APPROXIMATE QUERIES ON SET-VALUED ATTRIBUTES

MIKOLAJ MORZY, ZBYSZKO KRÓLIKOWSKI, TADEUSZ MORZY

Institute of Computing Science
Poznan Univeristy of Technology

{Mikolaj.Morzy,Zbyszko.Krolikowski,Tadeusz.Morzy}@cs.put.poznan.pl

Abstract: Sets and sequences are commonly used to model complex entities. Attributes containing sets or sequences of elements appear in various application domains, e.g., in telecommunication and retail databases, web server log tools, bioinformatics, etc. However, the support for such attributes is usually limited to definition and storage in relational tables. Contemporary database systems don't support either indexing or advanced querying of set or sequence attributes, such as executing set containment or set similarity queries. In this paper we focus on approximate queries on set and sequence attributes. We present the notion of an approximate query and we review similarity measures proposed so far for such attributes. We introduce a new similarity measure that can be successfully used with sequences. We present the hierarchical bitmap index – a novel and efficient indexing technique for sets and show how the hierarchical bitmap index framework can be extended to incorporate sequences as well. We conclude with algorithms for efficient approximate query processing using the hierarchical bitmap index.

Keywords: set and sequence valued attributes, set index, sequence index, approximate queries, similarity measures

1. Introduction

Set-valued attributes provide a concise manner to represent complex objects appearing in many different application domains. Depending on the application domain a set valued attribute can be used to represent a set of products purchased by a customer during a single visit to the supermarket (retail databases), a set of pages and links visited by a user during navigation through a web site (web server logs), a set of objects appearing on a picture or video (multimedia databases). Sequence-valued attributes can be used whenever time dimension appears, e.g., sequence-valued attributes can represent the order of purchases made by a customer in a supermarket (retail databases), a sequence of phone calls (mobile phone company database), or occurrences of recurrent illnesses (medical database). Set-valued attributes are a part of the SQL3 standard, yet the support for such attributes in contemporary databases is usually limited to definition and storage of set-valued attributes in relational databases. In most implementations set-valued attributes are represented as attributes of user-defined type or nested tables. There are no SQL extensions to formulate set-oriented queries and no physical structures, such as indexes, to support efficient retrieval of sets. Although the need to extend standard SQL language with set containment operators has been long acknowledged, no practical implementations follow. To the best of our knowledge sequence-valued attributes are not supported by any commercial database management system.

The ability to perform set-oriented queries can be utilized by many advanced applications. Queries concerning set-valued attributes can be categorized into four main classes. Given a searched set (or sequence) provided by a user. *Equality queries* search for all tuples that are

identical with the searched set. Equality queries are useful in accurate medical diagnosis (finding patients with exactly defined symptoms) or in identifying suspicious credit card usages (finding customers who made a set of purchases in some special order). Second class of set-oriented queries are *subset queries*. These queries search for all tuples that entirely contain the searched set. Subset queries can be used to create target groups of customers (finding customers who bought specific set of products and could be targeted for some promotional offer) or identifying popular navigation paths in web sites (finding frequent subsequences of web page requests). Third class of set-oriented queries are *superset queries* that find all tuples that are entirely contained in a searched set. This type of query can be useful to discover special groups of customers. Assume that the searched set contains all products offered at a reduced price. A superset query can be used to find those customers who visited the supermarket only to profit from a discount. The last class of set-oriented queries are *approximate queries*. Approximate queries search for all tuples that are sufficiently similar to the searched set, according to some similarity measure.

Approximate queries have numerous practical applications. Consider an on-line music store. Each customer purchasing an album is presented with a set of automatic recommendations. These recommendations are built based on the history of previous purchases made by that customer. The system generates the set of records that are modestly similar to the albums already purchased by the customer. Highly similar recommendation would include most albums already possessed by the customer, while lowly similar recommendation would not be relevant to the customer profile. Other applications may require finding highly similar tuples, e.g., airport security system should identify suspicious individuals based on the images from the surveillance cameras. Here the airport security system should be able to quickly compute the similarity of passenger images using sets of characteristic features. On the other hand, some applications may depend on efficient querying for strongly dissimilar tuples, e.g., to quickly discover fraud credit card usages, to identify and analyze dissimilar customer behavior patterns, and so on.

In this paper we present an overview of similarity notions proposed so far for sets and sequences. We introduce a new similarity measure for sequences and discuss its usability. We present the hierarchical bitmap index – a novel indexing structure for sets and we show how the hierarchical bitmap index can be extended to efficiently index sequences of elements as well. We conclude with a presentation of algorithms for approximate query processing using the hierarchical bitmap index.

2. Related work

Processing of set-oriented queries attracted a lot of work from the scientific community and resulted in many proposals. Set containment operators were proposed in [7]. In [2] the authors proposed to process similarity queries on sets by transforming sets into vectors in Hamming space and reduce the problem to finding similar vectors in Hamming space using similarity filter index. Other proposals for set indexing resulted in the development of many index types, among them inverted files [11], signature trees [1], RD-trees [3], hash group bitmap indexes [9], and hierarchical bitmap index [8]. An interesting comparison of different indexing techniques for sets can be found in [4]. Indexing of sequences was seldom researched and resulted only in few proposals, e.g., [6]. Many similarity measures have been proposed so far, both for set similarity [10] and sequence similarity [5].

3. Definitions

Given a database D of tuples, let D = $\{t_1, t_2, \dots, t_n\}$, where each tuple t_i contains a set. Let q denote a finite set of elements (called the searched set). The focus of our interest is to efficiently process the approximate queries of the form: $\{t_i \in D: sim(t_i,q) \geq \alpha\}$ for some similarity measure *sim()* and similarity threshold α. To compute the similarity between two sets S_1, S_2 it is necessary to provide a similarity measure. Until now, many different measures have been proposed, e.g.:

- matching: $C_M(S_1,S_2)=|S_1 \cap S_2|$
- dice: $C_D(S_1,S_2)=2*|S_1 \cap S_2| / (|S_1|+|S_2|)$
- Jaccard's coefficient: $C_J(S_1,S_2)=|S_1 \cap S_2| / |S_1 \cup S_2|$
- overlap: $C_O(S_1,S_2)=|S_1 \cap S_2| / \min(|S_1|,|S_2|)$
- cosine: $C_{COS}(S_1,S_2)=|S_1 \cap S_2| / \text{sqrt}(|S_1| \times |S_2|)$

In our research we adopted the Jaccard's coefficient. It is simple and intuitive and can be successfully used in many real-world applications. Although it is not a metric (recall that the metric is a distance function $d(x,y)$ such that $d(x,x)=0$, $d(x,y)=d(y,x)$ and $d(x,z) \leq d(x,y)+d(y,z)$), it can be easily converted to a metric as $d(S_1,S_2)=1-C_J(S_1,S_2)$.

Given a database of sequences, let D=$\{s_1, s_2, \dots, s_n\}$ where each sequence s_i is an ordered list of event pairs $s_i=(e_1,t_1)(e_2,t_2),\dots,(e_n,t_n)$ and every event pair contains the event type e_j and the time of the event occurrence t_j. Let q denote a finite sequence of event pairs (called the *searched sequence*). The focus of our interest is to efficiently process the approximate queries of the form: $\{s_i \in D: sim(s_i,q) \geq \alpha\}$ for some similarity measure *sim()* and similarity threshold α. The most popular similarity measure for sequences is the *edit distance* $d(s_i,q)$ [5] which represents the distance between a sequence s_i and a sequence q as the amount of work that must be done in order to transform one sequence into another. To transform one sequence into another three operations can be used:

- inserting: Ins(e,t) which adds an event e occurring at time t to the sequence
- deleting: Del(e,t) which deletes an event e at time t from the sequence
- moving: Mov(e,t,t') which moves an existing event e from time t to t' in the sequence

Every operation has a certain cost assigned to it. The cost of an insert operation c(Ins(e,t))=w(e) where w(e) is a constant proportional to the reciprocal of occurrences of the event e in the database D (this is to make inserting of frequent events less expensive than inserting of rare events). The cost of a delete operation is the same as the cost of an insert operation, c(Del(e,t))=w(e). The cost of a move operation is c(Mov(e.t.t'))=V*|t-t'|, where V is a constant and |t-t'| is the length of the move (notice that this cost assumes that the compared sequences use the same magnitude of event occurrence times). The cost of an operation o_i is denoted as $c(o_i)$. The cost of an operation sequence $c(O_j)=\Sigma_k c(o_k)$ for all operations $o_k \in O_j$. The edit distance between two sequences $d(S_1,S_2)=\min\{c(O_j) \mid$ where O_j is an operation sequence transforming S_1 into $S_2\}$. This distance function is a measure, but it is difficult to compute and implement using a sequence index.

For this reason we propose another similarity measure for sequences. Following is a slightly different formulation of the problem of approximate queries on sequences. Let a sequence be an ordered list of elements $S=<e_1,e_2,\dots,e_n>$ and let $t_S(e_i)$ denote the time of occurrence of the event e_i in sequence S. Given sequences P, Q, let

- $P \cap Q = \{e_i \mid e_i \in P \wedge e_i \in Q\}$,
- $P \cup Q = \{e_i \mid e_i \in P \vee e_i \in Q\}$,

- $P \underline{\cap} Q = \{(e_i,e_j) \mid e_i \in P \wedge e_j \in P \wedge e_i \in Q \wedge e_j \in Q\}$
- $P \underline{\cap}^{\rightarrow} Q = \{(e_i,e_j) \mid e_i \in P \wedge e_j \in P \wedge e_i \in Q \wedge e_j \in Q \wedge t_P(e_i) \leq t_P(e_j) \wedge t_Q(e_i) \leq t_Q(e_j)\}$

There are three factors that affect the similarity ratio between the two sequences:

- element similarity: $sim_E(P,Q) = |P \cap Q| / |P \cup Q|$
- order similarity: $sim_O(P,Q) = |P \underline{\cap} Q| / |P \underline{\cap}^{\rightarrow} Q|$
- period similarity: $sim_P(P,Q) = \Sigma(|t_P(e_i)-t_P(e_j)| / |t_Q(e_i)-t_Q(e_j)|) / |P \underline{\cap}^{\rightarrow} Q|$, for all $(e_i,e_j) \in P \underline{\cap}^{\rightarrow} Q$

The overall similarity between the two sequences P,Q is a weighted sum of element, order, and period similarity:

$$sim(P,Q) = w_1 * sim_E(P,Q) + w_2 * sim_O(P,Q) + w_3 * sim_P(P,Q), \text{ where } w_1+w_2+w_3=1.$$

The element similarity factor (which is simply a Jaccard's coefficient of similarity between the sets of events appearing in P and Q) measures the pure set similarity and doesn't consider the order and time constraints of events in P and Q. The order similarity factor measures the percentage of event pairs (e_i,e_j) which occur in both sequences and preserve the same order in both sequences. Finally, the period similarity factor computes the relative difference of time gaps between all pairs of events (e_i,e_j) occurring in both sequences. In the next sections we'll show how this simple and intuitive distance measure can be successfully used to answer approximate queries concerning sequences.

4. Hierarchical bitmap index

The hierarchical bitmap index originates from the well-known S-tree structure. The main difference is the way indexed sets are represented in index keys. To remove set representation ambiguity a hierarchical structure is built for every indexed set. The index consists of a set of index keys, each of them representing an indexed set. Given a set-valued attribute A. Every index key contains a very long bitmap B. The length of the bitmap B is determined by the size of the indexed set domain |dom(A)| and the length of the machine word *l*. The bitmap B must be long enough to map every element $a_i \in dom(A)$ to a distinct bit, i.e., the length of the bitmap $b = l * \lceil |dom(A)| / l \rceil$. The bitmap B is then divided into m=b/*l* nodes, called the *index key leaves*. Every element in the domain of the indexed set $a_i \in dom(A)$ is mapped via a mapping function $f(a_i)$ to a k^{th} position in the bitmap B, $k \in <1,b>$.

Now we present the procedure of creating an index key for a single set. Given the indexed set $S=\{a_1,a_2,\ldots,a_n\}$. For the sake of simplicity we assume the mapping function $f(a_i)=i$. An element a_i is thus represented by the i^{th} bit of the bitmap B set to `1'. This bit is in fact the $j^{th}$ bit in the $k^{th}$ index key node, where $k=\lceil i/l \rceil$ and $j=i-(\lceil i/l \rceil-1)*l$. Every element in the indexed set is represented analogously. Therefore, the entire set S is represented by n bits set to `1' on appropriate positions in index key leaves. Index key leaves which contain at least one bit set to `1' are called *non-empty leaves* whereas index key leaves that are entirely set to `0' are called *empty leaves*.

The number of index key leaves must be large enough to uniquely represent every element from the indexed domain. For most applications this signifies tens or hundreds of thousands of bits. On the other hand, even for large average indexed set size the majority of index key leaves would be empty. This leads to the idea of compressing the information about index key leaves by the next level of inner nodes. Every bit in an inner node corresponds to a single index key leaf. If the referenced leaf is non-empty, then the appropriate bit in the inner node is set to `1', otherwise it is set to `0'. The ith index key leaf is represented by the j^{th} bit in the k^{th} inner node, where $k=\lceil i/l \rceil$ and $j=i-(\lceil i/l \rceil-1)*l$. Every next level of the inner nodes contains *l*-times less nodes then the prior level. This procedure is repeated recursively until the level is

reached on which only one node is sufficient to represent all inner nodes at the subsequent level. This single node at the highest level is called the *index key root*.

The maximum number of elements that can be indexed in a single index key is determined by two parameters, namely, the size of a single index key node l (usually this is a machine word, 32 or 64 bits) and the depth of the index key d. These parameters are dynamic and depend on the application domain. Shallow index keys are faster to process, but they limit the maximum number of distinct elements that can be represented in the index key. Deep index keys are slower to process, but allow to uniquely represent huge domains. Note that the average size of the indexed sets, which doesn't have to be known in advance, is not relevant to the construction of the index. For example, assume l=32 and d=4. The root of a single index key can store information about 32 inner nodes at level 2. Each of those nodes stores information about another 32 inner nodes at level 3, which results in 32^2=1024 inner nodes at level 3. Each inner node at level 3 represents 32 inner nodes at level 4. This gives 32^3=32768 index key leaves, each representing 32 different elements from the indexed domain. As the result, a single index key of the hierarchical bitmap index with l=32 and d=4 allows to uniquely represent sets with domain of the size 32^4=1048576 elements.

To better illustrate the idea of the hierarchical bitmap index consider the following example of a single index key construction.

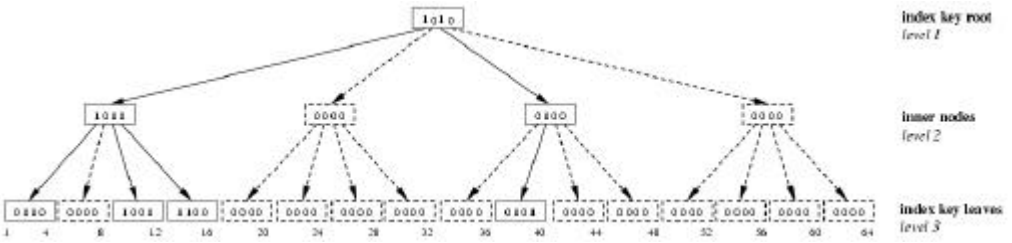

Figure 1 Single key of the hierarchical bitmap index

Example 1. Assume index key node length l=4 and index depth d=3. Assume also the mapping function $f(a_i)=i$. Given the set S={2,3,9,12,13,14,38,40}. The index key of the set S is depicted in Figure 1. At the lowest level 8 bits corresponding to the elements of the set S are set to `1', so index key leaf nodes 1,3,4 and 10 become non-empty (they are marked with a solid line). At the upper level 4 bits representing non-empty leaf nodes are set to `1'. In the root of the index key only first and third bits are set to `1`, which means that only first and third inner nodes at the level 2 are non-empty. Notice that the index consists of only 4 index key leaf nodes, 2 inner index key nodes at the level 2 and a single index key root. Empty nodes (marked with a dotted line) are not stored anywhere in the index and are shown in the figure for explanation purposes only.

All index keys combined form the hierarchical bitmap index. Index keys are divided into groups based on the number of non-empty nodes (both inner nodes and index key leaves). Only non-empty nodes are physically stored in the index. All index keys with equal number of non-empty nodes are stored in a single file of fixed-size records. This greatly simplifies the management and maintenance of the index. Index key roots with pointers to the remaining parts of the index keys are stored in the signature tree. The leaves of the signature tree contain the index key roots and the pointers to the remaining parts of each index key, while the internal nodes contain the descriptions of the referenced leaves. Each leaf of the signature tree is represented by the superposition of all index key roots contained in that leaf. Additionally,

signature tree leaves are connected via pointers to form a linked list which enables a linear scan of all index key roots of the hierarchical bitmap index.

5. Approximate queries using hierarchical bitmap index

The result of an approximate query is the set of all tuples which are similar to the user's query q provided some similarity measure $sim(t_i,q)$, i.e, $\{t_i \in R \mid sim(t_i,q) \geq \alpha\}$, α is the similarity threshold provided by the user. In this discussion we used the simple similarity measure of Jaccard's coefficient.

Let q denote a finite set of elements drawn from the domain dom(A) of the set-valued attribute. We will further refer to q as to the user's query. Given a relation $R=\{t_1,t_2,\ldots,t_n\}$. Each tuple t_i contains a set. Assume that there is a hierarchical bitmap index defined on the relation R. Let $K(t_i)$ denote the index key for the tuple t_i. Let $N_n^m(t_i)$ denote the n^{th} node at the m^{th} level of the index key of t_i. Let & denote the bitwise AND operation. The following algorithm is used to perform approximate search using the provided similarity threshold α.

```
for all K(ti) from the S-tree leaves /*recall that all index key roots are stored in an S-tree*/
        c=0;
        for all levels l
                for all index key nodes n at level l in q
                        p=skip(ti,l,n);
                        x=N^l_{p+1}(ti) & N^l_n(q);
                        c=c+count(x);
                end for;
        end for;
        s=count(K(q))+count(K(ti))-c;
        if c/s ≥ α then     return(true);
                else return(false);
        end if;
end for;
```

The main idea of the algorithm is to compare all pairs of corresponding nodes and count the number of positions on which both nodes contain `1's. If the ratio of common `1's to the number of `1's in compared sets is higher than the user defined threshold α then the tuple is added to the answer, else the tuple is rejected. For every tuple t_i the algorithm iterates over all nodes of q and bitwisely ANDs those nodes with corresponding nodes in $K(t_i)$.

Determining the corresponding node to be compared with a given node of $K(q)$ is difficult and is performed by the function $skip(t_i,l,n)$. Both compared index keys may contain different number of nodes as there are nodes in $K(t_i)$ which represent elements in t_i that are not relevant to the query q. So, for every node $N_n^l(q)$ from the index key $K(q)$ the function $skip(t_i,l,n)$ computes the number of nodes that have to be skipped in $K(t_i)$ in order to reach the node which corresponds to $N_n^l(q)$. This computation is performed in the parent node of the $N_n^l(a_i)$, which is the node $N_{n\%d+1}^{l-1}(t_i)$ (where % denotes the modulo operator). The number of nodes that must be skipped at the l^{th} level of the index key $K(t_i)$ is equal to the number of bits in $N_{n\%d+1}^{l-1}(t_i)$ set to `1' and preceding the position n%d+1 (these bits represent nodes at the level l which are not relevant to the query q).

The function $count(x)$ computes the number of bits set to `1' in $x$. When applied to an index key the function $count(K(t_i))$ returns the number bits set to `1' in the base bitmap B of the

index key. After the comparison of all node pairs is finished the ratio of common positions is calculated. If this ratio exceeds the user defined similarity threshold α the algorithm adds the given tuple to the result. The hierarchical bitmap index can be easily adopted to other similarity measures. The key feature of the index, which is the unique and unambiguous representation of the elements of the indexed sets, makes it suitable for other similarity measures as well.

6. Sequential hierarchical bitmap index and approximate queries on sequences

In this section we present a modification of the hierarchical bitmap index that allows to efficiently index sequences and store the time gaps between the consecutive elements of the indexed sequences. Modified structure is similar to the original hierarchical bitmap index and varies only in the representation of the leaf nodes of the index key. Recall that in case of original hierarchical bitmap index each index key leaf contains a part of the base bitmap B (usually every leaf consists of one machine word). With sequences it is also necessary to store not only the elements of the sequence, but the occurrence time of each event as well. In the sequential hierarchical bitmap index each index key leaf is a one page of memory. The first word on the page is the part of the base bitmap B. The rest of each page is filled with time occurrences of every event indexed by the current index key leaf. Above the leaf level all inner nodes of the index key are identical to the original hierarchical bitmap index framework, namely, they contain bit descriptions of lower leaves with a bit set to `1' to signify a non-empty index key leaf page. Sequential hierarchical bitmap index can be successfully used to process subsequence and supersequence queries analogously to the hierarchical bitmap index. In this paper we concentrate on efficient processing of approximate queries using sequential hierarchical bitmap index.

The structure of the index results in a two-phase search algorithm. First, the index is scanned to find the index keys which contain a sufficient number of common elements with the searched sequence provided by a user. This is done exactly in the same way as in case of traditional hierarchical bitmap index. Next, the index key leaf pages are scanned to compute the order and period similarity between the compared sequences. The order of two events e_i,e_j in a sequence S can be determined easily on the basis of their time occurrences, i.e.,

$$e_i \rightarrow_S e_j \text{ iff } t_S(e_j)-t_S(e_i)>0.$$

The result of an approximate query is the set of all sequences which are similar to the user's query *q* provided some similarity measure $sim(s_i,q)$, i.e, $\{s_i \in R \mid sim(s_i,q) \geq \alpha\}$, α is the similarity threshold provided by the user. In this discussion we used our combined measure of element, order, and period similarity.

Let *q* denote a finite sequence of events. We will further refer to *q* as to the user's query. Given a relation $R=\{s_1,s_2,\ldots,s_n\}$. Each sequence s_i contains a sequence. Let $t_S(e_i)$ denote the occurrence time of an event e_i in the sequence S. Assume that there is a sequential hierarchical bitmap index defined on the relation R. Let $K(s_i)$ denote the index key for the sequence s_i. Let $N_n^m(s_i)$ denote the n^{th} node at the m^{th} level of the index key of s_i. Let & denote the bitwise AND operation. The following algorithm is used to perform approximate search using the provided similarity threshold α.

```
for all K(s_i) from the S-tree leaves
        c=0; d=0; e=0;
        for all levels l
                for all index key nodes n at level l in q
                        p=skip(s_i,l,n);
                        x=N^l_{p+1}(si) & N^l_n(q);
                        c=c+count(x);
                        d=d+number of positions p in x where t_q(p)<t_si(p) / count(x);
                        e=e+SumDist(x, N^l_n(q), N^l_{p+1}(s_i));
                end for;
        end for;
        s=count(K(q))+count(K(s_i))-c;
        sim_E=c/s;
        sim_O=d;
        sim_P=e;
        if w_1*sim_E + w_2*sim_E + w_3*sim_E ≥ α
                then    return(true);
                else    return(false);
        end if;
end for;
```

The main idea of the algorithm is the following. The signature tree containing the sequential hierarchical bitmap index is scanned and for every index key the element similarity between the index key and the searched sequence is computed. While scanning the index key leaf pages the algorithm computes in parallel the order similarity (the number of common elements appearing in both keys in the same order) and the period similarity. The latter is computed by the function *SumDist*(x, $N^l_n(q)$, $N^l_{p+1}(s_i)$) which analyzes all positions in the common part x and computes the difference of the distances in s_i and q between all elements represented by x. This algorithm is currently being implemented and tested against previously proposed solutions.

7. Experiments

The experiments where conducted on top of the Athlon 1,4GHz PC with 512MB of memory. Data sets were created using DBGen from the IBM Quest Project. The parameters of synthetic data sets were chosen to imitate the real data sets occurring in retail databases. The number of distinct elements varies from 1000 to 200'000 elements, the number of indexed sets varies from 1000 to 1 milion sets, the average set size varies from 10 elements to 50 elements. We measured also the influence of data distribution on the hierarchical bitmap index. The data correlation was simulated by varying the number of frequent itemsets in the source data, this number changes from 5000 to 100'000 patterns (for 200'000 sets and 100'000 different products). The queries were generated based on the patterns appearing in a given data set. All measurements are given in processor ticks. Currently only the results for approximate queries on sets are available, the experiments on approximate queries on sequences are being conducted.

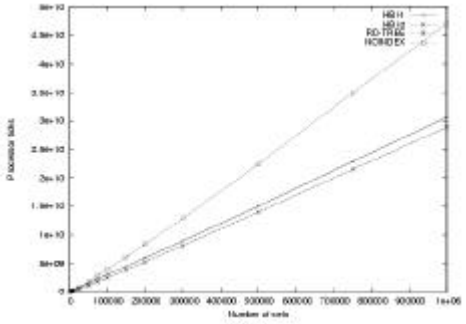

Figure 2

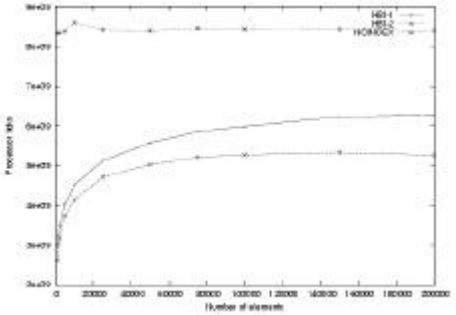

Figure 3

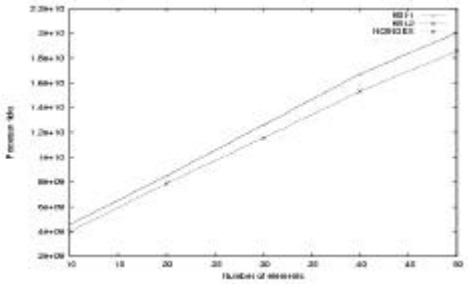

Figure 4

Figure 2 presents the search times for approximate queries on sets with regard to the number of indexed sets varying from 1000 to 1 milion. It can be easily seen that the hierarchical bitmap index performs better than the brute force approach (the full table scan) and that the performance is linear with regard to the number of indexed sets. In the next experiment (Figure 6) we varied the size of the indexed domain from 1000 different elements to 200 000 different elements. Again, the hierarchical bitmap index is beneficial in these circumstances. An interesting feature is the saturation of the index, starting from some border value the search time doesn't depend anymore on the number of different elements (this happens when elements become rare due to their number). Finally, Figure 4 presents the search times for approximate queries when the average size of the indexed sets changes from 10 elements to 50 elements.

8. Conclusions

Approximate query processing on sets and sequences is very important from the point of view of modern applications. However, the support for such functionality in commercially available database systems is very limited. In this paper we argued that new indexing techniques must be developed and new algorithms implemented in order to allow advanced set and sequence querying in relational database systems. We also revised a suitable solution, the hierarchical bitmap index. It is very efficient at processing different classes of set-oriented queries, in particular, it is capable of efficient approximate querying on sets with respect to different similarity measures. We also proposed a modification of the hierarchical bitmap index in order to allow it to efficiently index sequence data. We proposed a novel similarity measure for sequences and showed how it can be employed using the sequential hierarchical bitmap index.

Our future work agenda includes, among others, experimental evaluation of the newly proposed similarity measure for sequences, experimental comparison of the sequential hierarchical bitmap index with SEQ family of indexes [6], and using adjacency matrices with labeled distances between elements to index sequences.

9. Bibliography

[1] U. Deppisch, *S-tree: a dynamic balanced signature index for office retrieval*, in Proceedings of the 9th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, pages 77–87, Pisa, Italy, 1986. ACM.

[2] A. Gionis, D. Gunopulos, N. Koudas, *Efficient and tunable similar set retrieval*, in Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data. ACM Press, Jun 2001.

[3] J. M. Hellerstein, A. Pfeffer, *The RD-Tree: An index structure for sets*, Technical Report 1252, University of Wisconsin at Madison, 1994.

[4] S. Helmer, G. Moerkotte, *A study of four index structures for set-valued attributes of low cardinality*, Technical Report 2/99, Universit¨at Mannheim, 1999.

[5] H. Mannila, P. Ronkainen, *Similarity of event sequences*, in Proceedings of the 4th International Workshop on Temporal Representation and Reasoning TIME'97, May 10-11, 1997, Daytona Beach, Florida, USA

[6] Y. Manolopoulos, M. Morzy, T. Morzy, A. Nanopoulos, M. Wojciechowski, M. Zakrzewicz, *Indexing Techniques for Web Access Logs*, in Web Information Systems, Idea Group Publishing, 2003, (to appear)

[7] S. Melnik, H. Garcia-Molina, *Adaptive algorithms for set containment joins*, ACM Transactions on Database Systems (TODS) Volume 28 , Issue 1 (March 2003): 56 - 99 ACM Press New York, USA

[8] M. Morzy, T. Morzy, A. Nanopoulos, Y. Manolopoulos, *Hierarchical Bitmap Index: an Efficient and Scalable Indexing Technique for Set-Valued Attributes*, in Proceedings of the 7th East-European Conference on Advances in Databases and Informations Systems ADBIS 2003, Dresden, Germany (to appear)

[9] T. Morzy, M. Zakrzewicz, *Group bitmap index: A structure for association rules retrieval*, in Proceedings of the 4th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pages 284–288, New York, USA, Aug 1998. ACM Press.

[10] A. Nanopoulos, Y. Manolopoulos, *Efficient similarity search for market basket data*, VLDB Journal, 11(2):138–152, 2002.

[11] J. Zobel, A. Moffat, R. Sacks-Davis, *An efficient indexing technique for full text databases*, in Proceedings of the 18th International Conference on Very Large Data Bases, August 23-27, 1992, Vancouver, Canada, pages 352–362. Morgan Kaufmann, 1992.