**Implementation of classifier tool in Twister**

Magesh Vadivelu, Shivaraman Janakiraman

*{*[*magevadi, shivjana}@indiana.edu*](mailto:magevadi,%20shivjana%7d@indiana.edu)

School of Informatics and Computing

Indiana University Bloomington

**Abstract**

Mining frequent item-sets from large-scale databases has emerged as an important problem in the data mining and knowledge discovery research community. To overcome this problem, we have proposed to implement Apriori algorithm, a classification algorithm, in Twister, a distributed framework, that makes use of MapReduce. MapReduce is a programming model and an associated implementation for processing and generating large data sets. We specify a map function that processes a key-value pair to generate a set of intermediate key-value pairs, and a reduce function that merges all intermediate values associated with the same intermediate key. Programs written in this functional style are automatically parallelized and executed on a large cluster of machines. The run-time system takes care of the details of partitioning the input data, scheduling the program’s execution across a set of machines. Our implementation of Apriori algorithm runs on a large cluster of machines and is highly scalable. On an application level, we can use this apriori algorithm to identify the pattern in which customers buy products in a supermarket.

*.*

**Key Words:** Twister, MapReduce, Parallel Computing, Apriori Algorithm.

**1. Introduction**

The Apriori Algorithm[11] is an influential algorithm for mining frequent itemsets for Boolean association rules.

* Frequent Itemsets: The sets of item which has minimum support (denoted by Li for ith -Itemsets).
* Apriori Property: Any subset of frequent itemset must be frequent.
* Join Operation: To find Lk, a set of candidate k-itemsets is generated by joining Lk-1with itself.

With Hadoop [7], programmers can more focus on the business logic of their application without care too much about fault tolerance, task scheduling, and workload balancing issues. However Hadoop do not have built-in support for iterative programs, which is a common approach for many applications: K-means, PageRank, Markov chain.

Twister [9] is an enhanced MapReduce runtime that supports iterative MapReduce computations efficiently. It uses a publish/subscribe messaging infrastructure for communication and data transfers, and supports long running map/reduce tasks, which can be used in ‘configure once and use many times’ approach. These improvements allow Twister to support iterative MapReduce computations highly efficiently compared to other MapReduce runtimes.

**1.1 Apriori Algorithm**

Since the introduction of association rules by Agarwal et al [4], frequent pattern (itemset) mining has been a fundamental and essential component in data mining and knowledge discovery. The candidate generation-and-test methodology, called the Apriori technique, was the first technique to compute frequent patterns, based on the Apriori principle (i.e., the anti-monotone property).

The task of association rule mining has received a great deal of attention. Today the mining of such rules is still one of the most popular pattern-discovery methods in KDD. In brief, an association rule is an expression X=>Y, where X and Y are sets of items. The meaning of such rules is quite intuitive: Given a database D of transactions- where each transaction T€D is a set of items X=>Y expresses that whenever a transaction T contains X then T probably contains Y also. The probability or rule confidence is defined as the percentage of transactions containing Y in addition to X with regard to the overall number of transactions containing X. That is, the rule confidence can be understood as the conditional probability p(Y€T | X€T). The idea of mining association rules originates from the analysis of market basket data where rules like “A customer who buys products x1 and x2 will also buy product y with probability c%” are found. Their direct applicability to business problems together with their inherent understand-ability – even for non data mining experts – made association rules a popular mining method. Moreover it became clear that association rules are not restricted to dependency analysis in the context of retail applications, but are successfully applicable to a wide range of business problems.

**1.2 Twister**

Many parallel applications are only required to do Map and Reduce once, such as WordCount[12]. However, some other applications are inevitable to be in an iterative pattern such as Kmeans [10][13] and PageRank [8]. Their parallel algorithms require the program to do Map and Reduce in iterations in order to get the final result.

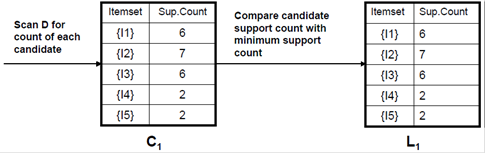
The basic idea of Twister is to let MapReduce jobs only be configured once, then let it run in one turn or several turns according to the client’s request. If there is only one turn execution, it is exactly the same as noniterative MapReduce. The result is produced from Reduce method directly. For iterative MapReduce, the output from “Reduce” is collected by “Combine” method at the end of each iteration. A client will send intermediate results back to compute nodes as new input of KeyValue pairs in next iteration of MapReduce tasks. Another important characteristic of many iterative algorithms is that some sets of input data are kept static between iterations. In Twister, these static data are allowed to be configured with partition file, loaded into Map or Reduce tasks, and then being reused through iterations. This mechanism significantly improves the performance of Twister on iterative MapReduce computing and makes it different from those methods, which mimic iterative MapReduce by simply re-executing MapReduce tasks without caching and reusing data and job configuration. In addition, because the data cached inside of Map and Reduce tasks is static, Twister still keeps “side-effect-free” nature [2]. In this workflow, Twister also provides fault tolerance solution for iterative MapReduce programming model. Twister can save the execution state between iterations. If Twister detects faults in execution, it can be rolled back few iterations and restart computing.

**2. Apriori algorithm in a nutshell**

The Apriori Algorithm is an influential algorithm for mining frequent itemsets for boolean association rules.

**Key Concepts**:

* Frequent Itemsets: The sets of item which has minimum support (denoted by Lifor ith-Itemset).
* Apriori Property: Any subset of frequent itemset must be frequent.
* Join Operation: To find Lk, a set of candidate k-itemsets is generated by joining Lk-1with itself.

****Fig 2 Generating 1-itemset frequent pattern

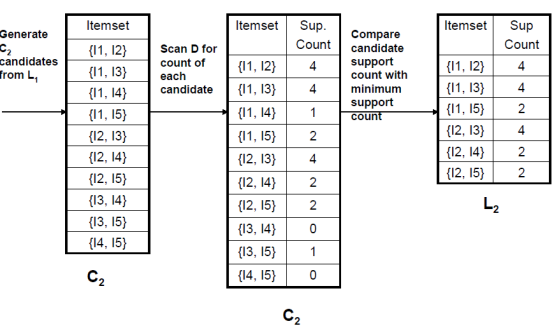
****

Fig 3 Generating 2-itemset frequent pattern

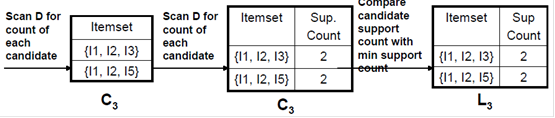
****

Fig 4 Generating 3-itemset frequent pattern

**3. Twister** **Architecture**

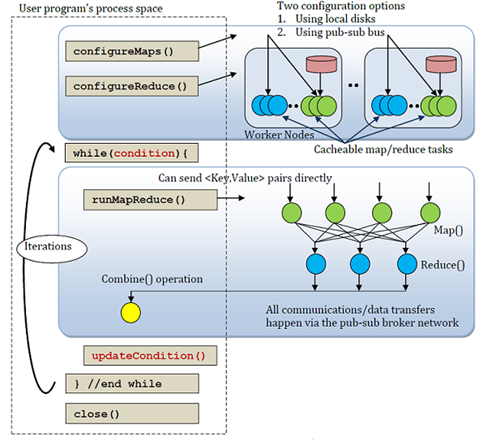


Fig 5 Twister Architecture

Twister has several components. Client side is to drive MapReduce jobs. Daemons and workers which live on compute nodes manage MapReduce tasks. Connection between components are based on SSH and messaging

software. To drive MapReduce jobs, firstly client needs to configure the job. It configures MapReduce methods to the job, prepares KeyValue pairs and configures static data to MapReduce tasks through partition file if required. Once the job is configured, client can run the MapReduce job and monitor it to completion. Between the iterations, it receives the results collected by Combine method. When the job is done, it terminates the job.

Messages including control messages and KeyValue pair data are transmitted through a network of message brokers with publish/subscribe mechanism. With a set of predefined interfaces, Twister can be assembled with different messaging software. Currently Twister supports two kinds of software. One is NaradaBrokering [16], another is ActiveMQ [15]. In our implementation we have used ActiveMQ as messaging middleware.

Daemons operate on compute nodes. They load Map Class and Reduce Class and start them as Map and Reduce workers which can be also called Mapper and Reducer. In initialization, Map and Reduce workers load static data from local disk according to records in partition file and cache the data into memory, and then they execute Map or Reduce function defined by users. Twister uses static scheduling for workers in order to let the local data cache be beneficial to computing [2]. So this is a hybrid computing model. Inside one node, workers are threads and managed by daemon processes. Between nodes, daemons communicate with the client through messages.

Twister uses scripts to operate static input data and some output data on local disks in order to simulate some characteristics of distributed file systems. In scripts, Twister uses “scp” command to distribute static data to compute nodes and create partition file by invoking Java classes. For data which are output to the local disks, Twister uses scripts to collect data from all compute nodes to a node specified by the user.

**4. Implementation**

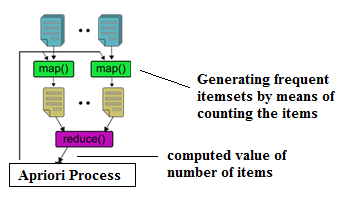


Fig 6 Implementation of the algorithm in twister

The implementation starts off with candidate generation from the input file containing sets of items and transactions. The generated candidates are broadcasted to the mappers who receive the candidates and by making use of candidates as the key and reading the value of the candidates from its corresponding partition file, it maps the count of the candidates to the candidate there by returning a hash map containing candidate as the key and its count local to the map. The hash map returned by the mappers is then sent to the reducers which sum up the count of the candidates in each map locally. This is then sent to a combiner that combines the result returned by each of the reducers into a single result. So the combiner returns a hash map containing candidates as the key and its corresponding total count as value. This is used as a parameter for frequent item-set generation. The result returned after frequent item-set generation is used for candidate generation in the next iteration. This apriori process goes on until the candidates are no more.

**5. Inputs**

Our implementation of Apriori algorithm on twister takes two files as input. Two inputs are explained below.

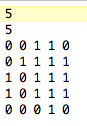
Configuration File:

One is a trivial configuration file, which tells the number of items and transactions and the minimum support count percentage.

Partition File:

Partition file is generated with the help of scripts in the twister package. Partition file is nothing but a pointer to the actual content of data. Twister makes use of this partition file to identify the real split data. Each split data has its own number of items and transactions represented in the first two lines of the start of the file.

Fig 7 Sample input file

****

One of the split data from our project looks like the figure 7. Fig 7 is a tiny sample of our data, just to show the structure of the input. First line represents the number of transactions in current file and the second line represents the number of items in current file. The following five lines is the matrix of items and transactions. Each element in the matrix represents an item and each row represents a transactions. A ‘1’ element represents that the placeholder item is present in that transaction. A zero element represents that the placeholder item is absent in current line of transaction.

In other words, even though there is only first two elements present the transactions and all the other elements are absent in a given transaction file of 25 elements, the trailing 23 placeholders must be filled by zeros.

**6. Challenges**

Programming in any parallel framework is challenging task. We must decompose our problem, think in parallel manner, identify the data dependencies etc. Thankfully Twister like MapReduce framework makes it easier to solve our problem. Some of the challenges we faced in implementing Apriori algorithm on Twister are explained below.

Choosing a right data structure to broadcast the candidates was a challenge. We had a string vector of candidates pruned in each iteration. This Vector of Strings needs to be passed as intermediate candidate as input to all mappers of next iteration. Twister has a handful of such data structures like DoubleArray, StringValue, StringKey etc. Twister can understand only those data structures. Why this is because, during the beginning of each map task, Twister serializes this data into bytes and at the receiver end of mapper this has been de-serialized to corresponding data structure. There are appropriate getters and setters to operate on those data.

We thought of creating one more data structure for our implementation called VectorValue. But we faced problems in serializing it into byte array. Finally our problem was solved by one of the Twister provided data structure itself. That is StringValue. What we had to actually broadcase is, String Vector. So, we appended the individual Strings of the vector into one String and broadcasted the candidates to all mappers.

There was also few experiments conducted on broadcast APIs[14] like

* runMapReduce()
* runMapReduce(List<KeyValuePair>)

We finally chose

runMapReduceBCast(StringValue)

because runMapReduce() method does not takes any argument but we need to broadcast the intermediate candidates to all mappers. Saving the intermediate candidates into a file on each iteration would not be a efficient solution. So that method is not useful.

runMapReduce(List<KeyValuePair>):

This method takes argument but that scatters the given list of KeyValuePair equally among the number of mappers. But our intermediate candidates needs to be common across all the mappers, else we would loose some of our comparisons against the main transaction data.

runMapReduceBCast(StringValue):

This is the suitable API for our implementation. This method sends the given StringValue data to all the mappers. This solves our problem. We pass the intermediate candidates to this runMapReduceBCast method on each iteration of our frequent itemset calculation.

**7. Results**

We compared the execution time against different number of transactions. The execution time increased gradually upon number of transactions. Figure 8 shows this observation.

Fig 8 Time vs. Transactions

The below observation (Fig 9) shows that execution time increases rapidly from less than ten seconds up to 200 seconds when we increase the number Itemsets. This experiment was conducted for standard number of 10000 transactions and five mappers. This increase in execution time shows that when we increase the number of Itemsets its more work for the mappers. The size of intermediate candidates generated increases by a large amount when we increase the Itemsets. Since we broadcast these intermediate candidates to all mappers on all iterations, which is more work for mappers.

Fig 9 Time vs. Item sets

The green line in Fig 9 is the time taken by 20 mappers to do work on the same number of items and transactions. That is less than what mappers took. Even though it is more work for 20 mappers to do parse the wider intermediate candidates, we gain some time because of the increased number of mappers.

**8. Implementation Timeline**

|  |  |
| --- | --- |
| Time | Task |
| Week 11/07/2011 | *Discuss the algorithm and design the*  *coding methodology sequentially* |
| Week 11/14/2011 | *Complete coding the algorithm*  *sequentially* |
| Week 11/21/2011 | *Complete coding the algorithm*  *sequentially* |
| Week 11/28/2011 | *Discuss the design an*  *implementation in twister* |
| Project Review | *Discuss the design with partial*  *implementation in twister* |
| Week 12/05/2011 | *Complete the implementation in*  *twister* |
| Week 12/12/2011 | *Do validation* |
| Project Review | *Presentation* |

**9. Validation**

The performance of the application in Twister framework is validated by determining speed-up while running the application in twister. This is done by drawing a speed-up chart between the input data-set and the speed-up. The speed-up is calculated as:

Sp = T1/Tp

Where:

* *p* is the number of [processors](http://en.wikipedia.org/wiki/Central_processing_unit)
* *T*1 is the execution time of the sequential [algorithm](http://en.wikipedia.org/wiki/Algorithm)
* *Tp* is the execution time of the [parallel algorithm](http://en.wikipedia.org/wiki/Parallel_algorithm) with *p* [processors](http://en.wikipedia.org/wiki/Central_processing_unit)

**10. Future Work**

In our implementation we passed the input data partition file to the Twister, this data is static and cached in the memory for the all the iterations. When this static data size increases, we can consider other NoSQL data formats such as HBase. It would be interesting to test the performance of this algorithm when the data is in some other storage like this. We can increase the number of nodes, mappers, Itemsets and observe the execution time.

**11. References**

[1] J. Han, J. Pei and Y. Yin, “Mining frequent patterns without candidate generation,” in ACM SIGMOD Int. Conf. on Management of Data,2000

[2] S.K. Tanbeer, C.F. Ahmed, B-S Jeong and Y-K Lee, “Efficient single-pass frequent pattern mining using a prefix-tree,” Information Sciences,2009

[3] G. Grahne and J. Zhu, “Fast algorithms for frequent itemset mining using FP-Trees,” IEEE Trans. On Knowledge and Data Engineering, 2005

[4] R. Agrawal, T. Imielinski and A.N. Swami, “Mining association rules between sets of items in large databases,” in ACM SIGMOD Int. Conf. on Management of Data, 1993

[5][http://labs.google.com/papers/mapreduce-osdi04.pdf](http://labs.google.com/papers/mapreduce-osdi04.pdf" \t "_blank)

[6][http://developer.yahoo.com/hadoop/tutorial/module3.html](http://developer.yahoo.com/hadoop/tutorial/module3.html" \t "_blank)

[7] Hadoop official website [http://hadoop.apache.org/](http://hadoop.apache.org/common/docs/current/index.html" \t "_blank)

[8] Page Rank

http://en.wikipedia.org/wiki/PageRank

[9]Twister Official website:

<http://www.iterativemapreduce.org/>

[10] Kmeans clustering:

http://projecteuclid.org/DPubS?service=UI&version=1.0&verb=Display&handle=euclid.bsmsp/1200512992

[11]Apriori Algorithm:

<http://webdocs.cs.ualberta.ca/~zaiane/courses/cmput499/slides/Lect10/sld054.htm>

[12] Word count:

<http://www.iterativemapreduce.org/samples.html#Word_Count>

[13] Kmeans clustering example:

<http://www.iterativemapreduce.org/samples.html#Kmeans_Clustering>

[14] Twister API:

<http://www.iterativemapreduce.org/docs/index.html>

[15] Apache ActiveMQ

<http://activemq.apache.org/>

[16] Narada Brokering

http://www.naradabrokering.org/