

Efficient Runtime Generation of Association Rules*

Richard Relue
Department of Mathematical
and Computer Sciences
Colorado School of Mines
Golden, Colorado 80401
rrelue@mines.edu

Xindong Wu
Department of Computer
Science
University of Vermont
Burlington, Vermont 05405
xwu@emba.uvm.edu

Hao Huang
Department of Mathematical
and Computer Sciences
Colorado School of Mines
Golden, Colorado 80401
hhuang@mines.edu

ABSTRACT

Mining frequent patterns in transaction databases has been a popular subject in data mining research. Common activities include finding patterns in database transactions, timeseries, and exceptions. The Apriori algorithm is a widely accepted method of generating frequent patterns. The algorithm can require many scans of the database and can seriously tax resources. New methods of finding association rules, such as the Frequent Pattern Tree (FP-Tree) have improved performance, but still have problems when new data becomes available and require two scans of the database.

This paper proposes a new method, which requires only one scan of the database and supports update of patterns when new data becomes available. We design a new structure called Pattern Repository (PR), which stores all of the relevant information in a highly compact form and allows direct derivation of the FP-Tree and association rules quickly with a minimum of resources. In addition, it supports run-time generation of association rules by considering only those patterns that meet on-line data requirements.

1. INTRODUCTION

Frequent pattern mining plays an essential role in many data mining tasks. The best-known approach is the Apriori algorithm [3, 8]. Apriori takes a transaction database, and identifies what are the frequent itemsets, which Boolean rules are possible and which are the most probable based on support and confidence levels. Apriori is an iterative approach with level-wise search. The k-itemsets found in level 1 are used to find those in level 2 and so on. Each level of search requires one full scan of the database. Frequent patterns or association mining is an essential tool for correlations,

*This research was supported in part by CASI (the Colorado Advanced Software Institute) under Technology Transfer Grant TTG-01-06.

causality, sequential patterns, etc. [3, 4, 6, 7, 9, 11, 14, 15, 16, 17, 22] in Data Mining analysis.

The problem with Apriori is the amount effort required to generate the rules. The algorithm can require too many resources for the following reasons:

1. Each iteration requires a full scan of the database. If the database is large this can take a long time.
2. More detailed rules require more iterations and thus more scans of the database. It is costly to analyze a large number of candidate sets by pattern matching.
3. When the database changes, the entire process must be repeated or some method must be used to merge the rules from old and new data.
4. The support and confidence levels are arbitrary. They are generally set high enough to get the algorithm to terminate with reasonable performance. It may reject useful rules simply because the algorithm did not go deep enough.
5. A large number of candidate sets at level 1 can generate huge candidate sets in subsequent level searches which strain memory resources.

There are various methods to improve Apriori performance. Generally these rules limit the amount of data scanned or improve the efficiency in retrieving records:

1. Hashtables to reduce retrieval time of itemset data [19].
2. Pruning the database to remove data that is not useful [18].
3. Knowledge type constraints on data examined to prevent rules which are not useful [10, 15].

While these measures improve the situation, there are still many problems:

1. Real time development of rules in response to changing requirements is not available.

2. Accuracy is sacrificed when the database is pruned.
3. Updating of the rules is still a problem when new data becomes available.
4. Resources are still an issue when the database is large.
5. Rules which are needed for a specific situation may be missed in analysis because confidence and support levels are set too high.

Consider a web site such as Amazon, which sells books. When shopping at such a site it would be useful to suggest books that the customer might want. A good association may require analysis of many factors specific to the customer to make an association that is genuinely useful. In addition making the analysis for all customers and updating this information frequently may not be practical due to the size of the database and resource requirements.

This same sort of problem can occur in other situations. One example would be a search engine on the Internet. The typical search engine uses keywords to identify URL's to list for the customer. Some search engines are more refined such as Google which uses links from potential list members to authoritative sites as an indicator of a better association. While this works better than most search engines, it still is very limited in factors that it considers for association. The size of the database is enormous and real time response is imperative.

What is required is an association method similar to Apriori, but with the following capabilities:

1. Development of rules in real time which are tailored to current requirements. Not all possible rules are considered, just those that are needed.
2. More efficient use of resources and real time required to develop the rules.
3. Data for association analysis is easy to update and use.
4. Computation process would be scalable and easy to distribute among several processors for execution.
5. Able to adjust support/confidence levels and number of rules found to suit the situation.
6. Minimize scans of the database.

Some previous work has touched on these improvements. Some of the notable algorithms are Tree Projection [2], Frequent Pattern Tree [13] and others reported in [1, 6, 10, 12, 18, 19, 23]. These methods do reduce resource requirements, but do not adequately address real-time rule development of rules and data update.

2. THE PATTERN REPOSITORY FROM A DATABASE

One way to reduce the size problem of databases is to limit our search to specific areas. Rather than looking for all possible associations, we are looking for specific ones. The problem then becomes how to efficiently identify the records that

are relevant to our data requirements and then identify association rules which target our current data requirements. We do not want to scan the entire database.

If this could be done real-time based on the current data requirements, we have significant advantages. We do not consider data and rules which do not meet the current data requirements. This is equivalent to running Apriori analysis constrained to rules only for those customers that a book selling site would get the next day. Under normal circumstances this would be impossible. It does not matter how large the database is as long as we can efficiently find the limited number of records meeting the criteria.

The second advantage is that the support and confidence settings can be modified to suit the specific situation under analysis. We can get the number and type of rules desired without performing the entire Apriori analysis again. The third advantage is that real-time analysis allows for updates of transaction data and rules on a daily basis.

Our design of the Pattern Repository attacks the above problem in the following ways.

First, create an intermediate structure to store all of the essential information of the transaction data that is compact. Ideally this structure would be small enough to fit in memory. This would allow extremely fast retrieval of itemset information.

Second, The intermediate structure must be updatable when new data becomes available. New data would simply be scanned once and added to the structure by simple modification of the previous one.

Third, previously read itemset information is permanently retained. Any given record in the database is scanned only once. No matter how often new records are added, all essential information on each itemset is maintained.

Fourth, patterns must be easy to extract from the structure. The structure must be easy to use. The itemset information must be easy to read in a useful form.

2.1 Pattern Repository

The above considerations formed the basis for the design of the Pattern Repository. The database is scanned once to convert the data and insert it into our Pattern Repository structure. As each record is inserted, a list of unique values or items found in all records is created with a count of each unique value.

From this point on when new data becomes available the new records can be inserted and the value counts updated. Thus we always know the frequency of itemset values in the database and the total count of itemsets in the database. The structure can be saved to the database for later reloading should the system be restarted.

As the records are retrieved, the Pattern Repository is created as follows. As each record is scanned, the items are checked to see if a frequency count exists. If not a new fre-

quency count is created for the item. Each item is replaced with a unique number or token id representing that item. If that item occurs in another record, the same token id is used to represent the item.

The token ids are inserted into a structure called a chain. The chain is an array of token ids representing the contents of the record. These chains are kept in a temporary list until all of the records for this cycle are read. The token ids provide a simple and precise way of representing an item. In addition, a number representation takes much less space than a string which usually represents the item.

The token ids in each chain are now sorted by numeric value in ascending order. Thus the first value in the chain is the lowest number and the last value in the chain is the highest number. Then the chains as a whole unit, are sorted in numerical order. At this point, identical chains are now next to each other in the temporary list. The list is now sorted in order and when identical chains are found, they are merged to one chain with a count matching the number of identical chains found.

The next step is to merge the new chains with any that already exist from previous import cycles. Each chain in the temporary list is checked to see if any matching chain exists in the permanent list. If it does, then the matching chain count is incremented. If it does not, then it is added to the permanent list. Note that this import cycle works the same way whether there are no records in the current structure or several million. When all the records are inserted and processed, the new permanent list is saved. This could be done using a flat file or database since the chains are just linear arrays.

The structure below illustrates the Pattern Repository as it is saved. The brackets [] indicates a single chain content. The number in parenthesis () indicates the chain count. The number in curly braces is the token id. There will be multiple token ids indicating the itemset contents. When stored in a list, the chains will be sorted numerically.

```
[(23), {10}, {23}, {40}, {55}, {67}]
[(11), {20}, {23}, {40}, {55}, {67}]
[(17), {21}, {23}, {40}, {55}]
[(44), {21}, {45}, {47}]
[(34), {47}, {67}, {81}, {92}, {98}]
```

Note the only difference between the first and second chain is the token id in the 2nd position. Thus the sort value of the second chain is higher. The count shows that this itemset in the first chain appears 23 times in the database. The values have been sorted by numeric value within the chain. Thus 10 must appear before 20 in the chain. Note that none of the chains will be repeated, they must all be unique. Any chain matching an existing one will simply increment the count.

2.2 Example Construction of Chains

Here is an example of how the Pattern Repository is constructed. Assume that there are 5 new records from the

database to be inserted. A new chain is created for each record and given a count of 1. The records can be of different length. Initially the data looks like this. The curly braces {x} represent an item in the record. The brackets [] indicate the contents of one record.

```
[ {a}, {c}, {d}, {f} ]
[ {b}, {x}, {m} ]
[ {a}, {c}, {d}, {f} ]
[ {c}, {a}, {d}, {f} ]
[ {b}, {m}, {x} ]
```

Now create a chain and replace the items with a token id. The token id values are assigned as follows. As each new unique item is discovered during the database scan, it is assigned the next unused number as a token id. As before the parentheses () denote a count while curly braces {} denote an item token id. Note that as before, the records in positions 1, 3 and 4 have identical items and the records in position 2 and 5 are identical.

```
[(1), {23}, {10}, {40}, {55}]
[(1), {11}, {83}, {45}]
[(1), {23}, {10}, {40}, {55}]
[(1), {10}, {23}, {40}, {55}]
[(1), {11}, {45}, {83}]
```

Resort the chain contents by token id values. Chains 1, 3 and 4 become identical.

```
[(1), {10}, {23}, {40}, {55}]
[(1), {11}, {45}, {83}]
[(1), {10}, {23}, {40}, {55}]
[(1), {10}, {23}, {40}, {55}]
[(1), {11}, {45}, {83}]
```

Sort chains by numerical value. At this point, identical records are next to each other.

```
[(1), {10}, {23}, {40}, {55}]
[(1), {10}, {23}, {40}, {55}]
[(1), {10}, {23}, {40}, {55}]
[(1), {11}, {45}, {83}]
[(1), {11}, {45}, {83}]
```

Now sort the list, merge identical chains and update counts.

```
[(3), {10}, {23}, {40}, {55}]
[(2), {11}, {45}, {83}]
```

At this point, the list can be merged with the permanent list. The simplest way to detect a previously existing chain would be to use a hashtable for the existing chains in the permanent list. If [(3), {10}, {23}, {40}, {55}] does not exist in the permanent list then it would be added. If it

already existed then just the count would be incremented. When the merge operation is complete, the permanent list is saved. During the save operation the occurrences of the frequent items in each record should also be recorded and saved. This allows a specific subset of the database to be selected based on the selection of specific items.

2.3 Completeness and Compactness of Itemsets

As mentioned previously, the process of inserting itemsets into the Pattern Repository starts with a chain that records all of the values in the itemset. The chain contents are sorted. The sorting ensures that even if the items do not occur in a fixed order in a set, we will detect records that are identical. Only identical records are merged with a count. Thus the record count in the Pattern Repository is exactly the same as the original database. By the method of construction, for every pattern in the Pattern Repository, there must be a least one matching pattern in the database and for every pattern in the database there will be one matching pattern in the Pattern Repository.

Given these observations, the Pattern Repository must duplicate all of the information that was in the original database. Otherwise, a contradiction occurs. If there is a pattern in the Pattern Repository that is not in the original database then the record count would not match. If there is a pattern in the database, that is not in the Pattern Repository then once again the record count would not match.

The size of chains is limited to the maximum number of items in any set from the current database. The number of chains is limited to the number of unique itemsets in the current database. Thus the size and number of chains derived is limited. Also any rules or patterns derived from the Pattern Repository will be of a limited number. This is in contrast to Apriori which can generate an exponential number of candidates.

2.4 Time Complexity

The creation of the Pattern Repository has no impact on the time required to calculate the rules, because the Pattern Repository is created before the rule development process. The time complexity of this creation process can be computed as follows.

1. n records in database scanned.
2. A chain with p items is sorted at a cost of $p * \log(p)$ per chain.
3. Sort chains numerically at a cost of $n * \log(n)$.
4. Locate the matching chain in the permanent list with hashtable is constant.

The time complexity becomes $n * n * \log(n) * p * \log(p) * \text{constant}$. Since p is much smaller than n in practice, we can consider $p * \log(p)$ a constant. Thus the time complexity reduces to $n^2 * \log(n)$.

3. PREPARING PATTERN REPOSITORY FOR USE

Once the creation process is finished, the next step is to extract the patterns in a usable form. This involves several steps to prepare the data in the Pattern Repository. At the end of these steps, a FP-Tree can be created and rule development can proceed as described in [13].

3.1 Pattern Repository Preparation

The basic steps are as follows.

First, decide whether all of the itemsets will be used or whether to filter for those that contain certain items that come from on-line data requirements. Suppose we are only interested in those itemsets that contain certain token ids.

Second, create a temporary list of chains that contain the required token ids. Then sort the chain contents by item frequency. The most frequent item will appear first in the chain and the least frequent will appear last in the chain. When items have the same frequency, use token id as a secondary sort criterion. The highest token id comes before a lower-valued one in this case.

Third, scan each chain from the bottom until a frequent item is encountered. Each item below it is an infrequent item and is removed from the chain. Infrequent items cannot be part of a valid Apriori pattern. The infrequent items are retained in the Pattern Repository because it is possible that an infrequent item may become frequent when more data is added.

Fourth, sort the chains by numerical value of their contents. Similar chains will be next to each other in the list. Build an FP-Tree by sequentially scanning the chains in order.

These methods strive to eliminate the common bottlenecks that occur in frequent pattern mining.

1. The data size is reduced to a minimum. Frequently occurring items generate nodes higher in the graph, thus will have a better chance of sharing nodes with less frequent ones.
2. Counting and accumulation are the means of getting candidates rather than exhaustive search and test. This is much less costly than the candidate generation of Apriori.
3. The candidate generation is done by divide and conquer rather than bottom up search of frequent candidates.

3.2 An Example

The following is an example of preparing the data for use as described above. We retrieve the selected chains from the stored Pattern Repository. In this example, it is assumed that all chains must contain 25. The chain contents are sorted by token id when retrieved from the Pattern Repository.

```

[(13), {10}, {25}, {34}, {56}, {68}]
[(10), {10}, {14}, {25}, {57}, {67}]
[(28), {10}, {25}, {34}, {66}, {78}]
[(67), {13}, {25}, {39}, {51}]
[(23), {10}, {21}, {25}, {45}]
[(15), {10}, {12}, {25}, {44}, {98}]

```

Suppose in the Pattern Repository, the frequency for each item is as follows.

```

{10} occurs 455 times
{12} occurs 389 times
{13} occurs 487 times
{14} occurs 415 times
{21} occurs 395 times
{25} occurs 385 times
{28} occurs 261 times
{34} occurs 15 times
{39} occurs 326 times
{44} occurs 285 times
{45} occurs 320 times
{51} occurs 291 times
{56} occurs 250 times
{57} occurs 391 times
{66} occurs 11 times
{67} occurs 17 times
{68} occurs 361 times
{78} occurs 12 times
{98} occurs 34 times

```

Sort by frequency with most frequent at the beginning of each chain. Note that frequency is for the database as a whole and not the frequency within this sample.

```

[(13), {10}, {25}, {68}, {56}, {34}]
[(10), {10}, {14}, {57}, {25}, {67}]
[(28), {10}, {25}, {34}, {78}, {66}]
[(67), {13}, {25}, {39}, {51}]
[(23), {10}, {21}, {25}, {45}]
[(15), {10}, {12}, {25}, {44}, {98}]

```

Sort chains numerically. The counts in the first position are not used in the sorting process.

```

[(15), {10}, {12}, {25}, {44}, {98}]
[(10), {10}, {14}, {57}, {25}, {67}]
[(23), {10}, {21}, {25}, {45}]
[(28), {10}, {25}, {34}, {78}, {66}]
[(13), {10}, {25}, {68}, {56}, {34}]
[(67), {13}, {25}, {39}, {51}]

```

Eliminate items which are not frequent items. Assume 40 instances are the minimum support. Tokens 34,66,67,78,98 are eliminated.

```

[(15), {10}, {12}, {25}, {44}]
[(10), {10}, {14}, {57}, {25}]

```

```

[(23), {10}, {21}, {25}, {45}]
[(28), {10}, {25}]
[(13), {10}, {25}, {68}, {56}]
[(67), {13}, {25}, {39}, {51}]

```

Finally, scan the above list and derive an FP-Tree (see Figure 1).

Since the first 5 chains contain 10, the occurrences are added together to create the 10 for the start of the graph. Since there is only one 13, the node is created directly with the count of 67. Then we proceed to the next position in the chain that contains two 25 entries after 10, and they are added to get 41. The rest are unique and the nodes are simply added with counts. This is repeated until the end of the chain is reached. When completed, a duplicate of the FP-Tree has been constructed which is identical to that would be derived by two database scans. However our version of the FP-Tree can be recreated after update of the Pattern Repository. The normal problems of updating the original database have been bypassed.

At this point, the normal rule analysis can be performed as described in [13].

3.3 Time Complexity

The time complexity of the above Pattern Repository preparation process can be computed as follows.

1. A chain with p items is sorted at a cost of $p * \log(p)$.
2. Sort chains numerically at a cost of $n * \log(n)$.
3. Scan n chains to create an FP-Tree

The time complexity becomes $n * n * \log(n) * p * \log(p) * \text{constant}$. Since p is much smaller than n we can consider $p * \log(p)$ a constant. Thus the time complexity reduces to $n^2 * \log(n)$.

3.4 Advantages and Similarities

Although the Pattern Repository does not look like a tree, it does in fact reproduce much of the information that the FP Tree [13] and Tree Projection [2] do with some significant advantages. The Pattern Repository is updatable and allows creation of a customized FP-Tree without scanning the original database. The biggest problem with the FP-Tree is handling updates and changes in frequency of itemsets. Once the relative frequencies for different items in a database have changed, a new FP-Tree will have to be reconstructed. Our construction method effectively bypasses these problems. In addition, only the needed portion of an original FP-Tree has to be created, thus minimizing memory resource requirements during analysis. The Pattern Repository can be created offline and then used in real-time to create the needed FP-Tree for frequent pattern analysis.

The main advantages of the Pattern Repository are as follows.

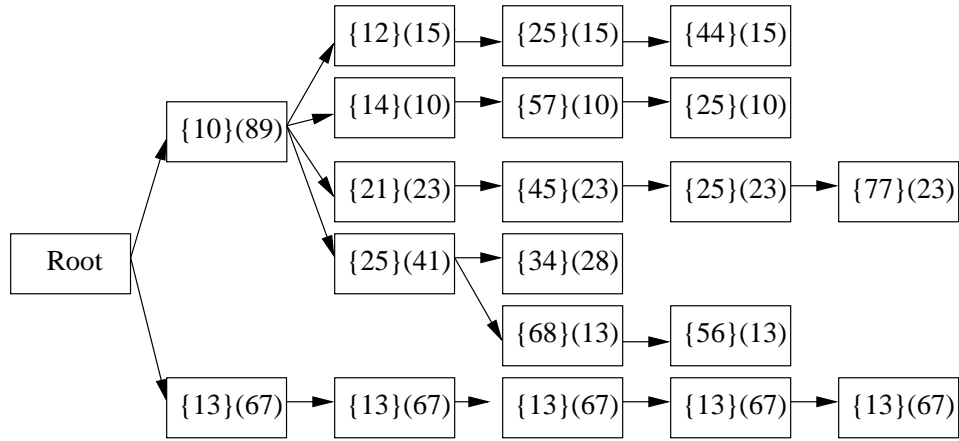


Figure 1: An FP-Tree

1. This structure is extremely easy to store and partition in a database. Thus we can examine just the part we need. The entire structure does not have to be in memory, which minimizes memory requirements.
2. The Pattern Repository is updatable whereas the other approaches mentioned are not.
3. Only one scan of the database is needed and this does not affect rule development time.

3.5 Implementation Issues

The best way to create the Pattern Repository from a large database is by batch updates with storage of the intermediate results. If a system crash occurs, the Pattern Repository update can continue from the last successfully completed batch. Several versions of the Pattern Repository could be stored to allow recovery from the last successful update.

Since the process has discrete steps, it is possible to load only those portions of the Pattern Repository necessary to perform the calculations. However if there is a lot of repetition in the itemsets, it should be possible to load the entire structure in memory.

Generally the expected use of the Pattern Repository would be to store the itemset information and recreate the necessary portions of the FP-Tree on a daily basis. Then the recreated FP-Tree could be used whenever necessary, to mine frequent patterns.

4. RUN-TIME RULE GENERATION: AN EXAMPLE

The Pattern Repository can be used to generate a complete FP-Tree or a subset of it, and subsequently generate appropriate association rules. However, a more productive use of the Pattern Repository is to generate targeted rules in real-time. Below is an example.

Remember in Section 1, a book-selling web site was mentioned. Suppose the customer has already selected two books, A and B, and we are trying to suggest other books based

on A and B. In this case, we are looking for relevant association rules where A and B imply another book C. The Pattern Repository is queried for all patterns that have A and B. This involves getting a list of patterns with A and another list with B. The final list consists of those patterns that exist in both lists. Assume that A and B map to token id 34 and 256 in the following chains.

```

[(13), {5}, {34}, {55}, {256}, {1023}]
[(10), {8}, {15}, {34}, {256}, {678}]
[(25), {5}, {34}, {55}, {256}, {1023}, {2345}]
[(5), {10}, {34}, {78}, {256}, {378}]
[(32), {7}, {34}, {55}, {256}, {1023}]
[(28), {5}, {34}, {256}]

```

Remove the infrequent items. Here we must look at the frequencies from the Pattern Repository. If the support requirement is 25 instances to become a frequent item, then tokens 2345 and 378 are eliminated. They have 20 and 15 occurrences in the database and are not frequent items. The remaining items are available for rule development.

```

[(13), {5}, {34}, {55}, {256}, {1023}]
[(10), {8}, {15}, {34}, {256}, {678}]
[(25), {5}, {34}, {55}, {256}, {1023}]
[(5), {34}, {78}, {256}]
[(32), {7}, {34}, {55}, {256}, {1023}]
[(28), {5}, {34}, {256}]

```

At this point, we could proceed as in Section 4.2 and create an FP-Tree. However, this is not necessary. We can get the results by inspection of the patterns. We will ignore 34 and 256 since they are in all of the patterns.

```

[(13), {5}, {55}, {1023}]
[(10), {8}, {15}, {678}]
[(25), {5}, {55}, {1023}]
[(5), {78}]
[(32), {7}, {55}, {1023}]
[(28), {5}]

```

{5} occurs 66 times
{7} occurs 32 times
{8} occurs 10 times
{15} occurs 10 times
{55} occurs 70 times
{78} occurs 5 times
{678} occurs 10 times
{1023} occurs 70 times

34 and 256 occurs 273 times which is just the total count for all of the matching patterns found. This meets the minimum support requirement for a frequent itemset. The next step is to determine support/confidence levels of the rules. We can now create candidate rules as follows.

{34} and {256} => {5} (66)
{34} and {256} => {7} (32)
{34} and {256} => {8} (10)
{34} and {256} => {15} (10)
{34} and {256} => {55} (70)
{34} and {256} => {78} (5)
{34} and {256} => {678} (10)
{34} and {256} => {1023} (70)

Assuming our support level requires at least 25 occurrences and a 20% confidence level is needed which translates to 55 occurrences. We can eliminate rules which occur less than 25 times. The rules ending with 8, 15, 78 and 678 fail on the minimum support. We can also eliminate 7 since it fails the required confidence level. The following rules exceed the threshold levels for both support and confidence.

{34} and {256} => {5} (66)
{34} and {256} => {55} (70)
{34} and {256} => {1023} (70)

Due to the limited resources needed for this kind of analysis, it is easy to accomplish this rule development in real-time and suggest the three book titles for the buyer.

5. CONCLUSION

This paper has designed a specialized structure, called Pattern Repository, which compacts the database itemsets into a list of chains that is updatable. An updated FP-Tree structure can be derived directly from this structure without scanning the original database. This method has several advantages. Updates of the itemsets can be done as soon as new data becomes available. Thus the rules can be developed on demand with the most recent data. No further scans of the original database are necessary. Since the rules are done on demand, the constraints and support/confidence can be adjusted as required to fit the current situation. The process is easily divided into discrete steps and rule development can be pursued just to the level needed to fit the situation. The tree size is bounded and thus the size of the problem is bounded.

The primary focus of the Pattern Repository is to keep the itemset data small enough to fit in the memory of a dedicated server or workstation. If the database is very large and does not compress much, then the Pattern Repository can be partitioned into components which are stored in a database and retrieved as needed. The creation of the Pattern Repository can be partitioned into discrete steps and allows storage of intermediate steps and freeing of computer resources.

One area of future work is to provide an efficient method to directly mine data patterns from the Pattern Repository. Since all of the information of the FP-Tree is present it should be possible to do the mining in a similar manner without actually creating an FP-Tree. This would be especially useful in cases where only a selected portion of the database is needed. The Pattern Repository could easily be partitioned into just the areas needed. Another area of future work is to combine the Pattern Repository with the adjacency lattice developed in [5] to achieve better performance in online generation of association rules.

6. REFERENCES

- [1] R. Agarwal, C. Aggarwal, and V. V. V. Prasad. Depth-first generation of large itemsets for association rules. IBM Tech. Report RC21538, July 1999.
- [2] R. Agarwal, C. Aggarwal, and V. V. V. Prasad. A tree projection algorithm for generation of frequent itemsets. *Journal of Parallel and Distributed Computing*, 2000.
- [3] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *VLDB'94*, pp. 487-499.
- [4] R. Agrawal and R. Srikant. Mining sequential patterns. In *ICDE'95*, pp. 3-14.
- [5] C.C. Aggarwal and P.S. Yu. A New Approach to Online Generation of Association Rules. *IEEE Transactions on Knowledge and Data Engineering*, **13**(2001): 527-540.
- [6] R. J. Bayardo. Efficiently mining long patterns from databases. In *SIGMOD'98*, pp. 85-93.
- [7] S. Brin, R. Motwani, and C. Silverstein. Beyond market basket: Generalizing association rules to correlations. In *SIGMOD'97*, pp. 265-276.
- [8] Ming-Syan Chen, Jiawei Han, and Philip Yu, Data Mining: An Overview from a Database Perspective, *IEEE Transactions on Knowledge and Data Engineering*, Volume 8, Number 6, December 1996, 866-883.
- [9] G. Dong and J. Li. Efficient mining of emerging patterns: Discovering trends and differences. In *KDD'99*, pp. 43-52.
- [10] G. Grahne, L. Lakshmanan, and X. Wang. Efficient mining of constrained correlated sets. In *ICDE'00*.
- [11] J. Han, G. Dong, and Y. Yin. Efficient mining of partial periodic patterns in time series database. In *ICDE'99*, pp. 106-115.

- [12] J. Han, J. Pei, and Y. Yin. Mining partial periodicity using frequent pattern trees. In CS Tech. Rep. 99-10, Simon Fraser University, July 1999.
- [13] J. Han, J. Pei, and Y. Yin. Mining Frequent Patterns without Candidate Generation. *Proc. 2000 ACM-SIGMOD Int. Conf. on Management of Data (SIGMOD'00)*, Dallas, TX, May 2000.
- [14] M. Kamber, J. Han, and J. Y. Chiang. Metarule-guided mining of multi-dimensional association rules using data cubes. In *KDD'97*, pp. 207-210.
- [15] M. Klemettinen, H. Mannila, P. Ronkainen, H. Toivonen, and A.I. Verkamo. Finding interesting rules from large sets of discovered association rules. In *CIKM'94*, pp. 401-408.
- [16] B. Lent, A. Swami, and J. Widom. Clustering association rules. In *ICDE'97*, pp. 220-231.
- [17] H. Mannila, H Toivonen, and A. I. Verkamo. Discovery of frequent episodes in event sequences. *Data Mining and Knowledge Discovery*, 1: 259-289, 1997.
- [18] R. Ng, L. V. S. Lakshmanan, J. Han, and A. Pang. Exploratory mining and pruning optimizations of constrained associations rules. In *SIGMOD'98*, pp. 13-24.
- [19] J.S. Park, M.S. Chen, and P.S. Yu. An effective hash-based algorithm for mining association rules. In *SIGMOD'95*, pp. 175-186.
- [20] S. Sarawagi, S. Thomas, and R. Agrawal. Integrating association rule mining with relational database systems: Alternatives and implications. In *SIGMOD'98*, pp. 343-354.
- [21] A. Savasere, E. Omiecinski, and S. Navathe. An efficient algorithm for mining association rules in large databases. In *VLDB'95*, pp. 432-443.
- [22] C. Silverstein, S. Brin, R. Motwani, and J. Ullman. Scalable techniques for mining causal structures. In *VLDB'98*, pp. 594-605.
- [23] R. Srikant, Q. Vu, and R. Agrawal. Mining association rules with item constraints. In *KDD'97*, pp. 67-73.