Hierarchical Bitmap Index: An Efficient and Scalable Indexing Technique for Set-Valued Attributes*

Mikołaj Morzy[1], Tadeusz Morzy[1], Alexandros Nanopoulos[2], and Yannis Manolopoulos[2]

[1] Institute of Computing Science
Poznań University of Technology, Piotrowo 3A, 60-965
Poznań, Poland
{Mikolaj.Morzy,Tadeusz.Morzy}@cs.put.poznan.pl
[2] Department of Informatics, Aristotle University of Thessaloniki
Thessaloniki, Greece
{alex,manolopo}@delab.csd.auth.gr

Abstract. Set-valued attributes are convenient to model complex objects occurring in the real world. Currently available database systems support the storage of set-valued attributes in relational tables but contain no primitives to query them efficiently. Queries involving set-valued attributes either perform full scans of the source data or make multiple passes over single-value indexes to reduce the number of retrieved tuples. Existing techniques for indexing set-valued attributes (e.g., inverted files, signature indexes or RD-trees) are not efficient enough to support fast access of set-valued data in very large databases.
In this paper we present the hierarchical bitmap index—a novel technique for indexing set-valued attributes. Our index permits to index sets of arbitrary length and its performance is not affected by the size of the indexed domain. The hierarchical bitmap index efficiently supports different classes of queries, including subset, superset and similarity queries. Our experiments show that the hierarchical bitmap index outperforms other set indexing techniques significantly.

1 Introduction

Many complex real world objects can be easily modeled using set-valued attributes. Such attributes appear in several application domains, e.g., in retail databases they can represent the set of products purchased by a customer, in multimedia databases they can be used to represent the set of objects contained in an image, in web server logs they correspond to web pages and links visited by a user. Finally, in data mining applications set-valued attributes are commonly used to store time-series and market basket data. Contemporary database systems provide the means to store set-valued attributes in the database (e.g., as

* Work supported by a Bilateral Greek-Polish Program

L. Kalinichenko et al. (Eds.): ADBIS 2003, LNCS 2798, pp. 236–252, 2003.
© Springer-Verlag Berlin Heidelberg 2003

contents of nested tables or values of user-defined types), but they don't provide either language primitives or indexes to process and query such attributes.

The inclusion of set-valued attributes in the SQL3 standard will definitely result in wide use of such attributes in various applications. Web-based interfaces to database systems open new possibilities but require prompt advances in query processing on sets. Examples of advanced applications making heavy use of set-valued attributes are, among others, automatic recommendations and discount offers, collaborative filtering, or web site personalization.

Another domain that would greatly profit from the possibility to perform advanced querying on sets is data mining. Several knowledge discovery techniques rely on excessive set processing. Shifting these computations directly to the database engine would result in considerable time savings. This can be observed especially in case of the apriori family of algorithms which perform huge number of subset searches. In the apriori framework the frequency of different sets of products is determined during repeated scans of the entire database of customer transactions. In each pass every transaction is tested for the containment of the so-called candidate sets, the sets of products that the algorithm suspects to appear frequently enough in the database. These repetitive tests are in fact subset queries. Presently, due to the lack of support from the commercial database management systems, these tests are conducted on the application side and not in the database.

Developing efficient mechanisms for set indexing and searching can serve as the basis for other improvements. The ability to retrieve sets based on their subset properties can lead to the improvement of many other algorithms that depend on database set processing. As an example let us consider a system for automated web site personalization and recommendation. Such systems are commonly used in various e-commerce sites and on-line stores. The system keeps information about all products purchased by the customers. Based on these data the system discovers patterns of customer behaviour, e.g., in the form of characteristic sets of products that are frequently purchased together. When a new potential customer browses the on-line catalog of available products, the system can anticipate which product groups are interesting to a customer by analyzing search criteria issued by a customer, checking visited pages and inspecting the history of purchases made by that customer, if the history is available. By issuing subset queries to find patterns relevant to the given customer the system can not only dynamically propose interesting products, preferably at a discount, but it can also tailor the web site navigation structure to satisfy specific customers requirements.

Let us assume that the customer started from the main page of an on-line store and issued a search with a key word "palm". Furthermore, let us assume that the identification of the customer is possible, e.g., by the means of a cookie or a registered username. The system retrieves then the history of all purchases made by that specific customer. In the next step, the system uses a subset query to find all patterns pertaining to palm devices. Another query is required to find those patterns that are eligible for the given customer. This is done by searching for palm related patterns that are subsets of the history of customer

purchases. Those patterns are used to predict future purchases of the customer. On the other hand, the system could also store web site navigation patterns discovered from web server logs. Such patterns tend to predict future navigation paths based on previously visited pages. The system could use a subset query to find those patterns that are applicable to the customer and determine that this specific customer is interested mainly in computers. The system could hide the uninteresting departments of the on-line store by shifting links to those departments to less accesible parts of the page and moving forward links to palm related products (software, accessories, etc.) to ease navigation. This scenario, which could be used in e-commerce sites, assumes that the subset searches are performed very efficiently and transparently to the customer. This assumption represents the fact that the entire processing must finish within a very short time-window in order not to let the customer become annoyed with the web site response time and not to let him move to another on-line store.

Unfortunately, currently available database systems do not provide mechanisms to achieve satisfactory response times for subset queries and database sizes in question. The ability to efficiently perform set-oriented queries in large data volumes could greatly enhance web-based applications. This functionality is impatiently anticipated by many potential users.

The most common class of queries which often appear in terms of set-valued attributes are subset queries that look for sets that entirely contain a given subset of elements. Depending on the domain of use subset queries can be used to find customers who bought specific products, to discover users who visited a set of related web pages, to identify emails concerning a given matter based on a set of key words, and so on. In our study and experiments we concentrate on subset queries as they are by far the most common and useful class of queries regarding set-valued attributes. We recognize other classes of queries as well and we describe them later on.

To illustrate the notion of a subset query let us consider a small example. Given a sample database of retail related data consisting of a relation `Purchases(trans_id,cust_id,products)`. An example of a subset query is:

```
SELECT COUNT(DISTINCT cust_id)
FROM Purchases
WHERE products ⊇ {'milk','butter'};
```

This query retrieves the number of customers who purchased products *milk* and *butter* together during a single visit to the supermarket. Such queries require costly set inclusion tests performed on very large data volumes. There could be as many as tens of thousands of customers per day, each purchasing several products. The number of different products in an average supermarket could easily amount to hundreds of thousands. Standard SQL language doesn't contain primitives to formulate such queries. Usually, those queries are written in an awkward and illegible way. For example, the aforementioned subset search query can be expressed in standard SQL in one of the following manners.

```
SELECT COUNT(DISTINCT A.cust_id)
FROM Purchases A, Purchases B
WHERE A.trans_id = B.trans_id
AND A.item = 'milk' AND B.item = 'butter';
```

or

```
SELECT COUNT(*) FROM (
   SELECT trans_id FROM Purchases
   WHERE item IN {'milk','butter'}
   GROUP BY trans_id
   HAVING COUNT(*) = 2 );
```

Both queries are very expensive in terms of the CPU and memory utilization and tend to be time-consuming. It is easy to notice that the number of self joins in the first query is proportional to the size of the searched set. The second query requires grouping of very large table, which may be very costly, too. Additionaly, both queries are cryptic, hard to read, and lack flexibility. Adding another element to the searched set requires in the first case modifying the `FROM` clause and adding two new predicates; in the second case it requires modifying the `WHERE` and `HAVING` clauses.

Typical solution to speed up queries in database systems is to use indexes. Unfortunately, despite the fact that the SQL3 supports set-values attributes and most commercial database management systems offer such attributes, no commercial DBMS provides to date indexes on set-valued attributes. The development of an index for set-valued attributes would be very useful and would improve significantly the performance of all applications which depend on set processing in the database.

Until now, several indexing techniques for set-valued attributes have been proposed. These include, among others, R-trees [10], signature files [4] and S-trees [6], RD-trees [11], hash group bitmap indexes [14], or inverted files [2]. Unfortunately, all those indexes exibit deficiencies that limit their use in real-world applications. Their main weaknesses include the non-unique representation of the indexed sets, the necessity to verify all hits returned from the index in order to prune false hits, and significant losses in performance when the cardinality of the indexed sets grows beyond a certain threshold. Another drawback of the aforementioned methods is the fact that most of them, excluding S-trees, can't be used to speed up queries other than subset queries.

In this paper we present the hierarchical bitmap index. It is a novel structure for indexing sets with arbitrary sized domains. The main advantages of the hierarchical bitmap index are:

- scalability with regard to the size of the indexed domain and to the average size of the indexed set,
- efficient support of different classes of queries, including subset, superset and similarity queries,
- compactness which guarantees that the index consumes the least amount of memory required,
- unique representation of the indexed sets which attains the lack of any false hits and avoids the cost of false hit resolution.

We prove experimentally that the hierarchical bitmap index outperforms other indexes significantly under all circumstances. In the next sections we describe the structure of the index in detail.

Our paper is organized as follows. We begin in Section 2 with the presentation of previously proposed solutions. Section 3 contains the description of the hierarchical bitmap index. Algorithm for performing subset queries using our index is given in Section 4. We present the results of the conducted experiments in Section 5. Section 6 contains the description of other classes of set-oriented queries, namely the superset and similarity queries, that can be efficiently processed using the hierarchical bitmap index. We also provide algorithms to process these classes of queries using the hierarchical bitmap index. Finally, we conclude in Section 7 with a brief summary and the future work agenda.

2 Related Work

Traditional database systems provide several indexing techniques that support single tuple access based on atomic attribute value. Most popular indexes include B+ trees [5], bitmap indexes [3] and R-trees [10]. However, these traditional indexes do not provide mechanisms to efficiently query attributes containing sets. Indexing of set-valued attributes was seldom researched and resulted in few proposals.

First access methods were developed in the area of text retrieval and processing. One of the techniques proposed initially for indexing text documents is inverted file [2]. Inverted file consists of vocabulary and occurrences. Vocabulary is the list of all elements that appear in the indexed sets. Identifiers of all sets containing given element are put on a list associated with that element. All lists combined together form the occurrences. When a user searches for sets containing a given subset, the occurrence lists of all elements belonging to the searched subset are retrieved and joined together to determine the identifiers of sets appearing on all occurrence lists. This technique can be successfully used when indexing words occurring in documents. The number of lists is relatively low (usually only the key words are indexed) and the occurrence lists are short. Unfortunately, in retail databases the size of the vocabulary and the number of occurrence lists are huge. For this reason inverted files are not well suited to index market basket data. Another disadvantage of the inverted file is the fact that this kind of index can't be used to answer superset or similarity queries.

Signature files [4] and signature trees (S-trees) [6] utilize the superimposition of set elements. Each element occurring in the indexed set is represented by a bit vector of the length n with k bits set to '1'. This bit vector is called the *signature* of the element. Signatures are superimposed by a bitwise `OR` operation to create a set signature. All set signatures can be stored in a sequential signature file [7], a signature tree, or using an extendible signature hashing. Interesting performance evaluation of signature indexes can be found in [12]. Much effort is put into further improving signature indexes [17]. Signature indexes are well suited to handle large collections of sets but their performance degrades quickly

with the increase of the domain size of the indexed attribute. Every element occurring in a set must be represented by a unique signature. Large number of different elements leads to either very long signatures or to hashing. Long signatures are awkward to process and consume large amounts of memory. On the other hand, hashing introduces ambiguity of mapping between elements and signatures, increasing the number of false hits. In case of very large databases this could lead to unacceptable performance loss.

Indexing of set-valued attributes was also researched in the domain of object-oriented and spatial database systems [16]. Evaluation of signature files in object-oriented database environment can be found in [13]. Processing of spatial data resulted in the development of numerous index types, among them R-trees [10] and BANG indexes [8]. These indexes can also be used to index non-spatial objects, provided there is a way to map an indexed object to a multidimensional space. In case of market basket data each product appearing in indexed sets must be represented as a distinct dimension with dimension members 0 (product not present in a given basket) and 1 (product present in a given basket). However, the number of possible dimensions involved in indexing set-valued attributes, which equals tens of thousands of dimensions, makes this approach unfeasible. A modification of the R-tree index, called the Russian Doll tree (RD-tree), was presented in [11]. In the RD-tree all indexed sets are stored in tree leaves, while inner nodes contain descriptions of sets from the lower levels of the tree. Every inner node fully describes all its child nodes placed on lower levels. RD-trees can be used only to answer subset queries and their performance drops significantly with the increase of the database size.

Subset search is crucial in many data mining activities. From the knowledge discovery domain originates another proposal, namely the hash group bitmap index [14]. This index is very similar to signature index. Every element appearing in the indexed set is hashed to a position in the bit vector. For large domains there can be several elements hashing to the same position. Set representations are created by the superimposition of bits representing elements contained in the given set. Subset search is performed in two phases. First, in the filtering step, the index is scanned to find matching entries (these are all sets that potentially contain the given subset). Then, in the verification step, all matching entries are tested for the actual containment of the searched set to eliminate any false hits. For very large domains the ambiguity introduced by hashing results in rapid increase of the number of false hits, which directly affects the performance of the index.

3 Hierarchical Bitmap Index

Hierarchical bitmap index is based on signature index framework. It employs the idea of exact set element representation and uses hierarchical structure to compact resulting signature and reduce its sparseness. The index on a given attribute consists of a set of index keys, each representing a single set. Every index key comprises a very long signature divided into n-bit chunks (called *index key*

leaves) and a set of *inner nodes* of the index key organized into a tree structure. The highest inner node is called the *root* of the index key. The signature must be long enough to represent all possible elements appearing in the indexed set (usually hundreds of thousands of bits). Every element o_i of the attribute domain A, $o_i \in dom(A)$ is mapped to an integer i.

Given an indexed set $S = \{o_1, o_2, \ldots, o_n\}$. The set is represented in the index in the following way. Let l denote the length of the index key node. An element $o_i \in S$ is represented by a '1' on the j-th position in the k-th index key leaf, where $k = \lceil i/l \rceil$ and $j = i - (\lceil i/l \rceil - 1) * l$. Therefore, the set S is represented by n '1's set at appropriate positions of the index key leaves. An index key node (either leaf node or inner node) which contains '1' on at least one position is called the *non-empty* node, while an index key node which contains '0' on all positions is called the *empty* node. The next level of the index key compresses the signature representing the set S by storing information only about the non-empty leaf nodes. A single bit in an inner node represents a single index key leaf. If this bit is set to '1' then the corresponding index key leaf contains at least one position set to '1'. The i-th index key leaf is represented by j-th position in the k-th inner index key node, where $k = \lceil i/l \rceil$ and $j = i - (\lceil i/l \rceil - 1) * l$. Every upper level of the inner nodes represents the lower level in an analogous way. This procedure repeats recursively to the index key root. The index key stores only the non-empty nodes. Empty nodes are not stored anywhere in the index key.

The two parameters which affect the shape and the capacity of the index are: l – the length of a single index key node and d – the depth of the index key tree structure. For example, if $d = 4$ and $l = 32$ then the root of the index key contains 32 positions, the second level contains $32^2 = 1024$ positions and the third level contains $32^3 = 32768$ positions that correspond to 32768 index key leaves. This allows storage in a single index key with $d = 4$ and $l = 32$ information about a set that could contain at most $32^4 = 1048576$ different elements. Furthermore, in case this size is not sufficient, extending the index key to five levels would allow indexing sets with the domain of 33554432 different elements.

The presented index has several advantages over previously presented approaches. The most important feature of the hierarchical bitmap is the fact that it supports indexing sets with no loss of accuracy and no ambiguity. Every indexed set is represented in the index uniquely. The index permits representing sets of arbitrary length drawn from a domain of arbitrary size as long as the size of the domain doesn't exceed the maximum cardinality. Note, from the numbers given above, that the index with only four levels allows indexing sets with the domain of 1048576 different elements. This size is sufficient for most domains in practical applications.

Example 1. Assume index key node length $l = 4$ and index depth $d = 3$. Assume also that the elements of the attribute domain have been already mapped to integer values. Given the set $S = \{2, 3, 9, 12, 13, 14, 38, 40\}$. The index key of the set S is depicted in Fig. 1. At the lowest level (level 3) of the index 8 bits

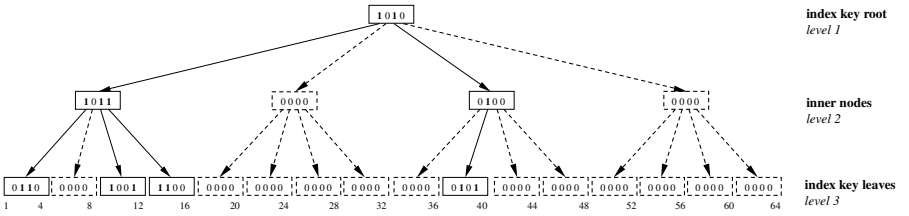

Fig. 1. A single key of the hierarchical bitmap index

corresponding to the given elements of the set S are set to '1'. As a result, index key leaf nodes 1,3,4 and 10 become non-empty (they are marked with a solid line). At the upper level (level 2) 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 to better explain the idea of the hierarchical bitmap index. Note that the index key node size has been set to 4 bits for demonstration purposes only. In real world applications it would be most likely set to one machine word, i.e., to 32 bits.

□

Let us now discuss briefly the physical implementation of the hierarchical bitmap index. An example of the entire index is depicted in Fig. 2. Every index key is stored as a linked list of all index key nodes (both internal and leaf nodes) except the index key root. Those linked lists are stored physically in a set of files, where each file stores all index keys containing an equal number of nodes (e.g., file 1 contains index keys composed of five nodes and file 2 contains index keys composed of 2 nodes). As the result, every file contains only the records of a fixed size. This allows efficient maintenance procedures, e.g., inserting a new index key requires counting the number of nodes in the key and appending a fixed sized record at the end of the appropriate file. To store the index key roots we propose a hybrid S-tree in which all tree leaves are connected with pointers to form a list. In this fashion the index can be efficiently processed by both S-tree traversal in case of subset queries and full scanned in case of superset and similarity queries. The leaves of the S-tree contain all index key roots and, for every index key root, the pointers to the locations where the rest of a given index key is stored. Internal nodes of the S-tree contain the superimposition of all index key roots contained in the lower level nodes that they point to. An interesting discussion on signature trees construction and improvement can be found in [17]. We believe that it is sufficient to store in the S-tree only the roots of the index keys. This allows early pruning of sets that don't contain the searched subset while it keeps the S-tree compact and easy to manage. This architecture makes update operations become more costly because updating a given key might require relocating the

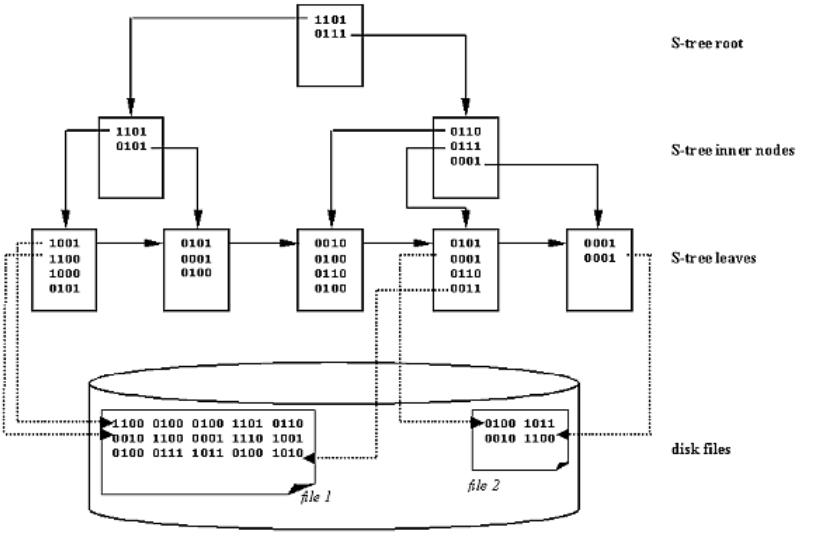

Fig. 2. S-tree with hierarchical bitmap index key roots

key to another file (if the number of nodes changed after the update). On the other hand, we argue that the update operations are not frequent enough to result in a noticeable preformance loss.

4 Search Algorithm

Hierarchical bitmap index can be used to speed up different classes of queries involving set-valued attributes. Given a relation $R = \{t_1, t_2, \ldots, t_n\}$ of tuples containing a set-valued attribute A. Let $\{A_1, A_2, \ldots, A_n\}$ denote the values of the attribute A in subsequent tuples of R, $t_i.A = A_i$. A query predicate is defined by the set-valued attribute A and a set comparison operator $\Theta \in \{\subseteq, \supseteq, \approx\}$. Let q denote a finite set of elements of the attribute A searched by a user. We will further refer to q as the user's query.

Assume the hierarchical bitmap index on the attribute A of the relation R is defined. Let $K(A_i)$ denote the index key of the set A_i. Let $N_n^m(A_i)$ denote the n-th node (either internal or leaf) at the m-th level of the index key of the set A_i. Let $N_1^1(A_i)$ denote the root of the index key of A_i. Let d denote the depth of the index key and & denote the bitwise `AND` operation.

The goal of the search algorithm is to find all tuples $t_i \in R$, such that $t_i.A \supseteq q$, given the search set q. The search algorithm is presented below.

1: **for all** $A_i \in R.A$ from the S-tree such that $N_1^1(A_i) \& N_1^1(q) = N_1^1(q)$ **do**
2: **for all** levels l **do**
3: **for all** internal nodes i at level l in q **do**

```
4:          p = skip(A_i, l, i)
5:          if N^l_{p+1}(A_i)&N^l_i(q) ≠ N^l_i(q) then
6:            return(false)
7:          end if
8:        end for
9:      end for
10:     return(true)
11: end for
```

The algorithm starts with defining the index key on the searched set q. Then, the algorithm traverses the S-tree to find all sets that potentially contain the searched subset. This is determined by comparing index key roots of each set A_i with the index key root of the searched set q.

For each set A_i found in the S-tree, the algorithm begins from the root of the index key of the searched set q and recursively tests all nodes at all levels of the index key of the searched set q comparing them with the corresponding nodes in the index key of A_i. Both index keys may contain different number of nodes because there can exist non-empty nodes in the index key of A_i that are not relevant to the query. These non-relevant nodes correspond to index key leaf nodes containing elements that are not present in the query. Thus, the main difficulty in comparing index keys is to determine pairs of corresponding nodes in compared index keys. To cope with this problem we introduce the function $\texttt{skip}(A_i, l, i)$ (line 4). The funcion $\texttt{skip}(A_i, l, i)$ computes at the level l the number of nodes that have to be skipped in the index key of A_i to reach the node corresponding to $N_i^l(q)$.

The function computes the number of nodes to be skipped on the basis of the parent node at the higher level. Given internal node i at the level l of the index key of q. The function retrieves the node $N_{i\%d+1}^{l-1}(A_i)$ (where % denotes the modulo operator; this is the parent of the i-th node). The number of nodes that must be skipped in A_i is equal to the number of positions in $N_{i\%d+1}^{l-1}(A_i)$ set to '1' and preceding the position $(i\%d)+1$. The test for the containment of corresponding nodes is performed in the line (5).

As can be easily noticed, the algorithm requires, for index key roots retrieved from the S-tree, accessing of some part of the linked list of non-empty nodes. This access introduces some overhead. Nevertheless, this additional cost of the linked list traversal is negligible when compared to the cost of veryfying every hit and resolving false hits as required by all other indexing techniques. Besides, the average cost of the linked list is relatively low because many index keys are rejected very early. The hierarchical bitmap index performs significant pruning at the upper level of the index keys. The degree of pruning depends on the mapping of set elements to positions in the signature at the lowest level of the index. We intend to investigate this dependency in greater detail in future experiments. The ability to prune index keys at the upper levels combined with the lack of necessity to verify false hits provides the biggest pay-off in comparison with other set-oriented indexes.

Example 2. Consider the index key of the set S from the Example 1. It is stored in the form of the linked list $K(S) = \langle 1010, 1011, 0100, 0110, 1001, 1100, 0101 \rangle$. Let user query contain elements $q = \{9, 13, 40\}$. The algorithm starts with defining the index key on the searched set q. The index key for the query q is $K(q) = \langle 1010, 0011, 0100, 1000, 1000, 0001 \rangle$. Then, the algorithm begins at the first level and tests whether $N_1^1(S) \& N_1^1(q) = N_1^1(q)$. This yelds **true** because $1010 \& 1010 = 1010$. Next, the algorithm moves to the next level and considers the node $N_2^1(q) = 0011$. Function $\texttt{skip}(S, 2, 1)$ returns 0 as there are no nodes to be skipped in $K(S)$. Comparing $N_2^1(S)$ and $N_2^1(q)$ returns $1011 \& 0011 = 0011$ which is true. Test on $N_2^2(S)$ and $N_2^2(q)$ also succeeds. Then the algorithm advances to the third level. Function $\texttt{skip}(S, 3, 1)$ returns 1 because the node $N_3^1(S) = 0110$ must be skipped as it represents the items $\{2, 3\}$ which are not relevant to the query. Thus the next test compares nodes $N_3^2(S) = 1001$ and $N_3^1(q) = 1000$. This procedure continues until there are no more non-empty nodes in the index key of q. It is easy to notice that the remaining comparisons will succeed – $N_3^3(S) = 1100$ contains $N_3^2(q) = 1000$ and $N_3^4(S) = 0101$ also contains $N_3^3(q) = 0001$. Therefore, we conclude that the set S contains the searched set q.

□

The algorithm is very robust. Most operations are bit manipulations (bitwise `AND`) that perform very quickly. The most important feature of the algorithm is the fact that the hierarchical bitmap index always returns an exact answer and never generates any false hits. Therefore, the search algorithm doesn't contain any verification steps which are required when using other set indexing techniques. The elimination of the verification phase and the lack of the false hits is the main reason for which hierarchical bitmap index is superior to other set indexes and outperforms them significantly. In the next section we will present the results of the experimental evaluation of the index.

5 Experimental Results

The experiments were conducted on top of the Oracle $8i$ database management system running under Linux with two Pentium II 450 MHz processors and 512 MB memory. Data sets were created using DBGen generator from the Quest Project [1]. Table 1 summarizes the values of different parameters that affect the characteristics of the data sets used in our experiments. Data sets tend to mimic typical customer transactions in a supermarket. Number of distinct elements varies from relatively small (1000) to rather large (30000) (this is the number of different products being sold in a typical mid-sized supermarket). The size of a single set (40 elements) also represents the real size of the average customer basket in a supermarket. Query sizes vary from general queries (considering one or two elements) to very specific (considering several elements). The number of indexed sets varies from 1000 (very small database) to 100000 (relatively big database). In our experiments we compare the hierarchical bitmap index with two similar techniques, namely the signature index and the hash index.

Table 1. Synthetic data parameters

parameter	value
number of itemsets	1000 to 250000
domain size	1000 to 30000
itemset size	40
query size	1 to 20

Figure 3 presents the average response time for different searched set sizes. The response time is the average response time computed for the database size varying from 1000 sets to 250000 sets and the size of the domain of the indexed attribute varying from 1000 to 30000 elements. All subsequent response times are also computed as an average over the entire parameter range in Tab 1. The most interesting about the hierarchical bitmap index is the fact that the response time remains constant. Very general queries using small searched sets (1–5 elements) exhibit the same response times as very specific queries using large searched sets (10–20 elements). For general queries the hierarchical bitmap index outperforms other indexes significantly, which can be explained by the fact that the hierarchical bitmap index doesn't produce any false hits and doesn't need any verification. On the other hand, both signature and hash indexes must verify every set returned from the index. For general queries we expect a lot of ambiguity in signature and hash indexes. Therefore, the answers are several times slower. We argue that these general queries are most frequent in real world applications and that this fact even stronger advocates the use of our index. Nevertheless, even for very specific queries the hierarchical bitmap index performs two times better than the other techniques.

The results of the next experiment are depicted in Fig. 4. It presents the average response time with respect to the varying number of elements in the domain of the indexed attribute. It can be observed that for all domain sizes the hierarchical bitmap index performs several times better than signature or hash indexes. Again, it is worth noticing that the response time remains almost constant for all domain sizes. For small domains (1000 elements) our index is 10–15 times faster than the other techniques. For larger domains the difference diminishes but remains substantial.

Figure 5 shows the scalability of the hierarchical bitmap index in comparison with the remaining two indexes with regard to the database size. The worst performance can be observed with the hash index, because the increase in the database size results in more hits that have to be verified. Signature index produces less false hits but all answers still have to be verified. Again, the biggest gain of the hierarchical bitmap index lies in the fact that it doesn't produce any false hits and doesn't require any verification step. The response time of the hierarchical bitmap index scales linearly with the database size.

Finally, Fig. 6 presents the number of verified sets with respect to the domain size. The hierarchical bitmap index processes only the correct answers to the user query. The signature index performs worse because it has to process some of the

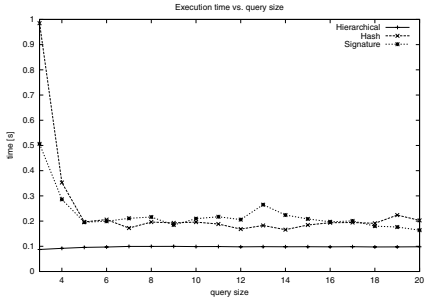

Fig. 3. Search time vs. query size

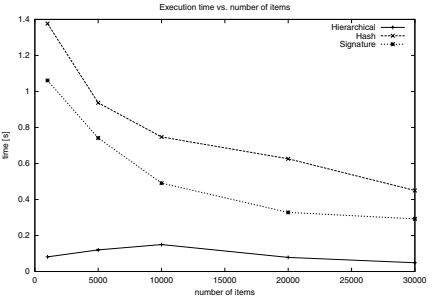

Fig. 4. Search time vs. number of items

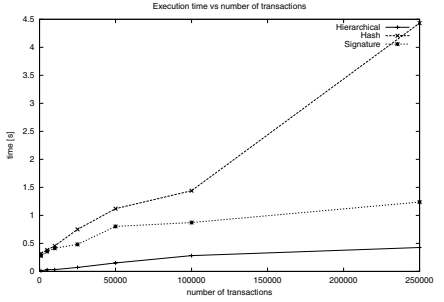

Fig. 5. Search time vs. number of itemsets

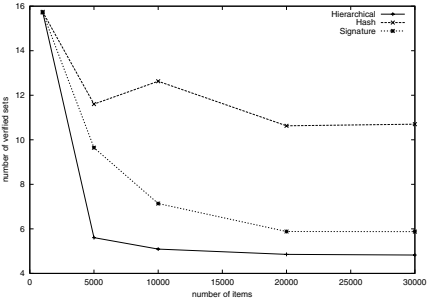

Fig. 6. Number of verified sets vs. number of items

sets that don't actually contain the searched set. Hash index reveals the poorest performance because of the excessive number of false hits (for 30000 different elements almost 70% of sets returned from the index tend to be false hits).

We measured also the performance of the traditional B+ tree index, but we omitted the results because B+ tree index was significantly inferior to all types of set-oriented indexes. For all database sizes and for almost all query sizes queries using B+ tree index ran ten times slower than the queries using the hash index.

6 Other Applications

As we said before the most commonly used class of set-oriented queries are subset queries. Beside subset queries there are two more classes of queries that can be used in terms of set-valued attributes. These are superset and similarity queries.

Superset queries retrieve all sets that are proper subsets of the searched set. Assume that the given set of products is offered at a reduced price. A superset query can be used to find the customers whose market basket is entirely included in the reduced product set, i.e., the customers who visited the shop only to profit from the discount. Another example of the superset query could be: given the pattern describing the correlation of sales of the set of products find all

customers who can be characterized using this pattern. Superset queries are also useful in web-based applications. If the searched set consists of web pages and links considered the main navigation path through the web site, the superset query can be used to identify inexperienced users who don't use any advanced features of the web site and who always stay within the main navigation route. Superset search procedure can't use the S-tree structure and has to test all index keys stored in the tree. The algorithm to perform a superset search on all sets contained in the attribute A of the relation R using the searched set q is given below.

```
1: for all A_i ∈ R.A do
2:   for all levels l do
3:     for all index key nodes i at level l in A_i do
4:       p = skip(q, l, i)
5:       if N_i^l(A_i)&N_{p+1}^l(q) ≠ N_i^l(A_i) then
6:         return(false)
7:       end if
8:     end for
9:   end for
10:  return(true)
11: end for
```

This algorithm is a slight modification of the subset search algorithm presented in Sec. 4. This time we are testing all nodes of the index key of S to see if they are entirely contained in the corresponding nodes of q. Function $skip(q, l, i)$ computes the number of nodes in q that must be skipped to reach the node corresponding to $N_i^l(A_i)$. It works exactly as in the subset search algorithm.

The third class of set-oriented queries contains the similarity queries. A similarity query retrieves all sets that are similar to the searched set with the similarity threshold provided by the user. There are many different measures of similarity between two sets. The most commonly used notion of similarity is the Jaccard coefficient which defines the similarity between sets A and B as

$$similarity(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

Similarity queries have numerous applications [9]. They can be used to cluster customers into distinct segments based on their buying behavior, to tune direct mail offerings to reach only the desired customers, to make automated recommendations to customers, to dynamically generate web page contents and direct customer-oriented advertisement, etc. Similarity analysis can operate on different levels of similarity. For example, to classify a new customer to one of the existing profiles we are interested in finding the most similar profile. On the other hand, when searching for customers who should be targeted with a new product catalog we are interested in finding those customers, whose purchases are similar to the set of offered products only to some degree (the most similar

customers already possess the majority of catalog products, hence they are not the right target for such offer). Finally, in many applications the contrary of the similarity, namely the dissimilarity, can be interesting to analyze. The analysis of the dissimilarity can reveal features and patterns responsible for differences in buying behaviour, web navigation habits, sales anomalies, etc.

For similarity queries no filtering can be applied before visiting all nodes belonging to a given index key. Some existing approaches [9,15] propose methods for efficient S-tree traversal in case of similarity queries. Incorporating those methods into hierarchical bitmap index is subject to our future work. 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 percent of common '1's is higher than the user defined threshold then the answer is positive, else negative. The algorithm to perform a similarity search on all sets contained in the attribute A of the relation R using the minimum similarity threshold *min_similarity* is given below.

1: **for all** $A_i \in R.A$ **do**
2: $c = 0$
3: **for all** levels l **do**
4: **for all** index key nodes i at level l in q **do**
5: $p = \texttt{skip}(A_i, l, i)$
6: $x = N_i^l(A_i) \& N_{p+1}^l(q)$
7: $c = c + \texttt{count}(x)$
8: **end for**
9: **end for**
10: **if** $\frac{c}{\texttt{numberofkeysinq}} \geq \texttt{min\_similarity}$ **then**
11: **return**(true)
12: **else**
13: **return**(false)
14: **end if**
15: **end for**

For every set A_i the algorithm iterates over all nodes of q and bitwisely `AND`s those nodes with corresponding nodes in the index key of A_i. The function `count(x)` computes the number of bits set to '1' in x. After the comparison of all node pairs is finished the percentage of common positions is calculated. If this percentage exceeds the user defined threshold of minimum similarity *min_similarity* the algorithm adds the given set to the result.

7 Conclusions

Our experiments have proven that existing indexing techniques are not suitable to efficiently index set-valued attributes. Hence, in this paper we have introduced the hierarchical bitmap index. It is a scalable technique to efficiently index large collections of sets. Our index is not affected either by the size of the domain

of the indexed attribute or the size of the indexed set. It scales well with the number of sets in the database and exhibits constant response time regardless the query size. Its compactness guarantees that the use of the index is memory efficient. All hierarchical bitmap index features stem from the compact and exact representation of the indexed sets. Because all answers obtained from the hierarchical bitmap index are exact, the hierarchical bitmap index doesn't need any verification steps that are required in case of all other indexing techniques.

This paper presents the results of the initial work on the hierarchical bitmap index. Further improvements will include:

- more sophisticated methods for mapping elements to signature bit positions. Storing elements that frequently occur together in the same index key leaves should lead to the improvement of the index compactness and should increase index filtering capabilities
- applying advanced methods of the S-tree construction to increase pruning at the upper levels of the tree and to improve the selectivity of the tree
- using advanced methods of the S-tree traversal to answer superset and similarity queries
- analyzing possible applications of the hierarchical bitmap index in advanced database querying, e.g., in analytical processing or data mining queries
- developing algorithms for index maintenance

References

1. R. Agrawal, M. J. Carey, C. Faloutsos, S. P. Ghosh, M. A. W. Houtsma, T. Imielinski, B. R. Iyer, A. Mahboob, H. Miranda, R. Srikant, and A. N. Swami. Quest: A project on database mining. In R. T. Snodgrass and M. Winslett, editors, *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, page 514, Minneapolis, Minnesota, may 1994. ACM Press.
2. M. D. Araujo, G. Navarro, and N. Ziviani. Large text searching allowing errors. In R. Baeza-Yates, editor, *Proceedings of the 4th South American Workshop on String Processing*, pages 2–20, Valparaiso, Chile, 1997. Carleton University Press.
3. C. Y. Chan and Y. E. Ioannidis. Bitmap index design and evaluation. In L. M. Haas and A. Tiwary, editors, *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, pages 355–366, Seattle, Washington, jun 1998. ACM Press.
4. S. Christodoulakis and C. Faloutsos. Signature files: an access method for documents and its analytical performance evaluation. *ACM Transactions on Office Information Systems*, 2(4):267–288, 1984.
5. D. Comer. The ubiquitous b-tree. *ACM Computing Surveys*, 11(2):121–137, 1979.
6. U. Deppisch. S-tree: a dynamic balanced signature index for office retrieval. In *Proceedings of the Ninth Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 77–87, Pisa, Italy, 1986. ACM.
7. C. Faloutsos. Signature files. In *Information Retrieval: Data Structures and Algorithms*. Prentice Hall, 1992.

8. M. Freeston, S. Geffner, and M. H rhammer. More bang for your buck: A performance comparison of bang and r* spatial indexing. In *Proceedings of the Tenth International Conference on Database and Expert Systems Applications DEXA '99*, volume 1677 of *Lecture Notes in Computer Science*, pages 1052–1065. Springer, 1999.
9. A. Gionis, D. Gunopulos, and 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.
10. A. Guttman. R-trees: A dynamic index structure for spatial searching. In B. Yormark, editor, *SIGMOD'84, Proceedings of Annual Meeting*, pages 47–57, Boston, Massachusetts, jun 1984. ACM Press.
11. J. M. Hellerstein and A. Pfeffer. The rd-tree: An index structure for sets. Technical Report 1252, University of Wisconsin at Madison, 1994.
12. S. Helmer and G. Moerkotte. A study of four index structures for set-valued attributes of low cardinality. Technical Report 2/99, Universität Mannheim, 1999.
13. Y. Ishikawa, H. Kitagawa, and N. Ohbo. Evaluation of signature files as set access facilities in oodbs. In P. Buneman and S. Jajodia, editors, *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 247–256, Washington, D.C., may 1993. ACM Press.
14. T. Morzy and M. Zakrzewicz. Group bitmap index: A structure for association rules retrieval. In R. Agrawal, P. E. Stolorz, and G. Piatetsky-Shapiro, editors, *Proceedings of the 4th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 284–288, New York, USA, aug 1998. ACM Press.
15. A. Nanopoulos and Y. Manolopoulos. Efficient similarity search for market basket data. *VLDB Journal*, 11(2):138–152, 2002.
16. K. Nørvåg. Efficient use of signatures in object-oriented database systems. In J. Eder, I. Rozman, and T. Welzer, editors, *Proceedings of the 3rd East European Conference on Advances in Databases and Information Systems (ADBIS)*, volume 1691 of *Lecture Notes in Computer Science*, pages 367–381, Maribor, Slovenia, sep 1999. Springer.
17. E. Tousidou, A. Nanopoulos, and Y. Manolopoulos. Improved methods for signature-tree construction. *The Computer Journal*, 43(4):301–314, 2000.