distinct tree is maintained and only trees of qualified types have to be scanned. An interesting point is that the MT-index without type-separated trees performs similar to the H-tree, losing only about 3% I/O performance.

A first non-obvious conclusion can be drawn: the type dimension approach in the MT-index framework is a competitive alternative to the maintenance of type-separated trees, e.g., H-trees. For multiattribute configurations, the performance advantage of the MT-index is most appealing for small query ranges like in this experiment (about 70% advantage over the H-tree for $\frac{d^Q}{d} = 0.2$). It can be seen in Fig. 27 that this advantage is slightly smaller in the case of larger query intervals.

With respect to exact match queries (not considered in [16, 20]), the results are completely different. In the case of the CH-index, a single tree traversal is always sufficient to answer an exact match query, whereas in the H-tree again $|T^Q|$, trees have to be traversed. For large $|T^Q|$ the MT-index performs significantly worse than the CH-index, however, in comparison to the H-tree, an up to 40% performance gain for $k = 1$ (up to 70% for $k = 2$) can be observed.

### 6.2.3 Experiment 2: varying query range

In this context, the focus is solely on range queries with varying ranges. The number of queried types is 5 and 10, respectively, in both cases out of ten indexed types. In particular, for $n = 500,000$, $d = 10,000$, and $|T| = 10$, the number of I/O operations is calculated. The query range is varied between 10% and 100% of the attribute domain, i.e., $0.1 \leq \frac{d^Q}{d} \leq 1$. The corresponding results are depicted in Fig. 27a and b for $|T^Q| = 10$ and Fig. 27b and c for $|T^Q| = 5$.

In the one-dimensional case, the results for all three approaches are almost the same for $|T^Q| = |T| = 10$. Compared to the single-key approaches, the MT-index causes a 3% (1%) increase in the number of I/O operations compared to the H-tree (CH-index). In the case of $k = 2$, the MT-index has a performance advantage of up to 80% for small query ranges. Very large query intervals, e.g., $\frac{d^Q}{d} = 0.5$ still yield almost 50%.

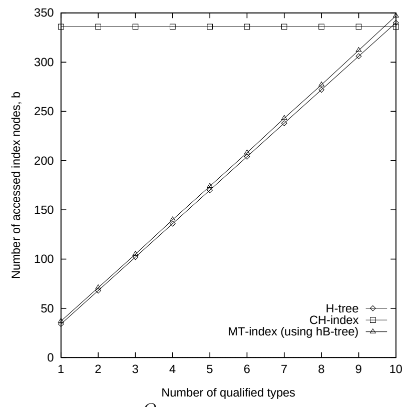
For $|T^Q| = 5$, the number of I/O operations yielded by the CH-index is the same as for $|T^Q| = 10$, whereas the H-tree and the MT-index take advantage of the enhanced index selectivity in this case.

### 6.2.4 Experiment 3: linearization and query performance

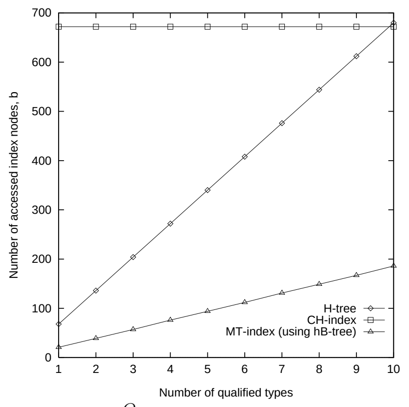
Although a complete description of the influence of the type domain linearization on the query performance is a complex issue, at least one experiment should provide some insights. In general, there are two (not mutually exclusive) scenarios in which the type scope of a query request cannot be mapped to an interval of the type domain containing exactly the types of the query scope.

– There is no optimal linearization in the definition of Sect. 3 for the type hierarchy. In this case, there are subhierarchies which do not correspond to minimal intervals of the type domain.
– Even if an optimal linearization exists the type scope of a query may be an arbitrary subset of the type hierarchy rather than a subhierarchy. In this case, a corresponding minimal interval on the type domain may still exist, how-

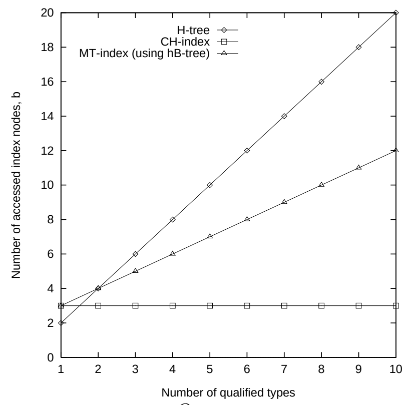
[3] We do not include the pre-splitting scheme mentioned in Sect. 5.2 in this analytical model.



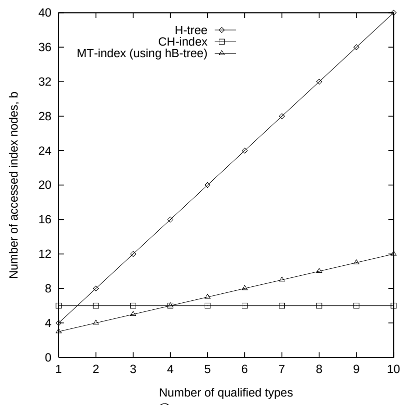
(a) $k = 1$, $\frac{d^Q}{d} = 0.2$ (20% range query)



(b) $k = 2$, $\frac{d^Q}{d} = 0.2$ (20% range query)



(c) $k = 1$, $d^Q = 1$ (point query)



(d) $k = 2$, $d^Q = 1$ (point query)

**Fig. 26a–d.** Query performance: results of Experiment 1

ever it cannot be guaranteed by the optimal linearization as defined above.[4]

If any of these two cases applies, there are two possible solutions for the query optimizer.

– The query request can be processed by a single scan of the smallest interval of the type domain containing all types of the query scope (and some types not qualified by the query). In this case, some of the fetched data has to be discarded due to their non-matching type.
– Alternatively, the query request can be processed by multiple scans referring to the minimal set of intervals such that exactly all qualified types are included. In this case, no data has to be discarded but there is a certain amount of I/O overhead due to multiple tree traversals.

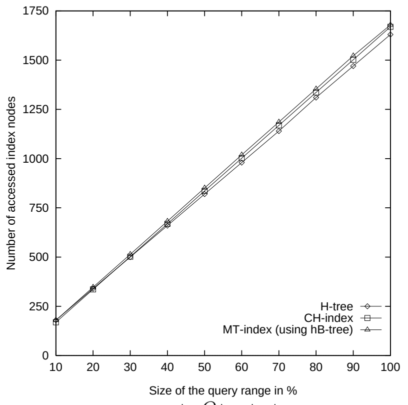
A few example evaluations should give an impression of the performance figures yielded by the different processing strategies. Using the example linearization from the previous sections and two query type sets which do not correspond to subhierarchies, the single scan strategy is compared to the multiscan strategy. To put the results in perspective, also the results for the CH-tree, the H-tree, and the MT-index with optimal linearization are shown. For both experiments, we use $n = 350,000$, $d = 50,000$, two indexed attributes ($k = 2$), and a fully overlapping attribute configuration ($nt = |T| = 7$).

In the first test setting, a query request qualifies the types Student, AssociateProfessor, and AssistantProfessor ($|T^Q| = 3$). Using the single scan strategy, the minimal interval including these three types contains five types. FacultyMember and TeachingAssistant are included in the interval but not qualified by the query. With respect to the multiscan strategy, the example yields a minimal set of two intervals and therefore two index scans, one retrieving the data scored for AssociateProfessor and AssistantProfessor and the other one referring to Student. The results are shown in Fig. 28a.

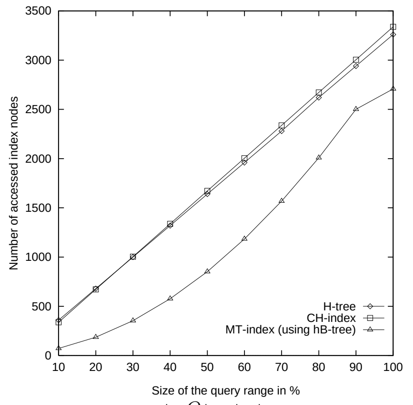
In the second example, the types FullProfessor, AssociateProfessor, FacultyMember, and Student are queried ($|T^Q| = 4$). The smallest interval containing all these types is [FullProfessor, Student]. Consequently, the single-scan strategy affects six types. The multiscan strategy results in three index scans, one for FullProfessor and AssociateProfessor, one for FacultyMember and one for Student. The results are shown in Fig. 28b.

In the first evaluation run the multiscan strategy outperforms the single-scan strategy, irrespective of the size of the query range. In this case, the overhead incurred by multiple scans is more than outweighed by the I/O cost for the two types not qualified by the query request. For practically relevant query range sizes, both strategies perform better than the H-tree. Even the more unfavorable single-scan strategy

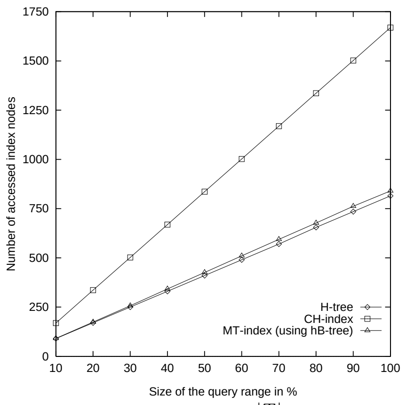
[4] Obviously, a priori knowledge about the query profile may be used in these cases, where the result of the linearization algorithm contains any degrees of freedom. For example, if in the linearization result an arbitrary permutation of a type subset is allowed, the final permutation for the type domain can be chosen according to the query profile.



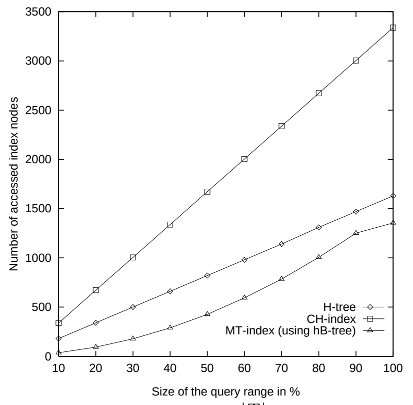
(a) $k$ = 1, $|T^Q|$ = $|T|$ = 10



(b) $k$ = 2, $|T^Q|$ = $|T|$ = 10

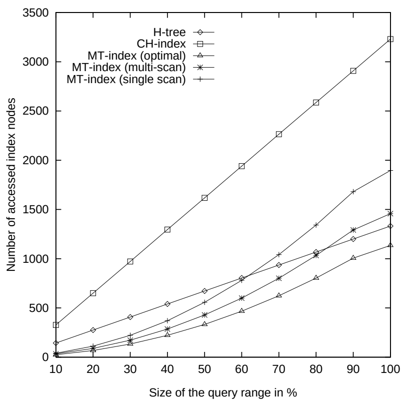


(c) $k$ = 1, $|T^Q|$ = $\frac{|T|}{2}$ = 5

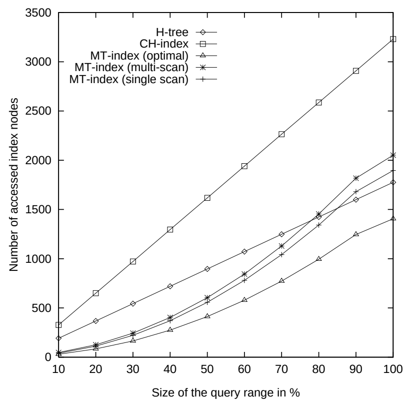


(d) $k$ = 2, $|T^Q|$ = $\frac{|T|}{2}$ = 5

**Fig. 27a–d.** Query performance: results of Experiment 2



(a)



(b)

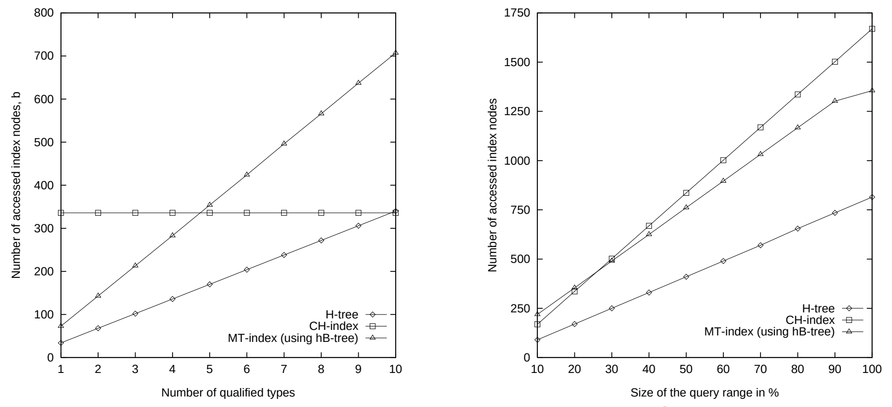
**Fig. 28a,b.** Query performance: results of Experiment 3

yields better I/O performance up to a query range size of about 60% of each queried attribute.

The results of the second experiment show a turn of the tide in the case of an increasing number of necessary scans for the multiscan strategy. Here, the increasing overhead caused by the multiple scans can even dominate the I/O cost of non-qualified types. Again, the H-tree is superior only for very large query range sizes ($>$ 70%). In both cases, the CH-tree is more costly than any other alternative over the full range of the experiments. An MT-index based on an optimal linearization is less costly than any alternative.

### 6.2.5 Experiment 4: partial range queries

There is an inevitable degradation in the I/O performance of an $m$-attribute search data structure if less than $m$ attributes are queried. In what follows, two final evaluations shed some light on this matter. In the context of partial range queries, an MT-index with $k$ = 2 is used to process query requests referring to only one of the two indexed attributes. Similar to previous evaluation experiments in one setting the size of the queried type set is fixed ($|T|$ = 10, $|T^Q|$ = 5) and the size of the query range is varied, whereas in the second setting, the size of the query range is fixed at 20% of the



(a) (b)

**Fig. 29a,b.** Query performance: results of Experiment 4

attribute domain ($\frac{d^Q}{d}$ = 0.2) and the value of $|T^Q|$ is varied between 1 and 10. The results are shown in Fig. 29.

Using the same parameters as in Experiments 1 an 2 above, the H-tree is the dominating structure in the context of partial range queries, because only the H-tree components supporting the queried attribute have to be scanned. Although the same holds for the CH-tree, the MT-index yields advantages for small queried subhierarchies.

## 7 Conclusions

The MT-index is described as an alternative to previous single-key approaches, which are in most cases based on B$^+$-trees. The proposal relies on optimal type hierarchy linearizations in such a way that each subhierarchy corresponds to an interval in the respective type domain of the index. We present an algorithm which computes all existing optimal linearizations for a given type hierarchy. Using this algorithm, the practical design and implementation of an MT-index by means of any existing multiattribute search structure is shown to be straightforward. In this context, a few design alternatives are outlined (i.e., leaf node directories and pre-splitting).

The performance evaluation for the MT-index includes index size as well as exact match and range query performance. The one-attribute setting is compared to the multiattribute setting. In the latter case also, partial range queries are considered. We compared our approach to a key-grouping structure, i.e., the CH-index, and to a type-grouping structure, i.e., the H-tree. Based on the results, we can provide the following rules of thumb:

– The size of an MT-index in case of $k$ = 1, i.e., a 2-dimensional hB-tree, is slightly larger (about 5–8%, depending on the attribute configuration) than the size of the single-key tree approaches. For $k > 1$, a sophisticated $k+1$-dimensional multiattribute search structure like the hB-tree clearly outperforms any set of $k$ single-key trees.
– Exact match queries always favor key-grouping structures. Consequently, the CH-index is the clear leader in this case. With respect to exact match queries, the new proposal performs better than the H-tree. The disadvantage of the MT-index compared to the CH-index decreases with increasing number of indexed attributes.
– Although range queries usually favor type-grouping structures, the H-tree, despite a 3% gain, is unable to clearly outperform the MT-index even for $k$ = 1. The results illustrate the clear advantage of the new proposal if more than one attribute (i.e., $k > 1$) has to be supported by the hierarchy index.
– In the case of partial range queries the H-tree is superior to both the CH-index and the MT-index.

Summing up the performance evaluation, the main result is that an MT-index based on a stable multiattribute search structure is a practically viable alternative to specialized single-key tree approaches even for the unfavorable case of $k$ = 1. This result is far less obvious than the fact that an MT-index is the best choice for $k > 1$. From a practical point of view, this result allows a recommendation of the MT-index if range queries are the dominating pattern in the query profile.

Abstracting from the technical details of index implementations, the MT-index can be seen as a mediator between key-grouping and type-grouping approaches. Informally, a symmetrical structure like the hB-tree does not enforce a particular preference for either key-grouping or type-grouping. However, giving such a preference is possible (if necessary in a particular application context, see the discussion of splitting alternatives in Sect. 5).

Work in progress deals with efficient heuristics for type domain splitting. The idea is to control the degree of type-grouping with respect to a particular query profile.

*Acknowledgements.* We would like to thank the reviewers for their comments and suggestions which led to a considerable improvement of the presentation.

## References

1. Bentley J (1975) Multidimensional binary search trees used for associative searching. Commun ACM 18(9):509–517
2. Bertino E (1991) An indexing technique for object-oriented databases. In: Proceedings Seventh International Conference on Data Engineering, Kobe, Japan. IEEE Computer Society Press, Piscataway, N.J., pp 160–170
3. Bertino E, Ooi BC, Sacks-Davis R, Tan KL, Zobel J, Shidlovsky B, Catania B (1997) Indexing Techniques for Advanced Database Systems. Advances in Database Systems. Kluwer, Dordrecht

4. Carey M, Schneider D (eds) (1995) Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, volume 24 of SIGMOD Record, San Jose, CA. ACM Press, New York
5. Chan CY, Goh CH, Ooi BC (1997) Indexing OODB instances based on access proximity. In: ICDE'97 [10], pp 14–21
6. Evangelidis G, Lomet D, Salzberg B (1995) The hB$^{\Pi}$-tree: A modified hB-tree supporting concurrency, recovery and node consolidation. In: Dayal U, Gray PMD, Nishio S (eds) Proceedings of 21th International Conference on Very Large Data Bases, Zürich, Switzerland. Morgan Kaufmann, San Mateo, Calif., pp 551–561
7. Freeston M (1995) A general solution of the n-dimensional B-tree problem. In: Carey, Schneider [4], pp 80–91
8. Gudes E (1996) A uniform indexing scheme for object-oriented databases. In ICDE'96 [9], pp 238–246
9. Proceedings Twelfth International Conference on Data Engineering, New Orleans, Louisiana, Mar. 1996. IEEE Computer Society Press, Piscataway, N.J.
10. Proceedings Thirteenth International Conference on Data Engineering, Birmingham, UK, Apr. 1997. IEEE Computer Society Press, Piscataway, N.J.
11. Kanellakis PC, Ramaswamy S, Vengroff DE, Vitter JS (19 ) Indexing for data models with constraints and classes. In: Garcia-Molina H, Jagadish H (eds) Proceedings of the Twelfth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, Washington, DC. ACM Press, New York, pp 233–243
12. Kilger C, Moerkotte G (1994) Indexing multiple sets. In: VLDB'94 [25], pp 180–191
13. Kim W, Kim KC, Dale A (1989) Indexing techniques for object-oriented databases. In: Kim W, Lochovsky FH (eds) Object-Oriented Concepts, Databases, and Applications. Addison-Wesley, Reading, Mass., pp 371–394
14. Lomet DB, Salzberg B (1990) The hB-tree: A multiattribute indexing method with good guaranteed performance. ACM Trans Database Syst 15(4):625–658
15. Low CC, Lu H, Ooi BC, Han J (1991) Efficient access methods in deductive and object-oriented databases. In: Delobel C, Kifer M, Masunaga Y (eds) Deductive and Object-Oriented Databases. Proceedings of the Second International Conference, DOOD, volume 566 of Lecture Notes in Computer Science, Munich, Germany. Springer, Berlin Heidelberg New York, pp 68–84
16. Low CC, Ooi BC, Lu H (1992) H–trees: A dynamic associative search index for OODB. In: Stonebraker M (ed) Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data, volume 21 of SIGMOD Record, San Diego, California. ACM Press, New York, pp 134–143
17. Mueck TA, Polaschek ML (1996) Indexing type hierarchies with multikey structures. In: Connor RCH, Nettles S (eds) Proceedings of the 7th International Workshop on Persistent Object Systems (POS 7). Morgan Kaufmann, San Mateo, Calif., pp 184–193
18. Mueck TA, Polaschek ML (1997) Index Data Structures in Object-Oriented Databases. Advances in Database Systems. Kluwer, Dordrecht
19. Mueck TA, Polaschek ML (1997) The multikey type index for persistent object sets. In: ICDE'97 [10], pp 22–31
20. Ooi BC, Han J, Lu H, Tan KL (1996) Index nesting – an efficient approach to indexing in object-oriented databases. VLDB J 5(3):215–228
21. Ramaswamy S, Kanellakis PC (1996) OODB indexing by class-division. In: Carey, Schneider [4], pp 139–150
22. Schauer M (1993) Adaptive Clusterbildung in Mehrattributsuchstrukturen. Dissertation, Universität Wien, in german.
23. Shidlovsky B, Bertino E (1996) A graph-theoretic approach to indexing in object-oriented databases. In: ICDE'96 [9], pp 230–237
24. Sreenath B, Seshadri S (1994) The hcC-tree: An efficient index structure for object-oriented databases. In: VLDB'94 [25], pp 203–213
25. Proceedings Twentieth International Conference on Very Large Databases, Santiago, Chile, Sept. 1994. Morgan Kaufmann, San Mateo, Calif.

## Appendix A
## Linearization algorithm

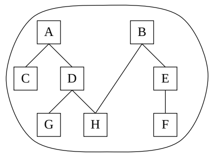
```
    $D \leftarrow \emptyset$
    $\boldsymbol{T'} \leftarrow$ order$(T, \leq)$

 1. begin order$(S, \leq)$
 2.   if $|S| < 3$ then return $S$ end
 3.   $M \leftarrow$ max$(S \setminus D, \leq)$
 4.   $\boldsymbol{L} \leftarrow \bigcup_{m \in M} \{(S_{\leq m})\}$
 5.   $D \leftarrow D \cup M$
 6.   $\boldsymbol{S'} \leftarrow S \setminus \bigcup_{m \in M} S_{\leq m}$
 7.   while $\exists \mathsf{A} \in \boldsymbol{L}$ do
 8.     $\boldsymbol{L} \leftarrow \boldsymbol{L} \setminus \{\mathsf{A}\}$
 9.     if $\exists \mathsf{B} \in \boldsymbol{L} : \bigcup A_i \cap \bigcup B_j \neq \emptyset$ then
10.       if $\mathsf{A} \circ \mathsf{B}$ is defined then
11.         $\boldsymbol{L} \leftarrow \boldsymbol{L} \setminus \{\mathsf{B}\} \cup \{\mathsf{A} \circ \mathsf{B}\}$
12.       else fail
13.       end
14.     else
15.       while $\exists x \in$ max$(\bigcup A_i \setminus D, \leq)$ :
            $|\{A_i | A_i \cap S_{\leq x} \neq \emptyset\}| > 1$ do
16.         if $\mathsf{A} * S_{\leq x}$ is defined then
17.           $\mathsf{A} \leftarrow \mathsf{A} * S_{\leq x}$
18.           $D \leftarrow D \cup \{x\}$
19.         else fail
20.         end
21.       end
22.       $\boldsymbol{S'} \leftarrow \boldsymbol{S'} \cup \{$(order$(A_1, \leq)$, order$(A_2, \leq), \cdots$
            $\cdots$, order$(A_{|\mathsf{A}|}, \leq))\}$
23.     end
24.   end
25.   return $\boldsymbol{S'}$
26. end order
```

$D$ and $\boldsymbol{T'}$ represent the set of marked types and final linearization result, respectively.

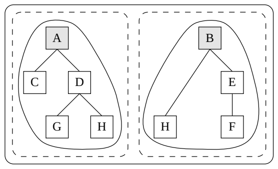
## Appendix B
## Execution trace

The tables below provide snapshot information for selected variables and expressions. In each table, the first column refers to the line numbers of the algorithm. 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 brackets are omitted, list elements are separated by white spaces, e.g., {(ABCD EF) (G)} instead of {({ABCD} {EF})({G})}. The initial call starts with the hierarchy



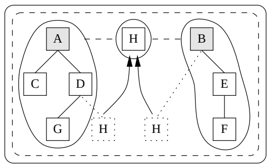
| **Call**: order(ABCDEFGH) | | | | | |
|---|---|---|---|---|---|
| | $D$ | $\boldsymbol{L}$ | $\boldsymbol{S'}$ | $M$ | |
| 6 | AB | {(ACDGH)(BEFH)} | $\emptyset$ | AB | |

After the initialization steps, $\boldsymbol{L}$ 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.



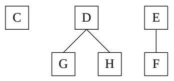
| | $D$ | $\boldsymbol{L}$ | $\boldsymbol{S'}$ | $\mathcal{A}$ | $\mathcal{B}$ |
|---|---|---|---|---|---|
| 7 | AB | {(ACDGH)(BEFH)} | $\emptyset$ | (BEFH) | – |
| 8 | AB | {(ACDGH)} | $\emptyset$ | (BEFH) | – |
| 9 | AB | {(ACDGH)} | $\emptyset$ | (BEFH) | (ACDGH) |
| 11 | AB | {(BEF H ACDG)} | $\emptyset$ | (BEFH) | (ACDGH) |

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 $\boldsymbol{L}$.



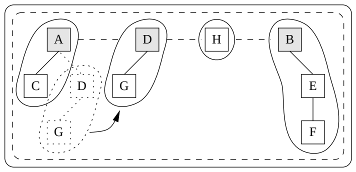
| | $D$ | $\boldsymbol{L}$ | $\boldsymbol{S'}$ | $\mathcal{A}$ | |
|---|---|---|---|---|---|
| 7 | AB | {(BEF H ACDG)} | $\emptyset$ | (BEF H ACDG) | |
| 8 | AB | $\emptyset$ | $\emptyset$ | (BEF H ACDG) | |

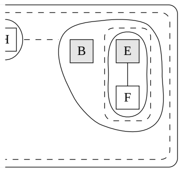
Refinement candidates are the maximal elements of $\bigcup A_i \setminus D$ (see right-hand side figure), in this case only D and E for ({BEF}{H}{ACDG}), since C is a leaf. The subhierarchy of D is {DGH}. It has a non-empty intersection with both, {H} and {ACDG}. So it is a possible operand for $*$,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



| | $D$ | $\boldsymbol{L}$ | $\boldsymbol{S'}$ | $\mathcal{A}$ | max (...) | $x$ | $S_{\leq x}$ |
|---|---|---|---|---|---|---|---|
| 14 | AB | $\emptyset$ | $\emptyset$ | (BEF H ACDG) | DE | – | – |
| 15 | AB | $\emptyset$ | $\emptyset$ | (BEF H ACDG) | DE | D | DGH |
| 17 | AB | $\emptyset$ | $\emptyset$ | (BEF H DG AC) | DE | D | DGH |
| 18 | ABD | $\emptyset$ | $\emptyset$ | (BEF H DG AC) | E | D | DGH |

After refinement, $\mathcal{A}$ 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 non-trivial 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 descents, the resulting situation is given in the following table:





| | $D$ | $\boldsymbol{L}$ | $\boldsymbol{S'}$ | $\mathcal{A}$ | |
|---|---|---|---|---|---|
| 22 | ABDE | $\emptyset$ | {({B(EF)} H DG AC)} | {({B(EF)} H DG AC)} | |
| **Result**: {({B(EF)} H DG AC)} | | | | | |