Index-Based Processing of Similarity Queries for Set and Sequence-Valued Attributes

Bogdan Czejdo[1], Zbyszko Królikowski[2], Mikołaj Morzy[2], and Tadeusz Morzy[2]

[1] Department of Mathematics and Computer Science
Loyola University, 6363 St. Charles Avenue
New Orleans, LA 70118
czejdo@loyno.edu
[2] Institute of Computing Science
Poznań University of Technology, Piotrowo 3A
60-965 Poznań, Poland
{zkrolikowski,mmorzy,tmorzy}@cs.put.poznan.pl

Abstract. Many complex real world objects can be easily modeled using sets or sequences. Attributes containing sets or sequences of elements appear in various application domains, e.g. in telecommunication and retail databases, multimedia systems, web server logs, genetic and molecular databases, etc. However, the support for such attributes is usually limited to definition and storage in flat relational tables. Currently available database systems support neither indexing nor advanced querying of attributes containing sets or sequences. SQL language does not offer any primitives to express set containment or set similarity queries.
In this paper we investigate similarity queries for set and sequence-valued attributes. We present the notion of a similarity query and we review similarity measures proposed so far for sets and sequences. We present hierarchical bitmap index, an efficient indexing technique for sets, and we show how the hierarchical bitmap index framework can be extended to incorporate sequences as well. We introduce a new similarity measure that can be successfully used with sequences. We present algorithms for efficient similarity query processing using hierarchical bitmap index and its variants. Our paper concludes with the results of conducted experiments.

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 a 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. They can represent e.g. 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 in relational databases. 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 with set containment operators has been long acknowledged, no implementations followed. To the best of our knowledge sequence-valued attributes are currently not supported by any general-purpose commercial database management system (of course, dedicated database management systems exist that aim at processing of solely sequence data; examples of such dedicated systems include Gen Bank, EMBL, PROSITE, among others).

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 (similar four classes of sequence-oriented queries can be formulated with respect to sequence-valued attributes). Given a query set provided by a user. *Equality queries* search for all tuples that are identical with the query set. Equality queries are useful in accurate medical diagnosis, e.g., to find patients with exactly defined symptoms. The second class of set-oriented queries are *subset queries*. These queries search for all tuples that entirely contain the query set. Subset queries can be used to create target groups of customers and to find customers who bought specific set of products and could be targeted for some promotional offer. The third class of set-oriented queries are *superset queries*, which find all tuples that are entirely contained in the query set. Let us assume that the query 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 *similarity queries*. Similarity queries search for all tuples that are sufficiently similar to the query set, according to some similarity measure.

Similarity 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 recommendations that are of medium similarity 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 of similarity queries may require finding highly similar tuples, e.g. airport security system should identify suspicious individuals based on the images from the surveillance cameras. On the other hand, some applications may depend on efficient querying for strongly dissimilar tuples, e.g. to quickly discover fraud credit card usages.

1.1 Organization of the Paper

This paper is organized as follows. In Section 2 we review the solutions proposed so far in the literature. Section 3 contains basic definitions. In this section we

introduce a new similarity measure for sequences and discuss its usability. We present a hierarchical bitmap index in Section 4 and we show how the hierarchical bitmap index can be used to efficiently process similarity queries on sets in Section 5. We show how to extend the hierarchical bitmap index to enable efficient indexing and similarity-based querying of sequences of elements in Section 6. Section 7 contains results of experiments conducted on the hierarchical bitmap index and its sequential variations. We conclude in Section 8 with a summary and a future research agenda.

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 [8]. In [5] the authors proposed to process similarity queries on sets by transforming sets into vectors in Hamming space and to 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 [15], signature trees [3], hierarchical bitmap indexes [9], and signature tables [5]. For an overview of similarity measures for set proposed so far see [10].

Indexing of sequences has been much researched and resulted in several proposals. In [1] authors present the F-Index and use Discrete Fourier Transform to convert sequences into the frequency domain and manage them as points in a multidimensional space using an R*-tree. This technique can also be extended to allow for similar sequence search [4]. Another technique for answering sequence similarity queries using R-trees was presented in [12]. Most approaches rely on the edit distance between compared sequences [2, 6], although other approaches are also possible, e.g., using time warping distance [14]. An example of an index-based approach using time warping function is presented in [11]. Several solutions to the sequence matching problem come from biological databases, where sequence alignment and protein matching problems have been addressed [13].

3 Definitions

Given a database D of tuples, let $D = \{t_1, t_2, \ldots, t_n\}$, where each tuple t_i contains a set. Let q denote a finite set of elements (called the *query set*). The focus of our interest is to efficiently process similarity queries of the form: $\{t_i \in D : \mathrm{sim}(t_i, q) \geq \alpha\}$ 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: $C_D(S_1, S_2) = \frac{2*|S_1 \cap S_2|}{|S_1|+|S_2|}$

– Jaccard's coefficient: $C_J(S_1, S_2) = \frac{|S_1 \cap S_2|}{|S_1 \cup S_2|}$
– overlap: $C_O(S_1, S_2) = \frac{|S_1 \cap S_2|}{\min|S_1|, |S_2|}$
– cosine: $C_{\cos}(S_1, S_2) = \frac{|S_1 \cap S_2|}{\sqrt{|S_1| \times |S_2|}}$

In our research we have 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[3], 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 $D = \{s_1, s_2, \ldots, s_n\}$ where each sequence s_i is an ordered list of event pairs $s_i = (e_1, t_1)(e_2, t_2), \ldots, (e_n, t_n)$ 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 *query sequence*). The focus of our interest is to efficiently process similarity queries of the form: $Q = \{s_i \in D : sim(s_i, q) \geq \alpha\}$ for some similarity measure $sim()$ and similarity threshold α. In our experiments we have adopted the following similarity function. Let $t_S(e_i)$ denote the time of occurrence of the event e_i in sequence S. Given sequences P,Q, let

– $P \cup Q = \{e_i \mid e_i \in P \ \vee \ e_i \in Q\}$
– $P \cap Q = \{e_i \mid e_i \in P \ \wedge \ e_i \in Q\}$
– $P \widehat{\cap} Q = \{(e_i, e_j) \mid e_i \in P \ \wedge \ e_j \in P \wedge \ e_i \in Q \wedge \ e_j \in Q\}$
– $P \widetilde{\cap} Q = \{(e_i, e_j) \mid e_i \in P \ \wedge \ e_j \in P \wedge \ e_i \in Q \wedge \ e_j \in Q \wedge$
$t_P(e_i) \leq t_P(e_j) \wedge \ t_Q(e_i) \leq t_Q(e_j)\}$

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

$$\text{element similarity: } \text{sim}_E = \frac{|P \cap Q|}{|P \cup Q|} \tag{1}$$

$$\text{order similarity: } \text{sim}_O = \frac{\left|P \widehat{\cup} Q\right|}{\left|P \widetilde{\cup} Q\right|} \tag{2}$$

$$\text{period similarity: } \text{sim}_P = \frac{\sum \frac{|t_P(e_i) - t_P(e_j)|}{|t_Q(e_i) - t_Q(e_j)|}}{\left|P \widetilde{\cup} Q\right|} \text{ for all } (e_i, e_j) \in P \widetilde{\cup} Q \tag{3}$$

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

$$\text{sim}(P, Q) = w_1 * \text{sim}_E(P, Q) + w_2 * \text{sim}_O(P, Q) + w_3 * \text{sim}_P(P, Q)$$

[3] recall that the metric is a distance function $d(x, y)$ such that $d(x, x) = 0$, $d(x, y) = d(y, x)$ and $d(x, z) \leq d(x, y) + d(y, z)$

where $w_1 + w_2 + w_3 = 1$.

The element similarity factor (1) measures the pure set similarity and it does not consider the order and time constraints of events in P and Q. The order similarity factor (2) 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 (3) computes the relative difference of time gaps between all pairs of events (e_i, e_j) occurring in both sequences. Although the above presented function is not a measure (because period similarity is not symmetrical), this intuitive function captures well the similarity between sequences.

4 Hierarchical Bitmap Index

Hierarchical bitmap index (HBI) originates from the well-known S-tree structure. The main difference is the way indexed sets are represented in index keys. To remove the ambiguity of set representation 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 domain of indexed set $|\mathrm{dom}\,(A)|$ and the length of the machine word l. The bitmap B must be long enough to map every element $a_i \in \mathrm{dom}\,(A)$ to a distinct bit, i.e. the length of the bitmap $b = l * \left\lceil \frac{|\mathrm{dom}(A)|}{l} \right\rceil$. The bitmap B is then divided into $m = \frac{b}{l}$ nodes, called *index key leaves*. Every element in the domain of the indexed set $a_i \in \mathrm{dom}\,(A)$ is mapped via a mapping function $f\,(a_i)$ to a kth position in the bitmap B, $k \in \langle 1, b \rangle$.

Given an 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 sets the ith bit of the bitmap B to '1'. This bit is in fact the jth bit in the kth index key leaf, where $k = \lceil \frac{i}{l} \rceil$ and $j = i - \left(\lceil \frac{i}{l} \rceil - 1\right) * l$. 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 equals tens or hundreds of thousands of bits. On the other hand, even for sets with large average size most index key leaves are 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 jth bit in the kth inner node, where $k = \lceil \frac{i}{l} \rceil$ and $j = i - \left(\lceil \frac{i}{l} \rceil - 1\right) * l$. 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 and the depth of the index key d. Shallow index keys are faster to process, but they limit the maximum number of distinct elements that can be represented in an 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 does not have to be known in advance, is not relevant to the construction of the index. For example, let us 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 let us consider the following example of a single index key construction.

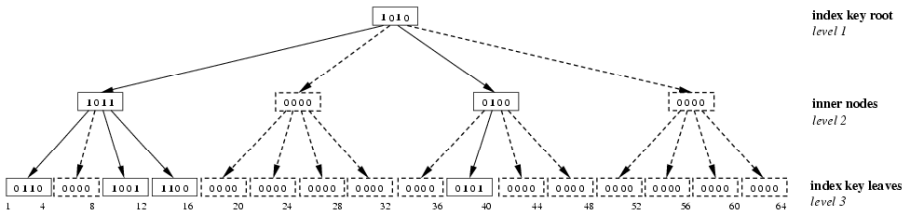

Fig. 1. Hierarchical bitmap index

Example 1. Let us assume index key node length $l = 4$ and index depth $d = 3$. Let us also assume 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 purpose 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 index keys are stored in the signature tree. The leaves of the signature tree contain index key roots and 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 Similarity Queries Using Hierarchical Bitmap Index

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

1: **for all** $K(t_i)$ **do**
2: $c = 0$;
3: **for all** levels l **do**
4: **for all** index key nodes n at level l in q **do**
5: $p = \texttt{skip}(t_i, l, n)$;
6: $x = N_{p+1}^l(t_i) \,\&\, N_n^l(q)$;
7: $c = c + \texttt{count}(x)$;
8: **end for**
9: **end for**
10: $s = \texttt{count}(K(q)) + \texttt{count}(K(t_i)) - c$;
11: **if** $\frac{c}{s} \geq \alpha$ **then**
12: **return**(true);
13: **else**
14: **return**(false);
15: **end if**
16: **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 $K(q)$ and bitwisely `AND`s 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 $\texttt{skip}(t_i, l, n)$. 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 $\texttt{skip}(t_i, l, n)$ 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 $N_n^l(t_i)$, 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 lth 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 $\texttt{count}(x)$ computes the number of bits set to '1' in x. When applied to an index key the function $\texttt{count}(K(t_i))$ returns the number bits set to '1' in the base bitmap B of the index key $K(t_i)$. 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. Notice that HBI 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 Sequence-Oriented Indexes and Similarity Queries on Sequences

In this section we present two modifications of the hierarchical bitmap index that allow to efficiently index sequences and store time gaps between consecutive elements of the indexed sequences.

6.1 Sequential Hierarchical Bitmap Index

The first structure is similar to the original hierarchical bitmap index and varies only in the representation of the leaf nodes of the index key. We will refer to it as the *sequential hierarchical bitmap index* (SHBI). Recall that in case of the original hierarchical bitmap index each index key leaf contains a part of the base bitmap B. With sequences it is also necessary to store the occurrence time of each event as well. In the sequential hierarchical bitmap index each index key leaf occupies 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. The sequential hierarchical bitmap index can be successfully used to process subsequence, supersequence, and similarity sequence queries.

The structure of SHBI 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 query sequence provided by a user. This is done exactly in the same way as in case of the traditional hierarchical bitmap index.

Next, 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 the time occurrences of their elements, i.e., $e_i \xrightarrow{S} e_j$ iff $t_S(e_j) - t_S(e_i) > 0$.

Let q denote a finite sequence of events. We will further refer to q as the *query sequence*. Given a database of sequences $D = \{s_1, s_2, \ldots, s_n\}$. Let $t_{s_i}(e_i)$ denote the occurrence time of an event e_i in the sequence s_i. Let us assume that there is a sequential hierarchical bitmap index defined on the database D. Let $K(s_i)$ denote the index key for the sequence s_i. Let $N_n^m(s_i)$ denote the nth node at the mth level of the index key of s_i. Let & denote the bitwise `AND` operation. The following algorithm is used to perform similarity search using the provided similarity threshold α.

1: **for all** $K(s_i)$ **do**
2: $c = 0; d = 0; e = 0;$
3: **for all** levels l **do**
4: **for all** index key nodes n at level l in q **do**
5: $p = \texttt{skip}(s_i, l, n)$;
6: $x = N_{p+1}^l(s_i) \& N_n^l(q)$;
7: $c = c + \texttt{count}(x)$;
8: $d = d +$ number of positions p in x where $t_q(p) < t_{s_i}(p) / \texttt{count}(x)$
9: $e = e + \texttt{SumDist}(x, N_{p+1}^l(s_i), N_n^l(q));$
10: **end for**
11: **end for**
12: $s = \texttt{count}(K(q)) + \texttt{count}(K(s_i)) - c;$
13: $\text{sim}_E = c/s; \text{sim}_O = d; \text{sim}_P = e;$
14: **if** $w_1 * \text{sim}_E + w_2 * \text{sim}_O + w_3 * \text{sim}_P \geq \alpha$ **then**
15: **return**(true);
16: **else**
17: **return**(false);
18: **end if**
19: **end for**

The main idea of the algorithm is the following. For every index key the element similarity between the index key and the query sequence is computed. While scanning the index key leaf pages the algorithm computes in parallel the order similarity and the period similarity. The latter is computed by the function $\texttt{SumDist}\left(x, N_{p+1}^l(s_i), N_n^l(q)\right)$ 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.

6.2 Two-Dimensional Hierarchical Bitmap Index

The second structure that can be derived from the hierarchical bitmap index is a *two-dimensional hierarchical bitmap index* (HBI2). The idea of the hierarchical compression of a sequence representation remains the same, while the meaning of the compressed bitmap changes. In HBI, the underlying bit vector contains bits set to '1' at positions representing sequence elements. In HBI2, the underlying structure is a $n \times n$ matrix M, where $n = |\text{dom}(S)|$ and S is the attribute

containing sequences. For a base matrix representing a sequence s_k, an element $M[i,j]$ is set to '1' iff $e_i \overset{s_k}{\rightarrow} e_j$, i.e., if an event e_i precedes an event e_j in the sequence s_k. Each base matrix M is divided into rectangles of the size $x \times y$, where $x * y \leq$ machine word. Each of those rectangles becomes an index key leaf at the lowest level of the HBI2 tree. Upper levels of the HBI2 tree are created analogously to the HBI framework, so only non-empty rectangles of the lower levels are represented at higher levels. This representation of an indexed sequence is more coherent and concise than that of SHBI, but requires usually more space, especially in case of long sequences, because the number of bits set to '1' in the base matrix M grows quadratically with the length of the sequence. The main drawback of HBI2 is the fact, that it does not allow to encode time gaps between consecutive elements of a sequence and it does not allow multiple occurrences of an element within a sequence. To overcome those obstacles we have developed an extension of HBI2, a *three-dimensional hierarchical bitmap index* (HBI3) that contains an additional dimension of time. We are currently investigating the properties of HBI3 and conducting experiments with this novel indexing structure.

HBI2 can be used to process similarity queries on sequences. The algorithm is very similar to the algorithm presented in Section 5 for HBI. Due to the lack of space we will skip minor differences between the two. For the same reason we have to resign from presenting the implementation details of all four indexes mentioned in this paper. In the next section we present the results of the experimental evaluation of HBI, SHBI, and HBI2 indexes of their ability to efficiently process similarity queries on sets and sequences.

7 Experiments

The experiments where conducted on top of the Athlon 1,4 GHz PC with 512 MB 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 million 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. For similarity queries on sequences we used a database of 1000 sequences with the domain size of 10 000 different elements and the sequence length varying from 3 to 25 elements on average.

Figure 2 presents the search times for similarity queries on sets with regard to the number of indexed sets varying from 1000 to 1 million. It can be easily noticed that HBI 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 3) we have varied the size of the indexed domain from 1000 to 200 000 different elements. Again, HBI is superior in these circumstances. An interesting feature is the saturation of the index, starting from some threshold value the search time does not depend anymore on the number of different elements (this happens when adding new elements to the domain does not influence the average number of non-empty HBI key nodes in HBI keys). Figure 4 presents the search times for similarity queries when the average size of indexed sets changes from 10 elements to 50 elements. Finally, Figure 5 displays the search times for similarity queries on sequences. In this experiment we have compared SHBI and HBI2 with SEQC [7].

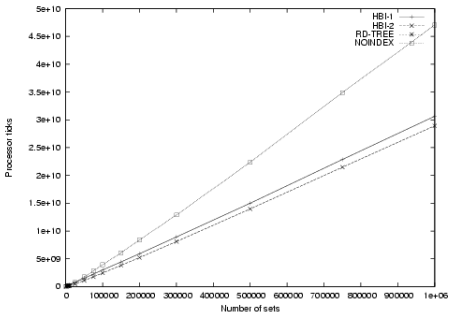

Fig. 2. Number of sets

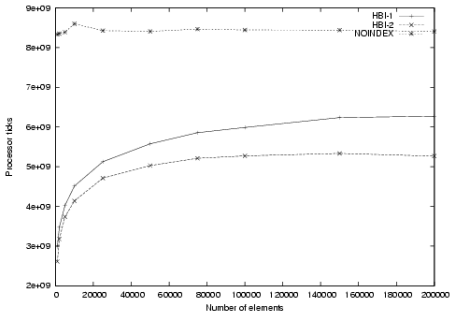

Fig. 3. Size of the domain

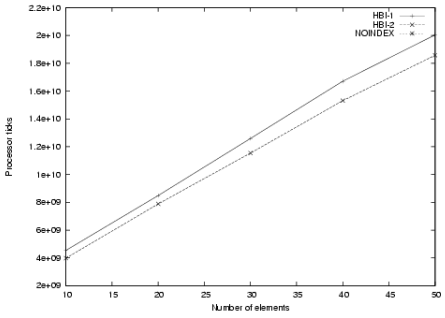

Fig. 4. Average size of a set

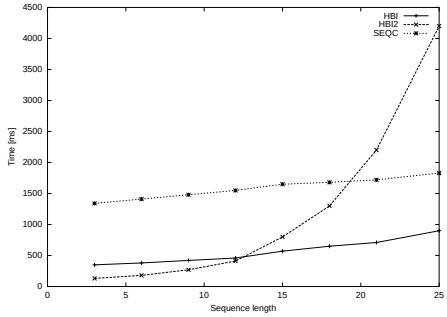

Fig. 5. Sequence similarity

8 Conclusions

Similarity 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 similarity querying on sets using different similarity measures. We also proposed a modification of HBI in order to allow it to efficiently index sequence data.

Our future work agenda includes, among others, experimental evaluation of the newly proposed similarity function for sequences, experimental comparison of the sequential hierarchical bitmap index with the entire family of SEQ indexes [7], and using adjacency matrices with labeled distances between elements to index sequences.

References

1. Agrawal, R., Faloutsos, C., Swami, A.: *Efficient Similarity Search In Sequence Databases*. Proc. of the 4th International Conference on Foundations of Data Organization and Algorithms FODO'93, pages 69–84, Chicago, Illinois, USA, October 13-15, 1993.
2. Bozkaya, T., Yazdani, N., Meral Özsoyoglu, Z.: *Matching and Indexing Sequences of Different Lengths*. Proceedings of the Sixth International Conference on Information and Knowledge Management CIKM'97, pages 128–135, ACM Press, Las Vegas, Nevada, USA, November 10-14, 1997.
3. Deppisch, U.: *S-tree: a dynamic balanced signature index for office retrieval*. Proc. of the 9th Annual International Conference on Research and Development in Information Retrieval ACM SIGIR, pages 77–87, ACM Press, Pisa, Italy, 1986.
4. Faloutsos, C., Ranganathan, M., Manolopoulos, Y.: *Fast Subsequence Matching in Time-Series Databases*. Proc. of the 1994 ACM SIGMOD International Conference on Management of Data pages 419–429, ACM Press, Minneapolis, Minnesota, USA, May 24-27, 1994.
5. Gionis, A., Gunopulos, D., Koudas, N.: *Efficient and tunable similar set retrieval*. Proc. of the 2001 ACM SIGMOD International Conference on Management of Data ACM Press, Santa Barbara, California, USA, May 21-24, 2001.
6. Mannila, H., Ronkainen, P.: *Similarity of event sequences*. Proc. of the 4th International Workshop on Temporal Representation and Reasoning TIME'97, Daytona Beach, Florida, USA, May 10-11, 1997.
7. Manolopoulos, Y., Morzy, M., Morzy, T., Nanopoulos, A., Wojciechowski, M., Zakrzewicz, M.: *Indexing Techniques for Web Access Logs*. Web Information Systems, Idea Group Publishing, 2004.
8. Melnik, S., Garcia-Molina, H.: *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
9. Morzy, M., Morzy, T., Nanopoulos, A., Manolopoulos, Y.: *Hierarchical Bitmap Index: an Efficient and Scalable Indexing Technique for Set-Valued Attributes*. Proc. of the 7th East-European Conference on Advances in Databases and Informations Systems ADBIS'2003, pages 236–252, Dresden, Germany, September 3-6, 2003.
10. Nanopoulos, A., Manolopoulos, Y.: *Efficient similarity search for market basket data*. VLDB Journal, 11(2):138-152, 2002.
11. Park, S., Chu, W., Yoon, J., Hsu, C.: *Efficient Searches for Similar Subsequences of Different Lengths in Sequence Databases*. Proc. of the 16th International Con-

ference on Data Engineering ICDE'00, pages 23–32, IEEE Computer Society, San Diego, California, USA, 28 February–3 March, 2000.
12. Rafiei, D., Mendelzon, A.O.: *Similarity-Based Queries for Time Series Data*. Proc. of the 1997 ACM SIGMOD International Conference on Management of Data pages 13–25, ACM Pess, Tucson, Arizona, USA, May 13-15, 1997.
13. Tsong-Li Wang, J., Chirn, G-W., Marr, T., Shapiro, B., Shasha, D., Zhang, K.: *Combinatorial Pattern Discovery for Scientific Data: Some Preliminary Results*. Proc. of the 1994 ACM SIGMOD International Conference on Management of Data pages 115–125, ACM Press, Minneapolis, Minnesota, USA, May 24-27, 1994.
14. Yi, B. K., Jagadish, H. V., Faloutsos, C.: *Efficient Retrieval of Similar Time Sequences Under Time Warping*. Proc. of the Fourteenth International Conference on Data Engineering ICDE'98, pages 201–208, IEEE Computer Society, Orlando, Florida, USA, February 23-27, 1998.
15. Zobel, J., Moffat, A., Sacks-Davis, R.: *An efficient indexing technique for full text databases*. Proc. of the 18th International Conference on Very Large Data Bases VLDB'92, pages 352–362, Morgan Kaufmann, Vancouver, Canada, August 23-27, 1992.