Efficiency of Hybrid Index Structures - Theoretical Analysis and a Practical Application

Richard Göbel, Carsten Kropf, Sven Müller
Institute of Information Systems
University of Applied Sciences Hof
Hof, Germany
{richard.goebel, carsten.kropf, sven.mueller}@iisys.de

Abstract— Hybrid index structures support access to heterogeneous data types in multiple columns. Several experiments confirm the improved efficiency of these hybrid access structures. Yet, very little is known about the worst case time and space complexity of them. This paper aims to close this gap by introducing a theoretical framework supporting the analysis of hybrid index structures. This framework then is used to derive the constraints for an access structure which is both time and space efficient. An access structure based on a B+-Tree augmented with bit lists representing sets of terms from texts is the outcome of the analysis which is then validated experimentally together with a hybrid R-Tree variant to show a logarithmic search time complexity.

Keywords— hybrid index structures, theoretical analysis, experimental validation

I. INTRODUCTION

Modern database systems often manage data of multimedia types. Texts, images or video data are stored inside those database systems. Some approaches with specialized database systems which allow storing and retrieving those data fast exist. Relational database management systems are still the most used technique, especially as data stores in enterprises, although NoSQL databases are also present. Mixing up different storage systems does not help in retrieving the data fast, because of having to search multiple systems and generating a finally intersected result set at the end. This implies, on the one hand, a large overhead of temporarily allocated memory and, on the other, a large overhead of time as the distinct search results must be combined to a final result set.

Most existing hybrid access structures focus on the efficient storage and retrieval of data composed by textual and geographical data. In this paper, we focus on a probably more common scenario of data consisting of texts and conventional relational (single-valued) data sets. For this purpose the access structure is based on a conventional B+-Tree augmented with bit lists for indicating the presence of terms below a node. Besides this structure, also an R-Tree based one is evaluated.

Although several of these hybrid approaches with the ability to index data of this mixed type are present, there is no evidence about the temporal and spatial worst case complexity.

The major contributions of this work are a theoretical basis to analyse hybrid access structures, an in-depth analysis of index structures leading to the theoretical construction of a hybrid index structure and an analysis of asymptotic time and space complexity (see section III).

Finally, a practical construction and evaluation of the previously analysed hybrid index structure with focus towards the theoretical analysis (see section IV) is carried out. Based on a lack of space, related work is only discussed shortly.

II. RELATED WORK

New hybrid indexing strategies, enhancements, variations and compositions of existing concepts, like the B-Tree [1] or the R-Tree [2] have been proposed to address performance issues on heterogeneous data. Approaches are present treating terms differently according to the occurrence frequency like [3]. Also a couple of different hybrid indexing methods or methods for management of data in hybrid data spaces like [4] exists. Approaches like [5] (KR*-tree), [6] ((M)IR*-Tree) or [7] (bR*-tree) investigate, among others, the use of hybrid index structures combining textual and spatial retrieval utilizing the R-Tree [2] or its variants (e.g. R*+Tree [8]) which augment the R-Tree with certain secondary structures (bitlists or inverted lists) to enable set annotations at R*-Tree elements. Approaches like [9], [10] or [11] represent hybrid index structures for textual and spatial types which differentiate the treatment of textual entries based on the relative or absolute term frequency.

III. ANALYSIS OF ACCESS STRUCTURES

This section analyses the worst case time and space complexity of hybrid access structures. For this purpose we will introduce a formal notation as a general basis for analysing non-trivial access structures.

The general idea to formalise an index structure is similar to the work of Hellerstein et al. (e.g. [12]). The differences in our approach are motivated by the fact that this paper deduces upper bounds for search time complexity instead of lower bounds as in [12].

We will also show that a hybrid tree providing information about both single-valued and multi-valued columns in the upper nodes of the primary tree structure ensures a time complexity of $O(\log(n) \cdot m)$ and a space complexity of $O(n)$. The upper nodes of the tree only have to be sorted according to the single-valued column.
A. Basic Definitions

For analysing the access structure we will consider a database table with “normalised” columns containing single values and “non-normalised” columns with multiple values. Although most of this analysis is more generic we will assume that a multi-valued column contains a set of terms. We will denote the set of entries for such a table by a capital $E$ and individual entries by $e$:

$$E = \{e_1, \ldots, e_n\}$$  (1)

For reasons of simplicity we assume a single set of values $V$ for all columns. A set of $k$ projection functions $p_i$ retrieves the values of the $k$ individual columns:

$$p_i : E \rightarrow 2^V \text{ with } i = 1, \ldots, k$$  (2)

Projection functions may also be applied to sets of entries:

$$p_i \{\{e_1, \ldots, e_j\}\} = p_i (e_1) \cup \ldots \cup p_i (e_j)$$  (3)

Single-valued (normalised) columns contain not more than one value per entry:

$$\forall e \in E : |p_i (e)| \leq 1$$  (4)

With $q_i$ we denote the intersections between the sets of values in the related column:

$$q_i \{\{e_1, \ldots, e_j\}\} = p_i (e_1) \cap \ldots \cap p_i (e_j)$$  (5)

The key idea of many access structures $\mathcal{A}$ is the assignment of entries to groups which are not necessarily disjoint:

$$\mathcal{A} = \{N_1, \ldots, N_p\} \text{ with } N_1, \ldots, N_p \subseteq E$$  (6)

With this approach not all of these groups need to be searched. For this purpose, each group $N$ usually corresponds to a value $v$ which occurs in all entries of $N$ at the related column $i$:

$$\forall N \in \mathcal{A} \exists v \in V : v \in q_i (N)$$  (7)

It is also important that each group $N$ provides every entry $e$ which contains the corresponding value $v \in q_i (N)$ in the related column:

$$\forall v \in V, e \in E, N \in \mathcal{A} : v \in p_i (e) \land v \in q_i (N) \Rightarrow e \in N$$  (8)

An index structure with this definition is usually called an inverted index. This definition, however, is sufficiently generic to represent the group of entries referenced from the (leaf) nodes of a tree structure (e.g. a B-Tree) for a normalised column as well ($\forall e \in E : |p_i (e)| \leq 1$). With condition (7) and condition (8) the groups of the access structure for a single-valued column are disjoint.

B. Complexity of Queries Addressing Single Columns

With the definitions from the previous section we are ready to introduce complexity measures for time and space required for processing queries. A simple search condition for a column $i$ is a set of alternative values $C_i \subseteq V$. With this approach we cannot only model conditions specifying a single value for a column but also other types of conditions like search ranges. All entries $e \in E$ which contain at least one value of this set ($p_i (e) \cap C_i \neq \emptyset$) are part of the result set.

Complex search conditions consist of multiple search conditions which may not only refer to different columns but also to the same column. An example is a set of words which all need to be included in a text column. For the lack of space we do not provide a formal definition of complex search conditions in this paper.

We need to visit a group of an access structure if at least one entry in the group satisfies the search condition. Accordingly we define the function visit returning exactly these groups:

$$\text{visit}(C_i, \mathcal{A}) = \{N | N \in \mathcal{A} \land (p_i (N) \cap C_i) \neq \emptyset\}$$  (9)

The result set for a search condition is the union of all groups of the access structure which need to be visited:

**Lemma 1.** Let $C_i = \{v_1, \ldots, v_p\}$ be a simple search condition and $\mathcal{A}$ be an access structure. Then the following function “result” provides all entries which satisfy $C_i$:

$$\text{result}(C_i, \mathcal{A}) = \bigcup_{N \in \text{visit}(C_i, \mathcal{A})} N$$

This lemma follows directly from conditions (7) and (8). With these definitions we are ready to specify functions as complexity measures for space and time. The first function sumarises the number of entries for all groups in the access structure $\mathcal{A}$ which need to be visited for a given search condition $C_i$:

$$\text{time}(C_i, \mathcal{A}) = \sum_{N \in \text{visit}(C_i, \mathcal{A})} |N|$$  (10)

The spatial complexity is given by the function space summarising the sizes of all groups of the access structure:

$$\text{space}(\mathcal{A}) = \sum_{N \in \mathcal{A}} |N|$$  (11)

With these definitions and the previous lemma the search time of an access structure is limited by the size of the result set.
Lemma 2. Let \( C_i = \{ v_1, \ldots, v_p \} \) be a simple search condition and \( A \) be an access structure. Then the search time is limited as follows:

\[
\text{time}(C_i, A) \leq p \cdot |\text{result}(C_i, A)|
\]

Proof. With the definition of the function \( \text{time} \) we get

\[
\text{time}(C_i, A) = \sum_{N \in \text{visit}(C_i, A)} |N|
= |N_1| + \ldots + |N_p|
\]

with \( \{ N_1, \ldots, N_p \} = \text{visit}(C_i, A) \)
and \( N_i \subseteq N_1 \cup \ldots \cup N_p \)
for \( i = 1, \ldots, p \)
\[
\leq p \cdot |\text{result}(C_i, A)| \quad \text{with Lemma 1}
\]

A direct conclusion is that the search time is not dependent on the number of entries \(|E|\) from the considered table (constant time complexity).

The next lemma shows that the space required for an access structure is limited by the number of entries and the average number of values in the considered column.

Lemma 3. Let \( E = \{ e_1, \ldots, e_n \} \) be a set of entries, \( A_i \) an access structure for column \( i \) and \( \text{avg}_i(A_i) \) the average number of values in this column \( i \):

\[
\text{avg}_i(A_i) = \frac{|p_i(e_1)| + \ldots + |p_i(e_n)|}{n}
\]

Then the space required by the access structure for column \( i \) is limited by the following expression:

\[
\text{space}(A_i) \leq n \cdot \text{avg}_i(A_i)
\]

Proof. With conditions (7) and (8) every entry \( e \) occurs in not more than \( |p_i(e)| \) groups. As a consequence the summarised number of entries in the access structure \( A_i \) is limited by the summarised number of values in column \( i \) of all entries:

\[
\leq |p_i(e_1)| + \ldots + |p_i(e_n)|
\]

This expression can be rewritten as follows:

\[
\leq n \cdot \frac{|p_i(e_1)| + \ldots + |p_i(e_n)|}{n} \leq n \cdot \text{avg}_i(A_i)
\]

This indicates that the space of the access structure grows linearly with the number of entries in the table, if we assume that the average number of values in the considered column \( i \) can be limited by a constant. This seems to be a reasonable assumption for most applications.

C. Complexity of Queries Addressing Multiple Columns

Queries addressing multiple columns can be already supported by separate access structures generated for each column. A standard approach is the selection of the most selective condition for one of these columns returning fewer entries than the conditions for the other columns. This approach tries to linearly filter items retrieved after having searched for the more selective condition. The efficiency of this approach strongly depends on the selectivity of the individual conditions and the size of the result set. In some cases this might lead to a linear time complexity.

Many existing approaches supporting search conditions addressing multiple columns define sets for all possible combinations of values from these columns. We will model this idea by considering the cross product between the previously defined access structures, like hybrid index structures or extended B-Trees including combinations of values (e.g. concatenation, bit interleaving, etc.). Without loss of generality we will consider only the combination of pairs of access structures. The subsequent analysis can be easily extended by repeatedly combining the relevant access structures.

The following definition introduces the concept of a combined access structure by computing the cross product between two existing access structures.

Definition 1. Let \( A_i \) and \( A_j \) be two access structures for the columns \( i \) and \( j \). Then the combined access structure \( A_{ij} \) is defined as follows:

\[
A_{ij} = \{ N_i \cap N_j \mid N_i \in A_i \land N_j \in A_j \}
\]

The function for the set of visited nodes as well as the function for the time complexity can be easily extended for two search criteria:

\[
\text{visit}(C_i, C_j, A_{ij}) = \{ N \mid N \in A_{ij} \land (p_i(N) \cap C_i) \neq \emptyset \land (p_j(N) \cap C_j) \neq \emptyset \}\n\]

\[
\text{time}(C_i, C_j, A_{ij}) = \sum_{N \in \text{visit}(C_i, C_j, A_{ij})} |N|
\]

The function \( \text{visit} \) applied to a combined access structure \( A_{ij} \) which was derived from two access structures \( A_i \) and \( A_j \) provides exactly the cross product of the groups which need to be visited for the access structures \( A_i \) and \( A_j \):

Lemma 4. Let \( A_i \) and \( A_j \) be two access structures and \( A_{ij} \) be the combination of these access structures. Then for every pair of search conditions \( C_i \) and \( C_j \) the following property holds:

\[
\text{visit}(C_i, C_j, A_{ij}) = \text{visit}(C_i, A_i) \times \text{visit}(C_j, A_j)
\]

The proof of this lemma is straight forward. We can show that every element of one set is also an element of the other set using the above definitions. Also lemma 1 can be extended for combined access structures:

Lemma 5. Let \( C_i \) and \( C_j \) be two simple search conditions and \( A_{ij} \) be a combined access structure. Then the following function \( \text{"result"} \) provides all entries which satisfy both search conditions:

\[
\text{result}(C_i, C_j, A_{ij}) = \bigcup_{N \in \text{visit}(C_i, C_j, A_{ij})} N
\]
Proof. The result for a query with the search conditions \(C_i\) and \(C_j\) is the intersection of the result sets for the individual conditions:

\[
\text{result}(C_i, C_j, A_{ij}) = \text{result}(C_i, A_i) \cap \text{result}(C_j, A_j)
\]

With lemma 1 we get:

\[
\text{result}(C_i, C_j, A_{ij}) = \left( \bigcup_{N \in \text{visit}(C_i, A_i)} N \right) \cap \left( \bigcup_{N \in \text{visit}(C_j, A_j)} N \right)
\]

With lemma 4 we get:

\[
\text{result}(C_i, C_j, A_{ij}) = \bigcup_{N \in \text{visit}(C_i, A_i) \times \text{visit}(C_j, A_j)} N \quad \square
\]

With these results we can deduce an upper bound for the search time complexity of a combined access structure.

**Lemma 6.** Let \(A_{ij}\) be an access structure derived from the two access structures \(A_i\) and \(A_j\) and \(C_i = \{v_1, \ldots, v_p\}\) and \(C_j = \{v_{j1}, \ldots, v_{jq}\}\) be two search conditions. Then the search time is limited by the following expression:

\[
\text{time}(C_i, C_j, A_{ij}) \leq p \cdot q \cdot |\text{result}(C_i, C_j, A_{ij})| \tag{19}
\]

**Proof.** With the definition of the function \(\text{time}\) we get:

\[
\text{time}(C_i, C_j, A_{ij}) = \sum_{N \in \text{visit}(C_i, A_i) \times \text{visit}(C_j, A_j)} |N|
\]

With \(\{N_1, \ldots, N_p\} = \text{visit}(C_i, A_i)\) and \(\{N_{j1}, \ldots, N_{jq}\} = \text{visit}(C_j, A_j)\) we can rewrite this expression as follows:

\[
= |N_1 \cap N_{j1}| + \ldots + |N_p \cap N_{j1}| + \ldots + |N_1 \cap N_{jq}| + \ldots + |N_p \cap N_{jq}|
\]

Since each expression of the form \(N_i \cap N_j\) is a subset of \(\text{result}(C_i, C_j, A_{ij})\) we get the following upper bound:

\[
\leq |\text{result}(C_i, C_j, A_{ij})| + \ldots + |\text{result}(C_i, C_j, A_{ij})| + \ldots + |\text{result}(C_i, C_j, A_{ij})| + \ldots + |\text{result}(C_i, C_j, A_{ij})|
\]

This formula with \(p\) columns and \(q\) rows can be simplified as:

\[
\text{time}(C_i, C_j, A_{ij}) \leq p \cdot q \cdot |\text{result}(C_i, C_j, A_{ij})| \quad \square
\]

We can conclude from this lemma that also for a combined access structure the search time is not dependent on the number of entries in the table but only on the size of the result set and on the number of values specified by the search condition.

Unfortunately the space required for a combined access structure is not always acceptable. The following lemma provides an upper bound for the space complexity of such an access structure.

**Lemma 7.** Let \(A_{ij}\) be an access structure derived from the two access structures \(A_i\) and \(A_j\). Further let \(\text{avg}_i(A_i)\) be the average number of values for column \(i\) and \(\max_j(A_j)\) be the maximum number of values in column \(j\):

\[
\max_j(A_j) = \max \{|p_j(e)| \mid e \in E\}
\]

Then the space of the access structure \(A_{ij}\) is limited by the following expression:

\[
\text{space}(A_{ij}) \leq n \cdot \text{avg}_i(A_i) \cdot \max_j(A_j) \tag{20}
\]

**Proof.**

\[
\text{space}(A_{ij}) = \sum_{N \in A_{ij}} |N|
\]

With conditions (7) and (8) every entry \(e\) occurs in not more than \(|p_i(e)|\) sets from \(A_i\) and \(|p_j(e)|\) sets from \(A_j\). Since every group in \(A_{ij}\) is an intersection of a group from \(A_i\) and a group from \(A_j\), the entry \(e\) will not occur in more than \(|p_i(e)| \cdot |p_j(e)|\) groups. As a consequence the summarised number of entries in the access structure \(A_{ij}\) is limited by the following expression:

\[
\text{space}(A_{ij}) \leq |p_i(e_1)| + \ldots + |p_i(e_n)| \cdot |p_j(e)|
\]

We can replace every size of a set of values in column \(j\) by the maximum number of values in this column:

\[
\text{space}(A_{ij}) \leq |p_i(e_1)| + \ldots + |p_i(e_n)| \cdot \max_j(A_j)
\]

With the definition of \(\text{avg}_i(A_i)\) we get:

\[
\text{space}(A_{ij}) \leq n \cdot \text{avg}_i(A_i) \cdot \max_j(A_j) \quad \square
\]

Note that we get a tighter bound with this lemma if we choose the column with the lower maximum number of values as the column \(j\). But even in this case a combined access structure may have unacceptably high space requirements. On the other hand this lemma ensures an acceptable space complexity if one column has a low maximum number of values. This is in particular true for combinations of single-valued columns with arbitrary other columns.

**D. Tree Structure for Navigating to Sets of Access Structures**

In this section we consider the tree structure required for the navigation to the sets of entries from an access structure. Nodes of these trees relate to sets of values representing sets of entries below the node. We only take into account structures for secondary storage. As we want to prove a logarithmic worst case time complexity for our structure, the B(+) Tree is the choice
as the R-Tree cannot provide this. It is a well-known fact that a B+-Tree has the following time complexity for a range search retrieving $m$ entries from a table with $n$ entries: $O(\log(n) + m)$. Similar to many other tree structures a B+-Tree has linear space complexity: $O(n)$.

There are three different possibilities of navigation to the sets of an access structure: concatenation of structures for different columns, alternating the nodes on a path (we will not take into account this option, here) and augmenting nodes to provide information about both columns.

We will start with the concatenation of tree structures and analyse the time and space complexity depending on the order of these trees (multi-valued first or single-valued first). We will show that only the approach of "single-valued first" has reasonable space requirements but unfortunately a linear time complexity. Finally we will prove that the augmentation of the nodes in the primary tree with information about values from the multi-valued column of entries below this node is sufficient to ensure a logarithmic search time complexity.

For our analysis we consider the access structures $A_i = \{N_{i1}, \ldots, N_{ip}\}$ and $A_j = \{N_{j1}, \ldots, N_{jq}\}$ for the two columns $i$ and $j$. The primary tree for this access structure refers to column $i$ and supports the navigation to the $p$ groups of the access structure $A_i$. Every group $N$ of this primary access structure is the starting point of a secondary tree. A secondary tree supports the navigation to groups which were generated by intersections between the group $N$ and the groups from $A_j$. The space required by this structure depends on the size of the primary tree and the number and sizes of the secondary trees. The size of a secondary tree depends not only on the number of entries from the group $N$ which is the starting point for the tree but also the number of values in column $j$ of these entries.

The average number of entries of a group $N \in A_i$ can be derived by dividing the space (summarised number of entries) divided by the number $p$ of groups in the access structure $A_i$:

$$\frac{\text{space}(A_i)}{p} = n \cdot \frac{\text{avg}_{i}}{p}$$

(21)

A common function estimating the number of separate values versus the number of entries (size of a data set) is Heaps' law [13]. We will use the index $j$ in this formula to indicate, that the parameters refer to column $j$ in our case. Heaps' law may be applied to the average size of groups in access structure $A_i$, here:

$$p_j(N) \approx c_j \cdot n \cdot \left(\frac{\text{avg}_j}{p}\right)^{\beta_j}$$

(22)

The parameters $c_j$ and $\beta_j$ are application specific parameters.

The number of values is an upper bound for the number of groups in the secondary access structure. Since the space required for the tree grows linearly with this number of groups and we have $r$ trees as secondary access structures, the total space required by these secondary trees can be estimated by:

$$c_j \cdot \left(n \cdot \frac{\text{avg}_j}{p}\right)^{\beta_j} \cdot p$$

$$= c_j \cdot n^{\beta_j} \cdot \text{avg}_j \cdot p^{1-\beta_j}$$

$$= c_j \cdot n^{\beta_j} \cdot \text{avg}_j \cdot p^{1-\beta_j}$$

(23)

Now we may consider two cases depending on whether column $i$ or column $j$ is the multi-valued column. As an immediate observation, the expression $\text{avg}_j$ has the value 1 if $i$ is the single-valued column. Otherwise this expression may have a high value, if the average number of values for column $i$ is also high. Also the number of groups $p$ in access structure $\text{space}(A_i)$ is usually much greater for a multi-valued column than for a single-valued column. This may result in an unacceptably high space requirement for a solution with the primary access structure for the multi-valued column.

In the following we will assume, that the primary access structure relates to the single-valued column. In this case the average number of values in entries for this column is less or equal than one. This means that we can rewrite the previous expression as follows:

$$c_j \cdot n^{\beta_j} \cdot p^{1-\beta_j}$$

(24)

We may safely assume that the number of values for the single-valued column $i$ may not grow faster than linear with the number of entries $n$. Therefore, $n$ is an upper bound for $p$ and we get the following expression indicating a linear space complexity for this tree:

$$c_j \cdot n^{\beta_j} \cdot n^{1-\beta_j} = c_j \cdot n$$

(25)

The time complexity for searching the concatenated B+-Trees can be directly deduced from the time complexity of an ordinary B+-Tree. The first B+-Tree for column $i$ needs to manage all $n$ entries from the table and supports the retrieval of an intermediate result set with $m_i$ entries satisfying only the related condition for column $i$. The entries of the intermediate result set are included in not more than $m_i$ groups of the primary access structure $A_i$. Each of these groups is the starting point for another tree structure managing only the $m_i$ entries from the group. These secondary tree structures will deliver $m_k$ entries satisfying both search conditions. With this approach the summarised search time complexity is given by the following expression:

$$\log(n) + \sum_{k=1}^{m_i} (\log(n) + m_k)$$

(24)

Since the primary access structure refers to a single-valued column, their groups are disjoint and the appended secondary access structures manage also disjoint groups of entries. As a consequence we may rewrite the previous expression as follows, assuming that $m_i$ is the total number of entries satisfying both search conditions:

$$\log(n) + \sum_{k=1}^{m_i} \log(n_k) + \sum_{k=1}^{m_i} m_k = \log(n) + \sum_{k=1}^{m_i} \log(n_k) + m$$

(25)
In comparison to the time complexity of a conventional B+-Tree this time complexity has the additional addend $\sum_{k=1}^{n} \log(n_k)$. In the worst case this addend may result in a linear time complexity. We can show that by a simple example. We consider for this example a wide range search for column $i$ which is satisfied by all entries. Further we consider a condition for column $j$ defining a value which occurs in no entry. In this case the intermediate result set contains all entries ($m_i = n$) and all secondary tree structures need to be searched without retrieving a result. If we assume that every secondary tree structure has at least one node this expression would imply at least a linear search time complexity:

$$\sum_{k=1}^{n} \log(n_k) = \sum_{k=1}^{n} \log(n) \geq n.$$ 

This follows from the problem that secondary trees need to be searched even if they do not return any results. It can be avoided if only those secondary trees need to be searched which guarantee the retrieval of a certain number $\varepsilon$ of entries which is not dependent on the total number $n$ of entries. We can guarantee this property by adding information about all values of column $j$ which occur at entries below a node or below the groups from the primary access structure. In the next section we will introduce such a structure augmenting every reference to nodes or secondary trees with bit lists indicating which values are still available in entries following a reference.

We summarise the result of the analysis in the following theorem.

**Theorem 1.** Let $A_0$ be a combined access structure for the single-valued column $i$ and the multi-valued column $j$ using concatenated $B+$-Trees for the navigation to the groups from $A_1$. The primary tree refers to column $i$ and the secondary tree refers to column $j$. The references of the primary tree are augmented in such a way that a search continues beyond a reference if at least $\varepsilon$ entries from the referenced structure satisfy the conditions for column $j$. Then the search time is limited by the following expression where $n$ is the number of entries and $m$ is the size of the result set:

$$O(\log(n) \cdot \frac{m}{\varepsilon} + m)$$

**Proof.** In the previous section we could already prove that the time complexity of an access structure does not depend on the number of entries. Therefore, we may focus on the time required to navigate through the tree structure. Based on equation (25) and our assumption that the search continues only beyond a reference of the primary structure if at least $\varepsilon$ entries satisfy the search condition for column $j$, means that not more than $\varepsilon$ secondary tree structures are searched. Additionally, the number of entries in a secondary access structure is limited by $n$:

$$\leq \log(n) + \sum_{k=1}^{m} \log(n) + m$$

Now we may rewrite this expression as follows:

$$\leq \log(n) + \frac{m \cdot \log(n)}{\varepsilon} + m$$

Since we need to consider only the fastest growing term for the $O$-notation we get the following expression as the upper bound for the search time complexity (see lemma 2):

$$O(\log(n) \cdot \frac{m}{\varepsilon} + m)$$

In the next section we use bit lists to indicate the presence of a value below a node. This means that none of these frequencies is zero and therefore their product is also greater than zero. This expression also indicates that the parameter $\varepsilon$ grows with the number of entries managed by a secondary access structure. As a probably surprising consequence, the efficiency of the hybrid access structure increases with the number of entries stored in the secondary access structures!

IV. VALIDATION OF THE ANALYSIS

The theory proved that a hybrid indexing approach might be used for retrieval of normalised and non-normalised values combined inside a hybrid data space using a logarithmic complexity. Yet, it did not propose a concrete implementation structure with the ability to do so. One key challenge for such a structure is the combined representation of the two value types. A second one is the basic storage structure to be selected. The first one is solved by the augmentation of the elements of a base structure with a bitlist. The second one is strongly application dependent. For the validation of the analysis, we implemented two different storage structures. One uses a B+-Tree as the primary structure and the other an adopted R-Tree variant ensuring a logarithmic retrieval performance under certain circumstances (see [14]). Hence, two hybrid index structures supporting the combined storage of normalised and non-normalised values using a B-Tree variant or an R-Tree variant (proposed in [10]) are validated regarding their retrieval behaviour to evaluate the analysis given in the previous section regarding a real-world application. Based on the limited space available in realistic environments, an assumption must be made based on Zipf's Law [15] to limit the amount to a reasonable number. A parameter called $H_{Limit}$ is introduced which separates the set of non-normalised values into two subsets of high and low frequently occurring values. Only the high frequent values are stored inside the hybrid index.

The tests are carried out by using a specially preprocessed document set from a Wikipedia dump\(^1\). Textual analysis comprises stop word removal, character normalization and stemming. Geographical analysis consists of assignment of coordinates to textual occurrences. Texts represent the non-normalised and the coordinates the normalised value parts. The measurements consist of a set of 699 queries constructed from the AOL Query Log\(^2\). The number of documents at each measurement point is doubled which leads to the possibility of showing logarithmic query behavior. The B-Tree applies only the latitude values of the respective documents and the R-Tree uses both dimensions.

---

\(^1\) http://dumps.wikimedia.org/enwiki/20111007/enwiki-20111007-pages-articles.xml.bz2, accessed 2012-10-17

\(^2\) http://www.gregsadetsky.com/aol-data/, from 2012-10-17
The goal of this evaluation is solely to experimentally validate the logarithmic behaviour of the retrieval complexity for this kind of index structures.

A typical example of the retrieval behaviour can be seen in the plot of figure 1 (left side). The abscissa is scaled logarithmic which means that the given linear increase of cumulative sum of loaded pages leads to the fact that the total retrieval complexity behaves logarithmic. The table in figure 1 shows the combined results of the respective experiments for B-Tree and R-Tree. It can be seen that most cases show a logarithmic behaviour, even for more than one search term in the query. Approximately 87.84% of the queries could be answered with logarithmic complexity for the R-Tree and 93.7% for the B-Tree case. Hence, a clear tendency towards the logarithmic behaviour can be stated from these data. The remaining cases are explainable based on corpus specific properties. This includes, e.g., a change in the vocabulary. Hence, either constant query cases for a very small number or a sudden rise in the number of references leads to these non-logarithmic query behaviours.

Thus, summarizing, the evaluation using the given document set and the adopted hybrid access methods experimentally validates the theoretical proves from section III.

V. SUMMARY AND FUTURE PROSPECTS

In this paper we showed how to combine multiple index structures caring about single-valued and multi-valued attributes simultaneously. We formed a theoretical basis which proved the efficiency of one possibility of combinations to manage the given types of data. The practical evaluation carried out in a real world relational database management system confirmed the assumptions asserted in the theoretical analysis part.

Therefore, we could show in the theoretical analysis as well as in the practical evaluation that an augmented hybrid B-Tree indexing concept can achieve a logarithmic time complexity regarding typical queries. This also holds for a specialized hybrid R-Tree variant with logarithmic base complexity.

There are still objects of study regarding the particular storage parts in the hybrid access structure. Especially, the proper set up of the arbitrary user defined upper bound $H_{Limit}$ for frequent terms will have to be investigated.

REFERENCES


