Indexing Multiple Sets

Christoph Kilger *Guido Moerkotte*

Universität Karlsruhe, Fakultät für Informatik, D-76128 Karlsruhe, Germany
email: [kilger|moer]@ira.uka.de

Abstract

Index structures for multiple sets can be classified into those that group entries according to their key values and those that group entries according to their set membership. The former are particularly suited for exact match queries on all indexed sets, the latter especially support range queries on a small number of all indexed sets. The goal is to thoroughly evaluate the performance of both grouping strategies.
There exist two reasons for adding a new index structure to the evaluation: (1) The performance potentials of set grouping index structures are not yet fully exploited. (2) Up to now, the database administrator has to choose between key grouping and set grouping index structures, supporting either exact match or range queries. What is needed is a more flexible index structure that can be tuned to a given query mix containing both, exact match and range queries. These two reasons led us to the development of the CG-tree. The focus of the paper is on introducing the CG-tree and on a thorough performance analysis of the CH-index [7], the H-tree [8, 9] and the CG-tree.

1 Introduction

Index structures facilitate the fast direct access to large sets of records by some key attribute. "Fast" means that the number of pages to be read to retrieve the qualifying records is small compared to the total number of pages the records occupy. Besides the direct access to records (*exact match queries*) an index structure should also support queries for records whose key is within a given range (*range queries*). The best general purpose index structure for exact match *and* range queries known so far is the B^+-tree [1].

Typically, B^+-trees index the elements of a single set. However, in many applications, there exists the necessity of indexing multiple sets by a common key. For example, in the relational model, the computation of an equi join of two relations requires the access to those records in both relations matching the join condition, i.e., having the same join attribute values. As shown by Härder [4] and Valduriez [12] a join operation is effectively supported using one combined index structure for both relations. Härder has generalized this idea to an arbitrary number of relations (*Generalized Access Path Structure*).

Index structures for multiple sets are also useful in object bases where objects are members of classes. The classes are related by the subclass relationship, which may form a hierarchy or a directed acyclic graph. Often, a subset relationship is tied to the subclass relationship. Queries in object bases may be evaluated on a single class, i.e., the set of the direct members of the class, or on a class including all its subclasses. Moreover, many query languages (e.g., O_2SQL [3] or GOMql [5]) allow queries to be formulated on arbitrary sets. As shown in [7, 8, 9] the access to the members of several classes by a common key attribute may be supported by a multiple set index structure for these classes. In the context of object bases two index structures based on the B^+-tree for supporting multiple set indexing have been proposed: the *class hierarchy index (CH-index)* [7] and the *H-tree* [8, 9].

If multiple sets are indexed on a common key, there exist two choices for grouping: grouping by key and grouping by set membership. A *key grouping index* (e.g. the CH-index) stores all entries (of all indexed sets) with the same key in one leaf page record. Within each leaf page record, a set directory is used to keep track of set membership. For exact match and range queries, the retrieval costs are independent of the number of sets queried.[1] If not all indexed sets are queried, an overhead exists since elements in sets not queried

Proceedings of the 20th VLDB Conference
Santiago, Chile, 1994

[1] This only holds if no leaf page record exceeds the page capacity.

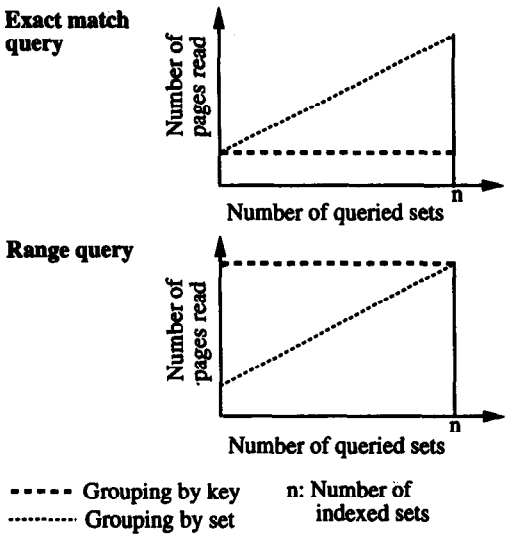


Figure 1: Grouping by key vs. grouping by set

are clustered together with those queried. Hence, for range queries, more pages than necessary are read.

A *set grouping index* (e.g. the H-tree) first groups all members of all sets by their set membership. Members of a single set are then grouped at the leaf page level according to their key values. Hence, leaf pages are occupied by members of one indexed set only. This implies that the retrieval costs are directly proportional to the number of sets queried.

Fig. 1 summarizes the above performance considerations for both multiple set indexing techniques. The experimental validation of this figure is given in the next section.

Within the rest of the paper, we consider multiple set index structures based on the B^+-tree. (For a discussion of possible alternatives see Section 2). One goal is to assess the performance of key and set grouping B^+-based indexes under various conditions.

Since the H-tree — the only set grouping index structure — does not fully exploit the potential of the set grouping approach, we introduce a new set grouping index. Besides performance arguments, there exists another reason to introduce a new multiple set indexing technique. As can be seen from Figure 1, the database administrator who has to decide which indexing technique to apply, finds herself in a dilemma. As soon as the application profile on hand consists of a mix of exact match and range queries which do not comprise the full number of indexed sets, there is none but a bad choice favoring only half of the application. What's missing is the possibility to tune a given index structure according to the application profile on hand. These considerations together with the potential for faster set grouping indexes led us to the development of a new set grouping index called CG-tree.

The rest of the paper is organized as follows. Section 2 briefly reviews the CH-index and the H-tree, and gives a first experimental validation of Figure 1. Section 3 introduces the CG-tree. Section 4 evaluates the retrieval costs of all three multiple set indexes. Section 5 introduces the flexibility necessary to allow the database administrator to appropriately tune the CG-tree, and presents benchmark results for the tunable version of the CG-tree. Section 6 concludes the paper.

2 Multiple Set Indexing Techniques

Before reviewing the CH-index and the H-tree, we should reflect whether well-known single-set or multi-dimensional index structures could be employed. Using ordinary single-set index structures for multiple set indexing, e.g., the B^+-tree, has been shown to be inefficient by Kim et al. [7].

Considering set membership of the entries as one key dimension, multi-dimensional index structures like the Grid-File [10] or the Buddy-Tree [11] can also be used for multiple set indexing. Multi-dimensional index structures were designed for large, ordered, open key domains. However the key domain representing the indexed sets is small, unordered, and its values are known in advance. This additional knowledge should be exploited by the index structure. Furthermore, there often exists a strong correlation between key values and set membership. As pointed out in [11] the performance of most multi-dimensional index structures strongly degrades if there is a correlation between the key dimensions.

These considerations led us to investigate only special multiple set index structures.

2.1 Grouping By Key

Key grouping index structures, e.g., the Generalized Access Path Structure [4] and the CH-index [7], store all elements of the indexed sets having the same key in one leaf page record. The elements stored in the same record are grouped by their set membership and a set directory is maintained containing for each set the offset of the corresponding group in the leaf page record. In the Generalized Access Path Structure the set-directory has one entry for each indexed set. In the CH-index the directory of the leaf page record with key K has one entry for each set that contains some element with key K. The layout of the leaf page records of the CH-index is shown in Fig. 2.

If the size of a leaf page record exceeds the page size, additional overflow pages are allocated. In this case, the offsets of the set directory refer to byte positions within the record's pages. In the CH-index, a leaf page and its overflow pages form a simple linked list.

The non-leaf pages in a key grouping index have the same layout as in the B^+-tree.

record length	key value	number of sets	overflow page	s_1	offset	...	s_k	offset	elements of set s_1	...	elements of set s_k

Figure 2: Layout of the leaf page records of the CH-index

2.2 Grouping by Set

Index structures that group elements by set have separate leaf pages for each of the indexed sets. Every leaf page stores elements of just one set and, hence, no set directories have to be maintained in the leaf page records.

The only set grouping index is the H-tree [8, 9]. The H-tree was proposed as an indexing structure for class hierarchies and requires the indexed sets to form a hierarchy. The H-tree maintains one B^+-tree for each set. These are nested according to the set hierarchy. Nesting is implemented via *link pointers* referring from a non-leaf node of the parent tree to a (leaf or non-leaf) node of the child tree. Fig. 3 illustrates the nesting of four set-trees. The links also reflect subset relationships on key ranges. For further details the reader is referred to [8, 9].

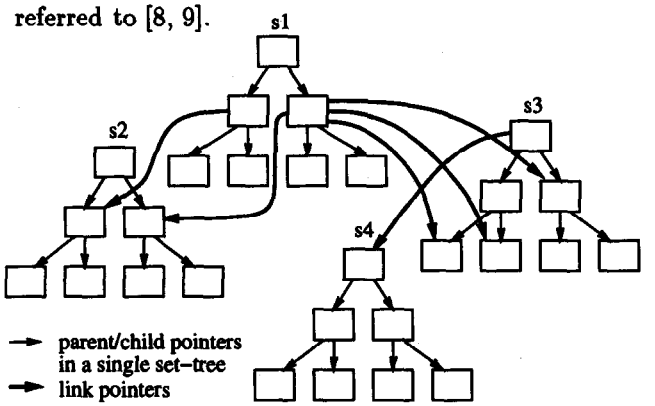


Figure 3: Nesting of set-trees in the H-tree

2.3 Validation of Figure 1

Fig. 4 gives an experimental validation of the theoretical considerations outlined in the introduction. (For details of the database, see Sec. 4.) The similarity of Figures 1 and 4 is striking. If for exact match queries all 8 indexed sets are queried, the performance of the H-tree is 7 times worse than that of the CH-index (Fig. 4, first plot). On the other hand, if only 1 set is queried with a range query, the H-tree performs almost 7 times as good as the CH-index (Fig. 4, second plot). These observations give the motivation of our work:

1. We designed the CG-tree in order to improve the performance of set grouping indexes on exact match queries thereby preserving their good performance on range queries.

2. Flexibility is added to the CG-tree such that intermediate states between the performance of the set grouping and the key grouping index approach (for both, exact match and range queries) can be achieved.

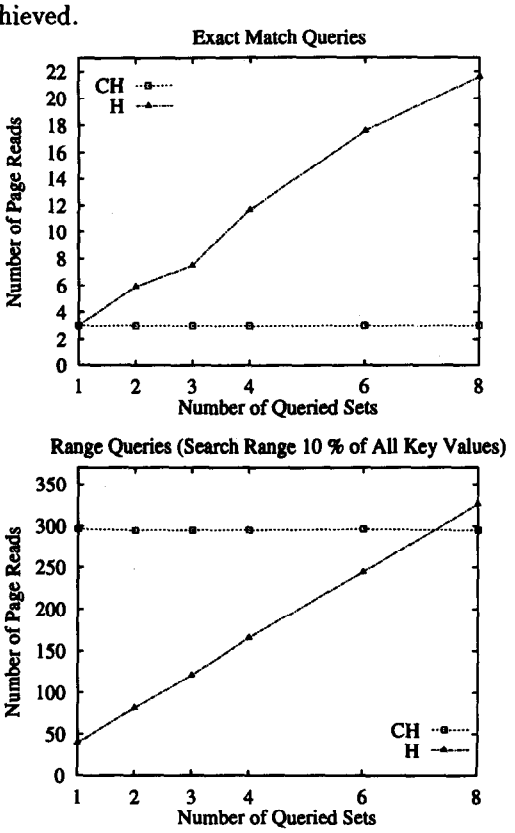


Figure 4: Query performance of the CH-index and the H-tree (8 indexed sets, with 600,000 entries in total, 20,000 key values, no overflow pages)

3 The CG-Tree

3.1 The Data Structure

The CG-tree is a set grouping index. Its organization is described from bottom to top. Its leaf pages (level 1)[2] are organized into n doubly linked lists — one list for each indexed set. Leaf pages exhibit the same layout as in the B^+-tree, and, hence, do not contain a set directory like the CH-index.

The records of the non-leaf pages at level 2 contain a vector (called *set directory*) of leaf page references of length n — one reference for each indexed set. The pages at level 2 are called *directory pages*. The internal structure of a directory page is given in Fig. 5. The i-th component of the set directory R_j (denoted by $R_j.s_i$) references the leaf page containing those elements of

[2]Leaf pages are at level 1, non-leaf pages are at levels 2 and above.

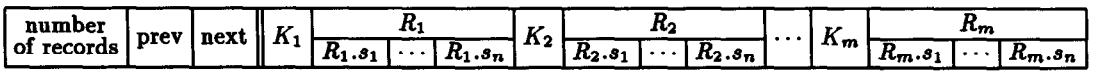


Figure 5: Internal structure of directory pages

set s_i whose keys are in $[K_j, K_{j+1})$. If s_i does not contain any such element, $R_j.s_i$ is null.

The remaining non-leaf pages at levels 3 and above are called *non-directory pages*. They exhibit the same structure as the non-leaf pages in the B$^+$-tree: each page contains a sequence of records of the form $((K_1, R_1), \ldots, (K_m, R_m))$ where R_i references the root of the subtree containing all entries whose keys are in the range $[K_i, K_{i+1})$.

Sharing of Leaf Pages

The cardinalities of the indexed sets and their distribution of key values may be non-uniform. In this case, the average filling degree of some leaf pages is low if each directory reference $\neq NULL$ points to its own non-shared leaf page. For this reason, leaf pages may be shared by several (neighbored) directory page records. Consider the CG-tree depicted in Fig. 6 (a), and assume that all records of L_1 and L_2 fit on one page. To increase the filling degree, all elements of set s_1 with keys in the range $[K_1, K_3)$ are stored on one leaf page L_{12} as shown in Fig. 6 (b). Only leaf pages storing elements of the same set may be shared. This condition is relaxed in Section 5.

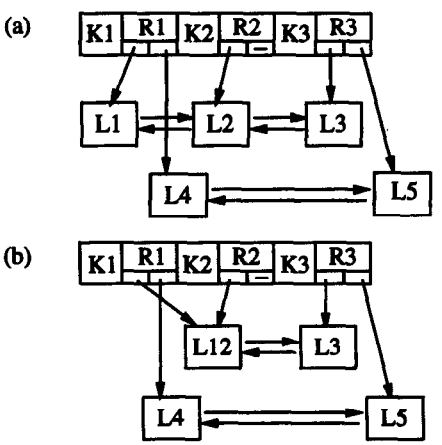


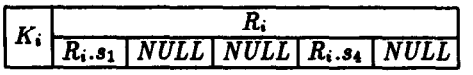
Figure 6: Sharing of leaf pages

Linking of Directory Pages

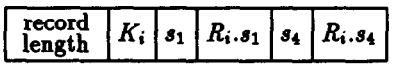
On the occurrence of a leaf page overflow, the leaf page is split and the directory page records referencing this leaf page have to be updated. In order to be able to efficiently determine all directory page records referencing some leaf page the directory pages are doubly linked by the *prev* and *next* fields in the header of a directory page (see Fig. 5).

Null Pointers in Directory Pages

If the indexed sets are disjoint, most of the references in the directory page records will be null. Hence, the fan-out is rather low. In order to increase the fan-out of directory pages, the set directories are internally stored as variable length records, containing only non-null directory components and the corresponding set identifiers (see Fig. 7). The record length (measured in the number of non-null directory components) and the set identifiers are represented with $\lceil \log_2(n) \rceil$-bits, where n is the number of indexed sets.



(a) Conceptual Representation



(b) Physical Representation

Figure 7: Representation of directory entries

3.2 Retrieval

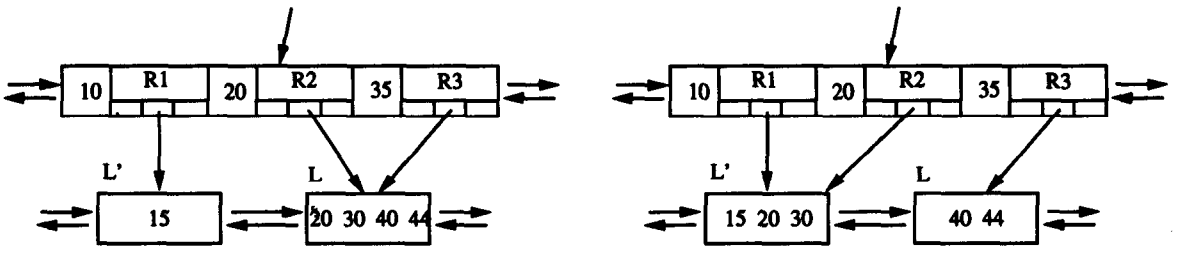
The retrieval algorithm produces k output streams $O_1, \ldots, O_k$, one for each of the queried sets $s_1, \ldots, s_k$. Stream O_i contains all elements of s_i whose keys are in the queried range $[K_{lb}, K_{ub})$. The retrieval is easily described. First, the tree is descended until the directory page for key K_{lb} is reached. Second, for each of the searched sets s_i, the leaf page L_i is determined containing the elements of s_i with the smallest key $\geq K_{lb}$. Third, starting at the leaf pages $L_1, \ldots, L_k$ the linked lists of leaf pages are traversed as long as keys in the queried range are found.

3.3 Updates

The initial CG-tree consists of a single directory page containing $(-\infty, (NULL, \ldots, NULL))$ as the only entry. In the sequel we discuss insertion. The detailed insertion and the deletion algorithm are beyond the scope of this paper. The reader is referred to [6].

Insert

For inserting a new entry (K_{new}, E_{new}), $E_{new} \in s_{new}$, into the tree, it is descended until the directory page D for key K_{new} is reached. Let (K_i, R_i) be the entry of D such that K_i is the largest key $\leq K_{new}$. If $R_i.s_{new} =$



(a) Before balancing L with L'

(b) After balancing L with L'

Figure 8: Balancing of leaf pages

NULL then we can either create a new leaf page or reuse (via sharing) an existing leaf page for set s_{new}.

In order to keep the average filling degree of leaf pages high, we first try to reuse an existing leaf page. The sequence of directory page records is traversed to the right starting at (K_i, R_i) until the first non-null reference for set s_{new} is found. If no leaf page is found, or if the found leaf page is full and cannot be split (because all records have the same key), the search restarts to the left of (K_i, R_i). Only if no existing leaf page can be reused, a new leaf page is created. $R_i.s_{new}$ is set to reference the reused or newly created leaf page, respectively.

Let L be the leaf page referenced by $R_i.s_{new}$. If (K_{new}, E_{new}) fits into L, it is inserted. Otherwise, if all records have the same key K, and K equals K_{new}, the new entry is inserted into one of L's overflow pages (if necessary, a new one is created); else L is split.

Let us review the search for a reusable leaf page to store a member of set s_{new} in some more detail. The search for a leaf page to be reused is stopped once the first non-null directory component for set s_{new} is encountered. Leaf pages further to the right (or left) are not used. This is the case because the following two *properties of the CG-tree* — which are essential for efficient retrieval and update — are always guaranteed:

1. A directory page record with key K_j contains a non-null leaf page reference for the i-th indexed set if and only if the i-th set contains an element whose key is in the range $[K_j, K_{j+1})$.

2. The list of directory pages and the lists of leaf pages are ordered by ascending keys.

These properties imply that if there are two directory page records (K_1, R_1) and (K_2, R_2) referencing the same leaf page L belonging to set s_i then for all directory page records (K, R) with $K_1 < K < K_2$, either $R.s_i = L$ or $R.s_i = NULL$ holds.

Balancing of Leaf Pages

If a new entry cannot be inserted into the appropriate leaf page, balancing with neighbor pages is tried first. Only if this does not work either, splitting is applied.

Let L be a leaf page holding elements of set s. Note that s is determined by L, since each leaf page holds elements of a single set only. For a given L, we denote by $< (K_1, R_1), \ldots, (K_m, R_m) >_L$ the sequence of all directory page records with

1. $K_i < K_{i+1} \quad (1 \leq i < m)$,

2. $R_1.s = R_m.s = L$,
   $\forall\, 1 < i < m : R_i.s = L \vee R_i.s = NULL$, and

3. for all directory page records (K, R) of the tree

$$R.s = L \Rightarrow (K, R) \in \{(K_1, R_1), \ldots, (K_m, R_m)\}$$

The sequence $< (K_1, R_1), \ldots, (K_m, R_m) >_L$ is called the *sequence of directory page records of leaf page* L. Because of the above properties of the CG-tree, this sequence is well defined for each leaf page. It contains exactly those directory page entries relevant to L.

As an example, consider the CG-tree depicted in Fig. 8 (a). The sequence of directory page records of L is $< (20, R_2), (35, R_3) >_L$.

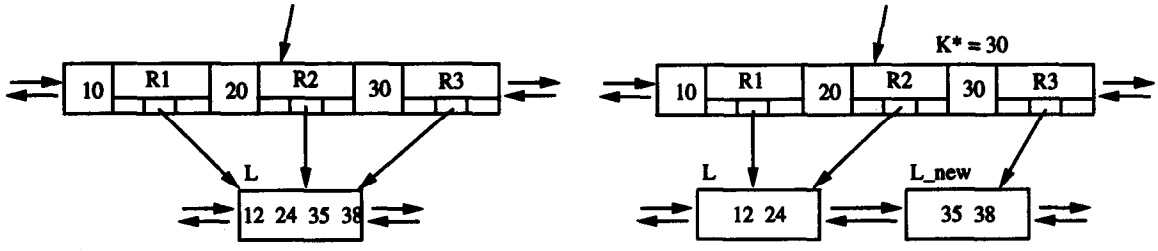
Assume that the new entry (K_{new}, E_{new}) does not fit into L. If the sequence of directory page records of L contains more than one record, we first try to balance L with one of its neighbors instead of creating a new leaf page.

For example consider the insertion of a record with key 36 into page L of the CG-tree shown in Fig. 8 (a). The page capacity is 4 records. L can be balanced with its left neighbor L', since all records of L whose keys are in the range $[20, 35)$ fit into L'. Fig. 8 (b) shows the CG-tree after balancing L with L'. Now, the record with key 36 can be inserted into L.

We consider the balancing of leaf pages only if there are at least two directory page records referencing the leaf page; thus, just one directory component has to be adapted. Opposed to the B$^+$-tree balancing of leaf pages, balancing in the CG-tree does never require the reorganization of the whole index up to the root.

Splitting of Leaf Pages

If L cannot be balanced with one of its neighbors a new leaf page L_{new} is created, a separation key K^*



(a) Before the split

(b) After the split

Figure 9: Splitting of leaf pages (separation key from directory page records)

for the records of L is determined (see below), and all records whose keys are greater or equal K^* are moved from L to L_{new}.

The insertion of the new leaf page requires some directory page records to be adapted in order to keep the tree correct with respect to the above properties. Consider the CG-tree shown in Fig. 9. The separation key K^* is 30. After moving the leaf page records, the directory record $(30, R_3)$ must be updated to point to the new leaf page L_{new}. In this example, the separation key $K^* = 30$ was chosen from the keys present in the directory page records.

If K^* is not chosen from the keys of the directory page records a new directory page record (K^*, R_{new}) is created and inserted. As an example, consider the CG-tree depicted in Fig. 10. We choose $K^* = 25$. The leaf page references for set s_2 are adapted the same way as before. But the insertion of the new directory page record (K^*, R_{new}) also effects the leaf page pointers for sets s_1 and s_3. For example, R_1 references leaf page B (belonging to s_3) before the split because the key range of R_1 is $[10, 30)$ (Fig. 10 (a)). After the split, the key range of R_1 is $[10, 25)$ (Fig. 10 (b)). Since leaf page B does not contain any record with a key within this range $R_1.s_3$ must be set to null. The same holds for $R_{new}.s_1$: $R_{new}.s_1$ is set to null, because leaf page A contains no element within the range $[25, 30)$.

Note, that — due to the sharing of leaf pages — all leaf pages of some set s_i can be split independently from the leaf pages of the other indexed sets.

Choosing a Separation Key

In the B$^+$-tree the separation key for splitting leaf page L is chosen such that nearly half of L's records are moved to the new page. In the CG-tree there is the additional objective to choose the separation key — whenever feasible — from the keys of the directory page records referencing page L, in order to keep the number of directory page records small.

To explain the method for choosing a separation key, we need some additional abbreviations. The size of the record of leaf page L with key K is denoted by $RecSize_L(K)$. If there is no record with key K in L then $RecSize_L(K) = 0$. Let $K'_1, \ldots, K'_l$ be the keys of the records of leaf page L in ascending order. Then, we define

$$AccuRecSize_L(K) \quad := \sum_{\substack{K'_i < K \\ 1 \le i \le l}} RecSize_L(K'_i)$$

$AccuRecSize_L(K)$ denotes the accumulated sizes of all records of L whose keys are less than K. If L is split with key K as separation key, $AccuRecSize_L(K)$ bytes remain on L.

The separation key is determined as follows: For $< (K_1, R_1), \ldots, (K_m, R_m) >_L$ we determine $K_{i^*} \in \{K_1, \ldots, K_m\}$, s.t. the term

$$| AccuRecSize_L(K_{i^*}) - 0.5 * \sum_{1 \le i \le l} RecSize_L(K'_i) |$$

is minimized. I.e., K_{i^*} is—among the keys of the directory page records—the best choice for dividing the records of L into two subsets of approx. the same size.

However, K_{i^*} may be a "bad" separation key, dividing the records of L into two extremely differently sized subsets. In our implementation, we require $AccuRecSize_L(K_{i^*})$ to be within the range of 30 % and 70 % of the page size. In this case, $K^* := K_{i^*}$. Otherwise, we arbitrarily determine K^* such that L is divided into two even subsets.

Splitting of Non-Leaf Pages

The splitting algorithm for non-leaf pages is the same as for the B$^+$-tree. This also holds for the directory pages with the additional maintenance of the directory page list.

4 Performance Evaluation

4.1 Benchmark Description

We implemented the CH-index, H-tree, and CG-tree in C++ using the Exodus Storage Manager [2]. Index nodes are represented as small Exodus objects of 4008 bytes. For all experiments reported in this paper, the database contained 600,000 entries of 12 bytes (we used Exodus OIDs as entries). The entries were distributed over either 8 indexed sets or 40 indexed sets.

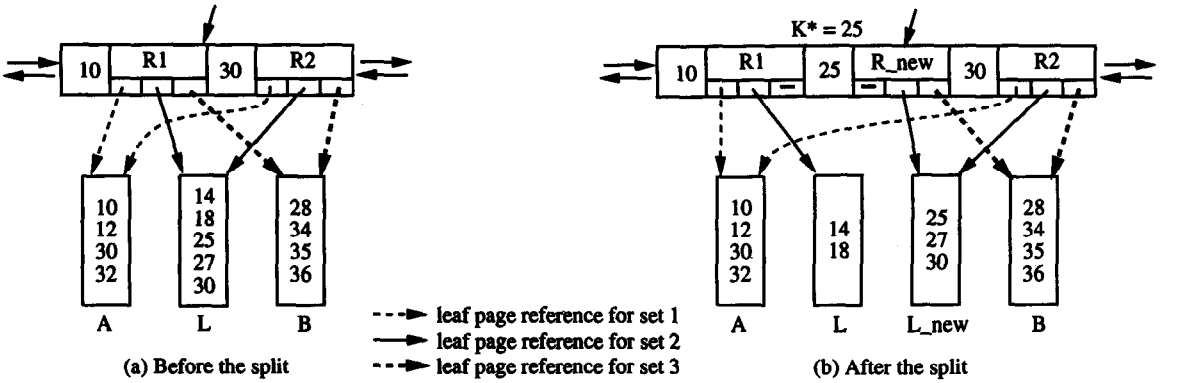


Figure 10: Splitting of leaf pages (arbitrary separation key)

The key size was 4 bytes, the size of a page reference 4 bytes. The number of different key values varies in the experiments from 100 to 600,000. In case of 600,000 key values, there is exactly one index entry for each key value (unique index).

We measured the number of page accesses of the index structures for exact match queries and range queries. For range queries, the search range comprises 10% of all key values.

The CH-index and the CG-tree do not require any additional structure over the set of indexed sets but the H-tree needs a definite set hierarchy. Hence, we defined a left-deep hierarchy for the 8-sets databases and a balanced hierarchy for the 40-sets databases (Fig. 11).

We measured the performance of the three index structures for several kinds of exact match queries and range queries: *downward* and *upward*. In the downward (upward) case, the queried sets are selected according to increasing (decreasing) set indices. This corresponds to a top to bottom (bottom to up) traversal of the set hierarchy. More specifically, the queried sets for the 8 and 40 indexed sets databases are:

8-Sets Databases

Number of Queried Sets	Queried Sets (downward)	Queried Sets (upward)
1	s_1	s_8
2	s_1, s_2	s_7, s_8
3	s_1, s_2, s_3	s_6, s_7, s_8
4	$s_1, \ldots, s_4$	$s_5, \ldots, s_8$
6	$s_1, \ldots, s_6$	$s_3, \ldots, s_8$
8	$s_1, \ldots, s_8$	$s_1, \ldots, s_8$

40-Sets Databases

Number of Queried Sets	Queried Sets (downward)	Queried Sets (upward)
1	s_1	s_{40}
2	s_1, s_2	s_{39}, s_{40}
4	$s_1, \ldots, s_4$	$s_{37}, \ldots, s_{40}$
8	$s_1, \ldots, s_8$	$s_{33}, \ldots, s_{40}$
12	$s_1, \ldots, s_{12}$	$s_{29}, \ldots, s_{40}$
20	$s_1, \ldots, s_{20}$	$s_{21}, \ldots, s_{40}$
30	$s_1, \ldots, s_{30}$	$s_{11}, \ldots, s_{40}$
40	$s_1, \ldots, s_{40}$	$s_1, \ldots, s_{40}$

The *Number of Queried Sets* will be used to label the x-axis of subsequent figures.

Kim et al. [7] identified three types of key distributions. In a *disjoint* distribution each key value is found in only one set. In a *total inclusive* distribution each key value is found in all sets. In a *partial inclusive* distribution each key value is found in some of the indexed sets. Besides the distribution of key values among set members, the cardinalities of the indexed sets and the (set specific) distributions of the key values have a strong impact on the performance of the multiple set index structures.

Due to lack of space, we present only part of the benchmark results. More results including thorough analysis' of the influence of several different distributions, the influence of the set hierarchy on the performance of the H-tree, the number of leaf and non-leaf pages, and the heights of the three different multiple set index structures can be found in [6].

First, we report the experiments where all sets have the same size, i.e., $600{,}000/n$ with n being the number of indexed sets, the key values are uniformly distributed across the indexed sets, and 100 and 600,000 key values are used.

For 100 key values, we have a total inclusive distribution. For 600,000 key values, the keys are unique and, hence, we have a disjoint distribution. These two cases are extreme cases, challenging all three multiple set index structures. The results for a moderate database with 20,000 key values are described in the next section (see also Fig. 4 in Sec. 2). All experiments were repeated 50 times and the results were averaged.

4.2 Benchmark Results

In the following figures, the x-axis of the plots is labeled with the *Number of Queried Sets*, the y-axis with the number of page accesses. The different curves within each figure are labeled by the according multiple set index structure CH, H, or CG. In parentheses, the kind of the selection of the queried sets, i.e., downwards or upwards, is specified by *down* or *up*, respectively. If an index structure X is not sensitive to the selection of the queried sets, there is only one curve

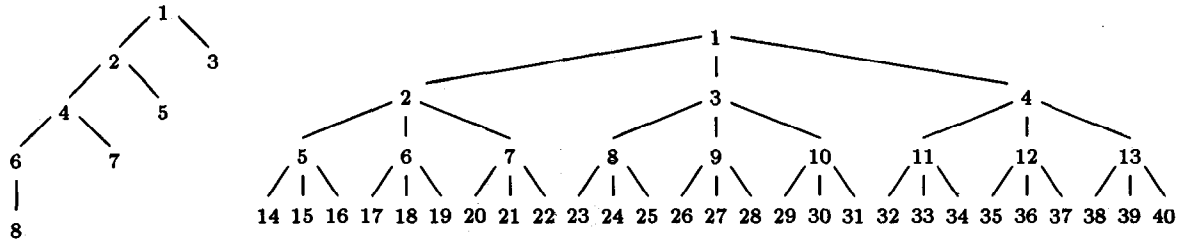


Figure 11: Set Hierarchies for the 8-Sets and 40-Sets Databases

labeled by $X(down\ \&\ up)$.

Uniform set sizes

The first row of Fig. 12 gives the results for exact match queries with unique key values. With constantly 3 page accesses, the CH-index exhibits the best performance. The CG-tree reads $2+n$ pages, where n is the number of queried sets. The performance of the H-tree depends on the selection of the queried sets. This is due to the implied differences in traversing the links between the nested set trees (cf. Fig. 3 in Sec. 2.2). The CG-tree clearly outperforms the H-tree in both (up and down) cases. If all sets are queried, the CH-index outperforms the CG-tree by a factor of 3 (14) in case of 8 (40) indexed sets.

The second row of Fig. 12 gives the results for range queries with unique key values. Both set grouping indexes (H- and CG-tree) clearly outperform the CH-index. This is due to the clustering of all entries of all indexed sets in the CH-index. Additional overhead occurs within the CH-index for the set directories being maintained in every leaf page record. For 8 indexed sets, the performance of the H- and the CG-tree is almost equal. For 40 indexed sets, the CG-tree performs about 15% better than the H-tree — due to the expensive link traversals of the H-tree.

The third row of Fig. 12 gives the results for exact match queries with 100 different key values. This database contains for a given key value more entries than would fit on one leaf page. Since the CH-index clusters the elements of all indexed sets by key, many overflow pages have to be allocated. Due to the organization of the overflow pages into a simple linked list, this introduces a dependency on the queried sets.[3] In the best case (down), where the queried sets correspond to the first pages of the list of overflow pages, the CH-index performs slightly better than the CG-tree (about 10%). In the worst case (up) — where sets s_8 and s_{40} are queried and, hence, all overflow pages are accessed — the CH-index is clearly outperformed by the CG-tree, if only part of the indexed sets have to be queried. The break even points are 7 (25) queried sets for 8 (40) indexed sets. The CG-tree clearly outperforms the H-tree in all cases.

The fourth row Fig. 12 gives the figures for range queries with 100 different key values. Due to its link traversals, the H-tree is outperformed by the CG-tree. In case of 8 indexed sets, the CH-index in its best case (down) performs better than the CG-tree. The break even point between the worst case (up) of the CH-index and the CG-tree still is at 7 queried sets. For 40 sets the CG-tree performs better than the CH-index.

Note that for uniform set sizes, the CG-tree is the most robust index structure since it is the only one independent of the selection of the queried sets.

Child-weighted set sizes

Now, we investigate the performance of the index structures under varying, child-weighted set sizes: for the 8 indexed sets case, set s_{i+1} contains twice as many elements as set s_i, $1 \leq i < 8$. For the 40 indexed sets case, set s_i contains i times the number of elements of the root set s_1. Hence, the child sets in the set hierarchy have larger cardinalities than their parent sets.

As the indexed sets are of different cardinalities, the sizes of the result sets depend on the queried sets. Hence, the number of pages to be read for exact match queries in case of overflow pages, and for range queries depend on the selection of the queried sets.[4]

The exact match query results for unique key values are qualitatively the same as those from the uniformly distributed databases (first row of Fig. 13). If all sets are queried, the CH-index outperforms the CG-tree by a factor of 3.3 (14) in case of 8 (40) indexed sets.

For the next rows of Fig. 13 the results differ. The plots of the second row visualize the results for range queries. Both, the H-tree and the CG-tree clearly outperform the CH-index. In all cases, the CG-tree needs less page accesses than the H-tree. Note the strong correlation between the H-tree and CG-tree results and the size of the queried sets. This correlation is nonexistent for the CH-index since — in this database — it does not allocate any overflow pages. The CH-index

[3] The entries in the (large) leaf page records (and their lists of overflow pages) of the CH-index are ordered by their set-ids, i.e., s_1, s_2, ..., s_n. Thus, exact match queries involving set s_i ($i \leq n$) have to traverse all pages of the accessed leaf page record occupied by the entries of the sets $s_1, \ldots, s_i$.

[4] Note, that the CH-index (in the case of overflow pages) and the H-tree additionally depend on the selection of the queried sets for the reasons stated earlier.

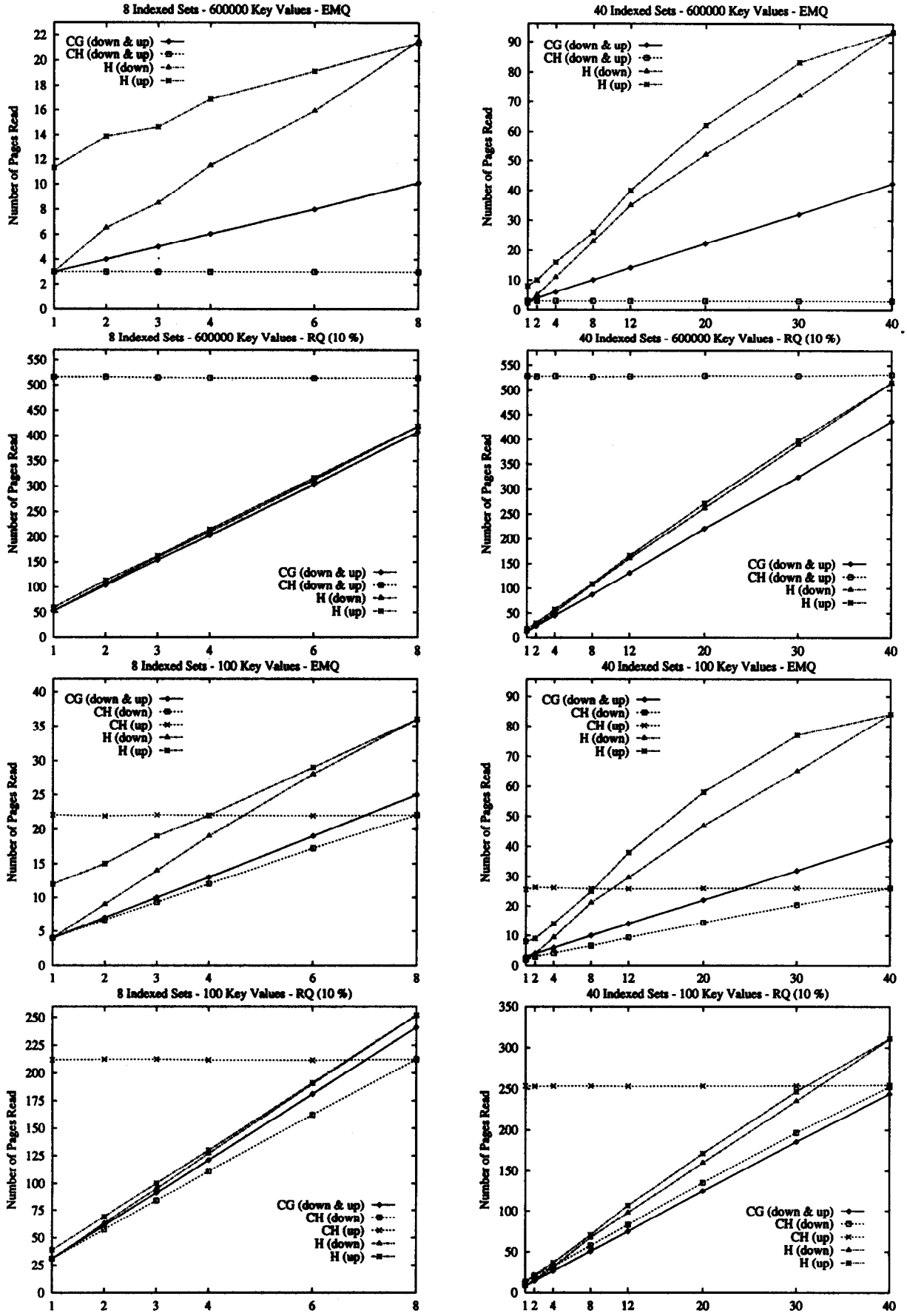


Figure 12: Uniform set sizes

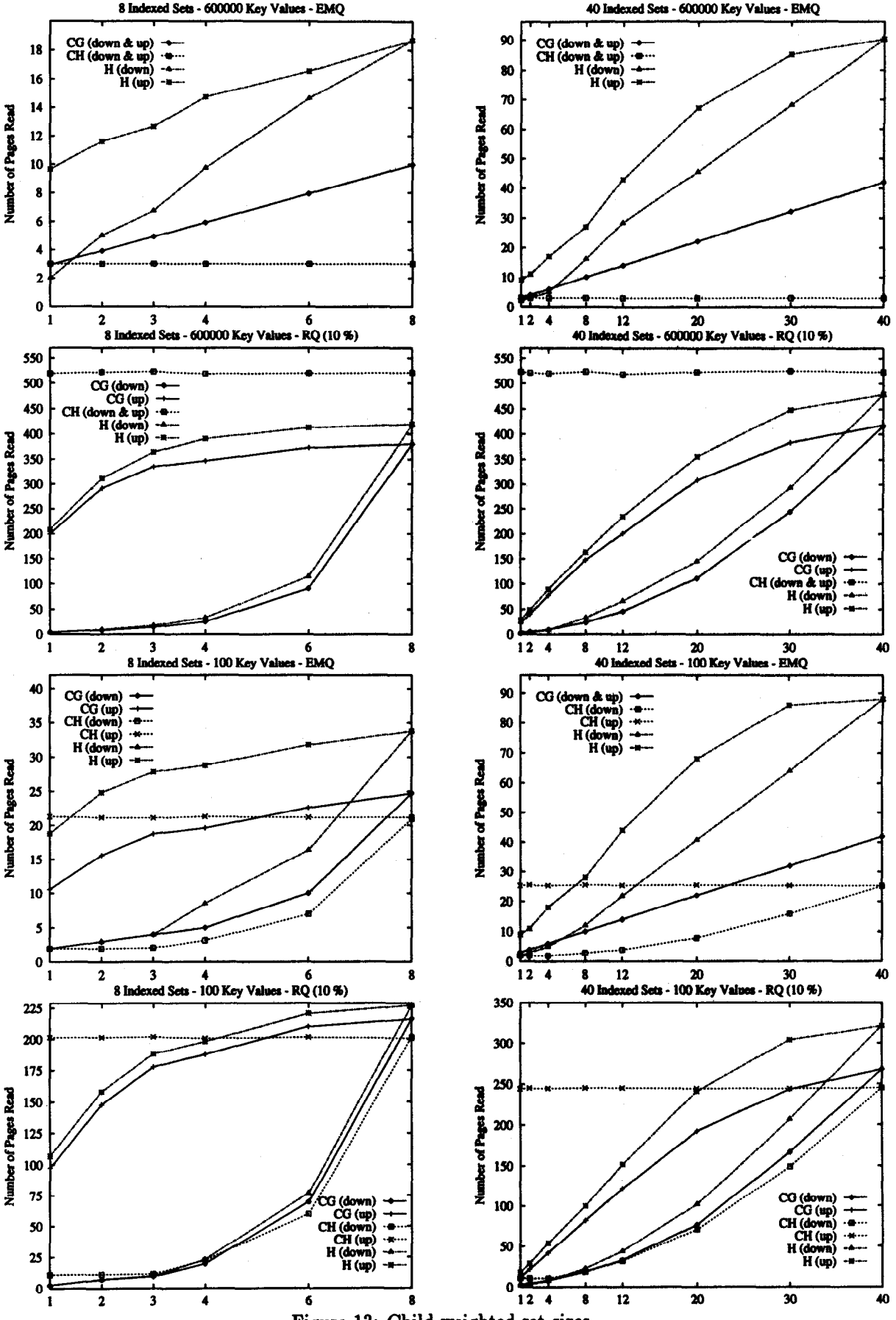


Figure 13: Child-weighted set sizes

accesses over 500 pages independently of the number and selection of the queried sets.

The third row contains the plots for exact match queries with 100 key values. In case of 8 indexed sets, the small number of key values results in overflow pages for all index structures. For the down case, the CH-index outperforms the CG-tree by approximately 20%, and the CG-tree outperforms the H-tree by about 40%. In the up case, the CG-tree performs best for less than 5 queried sets — the break even point. After the break even point, the CH-index performs best. In case of 40 indexed sets, only the CH-index needs overflow pages. It outperforms all other index structures in the down case. In the up case, there exists a break even point at 23 queried sets. Before this point, the CG-tree performs best; afterwards the CH-index.

The figures for range queries in the 100-keys database (fourth row) are more complex. The CH-tree needs overflow pages and, again, its performance strongly reacts to the up vs. down selection of the queried sets. In the down case, it slightly outperforms the CG-tree if more than 4 (12) sets are queried — for less than 4 (12) queried sets, the CG-tree performs best. In the up case, if only one set is queried, the CH-index is outperformed by the CG-tree by a factor of about 70 (40 indexed sets). This drastic figure diminishes with an increasing number of queried sets. The break even point is reached at 30 queried sets. In all cases, the CG-tree needs less page accesses than the H-tree.

5 Grouping of Indexed Sets

As we have seen in the previous section, the relative performance of the CH-index and the CG-tree strongly depends on the application profile. The differences can be an order of magnitude, e.g., for an exact match vs. range query profile. Hence, the database administrator finds herself in a dilemma. For a mix of exact match and range queries, only one extreme can be chosen: either support exact match queries or range queries. Hence, we propose a refinement of the CG-tree that exploits a grouping scheme for the indexed sets in order to provide the flexibility necessary for its tuning to a given application profile.

The idea is based on the observation, that in many applications, the indexed sets can be grouped such that all sets being members of the same group are often accessed together. For example, consider a personnel database containing the sets *TechStaff*, *Programmer*, *ProjLeader*, *GroupLeader*, and *Manager*. We assume that many applications execute separate queries to retrieve the administrative employees, the project related employees, and the technical staff. Thus, we can identify three groups of indexed sets — {*GroupLeader*, *Manager*}, {*Programmer*, *ProjLeader*}, and {*TechStaff*} — that are often co-accessed.

In order to support applications where groups of sets are often accessed together, the CG-tree is adapted as follows. Let $g_1, \ldots, g_m$ be the grouping of the indexed sets. Instead of having one list of leaf pages for each of the indexed sets, there is one list of leaf pages for each group. Every leaf page belongs to exactly one group, and stores only elements of the member sets of that group. There are two kinds of leaf pages:

1. For leaf pages that belong to a singleton group, no set directory is maintained in the leaf page records.
2. The records of leaf pages belonging to a group with more than one set contain a set directory, i.e., they have the same layout as in the CH-index.

The directory page records contain directories with m components, one for each group.

In the above example, the records of the leaf pages for the groups {*GroupLeader*, *Manager*} and {*Programmer*, *ProjLeader*} maintain set directories, whereas the records of the leaf pages belonging to the group {*TechStaff*} do not have a set directory.

Benchmark Results

We only give a brief validation of the grouping approach. We have chosen the moderate case with 20,000 key values, since these experiments also complete the benchmark results of the previous section. Again, we distinguish exact match queries and range queries, and selection of the queried sets by downward and upward traversal of the set hierarchy (ref. to the tables in Sec. 4.1). Because of space restrictions we included only the plots for the 40-sets databases; the results for the 8-sets databases can be found in [6].

The results are plotted in Fig. 14. For the CG-tree, three curves are shown corresponding to the different number of groups, i.e., 0, 4, and 10 groups. We applied the following grouping scheme:

40-Sets Databases

Number of Groups	Group Definition
4	$\{s_1, \ldots, s_{10}\}, \ldots, \{s_{31}, \ldots, s_{40}\}$
10	$\{s_1, \ldots, s_4\}, \ldots, \{s_{37}, \ldots, s_{40}\}$

The first plot of Fig. 14 shows that for exact match queries, the performance of the CG-tree with a small number of groups comes close to the performance of the CH-index. As shown by the second plot, the performance degradation on range queries is within acceptable limits.

6 Conclusion

We introduced a new set grouping multiple set index structure called CG-tree. The CG-tree outperforms

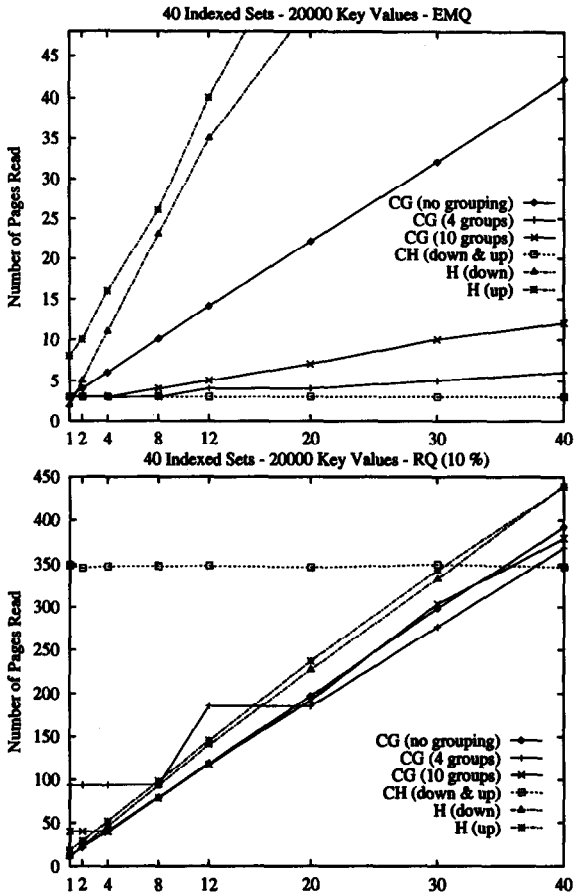


Figure 14: Grouping of indexed sets (the plots for the CG tree apply to both kinds of queries, down & up)

the also set grouping H-tree [8, 9] in all cases. Especially on exact match queries, the CG-tree shows a much better performance than the H-tree.

For exact match queries, the key grouping CH-index [7] is superior to the CG-tree in many cases; for range queries, the CG-tree is superior. Hence, the principle performance considerations for key and set grouping index structures depicted in Figure 1 are still valid.

These observations led us to the refinement of the CG-tree, where sets can be grouped according to their access probabilities. It was shown that with a small number of groups, the performance of the CG-tree on exact match queries comes close to that of the CH-index. The performance losses for range queries were moderate. This flexibility allows the database administrator to tune the CG-tree to better support application profiles containing exact match and range queries.

The following rules, being valid for a large variety of databases, summarize the results of our experiments:

- If no range queries occur, i.e., all queries are exact match queries, the CH-index should be applied.
- If the application contains range queries and the number of indexed sets is small, the CG-tree without grouping should be applied.
- If the application contains range queries and the number of indexed sets is large, the CG-tree with grouping should be applied.

Acknowledgments

We thank Kurt Moos for his help in implementing the index structures and extensive benchmarking.

References

[1] R. Bayer and E. McCreight. Organization and maintenance of large ordered indices. *Acta Informatica*, 1(3):173–189, 1972.

[2] M. J. Carey, D. J. DeWitt, J. E. Richardson, and E. J. Shekita. Object and file management in the EXODUS extensible database system. In *Proc. of VLDB*, pages 91–100, Kyoto. Japan, Aug 1986.

[3] S. Cluet and C. Delobel. A general framework for the optimization of object-oriented queries. In *Proc. of SIGMOD* 1992, pages 383–392.

[4] T. Härder. A generalized access path structure. *ACM TODS*, 3(3):285–298, Sep 1978.

[5] A. Kemper and G. Moerkotte. *Object-Oriented Database Management.* Prentice Hall, N.J., 1994.

[6] C. Kilger and G. Moerkotte. A performance evaluation of index structures for multiple sets. TR 6/94, Fak. f. Informatik, Univ. Karlsruhe, 1994.

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

[8] C. Low, H. Lu, B. Ooi, and J. Han. Efficient access methods in deductive and object-oriented databases. In *Proc. of DOOD* 1991, pages 68–84.

[9] C. Low, B. Ooi, and H. Lu. H-trees: A dynamic associative search index for OODB. In *Proc. of SIGMOD* 1992, pages 134–143.

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

[11] B. Seeger and H. P. Kriegel. The buddy tree: An efficient and robust access method for spatial data base systems. In *Proc. of VLDB* 1990, pp 590–601.

[12] P. Valduriez. Join indices. *ACM TODS*, 12(2): 218–246, June 1987.