TidFP: Mining Frequent Patterns in Different Databases with Transaction ID

C.I. Ezeife* and Dan Zhang

School of Computer Science, University of Windsor,
Windsor, Ontario,
Canada N9B 3P4
cezeife@uwindsor.ca, zhang3d@uwindsor.ca
http://www.cs.uwindsor.ca/~cezeife

Abstract. Since transaction identifiers (ids) are unique and would not usually be frequent, mining frequent patterns with transaction ids, showing records they occurred in, provides an efficient way to mine frequent patterns in many types of databases including multiple tabled and distributed databases. Existing work have not focused on mining frequent patterns with the transaction ids they occurred in. Many applications require finding strong associations between transaction id (e.g., certain drug) and the itemsets (e.g., certain adverse effects) to help deduce some pertinent lacking information (like how many people use this product in total) and information (like how many people have the adverse effects).

This paper proposes a set of algorithms TidFPs, for mining frequent patterns with their transaction ids in a single transaction database, in a multiple tabled database, and in a distributed database. The proposed technique scans the database records only once even with level-wise Apriori-based mining techniques, stores frequent 1-items with their transaction id bitmap, outperforms traditional approaches and is extendible to other tree-based mining techniques as well as sequential mining.

Keywords: Data mining, Transaction id, Frequent Patterns, Distributed Mining, Multiple Table Mining.

1 Introduction

Mining frequent itemsets from a database table has been solved largely by algorithms that are Apriori based (e.g., the Apriori algorithm [1]) and those that are pattern-tree growth techniques (e.g., FP-tree [6]). Algorithms for mining frequent patterns from sequential databases also exist and include GSP [11], PrefixSpan [10], SPADE [12], SPAM [2], WAP [9] and PLWAP [4]. The focus of all these existing techniques does not include generating frequent patterns, showing the records where they occurred or with their transaction ids as may

* This research was supported by the Natural Science and Engineering Research Council (NSERC) of Canada under an Operating grant (OGP-0194134) and a University of Windsor grant.

T.B. Pedersen, M.K. Mohania, and A M. Tjoa (Eds.): DaWaK 2009, LNCS 5691, pp. 125–137, 2009.

Table 1. Example Drug/Side Effects Database Records

Tid (Drug)	Items (Side Effects)
D_1	1 3 4
D_2	2 3 5
D_3	1 2 3 5
D_4	2 5

be needed by some applications. Existing algorithms are also designed for single table mining and not for mining multiple related tables in a not necessarily normalized database. Assume a pharmacovigilance database table which contains reports about the adverse events of certain drugs from medical professionals as well as patients as depicted in Table 1 where the set of items (adverse side effects) I = {1, 2, 3, 4, 5} and the set of transaction ids (Drugs) Tids = {D_1, D_2, D_3, D_4}.

Mining all drugs that have similar frequent side effects at minimum support of 50% would require generating frequent itemsets (or frequent side effects) with the transaction id (or Drug id) in the format [$< itemset >$ Tid-list] that allows mining more informative large itemsets as L = { [$< 1 > D_1D_3$], [$< 2 > D_2D_3D_4$], $< 3 > D_1D_2D_3$], [$< 5 > D_2D_3D_4$], [$< 1,2 > D_2D_3D_4$],[$< 1,3 > D_1D_3$], [$< 2,3 > D_2D_3$], [$< 2,5 > D_2D_3D_5$], [$< 3,5 > D_2D_3$], [$< 2,3,5 > D_2D_3$]}.

1.1 Contributions and Outline

This paper proposes a series of algorithms for mining frequent patterns with their transaction ids on different types of databases, including (i) from a single table, (ii) from a multiple database set of related tables, (iii) from a horizontally distributed database tables, and (iv) from a vertically distributed database tables. The objectives of the proposed techniques are:

1. Enabling more informative mining: For many applications, just producing the frequent patterns without linking them to the specific transactions they occurred in, may not be adequate. Also, enabling mining of multiple related tables either in a single or distributed database environment, provides answers to more complex queries.
2. Improving Mining Efficiency: This system aims at improving the mining efficiency by cutting down from several to one, the number of times the original database is scanned for purposes of support counting.

Section 2 presents related work, Section 3 presents the proposed systems: TidFP, TidFP-multi, TidFP-hordist and TidFP-vertdist miners for respectively mining single table, multiple tables, horizontally distributed tables, and vertically distributed tables. Section 4 describes the experimental results, while section 5 presents conclusions and future work.

2 Related Work

Association rule can be used to find correlation among items in a given transaction. Association rule mining was proposed in [7], where the formal definition of the problem is presented as: Let I = $\{i_1, \ldots, i_n\}$ be a set of literals, called items. Let database, D be a set of transaction records, where each transaction T is a set of items such that $T \subseteq I$. Associated with each transaction is a unique identifier, called its transaction id (TID). We say that a transaction T contains X, a set of some items in I, if $X \subseteq I$. An association rule is an implication of the form $X \rightarrow Y$, where $X \subseteq I$, $Y \subseteq I$, and $X \cap Y = \emptyset$. The rule $X \rightarrow Y$ holds in the transaction set D with confidence c if $c\%$ of transactions in D that contain X also contain Y. The rule $X \rightarrow Y$ has support s in the transaction set D if $s\%$ of transactions in D contain $X \cup Y$. An example database is shown in Table 1. Here, there are four transactions with TID D_1, D_2, D_3, and D_4. The rule $\{side\ effect\ 1\} \rightarrow \{side\ effect\ 2\}$ is an association rule because with a given minimum support of 50% or 2 out of 4 transactions, the 2-itemset *(1,2)* which, this rule is generated from, has a support of 3/4 or 75%. The confidence for this rule is 1/2=50%.

Several important association rule mining algorithms including the Apriori [7], [1], and Fp-growth [6] exist. The basic idea behind the Apriori algorithm [7], [1], is to level-wise, use shorter frequent k-itemsets (L_k) to deduce longer frequent (k+1)-itemsets (L_{k+1}) starting from candidate 1-itemsets consisting of single items in the set I defined above, until either no more frequent itemsets or candidate itemsets can be found. Thus, the Apriori algorithm finds frequent k-itemsets L_k from the set of frequent (k-1)-itemsets L_{k-1} using the following two main steps involving joining the $L_(k-1)$ with $L_(k-1)$ Apriori-gen way to generate candidate k-itemsets C_k, and secondly, pruning the C_k of itemsets not meeting the Apriori property or not having all their subsets frequent in previous large itemsets. To obtain the next frequent L_k from candidate C_k, the database has to be scanned for support counts of all itemsets in C_k. A modified version of the Apriori algorithm called AprioriTid [1] avoids re-scanning the database to enumerate frequent patterns. AprioriTid maintains a candidate set C'_k. Every entry of C'_k has two parts. One is transaction ID and the other is a list of k-itemsets. Instead of scanning the database to count the support for every candidate itemset, the algorithm iterates C'_k. Simultaneously, C_{k+1} is generated to enumerate (k+1)-itemsets. Although transaction Ids for every frequent itemsets can be obtained with this algorithm, experiments show that AprioriTid approach slows down performance once it processes large datasets. Since level-wise candidate generation as well as numerous scans of the database had been seen as a limitation of this approach, optimization techniques in the literature and alternative tree-based solution proposal with Frequent pattern tree growth FP-growth [5], [6] had also been used. The FP-growth approach scans the database once to build the frequent header list, then, represents the database transaction records in descending order of support of the F_1 list so that these frequent transactions are used to construct the FP-tree. The FP-tree are now mined for frequent patterns recursively through conditional pattern base

of the conditional FP-tree and suffix growing of the frequent patterns. None of the frequent itemset mining algorithms considers mining frequent patterns with their transaction ids.

Some existing sequential pattern mining algorithms with techniques using transactions IDs to generate frequent sequential patterns and count supports include SPADE [12] and SPAM [2]. SPADE uses a vertical id-list database format that associates each sequence to a list of objects in which it occurs along with the time-stamps and all frequent sequences are enumerated through temporal joins (or intersections) on id-lists. SPADE only needs to access the original database 3 times for support counting. Algorithm SPAM [2] has similar ideas as SPADE. However, instead of vertical representation of id-list, SPAM uses vertical bitmap representation of the entire database that fits in main memory. These sequential mining techniques are not focussed on generating Fps with their Tids and incur such limitations as inefficient memory utilizations and not suitable to scale to very large databases.

3 The Proposed TidFP Algorithms

Section 3.1 presents the main algorithm TidFp, being proposed for mining frequent patterns with the transaction ids where they occurred. Section 3.2 presents the version of the algorithm for mining multiple tables called TidFp-multi, section 3.3 presents the version of the algorithm for mining horizontally distributed database tables called TidFp-hordist, while section 3.4 provides the TidFp-vertdist for mining vertically distributed database tables.

3.1 TidFp for Mining Fps with Transaction Ids on a Single Table

Since an important goal of the TidFp algorithm is linking all frequent patterns to the database records or transactions where they came from, the TidFp algorithm represents each frequent k-itemset as an m-attribute tuple of the form $< F_{k_1}, T_{1k_1}, T_{2k_1}, \ldots, T_{mk_1} >$, where F_{k_1} is the first frequent k-itemset, and T_{mk_1} is the mth transaction id of the first frequent k-itemset. For example, given Table 1 and minimum support of 50%, the list of frequent 1-itemsets is $F_1 = \{< 1, D_1, D_3 >, < 2, D_2, D_3, D_4 >, < 3, D_1, D_2, D_3 >, < 5, D_2, D_3, D_4 >\}$. This implies as well that the candidate 1-itemsets listed by this technique is in the same form as: $C_1 = \{< I_1, T_{11}, T_{21}, \ldots, T_{m1} >\}$, where I_1 is the the first candidate 1-itemset, and T_{m1} is the mth transaction id of the first candidate 1-itemset. For our example drug database, the candidate 1-itemset is given as $C_1 = \{< 1, D_1, D_3 >, < 2, D_2, D_3, D_4 >, < 3, D_1, D_2, D_3 >, < 4, D_1 >, < 5, D_2, D_3, D_4 >\}$. Thus, with this TidFp technique, the database is scanned only once to obtain the candidate 1-itemsets with a list of their Tids. The Tids of each candidate itemset is implemented either as a bitmap stored for each itemset or as only one stored bitmap that itemsets point to. Then, the count of each candidate itemset's Tids is equivalent to the support of the itemset. The itemsets having support less than the minimum support are excluded from the frequent 1-itemset

list, leading to the itemset $< 4, D_1 >$ being deleted from the C_1 list to get F_1. In order to get the higher order candidate and frequent (k+1)-itemsets F_{k+1}, given a frequent k-itemset F_k, TidFp algorithm applies a modification of the Apriori-gen join function called the map-gen join function, which works on two components of the itemsets consisting of the itemset part and the transaction id part and obtaining higher order frequent (k+1)-itemsets does not require re-scanning the database for their supports as is needed with the Apriori-gen join. With the TidFp, the candidate (k+1)-itemsets C_k is obtained from the frequent k-itemsets for $k \geq 1$, by joining frequent k-itemsets F_k with itself mapgen way such that $C_{k+1} = F_k \bowtie F_k$. To join mapgen way, for each pair of itemsets M and $P \in F_k$ where each F_k itemset has the two parts "< itemset, transaction id list >", the following three conditions have to be satisfied: M joins with P to get itemset $M \cup P$ if the following conditions are satisfied.
(a) itemset M comes before itemset P in F_k,
(b) the first k-1 items in M and P (excluding just the last item) are the same,
(c) the transaction id list of the new itemset $M \cup P$ represented as $Tid_{M \cup P}$ is obtained as the intersection of the Tid lists of the two joined k-itemsets M and P and thus, $Tid_{M \cup P} = Tid_M \cap Tid_P$.
The formal algorithm TidFp is presented as Algorithm 1.

Algorithm 1. *(TidFp:Computing Frequent Patterns with Tids)*

Algorithm TidFp()
Input: A list of 1-items, Transaction Table of 1-items,
minimum support s%.
Output:A list of frequent patterns Fps.
Other variables: candidate sets C_k, Frequent k-itemsets F_k, k = 1 initially.
begin
1. Scan the DB once to compute
$C_k = \{< item_{k1}, Tidlist_{item_{k1}} >, \ldots, < item_{km}, Tidlist_{item_{km}} >\}$.
2. Compute frequent k-itemset F_k from candidate k-itemsets
C_k as F_k = {list of k-itemset with Tidlist count $\geq$ minsupport }.
3. While ($F_k \neq \emptyset$) do
begin
3.1. k = k+1
3.2. Compute the next candidate set C_{k+1} as F_k map-gen join F_k.
i.e. Each itemset $M \in F_k$ joins with another itemset
$P \in F_k$ if the following conditions are satisfied.
(a) itemset M comes before itemset P in F_k
(b) the first k-1 items in M and P (excluding just the last item) are the same.
(c) Tid list of the two joined k-itemsets M and P is:
$Tid_{M \cup P} = Tid_M \cap Tid_P$.
3.3. For each itemset in C_k do
3.3.1. Calculate all subsets and prune if not previously large.
3.4.If $C_k = \emptyset$ then break and go to step 4
end
4.Compute all Frequent patterns as $FP = F_1 \cup \ldots F_k$
end

Table 2. Example Patient/Drugs Database Records

Drug	Side Effects
P_1	D_1 D_2
P_2	D_1 D_2 D_3
P_3	D_3 D_4
P_4	D_1 D_2 D_4

3.2 TidFp-multi: Mining FPs on Multiple Tables with Transaction Ids

One advantage of mining frequent patterns with transaction ids is that it facilitates discovering more meaningful knowledge from not just a single database table but a database with a multiple of related tables. It can also be extended to distributed databases having partitioned tables distributed across a number of sites. For example, consider a pharmacovigilance database, which has reports about adverse events of certain drugs from medical professionals and patients and there is need to answer queries from two related database tables Drug/Side Effects (as in Table 1) and Patient/Drug shown as Table 2. We might be interested in answering with frequent pattern mining, questions like the following, which will not be easily answered with simple SQL or stored procedure queries.
1. How many people have various patterns and frequent patterns of adverse effects given minimum 50% total occurrence?
2. How many people use frequent combinations of products having minimum total occurrence of 50%?
3. Which drugs have dangerous combinations of adverse effects?
Answering query 1 above requires finding the TidFp of Table Drug/Side Effect (Table 1) to get $TidFp_{Drug}$, also finding TidFp of Table Patient/Drug (Table 2) to get $TidFp_{Patient}$ and getting the count of $TidFp_{Drug} \cap TidFp_{Patient}$. Query 2 is answered by mining TidFp on table Patient, while query 3 is answered by mining TidFp on Table Drug.

Thus, mining more complex knowledge from a multiple of related tables in a database, would normally entail applying the TidFp algorithm on the individual database tables and answering the queries by either integrating the mined FPs from different tables using set operations of intersection, union, minus as is suitable to answer the queries. Having the Tids with FPs makes this integration easy and possible.

3.3 TidFp-hordist: Mining Horizontally Distributed Tables with TidFp

Most existing algorithms on mining distributed databases including [3], [8] focus on privacy-preserving distributed mining of association rules, whereby the data at different sites are secure data that are not shared with other sites. Many

applications belong to the same organization (e.g., an automobile company), located at different sites and collaborative mining of distributed data at different sites would provide both local and global knowledge for marketing promotions among others. There are two main techniques for partitioning global data or table T, belonging to an organization into *f* fragments of the table based on some fragmentation criteria, to be distributed at perhaps *f* locations. First method is horizontal fragmentation where each horizontal fragment, F_i, has a number n_{F_i} of records of the global table T such that $\sum_{i=1}^{f} n_{F_i} = |T|$ meaning that the sum of the number of records in all f fragments will give back the cardinality of the global table T. On the other hand, each vertical fragment of T has only some attributes of T but has the same cardinality as T giving that the sum of the arity (number of attributes) in each of the f vertical fragments F_i, is the same as the arity of the global table T, that is, $\sum_{i=1}^{f} arity_{F_i} = arity_T$. Thus, although the global data T, is distributed either horizontally or vertically, the goal of distributed frequent pattern mining given a minimum support s threshold is to find all global frequent patterns GFPs and local frequent patterns LFPs that meet the minimum support threshold.

An Existing Algorthm on Distributed Mining
The FDM algorithm [3] for distributed FP mining first generates global candidate k-itemset $CG_{i(k)}$ by apriori-gen joining of global large (k-1)-itemsets at site *i*, $GL_{i(k-1)}$ with itself. The global (k-1)-itemsets at each site *i*, $GL_{i(k-1)}$, are obtained by intersecting the global large (k-1)-itemsets with the local large (k-1)-itemsets $LL_{i(k-1)}$. Next, the local database is scanned to prune the itemsets in the candidate k-itemset $CG_{i(k)}$ whose local support is less than the minsupport $s\%$, while the rest are put in the local large k-itemsets at site *i*, $LL_{i(k-1)}$. Each site then broadcasts its local large k-itemsets $LL_{i(k)}$ to all sites, the union of all the $LL_{i(k)}$ will give the $LL_{i(k)}$ from where each site computes the support of items in $LL_{i(k)}$, which are broadcasted to all sites so that they can combine them to compute the global frequent itemsets G_k.

The Proposed TidFp-Hordist Algorithm
The difference between the proposed TidFp-Hordist algorithm approach for mining horizontally distributed table and other approaches like those of the FDM [3] summarized above, is that the TidFp-Hordist takes advantage of the Tid's of each Fp when forming global large itemsets and thus, requires only one initial broadcasting of the first local frequent 1-itemsets from all sites to each site and global frequent 1-itemset GFP_1 is computed as the union of all local $LFP_{i(1)}$ itemsets while the next global candidate (k+1)-itemset, C_{k+1} is computed as global GFP_k map-gen join GFP_k. Thereafter, subsequent global GFP_k and candidate C_k are computed without any further broadcast and support counting from local databases. The formal TidFp-Hordist algorithm is presented as Algorithm 2.

Algorithm 2. *(TidFp-Hordist:FPs with Tids in Horizontally Distributed DBs)*

Algorithm TidFp-Hordist()
Input: A list of 1-items, a number i of sites,
a set of i horizontally fragmented Transaction
Tables of 1-items DB_i, minimum support $s\%$.
Output:A list of global frequent patterns GFps.
Other variables: global candidate sets GC_k, global
Frequent k-itemsets GF_k, local Frequent k-itemsets LF_k
k = 1 initially.
1. Scan each local DB_i once to compute
$LC_{i(k)} = \{< item_{k1}, Tidlist_{item_{k1}} >, \ldots, < item_{km}, Tidlist_{item_{km}} >\}$.
2. Compute each local i frequent k-itemset $LF_{i(k)}$
from local candidate k-itemsets $LC_{i(k)}$ as
$LF_{i(k)}$ = {list of k-itemset with Tidlist count $\geq$ minsupport}.
3. Let each site i broadcast its local frequent $LF_{i(k)}$.
4. Compute global GF_k as itemsets in the union of all local $LF_{i(k)}$
with support $\geq s\%$ of global $|DB|$.
5. At each site i, while global ($GF_{i(k)} \neq \emptyset$) do
begin
5.1. k = k+1
5.2. Compute the next global candidate set GC_k from GF_{k-1}
as GF_{k-1} map-gen join GF_{k-1}.
5.3. For each itemset in global GC_k do
5.3.1. Calculate all subsets and prune if not previously large.
5.4.If $GC_k = \emptyset$ then break and go to step 6
5.5. Compute GF_k as itemsets in GC_k with support count $\geq$ minsupport.
end
6.Compute all global Frequent patterns as
$GFP = GF_1 \cup \ldots GF_k$
end

Application of the TidFp-Hordist Algorithm
EXAMPLE 2: Given the two horizontally fragmented tables shown as Tables 3 and 4, which are equivalent to the example database of Table 1, and a minimum support threshold of 50%, use the TidFp-Hordist algorithm to obtain the global frequent patterns GFP, across the two distributed tables at two sites.

SOLUTION 2: Applying the algorithm TidFp-Hordist to the two horizontally distributed database tables above at minsupport of 50% to mine global frequent patterns GFPs would entail executing the steps of the algorithm as follows: Step 1,

Table 3. Horizontally Distributed Drug Table 1

Tid (Drug)	Items (Side Effects)
D_1	1 3 4
D_2	2 3 5

Table 4. Horizontally Distributed Drug Table 2

Tid (Drug)	Items (Side Effects)
D_3	1 2 3 5
D_4	2 5

we compute the local for site 1 candidate $C_{1(1)}$= { $< 1, T_1 >$, $< 2, T_2 >$, $< 3, T_1, T_2 >$, $< 4, T_1 >$, $< 5, T_2 >$ }. Then, we compute the local for site 2 candidate $C_{2(1)}$= { $< 1, T_3 >$, $< 2, T_3, T_4 >$, $< 3, T_3 >$, $< 5, T_3, T_4 >$ }. Step 2, we compute the local frequent $LF_{i(k)}$ as: $LF_{1(1)}$ = { $< 1, T_1 >$, $< 2, T_2 >$, $< 3, T_1, T_2 >$, $< 4, T_1 >$, $< 5, T_2 >$ }. $LF_{2(1)}$={ $< 1, T_3 >$, $< 2, T_3, T_4 >$, $< 3, T_3 >$, $< 5, T_3, T_4 >$ }. Step 3 entails each site having all local $LF_{i(k)}$ through broadcast. Step 4, we now get global GF_1 = itemsets in $\cup_{i=1}^{2} LF_{i(k)}$ with support of 50% of global DB cardinality with count of at least 2. GF_1 is from $LF_{1(1)} \cup LF_{2(1)}$ = { $< 1, T_1, T_3 >$, $< 2, T_2, T_3, T_4 >$, $< 3, T_1, T_2, T_3 >$, $< 5, T_2, T_3, T_4 >$ }. Step 5 computes the next global candidate set at each site GC_2 as: GF_1 map-gen GF_1 = { $< 1, 2, T_3 >$, $< 1, 3, T_1, T_3 >$, $< 1, 5, T_3 >$, $< 2, 3, T_2, T_3 >$, $< 2, 5, T_2, T_3, T_4 >$, $< 3, 5, T_2, T_3 >$}. The next global frequent GF_2 is computed from GC_2 as {$< 1, 3, T_1, T_3 >$, $< 2, 3, T_2, T_3 >$, $< 2, 5, T_2, T_3, T_4 >$, $< 3, 5, T_2, T_3 >$}. Back to beginning of step 5, next global candidate GC_3 = GF_2 map-gen GF_2 = {$< 2, 3, 5, T_2, T_3 >$} and the frequent GF_3 = {$< 2, 3, 5, T_2, T_3 >$}. The global GFPs = $\cup_{i=1}^{3} GFPi$, which are the same as the table mined undistributed.

3.4 TidFp-Vertdist: Mining Vertically Distributed Tables with TidFp

The version of the TidFp algorithm for mining vertically distributed database is very much like the one for mining the horizontally distributed database discussed in detail above. The difference is in how the tables are fragmented, which only affects how the supports of the local and global tables are computed. In the horizontally fragmented tables, the local support counts are different and based on the local cardinality, which is less or equal to the cardinality of the global database. On the other hand, in the vertically distributed tables, the support counts of the vertical fragments and the global database are the same. As defined in the previous section, a vertical fragment of a table has only some of the attributes of the original table but has all records of the table. For example, the example Drug Table T (Tid, Attr1, Attr2, Attr3, Attr4), can be fragmented vertically into two tables with schemas T_{v1} (Tid, Attr1, Attr2) and T_{v2} (Tid, Attr3,Attr4). Our application of the algorithm on the vertically fragmented tables produced the same correct results as well.

4 Experiments and Performance Analysis

To test the proposed TidFp algorithm, we conducted experiments to
(1) determine performance gain in terms of CPU execution time gain of the TidFp in comparison with the Apriori algorithm, which also determines Tids.

In this case, we first ran the Apriori algorithm and then scanned the database for each frequent pattern to collect the TIDs they appeared in.

(2) determine memory usages of the proposed TidFp in comparison with the Apriori algorithm.

Comparing TidFp and Apriori Execution Times and Memory Usages The two algorithms Apriori with Tid and our proposed TidFp algorithm were implemented in C++ with the same data structure in UNIX environment, where the programs are compiled with "g++ filename" and executed with "a.out". Then, the CPU execution times for the two algorithms were tested for transactional databases of different sizes of 250K (or 250 thousand) records, 500K, 750K, 1M (or 1 million), and 2M records generated with the IBM quest synthetic data generator publicly available at at http://www.almaden.ibm.com/cs/quest/ and used by other pattern mining research. The characteristics of the generated data are described as follows: $|D|$: Number of records in the database, $|C|$: Average length of the records, $|S|$: Average length of maximal potentially frequent itemset, $|N|$: number of items (attributes).

With the average length of records (C) in our data as 10, and number of attributes (N) as 10 with S as 5, a full description of one set of experimental data with number of records as 250 thousand is C10.S5.N20.D250K. All experiments are performed on on a more powerful multiuser UNIX SUN microsystem with a total of 16384 MB memory and 8 x 1200 MHz processor speed, which generally produces faster execution times than when run on microcomputer environment. The minimum support for the experiments range between 10% and 50%. The result of the experiments are summarized in five tables below to show:

(1) Execution time efficiencies of the algorithms at medium minimum support threshold of 40% and for different database sizes as shown in Table 5.
(2) scalability of the algorithms with a large database of 2 million records at different minimum supports with data C10.S5.N20.D2M as shown in Table 6.
(3) feasibility and scalability at a medium sized database of 500K records at different minimum supports with data C10.S5.N20.D500K as shown in Table 7.
(4) feasibility and scalability of the algorithms at a small sized database of 250K records at different minimum supports C10.S5.N20.D250K as shown in Table 8.
(5) Memory usages of the algorithms at medium minimum support threshold of 40% for different database sizes as shown in Table 9. The RES memory usage value is collected for the program process of "a.out" on UNIX with the "top -u" command.

It can be seen from the experiments that the proposed TidFp outperforms the Apriori algorithm to the tune of 25 times better in execution time and is more scalable than the Apriori algorithm and in particular, at low minimum support thresholds and large data sizes. From experiment 5 on Table 9, it can be seen though that the proposed TidFp algorithm requires more running memory than

Table 5. Execution times for different datasets at MinSupport of 40%

Algorithms	Runtime (in secs) for different Data sizes)				
	250K	500K	750K	1M	2M
TidFp	22	47	64	94	187
Apriori	542	1071	1623	2239	4495

Table 6. Execution times for dataset at different MinSupports (large data 2M)

Algorithms	Runtime (in secs) at different supports(%)				
	10%	20%	30%	40%	50%
TidFp	23236	1472	330	187	147
Apriori	crashed	39434	11571	4495	2141

Table 7. Execution times for dataset at different MinSupports (medium data 500K)

Algorithms	Runtime (in secs) at different supports(%)				
	10%	20%	30%	40%	50%
TidFp	6231	391	90	47	38
Apriori	crashed	8702	2729	1071	509

Table 8. Execution times for dataset at different MinSupports (small data 250K)

Algorithms	Runtime (in secs) at different supports(%)				
	10%	20%	30%	40%	50%
TidFp	2853	176	37	22	17
Apriori	crashed	4329	1348	542	267

Table 9. Memory Usages for Different Data Sizes at Minsupport of 40%

Algorithms	Memory Usages for Different Data Sizes				
	250K	500K	750K	1M	2M
TidFp	10MB	14MB	18MB	22MB	42MB
Apriori	2872KB	3664KB	4408KB	5424KB	10M

the Apriori algorithm but this is a reasonable tradeoff. This good performance of the TidFp algorithm is because the TidFp only needs to scan the database once, while the Apriori algorithm re-scans the database for every support counting. When the TidFp algorithm intersects the transaction IDs for 2 items sets, it uses bitmap representation. For example, the transaction ID bitmaps from the two F_1 itemsets $[< 1 > D_1D_3]$ and $[< 1 > D_2D_3D_4]$ obtained when the TidFp executes the operation $[< 1 > D_1D_3]$ mapgen-Join $[< 1 > D_1D_3D_4]$ are shown in Table 10. The Tid list of the resulting 2-itemset $< 1, 2 >$ obtained by

Table 10. Bitmaps for Itemsets $< 1 >$, $< 2 >$ and $< 1, 2 >$ Tids

Itemset	Transaction id Bitmap			
	D_1	D_2	D_3	D_4
$< 1 >$	1	0	1	0
$< 2 >$	0	1	1	1
$< 1, 2 >$	0	0	1	0

mapgen-joining the two 1-itemsets $< 1 >$ and $< 2 >$ is shown as the third row of Table 10. The Tid list of the resulting (k+1)-itemset is obtained from intersecting the Tid lists of the two joining k-itemsets, which is accomplished with bitmap AND operation.

5 Conclusions and Future Work

This paper proposes an intuitive approach for mining frequent patterns with transaction ids, which is useful for addressing the needs of several applications including mining multiple related tables in a database for more informative knowledge discovery. The paper also introduced versions of this algorithm for mining horizontally and vertically distributed databases and multiple related tables in a database. It has also been shown through experiments that TidFp execution time is up to 25 times better than the Apriori algorithm. It has also been shown that both number of database scans needed for support counting and communication costs are drastically reduced when this approach is employed. This approach is extendible to other types of mining like mining sequences and on pattern growth techniques for mining frequent patterns. Future work should also determine or analyze the gain made through saving broadcast delays and resources as well as execution time when mining distributed tables.

References

1. Agrawal, R., Srikant, R.: Fast Algorithms for Mining Association Rules in Large Databases. In: Proceedings of the 20th International Conference on very Large Databases Santiago, Chile, pp. 487–499 (1994)
2. Ayres, J., Flannick, J., Gehrke, J., Yiu, T.: Sequential Pattern Mining using A Bitmap Representation. In: Proceedings of the ACM SIKDD conference, Edmonton, Alberta, Canada, pp. 429–435 (2002)
3. Cheung, D.W.-L., Ng, V., Fu, A.W.-C., Fu, Y.: Efficient Mining of Association Rules in Distributed Databases. Transactions on Knowledge and Data Engineering 8(6), 911–922 (1996)
4. Ezeife, C.I., Lu, Y.: Mining Web Log sequential Patterns with Position Coded Pre-Order Linked WAP-tree. The International Journal of Data Mining and Knowledge Discovery (DMKD) 10, 5–38 (2005)
5. Han, J., Kamber, M.: Data Mining: Concepts and Techniques. Morgan Kaufmann Publishers, New York (2001)

6. Han, J., Pei, J., Yin, Y., Mao, R.: Mining Frequent Patterns without Candidate Generation: A Frequent-Pattern Tree approach. International Journal of Data Mining and Knowledge Discovery 8(1), 53–87 (2004)
7. Imielinski, T., Swami, A., Agarwal, R.: Mining association rules between sets of items in large databases. In: Proceeding of the ACM SIGMOD conference on management of data, Washington D.C., May 1993, pp. 207–216 (1993)
8. Kantarcioglu, M., Clifton, C.: Privacy-preserving Distributed Mining of Association Rules on Horizontally Partitioned Data. In: The proceedings of the ACM SIGMOD Workshop on Research Issues on Data Mining and Knowledge Discovery, DMKD 2002, pp. 24–31 (2002)
9. Pei, J., Han, J., Mortazavi-asi, B., Zhu, H.: Mining Access Patterns Efficiently from web logs. In: Proceedings, Pacific-Asia conference on Knowledge Discovery and data Mining, Kyoto, Japan, pp. 396–407 (2000)
10. Pei, J., Han, J., Mortazavi-Asl, B., Pinto, H., Chen, Q., Dayal, U., Hsu, M.C.: PrefixSpan: Mining Sequential Patterns Efficiently by Prefix-Projected Pattern Growth. In: Proceedings of the 2001 International Conference on Data Engineering (ICDE 2001), Heidelberg, Germany, pp. 215–224 (2001)
11. Srikanth, R., Aggrawal, R.: Mining Sequential Patterns: generalizations and performance improvements, Research Report, IBM Almaden Research Center 650 Harry Road, San Jose, CA 95120, 1–15 (1996)
12. Zaki, M.J.: SPADE: An Efficient Algorithm for Mining Frequent Sequences. Machine learning 42, 32–60 (2001)