ENHANCED GLADE AND ITS IMPACT ON COMPUTATIONAL DATA ANALYTICS

By
DALEY ALEX

A THESIS PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
UNIVERSITY OF FLORIDA
2012
To my mother who is my greatest critique and confidante
To my father for his toils in making my life a possibility
To my sisters for their ceaseless love and affection
To my closest friends, without whom I will never be myself
Last but not the least, to all the people I have ever had a chance to meet
ACKNOWLEDGMENTS

Foremost, I would like to express my gratitude to my advisor, Dr. Alin Dobra who has been a mentor to me throughout my Masters studies and without whose patience and guidance, I would have never learned the technicalities of conducting research.

I would like to thank my co-advisor, Dr. Sanjay Ranka for his invaluable insights and advices.

My sincere thanks also goes to Dr. Daisy Zhe Wang for taking out valuable time to serve on my committee.

In addition, I thank my colleague Sabari Ajay Kumar for his support and assistance throughout the course of my research.

Last but not the least, I would like to express my gratitude to my parents Alex George and Soosan Alex, my sisters Anju and Amritha Alex and also to the closest of my friends.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Acknowledgements</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of Tables</td>
<td>7</td>
</tr>
<tr>
<td>List of Figures</td>
<td>8</td>
</tr>
<tr>
<td>Abstract</td>
<td>9</td>
</tr>
<tr>
<td>Chapter 1 INTRODUCTION</td>
<td>10</td>
</tr>
<tr>
<td>Chapter 2 ORIGINAL SYSTEM</td>
<td>13</td>
</tr>
<tr>
<td>Chapter 3 SYSTEM ENHANCEMENTS</td>
<td>16</td>
</tr>
<tr>
<td>3.1 Iteration</td>
<td>16</td>
</tr>
<tr>
<td>3.2 State Passing</td>
<td>18</td>
</tr>
<tr>
<td>3.3 State Passing and Iteration</td>
<td>19</td>
</tr>
<tr>
<td>Chapter 4 ALGORITHMS</td>
<td>21</td>
</tr>
<tr>
<td>4.1 Item-Based Collaborative Filtering</td>
<td>21</td>
</tr>
<tr>
<td>4.2 K-Means Clustering</td>
<td>24</td>
</tr>
<tr>
<td>4.3 Decision Tree Classifier</td>
<td>25</td>
</tr>
<tr>
<td>4.4 Apriori</td>
<td>28</td>
</tr>
<tr>
<td>4.5 FP-Growth</td>
<td>32</td>
</tr>
<tr>
<td>4.5.1 FP-Tree Merge</td>
<td>33</td>
</tr>
<tr>
<td>4.5.2 Proof of Correctness</td>
<td>34</td>
</tr>
<tr>
<td>4.5.3 Time Complexity</td>
<td>36</td>
</tr>
<tr>
<td>Chapter 5 EMPIRICAL EVALUATION</td>
<td>37</td>
</tr>
<tr>
<td>5.1 Item-Based Collaborative Filtering</td>
<td>38</td>
</tr>
<tr>
<td>5.1.1 Book Crossing Dataset</td>
<td>38</td>
</tr>
<tr>
<td>5.1.2 Movielens</td>
<td>38</td>
</tr>
<tr>
<td>5.2 K-Means Clustering</td>
<td>39</td>
</tr>
<tr>
<td>5.3 Decision Tree Classifier</td>
<td>42</td>
</tr>
<tr>
<td>5.4 Apriori</td>
<td>42</td>
</tr>
<tr>
<td>5.5 FP-Growth</td>
<td>43</td>
</tr>
<tr>
<td>Chapter 6 DISTRIBUTED STATE</td>
<td>45</td>
</tr>
<tr>
<td>6.1 FP-Growth Distributed</td>
<td>46</td>
</tr>
<tr>
<td>6.2 Empirical Evaluation</td>
<td>46</td>
</tr>
<tr>
<td>Chapter 7 RELATED WORK IN MAPREDUCE</td>
<td>48</td>
</tr>
</tbody>
</table>
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5-1</td>
<td>Hadoop Configurations</td>
<td>37</td>
</tr>
<tr>
<td>5-2</td>
<td>Mahout Configurations</td>
<td>38</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>-------------</td>
<td>------</td>
</tr>
<tr>
<td>3-1</td>
<td>Iterative Mechanism in GLADE</td>
<td>17</td>
</tr>
<tr>
<td>4-1</td>
<td>FPTree Merge</td>
<td>35</td>
</tr>
<tr>
<td>5-1</td>
<td>GLADE Item Based Collaborative Filtering</td>
<td>39</td>
</tr>
<tr>
<td>5-2</td>
<td>Item Based Collaborative Filtering : GLADE vs Mahout [log scale]</td>
<td>40</td>
</tr>
<tr>
<td>5-3</td>
<td>Per Iteration Speedup, KMeans : GLADE vs Mahout [log scale]</td>
<td>41</td>
</tr>
<tr>
<td>5-4</td>
<td>Scalability of KMeans(GLADE) Algorithm</td>
<td>41</td>
</tr>
<tr>
<td>5-5</td>
<td>GLADE DecisionTree</td>
<td>42</td>
</tr>
<tr>
<td>5-6</td>
<td>GLADE Apriori vs GLADE FPGrowth</td>
<td>43</td>
</tr>
<tr>
<td>5-7</td>
<td>FPGrowth GLADE vs FPGrowth Mahout [In log scale]</td>
<td>44</td>
</tr>
<tr>
<td>6-1</td>
<td>The Distributed State</td>
<td>46</td>
</tr>
<tr>
<td>6-2</td>
<td>The Replicated State</td>
<td>47</td>
</tr>
<tr>
<td>6-3</td>
<td>Replicated vs Distributed State</td>
<td>47</td>
</tr>
</tbody>
</table>
Abstract of Thesis Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Master of Science

ENHANCED GLADE AND ITS IMPACT ON COMPUTATIONAL DATA ANALYTICS

By

Daley Alex

August 2012

Chair: Alin Dobra
Cochair: Sanjay Ranka
Major: Computer Engineering

The management and analysis of large amounts of constantly increasing data is required to facilitate better knowledge and understanding. Such analysis extracts less apparent information and forms fodder for enhancement of products and services by business institutions and also in a variety of other domains.

Two techniques that are available for managing large amounts of data and simultaneously performing complex computations are MapReduce and User Defined Aggregates (UDA). However, the flexibility with which you can express computations in either models require enhancements.

This thesis focuses on enhancing a computational model called GLADE which is already efficient at managing computations on large data but has a shortcoming in its flexibility to express computations. This work transforms GLADE into a comprehensive format for large data management and for carrying out complex analytic computations with improved flexibility and performance when compared to other existing computational models. The first part of the thesis deals with proposing the required enhancements in detail and in the second part, this flexibility is exhibited by encoding some of the classical data mining algorithms in this format. Also, comparisons are drawn wherever possible with existing implementations in Mahout, a MapReduce based machine learning library.
As more and more data is becoming available, the demand for large data analytics is constantly increasing. For performing complex analytics on large data, the mechanisms provided in the traditional databases are mostly unsuitable. Therefore, one of two approaches are adopted for performing such analytics, either a MapReduce like framework that does not fit into the notion of a traditional data management system or the use of User Defined Aggregates (UDAs) used in conjunction with an existing database system. UDAs have existed in traditional database systems for a long time, they have been used to perform simple analytical tasks on relational data. However, the advent of MapReduce [1] introduced a new format in encoding computations and became popular with computations involving large data. MapReduce introduced a mechanism to perform complex analytics on large amounts of data and moreover has an inherent ability for scaling to large clusters. However, it also has disadvantages in the form of a rigid structure that demands the encoding of data in the form of key-value pairs, a hash based load distribution that depended on the key values and no internal iterative mechanism.

The database community adopted either the MapReduce framework that focused mainly on unstructured data or chose to enhance the UDAs so that they could compete with the growing demand to carry out complex computations on large amounts of data. The latter has an advantage that it could very effectively leverage already advanced database technologies. Rusu et al. [2] introduced a new form of User Defined Aggregates called Generalized Linear Aggregates or GLAs and also its further implementation as a framework called GLADE on top of a state of the art database system called Datapath [3]. GLAs along with Datapath provided many new possibilities for computational data analytics as explained in Chapter 2.
However, UDAs including the original GLAs have some shortcomings that affected their flexibility to encode complex computations,

- data could only be moved in the form of relations
- like MapReduce, they also did not have an internal mechanism to carry out iterative computations

Our focus on iterative mechanisms lead to finding a more natural method of storing intermediate data, in the form of Constant States, Chapter 3. We were also able to effectively use the Constant States further for non-relational communication of data, a mechanism known as State Passing, Chapter 3. State Passing along with Iteration opened up new avenues in developing algorithms for computations, especially with regard to parallelization prospects. Our intention was to take the GLADE framework to the next level, wherein it can encode any given analytics with improved flexibility and also produce the same performance it initially did. Our starting point was to develop and adapt some of the classical data mining algorithms. Mahout, [4] seemed a good starting point as it is one of the most popular and comprehensive library of machine learning algorithms. We had the notion of enforcing changes to the system if a real world task demanded such changes rather than building functionalities that may never be used.

This thesis enhances the work done in [2].

- We further build on the original system and introduces two further enhancements to the GLADE framework and also the Datapath system, Iterative GLAs and State Passing, which allows us a flexible mode for expressing computations.
- We focus on developing GLADE based versions of Item Based Collaborative Filtering, frequent itemset mining both FP-Growth and Apriori, Decision Tree Classification and KMeans Clustering
- Wherever possible we compare and contrast our implementations with the corresponding implementations in Mahout.

**Thesis Organization**: The rest of the thesis is organized as follows: Chapter 2 explains the original system of GLADE and Datapath in detail, Chapter 3 details the enhancements that we propose and our motivation to make these changes, Chapter 4
focuses on the data mining algorithms mentioned earlier with detailed explanations of our proposed algorithms, Chapter 5 we further publish our speedups and also wherever possible a comparison is drawn with Mahout, Chapter 6 introduces a new form of state aggregation in GLAs for large intermediate states, Chapter 7 explores some of the iterative mechanisms proposed for MapReduce and we compare them to our mechanism and Chapter 8 concludes the thesis.
Generalized Linear Aggregates is at the core of the GLADE framework. It is an Abstract Data Type that has a state and some functions to modify that state. The state can be literally anything ranging from a simple integer to a complex hash map. The functions used to examine and modify the state as introduced by Rusu et al. [2] are as follows:

- **Initialize()**: sets up the state as required for the computation
- **AddItem(Tuple t)**: adds t to the current state
- **AddState(GLA &other)**: combines the state present in other to the state of the current GLA. AddState() is associative in nature.
- **Finalize()**: extracts the required information from the state grown by the GLA

Let's now look at a simple example to calculate the average of a large set of numbers. The information that you need to essentially maintain as the state here is the sum of all numbers and their count.

*State*: Sum, Count

*Functions*:

- **Initialize()**: initializes Sum and Count to 0.
- **AddItem(Tuple t)**: t is one of the numbers whose average is to be found. AddItem() adds the number to the Sum and increments the count.
  
  \[ \text{Sum} += t; \]

  \[ \text{++Count}; \]

- **AddState(GLA & other)**: It combines the Sum and Count of the two GLA's, respectively, thereby aggregating the two states. The AddState() without any additional cost, is associative as well.
  
  \[ \text{Sum} += \text{other.Sum}; \]

  \[ \text{Count} += \text{other.Count}; \]

- **Finalize()**: Here, Finalize infers the average from the maintained statistics, which can be done by a single division.
  
  \[ \text{Average} = \text{Sum} / \text{Count}; \]

The inherent properties of GLAs provides some useful features at the core:
there is no restriction as to the type of data taken as input
there is no requirement for a key value pair as enforced by MapReduce
there are no constraints as to how the GLAs can be aggregated as the AddState() is associative

The associative nature of the GLAs also leads to a natural ability to load balance which can be ensured by dividing the input dataset into smaller parts and building GLAs from each of these parts. The GLAs grown separately can be combined in any sequence because of the associative property of the AddState. MapReduce on the other hand, uses a hash distribution to load balance based on the values of the keys in its key-value pair. This may be inefficient in situations where the keys are not good indicators of the distribution of the dataset. This property also leads us to Datapath.

Datapath [3] is a state of the art data processing system. It is push based and data centric, which means that data circulates through the system and is utilized if and only if there exists a demand for it somewhere in the system. Datapath arranges its computation in the form of Workers and Waypoints. Waypoints are the entities in the system that manages computation and Workers are the entities that performs the actual computation. Waypoints are as generic as possible for a particular type of computation for example Scan, Print, Join, GLA etc. Workers are scheduled from the waypoints and they use one of the available threads and performs the computation allocated to it. The computations to be performed by the workers are as specific as it gets for a particular query. The code executed by the Workers are generated by a combination of user code and M4 templates. An incoming query Q1 is divided into as many waypoints as specified by the user and they are represented as a graph in the system.

Datapath has a column based storage and the data circulates through the system as chunks. The chunks essentially stores the data in columns and also a generalized schema is adopted to facilitate the carrying of multiple tuples by the same chunk. The schema would essentially have provisions to store all the different kinds of tuples.
present in the system at any time. This holds the constraint that data generated by a particular waypoint should be encoded in the form of tuples in order to convey it to another waypoint. To surpass this restriction we introduce State Passing (Chapter 3).

A second constraint in the original GLADE framework was its inability to perform internal iterations, that is there was no seamless way a particular query could have multiple passes over the input dataset. This affected the performance and flexibility of the original system as the iterations had to be performed externally and as a solution, we introduce Iterations (Chapter 3).
CHAPTER 3
SYSTEM ENHANCEMENTS

In order to describe Iteration and State Passing, we explore the notion of a Constant State. Constant State acts as an entity that can store and transmit information. In the case of iteration, the constant state carries information within the same waypoint and in state passing, between waypoints. We introduce some changes to the original format of GLAs,

- **GLA_Name() or GLA_Name(const ConstState*)**: Initialize() is absent and instead the arguments are passed to the constructor. The second variation is when a constant state is defined for that particular query in that waypoint.

- **AddItem(Tuple t) and AddState(GLA &other)**: exists as their former self.

- **bool Iterate(ConstState*&, int &numfrags)**: is used to pass the Constant State as a user modifiable parameter and also to get the number of fragments produced by the current iteration of the query. It also returns a boolean which says whether to iterate or not. Iterate() is optional and should only be defined if a constant state is declared.

- **Finalize()**: is used as a precursor to GetResult() when the results are extracted in multiple parts. The Finalize() call sets up the division of the state into multiple parts so that whenever GetResult() asks for a particular division of the state, the GLA will be able to provide that. This function is absent in the case of state passing.

- **GetResult(int &output1, int &output2,...........)**: is used to extract the tuples from the current GLA for the next waypoint. This function is also absent in the case of state passing.

3.1 Iteration

Iteration is required in cases where a computation requires multiple passes over the whole input data as in KMeans clustering, where the centers are gradually refined over multiple passes of data. In Datapath, the production of chunks are initiated by the final waypoint in the graph. It sends a message downstream which when received by a table scanner, it starts producing chunks. The chunks routed through the waypoints are utilized only by the ones that requires it. For each chunk processed by a required waypoint for a particular query, it sends an acknowledgement which is propagated
downwards. This lets any producer *waypoint* above to know that its data has been received.

Since the system is data driven, the chunks could arrive at a *waypoint* when there is not enough resources to process it. In such cases the system utilizes a form of load shedding and drops the newly arrived chunk. Since no acknowledgement is received for that particular chunk, it is routed again to this waypoint at a later time. So once *AddItem()* finishes the first pass over the entire dataset, all acknowledgements are sent out. However, we are able to determine whether another iteration is required only when *Iterate()* completes execution.

One way to re-route the chunks is to propagate as many drop messages downstream as there are chunks. This is a highly wasteful mode of transmission. Therefore, instead of dropping the chunks we initiate the production of the chunks once again as if for a new query. In case an iteration is required, the chunks are reinitiated and the *waypoint* is setup so that only the constant state remains from the previous iteration in the waypoint for that particular query.

If *constant state* is specified by the query for that *waypoint* then the constructor arguments are used to initialize the *constant state* initially by the system. Then it is
passed as a constant, read only state to the constructor of the GLA. Then the GLA uses the constant state to initialize itself and setup its state. The state is again sent to the GLA as a modifiable object in Iterate, wherein it can store more information or modify the current information for the next iteration if any.

In K-Means clustering, the constant state initially stores the K random cluster centers for starting the first iteration. After a single iteration, in PreFinalize, the newly generated centers are injected into the constant state and it carries the centers to the next iteration. Also, when the query finishes, the result would be present in the constant state as well. The constant state thereby maintains the volatile information between iterations.

We can infer that this is an efficient mechanism to incorporate iteration as it works orthogonally with the Datapath system as a whole. The mechanism of reinitiating chunks is essentially the same as the mechanism of initiating chunks for a new query. Also, the table scanner can choose to use a caching mechanism for chunks if it suspects a pending iteration which will further speedup the system.

Figure 3-1 shows the architecture. The chunk is sent by the producer waypoint and after processing it by calling AddItem() in User Space, the GLAWaypoint, the consumer here, sends an acknowledgement back. If an iteration is required, then it is determined in Iterate in User space. By the time user conveys the message to the GLAWaypoint, all acknowledgements have already been sent. It immediately sends a reinitiate message that prompts the producer to produce all the chunks again.

### 3.2 State Passing

Another advantage that came out of building the iterative mechanism was state passing. The only mode of communication between waypoints in the older system was through chunks that carried tuples. The mechanism was ineffective in situations where the state produced by one waypoint could be consumed by its successor. In such cases it was required to extract the tuples from the state in order to pass it to the next
Therefore, we needed a technique that could carry any non-relational state as it is, between waypoints. These states eliminated the generation of unnecessary artifacts wherever possible and also produced more efficient code. For example the algorithm of FPGrowth requires an initial scanning of the dataset to determine the frequency of each element. The main algorithm used this frequency to construct the FP-Tree. In such a situation, it would be too tedious to pass the element and its corresponding frequency as tuples.

Therefore, the constant states proved effective again. The chunks are modified so that they could carry the constant states just as they would have carried the tuples and the earlier mechanism of routing the chunks are applicable in this case as well. The only other problem for passing states in this manner is a mechanism to destruct them. For this, we included an acknowledgement in the chunk. The acknowledgement would be initialized to the number of waypoints that requires to receive the constant state. Whenever the chunks are received by a waypoint, the acknowledgement is decremented by one and if it is zero when it reaches a particular waypoint, then that constant state is passed into the user space to be destructed. This opened up further opportunities in some algorithms, especially in FP-Growth. We were able to analyze a part of the input dataset and produce an FP-Tree corresponding to that. Such disparately produced datasets could be sent to another waypoint using state passing, essentially the state being the FP-Tree. When this form of parallelism is coupled with an FP-Tree merge algorithm as detailed in section 4, we could produce a complete FP-Tree for the whole dataset.

3.3 State Passing and Iteration

State Passing and Iteration together produced even more powerful mechanisms for encoding computations. It allowed for dividing computations and merging them at a different waypoint in whatever format required. For example in Collaborative Filtering, the computations of the values in an output matrix is divided into blocks. These blocks
could be assimilated in another wayppoint without any additional costs due to State Passing.
CHAPTER 4
ALGORITHMS

We examine several data mining algorithms adapted for the GLADE environment starting with the Item-Based Collaborative Filtering, the K-Means Clustering, Decision Tree Classifier and Frequent Itemset Mining Algorithms including both Apriori and FP-Growth.

4.1 Item-Based Collaborative Filtering

Item Based Collaborative Filtering examines the relationship between items from an input data and based on these findings, it suggests recommendations for each of the items. Item Based Collaborative filtering involves the construction of a matrix of item-item pairs. The matrix represents the relationship between one item to the other. This matrix is constructed by using a similarity measure like pearson-correlation, cosine or or loglikelihood. For an incoming item A, the similarity is updated in the matrix between A and all other items linked to the same user. Once the matrix is built, recommendations for an item can be made based on its strongest relationships to other items. And a user who has bought several items will be provided a top-k of the recommendations found for each item from the matrix.

Proposed Algorithm: The construction of the item-item matrix pair is the most computationally intensive part of the item based collaborative filtering. The input is of the form

\[ UserId1, ItemId1, PreferenceValue1 \]
\[ UserId2, ItemId2, PreferenceValue2 \]
\[ UserId3, ItemId3, PreferenceValue3 \]

The item-item matrix, the central matrix that associates an item with all other items and their similarity measure, could be as huge as the square of the number of items. Therefore, it is not possible to maintain the whole matrix in memory for large datasets. In order to produce a scalable algorithm, the algorithm should focus on building a part of
the matrix at any point of time. It is also enough to consider either the upper triangular matrix or the lower triangular matrix as the matrix is symmetrical.

In our proposed implementation, we construct the matrix not on the ItemID values but on the hashed values of the ItemID. Therefore each row and column ranges from 0 to $2^{64}$ bits in size, for a 64 bit hash. This gives us uniformity for dividing the computations:

**Algorithm 4.1.0.1 State**

- **user2ItemPrefMap** - is an unordered map from UserIDs to a vector of a structure ItemPref. ItemPref carries ItemID and their Preference value
- **item2PrefMap** - is an unordered map from (Item1, Item2) and their statistics, an object of Statistics class as corresponding value
- **Statistics** - maintains number of elements n, sum of $X \times Y$, sum of $X$, sum of $Y$, sum of $X^2$ and sum of $Y^2$
- **ranges** - denotes a block of the matrix where computation is occurring currently. It will have an upperRow, lowerRow, upperColumn and lowerColumn values. This mechanism ensures that the intermediate data produced depends on the blocks size we specify, using the range values.

The algorithm is as shown in 4.1.0.2. In each iteration, the GLAs are initialized by the constant state $CFConst$ with a range. Each GLA only considers those tuples whose hashed value belong to the ranges. For each item, the current user has in the **user2ItemPrefMap**, statistics are updated for that particular item and the current item in the **itemItem2PrefMap**. Once the GLAs are accumulated using **AddState()**, PearsonCorrelation is applied to each of the maintained statistics in **Iterate()**. State Passing occurs by embedding the **user2ItemPrefMap**, **itemItem2PrefMap** and ranges in $CFConst$. Also, $CFConst$ produces a new set of ranges for the next iteration. The iterations subsequently finishes, when all the ranges have been exhausted.

Statistics class maintains enough information for applying Pearson Correlation.
Algorithm 4.1.0.2 CFItemGLA

```cpp
CFItemGLA(const ConstantState * CFConst)
ranges ← getNextRangesFrom(CFConst)
```

**AddItem(int UserID, int ItemID, int PrefValue)**

```cpp
hashedItemId ← hash(ItemID)
if hashedItemId ∉ ranges then
    return
end if
if UserId ∈ userItemMap then
    itemIDs ← user2ItemPrefMap.Find(UserID)
    for each item in itemIDs do
        Statistics ← item2PrefMap.
        Find(item.ID, ItemID)
        Statistics.Add(PrefValue, item.PrefValue)
    end for
    itemIDs.Add(itemID, PrefValue)
else
    user2ItemPrefMap.Add(UserID, ItemID)
end if
```

**AddState(GLA &other)**

```cpp
Merge(ranges, other.ranges)
Merge(user2ItemPrefMap, other.user2ItemPrefMap)
Merge(item2PrefMap, other.itemItem2PrefMap)
```

**bool Iterate(ConstantState * &CFConst, int &numFrags)**

```cpp
numFrags ← 1; // to perform State Passing
for each Statistics in item2PrefMap do
    ApplyPearsonCorrelation(Statistics)
end for
CFConst.currentRanges = ranges
CFConst.itemItemMap = itemItem2PrefMap
CFConst.userItemMap = user2ItemPrefMap
if CFConst.allRangesExhausted then
    return false
else
    return true
end if
```
Sample Pearson correlation coefficient,
\[ r = \frac{n\sum XiYi - \sum Xi\sum Yi}{\sqrt{(n\sum Xi^2 - (\sum Xi)^2))(n\sum Yi^2 - (\sum Yi)^2))}} \]

The Constant State here $CFConst$ provides the new ranges for each iteration. Each iteration computes a part of the matrix and the output can be accumulated as a state and sent to another waypoint that can either do post processing or serialize it to a file. The iteration ends when computations have occurred in each part of the matrix. If the right block size is selected, then the size of the intermediate data never gets too large.

### 4.2 K-Means Clustering

In K-Means, k random points are chosen initially from the whole dataset. Each point is assigned to one of the k groups based on the smallest distance between the point and the chosen centres. The distance can be measured using a distance measure such as Euclidean Distance. Once a pass over the dataset is completed, the new centres of the newly formed k groups are found. The algorithm terminates if the new centres and the previous centres only differ by a satisfiable threshold, called convergence delta. Otherwise, it continues to make a new pass over the dataset with the new centres.

**Proposed Algorithm:** K-Means clustering uses multiple passes over data to refine the cluster centers. The most computationally intensive part of the algorithm is the number of scans that have to be performed. We also eliminate a further overhead by delaying storing the association of point to the cluster centers until after all the iterations complete.

**Algorithm 4.2.0.3 State**

| centres    | stores k centres |
| statVector | statVector is an array of k Statistics object whose each index corresponds to one of the k centres. Also Statistics object stores int sum1, sum2, ..... (for each dimensions of the point) and int count |
Algorithm 4.2.0.4 Statistics Object for KMeans

AddPoint(int X, int Y, int Z) // Adding a Point to Statistics Object
\[
\text{sum}_1^+ = X; \quad \text{sum}_2^+ = Y; \quad \text{sum}_3^+ = Z; \quad ++ \text{count};
\]

Merge(Statistics other)
\[
\text{sum}_1^+ = \text{other}.\text{sum}_1; \quad \text{sum}_2^+ = \text{other}.\text{sum}_2; \quad \text{sum}_3^+ = \text{other}.\text{sum}_3; \quad \text{count}^+ = \text{other}.\text{count};
\]

Finalize()
\[
\text{sum}_1/ = \text{count}; \quad \text{sum}_2/ = \text{count}; \quad \text{sum}_3/ = \text{count};
\]

convergenceDelta is the tolerable difference in distance between the current centres and the newly calculated centres, a measure of convergence. The algorithm stops iterating when the difference is less than or equal to convergence delta.

Initially KMeansConst is initialized with K random centres.

The algorithm is as shown in 4.2.0.5. The constant state, KMeansConst stores the k random centres in the first iteration and stores the newly calculated centres in subsequent iterations. It initializes the GLA with the centres. For each incoming tuple in AddITem(), the GLA determines the centre to which the tuple has the minimum distance. The tuple is added to the statistics corresponding to this centre. The statistics maintained are the sum of each co-ordinates in the point and also the number of points. In Iterate(), once new centres are found, their difference in distance from the original centres are calculated and if they differ by more than the convergence delta, then a new iteration is performed.

EuclideanDist(point1, point2) calculates the Euclidean distance between two points.

Here, once iterations are complete, another function could run that associates each point to one of the centres.

4.3 Decision Tree Classifier

The training part of the decision tree classifier builds a tree whose leaf nodes contain one of the output classes and each of the intermediate nodes contains a decision. A decision denotes a particular attribute and one of its values. The trees are
Algorithm 4.2.0.5 KMeansGLA

KMeansGLA(const ConstantState * KMeansConst)
\[ k\text{Centres} \leftarrow K\text{MeansConst}.\text{getCentres} \]

AddItem( inTuple)
\[
\text{closestCentre} \leftarrow 0 \\
\text{minDistance} \leftarrow \text{EuclideanDist}(\text{kCentres}[0], \text{inTuple}) \\
\text{for } i \leftarrow 1; i < \text{kCentres}.\text{size}(); ++i \text{ do} \\
\quad \text{distance} \leftarrow \text{EuclideanDist}(\text{kCentres}[i], \text{inTuple}) \\
\quad \text{if } \text{distance} < \text{minDistance} \text{ then} \\
\quad\quad \text{minDistance} \leftarrow \text{distance} \\
\quad\quad \text{closestCentre} \leftarrow i \\
\text{end if} \\
\quad \text{Statistics} \leftarrow \text{statVector[closestCentre]} \\
\quad \text{Statistics}.\text{AddPoint(inTuple)} \\
\text{end for}
\]

AddState(GLA &other)
\[
\text{for } i \leftarrow 0; i < \text{statVector}.\text{size}(); ++i \text{ do} \\
\quad \text{statVector}[i].\text{Merge(other.statVector[i])} \\
\text{end for}
\]

bool Iterate(ConstantState *&KMeansConst, int &numFrags)
\[
\text{newCentres} \quad \triangleright \quad \text{new centre} \\
\text{numFrags} \leftarrow 1; \quad \triangleright \quad \text{to perform State Passing} \\
\text{iterateFlag} \leftarrow \text{false}; \\
\text{for } i \leftarrow 0; i < \text{statVector}.\text{size}(); ++i \text{ do} \\
\quad \text{statVector}[i].\text{Finalize} \\
\quad \text{newCentre} \leftarrow \text{statVector}[i].\text{getNewCentre} \\
\quad \text{distance} \leftarrow \text{EuclideanDist}(\text{kCentres}[i], \text{newCentre}) \\
\quad \text{if } \text{distance} > \text{convergenceDelta} \text{ then} \\
\quad\quad \text{iterateFlag} \leftarrow \text{true} \\
\quad\quad \text{numFrags} \leftarrow 0 \quad \triangleright \quad \text{no state is passed out} \\
\text{end if} \\
\quad \text{newCentres}.\text{Insert(newCentre)} \\
\text{end for} \\
\text{KMeansConst.centres} \leftarrow \text{newCentres} \\
\text{return iterateFlag}
\]
built so that during the testing part of the algorithm each incoming tuple traverses the
tree based on their corresponding value for each decision and ends up in one of the leaf
nodes which determines the class to which the tuple belongs.

For categorical attributes, if attribute $A = a, b, c$ then if at a node the decision is
taken on attribute $A$, then three splits will occur $A = a, A = b$ and $A = c$.

For real valued attributes the same cannot be said, as there will be $n$ splits if $n$ real
valued attributes are present. Therefore, a single value is chosen that will maximize the
efficiency of the split. $B = 1, 2, 3, 4 \ldots x, \ldots$, the decision will look like $B \leq x$ and $B > x$, hence resulting in a binary split. Here, $x$ is called the split value.

Different methods can be employed to choose an attribute at each node, such
as Information Gain or Gini Index. We focus on Information Gain for our calculations.
Information Gain chooses an attribute in such a manner that, that particular attribute
reduces the entropy for the whole dataset by the maximum. The classic algorithm
employs the method of splitting the dataset per decision taken and then repeating the
process at each new nodes. The computation is highly intensive as for calculating the
information gain, one pass through the subset of the dataset at each node is required.
For real valued attributes the problem becomes more intensive as the split value is also
required to be found which is done by assuming each value as the split value and finding
the information gain for that value. The value with the maximum information gain is
chosen as the split value for that particular attribute.

We focus our algorithm on the real valued attributes.

**Proposed Algorithm:** The proposed implementation focuses on level wise
building of decision trees with each level requiring one pass of data. As mentioned
earlier, the classical approach uses data division at each node to this effect but our
implementation does not perform partitioning. Instead we make a new iteration of the
data through the partially grown tree again. The tuples propagate through the tree and is
added to the statistics of the node that they ends up at. Therefore the focus is on a set
of nodes at a particular iteration and on their children, if any in the next iteration and so on. Let \( \text{curNodes} \) contain the current nodes.

### Decision Tree Node
- \( \text{decisionAtt} \) - for storing the decisive attribute
- \( \text{decisionVal} \) - stores its corresponding splitting value
- \( \text{leaf} \) - flag which shows whether it's a leaf or not
- \( \text{left and right node} \) - pointers to children

### State
- \( \text{root} \) - stores the root of the Decision Tree
- \( \text{node2StatMap} \) - maps each of the current nodes to a Statistics object. For each attribute, the Statistics objects maintain the count of each element.

### Statistics object
- \( \text{outputClass2CountMap} \) - maps the output class of the tuple to its Count
- \( \text{attributesStatVector} \) - holds an element2StatMap for each attribute
- \( \text{element2CountMap} \) - maps each element in a particular attribute to Count object.
- \( \text{Count object} \) - stores both the count of the elements and \( \text{outputClassCounter} \) - a map between outputClass to its count

### Algorithm 4.3.0.6 Decision Tree Statistics Object

**AddTuple(Tuple t)**

For each element of \( t \), count is incremented after reaching the element using the required indirection

The algorithm is shown in 4.3.0.7

### 4.4 Apriori

The algorithm was introduced in [5] and is based on the property that the subset of a frequent itemset must be frequent, this property allows the algorithm to incrementally build longer candidates from shorter ones. In the algorithm, first all the candidates
Algorithm 4.3.0.7 DTreeGLA

DTreeGLA(const ConstantState DTreeConst)

curNodes ← DTreeConst.getCurNodes
root ← DTreeConst.getRoot

AddItem(inTuple)

node ← root

while node! = leaf || node ! curNodes do
    if intuple[node.decisionAtt] < node.decisionVal then
        node = node.left
    else
        node = node.right
    end if
    if node ∈ curNodes then
        Statistics ← node2Map.Find(node)
        Statistics.AddTuple(inTuple)
    end if
end while

AddState(GLA &other)

Merge(node2StatMap, other.node2StateMap)

bool Iterate(ConstantState * & DTreeConst, int &numFrags)
	newNode2StatMap - holds the nodes for next iteration

for each element in node2StatisticsMap do
    node ← element.node
    Statistics ← element.Statistics
    if 1 < — no: of output class in Statistics then
        node.OutputClass ← Statistics.getOutputClass
        MakeLeaf(node)
        continue
    end if
    for each element2 CountMap in attributesStatVector do
        for each element in element2CountMap do
            (belowElementCount, belowClassCount)
            ← calculateElementsAndClassCountBelow(element)
            (aboveElementCount, aboveClassCount)
            ← calculateElementsAndClassCountAbove(element)
            Using above statistics, find maximumIG and splitValue for this attribute
        end for
        Find Attribute with maximumIG
        Store Attribute as decisionAttribute and its splitValue as decisionValue for this node
    end for
    Add Children of node to newNode2StatMap
end for
if 0 ← newNode2StatMap.size() then
  return false ▷ iterations have finished
end if

DTreeConst.node2StatMap ← newNodes ▷ store the new nodes in the constant state
return true ▷ iterate

are generated that have a size 1, 1-item candidates. The support count for these are counted and the ones that fall below the minimum support are eliminated. Then the 2-item candidates are generated from the 1-item and the process continues. The algorithm finishes when there are no more candidates to be generated or when the support count of none of the candidates is above the threshold. Some of the more famous implementations of Apriori are Ferec Bodon’s implementation using trie-based data structure and candidate hashing [6], Christian Borgelt’s implementation [7] using recursion pruning and Bart Goethals’ implementation using Agrawals algorithm [8]. Bit vectors have also been used for Apriori implementation [9], [10], where the transactions in the dataset are represented as bitmaps - a 1 if present and a 0 if not.

Both Apriori and FP-Growth requires a list of the unique elements in the dataset and their number of occurrences. We call that the F-List, Frequency List. One simple GLA waypoint computes the F-List by using one pass over the whole dataset. The F-List is sent to the main computational GLA waypoint via state passing.

Proposed Algorithm: We combine the Trie based Apriori with the bit vector implementation. The bit vector is used for support counting and the trie is used for candidate generation. The F-List is fed to two GLAs. One of them feeds it to a Trie and then the trie is passed as APConst to APGLA and the other will use it to make a bitMapper that maps each of the elements in F-List to one of the 64 or 32 bit indices of the bitmap. Using the bitmapper, it converts the incoming dataset to bit vectors and these vectors are fed to APGLA. The GLAs in each iteration counts the occurrences of each pattern, as shown in AddItem(). In the first iteration it counts the occurrences of 1-item candidates, in the second iteration 2-item candidates and so on. Once the
support count is found, it discards those patterns that are below the minimum support are eliminated. The new candidates are fed to the Trie in \textit{APConst} which generates the new set of candidates. A new iteration begins with the new set of candidates. Iterations finish when there are no more candidates left after support counting and elimination or when \textit{APConst} fails to generate new candidates.

\begin{verbatim}
State
Candidates2CountMap - Maps the candidates to its counts

The algorithm is as shown in 4.4.0.8

Algorithm 4.4.0.8 APGLA
APGLA(const ConstantState APConst)
Trie ← APConst.getTrie
Candidates2CountMap ← Trie.getNewCandidates()
bitMapper ← APConst.getBitMapper

AddItem( inTuple)
for candidate in Candidates2CountMap do
    if all bits in candidate are set in inTuple then
        Increment its Count
    end if
end for

AddState(GLA &other)
Merge(Candidates2CountMap, other. Candidates2CountMap)

bool Iterate(ConstantState *&APConst, int &numFrags)
Trie ← APConst.getTrie
for element in Candidates2CountMap do
    if element.count ≥ minSupport then
        Trie.Add(element.Candidate)
    end if
end for
Trie.generateNewCandidates
if ( Trie.getNewCandidates.size )
    return false ▷ No Iteration
else
    return true ▷ Iterate
\end{verbatim}

31
Candidate Generation in Trie: The candidates are generated using a trie. The Trie would only contain elements that are all above minimum support, so no pruning is required. The candidates are generated at level l+1 in the following manner:

- all nodes in level l-1 are added to a vector
- For each node in the vector, take each child as curChild
- For every curChild, add all the siblings to its right as its children
- generate all the words of length l+1, which will be our new candidates

4.5 FPGrowth

The significance of FP-Growth as introduced in [11] is that it reduces the number of scans of the dataset to two and eliminates candidate generation. This is an improvement over algorithms like Apriori which scans the database as many times as the depth of the mining and also generates candidates in each step. FP-Growth uses a compact structure called FPTree at its core. An FPTree is self sufficient in containing all the informations required for the mining tasks. FPTree is a prefix tree in which the nodes having the same elements are linked and once built, it is recursively mined to generate the frequent itemset. The initial focus on improving the algorithm has been on bettering the performance using single threaded implementations as in [12] and [13]. Pramudiono et al. [14] proposed an implementation for shared nothing architecture. After determining the frequency of each of the single items present in the database and forming an F-List by retaining the items that have an occurrence above minimum support, each of the generated pattern bases is mapped to and processed by one of the shared nothing nodes. With the advent of MapReduce, [15] designed a parallel FP-Growth algorithm that utilized the computational model. Here, each node grew its own FPTree in parallel. The F-List was partitioned into various lists called as g-lists and the g-lists were distributed to different nodes. During the mining a node added only those patterns that were present in its g-list to its FPTree. Zhou et al. [16] introduced a form of load balancing to the parallel FP-Growth implementation introduced in [15]. They made
estimations based on the initial $F$-List assuming that the number of occurrences of an element corresponds directly to the length of the longest frequent path in the conditional pattern base. And the division of the $F$-List into $g$-list is influenced by this notion of load.

**Proposed Algorithm**: The FP-Growth implementation in Mahout uses the mechanism introduced in [15]. It deals with growing the FPTrees in parallel, mining on them separately and then compiling each of the top-k items produced by them to obtain the global list. Our implementation focuses on an FPTree merge algorithm, 4.5.1. This enables the $AddState()$ to be associative. We build the the FPTrees in parallel in $AddItem()$. The individually built FPTrees can be accumulated to a single FPTree in $AddState()$ using FPTree merge. Also, with the use of State Passing, this tree can be passed to a different waypoint which can mine on it. State Passing and Iteration also gives further opportunities for parallelism here. The initial dataset can be divided and each of the divisions can be mined separately using individual instances of FPGlas.

The F-List is used to initialize $FPG-Const$, but it is finalized before initialization by eliminating elements below minimum support, $minSupport$.

---

**State**

- $FList$ - holds the F-List
- $FPTree$ - holds the FPTree

**Algorithm 4.5.0.9 FPGLA Statistics Object**

- $Merge(FPTree\ other)$
  - Refer to Section 4.5.1

The algorithm is as shown in 4.5.0.10

**4.5.1 FPTree Merge**

The algorithm for Merge is as follows:

Let node be a data structure that contains a data element, count of its occurrence and the links to its child nodes.

$Merge(FPTree\ other)$
Algorithm 4.5.0.10 FPGLA

FPGLA(const ConstantState FPGConst)

1. \( FList \leftarrow FPGConst.getFList \)

AddItem( inTuple)

\[
\begin{align*}
\text{sort}(\text{inTuple}, FList) & \quad \triangleright \text{sorts in decreasing order according to the frequencies in } FList \\
\text{seenElements} & \quad \triangleright \text{elements in t that have already been seen} \\
\text{for } i & \leftarrow t.size() - 1; i \geq 0; ++ i \text{ do} \\
& \quad \text{if } ! \text{seenElements.Find}(t[i]) \text{ then} \\
& \quad \quad TTupleup \leftarrow \text{CopyElementsFrom } t[0] \text{ to } t[i] \\
& \quad \quad FPTree.\text{AddTuple}(tup) \\
& \quad \text{end if} \\
& \text{end for}
\end{align*}
\]

AddState(GLA &other)

\[
\begin{align*}
FPTree.\text{Merge}(other.FPTree)
\end{align*}
\]

bool Iterate(ConstantState &FPTreeConst, int &numFrags)

\[
\begin{align*}
encode \ FPTree \ in \ \text{FPTreeConst} \\
\text{numFrags} \leftarrow 1 \quad \triangleright \text{for State Passing} \\
\text{return false;} \quad \triangleright \text{no iteration required}
\end{align*}
\]

1. Initialize, Node oRoot as other.root

2. Scan the child nodes of oRoot, store the reference in oNode and for each iteration do step 3

3. Call MergeRecursive(curNode, oNode)

\textbf{MergeRecursive(curNode, oNode)}

1. Checks whether the data of oNode is present in curNode’s children. If it is, let the newly found node be foundNode and go to Step 2. If not found then go to Step 3.

2. Add the count of oNode to the count of foundNode. And set curNode to foundNode.

3. Create a new node and add oNode’s data and count to it. Add the new node as a child to curNode. Set curNode to the new node.

4. Scan the children of oNode and for each iteration call mergeRecursive(curNode, oNode.child)

Merge utilizes a levelwise combination of nodes of one FPTree with the other.

\textbf{4.5.2 Proof of Correctness}

For the proof of correctness of the entire algorithm, two results are proved.
Insertion in FPTree is order independent
For this it is enough to prove that,
\[ ((S_1 + T) + S_2) = (S_1 + (T + S_2)) \]
where \( S_1 \) - Sequence 1, \( S_2 \) - Sequence 2, \( T \) - FPTree.
Sequences 1 and 2 can be considered as patterns mined from an incoming tuple. Two statements are considered for the proof:

**Statement 1**: FPTree is a prefix tree and in a prefix tree, if two sequences have the same prefix then they will follow the same path in the tree until the end of the common prefix.

**Statement 2**: Two different sequences, when added at a particular node of an FPTree, will diverge and the structure of the node and the added sequences will not differ based on the order of the addition of the sequences.

Assumption - Assuming that,
\[ ((S_1 + T) + S_2) \neq (S_1 + (T + S_2)) \]

Three scenarios are possible here,

1. In this case, \( S_1 = S_2 \). Therefore, we can consider all the prefixes to be same. So according to **Statement 1** they must follow the same path from root to leaf.

2. \( S_1 \neq S_2 \). Both sequences are entirely different and have no common prefix. In this case they will diverge at the root node itself. Therefore, by **Statement 2**, the order of addition of \( S_1 \) and \( S_2 \) does not render the structure of the tree different.

3. \( S_1 \neq S_2 \) but they have a common prefix. Here, the two sequences follow the same path until the common prefix ends, then they diverge from the last common element until their corresponding leaf nodes. Therefore, this is a combination of **Scenario (1)** and **Scenario (2)**, so the trees formed cannot be different.

Therefore, insertion in an FPTree is order independent.

**Correctness of FPTree merge for a single path FPTree** - Figure 4-1 shows the particular scenario, that is adding a sequence to an FPTree having only a single path.
Consider a single path FPTree to be merged to a larger FPTree. Merging a single path is completely synonymous to insertion. So if insertion as a process is correct, merging a single path is also correct.
Therefore, the same can be said for a multi path FPTree as well, which is recursive calling of a single path FPTree merge algorithm.
From the two proven results, the proof of correctness of the FPTree merge algorithm is established.

4.5.3 Time Complexity

Merging of two trees $T_1$ and $T_2$ requires traversing $T_2$ once using DFS and following the same path in $T_1$. If nodes are not found they are created in $T_1$ which is done in constant time. Therefore, if $T_2$ has $V$ vertices and $E$ edges, then the time complexity would be $2 \cdot O(V+E)$. That is $O(V+E)$ same as DFS traversal.
CHAPTER 5
EMPIRICAL EVALUATION

All experiments were conducted on an AMD Opteron(TM) Processor 6128 with 64 GB RAM and 16 core architecture. Our goal is to first of all compare the performance speedup of GLADE with the corresponding Mahout implementation on similar parametric conditions, whenever such an implementation exists in Mahout. Also, we show the scalability of GLADE for varying chunk sizes and cores in some of the algorithms. For fixed core comparisons, the algorithm was run on all 16 cores of the system. Hadoop, the MapReduce implementation on which Mahout runs was setup on a Pseudo Distributed mode where each Hadoop daemon runs in a separate Java process thereby giving the impression of a pseudo cluster. It was also configured to run 8 Map and 8 Reduce task. The complete set of Hadoop and Mahout configurations used for our experiments are as shown in Table 5-1 and Table 5-2 respectively.

Table 5-1. Hadoop Configurations

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
<th>File Used</th>
</tr>
</thead>
<tbody>
<tr>
<td>dfs.block.size(bytes)</td>
<td>1048576 bytes</td>
<td>hdf-site.xml</td>
</tr>
<tr>
<td>mapred.map.tasks</td>
<td>8</td>
<td>mapred-site.xml</td>
</tr>
<tr>
<td>mapred.reduce.tasks</td>
<td>8</td>
<td>mapred-site.xml</td>
</tr>
<tr>
<td>mapred.map.child.java.opts</td>
<td>-Xmx20000M</td>
<td>mapred-site.xml</td>
</tr>
<tr>
<td>mapred.reduce.child.java.opts</td>
<td>-Xmx20000M</td>
<td>mapred-site.xml</td>
</tr>
<tr>
<td>mapred.compress.map.output</td>
<td>false</td>
<td>mapred-site.xml</td>
</tr>
<tr>
<td>fs.inmemory.size.mb(MB)</td>
<td>2048 MB</td>
<td>core-site.xml</td>
</tr>
<tr>
<td>io.sort.mb(MB)</td>
<td>2048 MB</td>
<td>core-site.xml</td>
</tr>
<tr>
<td>HADOOP_HEAPSIZE(MB)</td>
<td>60000 MB</td>
<td>hadoop-env.sh</td>
</tr>
</tbody>
</table>
Table 5-2. Mahout Configurations

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
<th>File Used</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAHOUT_HEAPSIZE(MB)</td>
<td>60000</td>
<td>bin/mahout</td>
</tr>
</tbody>
</table>

5.1 Item-Based Collaborative Filtering

Two datasets are used to evaluate the performance of the algorithm. The first is the Book-Crossing Dataset \(^1\) mined by Cai-Nicolas Ziegler, DBIS Freiburg and the second is the MovieLens dataset\(^2\).

5.1.1 Book Crossing Dataset

The non-numerical data were discarded from processing here. As shown in figure 5-1, experimental results show that the algorithm is fastest when the size of the chunk is in a medium size range. The performance loss at lower chunk sizes can be attributed to the contention for data by the threads on the lock that allocates the chunks whereas the lesser speed at higher chunk sizes can be attributed to the memory bandwidth due to movement of large states. Also, some of the GLAs towards the end becomes idle as all the chunks have already been allocated.

5.1.2 Movielens

Movielens data contained three datasets - "100K" - consisting of 100K ratings (amounting to 956 KB), "1M" - consisting of 1 million ratings (amounting to 11.0 MB) and "10M" consisting of 10 million ratings (amounting to 123.4 MB). We used SIMILARITY_PEARSON_CORRELATION as the similarity class in both the GLADE as well as the Mahout implementation. In the GLA format the item-item similarity matrix was constructed taking into consideration all relevant users, items and preference values

\(^1\) http://www.informatik.uni-freiburg.de/~cziegler/BX/

\(^2\) http://www.grouplens.org/node/12
and no filtering was employed. The chunksize for the GLA was kept constant at 1000th of the input size. Experimental results showed that GLA based implementations were almost 3x faster when using 10M dataset, almost 9x faster on 1M dataset and almost a 100x faster on the 100K dataset, as shown in Figure 5-2. The initiation and recovery from the hadoop mappers and reducers could be the reason for this vast difference at lower level computations. This becomes tolerable as the computation time topples this overhead by a large margin.

5.2 K-Means Clustering

For running the KMeans algorithm we used the 281.6 KB, synthetic control dataset. We synthesized larger datasets of 141MB, 1.2 GB and 5GB from the same dataset. KMeans was run with a convergence delta of 0.5 on all cases and also we have taken the values averaged on 10 instances of the experiments so that it evens out the time difference that might be caused by the initial random seeding of the centres.

---

Figure 5-2 shows the comparison of GLADE with the Mahout implementation. We used the default script provided: `build-cluster-syntheticcontrol.sh` and changed the dataset accordingly. Also, the Mahout implementation had a maximum number of iterations parameter which was set to 10 and GLADE ran as many iterations as required for the convergence. Therefore, we decided that per iteration time would be a better comparison parameter. We obtained a speedup of 2332x speedup for 281 KB dataset, 246x speedup for the 141 MB dataset and a 441x speedup for the 1.2 GB dataset. In the 1.2 GB dataset, Mahout ran out of memory in the 10th iteration. For the 281KB dataset, time taken in initializing and terminating the map and reduce phases dominated the computational time which evens out for the 141MB size. Again the reduced performance in the 1.2 GB region is because of the large amount of intermediate data generated by Mahout.

Figure 5-4 shows the scalability for the GLADE implementation of KMeans. The experiment was run on the 5 GB dataset. We have used a convergence delta of 0.5 for the algorithm and the graph shows the variation on time due to k. Also, we have used
the computational time for the algorithms to converge rather than the per iteration time in this graph.

![K-Means: GLADE vs Mahout](image)

**Figure 5-3.** Per Iteration Speedup, KMeans : GLADE vs Mahout [log scale]

![K-Means (GLADE)](image)

**Figure 5-4.** Scalability of KMeans(GLADE) Algorithm
5.3 Decision Tree Classifier

Figure 5-5 shows the comparison of the implementation for various number of chunks. The algorithm was carried out on a synthesized dataset whose every attribute was real valued and the total file size amounted to 3.4 GB. Experimental results are shown when using 16 cores, 8 cores and 2 cores. It shows the scalability of the machine in the shared memory architecture.

![Decision Tree Classifier (GLADE)]

Figure 5-5. GLADE Decision Tree

5.4 Apriori

Our Apriori implementation was run on the Google ngrams dataset\(^4\), all the 1-grams [the one word dataset] combined to form a 9 GB file. 16 cores were used for the comparison with the FP-Growth algorithm as shown in figure 5-6. The comparison was made on the same dataset and the same number of cores. It can be observed that the algorithm was slower at lesser minSupports. This can be attributed to the fact that the depth of the mining is higher at this point. And as minSupport increases the depth

\(^4\) [http://books.google.com/ngrams/datasets](http://books.google.com/ngrams/datasets)
of mining decreases hence giving Apriori better performance. FPGrowth maintains a relatively constant performance throughout as it does not depend on the minSupport as much as the Apriori algorithm does.

![GLADE Apriori vs GLADE FPGrowth](image)

**Figure 5-6.** GLADE Apriori vs GLADE FPGrowth

### 5.5 FPGrowth

The experiments were carried out on Traffic Accidents Data Set provided by Karolien Geurts\(^5\). The data was 33.9 MB in size, containing 11500870 transactions. The comparison of the GLADE implementation versus Mahout can be found in Section 5-7. In Mahout, the whole computation was divided into five MapReduce tasks of which the most important was the ParallelFPGrowth task, which was actually responsible for generating the required patterns and growing the FPTrees. The Mahout implementation then mined on each of the FPTrees thus generated and produced a local top-k items. These items were compiled at a later stage. Since our algorithm focused on

---

\(^5\) [http://fimi.ua.ac.be/data/](http://fimi.ua.ac.be/data/)
generating a final FP-Tree, the tasks after the ParallelFPGrowth task were ignored for the comparison, although the GLA version has the extra task of combining the separate trees to a single one.

Experimental results showed a speedup of 17x over the Mahout implementation. The time for the computationally intensive part as well as the total time including preprocessing is shown in the graph. The number of mappers and reducers used were eight and the number of GLAs used were also 8. We decreased the block size from a default 30 MB to 1 MB, which should amount to around 30 map tasks and 30 reduce tasks for each mapreduce jobs as the input size is close to 30 MB.

Figure 5-7. FPGrowth GLADE vs FPGrowth Mahout [In log scale]
CHAPTER 6  
DISTRIBUTED STATE

The state that a GLA handles may grow very large. In such cases, it is difficult to manage the state and there is an overhead on performance as well. In this section, we propose a new mechanism to manage the large states in GLAs known as Distributed state as compared to the already established replicated state. In the replicated state, the GLAs are pairwise combined until a single GLA remains. Therefore, for large states, the states get even larger as aggregation proceeds. Such a technique is fast but loses some of its efficiency when the state becomes large and have to be moved around. Therefore, the movement of data becomes more difficult as we go up the aggregation tree. A successful substitute should be able to control its state from getting too large and also should be able to efficiently manage the transmission of the states for aggregation.

Distributed state does not use an aggregation tree for combining the GLAs. It maintains a global state, to which all GLAs combine. The global state makes sure that the amount of data that has to be moved is less than that in an aggregation tree. However, a locking mechanism is required for access to the global state which might degrade the performance. Therefore, we focus on partitioning the global state and locking each of these partitions. The local states would essentially maintain the same partitions and each of the local partition would combine with its corresponding global partition. We use a hash based dividing scheme as shown in 6.1. This scheme would reduce the contention for the locking mechanism, but also still enable better management of states.

However, there is still no guarantee as to how large the local states would grow for a particular GLA. For this we incorporated a flushing mechanism. As soon as a new chunk arrives for a GLA, it checks whether any of the partitions exceeds a predetermined size. If it does, then it combines that partition with its global state then and there and carries on with the computation. If the right flush size is specified, this mechanism would make
sure that local states never grow uncontrollably large and also only data the size of the flush size is transported through the system.

6.1 FP-Growth Distributed

The division of the global state is done by partitioning the F-List which acts as the key to the hash. Each division of the F-List corresponds to a bucket of the hash that grows an FPTree. An instance of the same hash is passed to all the GLAs as local state. So whenever a pattern is produced while scanning a tuple, its corresponding partition is found in the local hash and the pattern is added to it. And in this fashion the GLAs build trees on each partition. Aggregation occurs during flushing as well as when a GLA completes its computation. The algorithm for producing the FPTree remains the same from the distributed implementation.

![Figure 6-1. The Distributed State](image)

6.2 Empirical Evaluation

The Distributed state was evaluated on the same 16 core AMD Opteron(TM) machine as mentioned earlier. The mining was performed on the same Google ngrams dataset\(^1\), 9 GB dataset as mentioned in the 5.4. The comparison of the replicated state to the distributed state can be seen in figure 6-3. The mining was for the most frequent letters in the 1-grams. So the maximum length of the F-list was limited to 26. The number of GLAs used were 16 and the variant used for the comparison was the

\(^1\) [http://books.google.com/ngrams/datasets](http://books.google.com/ngrams/datasets)
Figure 6-2. The Replicated State

chunk size. The number of locks that were used were kept at a 1:1 ratio with the number of partitions. The speedup achieved was almost 3x faster than the replicated state.

Figure 6-3. Replicated vs Distributed State
CHAPTER 7
RELATED WORK IN MAPREDUCE

Iterative processes are not efficiently encodable in the original MapReduce system. Iterations can be performed by using individual MapReduce phases for each iteration and chaining them one after the other. However, in each iteration the map and reduce jobs have to be restarted. This results in wastage of resources as the same Map Reduce tasks are used in successive iterations. Also, the input data have to be processed and shuffled in every iteration, which also affects performance. MapReduce also lacks a mechanism to check for the terminating condition efficiently and it may require another MapReduce task to do that. Different algorithms have been proposed for facilitating iterative computation in MapReduce.

HaLoop [17] introduced a caching solution that cached the loop invariant data and also the output of the reduce tasks. But there was no improvement over the inherent problem of restarting tasks.

Twister [18] introduced a form of long running MapReduce tasks that reused the tasks for next iteration. However, the solution depended on the assumption that the input dataset and the intermediate data fits into the distributed memory of clusters which is not true in many situations.

iMapReduce [19] also produced an iterative model in its implementation but it made an assumption that the same key is used by both the maps and reducers. It also used a form of task reuse known as persistent tasks in its implementation, but still unnecessary resources were wasted as the persistent tasks remained dormant in memory until a decision was taken whether to reuse it or terminate it. The implementation also used a form of caching in the reduce phase to allow termination.

iHadoop [20] introduced an asynchronous implementation of MapReduce that made iterations execute asynchronously whenever the input data was available for it. The implementation still had no improvements in the reuse of tasks.
All the implementations mentioned above suffers from performance loss in carrying out iterative computations either due to the shortcomings in the proposed model itself or mainly due to the inherent shortcomings of the MapReduce framework.

Iterative GLAs in GLADE can easily be encoded as shown by the already presented algorithms. The overhead is also minimal as it only involves cleaning up some memory and propagating a message down the graph. We reuse the same waypoint and the generated code for iterative computations and this is not wasteful as no task remains idle or dormant due to task reuse. Also, the reinitiation of data for a new iteration is orthogonal to the Datapath framework and since it is push based and data centric, no overhead is incurred for this either. The termination condition is also encoded in a natural format as it can be specified in the GLA itself. Therefore, iterative mechanism in GLADE is tightly coupled to the already existing mechanisms present in Datapath as well as the format of the GLAs.
CHAPTER 8
CONCLUSION

This thesis introduced enhancements to the GLADE framework in the form of Iterative Computations and State Passing. Iterations enabled us to perform internal iterative computations and State Passing allowed communication of non-relational data. With the help of these enhancements, we efficiently adapted several machine learning algorithms such as FP-Growth, Apriori, ItemBased Collaborative Filtering, Decision Tree Classification and KMeans Clustering to the GLA based format. Experimental results showed upto 441x speedup for large datasets and 2332x speedup for small datasets when compared to their counterparts in Hadoop. We believe we have established a natural method of encoding complex computations in the form of GLAs in this thesis.
REFERENCES


BIOGRAPHICAL SKETCH

Daley Alex was born in Kerala, India. He completed the major part of his schooling in Cochin and went on to do his Bachelors in Information Technology at Mahatma Gandhi University, Kerala in the same city. Shortly thereafter he joined the Department of Computer and Information Science and Engineering at University of Florida, Gainesville for doing his master's in Computer Engineering from which he graduated in the Summer of 2012. His research interests are in Data Intensive Computing, Database Systems and Technology and in Parallel Computing.