Efficient Indirect Association Discovery Using Compact Transaction Databases

Qian Wan and Aijun An

***Abstract*— An indirect association is a special type of negative association that relates two items via a mediator. The two items in an indirect association are rarely present together, but each of them occurs frequently together with the mediator. In this paper, we propose *HI-mine**, an innovative optimization of the previously developed *HI-mine* algorithm for fast extracting indirect associations. This optimization is based on a novel strategy for compressing a transaction database into a Super Compact Transaction Database, which dramatically reduces not only the number of transactions in the database, but also the memory requirement for storing frequent-item projections in mining indirect associations. Our experimental results show that the *HI-mine** algorithm is effective and efficient, and improves the performance of indirect association mining significantly.**

***Index Terms*— Data mining, association rule, indirect association, algorithm.**

I. Introduction

An association rule is an implication of the form $X \Rightarrow Y$, which indicates that if itemset X occurs in a transaction, then itemset Y will likely also occur in the same transaction. The problem of association rule mining has been studied extensively. A number of algorithms have been proposed to improve the running time for generating association rules and frequent itemsets [1], [2], [3], [5], [12].

The importance of extending the current association rule framework to include negative associations was first pointed out in [2]. Ever since, many techniques for mining negative associations have been developed [6], [8], [12]. This problem was addressed in [6] by combining previously discovered positive associations with domain knowledge to constrain the search space such that fewer but more interesting negative rules are mined. A general framework for mining both positive and negative association rules of interest was presented in [12], in which no domain knowledge was required, and the negative association rules were given in more concrete expressions to indicate actual relationships between different itemsets.

In [8], a new class of patterns called indirect associations was proposed and its utilities were examined in various application domains. Indirect associations provide an effective way to detect interesting negative associations by discovering only "infrequent itempairs that are highly expected to be frequent" without using negative items or domain knowledge. Consider a pair of items, *x* and *y*, that are rarely present together in the same transaction. If each item highly depends on the presence of an itemset *M*, the pair (*x*, *y*) is said to be indirectly associated via *M*.

An *Apriori*-like algorithm, called INDIRECT, for mining indirect associations between pairs of items was given in [7], [8]. Similar to Apriori [1], it uses two join steps to generate frequent itemset candidates and indirect association candidates. Both candidate generation steps can be quite expensive, because each of them involves a great number of join operations. The join step for generating indirect association candidates is even more expensive than the one used in *Apriori* for generating frequent itemset candidates.

In order to reduce the cost in indirect association mining, we proposed the *HI-mine* algorithm in [10], [9], which is based on a novel data structure, *HI-struct*. With *HI-struct*, we do not need to do any join operation for candidate generation. Instead, we generate two new sets, *indirect itempair set* and *mediator support set*, by recursively building the *HI-struct* for the database. Then indirect associations are discovered from these two sets directly and efficiently. In [10], [9], we demonstrated that the *HI-mine* algorithm is significantly faster than the INDIRECT algorithm.

In this paper we present the *HI-mine** algorithm, a novel, effective and efficient optimization of the *HI-mine* algorithm. This optimization is based on a new strategy for compressing a transaction database into a *Super Compact Transaction Database* that is an extension of *Compact Transaction Database* [11]. We show that this strategy drastically cuts down the size of memory required to store *HI-struct* and also significantly improves the running time of *HI-mine* by allowing one database scan, identical transaction merging, analogical transaction combining, direct frequent item projecting and dynamic infrequent item pruning.

The rest of this paper is organized as follows. Section 2 gives the formal definition of indirect associations. The generation of super compact transaction databases is described in Section 3. Then, we present *HI-mine** algorithm in Section 4. Our empirical results are reported in Section 5. Finally, we conclude the paper in Section 6.

II. Preliminaries

Let *I* = $\{i_1, i_2, \ldots, i_m\}$ be a set of *m items*. A subset $X \subseteq I$ is called an *itemset*. A *k*-itemset is an itemset that contains *k* items. A *transaction database* $\mathcal{D} = \{T_1, T_2, \ldots, T_N\}$ is a set of *N* transactions, where each transaction T_n ($n \in \{1, 2, \ldots, N\}$) is a set of items such that $T_n \subseteq I$. A transaction *T* contains an itemset *X* if and only if $X \subseteq T$. An example transaction database *TDB* is shown in Table I. In *TDB*, *I* = {A, B, C, D} and *N* = 10.

Manuscript received December 30, 2005; revised February 2, 2006.

Qian Wan and Aijun An are with the Department of Computer Science and Engineering, York University, Toronto, Ontario, M3J 1P3, Canada (phone: 416-736-2100x44298; fax: 416-736-5872; e-mail: {qwan, aan}@cs.yorku.ca)

TABLE I

AN EXAMPLE TRANSACTION DATABASE

TID	List of itemIDs
001	A, B, C, D
002	A, B, C
003	A, B, D
004	B, C, D
005	C, D
006	A, B, C
007	A, B, C
008	B, C
009	B, C, D
010	C, D

The *support* of an itemset *X* is the percentage of transactions in $\mathcal{D}$ containing *X*: *sup*(*X*)=$\|\{t \mid t \in \mathcal{D}, X \subseteq t\}\| \ / \ \|\{t \mid t \in \mathcal{D}\}\|$, where $\|S\|$ is the cardinality of set *S*. An itemset *X* in a transaction database $\mathcal{D}$ is called as a *frequent itemset* if $sup(X)$ is not less than a user-specified minimum support, *min_sup*. Accordingly, an *infrequent itemset* is an itemset that does not satisfy the *min_sup*.

Definition 2.1 (Indirect Association [8]): An itempair {x, y} is indirectly associated via a mediator M, if the following conditions hold:

1. sup({x, y}) $< t_s$

2. There exists a non-empty itemset M such that:

(a) sup({x} ∪ M) $\geq t_f$, sup({y} ∪ M) $\geq t_f$;

(b) dep({x}, M) $\geq t_d$, dep({y}, M) $\geq t_d$, where dep(P, Q) is a measure of the dependence between itemsets P and Q.

The thresholds above are called *itempair support threshold* (t_s), *mediator support threshold* (t_f), and *mediator dependence threshold* (t_d), respectively. In practice, it is reasonably to set $t_f \geq t_s$. In this paper, notation $\langle x, y \mid M \rangle$ is used to represent the indirect association between *x* and *y* via *M*, and the *IS* measure [8] is used as the dependence measure for Condition 2(b).

Given a transaction database $\mathcal{D}$ and three thresholds: t_s, t_f and t_d, the *Indirect Itempair Set* and *Mediator Support Set* are defined as follows.

Definition 2.2 (Indirect Itempair Set): Let L_1 be the set of 1-itemset of $\mathcal{D}$. We define the indirect itempair set (*IIS*) of $\mathcal{D}$ as:

$IIS(\mathcal{D}) = \big\{\langle x,y \rangle \big| \{x\} \in L_1 \wedge \{y\} \in L_1 \wedge sup(\{x\}) \geq t_f \wedge sup(\{y\}) \geq t_f \wedge sup(\{x,y\}) < t_s \big\}$

Definition 2.3 (Mediator Support Set): Let L be the set of itemsets of $\mathcal{D}$, and L_1 be the set of 1-itemset of $\mathcal{D}$. The mediator support set (*MSS*) of item x is defined as:

$MSS(x) = \big\{M \big| M \in L \wedge sup(M \cup \{x\}) \geq t_f \wedge dep(M, \{x\}) \geq t_d \big\}$

Given a transaction database $\mathcal{D}$, our previous *HI-mine* algorithm scans $\mathcal{D}$ twice to build an initial *HI-struct*, dynamically mines the *HI-struct* to compute *IIS($\mathcal{D}$)* and all the *MSS(x)*s, and then generates the complete set of indirect associations from *IIS($\mathcal{D}$)* and *MSS(x)*s. In the next sections, we will first describe how to compress the database $\mathcal{D}$ into a *Super Compact Transaction Database*. Then we will describe the *HI-mine** algorithm that builds and mines a compressed version of *HI-struct* from the super compact transaction database to mine the complete set of indirect associations.

III. SUPER COMPACT TRANSACTION DATABASE

Our motivation for building a compact transaction database is based on the following observations. First, a number of transactions in a transaction database may contain the same set of items. For example, as shown in Table I, transaction {A, B, C} occurs three times, and transactions {B, C, D} and {C, D} both occur two times in the same database. Therefore, if the transactions that have the same set of items can be stored in a single transaction with their number of occurrences, it is possible to avoid repeatedly scanning the same transaction in the original database. Moreover, if the frequency count of each item in the given transaction database can be acquired when constructing the compact database before mining takes place, it is possible to avoid the first scan of the database to identify the set of frequent items as most approaches to efficient mining of frequent patterns do.

TABLE II

THE COMPACT TRANSACTION DATABASE

head				
Item	C	B	D	A
Count	9	8	6	5
body				
Count	List of itemIDs			
3	C, B, A			
1	C, B, D, A			
1	B, D, A			
1	C, B			
2	C, B, D			
2	C, D			

The compact transaction database (*CTDB*) of the example transaction database *TDB* is shown in Table II. It contains two parts: *head* and *body*. The *head* lists all the four items in *TDB* with their frequency counts, ordered in frequency-descending order, {C:9, B:8, D:6, A:5}. The *body* consists of 6 unique transactions with their frequency counts, instead of 10 transactions in *TDB*. The items in each transaction are ordered in frequency-descending order as well. In [11], we presented an efficient algorithm that converts a transaction database into its compact transaction database. The algorithm uses a data structure called *CT-tree* and scans the transaction database once.

The use of a *CTDB* can not only save storage space, but also greatly reduce the I/O time required by database scans during mining association rules or indirect associations. In fact, we can improve the performance even further by grouping the same set of items in a number of transactions into a single transaction, which leads to a new *Super Compact Transaction Database* (*STDB*).

In *STDB*, *analogical transactions*, in which only the last one item is different, are combined into a new transaction with two parts: the *front* part that contains the same set of items and the *back* part that contains the list of all different items with their count values. The *STDB* for the example *CTDB* in Table II is shown in Table III. The *head* parts of the two databases are exactly the same, while the *body* part of *STDB* consists of only 4 transactions. For instance, the second transaction in *STDB* records two analogical transactions in *CTDB*, the first one and the fifth one. Its *front* contains the same set of items: {C, B} and its *back* contains a list: {2:D, 3:A}.

Having introduced the concept of *STDB*, we now present an algorithm for building an *STDB* from a *CTDB*. The algorithm consists of two steps. In the first step, a *CT-tree* (first proposed in [11] to generate a *CTDB* from the original transaction database) is built from the *CTDB*, and in the second step an

TABLE III

THE *STDB* OF *TDB*

head				
Item	C	B	D	A
Count	9	8	6	5
body				
front	*back*			
items	count	item	count	item
C	1	B	2	D
C, B	2	D	3	A
C, B, D	1	A		
B, D	1	A		

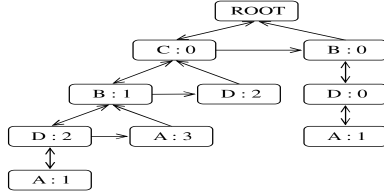

Fig. 1. *CT-tree* for *CTDB* in Table II.

STDB is generated from the *CT-tree*. A *CT-tree* represents a transaction database. It can be generated by scanning either the original transaction database or its *CTDB*. Fig 1 shows the *CT-tree* constructed by scanning the body of *CTDB* in Table II.

In a *CT-tree*, every tree node V (except the root of the tree, which is labelled as "ROOT") is a 2-tuple ($V_i : V_c$), where V_i is an item id and V_c is the count value of a unique transaction consisting of all the items in the branch of the tree from the root to node V. The process for generating the *CT-tree* in Fig 1 from the *CTDB* in Table II is briefly illustrated as follows.

The scan of the first three records in *CTDB* leads to the construction of three new branches of the tree: CBA, CBDA and BDA. The count value of each record is stored in the last node of each path, while all other nodes remain 0. The next two records change the count value of node B and D of the leftmost path into 1 and 2, respectively. Finally, for the last record, a new path CD is created, and node D stores the count of this record. Note that a *CT-tree* built from a *CTDB* has the property that each branch of the tree lists its items from high to low levels in frequency-descending order. Such a *CT-tree* can also be generated from the original transaction database by first sorting all the items in each transaction in frequency-descending order and then inserting them into the *CT-tree*.

Having the *CT-tree*, an *STDB* can be generated as follows. For each node N in the tree, if there exists a child node (C_i: C_c) of N where $C_c > 0$, then a new transaction is generated in *STDB* of which the *front* part records all the items in the branch from the root to node N and the *back* part records the list of all the child nodes whose count value is greater than 0.

The algorithm for generating an *STDB* from a *CTDB* using the *CT-tree* data structure is described as follows.

1: *head* of *STDB* $\leftarrow$ *head* of *CTDB*
2: root[*CTtree*] $\leftarrow$ "ROOT"
3: **for** each transaction T_n in the *body* of $CTDB$ **do**
4:    ***insert***(T_n, *CTtree*)
5: **end for**
6: **if** *CTtree* is not empty **then**
7:    ***write***(root, *STDB*)
8: **end if**

procedure *insert*(T, *CTtree*)

1: *thisNode* $\leftarrow$ root[*CTtree*]
2: **for** each item i in transaction T **do**
3:    *nextNode* $\leftarrow$ child[*thisNode*]
4:    **while** *nextNode* $\neq$ *null* **and** item[*nextNode*] $\neq i$ **do**
5:       *nextNode* $\leftarrow$ sibling[*nextNode*]
6:    **end while**
7:    **if** *nextNode* = *null* **then**
8:       item[*newNode*] $\leftarrow i$
9:       **if** i is the last item in T **then**
10:          count[*newNode*] $\leftarrow$ count[T]
11:       **else**
12:          count[*newNode*] $\leftarrow$ 0
13:       **end if**
14:       parent[*newNode*] $\leftarrow$ *thisNode*
15:       sibling[*newNode*] $\leftarrow$ child[*thisNode*]
16:       child[*newNode*] $\leftarrow$ *null*
17:       child[*thisNode*] $\leftarrow$ *newNode*
18:       *thisNode* $\leftarrow$ *newNode*
19:    **else**
20:       **if** item i is the last item in T **then**
21:          count[*thisNode*] = count[*thisNode*] + count[T]
22:       **else**
23:          *thisNode* $\leftarrow$ *nextNode*
24:       **end if**
25:    **end if**
26: **end for**

procedure *write*(*node*, *STDB*)

1: *numItems* $\leftarrow$ 0
2: *nextNode* $\leftarrow$ child[*node*]
3: **while** *nextNode* $\neq$ *null* **do**
4:    **if** count[*nextNode*] > 0 **then**
5:       *numItems*++
6:       count[*backList*[*numItems*]] $\leftarrow$ count[*nextNode*]
7:       item[*backList*[*numItems*]] $\leftarrow$ item[*nextNode*]
8:    **end if**
9:    *nextNode* $\leftarrow$ sibling[*nextNode*]
10: **end while**
11: **if** numItems > 0 **then**
12:    *nextNode* $\leftarrow$ *node*
13:    **while** *nextNode* $\neq$ root[*CTtree*] **do**
14:       *frontItems* $\leftarrow$ add item[*nextNode*]
15:       *nextNode* $\leftarrow$ parent[*nextNode*]
16:    **end while**
17:    **if** *frontItems* is not empty **then**
18:       *STDB* $\leftarrow$ write *frontItems*
19:       *STDB* $\leftarrow$ write *backList*
20:    **end if**
21: **end if**
22: **if** child[*node*] $\neq$ *null* **then**
23:    ***write***(child[*node*], *STDB*)
24: **end if**
25: **if** sibling[*node*] $\neq$ *null* **then**
26:    ***write***(sibling[*node*], *STDB*)
27: **end if**

The *head* of *CTDB* is copied into *STDB* in the first step. From step 2 to step 5, a complete *CT-tree* is built with one scan of the *body* of *CTDB* by calling the procedure *insert*(T_n, *CT-tree*). Then, after calling the procedure *write*(*root*, *STDB*) in step 7 recursively, every analogical transaction with two parts, *frontItems* and *backList*, is written into the *body* of *STDB* (step 18 and 19). Thus, a super compact transaction database is generated.

IV. *HI-mine** ALGORITHM

The *HI-mine** algorithm improves its predecessor, *HI-mine*, by mining indirect associations from an *STDB* instead of an

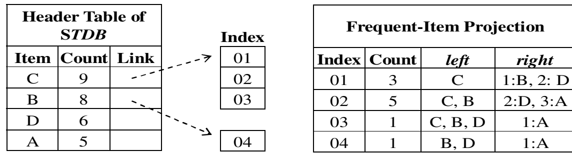

Fig. 2. The initial *HI-struct* of *STDB*.

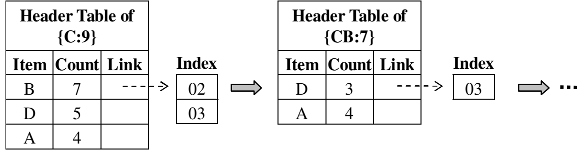

Fig. 3. Head table H_C and H_{CB}.

original database. The algorithm consists of two phases. In the first phase, it builds an *HI-struct* from an *STDB* and then dynamically adjusts and mines the *HI-struct* to compute the indirect itempair set (*IIS*) and the mediator support sets (*MSS*s) (see Definitions 2.1 and 2.2). In the second phase of *HI-mine**, the complete set of indirect associations is generated from *IIS* and *MSS*s.

Let's illustrate *HI-mine** using the following example. Suppose $t_s = t_f = t_d = 0.5$, where t_s, t_f and t_d are itempair support threshold, mediator support threshold and mediator dependence threshold, respectively (see Definition 2.1). The initial *HI-struct* of the *STDB* in Table III is shown in Fig 2, which consists of a header table H and frequent-item projections of the transactions in the *STDB*. A frequent-item projection of a transaction contains only the frequent items in the transaction with respect to t_f. In this example, all the items in the database are frequent.

The initial *HI-struct* is constructed as follows. First, we collect the set of frequent items *F* with respect t_f and their supports in support descending order from the *head* part of *STDB*. For the example database, *F* is $\{$C, B, D, A$\}$. Then a header table *H* is created, where each frequent item has an entry with three fields: its item-id, its support count, and a pointer to a queue. For each record *R* in the *body* of *STDB*, select the frequent items according to the order of *F*. Let the frequent item list in the *front* of *R* be $[t|T_f]$, where *t* is the first element and T_f is the remaining list. Let the *back* list of *R* be T_b and the sum of the count values of frequent items in T_b be *c*. Add $[c][t|T_f][T_b]$ to a frequent-item projection array, and append its index of the array to *t*'s queue in the header table H. Thus, all indexes of the frequent-item projections with the same first item (in the order of *F*) are linked together as a queue, and the entries in the header table *H* act as the heads of the queues.

The subsequent mining process involves building *IIS*(*STDB*) and *MSS* of each frequent item. We use a divide-and-conquer strategy to build these sets by partitioning each set into disjoined subsets and generating each subset in turn. Following the support descending order of frequent items: C, B, D, A, the complete *IIS*(*STDB*) and each *MSS* can be partitioned into 4 subsets as follows: (1) those containing item C; (2) those containing item B but no item C; (3) those containing item D, but no item C nor B; and (4) those containing only item A.

In order to find *IIS*(*STDB*) and *MSS*s that contain item C, a C-header table H_C (shown in the left side of Fig 3) is created by traversing the C-queue in the header table *H* once. In H_C, every frequent item, except for C itself, has an entry with the same fields as in *H*, i.e., item-id, support count and a pointer to a queue. The support count in H_C records the support of the corresponding item in the C-queue. For example, since item A appears 4 times in the frequent-item projections of C-queue, the support count in the entry for A in H_C is 4. And all the indexes of the frequent-item projections with the same first two items are linked together as a queue, and the entries in the header table H_C act as the heads of the queues. For instance, the B-queue in H_C stores all indexes of the frequent-item projections with the same first two items CB. Since A is locally infrequent with respect to C, itempair $\langle$A, C$\rangle$ is added to *IIS*(*STDB*). The other two items B and D are locally frequent, and the *IS* measure between C and each of these two items passes the minimum dependence threshold 0.5. Therefore, $\{$C$\}$ is added to *MSS*(B) and *MSS*(D).

Then, a header table H_{CB} (shown in the right part of Fig 3) is created by examining B-queue in H_C in the same manner as in generating H_C from the C-queue in *H*. Thus the algorithm recursively exams the CB-projected database to determine whether itemset $\{$C, B$\}$ belongs to *MSS*(D) and *MSS*(A). Since H_{CB} contains no frequent item, the search along path CB completes. Similarly, the mining process continues to discover *MSS*s that contain itemset $\{$C, D$\}$, itemset $\{$C, A$\}$ and so on.

In the next step, all the indexes in the C-queue of header table H are moved into the proper queues in H to mine *IIS*(*STDB*) and *MSS*s that contain item B but not C, and other subsets of them. The proper queue is the queue of the item right after item C in the corresponding frequent-item projection. The header table *H* after this adjustment is shown in Fig 4. The final results for *IIS*(*STDB*) and *MSS*s after mining B, D and A queues are listed as follows:

IIS(*STDB*) = $\{\langle$C, A$\rangle$, $\langle$B, D$\rangle$, $\langle$D, A$\rangle\}$
MSS(A) = $\{\{$B$\}\}$ *MSS*(B) = $\{\{$C$\}$, $\{$A$\}\}$
MSS(C) = $\{\{$B$\}$, $\{$D$\}\}$ *MSS*(D) = $\{\{$C$\}\}$

The second phase of the *HI-mine** algorithm is to compute the set of mediators for each *indirect itempair* in *IIS*(*STDB*). For example, the set of mediators for itempair $\langle$C, A$\rangle$ in *IIS*(*STDB*) is computed by intersecting *MSS*(C) and *MSS*(A), which results in $\{\{$B$\}\}$. In this way, two indirect associations are discovered in the example database: $\langle$C, A $|$ $\{$B$\}\rangle$ and $\langle$B, D $|$ $\{$C$\}\rangle$.

Compared to *HI-mine*, *HI-mine** uses the following techniques to optimize the performance of indirect association mining.

(1) *One database scan*: In *HI-mine*, two scans of the original database are needed to build the initial *HI-struct*. It first scans the database to get the set of frequent items and then scans it again to construct *HI-struct*. Only one scan of an

Fig. 4. *HI-struct* after mining C-queue.

STDB (whose size is usually much smaller) is needed in *HI-mine**. Even though it takes time to build an *STDB* from the original database, the database compression into the *STDB* is conducted only once. The subsequent mining of indirect associations from the *STDB* can be conducted multiple times with different settings of support and dependence thresholds. These multiple runs of *HI-mine** all benefit from the one-time database compression.

(2) *Identical transaction merging*: *HI-mine** avoids repeatedly scanning transactions that have the same set of items by merging them into a single transaction. This strategy, as well as the following one, saves a great amount of memory required to build the *HI-struct*.

(3) *Analogical transaction combining*: *HI-mine** avoids repeatedly scanning the same set of items by combining analogical transactions, in which only the last item is different, into a new transaction with two parts: the *front* part containing the same set of items and the *back* part containing the list of all different items with their count values.

(4) *Direct frequent item projecting*: Since the items in the *front* of each new transaction are stored in frequency descending order, *HI-mine** can directly add the selected items to the frequent-item projection array with no need to sort these items first, while these operations must be done during the second database scan of *HI-mine*.

(5) *Dynamic infrequent item pruning*: In *HI-mine**, items that are locally infrequent are dynamically pruned from the deeper level header table of *HI-struct*. For example, header table H_{CB} does not contain item A, because A's count is 4 in the header table H_C. Moreover, items in the *back* of each frequent-item projection will never be considered for adjusting proper queues, only the count values of these items are needed in the mining process.

Due to these features, *HI-mine** uses less amount of memory and performs much faster than *HI-mine*.

V. Experimental Studies

In this section, we report our experimental results on the generation of super compact transaction databases as well as the performance of the *HI-mine** algorithm using super compact transaction databases in comparison with the *HI-mine* algorithm using original transaction databases.

A. Environment of experiments

All the experiments are performed on a double-processor server, which has 2 Intel Xeon 2.4G CPU and 2G main memory, running on Linux with kernel version 2.4.26. All the programs are written in Sun Java 1.4.2. To evaluate the performance of the two algorithms over a large range of data characteristics, we have tested the programs on various real world and synthetic data sets.

The synthetic data sets, shown in the first 4 rows of Table IV, are generated using the procedure described in [1]. In these data sets, total number of items and number of maximal potentially frequent items are set to 1000 and 2000, respectively. Microsoft data set, obtained from UCI Machine Learning Repository, was created by sampling and processing the web logs of Microsoft. LiveLink data set

TABLE IV

Database characteristics

Database	# Items	# Trans	# Trans in *STDB*	Compr. rate
T10I5D100K	1000	100,000	83,859	16.1%
T20I10D100K	1000	100,000	88,157	11.8%
T15I10D200K	1000	200,000	141,339	29.3%
T20I15D200K	1000	200,000	132,264	33.9%
Microsoft	294	32,711	6,653	79.7%
LiveLink	38,679	30,586	17,201	43.8%
Retail	16,470	88,162	75,407	14.5%
Kosarak	41,270	990,002	364,573	63.2%

was first used in [4] to discover interesting association rules from Livelink web log data. This data set is not publicly available for proprietary reasons. The other two data sets are taken from the Frequent Itemset Mining Dataset Repository (http://fimi.cs.helsinki.fi/data), in which Retail contains the (anonymized) retail market basket data from an anonymous Belgian retail store and Kosarak contains (anonymized) click-stream data of a Hungarian on-line news portal.

B. Generation of STDB

To evaluate the effectiveness of our approach, we compared the super compact transaction database with the original database in terms of the number of transactions and the memory requirement to store the frequent-item projections.

Table IV shows the number of transactions in the transaction data base (*TDB*), the number of transactions in *STDB* and the compression rate in transaction numbers, defined as $\frac{\#\ of\ trans.\ in\ TDB - \#\ of\ trans\ in\ STDB}{\#\ of\ transactions\ in\ TDB}$. We can see that the proposed approach leads to a good compression in the number of transactions with an average compression rate of 22.8% among the synthetic databases and an excellent compression with an average rate of 50.3% among the real-world databases. In the best case, a compression rate of 79.7% is achieved in the Microsoft web data.

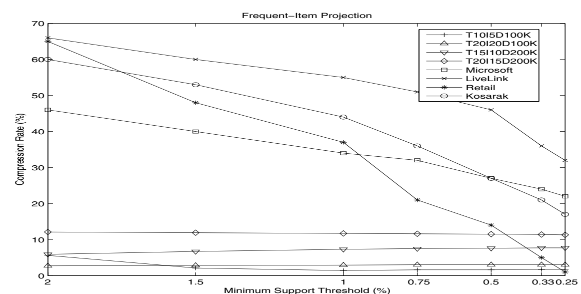

Fig. 5. Compression rates of *STDB*

Fig 5 shows the compression rates of the memory requirement for storing frequent-item projections of the *STDB*s from all the synthetic and real-world datasets mentioned above, for various values of minimum support from 0.25% to 2%. The compression rate is calculated as (*amount of memory needed in HI-mine - amount of memory needed in HI-mine**) / (*amount of memory needed in HI-mine*). We can observe from Fig 5 that, for the real-word datasets, the compression rates become much greater when the support threshold increases to relatively higher values. Even for very low support values, such as 0.5%, the compression rates are over 20% for all the real-world datasets except the Retail dataset. Moreover, we find once again that much higher compression rates are achieved in the real-world data sets than in the synthetic ones, which

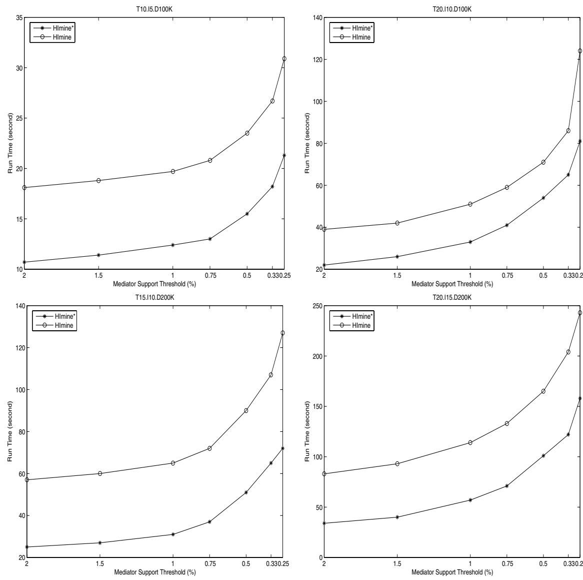

Fig. 6. Performance on synthetic data sets

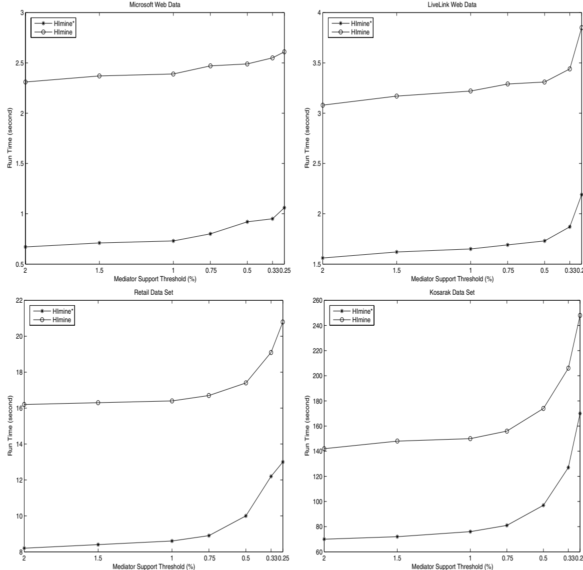

Fig. 7. Performance on real-world data sets

indicates that the super compact transaction database provides more effective data compression in real-world applications.

C. Evaluation of efficiency

To assess the efficiency of our proposed approach, we performed a number of experiments to compare the runtime of *HI-mine** with that of *HI-mine*. Fig. 6 and Fig. 7 illustrate the corresponding execution times for the two algorithms on two different types of databases with various support thresholds from 2% down to 0.25%.

From these performance curves, it can be observed that *HI-mine** achieves remarkably better runtimes than *HI-mine* in all situations, and shows the anticipated behavior. It is important to note that there is a significant efficiency gain by *HI-mine** in real-world data, where *HI-mine** is approximately two to three times faster than *HI-mine* for almost all support levels. This also indicates that our new approach offers more performance improvement in real-world applications.

VI. CONCLUSION

In this paper, we propose *HI-mine**, an innovative optimization of the previously developed *HI-mine* algorithm for fast extracting indirect associations. This optimization is based on a novel strategy that compresses a transaction database into a *super compact transaction database*, which dramatically reduces not only the number of transactions in the original database, but also the memory requirement for storing frequent-item projections and the runtime for mining indirect associations. Our experimental results verify the effectiveness and efficiency of our approach, and demonstrate that *HI-mine** consistently outperforms the previous algorithm in terms of both memory requirement and runtime on both real-world and synthetic databases. In particular, it achieves a significant efficiency gain in real-world data by a factor of two to three on average.

Our study has been confined to mining the complete set of indirect associations between itempairs from compact transaction databases. However, the method developed here can be extended for mining indirect associations between itemsets or other new types of interesting associations. We are studying these problems and will report our progress in the future. Furthermore, in [9] we proposed a solution for *HI-mine* to handle situations where frequent-item projections cannot be held in the main memory. The solution will be adapted to *HI-mine**.

REFERENCES

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

[2] S. Brin, R. Motwani, J. Ullman, and S. Tsur. Dynamic itemset counting and implication rules for market basket data. In *Proceedings of the International ACM SIGMOD Conference*, pages 255–264, Tucson, Arizona, USA, May 1997.

[3] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *Proceedings of ACM-SIGMOD Int. Conf. on Management of Data*, pages 1–12, 2000.

[4] X. Huang, A. An, N. Cercone, and G. Promhouse. Discovery of interesting association rules from livelink web log data. In *Proceedings of IEEE Int. Conf. on Data Mining*, 2002.

[5] J. Pei, J. Han, H. Lu, S. Nishio, S. Tang, and D. Yang. H-mine: hyper-structure mining of frequent patterns in large database. In *Proceedings of the IEEE International Conference on Data Mining*, San Jose, CA, November 2001.

[6] A. Savasere, E. Omiecinski, and S. Navathe. Mining for strong negative associations in a large database of customer transactions. In *Proceedings of the 14th International Conference on Data Engineering*, pages 494–502, 1998.

[7] P. Tan and V. Kumar. Mining indirect associations in web data. In *Proc of WebKDD2001: Mining Log Data Across All Customer TouchPoints*, August 2001.

[8] P. Tan, V. Kumar, and J. Srivastava. Indirect association: mining higher order dependencies in data. In *Proc. of the 4th European Conf. on Principles and Practice of Knowledge Discovery in Databases*, pages 632–637, 2000.

[9] Q. Wan and A. An. An efficient approach to mining indirect associations. *Journal of Intelligent Information Systems (JIIS)*. To appear.

[10] Q. Wan and A. An. Efficient mining of indirect associations using *HI-mine*. In *Proceedings of the 16th Conference of the Canadian Society for Computational Studies of Intelligence, AI 2003*, Halifax, Canada, June 2003.

[11] Q. Wan and A. An. Compact transaction database for efficient frequent pattern mining. In *Proceedings of IEEE Int. Conf. on Granular Computing*, Beijing, China, 2005.

[12] X. Wu, C. Zhang, and S. Zhang. Mining both positive and negative association rules. In *Proceedings of the 19th Int. Conf. on Machine Learning*, pages 658–665, 2002.