Frequent Set Mining for Streaming Mixed and Large Data

Rohan Khade  
Computer Science Department  
George Mason University  
Fairfax, VA 22030  
Email: rkhade@gmu.edu

Jessica Lin  
Computer Science Department  
George Mason University  
Fairfax, VA 22030  
Email: jessica@gmu.edu

Nital Patel  
Intel Corporation  
Chandler, AZ 85226  
Email: nital.s.patel@intel.com

Abstract—Frequent set mining is a well researched problem due to its application in many areas of data mining such as clustering, classification and association rule mining. Most of the existing work focuses on categorical and batch data and do not scale well for large datasets. In this work, we focus on frequent set mining for mixed data. We introduce a discretization methodology to find meaningful bin boundaries when itemsets contain at least one continuous attribute, an update strategy to keep the frequent items relevant in the event of concept drift, and a parallel algorithm to find these frequent items. Our approach identifies local bins per itemset, as a global discretization may not identify the most meaningful bins. Since the relationships between attributes change over time, the rules are updated using a weighted average method. Our algorithm fits well in the Hadoop framework, so it can be scaled up for large datasets.

Keywords—frequent set mining, discretization, large data, mixed data, streaming.

I. INTRODUCTION

Frequent itemset mining has received a great amount of attention from researchers and practitioners in the past few decades. Being useful in its own right for data exploration and summarization, it is also an important subroutine for other data mining tasks such as association rule mining. An example of its application is in the semiconductor manufacturing domain. Huge amounts of data are collected during the manufacturing process of semiconductor devices, and the amount of data collected increases exponentially as the demand for higher performance and smaller devices increases. Since the data is large and recipes to new chips do not change on a daily basis, the behavior of this data is predictable and can be assumed to be “normal”. At times we can see changes which are either persistent or short-term anomalies. Persistent changes indicate a concept drift, which needs to be recorded. Anomalies, on the other hand, are changes in the data that are short term. An example of an anomaly is an unexpected low yield for a batch. In order to detect the anomalies, one often needs to understand the typical “normal behavior” of the data. The discovery of frequent items or itemsets containing heterogeneous data in a streaming environment provides a mechanism for analysts to better understand how processes interact, or identify potential causes for a particular batch to have a low yield.

Let $I = \{i_1, i_2, \ldots, i_n\}$ be a set of all items. An itemset $C$ is a subset of $I$. Let $T$ be a dataset of transactions. If $|T|$ is the number of rows in the dataset then support $s$ of an itemset $C$ is the number of times $C$ occurs in $T$:

$$s(C) = \frac{\text{count}(C)}{|T|}$$ (1)

The goal of frequent set mining is to find items that co-occur in a dataset with a support greater than a specified threshold $\delta$. Finding the combinations of items that satisfy the minimum support threshold (i.e. frequent itemsets) is typically the first step of association rule mining. A brute-force method to calculate the support values of all itemsets has an exponential time complexity; therefore, there have been many algorithms and data structures proposed to improve the efficiency at this stage. In the seminal paper by Agarwal and Srikant [3], the authors proposed building a lattice of frequent items. The root node is null, which is connected to itemsets of size 1. Each successive level is a combination of frequent itemsets from the previous stages. Itemsets are pruned if they are infrequent. Due to the anti-monotone property, which states that the superset of an infrequent item is also infrequent, we no longer need to check itemsets containing the pruned itemsets. The main disadvantage of this algorithm is the number of database scans required. Many algorithms then sprang up to optimize the above solution. Some examples are AprioriTid and AprioriHybrid [3]. Another optimization proposed is the DHP (Direct Hashing and Pruning) algorithm [14]. DHP reduces the number of itemsets to be checked, first by creating a hash tree to estimate the count of the $(k + 1)$-itemset, and second by checking that $k$ subsets of size $k$ of the $(k + 1)$-itemset are frequent. These techniques try to reduce the number of items in $T$ after each iteration, which in turn improves the efficiency. If the exact itemset support is not required, we can improve the speed by estimating the lower bound of support as shown in [6]. We can also calculate multiple supports at the same time as shown in LCM (Linear time Closed itemset Miner) [15]. Other algorithms proposed include FP-Growth [9], which creates a compressed representation of the projected database using a suffix tree, AIS (Agrawal, Imielinski, Swami) [1], which creates a lexicographical tree of itemsets, and ECLAT (Equivalence CLAss Transformation) [17], which partitions the dataset into disjoint sets.

Since frequent set mining is a well-studied topic, the question arises as to why there is a need for a new algorithm. In the next few paragraphs we discuss the limitations of existing methodologies. More specifically, we tackle three main issues:

a) The data arrives in streams: Most existing work in
frequent set mining focuses on batch processing and improving the efficiency of the algorithm. However, in many real world applications, new data are continuously generated, e.g. manufacturing processes, sensors etc. More formally stated, a streaming dataset \( \tau \) is a sequence of indefinite number of transactions \( \{T_1, T_2, T_3, \ldots\} \). Let \( T_{i,j} = \{T_i, T_{i+1}, \ldots, T_j\} \) where \( i < j \). The support \( s \) of an itemset \( C \) in a time window \([i, j]\) is:

\[
s(C_{i,j}) = \frac{\text{count}(C_{i,j})}{|T_{i,j}|}
\]  

Batch frequent itemset mining algorithms require multiple passes over the database. This is not feasible in a streaming scenario since data comes in at high volume. Some trade-off in accuracy may be needed in order to process the data in a timely manner. In general, streaming algorithms work using the window model. In landmark window, the goal is to find itemsets between a fixed start point \( st \) and current time \( t \). The other option is to use a sliding window where we are only interested in items found in a time frame. For example, if the window width is \( w \), the goal is to find itemsets in the time \([t - w + 1, t]\). Sometimes newer data are more important than older data, in which case a damped window model [7] assigns higher weights to recent data by defining a decay rate. Time Tilted windows [8] are used to find frequent items over a set of windows. Importance is given to newer data and hence granularity is adjusted as data arrive. Since our goal is to find the general behavior of the data, we use a sliding window strategy, assigning equal weights to all the data.

There are two types of algorithms, true positive algorithms and true negative algorithms. A true positive algorithm does not allow false negatives. The seminal work by Manku and Motwani [13], called lossy counting, uses a user-defined parameter \( \epsilon \) and the result contains no itemsets with a support lower than \( \delta - \epsilon \). The main disadvantage of this approach is that the number of itemsets that needs to be checked increases exponentially to guarantee the lower bound. On the other hand, true negative algorithms [16] do not find false positives, but have a high probability of finding true frequent itemsets. The main limitations of these algorithms are that (1) it is difficult to implement them in parallel; (2) they cannot deal with continuous attributes; and (3) the space requirement may be too high if the data are very large.

b) The data is large: The usual way to deal with data that arrive at a high volume and speed is to distribute disjoint sets of data across nodes. In a shared memory scenario, we have a single memory space but multiple computational nodes. An advantage for this system is easy communication between nodes; however, a major challenge is keeping track of reading and writing in the memory space. In distributed memory systems, computational nodes do not share memory, and they communicate by passing messages. A recent paradigm for distributed memory system is MapReduce, which consists of two components: the mapper and the reducer. The mapper reads input data and sends key-value pairs to the reducer. The reducer aggregates and processes common keys, and writes the result to the output.

Count Distribution (CD) and Data Distribution (DD), proposed in [2], are parallel versions of the Apriori algorithm. Both CD and DD perform multiple passes over the data. In CD, data is split horizontally among nodes. Local candidate itemsets of size \( k \) in each partition are found and then combined to find global count. The result is then fed back to the nodes to find \( k+1 \) itemset. In DD, instead of partitioning the data, candidate itemsets are assigned to each node and the node is responsible for finding supports for their assigned itemsets by distributing the entire data among each node in a round robin way. For very large database, this is not feasible since it requires multiple passes over the data.

Hadoop, an open-source software framework based on MapReduce, has gained much popularity in recent years and is used widely to solve big data problems. Many algorithms take advantage of this framework with good success. PFP [11] is a parallel implementation of FP-Growth, which works by separating the data into group-dependent transactions. A major disadvantage of the FP-Growth algorithm is the high computational complexity to build the suffix tree and hence, to repeatedly rebuild it for streaming data may not be feasible. SPC, FPC and DPC [12] are Apriori-based algorithms which work level-wise and require multiple passes over the database. DPC tries to reduce the number of scans by combining multiple levels for each scan of the database. These algorithms are unsuitable for streaming data since the data arrive at a fast rate. Furthermore, exact counting may not be important in many streaming applications. Hadoop is a batch processing system and does not work well with streaming data.

c) The data has mixed attributes: There is considerably less work on the mining of quantitative frequent itemsets, i.e. discovery of itemsets where some of the attributes are continuous attributes. A straightforward approach is to use unsupervised binning such as equi-width or equi-frequency binning. These discretization techniques are fast, but often cannot find the best bins. To take attributes relationship into consideration, an alternative unsupervised discretization is clustering. However, clustering as a discretization method can be expensive, and choosing the appropriate number of bins is a non-trivial problem. A notable algorithm called Multivariate Discretization (MVD) [5] divides the ranges of continuous attributes into small bins. It then checks adjacent bins to see if they are different based on a statistical test. MVD takes into consideration the interaction between attributes; it performs well if the initial partitions are chosen well. However, MVD is not parallelizable, and its high computational cost makes it unsuitable for large datasets. All the algorithms discussed above are global discretization methods. In other words, there is one fixed binning for each attribute or combination of attributes. A global discretization scheme might not work well since attributes may have different distributions when coupled with different attributes. Alternatively, in our previous work, we proposed a local discretization technique to identify the best bins for each combination of items [10].

Based on our previous work on adaptive discretization [10], in this work we extend the approach to address the issues explained above. The remainder of the paper is further divided into the methodology, experimental results and conclusion.

II. METHODOLOGY

Our proposed algorithm can be summarized in Fig.1. Data arrives continuously and is stored in the current window pane.
Once we collect enough data, we run the frequent itemset mining algorithm along with previously learned instructions to find itemsets. Previously learned instructions may include rules that are trivial, and the bin boundaries of the continuous attributes. Using the updating strategy as explained later in this section, we update the Frequent Item database. We do not go into details of the user interaction with the system but note that users can interact with the rule database to visualize the frequent items, build models to analyze the data or even modify the database for future runs of the algorithm. We have implemented the algorithm in Hadoop. In the next few subsections we will describe the detail of each step.

### A. Sliding window

As new data arrive, the distribution and relationships may change among the attributes. We need to detect this change and update the rule database. The main goals are to remove rules that are no longer relevant, and to add emerging rules. If we find a new rule, we use a lazy approach to add it to the database. If an old rule is no longer significant, we phase it out. Fig.2 illustrates this idea. We keep a sliding window and, within the sliding window, we have blocks of data which arrive at different times. We call these blocks *panes*. A pane contains data collected in a given time period, e.g. 24 hours. The current pane contains the newest data.

![Sliding window with 3 panes](image)

**Fig. 2.** Sliding window with 3 panes, i.e. $\lambda = 3$.

### B. Finding Frequent Itemsets

Finding frequent itemsets in a pane for categorical attributes can be done easily using Equation 1 and various techniques described in the introduction. We will discuss the more challenging problem of finding frequent itemsets containing at least one continuous attribute. We notice that with each itemset, attribute interactions may change and hence a global discretization scheme may not find the best bin boundaries.

To illustrate the limitation of a global discretization scheme, Fig.3 shows the histograms of a continuous attribute $X$ combined with two different categorical attributes, $C1$ and $C2$, of varying cardinalities. In Fig.3 (Left), the categorical attribute ($C1$) has cardinality of 2 (i.e. there are 2 distinct values denoted by different shades of grey), whereas in Fig.3 (Right), the categorical attribute ($C2$) has cardinality of 4. We notice that the “best” number of bins for $X$ is 2 when it is combined with attribute $C1$, and 4 when it is combined with attribute $C2$. A global discretization strategy may produce too many or too few bins for each scenario.

![Histograms for the continuous attribute X showing interactions with two different categorical attributes](image)

**Fig. 3.** Histograms for the continuous attribute X showing interactions with two different categorical attributes. (Left) The categorical attribute has 2 distinct values. (Right) The categorical attribute has 4 distinct values.

In addition, imagine a scenario where the distribution changes for *some* attribute-pairs in a data stream. For example, suppose the boundary between the two classes in Fig.3 shifts to the left, but the boundaries between the four classes in Fig.3 remain the same. With a global discretization scheme that discretizes attribute $X$ regardless of other attributes, we would not be able to capture such a change in its relationship with $C1$ while retaining its relationship with $C2$. We define the purity for itemset $C$ having $n$ items with at least one continuous and categorical attribute as follows.

$$purity(C) = \frac{s(C_1 \cap C_2 \cap \ldots \cap C_n)}{s(C_1 \cup C_2 \cup \ldots \cup C_n)}$$  \hspace{1cm} (3)

When we have all continuous attributes

$$purity(C) = \frac{s(C_1 \cap C_2 \cap \ldots \cap C_n)}{\max(s(C_1), s(C_2), \ldots, s(C_n))}$$  \hspace{1cm} (4)

The intuition behind these formulas is to find the most homogeneous space of the current set of attributes.

We start the discussion with a high-level description of the algorithm and then work through it with an example. The algorithm SDAD-FIS (Supervised Dynamic and Adaptive Discretization for Frequent Itemsets) is show in Fig.4. The algorithm takes $C$, a candidate itemset containing at least one continuous attribute, as input, and works in a top-down fashion to find local partitions that are “frequent.” The function $\text{partition}(C)$ divides the continuous part of the itemset into ranges (Line 9). For example, a continuous attribute with a range 0 to 100 with median 35 will be divided into [0-35] and (35-100] after the first iteration. These partitions are fed into $\text{find_combs}(p)$ as input to find combinations of ranges (Line 10). For example, if there are two continuous attributes, plotting them on a scatter plot and dividing each attribute by its median creates four regions (spaces). Let $cont$ be the number of continuous attributes, then the number of partitions is $2^{cont}$. The next step is to find the support and purity of the attributes in each region (Line 12). If the support is less than $\text{Min\_support}$ we do not need to investigate this space further (Lines 13-14). If the current space is pure, i.e. $purity(C) \geq \text{Min\_purity}$, we also stop partitioning this space further since we have found a good itemset (Lines 15-16). We add this space and categorical items (if any) to the list of frequent items FIS. If $purity(C) > purity_{parent}(C)$, we partition this region further by calling the recursive function (Lines 17-22), else . If we find a more interesting child space.
we add the child itemset to the frequent itemset list and ignore the current space. If not, we add the current space.

After we find the partitions, we try to merge similar and contiguous partitions to get more general and interpretable itemsets (Lines 26-44). We can either merge the spaces after we find the frequent items at each recursive call or after we have found all the frequent items (which we have shown). In Fig.4, we keep track of the number of recursive calls made to reach the current function, so level = 1 is the first call to the function (Line 26). After exploring all the spaces at level = 1, to merge partitions, we first sort the spaces in increasing order of size (Line 27). If we plot the continuous attributes on a scatter plot, the space created by two continuous attributes is a rectangle, and the size is the area of the rectangle; by plotting 3 continuous attributes the space is a cuboid and the size is the volume of the cuboid. In general, hyperplanes create hypercubes, and the space is called the n-volume. If n is the number of continuous attributes, and li is the range of a continuous attribute for a frequent itemset i, then n-volume can be computed by:

\[ n-volume(C) = \prod_{i=1}^{n} l_i \] (5)

We then loop through all the sorted hyper spaces. Can_Merge (Line 32) checks to see if the 2 spaces are contiguous and similar. We consider 2 spaces similar if the purity difference between them is less than a user-defined threshold. If we can merge them, we update the support, purity, hyper volume and bin boundaries accordingly (Line 33). The updated purity is the weighted average (the weight is given by the support of the 2 spaces) of the purity measures of the 2 spaces being merged. We then delete the more specific itemsets and insert the new itemset in the appropriate place. The complexity of the algorithm is similar to merge sort which is \( O(n \log_2 n) \).

In Fig.5 we show an example of discretizing an itemset \( C = \{X, Y = \text{"A"}\} \), where X is a continuous attribute, and Y is a categorical attribute with value "A". The figures show the histograms of X, and the different shades of grey encode the two possible values for Y ("A" and "B"). The darker shade denotes Y = "A". Suppose 20% of all records have value "A" for attribute Y, and 80% of them have value "B". We first divide X into two regions at the median m, and we see that purity in the left region is 0 since there are no instances of \( \{X < m, Y = \text{"A"}\} \) (i.e. the support is 0). In the right region, however, the purity is 20/50 = 0.4. We continue dividing the region where we find \( \text{Min purity} > \text{purity} > 0 \) and \( \text{support} > \text{Min support} \). Dividing again at the median, the new purity becomes 20/25 = 0.8. We continue splitting until we get a purity of at least \( \text{Min purity} \) while support remains greater than \( \text{Min support} \). In Fig.5(Left), we show all the partitions when the algorithm is run to completion when \( \text{Min supp} = 0 \) and \( \text{Min purity} = 1 \). The final partition after merging contiguous regions is shown in Fig.5 (Right).

C. Updating Policy and Frequent Itemset database

We update the Frequent Itemset database (FDB) if we detect discrepancies in the Frequent Itemset(s) (FI) between the current pane and other panes in the window using the following update policy. Consider the example in Fig.2. Let the minimum support be \( \delta \), if the size of the window is \( \lambda \) then in Fig.2 we have \( \lambda = 3 \). Now suppose we have an FI in the FDB with support \( s_{\text{FDB}} \), and we find the same FI in the current pane with support \( s_{\text{current}} \). We can update the support of the FI in the database using a simple weighted average technique.

\[ s_{\text{new}} = \frac{(\lambda - 1) * s_{\text{FDB}}}{\lambda} + \frac{s_{\text{current}}}{\lambda} \] (6)

If we find a new FI in the current pane with a support greater than \( \delta \), but it is not in the FDB, then applying the
enough to be greater than $\frac{\lambda}{s_{\text{FDB}}}$ will be zero and $\frac{1}{\lambda} s_{\text{current}}$ will not be big enough to be greater than $\delta$. As a result, the FI will not be able to enter the FDB even if it is found consistently after some time period. To overcome this problem, we keep a buffer of new FIs. Let the buffer size be $\gamma$ time period. To overcome this problem, we keep a buffer of new FIs. Let the buffer size be $\gamma$, where $\gamma < \lambda$, and let the number of panes, starting from the first pane the itemset was found to be frequent, be $n$. We calculate the support for this new item by

$$s_{\text{new}} = \frac{(n - 1) \cdot s_{\text{buffer}}}{n} + \frac{s_{\text{current}}}{n}$$

(7)

We keep the itemset in the buffer until $n > \gamma$. After this point we say the itemset is indeed frequent and add it to the FDB. We keep incrementing $n$ until $n = \lambda$, and we modify Equation 6 to

$$s_{\text{new}} = \frac{(\min(\lambda, n) - 1) \cdot s_{\text{FDB}}}{\min(\lambda, n)} + \frac{s_{\text{current}}}{\min(\lambda, n)}$$

(8)

We use this lazy approach to add or purge itemsets because our goal is to maintain a consistent FDB which is representative of the population. For example, if we find that an FI occurs in only one pane, it may be an outlier and not representative of the population. However, if we find that an itemset that has been in the FDB for a long time but now becomes infrequent, we do not want to purge it out immediately.

The method explained can be used for frequent itemsets containing only categorical attributes. Updating FI’s with continuous attributes requires a different approach. It is unlikely that the FI’s have exactly the same bin boundaries in each pane vs. the FDB due to the noise in the data. Instead of re-binning at every pane, we bin the continuous attributes by re-using the bin boundaries found earlier. We then check to see if the supports are significantly different using a chi-square test. If the supports are significantly different, we run SDAD-FIS again and put these new itemsets in the buffer. If not, then we update the supports as explained earlier. In the next run we need to check if the data conform to these new bins or if this was just an anomaly.

D. Implementing in parallel

To handle large datasets, we scale up our algorithm by implementing it in parallel using Hadoop. Note that we do not claim to have improved the existing parallel implementations (and hence no comparisons on efficiency in the experimental section), but we provide a general solution for our algorithm to exploit the Hadoop framework. By using the updating strategy for continuous attributes, we can improve the speed if we expect the relationships between attributes to be generally the same. The main idea behind Hadoop is to break the problem into smaller and non-related problems. This works perfectly for SDAD-FIS since our approach works by finding appropriate bins for every combination of attributes. In a sense, our algorithm can be used with any other algorithms described in the introduction, and used when a continuous attribute is present in the current set of items.

The mapper reads each line and tokenizes it based on the separator. The mapper then finds combinations between each token. The size of the combinations can be user-defined. We set the keys as the combinations of attribute numbers, and the values as the combinations of the tokens. Note that in this parallel implementation we do not need a search algorithm like Apriori.

At the reducer, we receive all the combination of items. These columns can either be all categorical, in which case we can directly count the frequency, or they could contain some continuous attributes where we then use SDAD-FIS to find the FI’s.

III. EXPERIMENTAL RESULTS

We discuss the results on three datasets to demonstrate how our algorithm works. We compare our results to that of MVD [5] since MVD also performs multivariate discretization for set mining.

A. Simulated data

We start our experiments by demonstrating the algorithm’s ability to find meaningful partitions when there exist attribute interactions. We create a synthetic dataset similar to those used by Bay [5] (since their datasets are not publicly available). For this experiment, we keep a minimum support of 0.05 and min purity difference 0.1 (similar to [5]). The dataset consists of two multivariate Gaussians in the shape of an “X” as shown in Fig.6a. Each Gaussian has a categorical attribute associated with it.

![Discretizing continuous attribute 1 with continuous attribute 2](image1)

(a) Discretizing continuous attribute 1 with continuous attribute 2.

![Discretization continuous attribute 1 and continuous attribute 2 with a categorical attribute](image2)

(b) Discretization continuous attribute 1 and continuous attribute 2 with a categorical attribute.

In Fig.6a we show the bin boundaries when the itemset contains only the continuous attributes. The discretization finds meaningful bins. When discretizing with the continuous and categorical attribute as in Fig.6b, the algorithm finds 4 bins that covers the entire region. Although the partitions are not ‘pure,’ these bins avoid overfitting the data. MVD, on the other hand, finds static bins with boundaries similar to Fig.6a (with a bin in the center).
TABLE I. Bin boundaries found by MVD for the Adult dataset

<table>
<thead>
<tr>
<th>Variable</th>
<th>Cut points</th>
</tr>
</thead>
<tbody>
<tr>
<td>Age</td>
<td>19, 23, 25, 29, 33, 41, 62</td>
</tr>
<tr>
<td>Hours-Per-Week</td>
<td>30, 40, 41, 50</td>
</tr>
</tbody>
</table>

Fig. 7. (a) Initial discretizing continuous attribute 1 with continuous attribute 2. (b) and (c) Discretizing continuous attribute 1 and continuous attribute 2 after concept drift. (d) Final Discretization

B. Real data

In this section, we conducted experiments on a real dataset from the UCI Machine Learning repository [4]. The dataset consists of Adult census data. It has 14 attributes and 48842 instances. There are 4 continuous attributes including age, capital-gain, capital-loss and hours-per-week. Table I shows the bins found by MVD. Although MVD finds some meaningful bins and interesting frequent itemsets such as \{age - (41 – 62), Salary > 50K\} these boundaries do not provide the complete picture when other attributes also influence the attributes.

On the other hand SDAD-FIS finds interesting rules such as \{age-(35-90), Salary>50K\}, \{age-(29-80), education-Doctorate, Salary>50K\}, and \{age-(39-90), education-Bachelors, Salary>50K\}. This suggests that \{age\} alone is not as indicative as \{age, education\} with respect to salary level, especially with specialized degree. This is not found by MVD. We also note that our algorithm may not cover the entire range of the continuous attribute but only find rules in interesting regions.

C. Streaming data

To show how our algorithm handles concept drift in streaming data, we simulated multivariate gaussians whose correlation changes as shown in Fig.7a. Consider a concept drift in which two attributes go from positively correlated to negatively correlated.

Fig.7a shows the initial bins before we start seeing a concept drift. In Fig.7b and Fig.7c we see that the distribution is no longer the same and hence the algorithm has to add new bins. In Fig.7d the attributes again reach a stable state and we do not need to re-bin the attributes.

IV. Conclusion

In this work, we propose a scalable approach to find frequent itemsets in large streaming and mixed data. The algorithm works in a top-down, recursive fashion to find regions in continuous data that result in meaningful bins. Our algorithm automatically identifies the appropriate number and sizes of bins for the continuous attribute(s) by taking attribute interaction into account. We also show how we scale it up using Hadoop. Our updating strategy is a lazy approach using a weighted sliding window which performs well even with noise.

Although the algorithm shown was developed using the Hadoop framework, we would like to improve the efficiency further by sampling. An area which we feel has potential for improving efficiency is using active learning to sample combinations of attributes to reduce the number of key value pairs exchanged between the mapper and reducer.

ACKNOWLEDGMENT

This research is partially supported by the National Science Foundation under Grant No. 1218325.

These experiments were run on ARGO, a research computing cluster provided by the Office of Research Computing at George Mason University, VA. (URL: http://orc.gmu.edu).

REFERENCES

[9] J. Han, J. Pei, and Y. Yin, "Mining Frequent Patterns without Candidate Generation" in ACM SIGMOD Conference, 2000.