Localized Signature Table: Fast Similarity Search on Transaction Data

Qiang Jing
qiang.jing@nus.edu.sg
Rui Yang
yangrui@comp.nus.edu.sg
Panos Kalnis
kalnis@comp.nus.edu.sg
Anthony K. H. Tung
atung@comp.nus.edu.sg

Department of Computer Science
School of Computing
National University of Singapore
Singapore, 117543

ABSTRACT

Recently, techniques for supporting efficient similarity search over huge transaction datasets have emerged as an important research area. Several indexing schemes have been proposed towards this direction. Typically, these schemes provide a tradeoff between searching efficiency and indexing overhead in terms of space.

In this paper, we propose a novel indexing scheme for similarity search on transaction data. Based on well-studied clustering techniques, we develop a construction algorithm for the proposed index and a branch-and-bound searching strategy for answering similarity search. Unlike previous techniques, our indexing scheme exhibits high search efficiency and low space requirements by trading-off the pre-computation time. This behavior is ideal for applications with low update but high read volume (*e.g.*, data warehousing, collaborative filtering, *etc.*). Moreover, our experimental results illustrate that our method is robust to the varying characteristics of the datasets.

Categories and Subject Descriptors

H.3.1 [**Information Storage and Retrieval**]: Content Analysis and Indexing—*Indexing methods*; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*Search process*

General Terms

Algorithms, Design, Experimentation, Performance

Keywords

Transaction Data, Similarity Search, Indexing, Data Mining

CIKM'04, November 8–13, 2004, Washington, DC, USA.

1. INTRODUCTION

Transaction data are records consisting of a set of items from a *universal set*. Two typical examples of the transaction data are market basket datasets and text document databases which are summarized by a set of key words. With the development of web and financial applications, transaction datasets are becoming increasingly larger. Thus data analysis tools, such as clustering techniques, are needed. Similarity search is one of the core operations for performing data analysis. Although similarity search on numerical data has been studied extensively [15, 19, 16], such query techniques for transaction data just emerged in recent years [1, 12, 13].

Typically, similarity search refers to (i) *nearest-neighbor queries*, which return the closest data records for a given query and (ii) *range queries*, which return all data records within a given range to the query. A practical example of similarity search on transaction data is the recommendation system for e-Commerce sites [13]. In the e-Commerce sites, each transaction is associated with a list of goods purchased. For example, the set {*tennis racket, jogging shoes, walkman*} is a transaction. This is a subset of the items that can be purchased online. Given a customer purchase history, the system might want to look for the most similar transaction in the database and recommend a product to the customer based on the retrieved results.

In this paper we consider the problem of similarity search in transaction datasets. Previous work [1, 12, 13] has shown that extending the common methods from numerical, ordered domains is not straightforward. The reasons are: (i) Data domains of this problem do not have a natural order; (ii) The dimensionality of the transactions is very large while the datasets are very sparse. Therefore, traditional methods like the R-tree are not applicable. Our contribution in this paper is as follow:

- We propose a new signature based indexing structure for transaction datasets, named *Localized Signature Table(LST)* which is generated by performing both column and row clustering on the datasets. By doing so, transactions can be represented by signatures that lead to far less loss in information.
- Based on LST, a lower bound on the distance between a query and a group of transactions can be derived. We then developed a branch and bound method which make use of the lower bound to perform effective pruning for range and k-NN queries.

- We conduct extensive experiments on synthetic and real datasets to validate the effectiveness of our method. The results show that our scheme is superior to existing indexing techniques in terms of query performance and indexing overhead at the expense of increased construction cost, rendering it beneficial for applications with infrequent dataset updates.

The rest of the paper is organized as follows: In Section 2, we provide the background and an overview of the related work. Section 3 presents the proposed Localized Signature Tables indexing scheme. In Section 4 we show how to process similarity search by employing our indexing scheme. Section 5 includes the experimental evaluation of our approach and the comparison with several existing methods. Finally, Section 6 concludes the paper and discusses the future research issues.

2. BACKGROUND AND RELATED WORK

Given a set of items I=$\{i_1, i_2, \ldots, i_d\}$, a *transaction* t is a subset of I, where I is referred to as the *domain* of this transaction. A *transaction database* or *transaction dataset* $\mathcal{T}$ is a set of transactions with regard to the same domain I, *i.e.,* $\mathcal{T}$=$\{t_1, t_2, \ldots, t_N | t_u \subseteq I, 1 \leq u \leq N\}$. Let d be the cardinality of I. Transaction t_u can be represented by a d-bit vector $\overrightarrow{t_u}$=$\langle t_u[1], t_u[2], \ldots, t_u[d]\rangle$, in which the jth bit $t_u[j]$=1 if the jth item $i_j \in t_u$, else $t_u[j]$=0 if $i_j \notin t_u$.

In this paper, we will use the *Hamming distance* as our similarity measure of transaction data as it is the most commonly used metric; and it is easy to extend our methods to handle queries based on other metrics that are increasing functions of the number of matches and decreasing functions of Hamming distance as well, like the *Jaccard coefficient* and the *cosine* function [10]. Here, we give the definition of Hamming distance first. Let t_1 and t_2 be two transactions. The *Hamming distance* between t_1 and t_2 is the number of items in which t_1 and t_2 differ, *i.e.,* $HD(t_1, t_2)$=$|t_1 - t_2| + |t_2 - t_1|$.

For years, similarity search for numeric and ordered data has attracted much research interest, especially in the high dimensional spaces [9, 19, 16]. Normally, the data space is indexed by a tree-structured indices and subsequently a *branch-and-bound* search algorithm can greatly prune the searching area. However their performances degrade rapidly due to the presence of *dimensionality curse* [3]. Therefore the same method cannot be scaled to deal with transaction data for the huge number of dimensions. Inverted index structure together with list merging techniques is commonly used in information retrieval discipline to process similarity search. Unfortunately, it cannot support range queries. Moreover, the performance degrades when the density of the data increases.

The relevant problem of similarity search for sets and categorical data has also been studied in [8]. The proposed method first embeds the set data into binary vectors in Hamming space by using Min Hashing technique and error correcting codes; and then two data structures (*Similarity Filter Index* and *Dissimilarity Filter Index*) are used to construct an index. The optimization of the index is also brought forward to improve the accuracy. However, the method does not exploit the correlations among items which are extraordinarily important for transaction data. At the same time, it only retrieves the approximate answer to the query, while we mainly focus on exact results in this paper. Finally, only range queries are considered in [8] while another critical type of *k-Nearest-Neighbor* queries are ignored.

Recently, several indexing schemes on transaction datasets were brought forward and the similarity optimistic bounds derived by the indexing entries and the queries are used to facilitate the similarity search [1, 13, 12]. Aggarwal *et al* [1] proposed an index method named *Signature Table*. For a number K, the Signature Table method first partitions the items of I into K subsets G_j, $1 \leq j \leq K$, by using a variant of the *Single Linkage Clustering* algorithm [17]. The distance between any two items i_1 and i_2 is defined as the inverse of the *support* of the 2-itemset $\{i_1, i_2\}$ in $\mathcal{T}$. Next, by employing a user-defined constant r (called *Activation Threshold*), each transaction t can be mapped to a *Super-coordinate*, *i.e.,* a K-bit string $\{s_1, s_2, \ldots, s_K\}$, where s_j=1 ($1 \leq j \leq K$) if $|t \cap G_j| \geq r$, otherwise s_j=0. There are 2^K possible super-coordinates which are used as indexing entries.

Symbols	Size	Definitions
B	b	correlated blocks
$\overrightarrow{cg}$	$1 \times d$	grouping vector
c		Column
CG	K	Set of grouping vectors
G	K	column groups
I	d	Universal Item Set
i		Item
Sig	$1 \times K$	Signature
$Sig[j]$		The jth Bit of Sig
$\mathcal{T}$	N	Transaction Dataset
t		Transaction
TH	$b \times K$	Threshold Matrix

Table 1: Summary of Symbols

Other than the table structure, Mamoulis *et al.*[12] and Nanopoulos *et al.*[13] proposed tree structure indices based on signatures for the similarity search problem. They both use an R-tree based indexing structure, called *Signature Tree*, but with different ways to generate signatures. Each indexing entry in the tree nodes consists of one bitmap signature and one pointer to its child node. The bitmap signatures in the non-leaf nodes are formed by super-imposing (*i.e.,* bit-wise *OR* operation) the signatures in all the entries of the corresponding child nodes. In the leaf nodes, the signature of each entry is obtained directly from the corresponding transaction in the database and the child node pointer points to the position of the transaction on the disk.

To generate the bitmap signatures at leaf level, [12] uses the d-dimensional bitmap vector of the transaction as its signature, while in [13] a hash function is used to map the d-dimension space to a lower dimensional signature space similar to the *Super-coordinates* in signature table method.

Signature-based methods are also common in the Information Retrieval (IR) area. The Latent Semantic Indexing (LSI) method [5], for instance, applies Singular Value Decomposition to identify latent patterns. The goal of such methods is to decrease the dimensionality of the problem. Our proposal, on the other hand, aims at indexing efficiently a set of data with a given dimensionality. In this sense, our work is complementary to the IR methods.

3. LOCALIZED SIGNATURE TABLE

3.1 Global vs. Local Signature

In the following discussion, we will treat the transaction dataset as a bitmap table with one item per column. Signature table method partitions the set of all columns into K subsets. Hereafter, we refer to this partitioning operation as *column grouping*, and refer to each subset of columns as *column group*. The query efficiency of the signature table method is sensitive to the accuracy of the estimation of a lower bound for the similarity measure and it produces good pruning power if the transaction dataset is *globally correlated*. However, this is not the case in practice since datasets in modern

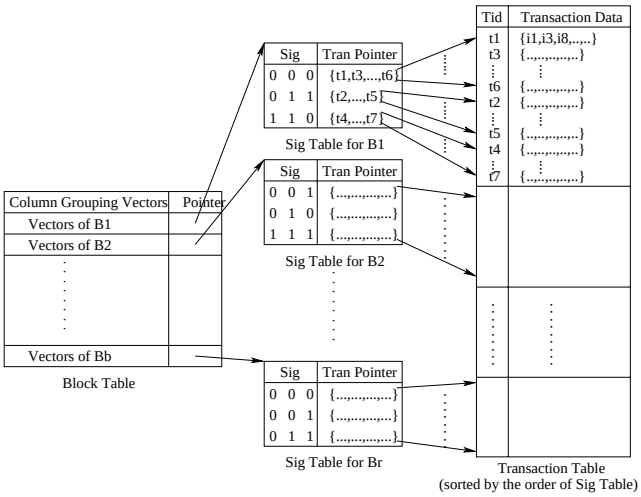

Figure 1: Index Structure of LST

applications tend to be large and different subsets of the data may have different data distributions. Using a single column grouping results in a significant loss of information in the signature table.

Even though the transactions in the database may not have the coherent global correlation, transactions in some subset can be highly correlated (*i.e., locally correlated*). We refer to these subsets as *correlated blocks* or simply *blocks*. In our method, we exploit these local correlations among transactions and perform column grouping on each block individually to maximize *precision*. However, this poses a cyclic problem: The block partition directly depends on the awareness of the local correlations, while local correlations can be exploited only after the blocks are ready. An analogical case of dimension reduction can be found in [4].

In our method, we break this cycle by an initial guess of the blocks. In the first step, an algorithm [14] for clustering high dimensional and sparse binary data is employed to initialize the blocks. Next, the columns in each block are clustered using an agglomerative hierarchical clustering algorithm. In the third step, the transactions are re-assigned to ensure that each block contains the most correlated transactions. Based on these blocks, we build the *Localized Signature Table* (LST in short) index. Figure 1 shows the structure of LST. The first tier is the *correlated block table*. There are two components for each entry: the *column grouping vector* of each block and the pointer to its signature table which consists the second tier. The third tier is a sorted transaction data table.

3.2 Initializing Correlated Blocks

Our initial blocks are identified by an *EM*-based clustering algorithm customized for high dimensional and sparse binary data. *EM* [6] is a well established clustering algorithm in the Statistics community. It assumes that the data $\overrightarrow{t}$ can be modelled by the mixture of multi-variate normal distributions. Let $P(\overrightarrow{t})$ be the mixture model probability function and b be the number of clusters. The clustering process attempts to maximize the *log-likelihood* L:

$$L = \sum_{\forall \overrightarrow{t} \in \mathcal{T}} log(P(\overrightarrow{t})) = \sum_{\forall \overrightarrow{t} \in \mathcal{T}} log(\sum_{j=1}^{b} w_j P(\overrightarrow{t}|j)) \quad (1)$$

where $\mathcal{T}$ denotes the entire dataset; $P(\overrightarrow{t}|j)$ is the probability[1] that $\overrightarrow{t}$ belongs to the jth cluster and w_j is the weight factor.

[1]The *EM* algorithm assumes the probability follows the normal distribution.

Recall that transaction data are seen as d-dimensional binary vectors. Ordeonez *et al.* [14] introduced an EM-based algorithm for high dimensional and sparse binary datasets. We employ this algorithm for the initial clustering of the transaction data. Our choice was based on the properties of the algorithm, *i.e.*, its suitable initialization, sufficient statistics, covariance matrix regularization, sparse distance computation and fast learning steps which make it quite efficient for the clustering of large transaction datasets.

3.3 Constructing Column Groups Locally

After the highly correlated transactions are clustered into blocks, we identify the column grouping for each block. These column groupings will be used for the reallocation of transactions and signature table construction. In this section, we first analyze the error due to some column grouping and the corresponding signature representation. Based on this, we define the "goodness" measure for the column grouping, which is used for clustering.

Given a column grouping, we can use a set of K *grouping vectors* to represent it: a grouping vector $\overrightarrow{cg_j}$ is a d-bit vector, in which each set bit indicates that the corresponding column belongs to the jth *column group*; a *column grouping* G=$G_1, G_2, \cdots, G_K$ can be expressed as a complete set of *mutually exclusive* grouping vectors on the set of all columns. For example, consider the following three column groups G_1={1,3,6,7,9,10}, G_2={4,5,11,12}, G_3={2,8}; the corresponding grouping vectors are shown in Table 2(a).

Based on the grouping vectors, we compute the *signatures* of the transactions. The formal definition is given below:

Definition 3.1 (Signature) Given a d-dimensional binary string t and a set of d-dimensional grouping vectors CG=$\{\overrightarrow{cg_1}, \ldots, \overrightarrow{cg_K}\}$, we define the *signature* function $\overrightarrow{Sig}(CG, t)$ as follows:

$$\overrightarrow{Sig}(CG, t) = \langle sig_1, sig_2, \ldots, sig_K \rangle$$
$$sig_j = \sum(\overrightarrow{cg_j} \,\&\, t) \quad (2)$$

A bitwise-*AND* operation ($\&$) is performed between t and the grouping vector $\overrightarrow{cg_j}$, resulting in a d-bit vector; next all d bits are summed up by *OR*ing ($\sum$) them together, to compute the jth bit of the signature.

bit	1	2	3	4	5	6	7	8	9	10	11	12
cg_1	1	0	1	0	0	1	1	0	1	1	0	0
cg_2	0	0	0	1	1	0	0	0	0	0	1	1
cg_3	0	1	0	0	0	0	0	1	0	0	0	0

(a) grouping vectors

tid	Data	Sig
t_1	0001 1000 0011	010
t_2	0100 1001 0010	011
t_3	0101 0001 0001	011
t_4	0100 0001 0000	001

(b) Signature Table

Table 2: Example of column grouping

Table 2(b) presents four transactions and their signatures generated according to the grouping vectors of Table 2(a). Given the grouping vector and the signature, we can reconstruct the transaction data:

Definition 3.2 (Reconstruction Data) Given a K-bit signature Sig and a set of d-dimensional grouping vectors CG=$\{\overrightarrow{cg_1}, \ldots, \overrightarrow{cg_K}\}$, we define the *reconstruction data* $\overrightarrow{RD}(Sig, CG)$ as follows:

$$\overrightarrow{RD}(Sig, CG) = \overrightarrow{\sum_{j=1}^{K}}(\overrightarrow{cg_j} \times sig_j) \quad (3)$$

tid	Original Data Grouped by G											
	c_1	c_3	c_6	c_7	c_9	c_{10}	c_4	c_5	c_{11}	c_{12}	c_2	c_8
t_1	0	0	0	0	0	0	1	1	1	1	0	0
t_2	0	0	0	0	0	0	*0*	1	1	*0*	1	1
t_3	0	0	0	0	0	0	1	*0*	*0*	1	1	1
t_4	0	0	0	0	0	0	0	0	0	0	1	1
CoverSig			Reconstructed Data Grouped by G									
sig_1	sig_2	sig_3	c'_1	c'_3	c'_6	c'_7	c'_9	c'_{10}	c'_4	c'_5	c'_{11}	c'_{12}	c'_2	c'_8
0	1	0	0	0	0	0	0	0	1	1	1	1	0	0
0	*1*	*1*	0	0	0	0	0	0	*1*	1	1	*1*	1	1
0	*1*	*1*	0	0	0	0	0	0	1	*1*	*1*	1	1	1
0	0	1	0	0	0	0	0	0	0	0	0	0	1	1

Table 3: Errors between Original and Reconstructed Data (in *Italics*)

where sig_j is the jth bit of the signature Sig, and ($\times$) represents the multiplication of a vector with a scalar. Since there are K bits in Sig, we get K vectors which are subsequently bitwise-*OR*ed ($\overrightarrow{\sum}$) together, to produce a d-bit reconstructed binary string.

The signature representation is a lossy compression of the original information. Consequently, the reconstructed data may differ from the original. For example, in Table 3, the original and the reconstructed data are presented (Columns are organized according to the column grouping CG). By comparing the reconstructed strings with the original ones, it is obvious that t_1 and t_4's signatures are perfect, but t_2 and t_3's signatures both lose 2 bits of information (the problematic bits are highlighted in *Italics*). The error of a grouping G with respect to a binary string t is formally defined as:

Definition 3.3 (Signature Error) *Signature Error* of CG on a binary string t, $SigErr(CG, t)$, is the Hamming distance between t and the reconstructed string computed from the signature of t:

$$SigErr(CG, t) = HD(t,\ \overline{RD}(\overrightarrow{Sig}(CG,\ t),\ CG)) \qquad (4)$$

A good column grouping for the block should minimize the reconstruction error. Assuming that there are m transactions in one block, we can give a criterion function E_l, which should be minimized by the column grouping:

$$E_l \quad = \quad \sum_{u=1}^{m} SigErr(CG, t_u) \qquad (5)$$

If each column of the binary matrix in Table 3 is viewed as a binary string, our grouping problem is analogous to a binary string clustering problem given the criterion function. In the column-wise view, we define the *cover signature* of each column group in one block by bitwise-*OR*ing all columns assigned in this column group:

$$\overrightarrow{CoverSig}(\overrightarrow{G_j}) = \overrightarrow{\sum} c_u,\ \forall c_u \in G_j \qquad (6)$$

According to the definition of data reconstruction, each bit in the reconstruction data should be the same as the corresponding *cover signature* bit of the associated *column group*; The criterion function E_l can be restated as:

$$E_l = \sum_{j=1}^{K} \sum_{\forall c_u \in G_j} HD(\ \overrightarrow{CoverSig}(G_j),\ c_u) \qquad (7)$$

However, E_l alone is not a good criterion, since for transaction data, the set bits are more important than reset bits. For example, assume that there are two pairs of transactions. One pair contains only one item each and the difference between them is two, while another pair which have 100 items each, also differ in two items. It is obvious that the latter pair should be ranked higher in terms of similarity. The same rationale applies to two groups of data. Thus, for the distance of two clusters, the factor of the number of set bits in the binary strings should be taken into consideration. Here we introduce the concept of *expected Hamming distance* into the criterion function, based on a probability model. Assuming that the uth column c_u in G_j has s_u set bits, the probability that one bit in column c_u is reset, is $\frac{m-s_u}{m}$. Then the probability of one bit in the *cover signature* being reset is the probability that this bit is reset in all the columns on the group, *i.e.*, $\prod_{c_u \in G_j} \frac{m-s_u}{m}$. It follows that the probability of a bit to be set in the cover signature is $1 - \prod_{c_u \in G_j} \frac{m-s_u}{m}$. From the construction of the *cover signature*, we know that a difference between the column and the *cover signature*, (*i.e.*, the corresponding reconstructed column) occurs only when a reset bit is changed to a set bit in the *cover signature*. Let the number of set bits in column $c_u,\ 1 \le u \le d$ be kept in the array $sb[u]$. Then the *expected Hamming distance* for the jth Column Group is

$$D'_j \quad = \quad \left(1 - \prod_{\forall c_u \in G_j} \left(\frac{m - sb[u]}{m}\right)\right) \cdot \sum_{\forall c_u \in G_j} (m - sb[u]) \qquad (8)$$

Therefore, the normalized criterion function $\tilde{E}_l$ for some column grouping is:

$$\tilde{E}_l = \sum_{j=1}^{K} \frac{card[j] \cdot \sum_{\forall c_u \in G_j} HD(\ \overrightarrow{CoverSig}(G_j),\ c_u)}{D'_j} \qquad (9)$$

where $card[j]$ denotes the cardinality of G_j.

Given the definition of $\tilde{E}_l$, the problem is to partition columns of each block into K groups to minimize $\tilde{E}_l$. We propose an agglomerative hierarchical clustering algorithm. We use a measure similar to $\tilde{E}_l$ to determine the best pair of column groups to merge at each step of the algorithm. For a pair of column groups G_u and G_v, let G_w denote the resulting group G_u after merging with G_v, *i.e.*, G_w=$G_u \bigcup G_v$. Let $P'[j]$=$\prod_{\forall c_l \in G_j} \left(\frac{m-sb[l]}{m}\right)$ and $S'[j]$=$\sum_{\forall c_l \in G_j} (m - sb[l]), 1 \le j \le h$ (h is the number of current column groups: $K \le h \le d$). Then, the *goodness measurement* $g(G_u, G_v)$ for merging column group G_u and G_v is defined as:

$$g(G_u, G_v) = \frac{(1 - P'[u] \cdot P'[v]) \cdot (S'[u] + S'[v])}{\sum_{\forall c_l \in G_w} HD(\ \overrightarrow{CoverSig}(G_w),\ c_l)} \qquad (10)$$

The steps involved in the hierarchical clustering are described in Algorithm 1. Assume that the block contains m transactions. The algorithm accepts as input the set $\mathcal{C}$ of d binary vectors (columns) to be clustered and the number of desired column groups K. Initially, each column forms a separate column group and $P'[\,]$ and $S'[\,]$ are computed accordingly. For each current column group $G_u, 1 \le u \le h$, we build a local heap $q[u]$ (Each member of the heap has

Algorithm 1 Hierarchical Clustering for Columns

Input: A set of binary vectors $\mathcal{C}$=$\{c_1, c_2, \ldots, c_d\}$, the number of column groups K.
Output: A set of column groups G=$\{G_1, G_2, \ldots, G_K\}$, s.t. $\bigcup_{u=1}^{K} G_u$=$\mathcal{C}$ and $\forall u, v \in 1..K, u \neq v, G_u \bigcap G_v$=$\varnothing$.

1: $h \leftarrow d$
2: **for** $1 \leq u \leq d$ **do**
3: $sb[u] \leftarrow SetBitCount(c_u)$, $G_u \leftarrow c_u$
4: $P'[u] \leftarrow \frac{m-sb[u]}{m}$, $S'[u] \leftarrow m - sb[u]$
5: **for** $1 \leq u \leq d$ **do**
6: $q[u] \leftarrow buildLocalHeap(G, u)$
7: $H \leftarrow buildGlobalHeap(G, q)$
8: **while** $h > K$ **do**
9: $u \leftarrow H[0].id$, $v \leftarrow q[u][0].id$
10: dequeue(H, u), dequeue(H, v)
11: $G_w \leftarrow G_u \cup G_v$, insert G_w into G
12: $P'[w] \leftarrow P'[u] \times P'[v]$
13: $S'[w] \leftarrow S'[u] + S'[v]$
14: **for** each $j \in H$ and $j \neq u|v$ **do**
15: $k \leftarrow g(w, j)$
16: dequeue$(q[j], u)$, dequeue$(q[j], v)$
17: enqueue$(q[j], w, k)$, enqueue$(q[w], j, k)$
18: $updateHeap(H, j, q[j][0].key)$
19: enqueue$(H, w, q[w])$
20: deallocate $q[u]$ and $q[v]$, delete G_u and G_v from G
21: $h \leftarrow h - 1$
22: Return G

two fields: id=v and key=$g(G_u, G_v)$ (*i.e.*, the *goodness* of merging G_u and G_v)). The root element of it always refers to the current best column group to merge with G_u according to the goodness measurement. In addition, our algorithm also maintains a global heap H for all column groups. Elements in H store the information of the root element of the local heaps and the root of H itself stores the best pair of column groups to be merged at each step.

In each WHILE-loop iteration in the algorithm, G_u and G_v are merged to create a new column group G_w containing columns of $G_u \bigcup G_v$. This involves five steps:

1. Pop entries for G_u and G_v from H.
2. In every local heap which contains G_u or G_v, replace the elements of G_u and G_v with G_w and update their respective local heap accordingly to maintain the heap property.
3. Create a new local heap for G_w.
4. For a column group $G_j, 1 \leq j \leq h, j \neq u, v$, whenever the maximum $g(G_j, \cdot)$ changes, relocate G_j in H to reflect information relating to the new best column group for G_j.
5. Push the best column group to be merged with G_w into H.

The loop ends when the number of column groups in G reaches K.

3.4 Building the Localized Signature Table

After the column groupings of all blocks are obtained, each transaction t is reassigned to improve the correlated blocks discovered before: t is assigned to a block $B_j \in B$ whose column grouping G^j minimizes the error for t. When the transactions are re-assigned, the set of correlated blocks B is updated accordingly.

Instead of using a static activation threshold (as in Ref. [1]) to assign transaction t to a block, we compute the activation threshold for each column group dynamically (see Algorithm 2). The

Algorithm 2 Computing the Activation Threshold

Input: A correlated block B_j, the corresponding *column groups* G^j, and the number of column groups K.
Output: A $1 \times K$ integer vector $TH[j]$

1: **for** each grouping vector $\overrightarrow{cg_u} \in CG^j, 1 \leq u \leq K$ **do**
2: $\forall t_p \in B_j$ Let t'_p be a d-bit binary vector, s.t. $t'_p[h] \leftarrow \overrightarrow{cg_u}[h]$ & $t_p[h]$, $\forall h \in 1..d$
3: $TH[j][u] \leftarrow \min_{\forall t_p \in B_j}(SetBitCount(t'_p))$.
4: Return $TH[j]$.

threshold of a group G^j is the minimum number of common set bits between its grouping vector and the transactions in the block. The thresholds are stored in a $b \times K$ matrix $TH[j]$. With the dynamic threshold, we generate the signature table by using an algorithm similar to Ref. [1]. Then the items of all the transactions in each block B_j are partitioned into K subsets $G^j_1, G^j_2, \cdots, G^j_K$ according to the block's column grouping. For a transaction t, the corresponding bit in the signature is set only if $\sum_{\forall l \in G^j_u} t[l] \geq TH[j][u]$. In this way, we build a K bit *Localized Signature Table* LST as shown in Figure 1:

- The first tier is the *(correlated) block table* which stores the column groupings of all correlated blocks and pointers to the starting address of the corresponding signature table.
- The second tier consists of independent *signature tables* for each correlated blocks. Each tuple contains two fields: the *representative signature* and a pointer list of the transactions indexed by it.
- The third tier is a sorted transaction data table. To avoid frequent random accesses to the data, all transactions are sorted according to the order of their signatures. For each entry in the signature table, only the offset of the first transaction and the number of transactions indexed by it are stored in the *Pointers* field (*i.e.*, sparse index).

A significant issue for the table structure in both ST and LST is the scheme for the organization of transaction data on disk. In the case of skewed distribution of the indexed dataset, the required organization scheme presents similarities with the one for storing data records in traditional indices (*e.g.*, B^+-tree). Therefore, the approach of *postings list* [7] can be followed to improve query performance [13]. With such scheme, ST and LST can also handle dynamic data as SG-Tree. For more detailed information about this issue, as well as the time and space complexity analysis, please refer to the full version of this paper [11].

4. QUERY PROCESSING WITH LST

In this section, we describe the similarity search scheme on transaction data by using the branch-and-bound strategy. First, we will describe how LST can be used to compute a optimistic bound of the distance between the query and the transaction which is represented by a signature. Then we illustrate our solutions for two typical similarity searches: the k-NN and the range query.

4.1 Optimistic Bound Computation

In order to avoid searching all the partitions of the dataset, the lower bounds (*optimistic bounds*) of the distance between the query Q and the transactions in each partition should be determined first by their signatures and the column grouping of their respective correlated blocks.

Algorithm 3 *GetOptDist*($t_q, G^l, TH[l], Sig$)

Input:A query transaction t_q, the *signature* Sig, the *column groups* G^l of the block that Sig belongs to, and the threshold vector $TH[l]$ of G^l.
Output: The optimistic bound of Sig to t_q.

```
1: y_opt ← 0
2: for each G_j^l ∈ G^l do
3:    z ← |t_q ∩ G_j^l|
4:    if Sig[j] = 0 then
5:       y_opt ← y_opt + z
6:    else
7:       if Sig[j] = 1 then
8:          y_opt ← y_opt + max(0, TH[l][j] − z)
9: Return y_opt.
```

The optimistic bound is computed as in Algorithm 3. Let Sig be the signature in LST, and let the entry be associated with the block B_l. G^l and $TH[l]$ is the corresponding column grouping and threshold vector of B_l, respectively. G_j^l denotes the jth group of columns in G^l. In Function $GetOptDist$, the lower bound on the Hamming distance from a query to all transactions indexed by the signature entry, consists of the sum of all portions contributed by each column group G_j^l. Below we explain how the algorithm processes each of the column groups in G^l:

1. For each column group G_j^l, if its corresponding signature bit $Sig[j]$=0, it implies that every transaction indexed by that entry must have no common items with G_j^l. If the target transaction t_q has z=$|t_q \bigcap G_j^l|$ items in common with G_j^l, the Hamming distance from t_q to the transactions indexed by this entry is z. Thus, for each $Sig[j]$=0, we add z to the optimistic bound.

2. Otherwise, if the corresponding signature bit $Sig[j]$=1, we know from the threshold computation method that every transaction indexed by that entry must have no less than $TH[l][j]$ common items with G_j^l. If t_q has $z < TH[l][j]$ items in common with G_j^l. The Hamming Distance from t_q to the transactions indexed by this entry must be at least $TH[l][j] - z$; obviously, the optimistic bound cannot be negative. Thus, for each $Sig[j]$=1, we add $\max(0,\ TH[l][j] - z)$ to the optimistic bound.

We next discuss the correctness of the optimistic bound. By correctness we mean that the algorithm returns the same answers as a sequential scan of the entire dataset. We have proved that the optimistic bound computed by Algorithm 3, always bounds the distance between the transactions indexed by the entry and the query, ensuring no false dismissals. Let $OptDist(t_q, Sig)$ be the optimistic distance bound of a signature entry Sig in the signature table. Let $realDist(t_q, t_i)$ be the actual Hamming distance between t_q and a transaction t_i indexed by Sig. Then, Lemma 4.1 stands. For the limitation of space, we do not include the proof here. Interested readers can refer to [11].

Lemma 4.1 (Optimistic Bounding Lemma) *For any transaction t_i indexed by the LST, the estimated optimistic bound of the distance between the query t_q and the associated entry Sig of t_i is always a* lower-bound *of the actual Hamming distance between t_q and t_i:*

$$\begin{aligned} realDist(t_q, t_i) &\geq OptDist(t_q, Sig) \\ &= GetOptDist(t_q,\ G,\ TH,\ Sig) \end{aligned} \quad (11)$$

where G, TH are the corresponding column grouping and threshold vector of Sig.

Alike signature table method, the upper bound of number of matches between a query and transactions can be derived from the column grouping and the signature entries too. So our method can handle similarity search based on any metric which is an increasing function of number of matches and a decreasing function of Hamming distance.

4.2 Similarity Search

A k-NN query $\mathcal{Q}_k$=$\langle t_q, k, \mathcal{T}\rangle$ retrieves a set of k transactions $\mathcal{R}$ from Dataset $\mathcal{T}$, such that for any two transactions $t \in \mathcal{R}, t' \notin \mathcal{R}, HD(t_q, t) \leq HD(t_q, t')$. Our search algorithm is similar to that of Ref. [1]. After the optimistic bounds of all entries in the LST are computed, the index entries are re-sorted in increasing order of the optimistic bound so that the entries which are most likely to represent the most similar transactions to the target are visited first. Then the branch and bound searching strategy is used. The blocks in the signature table are scanned one by one according to the order of their signature entries. If the optimistic bound is less or equal to the current pessimistic bound, then all the transactions indexed by the entry are read into the memory and the actual Hamming distances between them and t_q are computed. The computed Hamming distance values are used to update the current pessimistic bound and the current partial results. If the optimistic bound of the next element in the sorted index entries is greater than the current pessimistic bound, the search is terminated.

A range query $\mathcal{Q}$=$\langle t_q, \varepsilon, \mathcal{T}\rangle$ retrieves a set of transactions $\mathcal{R}$ from Dataset $\mathcal{T}$, such that $\forall t \in \mathcal{R}, HD(t_q, t) \leq \varepsilon$; and $\forall t' \notin \mathcal{R}, HD(t_q, t') > \varepsilon$. Similar to k-NN query processing, we first compute the optimistic bounds for all the entries in LST and sort them in ascending order. The blocks in LST are scanned in the same way as that of K-NN search algorithm, in order to compare the bound of each entry with the distance ε. Only if the lower bound of an entry is less than ε, it is necessary to compute the actual Hamming distances between t_q and all the transactions indexed by that entry. If the range constraint is satisfied, the transaction is added to the result set. As soon as the lower bound of an entry is greater than ε, the search is terminated according to the branch and bound strategy.

5. PERFORMANCE STUDY

In this section, we compare performance of LST with the signature tree and the signature table (denoted as SG-Tree and ST henceforth). Through experiments, we also show that sampling techniques can be used to improve the index construction efficiency. Finally, we show how LST performance depends on the number of local blocks b and of the column groups K. All experiments were conducted on a DELL workstation with one Intel Pentium IV 2.4GHz CPU and 1GB of RAM. The operating system was Redhat Linux 9 with the GNU C++ v.3.2.2 compiler. We implemented the proposed algorithms for the LST index in C++, and revised the C source code of SG-Tree and ST provided by the authors of [12] in C++.

We use both synthetic and real data in our experiments. Synthetic datasets are generated by a commonly used data generating algorithm for market basket data mining [2]. We use the TxIyDz naming convention for the synthetic datasets: x, y and z denote the mean size of the transactions, the mean size of the potentially large itemsets, and the number of transactions contained in the dataset respectively. We adopt the parameters from [1, 12] to generate the datasets: the number of items is 1000, and the number of maxi-

mal itemsets is 2000. The real life dataset is the "Census-Income Database", obtained from [18]. There are 199523 instances in the data file. The Census raw data is cleaned and transformed into the desired format according to the methods mentioned in [12].

Commonly in the experiments of other papers, queries are randomly drawn from the indexed dataset [13]. In order to examine the performance of our method in different query environments, we adopt a data corruption algorithm on the selected data. Each item in a tuple of a query transaction has probability p to be changed to another randomly chosen item. In all our experiments, we process queries of three noise levels: p=10%, 20% and 50%, to simulate different searches anchoring from very close to relatively far data areas. Each of them contains 100 queries. We present the average values of the execution of all query sets.

Recall and *Precision* are two primary measurements for evaluating the performance of the indices for similarity search. In our case, the result set is always complete (*i.e.*, not approximate), rendering 100% Recall. Therefore, the most important measure is Precision. Here we adopt the *Percentage of Accessed Data*, defined as

$$\left(\frac{|True\ Positive| + |False\ Positive|}{|Dataset|} \right) \times 100\% \qquad (12)$$

for measuring the performance. This is an alternative representation of precision and was also used in Ref. [12].

5.1 Sensitivity Test

In the first set of experiments, we carry out a series of sensitivity analysis of the different indexing structures to the parameters of the datasets. From the discussion in [1] and [12], we conclude that the signature table performance depends on the number of column groups K, and the activation threshold r. Here, we adopt the same default settings of K=15 and r=1, as in the previous work. For LST, the number of local blocks b and of the column groups K, affects the performance too. We chose the parameters that comparable with ST. And we constructed two versions of LST for each experiment, by setting b to be 100 and 1000 for comparison. Figures 2 through 5 show the relative performance of the methods for various parameter settings.

Figure 2 illustrates the performance of the indices when the mean length of the transactions T varies, with the mean length of maximal itemsets and the size of dataset fixed (I=6, D=200k). It is clear that with larger T, the performance of all indices degrades. This is due to the fact that the distance to the nearest neighbor usually increases when T increases and I is constant; therefore, more entries of the indices must be visited. As T increases, the LST indices outperform the others, managing to access fewer transactions. The CPU costs of the LST indices are 50% to 80% compared to the other two methods, independently of the value of T.

Figure 3 shows the percentage of accessed data and CPU cost for T=30 as the size of maximum itemsets I increases. This increase generates datasets in which the transactions are better clustered. Thus they have smaller average distances and favor all indexing structures. In all cases, the LST (b=1000) index has the best performance, especially in the datasets with large I values. These experiments show that LST can adapt well to varying data skew.

In the third set of experiments we increase the mean transaction length T together with the mean maximum itemset length I but fix the ratio T/I to 0.6 . The results are shown in Figure 4. The rationale is to test the robustness of the indexing methods against the dimensionality of the problem, when the data skew remains constant. Clearly, the ST index fails to index the long transactions well and the data accesses grow quickly when the transaction length increases; on the other hand, the SG-Tree and LST indices are much more robust to the transaction length while the LST indices outperform SG-Tree in terms of percentage of data accessed and CPU cost, when the transaction length increases.

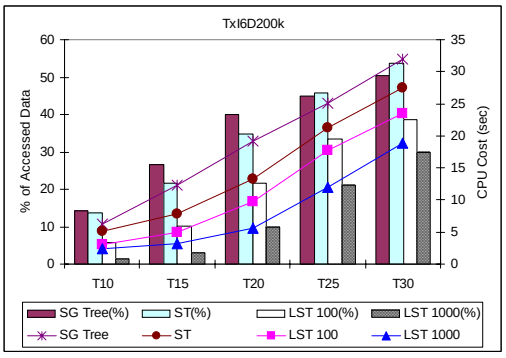

Figure 2: Sensitivity to Varying T

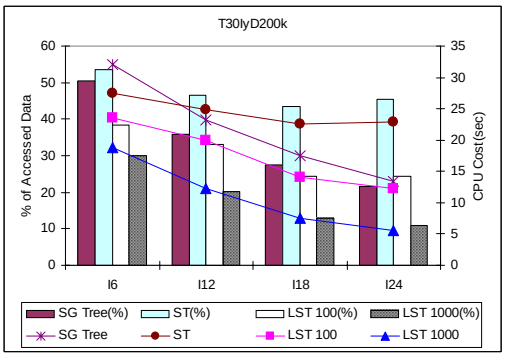

Figure 3: Sensitivity to Varying I

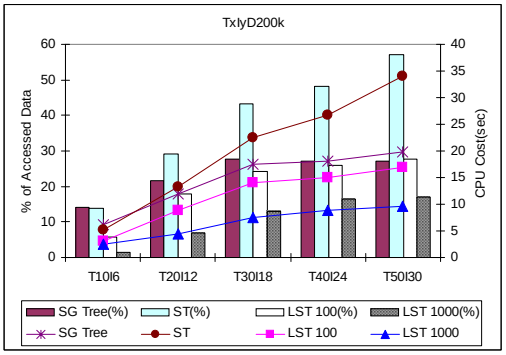

Figure 4: Sensitivity to Varying {T,I}

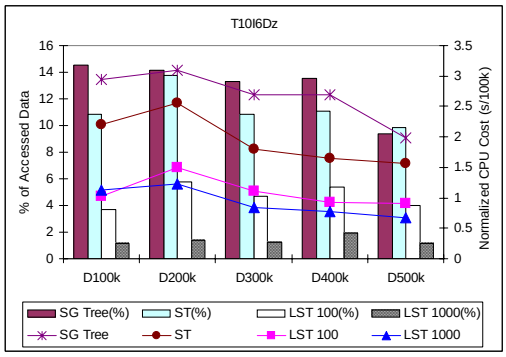

Figure 5: Sensitivity to Varying D

We also tested the scalability of the indexing structures to the database size, by setting all other parameters to default {T=10, I=6} and increasing the dataset cardinality D from 100k to 500k. Figure 5 shows the results, while the CPU costs are normalized by

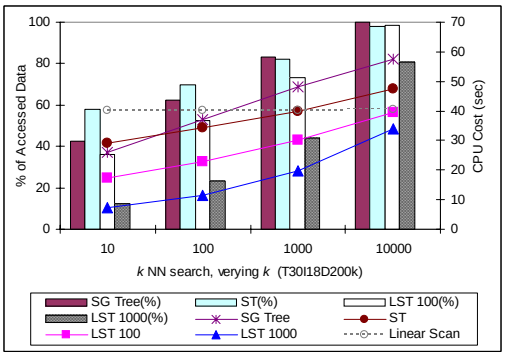

Figure 6: k-NN Searches on Synthetic Dataset

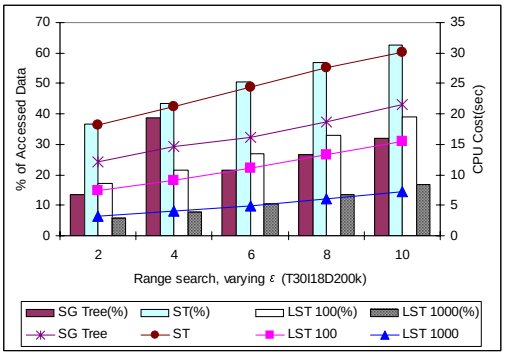

Figure 7: Range Searches on Synthetic Dataset

D. Apparently, ST and LST both outperform SG-Tree in terms of the CPU cost; especially the LST indices always have less than half of the CPU cost compared to ST or the SG-Tree. LST also accesses the least data for all database sizes.

5.2 k-NN and Range Queries

Figure 6 shows the performance of the indices for k-NN searches on the synthetic dataset T30I18D200k, for various values of k. For small values of k, all structures have limit data accesses and incur low CPU costs. LST however still outperforms the others. Note that, when k is larger than 1000, all indices except LST (b=1000) have to explore the whole dataset. We also use the brute force method to search sequentially for the result (the dashed line shown in Figure 6). In the case of large k (k >10000), all indices are less efficient than the simple linear scan. Nevertheless, the LST index is less sensitive to this effect and its performance degenerates at the smallest pace.

Figure 7 shows the results of range queries; ε, the distance threshold, varies from 2 to 10. As the value of ε increases, the searching sphere in the hyper data space becomes larger. Observe that the relative performances of the indices are almost the same as that of the k-NN queries.

We also performed the same series of experiments for the real dataset. Figures 8 depict the results for the range queries. The result are similar to these of the synthetic datasets. Notice that LST is especially suitable for real-life data; this is due to the fact that LST can better utilize high data skew which is often observed in practice. k-NN query results on the real dataset are consistent to that of the range queries.

Typically LST uses more memory than ST, since there are separate signature tables for each block. For fairness, here we compare the performance of ST when it is allowed to use the same amount of memory as LST. To achieve this, we increase the number of column groups for ST. In Figure 9, we present the results for k-NN queries on the Census dataset; the performance in the other cases is

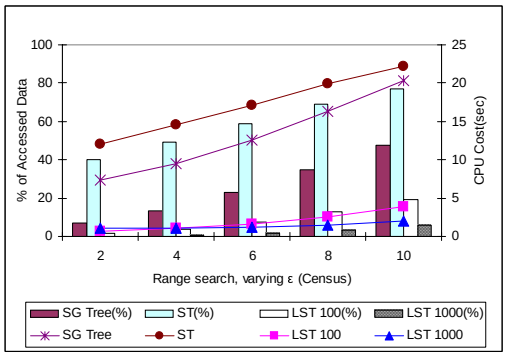

Figure 8: Range Searches on Real Dataset

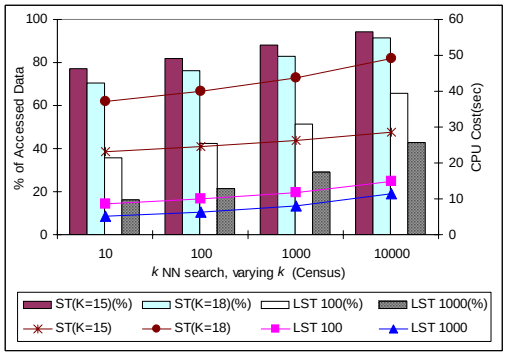

Figure 9: k-NN Queries Using the Same Amount of Memory on Real Dataset

similar. ST with K=15 already uses the same amount of memory as LST (b=100). To achieve the same for LST (b=1000), we set K=18 for ST.

In terms of percentage of data accessed, ST(K=18) performs better than ST(K=15). This happens because, with more column groups, ST can construct better clusters for the columns. However, its performance is still worse than the LST methods. Additionally, ST(K=18) needs more CPU time to find the results. This is caused by the sorting of the larger signature table (*i.e.*, 2^{18} entries).

5.3 Comparison of the Disk I/Os

This experiment compares the behavior of the ST and LST methods, in terms of the number of disk I/Os. We only show the results of k-NN (k=100) queries on the Census dataset in Figure 10, since other experiments with various parameters are consistent. The cache size is defined as the percentage of all disk pages of the dataset, while the page size was set to 1KB.

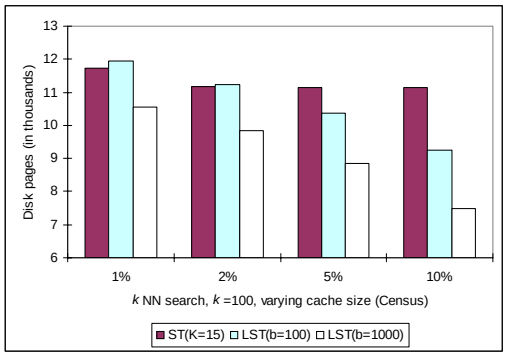

Figure 10: Disk I/O Comparisons

For small cache sizes (*i.e.* 1% and 2% of the dataset), the performance of LST with 1000 blocks is somewhat better than ST, while

LST with 100 blocks performs slightly worse. However, when the cache size increases, LST with 1000 and 100 blocks both outperform ST significantly. Compared to the results of the percentage of data accessed in the previous experiments, LST has smaller advantage against ST in terms of disk I/Os. This is due to relatively fewer data records being indexed into one signature entry in LST, thus more frequent data retrievals and disk page requests are incurred.

We employed a generic LRU cache in our experiments. A customized replacement algorithm which would consider the structure of LST, is expected to further improve the performance. The development of such an algorithm is part of our on-going work.

5.4 Comparison of the Indexing Overhead

Dataset	SG-Tree	ST	LST (100)	LST (1000)
Syn	94.2	0.8	1.9	3.9
Census	67.8	1.4	1.4	4.5

(a) Index Size (% of the dataset size)

Dataset	SG-Tree	ST	LST (100)	LST (1000)
Syn	69	17	853	2537
Census	54	7	194	1623

(b) Construction Time (sec)

Table 4: Indexing Overhead Comparisons

In this set of experiments, we evaluate the overhead of the structures in terms of the CPU time consumed during the index construction and the size of the resulting index.

The overhead is evaluated for the real-life dataset $Census$ and a synthetic dataset (T30I18D200k, shown in Table 4 as Syn). It is clear from Table 4, that ST outperforms the others in both construction time and index size, due to its simple construction algorithm and succinct data structure. Nevertheless, ST exhibits the worst query performance in most cases. The SG-Tree achieves a moderate construction time but its size is much larger than the others, because of the tree structure and the fact that the SG-Tree stores the entire bit strings of the transactions. The LST indices require the longest construction time. However, their sizes are small and they are very efficient in terms of query performance. According to the complexity analysis in [11], the typical running time of the LST construction algorithm is linear to the dataset cardinality. Therefore, they are more suitable in practice for very large datasets with heavy query load but infrequent updates (*e.g.*, data warehouses).

5.5 Effect of Random Sampling

The efficiency of the LST construction algorithm can also be improved by random sampling. When building the index, the *row clustering* and *column clustering* are first applied on a random sample of the dataset, while the transaction reallocation is done on the whole dataset.

We design a series of experiments for NN queries, in order to study the effect of the sampling ratio on the performance of the resulting indices. The experiments are performed on a synthetic dataset (T30I18D200k) and the Census data. We use the construction time as the measure of performance and vary the sampling ratio from 0.5% to 10%. If for LST with more blocks, we may need larger sampling ratio to capture the local patterns.

Figure 11 is the experimental results for $|B|$=100. We observe

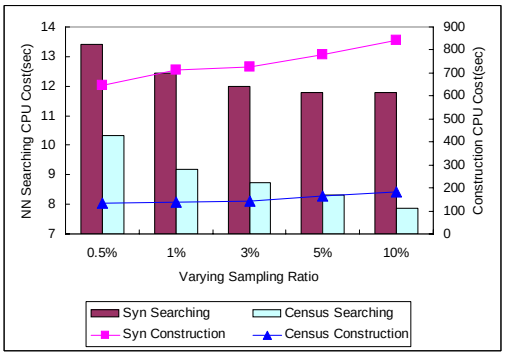

Figure 11: Effect of Sampling Ratio, $|B|$=100

that when the sampling ratio increases, the construction time increases but the query performance improves. The reason is that with more samples available, more computations are needed during construction, but more local patterns are discovered improving the quality of the index. However, there exists a *critical sampling ratio* after which the query performance is hardly improved while the construction time still increases (*e.g.*, 15% for the Census dataset). The critical sampling ratio achieves the best trade-off between construction speed and query performance.

5.6 LST Performance with Respect to b and K

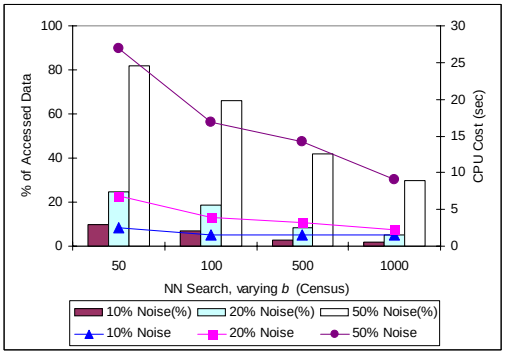

Figure 12: Performance Study on Varying b

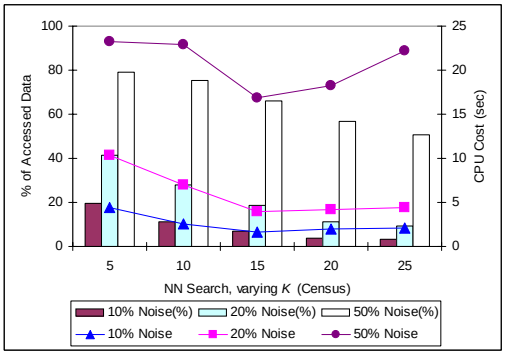

Figure 13: Performance Study on Varying K

In this section, we illustrate that how LST behaves with different settings of parameter b and K. Here we present the results for NN query on the Census dataset. The performance in the other cases is similar. Figure 12 illustrates the performance of the index when the number of column groups is fixed and the number of blocks varies from 50 to 1000. We can see that with increasing number of blocks, both the percentage of accessed data and the CPU cost for searching decrease. The reason is that with more blocks, we

can exploit more consistent local correlation information and obtain better clusters on columns. Thus the optimistic bound obtained from representative signatures can prune more computation of the real distance. Figure 13 illustrates the performance with various number of column groups. The number of blocks is fixed to 100 and the number of column groups varies from 5 to 25. We can see that with increasing number of column groups, we derive more accurate clustering on the columns and get better performance. But when K is larger than 25, the performance tends to be constant and with less performance gain. This is because more column groups require more time on the computation of optimistic bound and trade off the performance gain of processing less data. So K=15 should be an optimal value. We note that the optimal values for b and K can generally be determined by performance tuning on a sample of the dataset before building the LST.

6. CONCLUSION AND FUTURE WORK

With numerous emerging applications requiring efficient access to transaction datasets, there is a need for scalable and robust techniques to index such data. In this paper, we propose a novel indexing structure, called *Localized Signature Table(LST)*, which uses a signature based representation of transaction data. Besides global patterns, LST also exploits local patterns among transactions resulting to better query performance. We show how the branch-and-bound method can benefit from the properties of LST to achieve efficient processing of similarity queries. We also develop an efficient construction algorithm for the LST index. Extensive experiments illustrate that LST significantly outperforms the existing indexing methods in terms of query performance, scalability, sensitivity to the characteristics of datasets and robustness to the dimensionality.

Currently we are working on extending the method for more complex similarity measures with multiple targets, or range queries with multiple similarity functions. Also, we plan to incorporate more types of similarity searches (*e.g.*, Ref. [12]).

7. REFERENCES

[1] C. C. Aggarwal, J. L. Wolf, and P. S. Yu. A new method for similarity indexing of market basket data. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 407–418, 1999.

[2] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. of the 20th Int. Conf. on Very Large Data Bases*, pages 487–499, 1994.

[3] K. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft. When is "nearest neighbor" meaningful? *Lecture Notes in Computer Science*, 1540:217–235, 1999.

[4] K. Chakrabarti and S. Mehrotra. Local dimensionality reduction: A new approach to indexing high dimensional spaces. In *Proc. of the 26th Int. Conf. on Very Large Data Bases*, pages 89–100, 2000.

[5] S. C. Deerwester, S. T. Dumais, T. K. Landauer, G. W. Furnas, and R. A. Harshman. Indexing by latent semantic analysis. *J. of the American Society of Information Science*, 41(6):391–407, 1990.

[6] A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the EM algorithm. *J. of the Royal Statistical Society, Series B*, 39:1–38, 1977.

[7] C. Faloutsos and H. V. Jagadish. On B-tree indices for skewed distributions. In *Proc. of the 18th Int. Conf. on Very Large Data Bases*, pages 363–374, 1992.

[8] A. Gionis, D. Gunopulos, and N. Koudas. Efficient and tunable similar set retrieval. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, 2001.

[9] A. Guttman. R-trees: a dynamic index structure for spatial searching. *Readings in database systems*, pages 599–609, 1988.

[10] A. K. Jain and R. C. Dubes. *Algorithms for clustering data*. Prentice-Hall, Inc., 1988.

[11] Q. Jing, R. Yang, P. Kalnis, and A. K. H. Tung. Localized signature table: Fast similarity search on transaction data. *Technical Report*, 2003.

[12] N. Mamoulis, D. W. Cheung, and W. Lian. Similarity search in sets and categorical data using the signature tree. In *Proc. of the 19th IEEE Int. Conf. on Data Engineering*, 2003.

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

[14] C. Ordonez, E. Omiecinski, and N. Ezquerra. A fast algorithm to cluster high dimensional basket data. In *Proc. of the 17th IEEE Int. Conf. on Data Mining*, pages 633–636, 2001.

[15] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *Proc. of the ACM SIGMOD Int. Conf. on Management of data*, pages 71–79, 1995.

[16] Y. Sakurai, M. Yoshikawa, S. Uemura, and H. Kojima. The A-tree: An index structure for high-dimensional spaces using relative approximation. In *Proc. of the 26th Int. Conf. on Very Large Data Bases*, pages 516–526, 2000.

[17] R. Sibson. SLINK: an optimally efficient algorithm for the single-link cluster method. *The Computer Journal*, 16(1):30–34, 1973.

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

[19] R. Weber, H.-J. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *Proc. of the 24th Int. Conf. on Very Large Data Bases*, pages 194–205, 1998.