MULTIDIMENSIONAL TERM INDEXING FOR EFFICIENT PROCESSING OF COMPLEX QUERIES

Michal Krátký, Tomáš Skopal and Václav Snášel

The area of Information Retrieval deals with problems of storage and retrieval within a huge collection of text documents. In IR models, the semantics of a document is usually characterized using a set of terms. A common need to various IR models is an efficient term retrieval provided via a term index. Existing approaches of term indexing, e.g. the inverted list, support efficiently only simple queries asking for a term occurrence. In practice, we would like to exploit some more sophisticated querying mechanisms, in particular queries based on regular expressions. In this article we propose a multidimensional approach of term indexing providing efficient term retrieval and supporting regular expression queries. Since the term lengths are usually different, we also introduce an improvement based on a new data structure, called BUB-forest, providing even more efficient term retrieval.

Keywords: term indexing, complex queries, multidimensional data structures, BUB-forest

AMS Subject Classification: 68P05, 68P10, 68P20, 14Q15

1. INTRODUCTION

The area of Information Retrieval (IR) [1, 18, 22] deals with problems of storage and retrieval within a huge collection of text documents. The most widely used models in the IR, the vector model and the boolean model, specify a form in which the documents are represented, queried and stored. In both mentioned models, the semantics of a document is characterized using a set of terms. In general, the term can be a word or a multi-word phrase. A common need to various IR models is an efficient term retrieval provided using a term index. For efficient querying, the term index must be implemented using a suitable persistent data structure [13], because we must keep in mind that the term number can raise up to $10^6$. The choice of data structure is crucial since it must reflect query types to be supported.

One of the most important term index application is the inverted list (used in the boolean model) based on B-tree [22], where to each term in the list a set of relevant documents is assigned. However, existing approaches of term indexing, e.g. the inverted list, support efficiently only simple queries asking for a term occurrence.

---

Work is partially supported by Grant Agency of the Czech Republic under grant No. 201/03/0912.
In practice, we would like to exploit also more sophisticated querying mechanisms, in particular queries based on regular expressions. Such a general type of query is highly applicable in unstructured as well as in structured information retrieval (e.g. native XML databases). In XML retrieval [23], such query could look like /titles/title[keywords/keyword='*computing'].

1.1. Existing approaches

In the B-tree-based inverted list, a single term retrieval is realized in logarithmical time complexity (relative to the number of terms). The B-tree indexes the terms according to lexicographical order and thus it can efficiently process also the right extension queries, i.e. regular expressions of form <\text{string}>*. However, execution of another regular expression query could lead to a sequential scan over the whole term index. In addition, a finite automaton corresponding to regular expression is necessary to be constructed for searching the B-tree and all the terms must be put into this automaton. Hence, the B-tree is not suitable structure for non-trivial term indexing. In [11] the String B-tree was introduced for efficient querying of substrings. Unfortunately, we cannot efficiently exploit neither the traditional cost-expensive approaches like automata nor pattern matching algorithms [5, 20] since the volumes of term collections are huge. Furthermore, an automaton cannot be stored on a secondary storage device efficiently, in such case an efficient page transfer between primary and secondary storage devices is impossible during a query processing.

In [1] the suffix arrays were proposed for term indexing. Suffix arrays are a space-efficient implementation of suffix trees. This indexing structure views the text as a single long string. Each position in the text is considered as a text suffix, i.e. a string that follows from that text position to the end of the text. It is a main memory data structure. Persistence of suffix arrays is known but the overhead is too large. Moreover, the “interior” of words is not possible to retrieve and thus it is not able to use suffix arrays for complex regular expression queries, e.g. for queries of form *<\text{string}>*. In [22] string rotations were proposed for efficient processing of regular expression queries. Each term is stored \(n\) times, where \(n\) is the number of the term characters. This fact can be a problem for term collections of large volumes thus string rotations do not provide an efficient solution of this problem.

Our objective was to propose a method for term indexing satisfying the following conditions: a persistent method, minimal storage overhead, and an efficient support of complex queries. In [7] a multidimensional approach was introduced for an efficient term retrieval. The fundamental idea resides in modelling the term as an \(n\)-dimensional point. The multidimensional approach enables to process regular expression queries. In Sections 2 and 3, the approach is briefly recapitulated. In Section 4 we introduce a new data structure, called BUB-forest, which allows efficiently index terms of variable lengths. In Section 6, some experimental results are presented and the last section concludes contributions of the paper.
2. MULTIDIMENSIONAL TERM INDEXING FOR EFFICIENT PROCESSING OF COMPLEX QUERIES

In our approach [7], we model a term as a point in n-dimensional vector space [17], where \( n \) is the maximal term length. The space is called term space. Each term is thus uniquely represented with an \( n \)-dimensional tuple whose each coordinate value determines a character from fixed alphabet (e.g. an ASCII code character).

**Definition 1.** (Term as \( n \)-dimensional tuple.) Let \( D \) be domain, \( D = \{0, 1, \ldots, 2^l - 1\} \), \( |D| = 2^l \), \( \Omega \) be a discrete finite \( n \)-dimensional vector space, \( \Omega = D^n \), \( A \) be the character alphabet, \( s = c_1c_2 \ldots c_{n-1}c_n \) be a term of length \( n \), where \( c_i \in A \) is a character, \( 1 \leq i \leq n \). Then \( n \)-dimensional point (tuple) representing the term \( s \) is defined as \( t_s = (\text{code}(c_1), \text{code}(c_2), \ldots, \text{code}(c_n)) \), \( t_s \in \Omega \), \( \text{code}(c_i) \in D \), where \( \text{code} : A \rightarrow D \) is a function which encodes a character \( c_i \) into a binary number of the bit-length \( l \).

If a term length is lower than \( n \), the extra tuple coordinates are set to a blank value (in this case to zero). The terms, as a set of multidimensional points, are then indexed using a spatial access method [4]. We have chosen the UB-tree [2] for indexing. Spatial access methods (the UB-tree respectively) support range queries algorithms which can be, in turn, applied for implementation of regular expression queries. Due to the proposed multidimensional term model such regular expression query implementation is possible and efficient.

2.1. Regular expression query construction

In multidimensional term indexing we exploit the maximal term length due to which we are allowed to construct regular expression queries by a combination of several range queries. In other words, such sequence of range queries can be defined as a complex range query.

**Definition 2.** (Complex range query.) A complex range query is defined as \( q_b \cup q_b_2 \cup \ldots \cup q_b_q \), where \( q_b_i \) (the \( i \)-th query box, \( 1 \leq i \leq q \)) is an \( n \)-dimensional hyper-rectangle defining simple range query. If \( \bigcap_{i=1}^q q_b_i = \emptyset \) then the complex range query is processed by \( q \) range queries. Symbol \( \cup \) is meant for geometrical union and \( \cap \) for geometrical intersection.

A comprehensive description of the complex range query construction for general regular expressions is out of scope of this paper where we demonstrate the query construction just for three forms of regular expressions. Let \( k \) be the length of string, \( n \) be the dimension of the term space \( \Omega \), and \( \max D_i \) be the maximal value of domain \( D_i \). A right extension query (expression \(<\text{string}>\ast \)) is performed by a single range query \( (c_1, c_2, \ldots, c_k, 0, \ldots, 0) : (c_1, c_2, \ldots, c_k, \max D_{k+1}, \ldots, \max D_n) \).

A left extension query (expression \( \ast<\text{string}> \)) is processed by a complex range query \( q_b_1 = (c_1, c_2, \ldots, c_k, 0, \ldots, 0) : (c_1, c_2, \ldots, c_k, 0, \ldots, 0) \cup q_b_2 = (0, c_1, c_2, \ldots, c_k, 0, \ldots, 0) \cup q_b_3 = (0, 0, c_1, c_2, \ldots, c_k, 0, \ldots, 0) \cup \ldots \).
A left-right extension query (expression \texttt{string}*) is processed by a complex range query \( q_{b1} = (c_1, c_2, \ldots, c_k, 0, \ldots, 0) \times (c_1, c_2, \ldots, c_k, \max_{D_{k+1}} \ldots, \max_{D_n}) \)

\( \cup \ q_{b2} = (0, c_1, c_2, \ldots, c_k, 0, \ldots, 0) : (\max_{D_1}, c_1, c_2, \ldots, c_*, 0, \ldots, 0) \cup \ldots \cup q_{b_{n-k+1}} = (0, \ldots, 0, c_1, c_2, \ldots, c_k) : (\max_{D_1}, \ldots, \max_{D_{n-k}}, c_1, c_2, \ldots, c_k) \).

Since the term space domains \( D_i \) are equal, we can label the maximal domain values \( \max_{D_i} \) as \( \max_{D} \), where \( \max_{D} = 2^l - 1 \). If the ASCII encoding is used then \( \max_{D} = 2^8 - 1 = 255 \). The complex query can be processed either by several "rectangular" range queries or by a single complex-shape range query (created by hyper-rectangle union). In [10] is described how to efficiently process complex-shape range queries. The result of a range query consists of all the relevant tuples. The tuples retrieved from the multidimensional term index are inversely decoded back into terms where the decoding is performed using an inversion function \( \text{code}^{-1} \). A list of decoded terms is returned to the user as a query result.

2.2. Term clustering using Z-ordering

The main idea of multidimensional structure UB-tree [2] as well as BUB-tree [9] resides in vector space ordering. If we order all the points within a discrete finite vector space we will get an ordering according to which the tuples can be indexed by a single-dimensional indexing structure (e.g. by the B+-tree). An important property of such ordering is that it should partially preserve the tuple distances. In other words, tuples that are "close" in the space (using a metric) should be "close" also within the ordering. In [8] the Gray codes and the Z-ordering were introduced for partial matching and range queries, respectively.

\textbf{Definition 3.} (Z-address.) Let \( \Omega \) be a discrete finite \( n \)-dimensional vector space, \( \Omega = D^n \), where \( D = \{0, 1, \ldots, 2^l - 1\} \), \( |D| = 2^l \). For a tuple \( t \in \Omega \) of the length \( n \), \( t = (a_1, a_2, \ldots, a_n) \) and a binary representation of the coordinate value \( a_i = a_{i,l-1}a_{i,l-2} \ldots a_{i,0} \), where \( a_i \in D \), \( l \) is the bit-length of the value \( a_i \), \( a_{i,j} \) is \( j \)-th bit value of \( a_i \), \( 1 \leq i \leq n \), \( 0 \leq j < l \), the function \( Z(t) \) (Z-address) is defined:

\[
Z(t) = \sum_{j=0}^{l-1} \sum_{i=1}^{n} a_{i,j}2^n+i-1.
\]

In our approach, we exploit the Z-ordering where a position of a tuple in the Z-ordering is called Z-address. If we calculate the Z-addresses for all the points of \( n \)-dimensional space \( \Omega \) we will get a Z-curve filling the entire space \( \Omega \). See Z-addresses and the Z-curve for 2-dimensional space 8 \times 8 in Figure 1 a.

In our application, the tuples of terms which are close in the term space could be considered as similar (from the lexical point of view). Due to the Z-ordering properties we can say that similar terms have close Z-addresses, see Example 1.
Example 1. (Transformation of terms into tuples.) Let us take terms to, we, dot, tab, and ten. In this case, function code uses the ASCII encoding. The domain is $D = \{0, 1, \ldots, 255\}$, the cardinality of domain is $|D| = 256, 1 \leq i \leq 3$. Maximal length of a term is 3 thus the dimension of the term space $n = 3$. Resultant term tuples are:

- $t_{to} = (\text{code}(\text{'t'}), \text{code}(\text{'o'}), 0) = (116, 111, 0)$
- $t_{we} = (\text{code}(\text{'w'}), \text{code}(\text{'e'}), 0) = (119, 101, 0)$
- $t_{dot} = (\text{code}(\text{'d'}), \text{code}(\text{'o'}), \text{code}(\text{'t'})) = (100, 111, 116)$
- $t_{tab} = (\text{code}(\text{'t'}), \text{code}(\text{'a'}), \text{code}(\text{'b'})) = (116, 97, 98)$
- $t_{ten} = (\text{code}(\text{'t'}), \text{code}(\text{'e'}), \text{code}(\text{'n'})) = (116, 101, 110)$

For clarity, we present the Z-addresses in path notation, where each binary Z-address (the left-most bit is the most significant) is shown as a block of $n$-bit numbers (in this example $n = 3$). Z-addresses and their path notations in decimal form are:

- $Z(t_{to}) = 000011011010100110010010 = 0.3.3.1.2.3.2.2$
- $Z(t_{we}) = 00001101101000011001011 = 0.3.3.1.0.3.1.3$
- $Z(t_{dot}) = 000111111100111011101001 = 0.7.7.4.2.7.2.2$
- $Z(t_{tab}) = 000111111100000110010010 = 0.7.7.1.0.1.4.2$
- $Z(t_{ten}) = 000111111101101111010010 = 0.7.7.1.4.7.4.2$

It is obvious from the example that Z-addresses of tab and ten or to and we are closer than Z-addresses of dot and tab or we and tab. In simple words, using the Z-ordering the similar terms are approximately clustered together.
3. THE BOUNDING UNIVERSAL B-TREE (BUB-TREE)

The Bounding Universal B-tree\(^2\) (BUB-tree) [9] is a multidimensional indexing structure exploiting the Z-ordering. The idea of the (B)UB-tree is based on the Z-ordering and the \(B^+\)-tree [21]. The \(B^+\)-tree is a balanced and persistent tree which provides logarithmical complexities for basic operations and a minimal overhead. A node utilization [13] over 50\% is guaranteed.

The (B)UB-tree indexes the Z-addresses of \(n\)-dimensional tuples into the \(B^+\)-tree. Within a (B)UB-tree node hierarchy the Z-regions represent clusters of tuples that are close (according to the Z-ordering). Each Z-region resides in a single disk page. Z-regions allow an efficient processing of multidimensional range queries. A Z-region \([\alpha: \beta]\) is defined as a space area bounded by the interval \((\alpha, \beta)\) on the \(Z\)-curve, \(\alpha \leq \beta\). In Figure 1b, four Z-regions in 2-dimensional space \(8 \times 8\) are depicted. In the case of UB-tree, the Z-regions define an ordered disjunctive partitioning of the entire \(n\)-dimensional space. The BUB-tree does not partition the entire space but it follows the tuples distribution thus it does not index the “dead space”. E.g. the empty interval \((31, 56)\) between Z-regions \([25:30]\) and \([57:62]\) in Figure 1b is not indexed by the BUB-tree.

Let there be \(m\) tuples inserted into the UB-tree (BUB-tree respectively). The space \(\Omega\) will then be partitioned by \(r\) disjunctive Z-regions \([\alpha_i: \beta_i]\), \(1 \leq i \leq r\). Let \(\alpha_{\text{min}} = 0\) be the minimal Z-address and \(\beta_{\text{max}} = 2^{n \times t} - 1\) be the maximal Z-address (see Definition 3). Then

\[
[\alpha_i: \beta_i] \cap [\alpha_j: \beta_j] = \emptyset, i, j \geq 1, i, j \leq r, i \neq j.
\]

For UB-tree

\[
\bigcup_{i=1}^{r} [\alpha_i: \beta_i] = [\alpha_{\text{min}}, \beta_{\text{max}}]
\]

\[
\alpha_1 = \alpha_{\text{min}}, \beta_r = \beta_{\text{max}}
\]

\[
\alpha_{i+1} = \beta_i + 1, \text{ for } i < r.
\]

For BUB-tree

\[
\bigcup_{i=1}^{r} [\alpha_i: \beta_i] \subseteq [\alpha_{\text{min}}, \beta_{\text{max}}]
\]

\[
\alpha_1 \geq \alpha_{\text{min}}, \beta_r \leq \beta_{\text{max}}.
\]

Because the shapes of Z-regions evolve during the tuples insertion, the BUB-tree does not index the “dead space” (contiguous empty space). This is an improvement over the UB-tree which indexes the entire space. Due to this fact the range query processing is more efficient in the BUB-tree. The (B)UB-tree hierarchy is depicted in Figure 2. The leafs contain indexed tuples, the inner nodes contain Z-regions.

Basic operations (insertion, deletion and point query) share a common technique: Transform the argument tuple into Z-address, find an appropriate leaf (the Z-region of which matches the tuple’s Z-address) and execute the operation on that leaf. If a

\(^2\)UB-tree, the ancestor of the BUB-tree, was introduced in [2].
node overflows by a tuple insertion, the node must be split. Using a suitable splitting policy a node utilization of up to 75% can be achieved [14]. In the case of BUB-tree, the splitting policies can be further enhanced by various heuristic methods keeping the BUB-tree's efficiency at maximum.

Fig. 2. This BUB-tree indexes the tuples presented in Figure 1b.

Unlike regions of other persistent multidimensional structures, e.g. the R-tree [12] based on a hierarchy of bounding boxes, the Z-regions of (B)UB-tree are disjunctive. This fact is very important especially for higher dimensionalities where the curse of dimensionality takes place. The disjunction of Z-regions significantly helps to reduce the negative aspects of searching in high-dimensional spaces. Some other structures are also designed to keep their regions disjunctive, e.g. the $R^+$-tree [3], but here the complexity just moves from the querying operations to the inserting operation.

It is clear from the previous descriptions that the BUB-tree storage overhead must be greater than by the UB-tree. However, in our implementation of the BUB-tree the leaf capacity is approximately two times higher than the inner node capacity. This refinement causes the BUB-tree index file is approximately of the same size as an equivalent UB-tree index file.

The most important and most difficult algorithm in the (B)UB-tree is the range query algorithm. An exponential (according to the dimension) algorithm is presented in [2]. A linear (according to the Z-address bit-length) algorithm is presented in [14, 16], but that description is very vague. For that reason, we have developed our own linear algorithm implementation [19] based on intersection operation of query box and Z-region.

4. BUB-FOREST

As in the case of term tuples, some tuple sets are of different dimensionalities. Such variously dimensional tuples can be indexed in a single n-dimensional vector space, where n is the maximal dimension over all the tuples in a given set. Shorter (lower-dimensional) tuples can be aligned to n-dimensional tuples, where the extra dimensions are set to a blank value. This solution was used in our previous UB-tree-based term indexing. On the other side, this simple approach has a major drawback. Since all the tuples are modelled in high-dimensional space there is a large amount of redundant information (the blank values) stored within the extra dimensions of possibly great number of aligned tuples.
We deal with term indexing, let us take a term dataset as an example. The term dataset was extracted from the TREC’s collections of text documents [15], in particular from LATIMES and FBIS collections. These collections contain 816,716 unique terms. Figure 3a shows a term length frequency histogram of the whole term dataset. Figure 3b is another interpretation of Figure 3a and shows how the number of terms grows with growing maximal allowed term length (i.e. with growing dimension of term space). We can observe that for majority of the terms the term length is smaller than 15. Thus, creation of 40-dimensional BUB-tree will lead to unnecessary storage overhead.

![Fig. 3. a) Term length frequency histogram of term dataset extracted from the TREC’s document collections LATIMES and FBIS. b) Number of terms with growing maximal term length.](image)

In this section we introduce a new multidimensional data structure, *BUB-forest*, which was designed just to avoid unnecessary storage and performance overhead when indexing variously dimensional tuple sets.

**Definition 4.** (The BUB-forest.) BUB-forest $BF_k(n_1, n_2, \ldots, n_k)$ is a data structure forest consisting of $k$ BUB-trees $BT_i(n_i)$, $1 \leq i \leq k$. Let $n$ be the dimension of the original high-dimensional vector space. Every BUB-tree $BT_i(n_i)$ indexes an $n_i$-dimensional space, where $1 \leq i \leq k$, $1 \leq j < k \Rightarrow n_{j+1} > n_j$, and $n_k = n$.

Let us have $m$ tuples $t_i$ of various dimensionalities, where $d_i$ is the dimension of the tuple $t_i$, $1 \leq i \leq m$. Then tuple $t_i$ is indexed by such BUB-tree $BT_j(n_j)$ for which $d_i \leq n_j \land j > 1 \Rightarrow d_i > n_{j-1}$, for $1 \leq i \leq m$, $1 \leq j \leq k$. If $d_i < n_j$ then dimension of $t_i$ is increased to $n_j$ and values $t_{ik} = \text{blank value} \in D, d_i < k \leq n_j$.

In other words, the BUB-forest indexes each tuple $t_i$ using such BUB-tree $BT_j(n_j)$ the dimension of which is the lowest but greater or equal to the dimension of $t_i$. The blank value is often zero. An example of BUB-forest $BF_2$, i.e. consisting of two BUB-trees, is presented in Figure 4. In this example, the BUB-trees are of the same heights but that is not a rule.
4.1. Operations on BUB-forest

Basic operations, i.e. insertion, deletion and point query, are performed by such a BUB-tree $BT_j$ to which the argument tuple is assigned.

![Example of BUB-forest $BF_2$.](image)

**Definition 5.** (BUB-forest range query.) Let a query box $QB$ is defined by two $n_{qb}$-dimensional boundary points $QL$ and $QH$, $n_{qb} \leq n$. The range query is in the BUB-forest $BF_k(n_1, n_2, \ldots, n_j, \ldots, n_k)$ processed as a sequence of range queries $QB_j$, defined by $n_j$-dimensional boundary points $QL_j$ and $QH_j$, on BUB-trees $BT_i(n_i)$, for which $n_{qb} \leq n_i$, $1 \leq i \leq k$. Let $1 \leq l \leq n_j$. If $l \leq n_{qb}$ then $QL_{jl} = QL_l$ and $QH_{jl} = QH_l$, else $QL_{jl} = QH_{jl} = \text{blank value}$. Query result of the BUB-forest range query is the union of the particular BUB-tree query results.

**Notes:** In similar way there could be defined forests also for other existing data structures, e.g. the B-tree or the R-tree. In the case of B-tree, there is only one dimension but the use of forest could serve as a compression tool since keys of variable lengths are stored in multiple B-trees. The forest data structure has been already applied to S-trees [6] (signature trees).

Since BUB-forest is a persistent data structure we can further consider two variants of disk cache. First, the BUB-trees of a BUB-forest share a single disk management and thus single disk cache. Second, each BUB-tree has its own disk cache. The latter possibility is obviously more efficient.

Each BUB-tree of a BUB-forest can index different number of tuples thus the tree heights can differ.

4.2. Storage volume reduction

Let us now compare the storage volume required when indexing variously dimensional tuple sets using a single BUB-tree and using a BUB-forest. Let the tuple set consists of $m$ tuples, where $m_i$ is the number of $d_i$-dimensional tuples, $\sum_{i=1}^{n} m_i = m$, $1 \leq i \leq n$.

The following calculations are only auxiliary since the disk management of BUB-tree as well as of BUB-forest produces some additional storage overhead. Suppose that single coordinate of a tuple requires $b$ bytes for storage.

When using a single space for indexing, the number of bytes $V_{bt}$ required to store the whole tuple set is $V_{bt} = m \times n \times b$. When using $k$ spaces of dimensionalities $n_1, n_2, \ldots, n_k$ for indexing, the number of bytes $V_{bf}$ required for the tuple set storage
Thus the number of saved bytes (when using $k$ spaces) is

$$V_{bt} - V_{bf} = \sum_{i=1}^{k} \left( \sum_{j=n_i-1}^{n_i} (m_j \times n_i \times b) - \sum_{j=n_i-1}^{n_i} m_j \times n_i \times b \right)$$

where $n_0 = 1$. If we consider the minimal storage volume $V$ for the tuple set $V = \sum_{i=1}^{n} m_i \times i \times b$ bytes, then obviously $V \leq V_{bf} \leq V_{bt}$. Equality $V = V_{bf}$ can be achieved if we increase the number of BUB-trees in BUB-forest to $k = n$. Simultaneously, we must realize that greater number of BUB-trees in BUB-forest leads to greater number of range query executions. Thus, the number of BUB-trees should be chosen heuristically, following the statistical distribution of tuples. In general, the lower-dimensional BUB-trees should index major part of the tuple set.

**Example 2.** (Storage volume reduction for term index.) Let us take the term dataset used in Figure 3. Let the maximal term length be $n = 40$. When using 40-dimensional space, the storage volume will be $V_{bt} = 816,716 \times 40 \times 1 = 31.2$ MB.

If we use $k$ spaces of dimensionalities 9, 17 and 40, we will get storage volume $V_{bf} = 509,258 \times 9 \times 1 + 280,050 \times 17 \times 1 + 27,408 \times 40 \times 1 = 10$ MB. The smallest possible storage volume is $V = 7.1$ MB. This simple example shows that the BUB-forest saves (theoretically) 68% of the single BUB-tree's storage volume. Furthermore, the storage overhead is still about 41% higher when compared with the ideal case.

Usage of BUB-forest significantly reduces the storage volume and it can be calculated that the additional BUB-forest overhead is relatively low (thanks to the node utilization over 50%). The number of BUB-trees in BUB-forest was chosen in order to maximize the range query efficiency. Section 6 presents experimental results which prove the above mentioned auxiliary outcomes.

**5. COST ANALYSIS**

A simple range query is processed by retrieval of those Z-regions (BUB-tree nodes respectively) that intersect a given query box. Let $r$ be the number of such Z-regions and $m$ be the number of indexed tuples. Then complexity of the range query is $O(\log_c(m) \times r)$, where $c$ is a fixed node capacity (tree arity respectively). If a complex range query is processed, the complexity is $O(\sum_{i=1}^{q} \log_c(m) \times r_i)$, where $q$ is the number of range queries and $r_i$ is the number of Z-regions intersecting the $ith$ query box. In the case of BUB-forest $BF_k$, the complexity of a particular range query is $O(\sum_{i=1}^{r} \log_c(m_i) \times r_i)$, where $r_i$ is the number of Z-regions intersecting the $ith$ query box (i.e. query box constructed for the BUB-tree $BT_i$).

In the case of term indexing, the blank value $= 0$. Let $q_{bt}$ be the number of range queries required for realization of a regular expression query using a single
high-dimensional BUB-tree. Then (according to Definition 5), the number of range queries required for performance of the regular expression query using BUB-forest can be smaller than \( k \times q_{bt} \).

**Example 3.** (Term indexing and querying in BUB-forest.) Let us take term dataset from the Example 1. We want to index this dataset in the BUB-forest \( BMT_2(2,3) \) consisting of one 2-dimensional BUB-tree and one 3-dimensional BUB-tree. Tuples of length 2 are modelled in 2-dimensional space (see Figure 5 a) while tuples of length 3 are modelled in 3-dimensional space (see Figure 5 b).

For regular expression query \( t^{*} \), the appropriate range query in the 3-dimensional term space will be \((\text{code}(t^{'}) , 0, 0) : (\text{code}(t^{'}) , \text{max}_D, \text{max}_D)\) (see Section 2). Execution of this query will retrieve all the tuples beginning with character 't'.

When using the BUB-forest, two query boxes must be constructed. The first one, for the 2-dimensional BUB-tree, will be \((\text{code}(t^{'}) , 0) : (\text{code}(t^{'}) , \text{max}_D)\). The second one, for the 3-dimensional BUB-tree, will be \((\text{code}(t^{'}) , 0, 0) : (\text{code}(t^{'}) , \text{max}_D, \text{max}_D)\). These query boxes, labelled as 1-1 (1-2 respectively), are depicted in Figure 5. The 1-1 query will retrieve term to while the 1-2 query will retrieve terms tab and ten.

As we can see, the coordinates of query boxes constructed for regular expression queries often contain either the same values or the minimal or maximal value, see Section 2 and Example 3. Our experiments have shown that such range queries (so-called narrow range queries) process only a small part of the BUB-tree which means the query boxes intersect only a small number of Z-regions during the range query execution.

6. EXPERIMENTAL RESULTS

In our experiments\(^3\), we used terms from the TREC's document collections (see Figure 3), including 816,716 unique terms. Several regular expression queries were

---

\(^3\)The experiments were executed on an Intel Pentium \(^\circ\) 4 2.4Ghz, 512MB DDR333, under Windows XP.
processed, each by the classical B-tree-based inverted list as well as by the multi-
dimensional approach – using UB-tree, BUB-tree, and BUB-forest. Several term
datasets were created, according to choice of the maximal allowed term lengths.
The size of a dataset in the case of maximal term length 9 was 509,258, in the case
maximal term length 40 it was dataset consisting of 816,716 terms. All the term
datasets were indexed by B-tree, UB-tree, BUB-tree and BUB-forest.

The following tables summarize the index characteristics:

**B-tree characteristics**
- tree height: 4
- nodes: 27,857–46,494
- utilization: 73–68%
- node capacity: 26
- index file: 9.5–52.9 MB

**UB-tree characteristics**
- \(|D|\) \(2^8\)
- dimension: 9–40
- tree height: 4
- nodes: 29,121–46,657
- Z-regions: 27,182–43,594
- utilization: 71.3–71.2%
- node capacity: 26
- node size: 355–1192 B
- index file: 9.9–53 MB

**BUB-tree characteristics**
- \(|D|\) \(2^8\)
- dimension: 9–40
- tree height: 4
- nodes: 26,358–37,027
- Z-regions: 24,322–34,180
- utilization: 66.9–66.6%
- leaf capacity: 31–36
- node capacity: 19
- node size: 422–1600 B
- index file: 10.6–56.5 MB

According to the maximal allowed term lengths, BUB-forests
\(BF_1(9) – BF_4(9, 13, 17, 40)\) were used.

**BUB-forest \(BF_4(9, 13, 17, 40)\) characteristics**
- \(|D|\) \(2^8\)
- index file: 20.8 MB
- \(BT_1\): dimension: 9
- tree height: 4
- items: 509,258
- nodes: 26,358
- Z-regions: 24,322
- utilization: 67.8%
- node capacity: 19
- leaf capacity: 31
- node size: 422 B
- \(BT_4\): dimension: 40
- tree height: 3
- items: 27,408
- nodes: 1,252
- Z-regions: 1,159
- utilization: 68.2%
- node capacity: 19
- leaf capacity: 36
- node size: 1600 B

The left, right and left-right extensions were tested. For the left extension, expressions
*soft*, *atom*, and *sub* were specified, for the right extension, expressions
*soft*, *less*, and *session* were specified, and for the left-right extension, expressions
*machine*, *nalist*, and *scient* were specified. In all cases, disk access
costs (DAC), number of compared terms, and query processing realtimes were ob-
erved with respect to increasing length of terms. The values of particular results
were averaged. The DAC was computed as the number of logical accesses to disk
pages times the size of disk page (which is fixed). In order to particular regular
expression query and the maximal length of terms, the number of retrieved terms
(i.e. the query selectivity) was between 0 and 1182.
Results of the right extension query are presented in Figure 6. In the case of B-tree, this query was performed very efficiently since the disk access costs and the number of compared terms are lowest. This fact is also reflected by the achieved real times. When compared with UB-tree and BUB-tree, the BUB-forest stores shorter Z-addresses which is reflected by lower disk access costs and the query processing times (see Figure 6 a and 6 c).

Results of the left-right extension query and the left extension query are presented in Figures 7 and 8. For processing of the queries by B-tree, all the terms must be sequentially retrieved and compared against the query (see Figure 7 b). The costs are thus linear according to the number of terms. For the multidimensional approach, the number of compared terms (Figures 7 b and 8 b) as well as the number of disk access costs (see Figure 7 a and 8 a) are lower than by the B-tree. For the multidimensional approach, the efficiency significantly decreases with growing dimension since for dimensionalities 9 and 15 the number of indexed terms increased by 50% but the number of compared terms increased up to 32 times during the queries execution.

The results show that the BUB-forest does not solve the problem of curse of dimensionality itself. However, storage of shorter Z-addresses is beneficial as we can observe from the disk access costs (see Figure 7 a and 8 a) as well as from the query processing realtimes (see Figure 7 c and 8 c). The efficiency improvement of
the BUB-forest over the UB-tree or the BUB-tree is up to 50%.

![Graphs showing disk access costs, number of compared terms, and query processing time for different structures: UB-tree, BUB-tree, BUB-forest.](image)

**Fig. 8.** Statistics of left extension test.

If we take into account that most of the real-world terms are shorter than 15 characters (see the term length distribution in Figure 3), we could claim that multidimensional approach is very efficient.

7. CONCLUSION

The paper described a multidimensional approach for an efficient term indexing for complex queries processing. The multidimensional approach provides broad usability for efficient term retrieval, especially in the sense of regular expression queries. When compared with the B-tree-based inverted list, for some types of regular expression queries the multidimensional approach offers much better efficiency. Regular expression queries are processed using UB-tree or BUB-tree by a single range query or by a complex range query (a sequence of range queries). This paper introduced a new indexing structure, the BUB-forest, even more reducing the storage and retrieval costs of the multidimensional approach.

In our future work, we would like to further improve the abilities and the efficiency of the multidimensional approach. In particular, we are going to develop method for a range query construction supporting all (or at least a significant subset) of regular expression queries. Furthermore, at the current state a single disk page can be retrieved and processed multiple times during the complex range query processing consisting of several simple range queries. For that reason, it could be useful to develop an algorithm executing the complex range query more efficiently. Basics of such algorithm were proposed in [10].

The boundary points of query boxes often have fixed coordinates when constructing regular expression queries and this fact negatively reflects in higher number of non-relevant Z-regions processing. Thus we want to enhance the index structures to better support this narrow range query.
ACKNOWLEDGEMENTS

We would like to thank Professor Jaroslav Pokorný, Charles University, Prague, Czech Republic, for many valuable comments.

(Received October 28, 2003.)

REFERENCES


Michal Krátký, Tomáš Skopal, and Václav Snášel, Department of Computer Science, VŠB Technical University of Ostrava, 17. listopadu 15, 708 33 Ostrava. Czech Republic. e-mails: michal.kratky, tomas.skopal, vaclav.snasel@vsb.cz