Pattern Mining on Stars with FP-Growth

Andreia Silva and Cláudia Antunes

Instituto Superior Técnico
Av Rovisco Pais 1
1049-001 Lisboa
{andreia.silva,claudia.antunes}@ist.utl.pt

Abstract. Most existing data mining (DM) approaches look for patterns in a single table. Multi-relational DM approaches, on the other hand, look for patterns that involve multiple tables. In recent years, the most common DM techniques have been extended to the multi-relational case, but there are few dedicated to star schemas. These schemas are composed of a central fact table, linking a set of dimension tables, and joining all the tables before mining may not be a feasible solution. This work proposes a method for frequent pattern mining in a star schema based on FP-Growth. It does not materialize the entire join between the tables. Instead, it constructs an FP-Tree for each dimension and then combines them to form a super FP-Tree, that will serve as input to FP-Growth.

1 Introduction

While most existing data mining approaches look for patterns in a single data table, multi-relational data mining (MRDM) approaches look for patterns that involve multiple tables (relations) from a relational database or data warehouse. In recent years, the most common types of patterns and approaches considered in data mining have been extended to the multi-relational case and MRDM now encompasses multi-relational (MR) association rule discovery, MR decision trees and MR distance-based methods, among others. MRDM approaches have been successfully applied to a number of problems in a variety of areas[4].

From those works, just a few are dedicated to frequent itemset mining on star schemas [2,8,10].

This work aims to find frequent patterns in a set of tables of a data warehouse, following a star schema, without materializing the join of its tables.

A data warehouse is a subject-oriented, integrated, time-variant, and non-volatile collection of data in support of management's decision-making process [7]. In terms of data modeling, a data warehouse consists of one or several dimensional models that are composed of a central fact table and a set of surrounding dimension tables, each corresponding to one of the dimensions of the fact table. The most used dimensional model is the star schema, which consists of multiple dimension tables that are associated by foreign keys to a central fact table.

At first glance, it may seem easy to join the tables of a star schema, and then do the mining process on the joined result[8]. However, when multiple tables

are joined, the resulting table will be much larger and the mining process more expensive and time consuming. There are two major problems: First, in large applications, often the join of all related tables cannot be realistically computed because of the distributed nature of data, large dimension tables and the many-to-many relationship blow up. Second, even if the join can be computed, the multifold increase in both size and dimensionality presents a huge overhead to the already expensive pattern mining process:
(1) the number of columns will be close to the sum of the number of columns in the individual tables.
(2) If the join result is stored on disk, the I/O cost will increase significantly for multiple scanning steps;
(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 joined result. The number of times that a tuple appears in the fact table is the number of times the whole tuple will be read in the joined result.

One of the great potential benefits of MRDM is the ability to automate the mining process to a significant extent. Fulfilling this potential requires solving the significant efficiency problems that arise when attempting to do data mining directly from a relational database, instead of a single pre-extracted flat file[3].

The proposed algorithm is an adaptation of FP-Growth [5] to mine a star schema. The main idea is to adapt the construction of the FP-Tree, so that FP-Growth can run.

Like FP-Growth, it scans each table only twice: first to count the support of each item, and second to construct the FP-Tree. It is divided in 3 stages:

1. **Support Counting:** The fact table is scanned to count the support of each foreign key.
2. **Local Mining:** An FP-Tree is constructed for each dimension table (DimFP-Tree), with a slight modification of the original FP-Tree, taking into account the support calculated in the previous step.
3. **Global Mining:** The FP-Trees of each dimension are combined to form a Super FP-Tree, according to each fact and an established order of dimensions. This Super FP-Tree is then mined with FP-Growth, without a change, giving all the frequent patterns.

Several orders for the dimensions were studied and are presented and compared in this work.

The rest of this paper is organized as follows. Section 2 presents the related work on MRDM on star schemas. The proposed algorithm is described on section 3. Section 4 gives some experimental results and section 5 presents the conclusions.

2 Related Work

The work related to multi-relational pattern mining on star schemas is increasing. Experiments showed that the approach of “mining before join” outperforms the approach of “join before mining” even when the latter adopts known to be fastest single-table mining algorithms[8].

Jensen and Soparkar (2000) presented an Apriori based algorithm [2], that first generates frequent itemsets in each single table using a slightly modified version of Apriori[1], and then looks for frequent itemsets whose items belong to distinct tables via a multi-dimensional count array. It does not construct the whole joined table but processes each row as the row is formed, thus storage cost for the joined table is avoided. However, the number of candidates generated explodes as the number of dimensions, attributes and values increase.

Ng et al.(2002) proposed an efficient algorithm without actually performing the join operation [8]. They perform local mining on each dimension table, and then “bind” two dimension tables at each iteration, i.e. mine all frequent itemsets with items from two different tables without joining them. After binding, those two tables are virtually combined into one, which will be “binded” to the next dimension table. They use vertical data format, and a prefix tree to compress the fact table and to speed up the calculation of support.

Xu and Xie (2006) proposed MultiClose [10], which discover frequent closed itemsets without materializing join tables. It first converts the dimension tables to vertical data format, and then mines each of them with a closed algorithm. After local mining, frequent closed itemsets are stored in two-level hash table result trees, and the frequent closed itemsets across two tables are discovered by traversing those result trees.

Several multi-relational methods have been developed by the Inductive Logic Programming community over the recent years, but they are usually not scalable with respect to the number of relations and attributes in the database. Therefore they are inefficient for databases with complex schemas. Another drawback of the ILP approaches is that they need the data in the form of prolog tables.

There are other algorithms for finding multi-relational frequent itemsets, however they just consider one common attribute at a time. They would have to run as much times as the number of dimensions, since there is no attribute common to all the tables. Instead, in a star schema, the fact table has one attribute in common with each dimension, and the dimensions have no common attribute between them. The patterns discovered by those algorithms will not reflect the relationships between dimensions.

The proposed algorithm also mines star schemas without computing the entire join nor materializing join tables. After constructing an FP-tree for each dimension, the corresponding tables are discarded. The trees already take into account the minimum frequency and incorporate transaction ids. The super FP-tree that represents the whole star is then constructed, by aggregating the dimension trees all together, based on the fact table. Finally, this tree in mined using the known pattern growth method, FP-Growth[5].

3 Mining Stars

Consider a relational database modeled as a star schema.

There are multiple dimension tables, which we will denote as A, B, C, ..., each containing only one primary key, denoted by transaction id (*tid*), some other attributes and no foreign keys. In fact, what we want is to discover potentially useful relationships among the attributes, other than primary keys. The set of values for an attribute is called the domain of the attribute.

In order to simplify our discussion we assume the fact table, denoted as FT, only contains the tids from dimension tables as foreign keys (tid_A, tid_B, tid_C, ...). If it contains some fields other than primary keys, we can place them into an extra dimension table and insert a new foreign key corresponding to it into the fact table. They are considered facts or measures.

As example, lets consider a dataset of the real movies database used in the experiments, only with three dimensions: Award (A), Studio (B) and Movie (C). Table 1 presents a conceptual representation of the dimension tables, where a_i, b_i and c_i denote the *tid* of dimension tables A, B and C, respectively, and x_i, y_i and z_i denote each possible value of A, B and C. Table 2a shows the fact table.

tid_A	**Itemsets**	**Support**
a_1	x_1	3
a_2	$x_2x_3x_4$	1
a_3	$x_5x_6x_7$	5
a_4	$x_8x_6x_7$	1

(a) Table A

tid_B	**Itemsets**	**Support**
b_1	y_1	2
b_2	$y_2y_3y_4$	1
b_3	$y_5y_3y_6$	1
b_4	$y_7y_3y_6$	2
b_5	y_8y_3	1
b_6	$y_9y_3y_6$	2
b_7	$y_{10}y_{11}y_6$	1

(b) Table B

tid_C	**Itemsets**	**Support**
c_1	z_1z_2	1
c_2	z_3z_4	1
c_3	z_5z_4	1
c_4	z_6z_4	1
c_5	z_7	1
c_6	z_8z_2	1
c_7	z_9z_4	1
c_8	$z_{10}z_{11}$	1
c_9	$z_{12}z_4$	1
c_{10}	$z_{13}z_4$	1

(c) Table C

Table 1: Dimension Tables

tid_A	tid_B	tid_C
a_3	b_2	c_1
a_3	b_4	c_2
a_3	b_6	c_3
a_3	b_4	c_4
a_1	b_1	c_5
a_1	b_5	c_6
a_3	b_7	c_7
a_1	b_1	c_8
a_2	b_3	c_9
a_4	b_6	c_{10}

(a) Fact Table

$Itemsets_A$	$Itemsets_B$	$Itemsets_C$
$x_6x_7x_5$	y_3	–
$x_6x_7x_5$	y_3y_6	z_4
$x_6x_7x_5$	y_3y_6	z_4
$x_6x_7x_5$	y_3y_6	z_4
–	–	–
–	y_3	–
$x_6x_7x_5$	y_6	z_4
–	–	–
–	y_3y_6	z_4
x_6x_7	y_3y_6	z_4

(b) Denormalized Fact

Table 2: Fact table and the frequent itemsets corresponding to each *tid*

This example will be used to show how the proposed algorithm works, with a minimum support equals to 40% of the database. The sample of the database has 10 transactions, therefore 40% of it corresponds to 4 transactions. This means that an itemset is frequent if its support is no less than 4 transactions.

Let $I = \{i_1, i_2, \ldots, i_m\}$ be a set of distinct literals, called items. A subset of items is denoted as an itemset. A transaction $T = (t_{id}, X)$ is a tuple where t_{id} is a transaction-id and X is an itemset in I. Each table, in a relational database D, is a set of transactions. The *support* (or occurrence frequency) of an itemset It, is the number of transactions containing It in the database. It is frequent if its support is no less than a predefined minimum support threshold, σ.

In a database modeled as a star schema, where there are several tables, we have to be more specific: the *local support* of an itemset It, with items belonging to a table A (A.localSup(It)), is the number of occurrences of It in A. For example, on table 1a, A.localSup(x_1) $= 1$ and A.localSup(x_6) $= 2$. The *global support* (or just *support*) of an itemset It is the number of transactions of the fact table containing all the *tid*s that contain It, as in equation 1:

$$globalSup(It) = \sum_{tid}^{tid(It)} FT.localSup(tid) \tag{1}$$

Following the example above, globalSup(x_1) $=$ FT.localSup(a_1) $= 3$ and globalSup(x_6) $=$ FT.localSup(a_3) $+FT$.localSup(a_4) $= 5 + 1 = 6$.

3.1 The Algorithm

Star FP-Growth mines multiple relations for frequent patterns in a database following a star schema. The result is the same as mining the joined table, but without materializing it.

It is based on FP-Growth [5], and the main idea is to construct a Super FP-Tree, combining the FP-Trees of each dimension, so that the original FP-Growth can run and find multi relational patterns. Like FP-Growth, it scans each table only twice: first to count the support of each item and second to construct the FP-Tree.

The overall steps are:

Step 1: Support Counting: The fact table is scanned to count the support of each *tid* of each dimension. In the example, the *tid* support is shown in the third column of each dimension table (Table 1).

Step 2: Local Mining: An FP-Tree is constructed for each dimension table (DimFP-Tree), with a slight modification of the original FP-Tree, taking into account the support calculated in the previous step.

Step 3: Global Mining:

Step 3.1: Construct the Super FP-Tree: The DimFP-Trees of each dimension are combined to form a Super FP-Tree, according to each fact and an established order among dimensions.

Step 3.2: Mining the Super FP-Tree: Run FP-Growth [5], without a change, with the Super FP-Tree and the minimum support threshold. The result of this step is a list of all patterns, not just those relating to one dimension, but also those which relate the various dimensions.

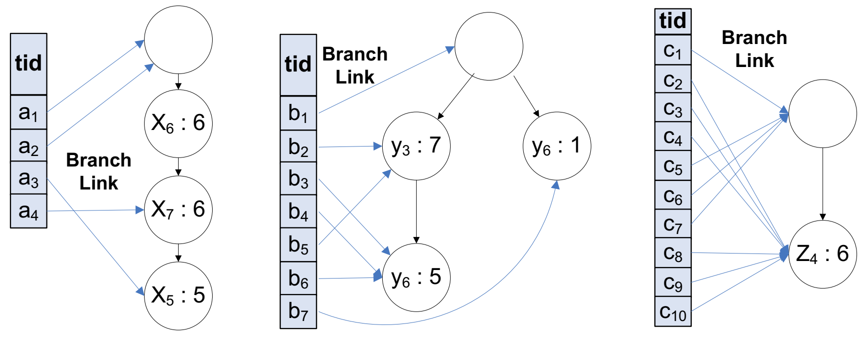
Constructing the DimFP-Tree A DimFP-Tree is very similar to an FP-Tree[5]. There are two major differences between the construction of a DimFP-Tree and an FP-Tree:

First, the support used here is the global support of each item, i.e. we consider the occurrences of an item in all database, not only in the item's table. Therefore, a node does not start with the support equals to one, but with support = support(T), with T the *tid* of the transaction that originated the node. It also is not incremented by only one, but by support(T).

For example, b_4 has a support = 2, which means that that transaction occurs two times in the database. Adding two times the same transaction with support = 1 is the same as adding it one time with support = 2. Starting a node with support = support(T) and incrementing by support(T) avoids being repeatedly inserting the same transaction.

Second, instead of the header table, the DimFP-Tree has other structure, the branch table, that keeps track of the path correspondent to each *tid*. It stores the last node of that path for each *tid*. This structure will help the global mining. If we want to know which frequent items belong to a *tid*, we follow the link in that table to find the last node, and then we just have to climb through its parents till we reach the root node. The items of the nodes in the path we took are the frequent items of that transaction.

The result is the same as if we have the table with the transactions and their frequent sorted items. However, the size of the tree is usually much smaller than its original database, therefore, not having that table materialized in memory usually saves a lot of space and avoids duplicates [6]. If we keep the table instead of the tree, and if two transactions have the same frequent items, those items will be repeated two times. With the tree, there is just one path corresponding to the two transactions. Further, shared parts can also be merged using the tree. Therefore, in a larger scale, the more transactions there are, the greater the difference.



(a) DimFP-Tree A (b) DimFP-Tree B (c) DimFP-Tree C

Fig. 1: DimFP-Trees for each dimension table

Lets consider the construction of the DimFP-Tree (figure 1b) of table B:

Step 2 starts with a first scan to the table B to calculate the support of each item. Only y_3 and y_6 are frequent, i.e. have a support no less than four transactions ($sup(y_3) = 7$ and $sup(y_6) = 6$). Therefore, only the itemsets containing just y_3 and/or y_6 would be frequent, according to the anti-monotone property[1]. For each transaction $b_1, b_2, \ldots, b_7$, frequent items are selected and sorted according to the support descending order, and then inserted in the tree. At the same time, the branch table is constructed, linking each *tid* to the respective node in the tree. b_1, for example, does not have any frequent item, therefore its branch link links to the root node. b_3 corresponds to the first path of the tree, therefore its branch link points to the last node in that path. The other transactions follow the same reasoning. Figure 1 shows the results for all dimensions.

Constructing the Super FP-Tree The Super FP-Tree is just like an FP-Tree, since it will serve as input to FP-Growth. The construction is very similar to the construction of an FP-Tree [5]. Despite this, there are three differences:

First, it is not necessary the first scan to any table to calculate the supports. They are already calculated and stored in each dimension tree.

Second, a fact is a set of *tid*s, therefore the denormalization of each fact is necessary before ordering the items or inserting them in the tree. A denormalized fact is an itemset with the items corresponding to its *tid*s. Through the branch table of each DimFP-Tree we can get the path corresponding to the itemset of each *tid*. Furthermore, according to the anti-monotone property, if an itemset is not frequent, no other itemset containing it will be. Thus, we only check the frequent items in the transactions of each *tid*, ensuring that the final tree has only the frequent items of each dimension. Table 2b, one can see the result of denormalizing each fact of our example.

And third, the ordering of items in a transaction does not have to be the frequency descending order. As verified in the improvement of FP-Growth proposed in [6], an FP-tree based on frequency descending ordering may not always be minimal.

The support descending ordering enhances the compactness of the FP-tree structure. However, this does not mean that the tree so constructed always achieves the maximal compactness. With the knowledge of particular data characteristics, it is sometimes possible to achieve even better compression.

There are two related and important properties of the FP-tree that can be derived from its construction process [6].

On one hand, given a transaction database DB, and without considering the root, the size of an FP-tree is bounded by $\sum_{T \in DB} |freq(T)|$.

The height of the tree is bounded by $max_{T \in DB}\{|freq(T)|\}$, where $freq(T)$ gives the frequent items of transaction T.

This means that the number of nodes of an FP-Tree (size) is, at most, the number of frequent items in all the transactions, and the number of levels (height) is, at most, the maximal number of frequent items in a transaction.

On the other hand, given a transaction database DB, the number of paths in an FP-tree is bounded by $|DB|$, i.e., it is, at most, the number of transactions in the database, if each transaction contributes to one different path of the FP-tree, with the length equal to the number of frequent items in that transaction.

With those lemmas in mind, several orders among dimensions were studied and compared in terms of the properties defined above. The three most relevant are the following:

1. Support descending order of items

This ordering does not have into account any order of dimensions. After denormalizing the fact, the transaction may have items from multiple dimensions. Sorting them in a support descending order may result in an itemset with items from multiple dimensions intermixed. This is the order used in the original FP-Growth. So, the tree resulting from applying this ordering is the same as the tree resulting from joining the tables in one and applying directly FP-Growth to it (but in this case, the joining is not materialized).

2. Support descending order of dimensions

As the support descending ordering enhances the compactness of the FP-tree structure, it is a promising order for the dimensions. Dimensions with higher support are more likely to be shared and thus arranged closer to the top of the FP-tree.

Since each dimension can have multiple items with different supports, we consider that the dimension support corresponds to the support of its least frequent item. Dimensions with the same support are ordered alphabetically, and the items of a dimension are also ordered in a support descending order.

Note that, with this ordering, items from multiple dimensions are not intermixed. Items from dimensions with higher support will always appear before those from dimensions with lower support.

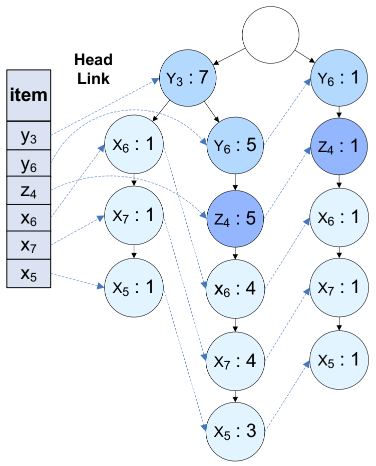


Fig. 2: Super FP-Tree with a support descending order of dimensions

In our example, the lowest support in dimensions is five in dimension A, and six in B and C. The support descending order of these dimensions is $B \rightarrow C \rightarrow A$ $((y_i s) \rightarrow (z_i s) \rightarrow (x_i s))$. Figure 2 presents the resulting Super FP-Tree. The branch table shows the items according to this ordering.

3. Path ascending order of dimensions

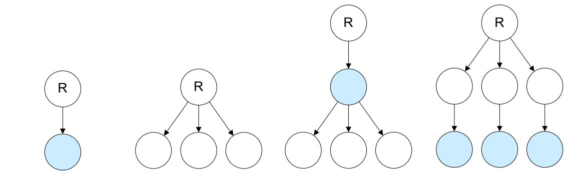
If we look at the number of paths of a tree, $|paths|$ (or just P), we can state that, when joining 2 or more trees of different $|paths|$ (i.e. adding one tree to every leaf of the other), the order in which the trees are joint influences the size of the resulting tree. Note that the $|paths|$ of the joined tree is always the same and equals to $\prod_{t \in TS} |P(t)|$, where TS is the set of the trees we want to join. The number of paths of a tree is the same as the number of leafs.

Its size (without the root) is given by

$$\sum_{i=1}^{|TS|} \left(\prod_{j=1}^{i-1} P(t_j) \right) \times size(t_i) \tag{2}$$

where $P(t)$ gives the number of paths in the tree t and TS is the set of trees in the order they are joint. The explanation is the following: a tree is inserted in each leaf of the tree immediately above, which in turn, was also inserted in each leaf of the preceding tree.

For example, imagine we have a tree A with 1 path, and another, B, with 3 paths (Figure 3a and 3b). If we join A with B, a copy of B is inserted in each leaf of tree A. Therefore, the resulting tree will have 4 nodes (without the root), as shown in figure 3c. Joining B with A will result in a tree with more nodes, 6 (figure 3d). This is because, the more leafs the tree above had, more copies of the tree below are needed.



(a) Tree A (b) Tree B (c) A with B (d) B with A

Fig. 3: Joining two trees

As we stated, joining trees in a path ascending order, will result in the smallest tree. However, joining two DimFP-Trees is not that linear. We may have to insert one tree not just in the leafs of the other, but also in a middle node, if that node corresponds to the last node of one transaction. Expressions above are also valid for this case, if we consider that $P(t)$ gives the number of nodes that correspond to the last node of a transaction.

Summing, this ordering consists in joining DimFP-Trees in a path ascending order. Dimensions with the same number of paths are ordered alphabetically, and the items of one dimension are ordered in a support descending order, like the other orderings.

These orderings have been applied to the movies database and the results are presented in section 4. Note that the set of frequent items is independent of the order applied. The result is the same for every orderings.

4 Experimental Results

The Super FP-Tree is the main structure of this algorithm. This is the tree that will be mined with FP-Growth, and this is the tree that holds the patterns we want to find. Therefore, the Super FP-Tree is the central object of these experiments.

Our goal is to analyze the impact of different orderings on the performance of our algorithm. In order to do that, the three orderings for the construction of the Super FP-Tree are compared, varying the minimum support threshold and the time spent in each step is analyzed.

The dataset is a real movies database[9]. The real database has six dimensions, with different numbers of records, from about 20 to 11000, and with a fact table with about 11000 transactions.

Among the data, we encounter a description of the directors, producers and awards received for each film and time information about them.

To achieve reliable results, the data was split into five equal datasets. The tests were applied to each dataset and we considered the average of each local result. Therefore, we analyze about 2000 facts at each time.

When comparing the size of the Super FP-Tree (figure 4), i.e. the number of nodes, the tree that is more compact is the one resulting from applying some order to the dimensions. In terms of compression, the support descending and path ascending orders of dimensions are very similar, and better than the support descending order of items. In this experiments, this ordering gave always the tree with more nodes. This difference happens because assigning an ordering for the dimensions, taking into account their characteristics and the properties described above, increase the number of shared nodes, and therefore, the compactness of the Super FP-Tree.

Note that the support descending order of items gives the same results as constructing an FP-Tree from the flat table (resulting from the join of the star), therefore, it serves as a reference to the other orderings.

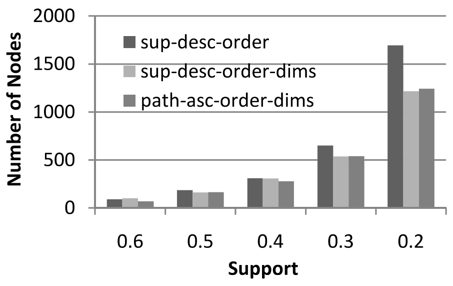


Fig. 4: Average size of the Super FP-Tree

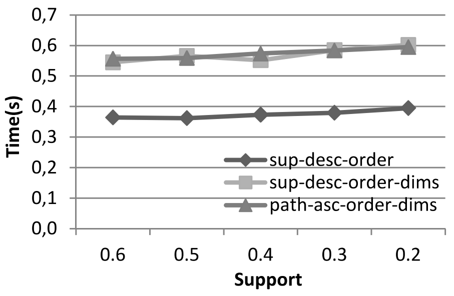


Fig. 5: Average time on Super FP-Tree construction

In terms of the number of paths in the Super FP-Tree, the resulting trees are very similar. Although the support descending order of items gives the less compact tree, it gives a tree with slightly less paths than the other orders.

The average time spent in the mining process was also studied. On average, 98% of the time is spent in the construction of the DimFP-Trees (step 2), and counting the global support (step 1) only takes 0,20% of the time. The difference between the application of the three orderings is mostly seen in step 3. As can be seen in figure 5, the support descending order of items takes less time constructing the Super FP-Tree than the other orders. With the support descending and the path ascending orders of dimensions, each transaction of the fact table has to be ordered according to that ordering before it can be inserted in the tree, yielding the previous results.

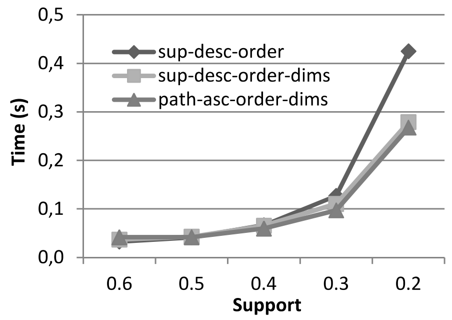


Fig. 6: Average time of FP-Growth

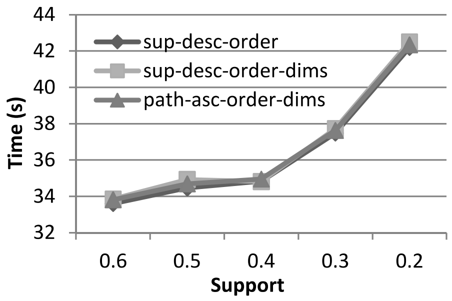


Fig. 7: Average total time on Star FP-Growth

Even though the time for the construction of the Super FP-Tree is smaller for the support descending order of items, the FP-Growth will take longer to execute (figure 6), due to resulting tree's size (figure 4 shows its size is bigger). Therefore, in the end, the total time needed for running Star FP-Growth (figure 7) is very similar for all orderings.

The size of the trees, as well as the time spent, depend not just on the size and characteristics of the data, but also on what the user wants. As minimum support decreases, the number of patterns increases, and therefore the tree and time will also increase. However, the memory needed to keep the trees will generally be much less than the memory needed to keep the tables.

The computer used to run the experiments was an Intel Xeon E5310 1.60GHz (Quad Core), with 2GB of RAM. The operating system used was GNU/Linux amd64 and the algorithm was implemented using the Java Programming language (Java Virtual Machine version 1.6.0_02). The tables were maintained in memory, as well as all the trees. However, the dimension tables were freed before Global Mining, as well as the fact table before running FP-Growth.

5 Conclusions

Star FP-Growth is a simple algorithm for mining patterns in a star schema. It does not perform the join of the tables, making use of the star properties. Building a tree for each dimension taking into account the global support of items allows us to discard the respective table and to keep only the frequent items. The main purpose is to prepare the FP-Tree that represents the data, combining the dimension trees, so that it can serve as input to FP-Growth.

Three orderings for dimensions were analyzed and the results state that applying a support descending or a path ascending order for the dimensions achieve better compression than the usual support descending order of items. The time spent in the mining process was very similar for both orderings, but it actually depends on the size and characteristics of the data, and on what we want: minimum support, performance, memory.

Using a pattern growth method and the FP-Tree gives us an important benefit: the size of an FP-tree is bounded by the size of its corresponding database because each transaction will contribute at most one path to the FP-tree, with the length equal to the number of frequent items in that transaction. Since there are often a lot of sharing of frequent items among transactions, the size of the tree is usually much smaller than its original database. Unlike the Apriori-like method which may generate an exponential number of candidates in the worst case, under no circumstances, may an FP-tree with an exponential number of nodes be generated.

If the tree cannot be maintained in main memory, several techniques can be used, whether representing and storing the tree in hard disk, or partitioning the database into a set of projected databases, and then for each projected database, constructing and mining its corresponding FP-tree [6].

The proposed algorithm can also 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 other dimension tables. We can consider mining across dimension tables related by FT' first. Then consider the resulting Super FP-Tree as a derived DimFP-Tree and continue processing the star structure with FT. This means that mining a snowflake starts from their "leaves".

References

1. Agrawal, R., and Srikant, R. Fast algorithms for mining association rules in large databases. In *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases* (Sept. 1994), pp. 487–499.
2. Crestana-Jensen, V., and Soparkar, N. Frequent itemset counting across multiple tables. In *PADKK '00: Proceedings of the 4th Pacific-Asia Conference on Knowledge Discovery and Data Mining, Current Issues and New Applications* (London, UK, 2000), Springer-Verlag, pp. 49–61.
3. Domingos, P. Prospects and challenges for multi-relational data mining. *SIGKDD Explor. Newsl. 5*, 1 (2003), 80–83.
4. Džeroski, S. Multi-relational data mining: an introduction. *SIGKDD Explor. Newsl. 5*, 1 (2003), 1–16.
5. Han, J., Pei, J., and Yin, Y. Mining frequent patterns without candidate generation. In *SIGMOD'00: Proceedings of the 2000 ACM SIGMOD international conference on Management of data* (New York, NY, USA, 2000), ACM, pp. 1–12.
6. Han, J., Pei, J., Yin, Y., and Mao, R. Mining frequent patterns without candidate generation: A frequent-pattern tree approach. *Data Mining and Knowledge Discovery 8*, 1 (2004), 53–87.
7. Inmon, W. H. *Building the data warehouse (2nd ed.).* John Wiley & Sons, Inc., New York, NY, USA, 1996.
8. Ng, E. K. K., Fu, A. W.-C., and Wang, K. Mining association rules from stars. In *In Proceedings of the 2002 IEEE International Conference on Data Mining* (2002), pp. 322–329.
9. Wiederhold, G. Movies database documentation, 1989.
10. Xu, L.-J., and Xie, K.-L. A novel algorithm for frequent itemset mining in data warehouses. *Journal of Zhejiang University - Science A 7*, 2 (2006), 216–224.