SEED Miner - A Scalable Data Mining Framework

J.A.A.D De Silva ,S.A. Fernandopulle,N.A.B.P. Wickramarathne ,O.P.M.C.A. Ariyarathne, R.Chulaka Gunasekara and Amal Shehan Perera
Department of Computer Science and Engineering, University of Morautwa, Sri Lanka.

Abstract— **Through this paper we consider how the representation, access and organization of the data can drastically affect the performance of Data Mining Techniques. The framework we propose utilizes vertical data representation which is an emerging data representation technique, combined with couple of compression schemes to facilitate efficient data mining, scaling over large datasets. The key aspect of using a compression scheme in SEED Miner lies in its vertical data representation (where a column-based data representation is considered in contrast to the conventional horizontal row-based representation) and we also provide the results of empirical simulations to validate our analysis of WAH compression applied on top of vertical data would provide the scalability and efficiency of the applications and algorithms embedded in SEED Miner.**

I. Introduction

The proliferation of databases has created a great demand for new, powerful tools for turning data into useful, task-oriented knowledge, thus giving birth to a separate field called Data Mining. Data Mining can be defined as semi-automatically finding useful patterns existing in large databases [2]. Data mining inherently deals with large databases usually of sizes starting from several gigabytes to several terabytes. Though there have been many standard algorithms to analyze data, with the escalating amounts, maintaining efficiency has become quite a challenge due to which many research have been carried out on finding special structures, data representation models which could preserve the efficiency of data mining solutions.

Since Data Mining Solutions are applied to a large variety of data sets, developing general solutions and scaling up has become another major problem [2, 3].

Most of the existing data mining solutions are at two extreme ends. Either they are optimized to a particular data set or application domain with scalability or generalized to work with multiple domains but without scalability and high performance [19].

Mining large datasets efficiently is strictly coupled with the efficient usage of system resources such as memory and perhaps processing power, which becomes a considerable challenge for algorithm developers. As a result, a significant researching effort has put forward on improving the efficiency of the mining algorithms. We, through SEED Miner, consider how the representation, access and organization of data can affect the performance. We, in our approach make use of vertical data representation in overcoming the above mentioned challenges and to provide a general Data Mining framework.

Distribution mechanisms (Data Distribution and load Distribution) [4, 5, 6, 7, 8 , 9] and using secondary storage as an intermediate memory [10, 2] are two methods widely used in achieving scalability. Though manipulation in main memory is the best way to obtain a significant performance gain, it limits the ability to scale. In our framework we utilize main memory for algorithm execution, while using the hard disk for storing the data set. The Vertical structure allows us to reduce the amount of disk accesses, thus improving the performance of the algorithms.

Since the algorithm optimization is done by the external developer who designs it, we focused more on providing an efficient method of representing, storing and manipulating a huge dataset in the main memory and provide the data into the algorithm for faster execution. In achieving this particular objective, our research was mainly focused on initially finding a data representation mechanism and in fact more preferably a representation mechanism where we could apply a compression scheme to cater for large amount of data in the faster main memory. In figuring out a suitable internal data representation, we came across an emerging data representation technique called **vertical data representation**. According to some of the earlier research, it has been found out that there are considerable advantages in vertical data representation over conventional horizontal layouts for data mining [2, 11].

Initially, in the case of data mining algorithms such as Association Rule Mining, vertical representation gives us a considerable advantage in computing the supports and confidences of itemsets in a simpler and faster way, as it involves carrying out bitwise operations like AND,OR and NOT among vertical bit streams[11]. And also in deducting association rules, it's about finding patterns in the vertical data bit streams which can also be carried out quite simply using bit stream intersections. But as in contrast, in horizontal counterpart, it requires complex hash-tree data structures and functions to perform the same task which is more complex and will be comparatively inefficient as well. (e.g. [1])

Data used for Data mining are inherently redundant. After converting to the vertical bit sliced format, the sequences of 0's and 1's provide means for compression, which is the aspect we concentrate when applying compression schemes to achieve a performance gain. Later throughout this paper, vertical bit level compression is discussed with experiments.

Within the scope of our research regarding SEED Miner, we do not provide the flavor of the parallelism but we have a very strong belief that parallelism applied on

top of SEED Miner will provide more flexibility and efficiency for the data mining applications and algorithms embedded in SEED Miner.

The rest of the paper is organized as follows. In Section two we present related work. Section three describes the design and implementation, of the framework. And finally we are showing scalability to through presenting some data mining results.

II. Related Work

Numerous works have been carried out in coming up with general purpose data mining techniques. **VIPER** [12] is one such effort, which is a scalable Association Rule Mining algorithm making no assumptions about the underlying data base.

In **VIPER** data is stored as compressed bitstreams called "snakes", which provides efficiency in snake generation, intersection, counting and storing [12]. When a horizontal dataset is provided during the execution of the algorithm, internally data is converted to the corresponding vertical data representation called *Vertical Tid List* (*VTL*). Golomb Encoding scheme [13] is used to increase the redundancy and to gain a better compression ratio, when storing generated frequent itemsets. During the mining process these compressed frequent item sets have to be decompressed over and over.

DiffSet is another vertical mining structure which is an optimized data representation for Association Rule mining. It is a representation that only keeps track of the difference in the transactionID's(tids) of a candidate pattern from its generating frequent patterns. It significantly cuts down the size of memory required to store intermediate results. The initial database stored in the diffset format, instead of the tidset, can also reduce the total database size [21].

PTree is another distributed compressed vertical data mining ready data structure. Formally, PTrees are treelike data structures that store relational data in column-wise, bit-compressed format by splitting each attribute into bits (i.e. representing each attribute value by its binary equivalent)[14]. Once the PTree representation is there, Data querying can be achieved through logical operations –such as AND, OR and NOT–referred to as the PTree algebra in the literature. [15][16]

Since we are using a vertical data representation in our framework, and the particular representation provides the opportunity for compression as one of its advantages, we implemented two compression schemes on top of the corresponding vertical data. In our quest for an appropriate compression scheme, we came across numerous encoding schemes like **Huffman Coding** [17], **Golomb Coding** [13], and **Byte-aligned Bitmap Code (BBC)** [22].

Huffman compression schema is another methodology that can be used to compress vertical bit slices. The importance of this schema is that we can achieve high compression ratio since the symbols are assigned according to a probability distribution. But the compression scheme is a variable length coding scheme where it is rather cumbersome to identify and separate each encoded bit word when it comes to decoding. There has been some algorithms designed using an unbalanced binary tree to traverse through it until a leaf node is found in order to retrieve a particular data item as well [17]. Although there are research carried out on fast searching of data items using encoded data stream, no method has been proposed to compute basic operations AND, OR and NOT , which is frequently using operations in data mining, without decompressing the data stream.

William A. Maniatty and Mohammed J. Zaki specify the requirements of a Data mining framework in [3]. Though Data Mining Phase ,only a single step in the total KDD Process, since 80% of times is spent on data integration , preprocessing and post-processing steps they suggest that a Data Mining framework should also support all the steps in KDD such as preprocessing (discretizing , subset selection), post processing (rule grouping, pruning ,summarization) caching , efficient retrieval and meta level mining. Under the algorithm evaluation capability of such a framework, they mention that reducing the number of database scans should be another requirement of such a framework.

The work by Perrizo *et al.* in [19] describes the structure and composition of a distributed vertical data mining framework. In their framework Ptree, a block-wise lossless compressed structure is used for vertical data representation.

III. Design and Implementation

Keeping with the basic requirement of the supporting a number of different data sources and a number of different algorithms, the first step was to decouple data mining phase from data gathering phase. This was done by introducing a common storage and creating separate modules for data extraction and data mining. Logically, the framework consists of three main layers as illustrated in Fig.1;

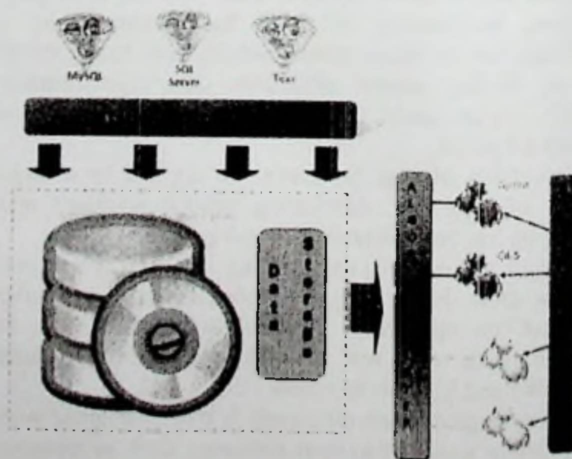

Fig. 1. The high level architecture of the framework.

Data Feeding Layer:

This layer focuses on gathering data from different sources and converting them to the format supported by the internal storage. Data extraction module implements this layer.

Data Storage:

Manages the vertical data structure and provides means to access elements in the dataset. Data Extraction Module and Compression Module implement this layer.

Algorithm Layer:
Provides basic component algorithms for manipulating the vertical data source. These component algorithms can be used by for developing actual data mining algorithms (such as Apriori.) The Algorithm Module implements this layer. Preprocessing Module is a module built on top of this layer.

The raw data extracted from external data sources are stored column wise. The raw data is then analyzed and encoded to their corresponding vertical representation. While doing so, we keep track of some important Meta information depending on the type of data retrieved from the external data source. According to the structure of our framework, the highest level entity will keep track of the data sources available and encoded. Each data source will keep track of the attribute wise information such as cardinality of the dataset, occurrence of null values, labels for categorical attributes. Apart from basic Meta information it maintains the list attributes and in those Attributes framework keeps track of the details of the actual values that are retrieved. The Meta information stored in an Attribute entity depends on the type of the data each attribute holds (which will be described further in later). In the same time, an attribute contains a list of vertical bitstreams which are the vertical representation of the horizontal raw data. A bit stream entity which is the basic entity will keep track of the actual vertical bit stream and also some important Meta information like bitstreamID, bitStreamAllocationName, etc... The structure is illustrated in the Fig.2.

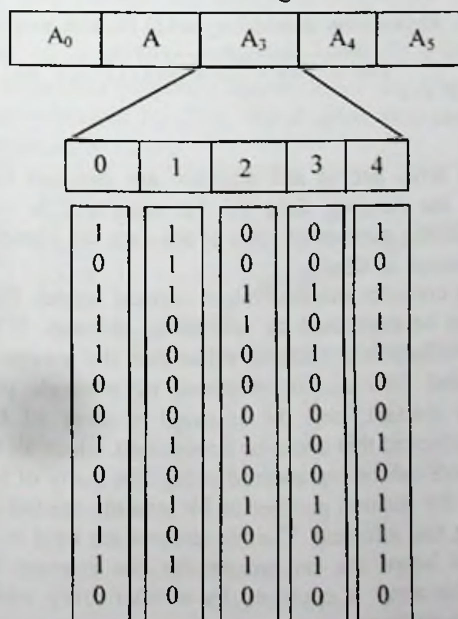

Fig. 2. A high level representation of data organization within the framework

In a design point of view, our framework execution is categorized into four predominant components.

A. Data Extraction Module

Data Extraction module is acting as the interface to external raw datasets and responsible for taking in whatever the selected amount of data supplied by the external source. The external source can be a conventional SQL database, a text file, a CSV file or different type of data file where user defined field and line separators can be specified. New types of data sources can be attached to the framework by extending this module.

Once the data is extracted into the system, the vertical data representation of those horizontal raw data will be carried out by the module. When it comes to integer type of data, direct bit representation of them will be taken using the dynamic bitset provided by boost [20] and then partition the data into vertical representation. In case of floating point type of data, a particular precision will be decided internally by inspecting the statistics of the raw data and then convert the floating point values into integer type by multiplying it by the precision and then they will be treated same as the above mentioned procedure and here an additional metadata entry will be there to keep track of the precision value. Partitioning categorical data is being done by assigning integer index value to each categorical data, while the index is prepared by scanning through the raw data once and finding out the unique data items and assigning those unique items to a vector and then sort it and the corresponding index of the vector will be considered as the index value assigned to each categorical data while partitioning. Once an integer set is obtained where indexes are replaced for categorical data values, it then will be treated as a conventional integer type of data and here an additional metadata entry will be there to keep track of the unique data items. In values which are in the format of Date, we convert the date value to an integer in the form of YYYYMMDD and treat it as an integer type value.

We also in our framework provide the facility to save whatever the extracted and partitioned data into the secondary storage as an XML file, and load the data from that external XML file later, which may be useful at a later time when the same dataset is to mined for a different purpose. This facility is provided due to the fact that the extracting external data from conventional database or a text or CSV file and then vertically partitioning them going through each value will be a time consuming activity and hence in order to save the time and space, it will be an unnecessary effort to carry out the same thing twice.

B. Data Preprocessing Module

Data Preprocessing is done prior and after the data mining is carried out, once we have the wrapped data source which will be created during data extraction or data loading via the saved data in XML. Data processing is available as three flavors in our framework; NULL Elimination, converting continuous data into discrete data and splitting numerical data into ranges as a categorical data attributes. There can be different types of other preprocessing filters available in other frameworks and applications, which can also be implemented in the same way in our framework as well.

C. *Data Compression Module*

If the complete data set can be loaded in to the memory at once, mining can be performed much efficiently since the disk accesses can be reduced to a greater extent. Compression module enables compression of original data and generation of compressed bitstreams of intermediate results. Even though compression makes it possible to store large amounts of data, if it is required to decompress during execution, then it'll incur a considerable overhead during the execution phase. Since all the high level operations can be reduced to a set of basic operations such as bit stream intersection, (AND) bit stream union (OR), it would be sufficient to find an encoding scheme, capable of performing these operations while compressed. We have included such a scheme, described in [18]. Theoretically this scheme compresses only a subset of the all possible bitstreams. Hence the framework will determine if space can be saved by compressing the data source.

D. *Algorithm Module*

This module provides efficient implementations for a set of recurrently used algorithms such as attribute sum, range count, etc... In data mining, a lot of time is spent on taking count of certain patterns occurring in a dataset. Though implementing such operations are straightforward in a horizontal model, in vertical model it may appear bit awkward, which may lead the developers to create inefficient algorithms. Due to this, we have implemented the heavily used algorithms as described in [19] . When performing a complexity analysis it can be seen that the vertical implementation has the same complexity as its horizontal counterpart, but executing in an efficient manner.

IV. Experimental Results

This section discusses how common operations such as attribute sum, pattern count and range count are performed using vertical structures and how compression contributes in reducing corresponding execution times. First we present the detailed analysis of some component algorithms along with their complexities. By analyzing algorithm complexities we argue that by using a vertical model for data representation the algorithm complexity doesn't change. Then we move into a probabilistic analysis to reason out why vertical model performs better than the horizontal model.

A. *Experiment Description*

Most of the component algorithms were designed for manipulating numerical attributes. To test these algorithms, a dataset having cardinality of 10000000 was used. When carrying out the tests for numerical algorithms, accuracy was tested by performing the same operation in the horizontal model and comparing the answers. The STL implementations were used when performing horizontal operations.

For testing the data mining algorithms, standard datasets provided in UCI repository were used [23]. For performing the scalability test Pokerhand dataset [24] was used. Though originally this contained 1 million instances, this was randomly multiplied to generate a 5 million dataset. For observing the relationship with time vs. confidence for Apriori, soybean dataset [25] was used (This was also multiplied to create a 5-million dataset). For testing accuracy of Naïve Bayes, intrusion dataset was used [26].

When testing data mining algorithms, they were first tested for their accuracy. When implementing the data mining algorithms the code provided by Weka [27] was used. When testing the accuracy same dataset was mined by both Weka and SEEDMiner and the results were compared.

B. *Experimental Setup*

All the tests were performed on an Intel Core Duo, 1.83GHz speed, with 1 GB of main memory. When measuring time all the algorithms were executed 5 times and the average was taken.

C. *Experiment Results and analysis*

Before going deeply into the algorithm analysis we first describe a horizontal model to execute the same algorithm. This will provide a basis for a detailed analysis of algorithm execution times and the related complexities comparatively. Let's assume that data is first arranged in to a relational model, where a dataset can be expressed in terms of attributes and Tuples. Many models could be proposed to represent this data using an Object oriented approach, but for simplicity let's consider a two dimensional matrix, capable of holding any data type. In this representation each attribute could be accessed by providing the row id and the column id.

Let's consider an algorithm for taking the sum of a particular numeric attribute.

```
Algorithm: Attribute Sum(data_set[][],att_no)
        for i =0 until data_set.length
                sum ← sum + data_set[i][att_no];
        end for
return sum;
```

Since the array access and addition are constant time operations, the running time of the algorithm is only dependant on the number of rows of the data set. Hence it can be expressed as $O(n)$.

Now let's consider the equivalent vertical model. Each attribute can be expressed by several bit streams. If the particular attribute is a numeric value then the maximum value for that field can be obtained by a single pass through the dataset, and the relevant number of bits needed to represent that could be determined. Since all the other numbers can be represented using this many of bits this will be the optimal number of bit streams needed for representing that attribute. The bit streams are held in an array which keeps the bit stream for the relevant bit position. This array is enclosed by another array which represents an attribute.

The bit stream corresponding to the first bit position of the third attribute could be obtained by **data_set[3][0]**.

D. *Comparison of attribute Sum*

By taking the count of each vertical bit stream, the sum of a numerical attribute can be easily calculated.

```
Attribute Sum(data_set[][],att_no)
// Obtaining the number of bits used for representing the attribute
max_bits ← data_set[att_no].cardinality
       for i =0 until max_bits -1
              sum   ←   sum   +   2^i   *
data_set[att_no][i].count();
       end for
return sum;
```

Here count() operation returns the number of true bits in the bit stream. When analyzing this algorithm it is evident that the outer loop is independent of the number of rows. It'll run *log₂(max_value[att_no])* number of times. When considering a 32 bit integer the maximum times the outer loop will run will be 32. Since the count operation is not a constant time operation we should take its complexity as well.

The simplest implementation of the count operation would be as follows.

```
Algorithm:Count()
for i=0 until num_of_bits
       if(bitstream[i] =1)
        sum++;
       end if;
end for;
return sum;
```

Usually more efficient implementations are used for taking the count of a bitmap, such as separating the bitmap to a number of bytes and using each byte as an index to a lookup table, which contains the count for each byte [28]. Though the algorithm complexity remains the same, execution of the latter implementation is much faster. Since we are comparing complexities let's refer to the algorithm proposed above. Since the complexity of count proves to be *O(n)*, the attribute sum also yields a complexity of *O(n)* .

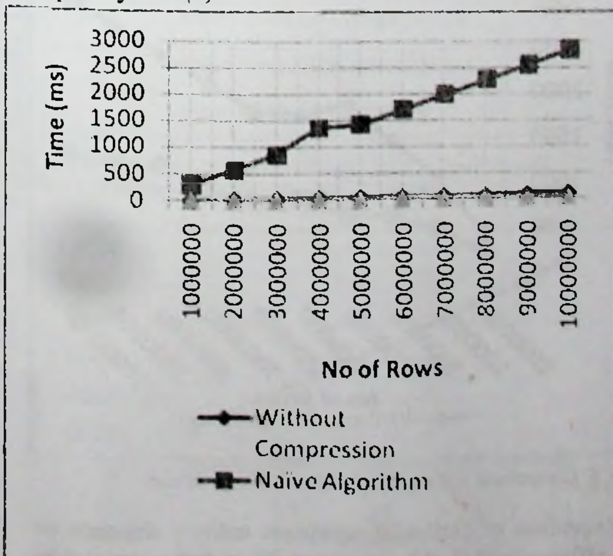

Fig. 3. Attribute Sum over a large Dataset

Since both the algorithms fall into the same complexity class, one may conclude that no performance gain can be obtained by using a vertical model. However, as illustrated by Fig.3. the test results show a significant difference in the two executions.

Here Naïve algorithm refers to the algorithm developed on the horizontal model. Even though it scales with the data set, still it takes a considerable time when compared to the vertical algorithm. When observing the graph we can see that even the times recorded for the horizontal algorithm is larger than they are not asymptotically larger than the horizontal model.

Let's consider the effect of compression over algorithm executions.

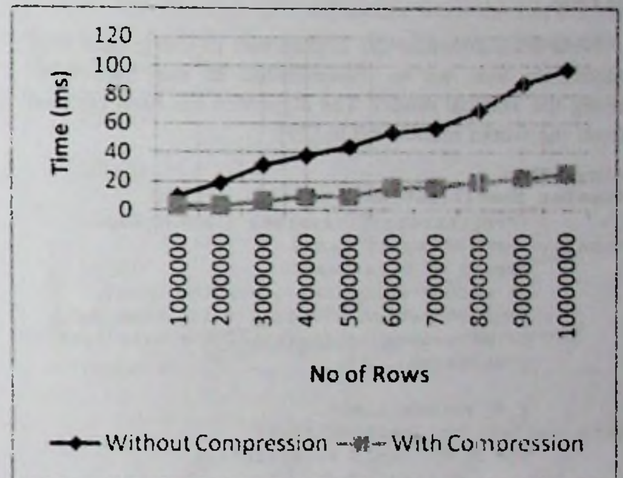

Figure 4: Effect of compression on the algorithm Attribute Sum

By observing the Fig. 4 it is evident that compression works on reducing the overall execution time of attribute sum. Even though the numbers used are random, when they are aligned closely certain bit positions may create long recurring segments of pure ones or zeros making them easily compressible. When performing basic operations as count, since the bit streams are compressed the corresponding inner loops will execute less times than n/8. This feature will help in reducing the overall execution time. On the vertical model algorithm exhibit a linear scale up.

E. *Comparison of Range Querying*

Range queries are heavily used in classifiers such as C4.5 and Naïve Bayes for calculating probability and creating data partitions according the information gains. For most of the occurrences, obtaining the number of instances satisfying a particular query may be equally important as obtaining the places they are occurring. Since vertical model is capable of giving an existence bitmap of the results satisfying a particular condition, when creating the equivalent horizontal model we had to consider about a mechanism to track down the occurrences of those result. Therefore in a horizontal model, the algorithm for taking range queries is as follows;

```
Algorithm:
Greater_Than(limit,data_set[][],att_no)
            for i ←0 until data_set.length-1
                    if data_set[i][att_no]>
            limit
                    existence_map[i] ← 1
                    end if
            end for
        return existence_map;
```

Since the array access, assignment and the greater than comparison (>) can be treated as constant time operations, the running time of the algorithm is only dependant on the number of rows of the data set. Hence it can be expressed as *O(n)*.

Now let's consider the counterpart of the greater than algorithm that we've implemented in our framework using the vertical model. The algorithm has been adopted from the works mentioned in [29].

```
Algorithm:
Greater_Than(limit,data_set[][],att_no)
        // bitstream(0) assigns a bitstream
totally consisting of zeros
        result ← bitstream(0)
        /* encode functions gives the binary
        representation of limit using that many
        of bits used to represent the attribute
        considered.  */

        v ← encode(limit,
data_set[att_no].cardinality);
        for i ←0 until v.length - 1
                if(vi  = 1)
                        i← i+1;
                end if
        end for
        if i < v.length
                result ← data_set[att_no][i];
        end if;
        for i←i+1 until v.length - 1
                if vi = 0
                        result ← result |
        data_set[att_no][i];
                else
                        result ← result &
        data_set[att_no][i];
                end if
        end for;
        return result;
```

When analyzing this algorithm, it is evident that the running time is dominated by the running time of bitmap intersection (AND) and bitmap union (OR) operations in the last for loop where the resultant bitmap is generated. Since both of those operations takes running time of *O(n)*we can consider that the above algorithm for calculating the Greater_Than procedure has a running time of *O(n)*.

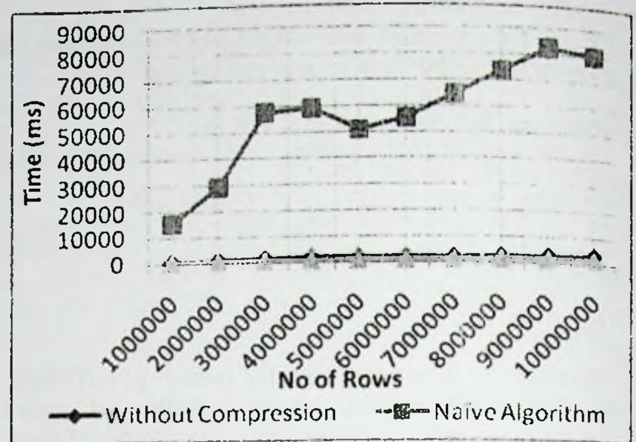

Fig. 5. Range Querying over large dataset using vertical, vertical with compression and horizontal models

Although the running times of both models are the same, we can observe from Fig. 5 that a significant performance gain can be obtained, even without applying compression, using the vertical model.

Let's consider how the compression affects the performance of the algorithm. As the graph in figure 6 clearly indicates, we can achieve a considerable amount of performance gain when applying compression on vertically modeled dataset. Even though the numbers used are random, when they are aligned closely certain bit positions may create long recurring segments of pure ones or zeros making them easily compressible. When performing basic bitwise operations as AND and OR, since the bit streams are compressed the corresponding bit slices that are used to carry out Intersection and Union operation are fewer than that of uncompressed version. This feature will help in reducing the overall execution time.

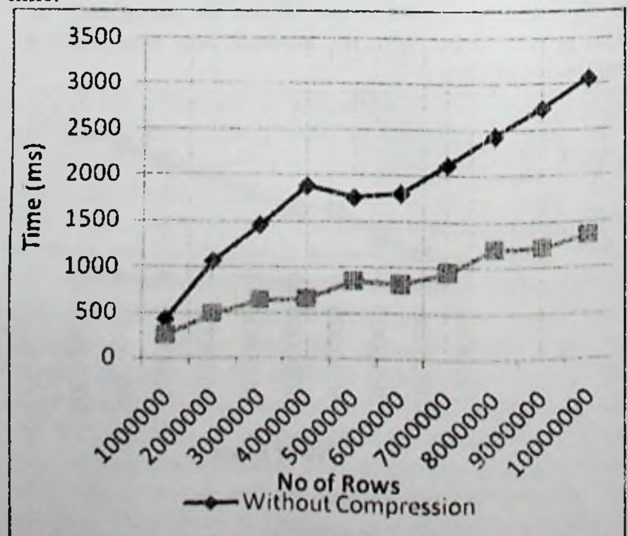

Fig. 6. Comparison of Range querying over a large dataset.

Execution of particular operation entirely depends on the efficiency of the bitmap count. Though the complexity of this operation is *O(n)*, since hamming weights are used for taking the count [28], the actual number of

instructions executed are a fraction of n (which is the cardinality of the dataset), which gives a better performance than the horizontal model.

F. *Experimental results for Data Mining Algorithms*

By performing tests for both vertical and horizontal models proved that vertical model is good for certain numerical algorithms. But in order to prove that the framework can be used to design efficient data mining algorithms, we performed another series of tests by implementing some widely used algorithms both in vertical and horizontal models.

Our first candidate was the Naïve Bayes Classifier. This is a classification algorithm using the naive assumption that events are independent of each other [30]. When implementing data mining algorithms we simply adopted the implementations provided by Weka [27]. Weka followed a horizontal model. Since we have only changed the pattern counting methods when implementing the algorithm on the vertical model, we could isolate the effect of vertical computations. Since Weka is a Java application before performing the tests, we had to implement the same algorithm in C++.

Since the dataset can increase in row wise and also column wise, we had to test the algorithm for both of those cases. Figure 7 shows the scalability test when number of rows is increased, and Figure 8 shows the scalability test when increasing the number of attributes.

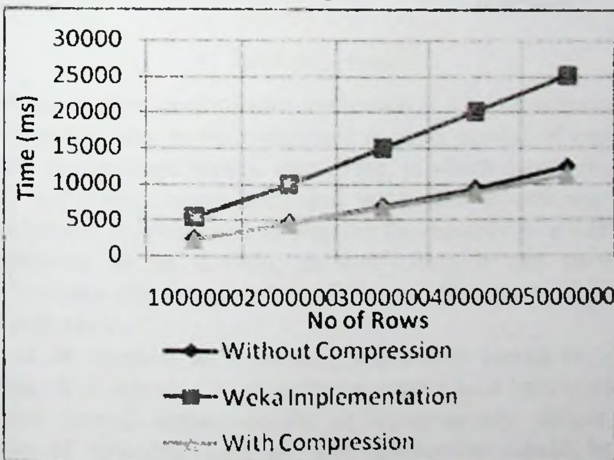

Fig. 7. Performance Comparison of Naive Bayes Classifier with WEKA's implementation (increasing # of columns)

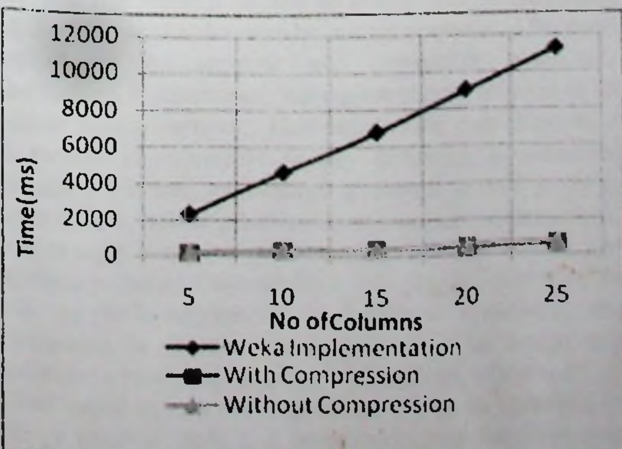

Fig. 8. Performance Comparison of Naive Bayes Classifier with WEIKA's implementation (increasing # of columns)

By observing the graphs we can see that for both cases, vertical model simply outperforms the horizontal model. In the horizontal model, the running time directly depends on the number of attributes. Due to that we can see that when the numbers of attributes are increased, the corresponding scale up in the horizontal model is worse than that of the previous case (when number of rows is increased). Though the horizontal model does have a linear scale up, it is much worse than the vertical model.

Since it is a known fact that vertical model performs well for Apriori[11, 2] , we tested the vertical model with another approach. In [10], another flavor of Apriori algorithm is presented which uses a tree structure called Trie. Though the data is accessed horizontally, since the internal data structure is independent from the horizontal model , we thought of selecting this for our experiments. The results are illustrated in Fig.9;

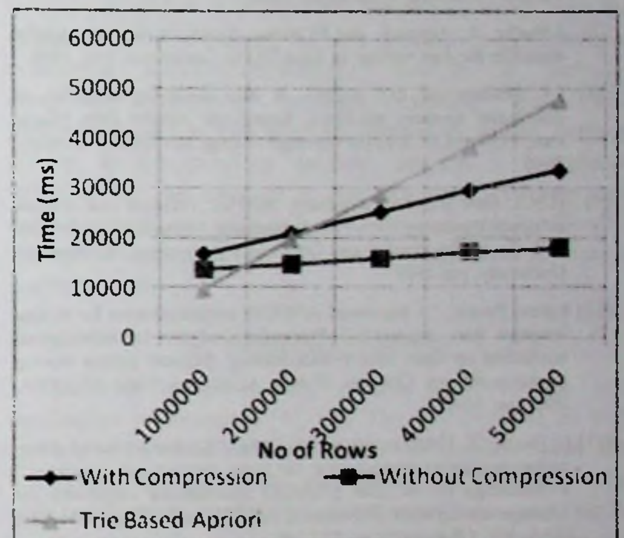

Fig. 9. Total Execution Time in Apriori

Here when measuring time we had to consider the time taken for compression as well. Since a significant time is spent in compressing data, the compressed version tends to be slower than the uncompressed one. When observing the graph we can see that the Trie based approach [10], is good for small datasets. But when the cardinality is increasing, the vertical implementations simply outperform the Trie based approach.

V. Conclusion

In this paper we presented requirements, design, and implementation of a scalable data mining framework. As the first step in making computations efficient, we have chosen a vertical representation model which is proven to be efficient for computationally intensive algorithms. We have implemented the framework in a manner that it can be used in the complete cycle of KDD process. By analyzing algorithm complexities we have proven that the vertical algorithms have the same complexity as the horizontal algorithms. Further we have shown the efficiency of the framework by using experimental results. Therefore, this framework can be introduced as a useful piece of work for data mining since it addresses the two main problems scalability and efficiency.

References

[1] Agrawel R. and Srikant R., September 1994, Fast Algorithms for mining association rules. *In Proc. of 20th Intl Conf. Very Large Databases (VLDB)*, Santiago, Chile.

[2] Jiawei Han, Micheline Kamber , "Introduction", in *Data Mining: Concepts and Techniques,*2nd ed., San Francisco: Morgan Kaufmann Publishers, 2006, pp. 1-38.

[3] William A. Maniatty and Mohammed J. Zaki, "Systems Support for Scalable Data Mining," *SIGKDD Explorations*, vol. 2, Dec. 2000, pp. 56 - 65.

[4] R.Agrawal and J.Shafer. Parallel mining of association rules. IEEE Trans. on Knowlledge and Data Engg.,8(6):962-969, Dec.1996

[5] D. Cheung, J.Han, V.Ng, A.Fu and Y.Fu. A fast distributed algorithm for mining association rules. In 4th Intl. Conf. Parallel and Distributed Info. Systems, Dec. 1996.

[6] M. Joshi, G. Karypis, and V. Kumar. ScalparC: A scalable and Parallel classfication algorithm for mining large datasets. In Intl. Parallel Processing Symposium, 1998.

[7] J.Shafer, R. Agrawal, and M.Mehta. Sprint: A scalable parallel classifier for data mining. In 22nd VLDB Conference, Mar. 1996.

[8] I.S. Dhillon and D.S Modha. A data clustering algorithm on distributed memory machines. large-Scale parallel Data Minng, vlm, 1759 of LNCS/LNAI. Springer-Verlag, Heidelberg, Germany, 2000.

[9] H.N.S. Goil and A. Choudhary. MAFIA: Efficient and scalable subspace clustering for very large datasets, Technical Report 9906-010, Center for Parallel and Distributed Computing, Northwestern University, jun 1999

[10] Ferenc Bodon, "A trie-based APRIORI implementation for mining frequent item sequences," Proceedings of the 1st international workshop on open source data mining: frequent pattern mining implementations, Chicago, Illinois: ACM New York, NY, USA, 2005, pp. 56-65.

[11] C. Binnig, S. Hildenbrand, and F. Färber, "Dictionary-based order-preserving string compression for main memory column stores," *Proceedings of the 35th SIGMOD international conference on Management of data*, Providence, Rhode Island, USA: ACM New York, NY, USA, 2009, pp. 283-296.

[12] Shenoy P., Bhalotia G., Harista J.R., Bawa M., Sudarshan S. and Shah D. Turbo-charging Vertical Mining of Large Databases.

[13] Golomb S.W., July 1966, Run Length Encoding, *IEEE Trans on information Theory*, vol.12: pp. 399- 401

[14] Masum Serazi, Amal Perera, Qiang Ding, VasilyMalakhov, Imad Rahal, Fei Pan, Dongmei Ren, Weihua Wu, and William Perrizo. DataMIME™. SIGMOD 2004, Paris, France.

[15] Ding. Q, Khan, Roy, and Perrizo W., 2002, *The P-tree algebra.* Proceedings of the ACM SAC, Symposium on Applied Computing (Madrid, Spain)

[16] Han J., Pei J. and Yin Y., 2000, *Mining Frequent Patterns without Candidate Generation*. Proceedings of the ACM SIGMOD, International Conference on Management of Data (Dallas, Texas).

[17] Hashemian, October 1995, *Memory Efficient and High-speed Search Huffman Coding*, IEEE Transactions On Communications, Vol. 43, No. 10: pp 2576 -2581

[18] Chan. C. Y. and Ioannidis. Y. E.. "An efficient bitmap encoding scheme for selection queries".In Delis. A., Faloutsos. C., and Ghandeharizadeh. S., editors, *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadelphia ,Pennsylvania, USA*. ACM Press.

[19] Serazi M., Perera A., Abidin T., Hamer G., and Perrizo W., "An API for Transparent Distributed Vertical Data Mining."

[20] dynamic_bitset<Block, Allocator>. Boost C++ Libraries, Available at: http://www.boost.org/doc/libs/1_43_0/libs/dynamic_bitset/dynamic_bitset.html. **[Accessed July 30, 2010].**

[21] M. J. Zaki and K. Gouda, "Fast vertical mining using diffsets" *Technical Report 01-1*, Rensselaer Polytechnic Institute, USA, 2001.

[22] G. Antoshenkov, *Byte-aligned bitmap compression*, Technical report, Oracle Corp., 1994. U.S. Patent number 5,363,098.

[23] A. Frank and A. Asuncion, "UCI Machine Learning Repository," *UCI Machine Learning Repository*, 2020. Available at: http://archive.ics.uci.edu/ml/ [Accessed July 15, 2010]

[24] Poker Hand Data Set. UCI Machine Learning Repository. Available at: http://archive.ics.uci.edu/ml/datasets/Poker+Hand **[Accessed July 15, 2010].**

[25] Soybean (Large) Data Set , UCI Machine Learning Repository. Available at: http://archive.ics.uci.edu/ml/datasets/Soybean+(Large) **[Accessed July 15, 2010].**

[26] KDD Cup 1999 Data. The UCI KDD Archive. Available at: http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html **[Accessed July 15, 2010].**

[27] Weka Repository https://svn.scms.waikato.ac.nz/svn/weka/ **[Accessed July 15, 2010].**

[28] Peter Wegner, "A technique for counting ones in a binary computer," Communications of the ACM, vol. 3, May 1960, p. 322.

[29] Chee-Yong Chan and Yannis E. Ioannidis, "Bitmap index design and evaluation," *ACM SIGMOD Record*, vol. 27, Jun. 1998, pp. 355-366.

[30] H. Zhang, "The optimality of naive Bayes", *Proceedings of the 17th International FLAIRS conference (FLAIRS2004)*, AAAI Press(2004).