Mining Association Rules from Stars

Eric Ka Ka Ng*, Ada Wai-Chee Fu*, Ke Wang+

*Chinese University of Hong Kong
Department of Computer Science and Engineering
{kkng1,adafu}@cse.cuhk.edu.hk

+Simon Fraser University
Department of Computer Science
wangk@cs.sfu.ca

Abstract

Association rule mining is an important data mining problem. It is found to be useful for conventional relational data. However, previous work has mostly targeted on mining a single table. In real life, a database is typically made up of multiple tables and one important case is where some of the tables form a star schema. The tables typically correspond to entity sets and joining the tables in a star schema gives relationships among entity sets which can be very interesting information. Hence mining on the join result is an important problem. Based on characteristics of the star schema we propose an efficient algorithm for mining association rules on the join result but without actually performing the join operation. We show that this approach can significantly out-perform the join-then-mine approach even when the latter adopts a fastest known mining algorithm.

1 Introduction

Association rules mining [AIS93, AS94] is identified as one of the important problems in data mining. Let us first define the problem for a database D containing a set of transactions, where each transaction contains a set of *items*. An **association rule** has the form of $X \Rightarrow Y$ where X and Y are sets of items. In such a rule, we require that the frequency of the set of items $X \cup Y$ is above a certain threshold called the *minsup*. The **frequency** (also known as **support**) of a set of items Z is the number of occurrences of Z in D, or the number of transactions in D that contain Z. The **confidence** of the rule should also be above a threshold. By confidence we mean the probability of Y given X.

The mining can be divided into two steps: first we find the sets of items that have frequencies above $minsup$, which we call the **frequent itemsets**. Second, from the sets of frequent itemsets we generate the association rules. The first step is more difficult and is shown to be NP-hard. In our subsequent discussion we shall focus on the first step.

The techniques in association rule mining has been extended to work on numerical data and categorical data in more conventional databases [SA96, RS98], some researchers have noted the importance of association rule mining in relation to relational databases [STA00]. Tools for association rule mining are now found in major products such as IBM's Intelligent Miner, and SPSS's Clementine.

In real databases, typically a number of tables will be defined. In this paper, we examine the problem of mining association rule from a set of relational tables. In particular we are interested in the case where the tables form a star structure [CD97] (see Figure 1). A star schema consists of a *fact table* (FT) in the center and multiple dimension tables. We aim to mine association rules on the join result of all the tables [JS00]. This is interesting because the join result typically tells us the relationship among different entities such as customers and products and to discover cross entities association can be of great value. The star schema can be considered as the building block for a snowflake schema and hence our proposed technique can be extended to the snowflake structure in a straightforward manner.

A (Student)

	address	age	GPA
a_1	Shatin	20-25	2.0-2.5
a_2	Taipo	25-30	2.5-3.0
a_3	Fotan	25-30	3.0-4.0

B (Course)

	subject	semester	credit
b_1	Maths	Fall 02	3
b_2	Phy	Spring 02	2
b_3	CS	Fall 02	3

FT

Tid(A)	Tid(B)	Tid(C)	o
a_1	b_1	c_1	o_1
a_1	b_1	c_1	o_1
a_2	b_3	c_3	o_2
a_3	b_2	c_2	o_2

C (Teacher)

	age	dept	office
c_1	30-39	CS	Rm 103
c_2	20-29	Phy	Rm 122
c_3	40-50	Mat	Rm 200

Figure 1. Star with 3 Dimensional Tables

At first glance it may seem easy to join the tables in a star schema and then do the mining process on the joined result. However, when multiple tables are joined, the resulting table will increase in size many folds. There are two major problems: Firstly, in large applications, often the join of all related tables cannot be realistically computed because of the many-to-many relationship blow up, large dimension tables, and the distributed nature of data.

Secondly, even if the join can be computed, the multifold increase in both size and dimensionality presents a hugh overhead to the already expensive frequent itemset mining step: (1) The number of columns will be close to the sum of the number of columns in the individual tables. As the performance of association rule mining is very sensitive to the number of columns (items) the mining on the resulting table can take much longer computation time compared to mining on the individual tables. (2) If the join result is stored on disk, the I/O cost will increase significantly for multiple scanning steps in data mining. (3) For mining frequent itemsets of small sizes, a large portion of the I/O cost is wasted on reading the full records containing irrelevant dimensions. (4) Each tuple in a dimension table will be read multiple times in one scan of the join result because of duplicates of the tuple in the join result.

We exploit the characteristics of tables in a star schema. Instead of "joining-then-mining", we can perform "mining-then-joining", in which the "joining" part is much less costly. Our strategy never produces the join result. In the first phase, we mine the frequent itemsets locally at each dimension table, using any existing algorithm. Only relevant information are scanned. In the second phase, we mine global frequent itemsets across two or more tables based on local frequent itemsets. Here, we exploit the following pruning strategy: if $X \cup Y$ is a frequent itemset, where X is from table A and Y is from table B, X must be a frequent itemset and Y must be a frequent itemset. Thus, the first phase provides all local frequent itemsets we need for the second phase. The difficulty lies in the second phase.

One major challenge in the second phase is how to keep track of the many-to-many relationship in the fact table without generating the join result. We make use of the feature that a foreign key for a dimension table A can appear many times in the join result, which allows us to introduce some structure to record the key once, together with a counter for the number of duplications. We also make use of the idea of semi-join in relational databases to facilitate the mining process. From these ideas we propose a set of algorithm and data structures for mining association rules on a star schema which does not involve a join step of the tables involved. Experiments show that the proposed method is efficient in many scenarios.

2 Problem Definition

Consider a relational database with a **star schema**. There are multiple **dimension tables**, which we would denote as A, $B, C, ...$, each of which contains only one primary key denoted by **transaction id (*tid*)** , some other attributes and no foreign keys. (Sometimes we simply refer to A, B, C, ..., as dimensions.) a_i, b_i, c_i denote the transaction id (*tid*) of dimension tables A, B, C, respectively. We assume that the attributes in the dimension tables are unique. (If initially two tables have some common attributes, renaming can make them different.) We assume that attributes take categorical values. (Numerical values can be partitioned into ranges, and hence be transformed to categorical values [RS98].) The set of values for an attribute is called the **domain** of the attribute.

Conceptually, we can view the dimension table in terms of a binary representation, where we have one binary attribute (or we call an "item") corresponding to one "attribute-value" pair in the original dimension table. We also refer to each tuple in A or the binary representation as a **transaction**. For example, consider Figure 2, v_1, v_2, v_3 are attribute names for dimension table A, and the value of attribute v_1 for transaction a_1 is R_2. In the conceptual binary representation in Figure 2, we have attributes for "$v_1 = R_0$", "$v_1 = R_1$" "$v_1 = R_2$", ... (we call them x_1, x_2, x_3, ..., respectively). For transaction a_1 in table A, the value of attribute x_3 is 1 ($v_1 = R_2$ is TRUE), and the values of x_1 and x_2 both equal to 0 (FALSE). In our remaining discussions, binary items (one item for each "attribute-value" pair) in the conceptual binary representation would be used. $x_{i_1}x_{i_2}x_{i_3}...$ denotes the itemset that is composed of items x_{i_1}, x_{i_2}, x_{i_3}, ... We assume an *ordering* of items which is adopted in any transaction and *itemset*. E.g. x_1 would always appear before x_2 if they exist together in some transaction or itemset. This ordering will facilitate our algorithm.

A (Student)

	v_1	v_2	v_3
a_1	R_2	R_0	R_1
a_2	R_0	R_0	R_1
a_3	R_1	R_1	R_0

conversion

	$v_1=R_0$	$v_1=R_1$	$v_1=R_2$	$v_2=R_0$	$v_2=R_1$	$v_3=R_0$	$v_3=R_1$
	x_1	x_2	x_3	x_4	x_5	x_6	x_7
a_1	0	0	1	1	0	0	1
a_2	1	0	0	1	0	0	1
a_3	0	1	0	0	1	1	0

Figure 2. Dimension Table and its Binary Representation

There is one **fact table**, which we denote as FT. FT has attributes of $(tid(A), tid(B), tid(C), ...)$, where $tid(A)$ is the tid of table A. That is, FT stores the *tid*s from dimension tables $A, B, C, \ldots$ as foreign keys. (Later we shall discuss the more general case where FT also contains some other attributes.) In an ER model, typically, each dimension table corresponds to an entity set, and FT corresponds to the relationship among the entity sets. The relationships among entity sets can be of any form: many-to-many, many-to-one, one-to-many, or one-to-one.

We are interested to mine association rules from the star structure. In particular we shall examine the sub-problem of finding all **frequent itemsets** in the table T resulting from a natural join of all the given tables ($FT \bowtie A \bowtie B \bowtie C \ldots$). The join conditions are given by $FT.Tid(A) = A.tid$, $FT.Tid(B) = B.tid$, $FT.Tid(C) = C.tid$, ... In the following discussions, when we mention frequent itemset we always refer to the frequency of the itemset in the table T. We assume that a **frequency threshold** of $minsup$ is given for the frequent itemsets. A frequent item corresponds to a frequent itemset of size one.

In our mining process, the dimension tables will be kept in the form of the VTV (**Vertical Tid-Vector**) representation [SHS00] with counts. Specifically, suppose there are T_A, T_B, T_C transactions in tables A, B, C respectively. For each frequent item x in table A, we store a column of T_A bits, the i^{th} bit is 1 if item x is contained in transaction i, and 0 otherwise. We also keep an array of T_A entries where the i^{th} entry corresponds to the frequency of tid i in FT.

3 The Proposed Method

First we present a simple example to show the idea of discovering frequent itemsets across dimension tables without actually performing the join operation. We shall use a data type called tid_list in our algorithm. It is an ordered list of elements of the form $tid(count)$, where tid is a transaction ID, and **count** is a non-negative integer. Given two tid_lists L_1, L_2, the **union** $L_1 \cup L_2$ is the list of $tid(count)$, where tid appears in either L_1 or L_2, and the count is the sum of the counts of tid in L_1 and L_2. The **intersection** of two tid_lists L_1, L_2 is denoted by $L_1 \cap L_2$, which is a list of $tid(count)$, where tid appears in both L_1 and L_2 and the $count$ is the smaller of the counts of tid in L_1 and L_2. Suppose we have 2 dimension tables A, B, and a fact table FT. The following are some of the tid_lists we shall use.

• $tid_A(x_i)$: a tid_list for x_i, where x_i is an attribute (item) of table A. In each element of $tid(count)$ in the list, tid is the id of a transaction in A that contains x_i, and $count$ is the number of occurrences of the *tid* in FT. If the tid of a transaction that contains x_i does not appear in $tid_A(x_i)$, the count of it is 0 in FT.

E.g. $tid_A(x_3) = \{a_1(5), a_3(2)\}$ means that the tids of transactions in A that contain x_3 are a_1 and a_3; a_1 appears 5 times in FT, and a_3 appears 2 times.

• $tid_A(X)$ where X is an itemset with items from A, it is similar to $tid_A(x_i)$ except x_i is replaced by X. $tid_A(x_i x_j)$ can be obtained by $tid_A(x_i) \cap tid_A(x_j)$.

• $B\_key(a_n)$: Given a tid a_n from A, $B\_key(a_n)$ denotes a tid-list of $tid(count)$, where tid is a tid from B and $count$ is the number of occurrences of tid together with a_n in FT.

E.g. $B\_key(a_1) = \{b_3(4), b_5(2)\}$ means that $a_1 b_3$ occurs 4 times in FT, and $a_1 b_5$ occurs 2 times.

• $B\_tid(x_i)$: Given an item x_i in A, $B\_tid(x_i)$ denotes a tid-list, of $tid(count)$, where tid is a tid of B, and $count$ is the number of times tid appears together with any tid a_j of A such that transaction a_j contains x_i in A.

• $B\_tid(X)$: similar to $B\_tid(x_i)$ except item x_i is replaced by an itemset X from A.

Example 3.1 *Suppose we have a star schema for a number of dimension tables related by a fact table FT. The following figure shows 2 of the dimension tables A and B, and the projection of FT on the two columns that contains transaction ID's for A, B, but without removing duplicate tuples.*

A

Tid	Items
a1	x1,x3,x5
a2	x2,x3,x6
a3	x1,x3,x6
a4	x1,x4,x6

Tid(A)	Tid(B)
a1	b5
a1	b3
a1	b2
a3	b2
a3	b5
a1	b5

FT

B

Tid	Items
b1	y1,y3,y5
b2	y1,y3,y6
b3	y2,y4,y6
b4	y1,y4,y5
b5	y1,y4,y6

Some of the tids from the dimension tables may not appear in FT (e.g. a_2, b_1). Suppose minsup is set to 5. We first mine frequent itemsets from each of A and B. For example, x_1 and x_3 appear together in a_1, a_3, the total count of a_1, a_3 in FT is 6, hence $x_1 x_3$ is a frequent itemset. Next we check if a frequent itemset from A can be combined with a frequent itemset from B to form a frequent itemset of greater size. $y_1 y_6$ is a frequent itemset from B with frequency = 5. We want to see if $x_1 x_3$ can be combined with $y_1 y_6$ to form a frequent itemset, the steps are outlined as follows:

1. $tid_A(x_1) = \{a_1(4), a_3(2)\}$, $tid_A(x_3) = \{a_1(4), a_3(2)\}$, $tid_A(x_1 x_3) = tid_A(x_1) \cap tid_A(x_3) = \{a_1(4), a_3(2)\}$, $tid_B(y_1 y_6) = \{b_2(2), b_5(3)\}$.
2. $B\_key(a_1) = \{b_2(1), b_3(1), b_5(2)\}$, $B\_key(a_3) = \{b_2(1), b_5(1)\}$.
3. $B\_tid(x_1 x_3) = B\_key(a_1) \cup B\_key(a_3)$ $= \{b_2(2), b_3(1), b_5(3)\}$.
4. $B\_tid(x_1 x_3) \cap tid_B(y_1 y_6) = \{b_2(2), b_5(3)\}$.
5. *The combined frequency = total count in the list* $\{b_2(2), b_5(3)\}$ *= 5.*

Hence the itemset $x_1 x_3 y_1 y_6$ is frequent. □

In general, to examine the frequency for an itemset X that contains items from two dimension tables A and B, we can do the following. We examine table A to find the set of transactions T_1 in A that contain the A items in X. Next we determine the transactions T_2 in B that appear with such transactions in FT. Note that this is similar to the derivation of an intermediate table in a **semi-join** strategy, where the result of joining a first table with the key of a second table are placed, the key of the second table is a

foreign key in this intermediate table. In the mean time, the set of transactions T_3 in B that contain the B items in X are identified. Finally T_2 and T_3 are intersected, and the resulting count is obtained.

The use of tid_list is a compressed form of recording the occurrences of tid's in the fact table. Multiple occurrences would be condensed as one single entry in a tid_list with the count associated.

Initial Step: In order to do the above, we need to have some initial information about $tid_A(x_i)$ for each item x_i in each dimension table A. One scan of a dimension table can give us the list of transactions for all items. In one scan of FT we can determine all the counts for all transactions in all the dimension tables. In the same scan, we can also determine $B\_key(a_i)$ for each $tid\ a_i$ in each dimension table.

3.1 Overall Steps

For simplicity, let us first assume that there are 3 dimension tables A, B, C. The overall steps of our method are:

Step 1 : Preprocessing
Read the dimension tables, convert them into VTV (Vertical Tid-Vector) format with counts (see Section 2).

Step 2 : Local Mining
Perform local mining on each dimension table. This can be done with any known single table algorithm with a slight modification of refering to the counts of transactions in FT which has been collected in the initial step described in Section 2. The time taken for this step typcially is insignificant compared to the global mining steps.

Step 3 : Global Mining

Step 3.1 : Scan the Fact Table
Scan the Fact Table FT and record the information in some data structures.

We set an ordering for A, B, C. First we handle tables A and B with the following 2 steps:

Step 3.2 : Mining size-two itemsets
This step examines all pairs of frequent items x and y, which are from the two different dimension tables.

Step 3.3 : Mining the rest for A and B
Repeat the following for $k = 3, 4, 5$..... Candidates are generated by the union of pairs of mined itemsets of size $k - 1$ differing only in the last item. The technique of generation would be similar to FORC [SHS00]. Next count the frequencies of the candidates and determine the frequent itemsets.

After Steps 3.2 and 3.3, the results will be all frequent itemsets formed from items in tables A and/or B. This can be seen as the frequent itemset mined from a single dimension table AB. Similar steps as Steps 3.2 and 3.3 are then applied for the tables AB and C to obtain all frequent itemsets from the star schema.

3.2 Binding multiple Dimension Tables

We can easily generalize the overall steps above from 3 dimension tables to N dimension tables. Suppose there are totally N dimension tables and a fact table FT in the star schema. We start with two of the dimension tables, say A and B. We apply Steps 3.2 and 3.3 above to mine all frequent itemsets with items from A and/or B without joining the tables with FT. This set of itemsets is called F^{AB}. We call Steps 3.2 and 3.3 a **binding** step of A and B. After binding, we treat A and B as a single table AB and begin the process of binding AB with another table, this is repeated until all N dimensions are bound. Some notations we shall use are:

- F^A denotes the set of frequent itemsets with items from A, F^{AB} denotes the set of frequent itemsets with items from tables A and/or B. F^{AB_k} denotes the set of frequent itemsets of the form XY, where X is either empty set or an itemset from A, and Y is either an empty set or an itemset from B with size k. E.g. $F^{AB} = \{x_1, y_1, x_1y_1\}$. E.g. $F^{AB_2} = \{y_1y_2, y_2y_3, x_1y_1y_2, x_1y_2y_3\}$.
- F^{A_k} denotes the set of frequent itemsets of size k from A. $F^{A_iB_j}$ denotes the set of frequent itemsets in which the subset of items from A has size i and subset of items from B has size j. E.g. suppose x_i's are items from table A, and y_j's are items from table B, we may have $F^{A_2B_1} = \{x_1x_2y_1, x_3x_4y_2\}$.

After performing "binding", we can treat the items in the combined itemsets as coming from a single dimension table. For example, after "binding" A and B, we *virtually* **combine** A and B into a single dimension table AB, and all items in F^{AB} are from the new dimension table AB. We always "bind" 2 dimension tables at each step, and iterate for $N - 1$ times if there are totally N dimension tables. At the end all frequent itemsets will be discovered.

Figure 3 shows a possible ordering of the "bind" operations on four dimension tables: A, B, C, D.

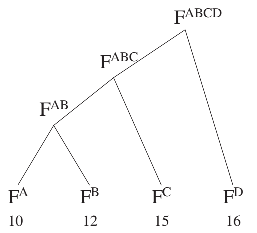


Figure 3. An example of "binding" order

We need to do two things to combine two dimension tables: (1) To assign each combination of tid from A and tid from B in FT a new tid, and (2) to set the tid in the tid_lists for items in AB to the corresponding new tid.

Consider an example in Figure 4, for a FT relating to

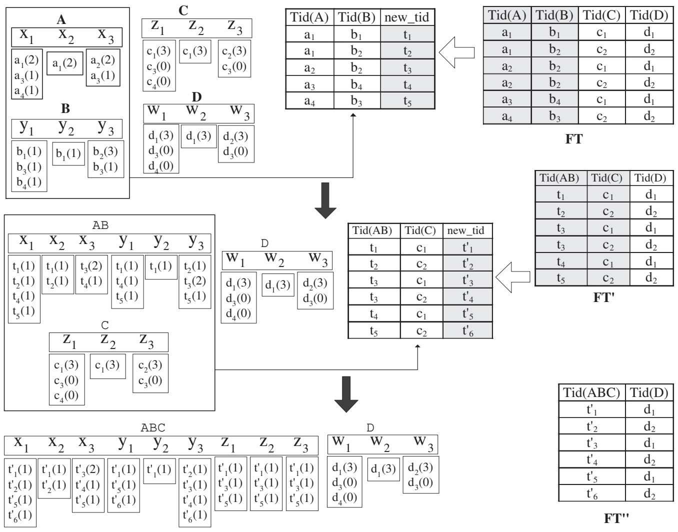


Figure 4. Concatenating tids after "binding"

4 dimensions A, B, C, D, after "binding" A and B, the columns storing tid(A) and tid(B) would be concatenated. Each combination of tid(A) and tid(B) would be assigned a new tid. A and B would be combined into AB. For example, before "binding", item x_1 appears in transactions a_1, a_3 and a_4. $tid_A(x_1) = \{a_1(2), a_3(1), a_4(1)\}$. After "binding", since a_1 corresponds to new tid t_1 and t_2, a_3 corresponds to new tid t_4, a_4 corresponds to new tid t_5. Therefore $tid_A(x_1)$ is updated to $tid_{AB}(x_1) = \{t_1(1), t_2(1), t_4(1), t_5(1)\}$.

Similarly, when F^{AB} is then "bound" with F^C, AB is combined with C and FT would be updated again. Note that in Figure 4, the tables with attribute new_tid and the multiple fact tables are not really constructed as tables, but instead stored in a structure which is a prefix tree.

We always bind a given dimension table with the result of the previous binding because the tid of the dimension table allows us to apply the technique of a foreign key as described in the previous section. The ordering can be based on the estimated result size of natural join of the tables involved, which can in turn be estimated by the dimension table sizes. A heuristic is to order the tables by increasing table sizes for binding.

3.3 Prefix Tree for FT

In Step 3.1 of the overall steps, the fact table FT is scanned once and the information is stored into a data structure which can facilitate the mining process. The data structure is in the form of a *prefix tree*. Each node in the prefix tree has a label (a tid) and also a counter. We need only scan FT once to insert each tuple into the prefix tree. Suppose we have 3 dimensions A, B, C, and a tuple is a_3, b_2, c_2, we enter at the root node and go down a child with label a_3, from a_3 we go down to a child node with label b_2, from b_2 we go to a child node labeled c_2. Every time we visit a node, we increment the counter there by 1. If any child node is not found, it is created, with the counter initialized to 1. Hence level n of the prefix tree corresponds to tid's of the n^{th} *dimension table* that would be "bound". When searching for a foreign tid_list, we can go down the path specified by the prefix. In this way, the *foreign key* and the global frequency in the i^{th} iteration can be efficiently retrieved from the $i+1^{th}$ level of the *prefix tree*. Figure 5 shows how a fact table is converted to a prefix tree.

order of join
A,B,C,D

Tid(A)	Tid(B)	Tid(C)	Tid(D)
a_1	b_1	c_1	d_1
a_1	b_2	c_2	d_2
a_2	b_2	c_1	d_1
a_2	b_2	c_2	d_2
a_3	b_4	c_1	d_1
a_4	b_3	c_2	d_2

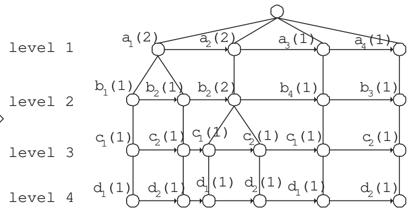


Figure 5. Prefix Tree structure representing FT

Tid(AB)	Tid(C)	Tid(D)
t_1	c_1	d_1
t_2	c_2	d_2
t_3	c_1	d_1
t_3	c_2	d_2
t_4	c_1	d_1
t_5	c_2	d_2

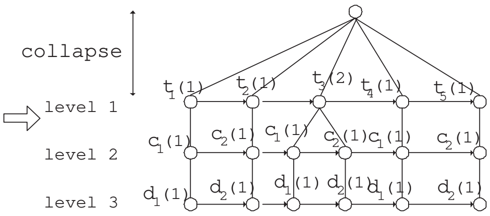


Figure 6. Collapsing the prefix tree

Use of the prefix tree – the foreign key: The prefix tree is a concise structuring of FT which can facilitate our mining step. When we want to "bind" F^A with F^B, we have to check whether an itemset (e.g. x_1) in F^A can be combined with an itemset in F^B (e.g. y_1). We need to obtain the information of a foreign key in the form of tid_list (e.g. $B\_tid(x_1)$). Let $tid_A(x_1) = \{a_1(2), a_2(1)\}$. We can find $B\_key(a_1)$ by searching the children of a_1 which are labeled $b_1(1)$, $b_2(1)$, similarly let $B\_key(a_2) = \{b_2(2)\}$. As a result, $B\_tid(x_1 y_1) = B\_key(a_1) \cup B\_key(a_2) = \{b_1(1), b_2(3)\}$.

Collapsing the prefix tree: Suppose A and B are bound, AB is the derived dimension. If B is not the last dimension to be bound, we can collapse the prefix tree by one level. A new root node is built, each node at the original second level becomes a child node of the new rootnode. The subtree under such a node is kept intact in the new tree. Figures 5 and 6 illustrate the collapse of one level in a prefix tree.

To facilitate the above, we create a horizontal pointer for each node in the same level so that the nodes form a linked list. A unique AB_tid is given to each of the nodes in the second level, which corresponds to the collapsed table AB. These unique tids at all the levels can be assigned when the prefix tree is first built.

Updating tid: We need to do the following with the collapse of the prefix tree. After binding two tables A and B, a "derived dimension" AB is formed. We update the tid_lists stored with the frequent itemsets and items that would be used in the following iteration, so that all of them are referencing to the same (derived) dimension table. For example, $tid_A(X)$ or $tid_B(Y)$ are updated to $tid_{AB}(X)$ or $tid_{AB}(Y)$.

3.4 Maintaining frequent itemsets in FI-trees

In both local mining and global mining (Steps 2 and 3), we need to keep frequent itemsets as they are found from which we can generate candidate itemsets of greater sizes. We keep all the discovered frequent itemsets of the *same size* in a tree structure which we call an **FI-tree** (FI stands for Frequent Itemset). Hence itemsets of $F^{A_3B_1}$ is mixed with itemsets of $F^{A_2B_2}$, the first one belongs to F^{AB_1}, the second belongs to F^{AB_2}.

4 Related Work

Some related work can be found in [JS00], where the joined table T is computed but without being materialized. When each row of the joined table is formed, it is processed and thereby storage cost for T is avoided. In the processing of each row in the table, an array that contains the frequencies for all candidate itemsets is updated. As pointed out by the authors, all itemsets are counted in one scan and there is no pruning from one pass to the next as in the apriori-gen algorithm in [AS94]. Therefore there can be many candidate itemsets and the approach is expensive in memory costs and computation costs. The empirical experiments in [JS00] compare their approach with a base case of applying the apriori algorithm on a materialized table for T. It is shown that the proposed method needs only 0.4 to 1 times the time compared to the base case. However, there are new algorithms in recent years such as [HPY00, SHS00] which are shown by experiment to often run many times faster than the apriori algorithm. Therefore, the approach in [JS00] may not be more efficient than such algorithms.

5 Experiments

We generate synthetic data sets in a similar way as [HPDW01]. First, we generate each dimension table individually, in which each record consists of a number of attribute values within the attribute domains, and model the existence of frequent itemsets. The parameters are listed in the following table:

D	number of dimensions
n	number of transactions in each dimension table
m	number of attributes in each dimension table
s	largest size of frequent itemsets
t	largest number of transactions with a common itemset
d	domain size of each attribute (same for all attributes)

The domain size d of an attribute is the number of different values for the attribute, which is the number of items derived from the attribute-value pairs for the attribute.

The value of n is set as 1000 in our experiment. After generating transactions for each dimension table, we generate FT based on parameters in the following table:

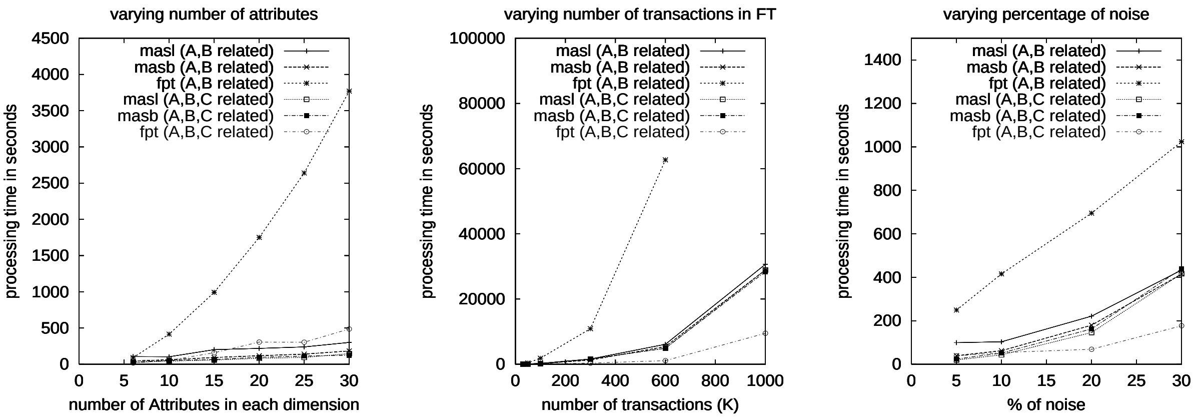
sup	target frequency of the association rules
$\|L\|$	number of maximal potentially frequent itemsets
N	number of noise transactions

In constructing FT, there can be correlations among two or more dimension tables so that some frequent itemsets contain items from multiple dimensions. For the case of two dimensions, we want the *tids* associated with the same group of transactions with common frequent itemsets from one dimension table to appear at least sup times together with another group of *tid* sharing common frequent itemsets from another dimension table. In doing so, frequent itemsets across dimensions from these 2 groups would appear with a frequency count greater than or equal to sup, after joining the two dimension tables and FT. We repeat this process for $|L|$ times, so that $|L|$ maximal potentially frequent itemsets would be formed (by **maximal**, we mean that no superset of the itemset is frequent).

In order to generate some random noise, transactions which do not contain frequent itemset are generated. N rows in FT are generated, in which each tid from the dimension tables is picked randomly.

We compare our proposed method with the approach of applying FP-tree algorithm [HPY00] on top of the joined table T. We assume T is kept on disk and hence requires I/O time for processing. FP-tree requires two scanning of T during the FP-tree mining. The I/O time required is up to 200 seconds in our experiments. It turns out that the table join time is not significant compared to the mining time.

All experiments are conducted on SUN Ultra-Enterprise Generic_106541-18 with SunOS 5.7 and 8192MB Main Memory. The average disk I/O rate is 8MB per second. Programs are written in C++. We calculate the total execution time of mining multiple tables as the sum of required



(a) (b) (c)

Figure 7. Running time for (A,B) related and (A,B,C) related datasets

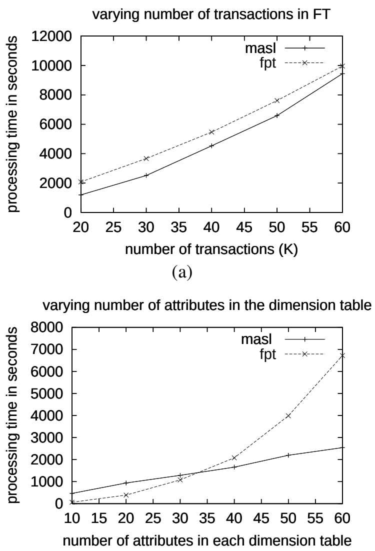
CPU and I/O times, and that of mining a large joined table as the CPU and I/O times for joining and FP-tree mining.

For the local mining step in our approach, we use a vertical mining technique as in FORC [SHS00]: frequent itemsets of increasing sizes are discovered The tid_list of the itemsets are used to generate size $k+1$ frequent itemsets from size k frequent itemsets, which is by intersecting the tid_lists of pairs of size k frequent itemsets with only one differing item. In the experiments, we compare the running time of **masl** (our proposed method, implementing tid_list as a linked list structure), **masb** (our proposed method, implementing tid_list as a fixed-size bitmap and an array of count), and **fpt** (the join-before-mine approach with FP-tree) with different data setting in 3 dimension tables A, B, C and a fact table FT. In most cases, $masb$ runs slightly faster than $masl$, but needs about 10 times more memory.

In the first dataset, we model the situation that items in A and B are strongly related, such that frequent itemsets contain items across A and B, while items in C are not involved. In such cases, transactions containing frequent itemsets from A and B can be related to Br transactions in C randomly. Br is set to 100 in all of our experiments reported here. (We have varied the value of Br and discovered little change in the performance.) The default values of other parameters used are :

number of transactions in the joined table	50K
number of attributes in each dimension table	10
size of each attribute domain	10
random noise	10%
max. size of potentially maximal frequent itemset	8

When we increase the number of items, the running time of fpt increases steeply, while that of both $masl$ and $masb$ would increase almost linearly. Running time of FP-tree



(a) (b)

Figure 8. Running time for mixture datasets

grows exponentially with the depth of the tree, which is determined by the maximum number of items in a transaction. In this case, performance of our proposed method outperforms FP-tree, especially when the number of items in each transaction is large. (see Figure 7(a))

When the number of transactions in the joined table increases, running time of both methods would increase greatly. $masl$ and $masb$ are about 10 times faster than fpt (see Figure 7(b)). We also vary the percentage of random noise being included in the datasets, (see Figure 7(c)), both $masl$ and $masb$ are faster than fpt.

In the second dataset, we model the case that A, B, C are all strongly related, so that maximal frequent itemsets always contain items from all of A, B and C. Compared with the previous dataset, performance of our approach does not vary too much, while the running time of fpt is faster in some cases (Figures 7(b) and (c)). With the strong correlation, there would be less different patterns to be considered and the FP-tree will be smaller. However, we believe that in real life situation such a strong correlation will be rare.

In real life application, there are often mixtures of relationships across different dimension tables in the database. In the third group of dataset, we present data with such mixture. In particular, 10% of transactions contain frequent itemsets from only A, B, C, respectively, 15% contain frequent itemsets from AB, BC, AC respectively, 10% contain frequent itemsets from ABC, and 15% are random noise. We investigate how the running times of $masl$ and fpt vary against increasing number of items in each transaction, and increasing number of transactions in T.

In Figure 8(a), we vary the number of transactions from 20K to 60K, while keeping the number of attributes in each dimension table to be 30 In Figure 8(b), we vary the number of attributes in each dimension table from 10 to 60, and keep the number of transactions to be 30K. In this case, running time of fpt grows much faster than our approach. This demonstrates the advantage of applying our method, which would be more significant when we have more dimension tables so that the number of items in the *T* will be large.

6 Conclusion

We propose a new algorithm for mining association rules on a star schema without performing the natural join. We show by experiments that it can greatly outperform a method based on joining all tables even when the naive approach is equipped with a state-of-the-art efficient algorithm.

Our proposed method can be generalized to be applied to a **snowflake** structure, where there is a star structure with a fact table FT, but a dimension table can be replaced by another fact table FT' which is connected to a set of smaller dimension tables. We can consider mining across dimension tables related by FT' first. We then consider the result as a single derived dimension, and continue to process the star structure with FT. This means that we always mine from the "leaves" of the snowflake.

Our current experiments assume that the data structures we use can be kept in main memory. It will be of interest to study the case where disk memory is required for the intermediate steps. Since disk access is more expensive and our intermediate structure has been designed to be more compact than the fact table, we expect good performance to be found for the proposed approach in such cases. We have not examined in our experiments the cases where the number of dimension tables is large. More study will be needed for these considerations.

In general, FT can contain attributes other than *tid* from dimension tables. In this case, we group all these attributes, put them in a separate new dimension table, and apply the same techniques to mine it as other dimension tables.

Acknowledgements This research is supported by the RGC (the Hong Kong Research Grants Council) grant UGC REF.CUHK 4179/01E. We thank Yin Ling Cheung for her help in the experiments and performance analysis.

References

[AIS93] R. Agrawal, T. Imilienski, and A. Swami. Mining association rules between sets of items in large datasets. Proceedings of the ACM SIGMOD International Conference on the Management of Data, pp 207-216, 1993.

[AS94] R. Agrawal and R. Srikant. Fast algorithm for mining association rules. Proceedings of the 20th VLDB Conference, pp 487-499, 1994.

[CD97] S. Chaudhuri and U. Dayal. An Overview of Data Warehousing and OLAP Technology. ACM SIGMOD Record, Vol. 26 No.1, pp 65-74, March 1997.

[HPDW01] J. Han, J. Pei, G. Dong, and K. Wang. Efficient computation of iceberg cubes with complex measures. Proceedings of the ACM SIGMOD International Conference on the Management of Data, pp 1-12, 2001.

[HPY00] J. Han and J. Pei and Y. Yin. Mining Frequent Patterns without Candidate Generation. Proceedings of the ACM SIGMOD International Conference on the Management of Data, pp 1-12, 2000.

[JS00] V.C. Jensen and N. Soparkar. Frequent Itemset Counting across Multiple Tables, Proceedings of Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD), pp 49-61, 2000.

[RS98] R. Rastogi and K. Shim, Mining Optimized Association Rules with Categorical and Numeric Attributes, Proc. of International Conference on Data Engineering (ICDE), pp 503-512, 1998.

[SHS00] P. Shenoy, J.R. Haritsa, S. Sudarshan. Turbo-charging vertical mining of large databases. Proceedings of the ACM SIGMOD International Conference on Management of Data, pp 22-33, 2000.

[SA96] Ramakrishnan Srikent, Rakesh Agrawal, Mining Quantitative Association Rules in Large Relational Tables, Proceedings of the ACM SIGMOD International Conference on Management of Data, pp 1-12, 1996.

[STA00] S.Sarawagi, S. Thomas, R. Agrawal Integrating association rule mining with relational database systems: Alternatives and implications. Data Mining and Knowledge Discovery, 4(2/3), 2000. (Also appeared in SIGMOD, pp 343-354, 1998.)