Similarity Search in Sets and Categorical Data Using the Signature Tree

Nikos Mamoulis, David W. Cheung, and Wang Lian
Department of Computer Science and Information Systems
University of Hong Kong
Pokfulam Road, Hong Kong
{nikos,dcheung,wlian}@csis.hku.hk

Abstract

Data mining applications analyze large collections of set data and high dimensional categorical data. Search on these data types is not restricted to the classic problems of mining association rules and classification, but similarity search is also a frequently applied operation. Access methods for multidimensional numerical data are inappropriate for this problem and specialized indexes are needed. We propose a method that represents set data as bitmaps (signatures) and organizes them into a hierarchical index, suitable for similarity search and other related query types. In contrast to a previous technique, the signature tree is dynamic and does not rely on hardwired constants. Experiments with synthetic and real datasets show that it is robust to different data characteristics, scalable to the database size and efficient for various queries.

1 Introduction

Similarity search is a core operation of many data analysis tasks in data mining, multimedia and time-series databases, biological and scientific databases. Database research has primarily focused on the special case, where the data and queries are points in a multidimensional space and the domains of the dimensions are numerical. However, in many applications multivariate analysis is applied on complex data domains which do not have a natural order.

Consider, for example, a database $\mathcal{D}$ that contains consumer transactions. Given a transaction Q, corresponding to a customer, a search problem is finding the most similar transactions in the database, in order to provide recommendations about items the customer would be interested in. If N is the total number of available items, this problem can be thought of as nearest neighbor search in an N-dimensional space, where the (discrete) domain of each dimension is $\{0, 1\}$. A related problem is similarity search in a multidimensional space, where the dimensions have categorical domains. It is not hard to see that it is a special case of the search problem in transactional data described above; the items correspond to values of categorical attributes and they are divided into g groups $G_1, G_2, \ldots, G_g$, which correspond to the natural dimensions (i.e., the attributes). The data are g-tuples $\langle t_1, t_2, \ldots, t_g \rangle$, where t_i is an element of group G_i. In this case the data tuples have fixed size and no two items (i.e., values) of the same group co-exist in a tuple; essentially, an attribute takes exactly one value in each tuple.

Although these problems are fundamental in data analysis tasks, they have not received much attention from the database literature, as opposed to the extensive work (e.g., [20, 18, 23, 21, 6]) for similarity search in low and high dimensional spaces of ordered domains. On the other hand, categorical and set data types are ubiquitous. For example, in most high-dimensional datasets of the UCI-KDD Archive [22], collected by real application domains, the majority of the attributes are categorical. In addition, *set* data types (e.g., market basket transactions) are frequently used to describe complex data in object-oriented/object-relational systems [11].

In this paper we show how a hierarchical index can be used to process efficiently similarity search and other related query types on sets and categorical data. In contrast to a previous method [1], the *signature tree* (SG–tree) is suitable for a dynamic environment with frequent updates and does not rely on hardwired constants, which are hard to define a-priori. The SG–tree is a natural extension of the B$^+$–tree and the R–tree [13], found in many commercial DBMSs. Thus, the index carries many advantages of these structures; it is (i) easy to implement (sharing most of its modules with them) and (ii) appropriate for various query types.

The remainder of the paper is organized as follows. Section 2 defines the problem of similarity search in sets and reviews related work. In Section 3 we describe the hierarchical indexing method. Section 4 shows how similarity search and other related queries can be evaluated using the SG–tree. Section 5 includes an experimental study which demonstrates the efficiency and applicability of our approach on synthetic and real data. Finally, Section 6 concludes the paper, with a discussion on issues not covered but well-worth studying in the future.

2 Background and Related Work

In this section we formalize the similarity search problem. The data space is defined and also the similarity function we use throughout the paper. Then we review related work with a focus on the previous method used to solve the problem.

2.1 Definitions

Consider a set S of items, with cardinality $|S|$, which are available in a supermarket, and assume that a customer transaction T can be modeled by a subset of S, indicating the items the customer bought. We can map T to an S-dimensional point, where each coordinate takes values from $\{0,1\}$, or an $|S|$-length bitmap, called signature:

Definition 1 *Let S be an ordered collection of interesting items. Let T be a subset of S containing the items bought in a transaction. The* **signature** *$sig(T)$ of T with respect to S is an $|S|$-length bitmap. For each $i, 1 \leq i \leq |S|$, the i-th bit of $sig(T)$ is 1 iff the i-th item of S is present in T.*

For example, let $S = \{a,b,c,d,e,f\}$. Transactions $T_1 = \{a,c\}$ and $T_2 = \{c,d,f\}$ can be represented by signatures $sig(T_1) = 101000$ and $sig(T_2) = 001101$, respectively. We now define some useful operations on signatures.

Definition 2 *Let s be a signature, i.e. a fixed-length bitmap. The* **area** *of s, denoted by $Area(s)$, is defined by the number of 1's in s. Let s_1, s_2 be two signatures. Let $\wedge$, $\vee$, and $\otimes$ denote the bitwise operations AND, OR and XOR, respectively, on bitmaps of the same length. The* **overlap** *$Ovr(s_1,s_2)$ between s_1 and s_2 is defined by $Area(s_1 \wedge s_2)$. The* **difference** *$Diff(s_1,s_2)$ of s_1 from s_2 is defined by $Area(s_1 \otimes (s_1 \wedge s_2))$. Finally, s_1 is said to* **contain** *or* **cover** *s_2 iff $s_1 \wedge s_2 = s_2$.*

In other words, the overlap between two signatures is the number of common 1-bits in them, and the difference $Diff(s_1,s_2)$ (non-commutative) is the number of 1-bits in s_1 but not in s_2. Finally, s_1 contains (or covers) s_2 if all set bits in s_2 are also set in s_1. For example, $Area(101100) = 3$, $Ovr(101100, 000110) = 1$, $Diff(101100, 000110) = 2$, and 101110 covers both 101100 and 000110.

Thus, given two transactions T_1, T_2, $Area(sig(T_1)) = |T_1|$, $Area(sig(T_2)) = |T_2|$, $Ovr(sig(T_1), sig(T_2)) = |T_1 \cap T_2|$, $Diff(sig(T_1), sig(T_2)) = |T_1 - T_2|$, and $sig(T_1)$ contains $sig(T_2)$ iff $sig(T_1) \cap sig(T_1) = sig(T_2)$. The reason for which we define these operations separately for transactions and signatures is that, as we will see later, they apply not only for single transactions, but also for sets of transactions. In addition, their computation on signatures is cheap and straightforward.

Similarity functions like the Euclidean distance are inappropriate for set (and categorical) data [12]. We use the *hamming distance*, a popular metric for this data space. Other metrics with similar effects are the *Jaccard coefficient* and the *cosine* function [16].[1]

Definition 3 *Let T_1, T_2 be two transactions. Their (hamming)* **distance** *is defined by the number of items in either transaction, but not in both of them: $Dist(T_1,T_2) = |T_1 - T_2| + |T_2 - T_1|$. Let s_1, s_2 be two signatures. The* **distance** *$Dist(s_1,s_2)$ between s_1 and s_2 is defined by $Area(s_1 \otimes s_2)$.*

For example, consider two transactions $T_1 = \{1,2,6\}$, $T_2 = \{2,3\}$. Their distance $Dist(T_1,T_2)$ can be calculated by $Dist(110001, 011000) = 3$. We are now ready to define the similarity search problem in set spaces with which we deal in this paper:

Definition 4 *Let S be an ordered collection of interesting items and let $\mathcal{D}$ be a database of transactions, such that each $T \in \mathcal{D}$ is a subset of S. Let $Q \subseteq S$ be a* **query transaction** *and k an integer smaller than the size of the database $|\mathcal{D}|$. The* **similarity search query** *retrieves from $\mathcal{D}$ the k-closest transactions to Q with respect to the distance function.*

Definition 4 is just one of the possible variations of similarity search, modeled by the k-nearest-neighbors query. There are several other variants which are useful in data analysis tasks. In Section 4 we define some of the most popular ones and show how they can be processed efficiently, using the SG–tree.

2.2 Related work

Similarity search in multidimensional spaces of numeric, ordered attributes is a well-studied topic. When the data are indexed by a hierarchical multidimensional access method [8], like the R–tree [13], a branch-and-bound nearest neighbor search algorithm [20, 15] can prune the search space efficiently. We propose a similar method to perform search on the SG–tree. In [19, 5] this method is combined with a spatial join algorithm [4] to evaluate similarity joins and closest pair queries in low-dimensional spaces. As the dimensionality increases, the effectiveness of R–tree to cluster together points close to each other degenerates and alternative

[1]Given two transactions T_1,T_2, their Jaccard coefficient is defined by $\frac{|T_1 \cap T_2|}{|T_1 \cup T_2|}$. Their cosine angle is defined by $\frac{|T_1 \cap T_2|}{\sqrt{|T_1|} \cdot \sqrt{|T_2|}}$.

methods are considered for nearest neighbor search. Some (e.g., [18]) are based on dimensionality reduction, others (e.g., [23, 21]) on compression and others (e.g., [6]) on data or query skew.

However, extending these methods to operate on set and categorical data is not straightforward. To our knowledge the only method previously proposed for similarity search in set and categorical data spaces is [1]. Due to its high relevance to our approach, we describe it in detail in the following paragraph. The similarity search problem for sets has also been studied in [11], where hash-based indexes which provide approximate results are proposed. In this paper, we deal with the problem of finding the exact answers to queries, thus our method is not directly comparable to these indexes. Finally, a similar hierarchical index to the SG–tree was proposed in [7]. Nevertheless optimization of insertions and splits has not been studied and the method is tuned/tested only for exact retrieval of signatures. Moreover, as shown in [14] signature trees are not appropriate for set equality or subset queries, which are best processed by inverted indexes and hash-based indexes. In this paper we demonstrate that the SG–tree is conversely suitable for similarity search.

The related problem of clustering categorical data has been studied during the past few years and several algorithms have been proposed (e.g., [10, 12, 9]). These methods apply on a static set of categorical data and consider the number of common neighbors as a metric of similarity between two transactions. Using the same techniques for nearest neighbor search requires preprocessing information which may not be available in a dynamic environment.

2.2.1 The signature table

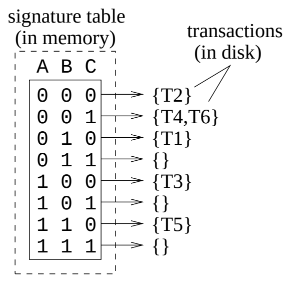
The *signature table* (SG–table) is a hash-based index, built from a static set of market-basket data $\mathcal{D}$. It is used to hash the transactions into a set of buckets based on their similarity to K frequent itemsets called signatures in [1]. In our paper we will refer to them as *vertical signatures*, to distinguish them from the signature definition that we use.

The SG–table [1] is constructed in two steps. First a *minimum spanning tree* algorithm is run to cluster the set of items S into K groups each containing frequently correlated items. The grouping process starts by considering each item a separate cluster and progressively refines the clusters by merging item pairs with the maximum co-occurrence frequency. In order to achieve clusters whose contents appear with approximately the same frequency in some transaction, groups for which the total support in the database of their contents exceeds a certain threshold (called *critical mass*) are removed before they grow larger.

The itemsets of the resulting clusters formulate the set of K vertical signatures, which are used to construct the SG–

S = {a,b,c,d,e,f,g}	T1 = {c,d}
	T2 = {a,b,c}
A = {a,e}	T3 = {a,b,e}
B = {c,d}	T4 = {b,d,f,g}
C = {b,f,g}	T5 = {a,b,c,d,e}
	T6 = {b,e,f}

(a) dictionary, vertical signatures, and transactions



(b) the signature table

Figure 1. Example of signature table

table and hash the transactions. Let r be a small constant called *activation threshold*. If a transaction T has at least r common items with a vertical signature V ($|T \cap V| \geq r$), T is said to *activate* V. Based on which vertical signatures a transaction activates it is hashed into one of the 2^K entries of the SG–table. Figure 1 shows an example of a signature table and a set of transactions $\{T_1, \ldots, T_6\}$ hashed into it. The items in the dictionary S are split into three groups A, B, C, and the activation threshold is set to 2. For example, transaction T_3 activates only the vertical signature A ($|T_3 \cap A| \geq 2$), and is hashed to the partition with binary code 100.

The index is used to answer similarity queries as follows. The query transaction is compared to each signature V and a lower bound for the distance between Q and the transactions indexed by the table entries, depending on whether their V-th bit is 0 or 1, is computed. These lower bounds are accumulated for each table entry in an (optimistic) estimation of the distance between Q and the transactions indexed by that entry. The table entries are sorted in increasing order of their lower-bound distance and the hash buckets are read in this order to be compared with Q. If after reading a partition the distance between Q and the k-nearest neighbor found so far is smaller than the optimistic bound in the next table entry (in the sorted order) the search stops, since none of the remaining entries may point to a closer transaction in the worst case (see [1] for more details).

Although the signature table can be fast for nearest neighbor search queries, it suffers from certain drawbacks. First, its performance is sensitive to various parameters (number of vertical signatures, critical mass, activation threshold) which are hard to determine a-priori and have to be tuned to achieve good performance. Second, it is appropriate for static data, on which a clustering algorithm has to be applied in order to determine the vertical signatures. The preprocessing cost is rather high and the index is sensitive to data updates (which may change the correlations between the items and their optimal grouping). Thus expensive periodic re-organization of the index is required in a dynamic environment. Finally, the SG–table is not efficient when the memory resources are limited. The experiments in [1] indicate that its performance drops fast as the space allocated for the memory-resident table decreases. Moreover, since the size of the table is hardwired at construction time, it does not adapt to dynamic changes in memory resources. In the next section we show how a hierarchical index can alleviate these problems.

3 The Signature Tree (SG–tree)

A nice property of the signatures is that we can use the *same* representation (i.e., a bitmap) for transactions and groups of transactions. In other words, assuming that $\mathcal{T}$ is a group of transactions, we can characterize it by a signature which has 1 in a position iff the corresponding item exists in *at least one* transaction in $\mathcal{T}$. Formally:

Definition 5 *Let $\mathcal{T}$ be a set of transactions. The signature of $\mathcal{T}$ is defined by*

$$sig(\mathcal{T}) = sig(\bigcup_{T \in \mathcal{T}} T) = \bigvee_{T \in \mathcal{T}} sig(T) \qquad (1)$$

This property is employed by a simple, yet efficient, hierarchical index for signatures. The SG–tree (or *signature* tree) is a dynamic balanced tree similar to R–tree [13] for signature bitmaps. Each node of the tree corresponds to a disk page (using multipage nodes is a potential implementation) and contains entries of the form $\langle sig, ptr \rangle$. In a leaf node entry, sig is the signature of the transaction and *ptr* is a *transaction-id*.[2] The signature of a directory node entry is the logical OR of all signatures in the node pointed by it and ptr is a pointer to this node. In other words, the signature of each entry is the signature of all transactions in the subtree pointed by it. All nodes contain between c and C entries, where C is the maximum capacity and $c \leq C/2$, except from the root which may contain fewer entries. Figure 2 shows an example of a signature tree. The leaf entries contain the signatures and ids of nine transactions. In this graphical example the maximum node capacity C is three and the signatures are six bits long. In practice, C is in the order of several tens and the length of the signatures in the order of several hundreds.

The SG–tree is not useful only for restricted types of queries, but can serve as a general-purpose index for set data. In Section 4 we will describe how it can be used to evaluate similarity search queries. Here we will discuss briefly how it can be used for simple queries, like *itemset containment queries*, e.g., find all transactions containing items b and f. Assuming that $S = \{a, b, c, d, e, f\}$, this

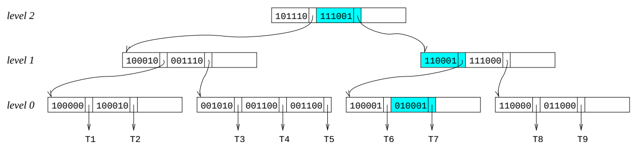


Figure 2. Example of a signature tree

query can be transformed to a signature $sig(Q) = 010001$ and the tree is traversed in a depth-first fashion to evaluate it. The search algorithm follows entries whose signature contains $sig(Q)$; if the signature of an entry does not contain $sig(Q)$, no transaction indexed in the subtree below it can participate in the result. Consider for example the tree of Figure 2. Since the first entry of the root has 0 in the sixth position, we know that no transaction indexed in the subtree under it can participate in the query result. On the other hand, the second entry should be followed and the rightmost node of the next level (i.e., level 1) is visited. Only the first entry of this node contains the query signature, and it is followed. Finally, the query result is found in the third leaf node. The qualifying entries are highlighted in the figure. Observe that the number of visited pages in this case is optimal. On the other hand, assuming that we are looking for transactions containing item c, multiple paths are traversed and a significant part of the tree is accessed. Therefore the efficiency of the tree increases if transactions with similar signatures are clustered together in the leaf nodes. This observation holds for all query types including similarity search.

3.1 Construction and updates

The insertion algorithms of hierarchical access methods aim at a common goal: to bring together indexed units which have small distance between them and separate well ones with large distance. The B$^+$–tree uses the natural order of the indexed domain to solve the problem optimally. Multidimensional access methods like the R–tree employ heuristics to achieve this goal, since there is no total ordering of objects in space that preserves spatial proximity [8]. Figure 3 shows the generic insertion algorithm used for these hierarchical access methods. When a new entry e needs to be inserted, the algorithm is called with parameters the root node and e and recursively traverses the tree in order to find the most appropriate leaf node to accommodate e. If the leaf node overflows a *split* algorithm divides the entries into two groups and moves one group to a newly created node. A pointer to the new node is returned to the parent directory node and a new entry is created for it. Splits are recursively propagated upwards.

The core components of the $insert$ function are $choose\_subtree$ and $split$. The first chooses the most ap-

[2] The transaction-id, although not necessary during nearest neighbor search, may be useful when search on the tree is combined with other operations (e.g., there may be additional features related to a transaction, like customer class).

```
function $insert$(Node $n$, Entry $e$): Node $splitnode$ {
  if $n$ is a leaf node then{
    insert $e$ into $n$;
    if $n$ overflows then
      $newnode$ := $split(n)$;
      return $newnode$;
  }
  else { /* $n$ is a directory node */
    $next$ := $choose\_subtree(n, e)$;
    $splitnode$ := $insert(next, e)$;
    if $splitnode \neq null$ then
      insert new entry pointing to $splitnode$ in $n$;
      if $n$ overflows then
        $newnode$ := $split(n)$;
        return $newnode$;
  }
  return $null$;
}
```

Figure 3. Insertion in balanced tree indexes

propriate entry of the current node n in order to insert e under it. The second divides the entries of an overflowed node into two groups. Both functions should be tuned to maximize the efficiency of the tree. For the SG–tree, we need to define quality criteria based on which these functions operate. The directory node entries of a good SG–tree should have (i) a small area[3], which intuitively decreases the distance between the transactions in the subtrees indexed by them and (ii) small overlap between them, if they are at the same level, which intuitively discriminates as much as possible the branches of the search process and maximizes the data that are pruned during search.

The $choose\_subtree$ algorithm we used in our SG–tree implementation can be described as follows. When a entry e is to be inserted in the subtree under node n three cases are considered. In the first case, only one entry $e_i \in n$ contains the new entry e and it is directly chosen. In the second case, multiple entries contain e. The algorithm chooses the one with the minimum area, since this refines the structure (in analogy to choosing the smaller MBR that contains the new entry in R–trees). Finally, the third case applies when no $e_i \in n$ contains e. The algorithm in this case picks the entry which requires the smallest area enlargement to index e under it, or more formally the entry for which $Diff(e, e_i)$ is the minimum. Ties are broken by choosing the entry with the minimum area. We also implemented another version of $choose\_subtree$ that picks the entry which, after extended, has the minimum overlap increase with the rest of the entries in the same node. Nevertheless, through experimentation we found that the minimum area enlargement heuristic creates trees of the same quality at a much lower insertion cost.

[3]For simplicity, we extend the function definitions of Section 2.1, to apply on SG–tree node entries, e.g., $Area(e) \equiv Area(e.sig)$.

For the split algorithm of the SG–tree we consider several alternatives. The first one is based on the quadratic-split method of the R–tree [13]. We first pick the pair of entries in the overflowed node with the maximum distance. We call these two entries *seeds* and assign them to two groups, with initial signatures same as the seeds. The rest of the entries are assigned to the group that requires the smallest signature area enlargement to include them. Ties are broken by choosing the group with the minimum area. In case of a new tie the group with the minimum number of entries is selected. If at some point the cardinality of a group plus the number of remaining entries equals c, the remaining entries are assigned to the group to avoid underflow in the new node.

We also consider two more approaches for splitting a node. The first is based on hierarchical clustering with *group average* [16]. Initially, all entries are considered as clusters. Then clusters are hierarchically merged until only two remain; these will form the new nodes after the split. The next pair of clusters $\langle C_1, C_2\rangle$ to be merged is the one for which the average distance $Dist(e_1.sig, e_2.sig)$ between pairs $\langle e_1, e_2\rangle$ of entries $e_1 \in C_1, e_2 \in C_2$ in them is the smallest. In order to avoid under-utilization of a node, when a cluster grows above a threshold (according to c) the other clusters are immediately merged and the algorithm terminates. We denote this method by ga_split. The last split policy is based on hierarchical clustering according to the *minimum spanning tree*; the next pair of clusters $\langle C_1, C_2\rangle$ to be merged is the one containing the closest pair of entries $\langle e_1, e_2\rangle, e_1 \in C_1, e_2 \in C_2$. We denote this method by mst_split. In Section 5 we compare r_split, ga_split, and mst_split.

Finally, deletions in the SG–tree are handled as in the R–tree; if a leaf node underflows, it is deleted, the entries are put in a temporary buffer and reinserted to the tree. This increases space utilization and the quality of the tree.

3.2 Compression

In many cases the signatures are very sparse, i.e., a single transaction contains only a small percentage of the possible items. Only few bits are then set in the signatures and saving them as bitmaps would waste a lot of space. In order to alleviate this problem we use a compression technique; if a bitmap is too sparse we choose to encode the signature as a list of positions, where the bits are set (or else as a list of item-ids). For example a 256-bit signature having only 10 1's would be encoded by a sequence of 10 characters indicating the positions of the 1's which occupy 10 bytes as opposed to 32 bytes needed to store the bitmap. We also store an extra flag-byte, which stores the number of 1's and also indicates that the next bytes contain the positions of 1's. Other compression schemes can also be employed, but it is out of the scope of this paper to study their effectiveness.

4 Query Processing

In this section we describe how the branch-and-bound techniques for similarity search on R–tree-like structures can be adapted to perform search on the SG–tree efficiently. We discuss first the most common and simple types of similarity search and then some more complex queries.

4.1 Similarity search

Given a query transaction Q, we can identify two types of similarity search queries on a database $\mathcal{D}$ of transactions, which constitute components of various data analysis tasks. The first is the *similarity range query*, asking for all $T \in \mathcal{D}$ within some distance ϵ from Q. The second is the *nearest neighbor search query*, asking for the k closest $T \in \mathcal{D}$ to Q, given a (small) constant k. Both queries can be evaluated efficiently if the database is indexed by an SG–tree. The search algorithms are adaptations of the equivalent ones that apply on an R–tree and they take advantage of the *coverage* property of the entries in the directory nodes to derive distance bounds for the transactions indexed by them. We first confine our discussion on the evaluation of nearest neighbor queries, where k=1 (i.e., simple nearest neighbor queries) and then show how the same techniques can be extended for the other cases.

Figure 4 shows a depth-first search algorithm for nearest neighbor queries on R–trees [20], adapted for the SG–tree. Two variables NN and $NNdist$ are initialized to *null* and ∞, corresponding to the nearest neighbor found so far and its distance from Q. The branch-and-bound $DF\_nnsearch$ algorithm is initially called for the root of the SG–tree. It recursively traverses the tree, following the entries that are most likely to contain the query result. When visiting a directory node, the entries e are sorted according to $Diff(Q, e.sig)$, which provides a lower (optimistic) bound for the nearest neighbor of Q in the subtree indexed by e. Intuitively, by visiting the subtrees in this order, the chances of finding early the result are maximized. Ties between entries having the same lower bound are broken by picking first the one with the minimum area. This secondary sorting key is due to the fact that among several subtrees with the same number of common items with Q the one with the smallest area is more likely to index an entry with exactly these common items (i.e., the optimistic nearest neighbor). In other words, given two groups of transactions G_1,G_2, where $|G_1| = |G_2|$ and $Area(sig(G_1)) < Area(sig(G_2))$, and an itemset I that could be included in both G_1 and G_2 (i.e., both $sig(G_1)$ and $sig(G_2)$ cover $sig(I)$), probabilistically the group with the smallest area (i.e., G_1) is more likely to contain I.[4]

```
function $DF\_nnsearch$(Node $n$, $Q$, $NN$, int $NNdist$) {
  if $n$ is a directory node then {
    sort entries $e$ in $n$ in ascending order of $Diff(Q, e.sig)$;
    break ties by placing first the entries with the smallest area;
    for each entry $e$ in this order do /* recursive call */
(1)   if $Diff(Q, e.sig) < NNdist$ then
        $DF\_nnsearch(e.ptr, Q, NN, NNdist)$;
      else break for loop; /* no need to visit other subtrees */
  }
  else /* $n$ is a leaf node */
    for each entry $e$ in $n$ do
(2)   if $Dist(Q, e.sig) < NNdist$ then /* new NN found */
        $NN$ := $e$; $NNdist$ := $Dist(Q, e.sig)$;
}
```

Figure 4. A depth-first search algorithm for NN queries

The nodes under the entries are visited in this order. If the optimistic bound for some subtree is greater than the distance of the nearest neighbor found so far, search is not required for this subtree and the remaining ones in the order, since they may not contain a closer neighbor to the one already found. When a leaf node is visited during the search process, the distances between Q and all its entries are computed and the bounds NN and $NNdist$ are updated if a closer neighbor is found.

The search algorithm of Figure 4 is appropriate for finding one nearest neighbor of Q. It can be easily adapted for finding all nearest neighbors with the same (minimum) distance from Q, by maintaining a set of current nearest neighbors instead of a single variable NN, and changing the predicates in lines (1) and (2) of the algorithm to '$\leq$'. In the general problem, where the k nearest neighbors are required (k-NN search) the parameter NN is replaced by a priority queue of size k, organizing the k nearest neighbors found so far, and $NNdist$ bound corresponds to the first element of the queue, i.e., the one with the largest distance, among the k-NN found so far.

The algorithm of Figure 4 can also be used to evaluate similarity range queries. In this case, the $NNdist$ bound is replaced by the (fixed) query parameter ϵ and all transactions within this distance from Q are retrieved. The directory entries with $Diff(Q, e.sig) > \epsilon$ are pruned as before, filtering out large parts of the data early.

Finally, we need to mention that the $DF\_nnsearch$ algorithm of Figure 4 is, in fact, sub-optimal for nearest neighbor search on the SG–tree. An optimal $BF\_nnsearch$ algorithm (in terms of node accesses) follows a *best-first search* paradigm [15] and employs a priority queue. This queue organizes $\langle e, Diff(Q, e.sig)\rangle$ tuples for directory

[4]G_1 has higher density than G_2 and thus higher probability to include I.

node entries. The first element of the queue contains always the entry with the minimum $Diff(Q, e.sig)$ and ties are broken using the minimum area as above. Initially, the queue contains the root entries, and the nearest neighbor information (NN and $NNdist$) is initialized as in $DF\_nnsearch$. At each step $BF\_nnsearch$ gets the entry e_f in the first element of the queue and the corresponding SG–tree node is loaded. If it is a non-leaf node, its entries are inserted into the queue. Otherwise, the transactions are compared with Q to potentially update NN and $NNdist$. If at some point, after removing a queue element e_f, $Diff(Q, e_f.sig) \geq Ndist$, search stops since we know that the remaining elements of the queue may not point to a closer leaf entry than the already found NN.

4.2 Other query types

Since the SG–tree has similar properties with the R–tree, other data analysis queries can be evaluated by adapting the corresponding algorithms used for R–trees. In this section we provide some examples, without intending to exhaustively cover all potential query types. The evaluation of these queries is based on tree traversal; the signatures at the upper levels of the tree(s) are used to derive some bounds that facilitate pruning large parts of the search space.

An extension of the nearest neighbor search query is when the query transaction is not a simple transaction Q, but a set of transactions $\mathcal{Q}$ and we wish to find the transaction T in $\mathcal{D}$ which minimizes an aggregate function of its distances from the queries in $\mathcal{Q}$ [1]. An example query is 'find the transaction with minimum average distance from $\mathcal{Q} = \{Q_1, Q_2, Q_3\}$'. More formally, assuming that f is an aggregate function (e.g. *average*), the nearest neighbor of $\mathcal{Q}$ in $\mathcal{D}$ is defined by: $T \in \mathcal{D} : \neg\exists T' \in \mathcal{D}, f(Dist(Q,T), \forall Q \in \mathcal{Q}) > f(Dist(Q,T'), \forall Q \in \mathcal{Q})$. This query can be easily evaluated by applying the nearest neighbor search algorithms described in the previous section. Consider for simplicity the $DF\_nnsearch$ algorithm. At intermediate nodes the entries are sorted using the aggregate of the $Diff(Q, e.sig)$ of all $Q \in \mathcal{Q}$. This value is used to prune subtrees, if the lower distance bounds are *monotonic* to the application of the aggregate function. The most popular aggregate functions (e.g., weighted average, min, etc.) have this monotonicity property.

Other query types used for automated data analysis tasks like clustering are *similarity join queries* [19] and *closest pairs queries* [5]. They can be thought of as extensions of the similarity range query and nearest neighbor search, respectively. The similarity join query retrieves from two datasets $\mathcal{D}_1$ and $\mathcal{D}_2$ the pairs of transactions $\langle T_1, T_2\rangle$, $T_1 \in \mathcal{D}_1$ and $T_2 \in \mathcal{D}_2$, such that $Dist(T_1, T_2) < \epsilon$, where ϵ is a small constant. The closest pairs query retrieves from two datasets $\mathcal{D}_1$ and $\mathcal{D}_2$ the k closest pairs of transactions $\langle T_1, T_2\rangle$, $T_1 \in \mathcal{D}_1$ and $T_2 \in \mathcal{D}_2$. Both query types have been defined and studied before for numerical data, but to our knowledge not for set data and categorical data. Indexing $\mathcal{D}_1$ and $\mathcal{D}_2$ using SG–trees can facilitate processing them, if we use extensions of the search algorithms defined for R–trees (see [19, 5] for details).

5 Experimental Evaluation

In this section we evaluate the efficiency of the SG–tree for various similarity search queries over synthetic and real datasets. We implemented the SG–tree and the SG–table [1], the previous method for similarity search in market-basket data. The parameters K and activation threshold r for the SG–table were tuned to $K = 15$ and $r = 1$.[5] For the SG–tree we used a node size of 4K. The experiments were run on a PC with a Pentium III 800MHz processor and 256MB of memory, running Linux 2.4.7-10. In the next subsection we describe the characteristics of the synthetic and real data used in the experiments.

5.1 Description of the datasets

We generated synthetic market basket data using the generator from [2, 1], which produces large collections of transactions simulating actual customer behavior. Given a set of $N = 1000$ items (i.e., $|S| = N$), initially a set of $L = 2000$ *maximal itemsets* is generated, modeling maximal itemsets frequently found in transactions. The number of items in each itemset is a random variable from a Poisson distribution of mean value I. The items in the first itemset are picked randomly. To simulate the fact that common items are often found in frequent itemsets, each successive itemset is generated by picking half of the items from the previous one and generating the other half randomly. Each of the L itemsets are assigned a weight from an exponential distribution with unit mean. The generated transactions are noisy combinations of these itemsets. The size of a transaction follows a Poisson distribution of mean value T. Itemsets are randomly chosen by rolling a weighted L-sided die, according to their weights, and inserted to a transaction until it is full. If an itemset does not fit to a transaction it is inserted to it with probability 0.5. Some noise is added while adding an itemset in the transaction; we keep dropping an item from the itemset as long as a uniformly random number between 0 and 1 is less than 0.5. More details about the generator can be found in [2, 1].

Thus, each dataset is characterized by three parameters: the mean transaction size T, the mean size of a potentially

[5]Small values of K lead to fewer entries on the SG–table and a less refined data partitioning. On the other hand, large values of K assign few transactions to each table entry, increasing the number of random accesses and the costs of comparing Q to each table entry and sorting them.

large itemset I, and the cardinality D. For example, a dataset with 200,000 transactions of mean size 10 and large itemsets of mean size 6 is denoted by T10.I6.D200K. By tuning these parameters we were able to generate a wide range of datasets with various characteristics.

We also experimented with a real, categorical dataset from the UCI KDD Archive [22]. The dataset contains census data extracted from the 1994 and 1995 current population surveys conducted by the U.S. Census Bureau. Each tuple corresponds to an individual and includes demographic and employment related information. After removing some numerical attributes and cleaning the data (e.g., missing values were replaced by an extra special value for each attribute), we ended up with 36 categorical attributes, the domain sizes of which vary from 2 to 53 (the total number of values is 525). The data are split into two datasets with 200K and 100K tuples, respectively. We indexed the first dataset (which we denote as CENSUS) and we used random samples from the second for querying it.

5.2 Comparison between split policies

In the first experiment we compare the three SG–tree split policies described in 3.1. We generated three (uncompressed) SG–trees for the CENSUS dataset, using r_split, ga_split, and mst_split, respectively. Table 1 compares the characteristics of the resulting trees and shows their relative performance averaged on 100 nearest-neighbor queries.

Table 1. Comparison of the three split policies

comparison metric	r_split	ga_split	mst_split
average $Area(e)$ at level 1	90	73	74
average $Area(e)$ at level 2	210	158	154
average $Area(e)$ at level 3	458	325	348
insertion cost (msec)	0.331	0.655	0.645
% of data accessed	15.79	4.78	5.72
CPU time (msec)	119	34.6	41.8
I/O accesses	862	266	323

All three trees have 4 levels. The entries in level 0 (leaf level) have fixed area 36 (since all data tuples have 36 values). The first three rows of Table 1 show the average area of the entries at levels 1,2, and 3 (root). This can be considered as a quality metric for the three split policies; the smaller the average area of the entries at the intermediate levels, the better the quality of the clustering. The ga_split and mst_split policies construct much better trees than r_split, and this can be validated from the last three rows of the table which show the average pruning (in terms of data accessed), the average CPU cost, and average number of node accesses at nearest neighbor search queries. On the other hand, r_split has the lowest average insertion cost and tree construction time. Experimentation with other datasets shows similar results. In the sequel we use ga_split as the standard split policy for the SG–tree, since it achieves the best quality of the three at an acceptable cost.

5.3 NN search on synthetic data

We compared the performance of SG–table and SG–tree on nearest neighbor search by generating a series of synthetic datasets and using the same itemsets and parameters to also generate a number of queries for each dataset. Figures 5 through 12 show the relative performance of the methods for various parameter settings. For each experimental instance, the results were averaged over 100 queries. Figures 5, 7, 9, 11, and 12 show in combined diagrams the pruning efficiency (bars) and computational cost (lines) of the two methods. The pruning efficiency is measured in terms of the transactions accessed and compared with the query transaction (percentage). Figures 6, 8, and 10 compare the number of random I/Os on the two indexes for three of the five experimental instances.

Figures 5 and 6 show the performance of the indexes when the size of itemsets is fixed (I=6), the size of the dataset is 200K, and the size of the transactions (T) varies. When T is small, both indexes have similar performance, but as T increases the SG–tree starts to (slightly) outperform the SG–table, managing to prune more transactions. Especially the I/O cost difference is high for large values of T, since in that case the distance of the nearest neighbor usually increases and the contents of many entries of the SG–table need to be visited.

Figures 7 and 8 show the relative costs for T=30 as the size of the large itemsets (I) increases. This increase generates datasets where the transactions are better clustered having smaller average distance between them and favors both structures. Observe that the relative performance between them increases, and the SG–tree becomes significantly faster than the SG–table when both T and I are large.

In the third experimental instance (Figures 9 and 10) we fix the ratio I/T to 0.6 and increase the transaction size. The rationale is to test the robustness of the indexing methods to the dimensionality of the problem, when the data skew remains constant. Clearly, the SG–tree is robust to the transaction size, whereas the SG–table fails to index well large transactions even if they contain well-clustered data. This observation is also validated at the comparison of the structures for real categorical datasets of high dimensionality (see Section 5.4).

We also tested the robustness of the two structures to the database size, by fixing T=10 and I=6 (two parameter values for which the SG–table performs well) and increasing the dataset cardinality D. Figure 11 shows that the relative pruning efficiency of the SG–tree increases with the database size. The I/O cost diagram is omitted since it

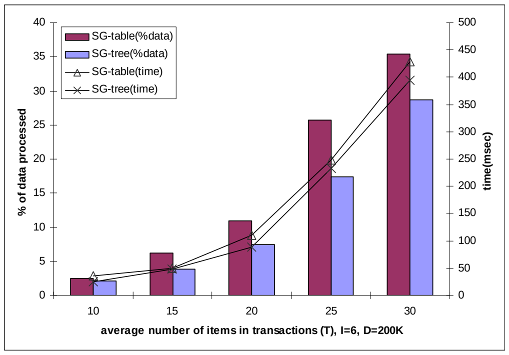


Figure 5. Pruning and CPU time, varying T

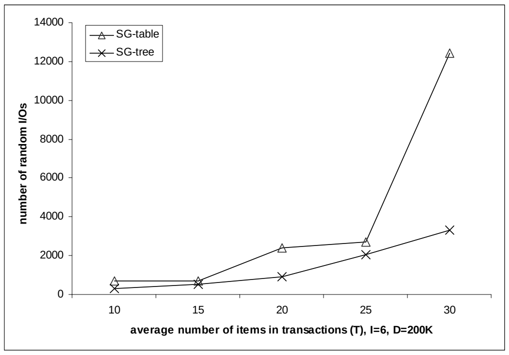


Figure 6. Random I/Os, varying T

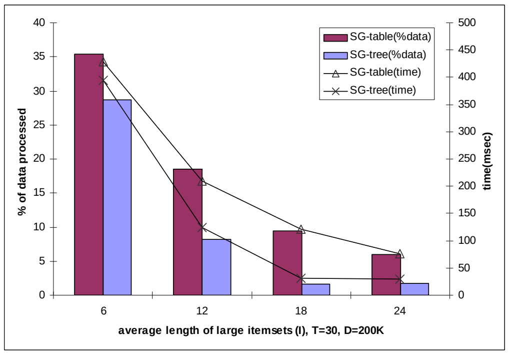


Figure 7. Pruning and CPU time, varying I

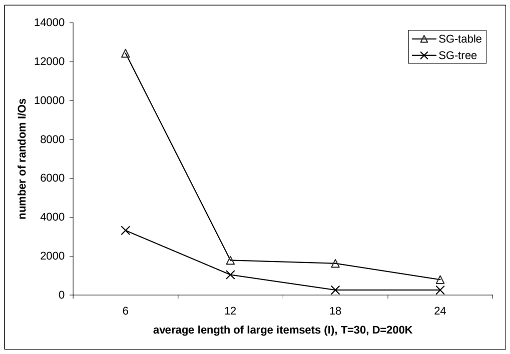


Figure 8. Random I/Os, varying I

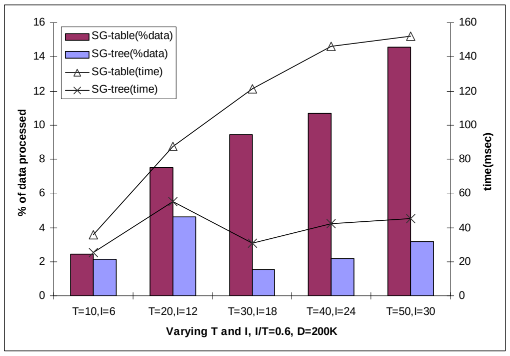


Figure 9. Pruning and CPU time, fixed I/T

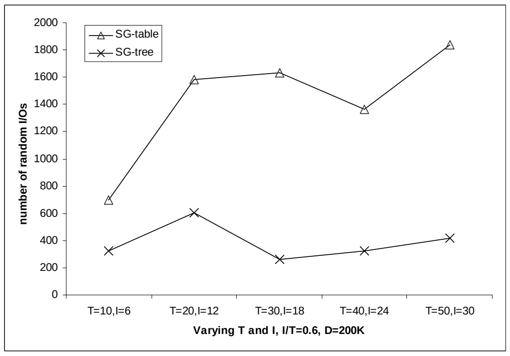


Figure 10. Random I/Os, fixed I/T

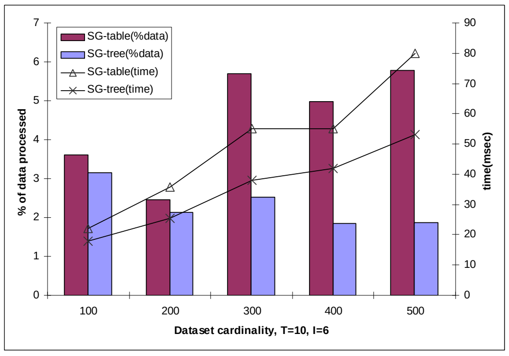


Figure 11. Pruning and CPU time, varying D

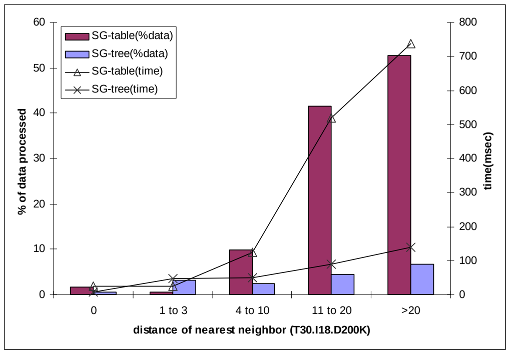


Figure 12. Pruning and CPU time, var. $NNdist$

shows a pattern similar to the CPU cost (as in the previous experiments).

During the experiments we observed that queries having a close nearest neighbor were processed fast using both structures, whereas for cases with distant neighbors the SG–tree was significantly faster than the SG–table. We validated this observation by running 1000 queries on the T30.I18.D200K dataset and averaging the query costs for various distance ranges of the nearest neighbor. Figure 12 shows the average pruning and CPU cost for five distance ranges. When the distance is small search is fast for both methods (actually for distances in the range 1–3, the SG–table outperforms the SG–tree). However the distant cases are handled much faster by the SG–tree, showing that this access method is more robust to 'outlier' queries.

As a general conclusion from this set of experiments, the SG–tree is a more efficient and robust access method than the SG–table, in addition to its other inherent advantages (dynamic data handling, independence to hard-wired constants). In the next subsection we compare the indexes for other query types on both synthetic and real data.

5.4 Real data and other queries

Figures 13 and 14 show the performance of the indexes for k-NN queries on the T30.I18.D200K synthetic dataset and the CENSUS dataset, respectively, for various values of k. The results for each experimental instance were averaged over 100 queries. In both figures, for small to medium values of k the SG–tree is significantly faster than the SG–table. When k is large ($\geq$ 1000), the fraction of the data that need to be visited becomes too large for the indexes to be useful. This is due to the fact that the search space becomes less appropriate for search. For example, when $k = 10000$ we observed that the average distance of the k-th neighbor is very large (31.81 for T30.I18.D200K and 18.06 for CENSUS) and very close to the average distance of all transactions from Q. This is due to the 'dimensionality curse' effect [3] often observed in high-dimensional search problems. Observe, that the SG–tree is less sensitive to this effect, since its performance degenerates at a smaller pace, especially for the real dataset.

We also compared the indexes for similarity range queries (Figures 15 and 16). The same datasets and queries as before are used and the distance threshold from the query varies from 2 to 10. For $\epsilon = 2$, the SG–table outperforms the SG–tree on the synthetic dataset. In all other cases the tree is much faster. Observe that on the real dataset, in particular, for both k-NN queries and range queries the performance difference quite large in favor of the tree. This indicates that the structure can perform very well in real life cases.

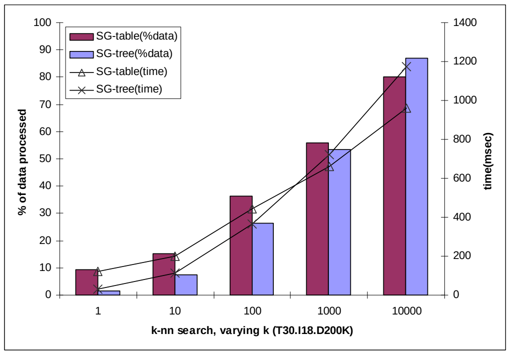


Figure 13. k-NN queries (T30.I18.D200K)

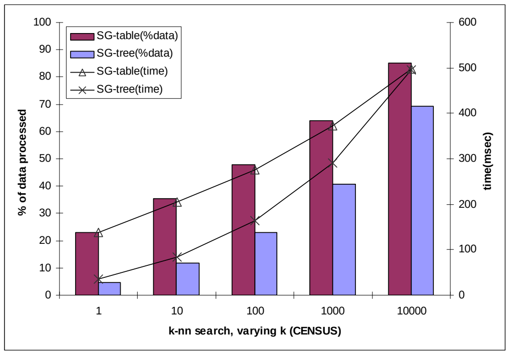


Figure 14. k-NN queries (CENSUS)

5.5 Dynamic data changes

In this experiment we compare the structures simulating a case where the nature of the data changes dynamically. We generated a synthetic dataset T10.I6.D100K and built an SG–table and SG–tree for it. We then gradually updated the structures by inserting batches of 100K transactions each with the same characteristics (i.e., T=10, I=6), but putting different seeds to the random generator (i.e., the large itemsets used were different for each batch). We ran nearest neighbor queries on the two structures after each insertion phase. The queries for phase b (after batch b has been inserted, $1 \leq b \leq 5$) are generated as follows. For each query (i) a random number i from 1 to b is chosen and (ii) the generator parameters (i.e., large itemsets) for batch i are used to produce the query. For example, a query for the phase where the dataset contains 300K data is generated using randomly one of the generators of batches 1, 2 or 3.

Figure 17 shows the average pruning efficiency and CPU time of the two structures. Initially, both have similar performance, but as more data with different characteristics are inserted into the structures the performance of the SG–table degenerates, since it is optimized for the first 100K data.

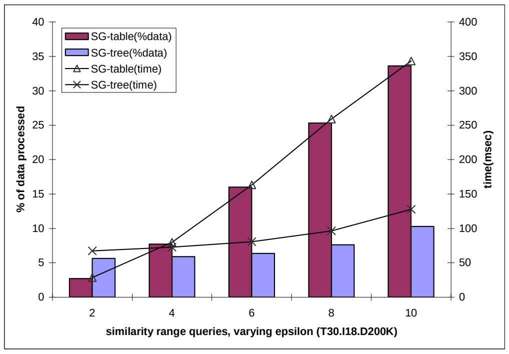


Figure 15. Range queries (T30.I18.D200K)

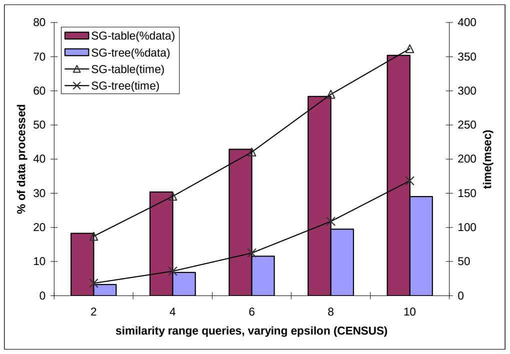


Figure 16. Range queries (CENSUS)

On the other hand, the SG–tree is robust to updates and exhibits very good query performance, since each batch contains skewed data (generated from a different collection of large itemsets).

6 Conclusions and Future Work

We presented a hierarchical indexing method for similarity search in sets and categorical data. The SG–tree is a disk-based height-balanced tree that organizes fixed-length bitmaps and is appropriate for various query types. We have shown how several branch-and-bound methods, which apply on R–tree-like structures, can be adapted for efficient similarity search on the SG–tree. Extensive experimental evaluation has shown that the SG–tree is in most cases much faster than the SG–table, a previous, hash-based index. The advantages of the SG–tree can be summarized as follows:

- It is efficient and robust to various data types (both categorical and set data) and characteristics (cardinality, density, dimensionality). It is a versatile structure that can be used for several query types.
- The tree is dynamically adapted to updates and requires no preprocessing of the data. Thus it can be useful for analyzing data which change dynamically over time.
- It relies on no hardwired constants, and requires no tuning using a-priori defined parameters.
- It is a disk-based, paginated data structure, so it can operate with limited memory resources, and dynamically changing memory resources. Caching policies, previously used for the B$^+$–tree and the R–tree can be seamlessly applied on this structure.

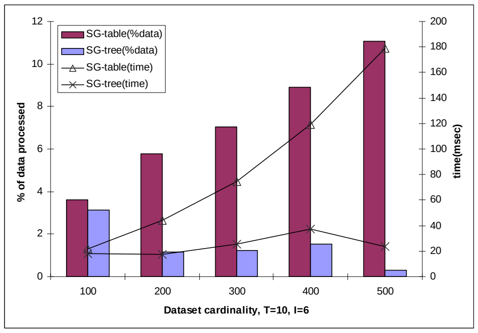


Figure 17. NN search after dynamic updates

There are several directions for extending the current work. In our study we used hamming distance as the similarity metric. However, the SG–tree can also be defined, tuned and searched for other set theoretic similarity metrics. For example if the Jaccard coefficient is used, the lower distance bound (in fact the upper similarity bound) for nearest neighbor search can be defined by $Area(Q - e.sig)/Area(Q)$. We plan to test the effectiveness of the structure using alternative metrics.

Another direction for future work is to study methods for bulk-loading SG–trees, instead of inserting the data one-by-one. We can adapt categorical clustering algorithms [12, 9] for this purpose. Another approach is to sort the transactions using gray codes as key, in analogy to using space-filling curves for bulk-loading multidimensional data to an R–tree [17]. Alternatively, hashing techniques can be used to group similar signatures together. The resulting 'globally-optimized' tree could have much better quality characteristics, while being built faster. In a reverse direction, we can investigate whether the SG–tree can be used for clustering large, dynamic collections of set and categorical data. The cost of existing methods is at least $O(n^2)$ and the tree could be used to derive good clusters much faster (e.g., by merging the leaf nodes using their signatures as guides).

Finally, we plan to empirically test the efficiency of the tree to the query types, discussed in Section 4.2. In addition, for some data types search can be further optimized. For example, if the indexed categorical data have fixed-dimensionality d we know that the area of each indexed signature is fixed to d. We can use this property to derive stricter lower bounds for the directory node entries e, instead of the rather relaxed $Diff(Q, e.sig)$. For this example, a better bound is $Diff(Q, e.sig) + (d - Ovr(e.sig, Q))$. We plan to study such search optimizations, using domain properties or statistics from the indexed data.

References

[1] C. C. Aggarwal, J. L. Wolf, and P. S. Yu. A New Method for Similarity Indexing of Market Basket Data. *SIGMOD Conference*, pages 407–418. 1999.

[2] R. Agrawal and R. Srikant. Fast Algorithms for Mining Association Rules in Large Databases. *VLDB Conference*, pages 487–499. 1994.

[3] K. S. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft. When Is “Nearest Neighbor” Meaningful? *International Conference on Database Theory*, pages 217–235. 1999.

[4] T. Brinkhoff, H.-P. Kriegel, and B. Seeger. Efficient Processing of Spatial Joins Using R-Trees. *SIGMOD Conference*, pages 237–246. 1993.

[5] A. Corral, Y. Manolopoulos, Y. Theodoridis, and M. Vassilakopoulos. Closest Pair Queries in Spatial Databases. *SIGMOD Conference*, pages 189–200. 2000.

[6] A. P. de Vries, N. Mamoulis, N. Nes, and M. Kersten. Efficient k-NN Search on Vertically Decomposed Data. *SIGMOD Conference*, pages 322–333. 2002.

[7] U. Deppisch. S-Tree: A Dynamic Balanced Signature Index for Office Retrieval. *ACM SIGIR Conference*, pages 77–87. 1986.

[8] V. Gaede and O. Günther. Multidimensional Access Methods. *ACM Computing Surveys*, 30(2):170–231, 1998.

[9] V. Ganti, J. Gehrke, and R. Ramakrishnan. CACTUS–clustering categorical data using summaries. *ACM SIGKDD Conference on Knowledge Discovery and Data mining*, pages 73–83. 1999.

[10] D. Gibson, J. M. Kleinberg, and P. Raghavan. Clustering Categorical Data: An Approach Based on Dynamical Systems. *VLDB Conference*, pages 311–322. 1998.

[11] A. Gionis, D. Gunopulos, and N. Koudas. Efficient and Tunable Similar Set Retrieval. *SIGMOD Conference*. 2001.

[12] S. Guha, R. Rastogi, and K. Shim. ROCK: A Robust Clustering Algorithm for Categorical Attributes. *International Conference on Data Engineering*, pages 512–521. 1999.

[13] A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. *SIGMOD Conference*, pages 47–57. 1984.

[14] S. Helmer and G. Moerkotte. A Study of Four Index Structures for Set-Valued Attributes of Low Cardinality. *Technical Report, University of Mannheim*, number 2/99. 1999.

[15] G. R. Hjaltason and H. Samet. Distance Browsing in Spatial Databases. *TODS*, 24(2):265–318, 1999.

[16] A. K. Jain and R. C. Dubes. *Algorithms for Clustering Data*. Prentice-Hall, 1988.

[17] I. Kamel and C. Faloutsos. Hilbert R-tree: An Improved R-tree using Fractals. *VLDB Conference*, pages 500–509. 1994.

[18] F. Korn, N. Sidiropoulos, C. Faloutsos, E. Siegel, and Z. Protopapas. Fast Nearest Neighbor Search in Medical Image Databases. *VLDB Conference*, pages 215–226. 1996.

[19] N. Koudas and K. C. Sevcik. High Dimensional Similarity Joins: Algorithms and Performance Evaluation. *International Conference on Data Engineering*, pages 466–475. 1998.

[20] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest Neighbor Queries. *SIGMOD Conference*, pages 71–79. 1995.

[21] Y. Sakurai, M. Yoshikawa, S. Uemura, and H. Kojima. The A-tree: An Index Structure for High-Dimensional Spaces Using Relative Approximation. *VLDB Conference*, pages 516–526. 2000.

[22] The UCI KDD Archive. http://kdd.ics.uci.edu.

[23] R. Weber, H.-J. Schek, and S. Blott. A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces. *VLDB Conference*, pages 194–205. 1998.