Performance Evaluation of Apriori Algorithm on a Hadoop Cluster

JÁNOS ILLÉS
Department of Automation and Applied Informatics
Budapest University of Technology and Economics
Magyar tudósok krt. 2. (building Q), 1117 Budapest
HUNGARY
janos.illes@aut.bme.hu

ISTVÁN VAJK
Department of Automation and Applied Informatics
MTA-BME Control Research Group
Budapest University of Technology and Economics
Magyar tudósok krt. 2. (building Q), 1117 Budapest
HUNGARY
vajk@aut.bme.hu

Abstract: Frequent Itemset Mining is a well-known concept in data sciences. If we feed frequent itemset miner algorithms with large datasets they become resource hungry fast as their search space explodes. This problem is even more apparent when we try to use them on Big Data. Recent advances in parallel programming provides good solutions to deal with large datasets but they present their own problems when we try to modify existing data mining algorithms for the new paradigms. The Apriori-algorithm is a classic solution for mining frequent item-sets. In this paper, we provide a parallel implementation of the Apriori algorithm for the Hadoop platform. We introduce a method to measure the performance of the distributed algorithm. In our experimental results we find choke points in the algorithm and provide resolutions.

Key–Words: Hadoop, MapReduce, Apriori-algorithm, Frequent itemset mining, Cloud computing

1 Introduction

Intelligent analysis of large databases and finding interesting relations in them is an important field of knowledge discovery and data-mining.

Finding frequent item-sets (Frequent Itemset Mining, FIM) is an essential part of exploring interesting connections and analyzing data. FIM tries to find information based on how frequent the interesting events occur in the database. These occurrences can be various things, all based on what kind of data the underlying database contains. In general, the source of this information can vary from items in real or online shopping carts to databases of user generated metadata, like tags or simple log files generated by web-servers in large quantities. When we decide that an itemset is frequent or not depends on a user-submitted minimal threshold. There are various algorithms [1, 2, 3] for mining frequent items and itemsets in databases, but those tend to fall short if we try to feed them with really large volumes of (big) data.

Mining really large databases can be problematic, especially if the data does not fit in the available memory of one computer. This problem can be solved by using algorithms that read the input database more than once and count/store interesting parameters instead of keeping the database in the memory.

Recent developments both in the academic and commercial sectors resulted in increased database sizes. Doing non-trivial calculations on these databases becomes increasingly difficult once it grows out of the available memory of a single machine.

Distributed systems are readily accessible more than ever. It comes natural that any computer bought in the recent years contains more than one CPU core. Even mobile phones are starting to integrate multiple cores [4]. To achieve optimal performance it become necessary to use parallel programming techniques.

In theory the improvement time is linearly increasing every time we add a new machine, but this is not the case in practical scenarios. Most of the existing algorithms has to be changed or fully re-implemented to work in a distributed environment and even after adapted to work in parallel, the speed we gain is lower than expected in most of the cases.

In this paper we will show an implementation and analysis of the classical Apriori algorithm [5]. Our version was implemented to run on the Hadoop distributed computing framework.

Applying the original Apriori-algorithm to huge databases can be problematic. The algorithm's search space increases heavily and even if the database itself can fit in the memory, the incremental steps of the algorithm can generate more and more data in each iteration.

MapReduce is distributed computation paradigm, invented by Google. Google has their own proprietary implementation which they use internally. The Apache foundation supported a free and open source implementation of the paradigm called the Hadoop

ISBN: 978-960-474-374-2

project [6]. The Hadoop projects is collection of different software components. The number of components grew steadily over the years. The two most important parts are the distributed file system called the Hadoop Distributed File System (HDFS) and the software package that implements the MapReduce paradigm called Hadoop framework. All the software in the Hadoop project is implemented on the Java virtual machine.

Using MapReduce paradigm to mine frequent item-sets means we can harness the available computing power of large clusters and thanks to the zero-communication nature of the MapReduce paradigm we can try to achieve the theoretical limits of speedup.

The MapReduce paradigm and the Hadoop framework is recognized for its high throughput, great scalability and fault tolerance. These design goals of the framework means that the development focus was more about to ensure these things than create the fastest possible computation environment. This could result in relatively poor performance when running small jobs in a Hadoop cluster. There is no distinct difference between short and long jobs but the term 'short job' is usually used for jobs running under 400 seconds [7] while the long jobs can take up hours to run. Today's cloud providers, like Amazon EC2, introduced pay-by-time based pricing models. Understanding which parts of the Hadoop jobs use up the time and optimising it can result in savings in the budget.

2 Preliminaries

First we elaborate on some basic concepts of association rules using the formalism presented in [1]. Let $I = \{i_1, i_2, \dots i_m\}$ be a set of literals, called items. Let $D = \{t_1, t_2, \dots t_n\}$ be a set of transactions, where each transaction t is a set of items such that $t \subseteq I$. The itemset X has support s in the transaction set D if $s\%$ of transactions contains X, here we denote $s = support(X)$. An association rule is an implication in the form of $X \rightarrow Y$, where $X, Y \subseteq I$ and $X \cap Y = \emptyset$. Each rule has two measures of value, support and confidence. The support of the rule $X \rightarrow Y$ is support($X \cup Y$). The confidence c of the rule $X \rightarrow Y$ in the transaction set D means $c\%$ of transactions in D that contain X also contain Y, which can be written in $S(X \cup Y)/S(X)$ form. The problem of mining association rules is to find all the rules that satisfy a user specified minimum support and minimum confidence. If $support(X)$ is larger than a user defined minimum support (denoted here min_sup) then the itemset X is called frequent itemset.

The association rule mining (ARM) can be decomposed into two subproblems. The first is finding all of the frequent itemsets and the second is generating rules from these large itemsets. Most of the ARM algorithms only differ from each other in frequent itemset mining step as the second sub problem is much easier than the first one. The Apriori algorithm [5] is an iterative algorithm that finds the frequent itemsets in a database. It starts by counting the frequent items and then it combines the frequent items to generate the candidate 2-itemsets. It counts the occurrences of the candidates by doing a full read on the database, and generated the next candidate itemsets which length are longer by one item. Before counting the support of the candidate itemsets the algorithm prunes the candidates that contain non-frequent subsets. The algorithm continues until it no longer can generate a new candidate itemset.

Parallel algorithms for mining frequent item-set are quite common. Distributed algorithms were proposed almost as soon as non-distributed versions were introduced in [8, 9].

MapReduce was introduced by Google employees [10]. MapReduce is a programming model for parallel computations. Google's own implementation is also called MapReduce and it not available to the public, but the underlying ideas were published. Apache Hadoop is free and open source implementation of the MapReduce paradigm. It is a Java language based implementation widely used by both industry and academic research.

The MapReduce paradigm was designed to work on really large datasets, to solve "embarrassingly parallel" problems. A parallel computing problem is called embarrassingly parallel if the effort needed to separate the solving process into independent, parallel tasks is negligible.

The main idea behind MapReduce is to split the problem the computation into two subproblems: a *map* and a *reduce* phase. First, the input data get sliced into smaller chunks and every chunk is given to a machine that executes a mapper. The mapper processes that chunk and provides key value pairs as the output. Than mapreduce framework collects these pairs, sorts them based on the key and passes them to reducers. A reducer receive a key and a list of values that belongs that key and provides the results. The process can be seen on Figure 1. The strength of the MapReduce paradigm is that it does not allow any communication between the map nodes during the map phase and no communication between the reduce nodes during the reduce phase. The data transfer between the map and the reduce phases are automatic and the programmer does not have to explicitly do anything for it to happen.

This approach resolves or avoid the typical falla-

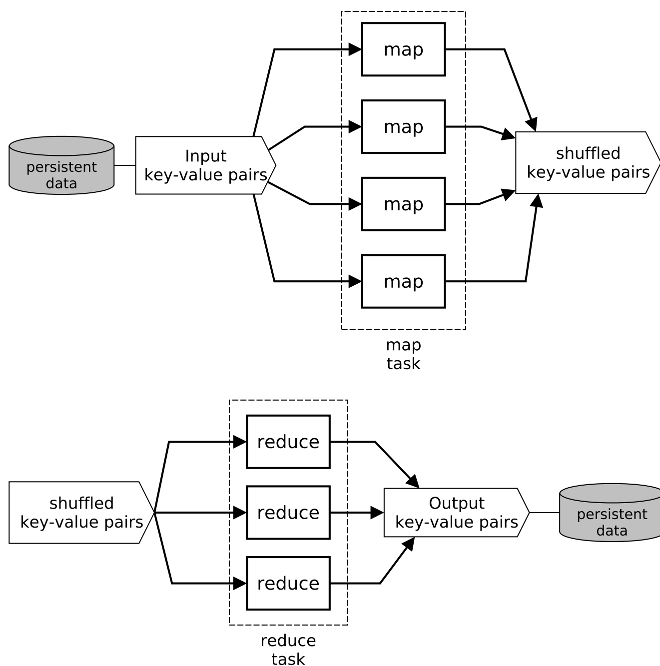

Figure 1: Hadoop mapreduce execution overview

cies of distributed computing. It hides the problem of data distribution and workload balance, the programmer can focus on the algorithm. There are some limitations of course, the mapreduce paradigm works only on key-value pairs, input and output data has to be key-value pairs, as well as the intermediate data that moves between the map and reduce phases.

Some data processing tasks nicely fit to this paradigm, and the conversion of the sequential algorithm is straightforward. Translating other problems and algorithms to the mapreduce paradigm may raise challenges or simply prove unrealizable. The challenges of porting an algorithm and in using a mapreduce framework include the separation the algorithm into two, non-communicating map and reduce phase. Another issue is that a paradigm consists of two steps, the map and the reduce phase and the computation only provides a final result after the final computer in the reduce phase finishes its job. Porting a recursive or cyclical algorithm means that we have to restart the map-reduce loop and feed the output of the previous cycle as the input for the next many times before reaching the desired result.

3 Related work

Data mining and frequent item-set mining is a well researched field, various, well-established algorithms available for such tasks. The distributed versions of such algorithms are almost as old as the originals, but they tend to fail on the scale of Big Data.

The iterative mechanics of the Apriori algorithm does not fit well with the MapReduce paradigm because of the high overhead introduced by the framework itself at the start and the stop of every iteration. Li et al. proposes [11] a Single Pass Counting (SPC) based method for distributed calculation of the Apriori algorithm. The main idea behind the SPC algorithm is that it uses a MapReduce phase for counting the items and generating candidate itemsets. Another proposed algorithm is the Fixed Passes Combined-Counting (FPC) start to generate candidates with n different lengths after p phases and counts the frequiencies in one database scan, where n and p are given as paramters. The last algorithm in their paper is called the Dynamic Passes Counting (DPC) which is similar to FPC but n and p is determined dinamycally at each phase by the number of generated candidates. They also proposed the PApriori algorithm in [12] which is very similar to SPC.

Another well researched single-machine algorithm for frequent itemset mining is the FP-Growth algorithm [3] which uses a datastructure called FP-Tree to count the frequent items. This datastructure compresses the input database in most of the practical inputs. A mapreduce based parallel version of this algorithm is called Parallel FP-Growth (PFP) was proposed in [8]. The PFP algorithm groups the items and distributes their conditional databases to the mappers. The mappers independently builds and FP-tree corresponding to their own slice of input data. PFP efficiency is not optimal, because some of the nodes has to read almost the whole database into the node's memory which is undesired or outright impossible in many use-cases of Big Data scenarios.

The PARMA algorithm proposed in [13] provides great improvements to the runtime of finding association rules. PARMA achieves this by utilizing probabilistic results, it only approximates the answers. Another statistical approach was presented in [14]. This solution uses clustering to create groups of transactions and chooses candidate sets from the representative itemsets in the clusters.

The *count distribution* (CD) algorithm [15] is a fundamental distributed association rule algorithm. The basic idea of this algorithm is that each of the nodes keeps large itemsets and counters of candidates locally, which are related to the whole database. These counters are maintained in accordance with the local dataset and incoming counter values. The nodes locally execute the Apriori algorithm and after reading through the local dataset they broadcast own counters to the other nodes. Each of the nodes can generate new candidates on the basis of the global counter values. Listing 1 shows the pseudo code of the CD algorithm.

Drawback of this approach is that the itemset counting phases are always synchronized to the slowest node.

```
D_local = receiveDataset()
C_local_1 = countItems(D_local)
broadcast(C_local_1)
L1 = gatherCounters()
C_local_2 = generateCandidate(L1)
i = 2
while Ci != 0 do
    for all t IN D_local do
        incrementCounter(C_local_i, t)
    broadcast(C_local_1)
    Li = gatherCounters()
    i++
    Cib = generateCandidate(L_i-1)
return merged(Li)
```

Listing 1: Count distribution algorithm pesudo code

4 Frequent itemset mining on Hadoop

There is no well established frequent-itemset miner algorithm for the Hadoop framework. The Apacahe Mahaout software collection contains an algorithm that can be used to mine frequent itemsets, but as showed in [16] the the implemented algorithms does not perform well on real world data, on some inputs they did not even provide any meaningful result.

In this paper we will analyze the performance of the algorithm we proposed for frequent item-set mining in our previous paper [17] which is using a similar algorithm to Li et al.'s SPC algorithm.

MapReduce paradigm requires key-value pairs for input and output, so first the dataset has to be converted to this format. Transactional databases can be converted to key value pairs where the key is the transaction ID and value is the set of items from the transaction. The algorithms also have to use key-value pairs to represent their intermediate state. This internal state is passed to the reduce machines at the end of the map phase.

Large part of the challenges of converting an algorithm to use the Map Reduce paradigm is to find the right place to separate the internal data and find a good key-value representation for the data passing between the map and the reduce phase.

4.1 Experimental Setup

For the presented measurements we used the following tools. We used a small Hadoop cluster consisting of four computing nodes. All of the computing machines were running in a virtualized environment on the same host hardware. All virtual machines were allocating the same amount of resources in the virtualized environment. The available memory was 2 Gigabytes per machine. We were using version 1.1.2 of the Hadoop software framework. All of the machines were running the Ubuntu distribution of the Linux operating system with the latest stable Java runtime installed.

We used various input datasets for the measurements. A synthetic dataset generator tool was published with [5]. Most measurements were using the data generated by this tool. We were using other datasets used for research in the frequent itemset mining areas were obtained from [18].

We also conducted measurements on our own dataset. This database was based on real world data acquired from web-server access logs of different high-traffic websites.

An access log by itself is not a transactional database that can be used for frequent itemset mining. We had to convert the logs into a format that is meaningful in a frequent itemset mining scenario. Each visited website was considered as an item and given a unique identifier. Websites visited by the same user within a period of time are considered to be in the same itemset. The time period chosen is arbitrary, in our case it was chosen to be 60 seconds.

The file format of the transactional databases slightly varies between the different data-generators and databases. As a step before loading the data into the distributed filesystem we convert the them into an intermediate file format. In the intermediate file format every transaction is represented by one line. Line base separation is handled well by the Hadoop framework. The lines start with a transaction identifier and followed by the list of items. If there is a meaningful lexicographic ordering available on the items then we store them in ascending order.

The algorithm implementation consists of multiple mapreduce jobs and it generates an output in every phase. These output files are used by the next job, except the last one. The output of the last job is considered as the output of the whole computation. We do not use this output for further analysis, currently we are only interested in the running speed of this implementation. Because of the size of the output files can grow significantly, the output files of the previous runs gets discarded every time a new job is started.

4.2 Mapper setup

The first step of any MapReduce job is the map step. Before the start of this step the Hadoop framework will split the D input database into smaller D_n chunks. The size of these split parts depends on the framework configuration and how the data is dis-

ISBN: 978-960-474-374-2

tributed across the on the file-systems of the machines in the cluster. Computation and data storage takes place on the same computers. The frameworks tries to minimize the use of the network and will try to feed local data to the computation. Each chunk is provided as input for a mapper instance. It is possible to create both more and less chunks than the available mapper nodes thus sometimes database chunks will wait for nodes to free up.

The purpose of the very first map task is to count all the items. This is implemented in a really straightforward way. The input data is given to the mapper line by line then the line is split into items and the output key-value pair consists of the item and the value 1. This is the local frequency of the item. The pseudo code of the first map task is on Listing 2.

```
for all transaction in DB_chunk do
    for every item in transaction
        ouput key=item value=1
    end for
end for
```

Listing 2: Pseudo code for the first map task

In the following phases the mapper uses a different implementation. Starting from the second phase the mapper get a chunk of the original dataset and reads another input database that was generated by the previous reduce task. This database contains all of the candidate itemsets. In the nth phase this databases contains the list of the candidate itemsets of length n. The goal of the mapper is to count the frequency of each candidate. To achieve this, the first step of mapper is to read all the available candidates to the memory and then read process the database chunk and count the occurrence of each candidate. The following pseudo code in Listing 3 represents the algorithm of the map task.

```
Candidates = read_from_HDFS()
for all t in DB_chunk do
    incerement_counter(Candidates, t)
end for
for all c in Candidates
    output key=candidate value=candidate count
endfor
```

Listing 3: Pseudo code for the general map task

4.3 Reducer Setup

The reducer task get its input key-value pairs from the output of the previous map task. The pairs are ordered and there is a guarantee that if a reduce task receives a key it will also receive all values with the same key. The ordering and moving of the intermediate key-value pairs is done automatically by the framework and it is called the *shuffle* step.

In the our current implementation of the Apriori algorithm we limit the number of the reducer nodes to one. To generate all the candidate itemsets we have to use all large itemsets. If we do not limit the number of available reducer tasks then we can not get a global state where we combine all large itemsets into larger candidate itemsets. The reduce task always does the same with every key-value pair, presented on Listing 4.

```
itemset = input key
counts = input value
large_itemset = empty array
if sum(counts) > minimum_support
   largeitemsets += itemset
end if
```

Listing 4: Pseudo code of the reduce task, first phase

After summing up all the itemsets the reduce task creates the candidate database. This is then written to the DistributedCache. The pseudo code is in Listing 5.

```
candidates = generate_candidates(large_itemset)
if candidates is empty
    STOP
else
    output candidates
endif
```

Listing 5: Pseudo code of the reduce task, second phase

The reduce task starts with the itemsets of length 1 and generates candidates with length 2. During step k of the algorithm it will start with length n itemsets and generate length $k + 1$ candidate itemsets. If the reduce task cannot generate bigger candidate itemsets it will stop the whole computation.

4.4 Job Orchestration

One map task followed by a reduce task will generate the $k + 1$ sized frequent item candidates called C_{k+1}. To reach every frequent itemset, the algorithm requires multiple passes over the input database. Because Hadoop jobs inherently consists of only one map and one reduce task we had to create a tool that can decide if there is a need for another iteration. At the end of the reduce task, the single reducer can tell how many candidate itemsets were generated. If there are no more candidates the reduce task will signal the tool and the computation is finished.

If there are candidates available value is greater than zero then a new job must be started. The map task in this job requires the generated candidate itemsets as input as well as the split of the input database. However in Hadoop a map task can only get have the

input database split provided by the Hadoop framework.

The candidate itemset database is written into the Hadoop Distributed-cache where every mapper instance can read its contents and process it. The Distributed-cache is a Hadoop specific feature and can be populated with arbitrary files. These files are available to every machine in the cluster. The schematic overview of this setup can be seen in Figure 2.

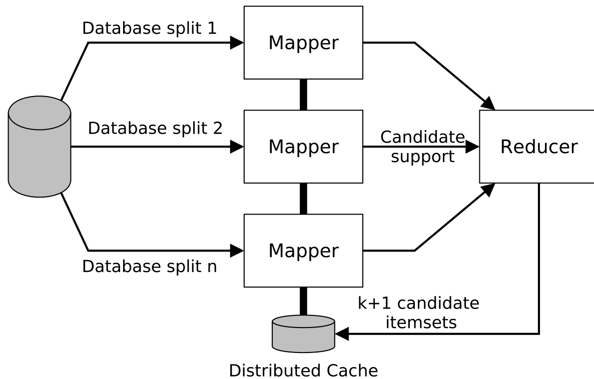

Figure 2: The setup of the Apriori algorithm implementation

4.5 Execution Time Analysis

We run a task on a different input datasets with the same minimum support values. We started on single-machine pseudo cluster then enabled another machine and finally we enabled all four. The results can be seen on Figure 3. As we expected, the computation got faster when more machines were enabled. We can see however that the difference between the elapsed time is narrow when the input dataset was small.

We monitored many aspects of the cluster during the jobs. We are interested in making the frequent itemset mining algorithm faster on the Hadoop framework so the most relevant was how much time it took to run a full computation, but it is important to know how much time is spent on the different parts of the algorithm.

Measuring the spent time of an algorithm that runs on multiple machines in a parallel environment is not a trivial task. The Hadoop framework can tell how long a job was running but our computations are using multiple jobs. We can sum up these jobs of course but the framework does a lot of things in the background like copying code and data between the nodes as well as ordering and shuffling the intermediate job data.

We need more fine grained values so we created a scaffolding framework for our implementation that is responsible for measuring the spent time on each task. This information was collected on each of the computers that were part of the measurements. The resulting

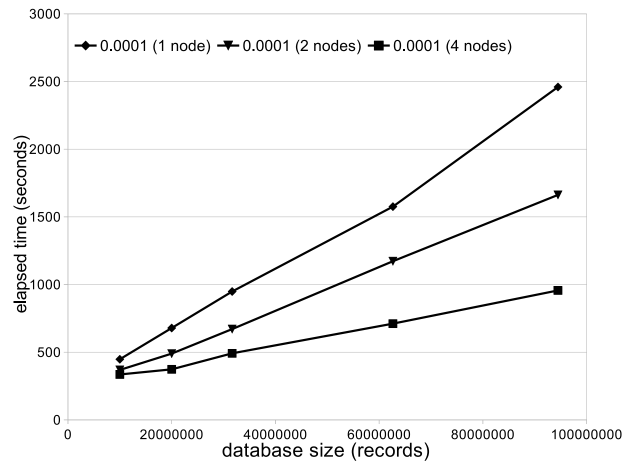

Figure 3: Running time of the Apriori algorithm

data was written to the distributed filesystem. Every machine gets a unique identifier and logs its execution time analysis to a different file named with this id. Appending to the same file on the distributed file system is not recommended due to performance reasons and it was not even possible in the earlier versions of Hadoop. We can extract and accumulate the data as one file easily, thanks to the merging capabilities of the HDFS. At the end of the jobs the files are collected and merged into one. These files are really small and only accessed a few times during a job, so creating and writing to them did not have any measurable impact on the jobs.

We can use the data to measure how much time was spent on each iteration. From this data we can see that the first step of the algorithm, the first MapReduce job that will calculate the frequent pairs will be using significant time compared to the later steps.

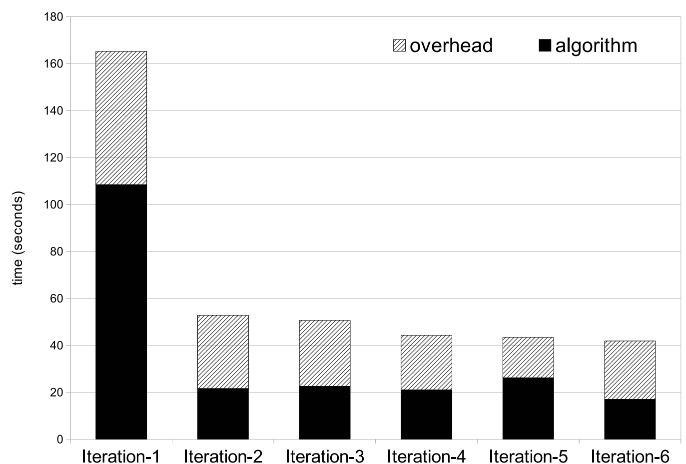

Figure 4: Hadoop mapreduce framework overhead

ISBN: 978-960-474-374-2

On Figure 4 we can see the overhead of the mapreduce steps. What we call overhead is the time spent when the job is running but it not executing any of the code we written. The time used by the framework is mainly consists of the shuffle step and the copying of the data. Since in each iteration we have to distribute the output of the reduce step to every mapper machine the time of this copy is reasonably big. As seen in the figure, in some iterations the time used by the Hadoop framework is larger than the time used up by our job, this is the reason why we see smaller differences with small datasets on Figure 3. The overhead of the framework is more significant if the dataset is small.

4.6 Impact of the Reduce Step

Since the reduce step is always limited to a single machine we assumed that it will produce a bottleneck. We analyzed the data to find out if it is really the case.

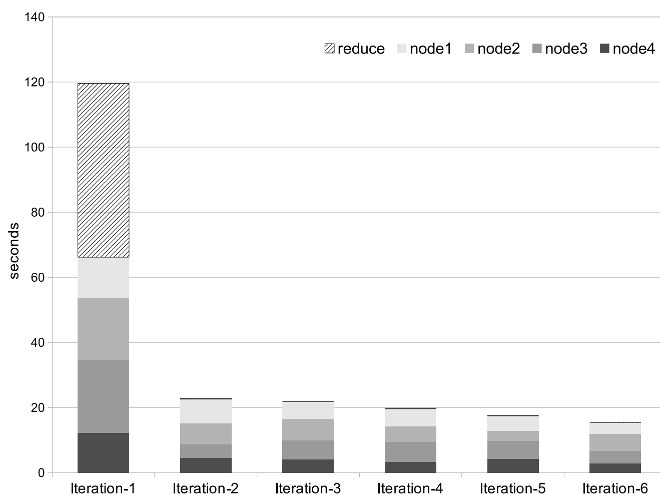

Figure 5: Map and reduce task runtimes

In Figure 5 we can see how much time the reduce task and the map tasks take during one job iteration. In the first job the time of the reduce phase is on par with the all the map phases combined but in the following jobs the reduce task takes negligible time.

This means that the generation of the length 2 candidate itemsets is indeed takes a significant time but in the later iterations, the time needed to generate the $k + 1$ itemsets is not comparable to the time of counting the candidate itemsets in the map task.

Breaking up the reduce-step into smaller tasks would produce benefit only during the first iteration but nothing in the later iterations of the algorithm. A parallel reduce implementation can add to the already significant overhead of the framework by shuffling and distributing the data for the reducers across the network and then collecting the results. This means we should focus more on eliminating steps from the whole algorithm and decreasing the steps needed to get the final result so we can decrease the significant overhead that is introduced by the Hadoop framework as seen on Figure 4.

Figure 5 also shows us that the map runtime varies between different nodes. In extreme cases one mapper node takes up more than twice as much time than the others in that job. The computation can only continue to the next reduce step if all of the mapper nodes are finished. This means that the map task will be finish when the slowest node finishes. While this sounds bad, in reality the Hadoop framework starts to sort and shuffle the output of the nodes as soon as they finished with their chunk of the input data. Still it worth noting that the slicing of the input database impacts the workload balance of the map tasks.

5 Conclusion

We provided a runtime analysis of our approach to frequent item-set mining on Hadoop and pointed out various performance choke points in the algorithm. We found that using a single reducer does not necessarily introduces a bottleneck and by trying to split up the single-machine reduce step to multiple machines we can cause significant framework overhead which is greater than the time we gain by making the reduce task parallel.

In Section 4.6 we showed that the single-machine reducer algorithm works great because it does not introduce unnecessary overhead. There is a possibility that the input data has a large number of long candidate itemsets. In that case, the reduce step will take a significant time with every iteration. A parallel solution, where the reduce step is broken down to several machines the time gained can overcome the introduced overhead, but we can not make a parallel reducer the default option. In the first iteration of the algorithm, it still takes large amount of time to create the next candidate itemsets on a single-machine reduce task. Making only parts of the reduce task parallel is worth exploring.

As seen in section 5, the runtime of the map nodes varies heavily. Further research is needed in this area to determine how to load the input data to the nodes for a more balanced workload. The Hadoop DistributedCache could be replaced with alternative solutions, such as a remote database. These alternative solutions to the DistributedCache are part of a further research issue.

Acknowledgments

This work was partially supported by the European Union and the European Social Fund through project FuturICT.hu (grant no.: TAMOP-4.2.2.C-11/1/KONV-2012-0013) organized by VIKING Zrt. Balatonfüred.

This work was partially supported by the Hungarian Government, managed by the National Development Agency, and financed by the Research and Technology Innovation Fund (grant no.: KMR 12-1-2012-0441).

References:

[1] R. Agrawal, T. Imielinski, and A. N. Swami, "Mining association rules between sets of items in large databases," in *Proc. of the ACM SIGMOD International Conference on Management of Data, (SIGMOD'93)* (P. Buneman and S. Jajodia, eds.), (Washington, D.C.), pp. 207–216, 26–28 1993.

[2] S. Brin, R. Motwani, J. D. Ullman, and S. Tsur, "Dynamic itemset counting and implication rules for market basket data," *SIGMOD Rec.*, vol. 26, pp. 255–264, June 1997.

[3] J. Han, J. Pei, and Y. Yin, "Mining frequent patterns without candidate generation," *SIGMOD Rec.*, vol. 29, pp. 1–12, May 2000.

[4] R. R. Schaller, "Moore's law: Past, present, and future," *IEEE Spectr.*, vol. 34, pp. 52–59, June 1997.

[5] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules in large databases," in *Proc. of the 20th International Conference on Very Large Data Bases (VLDB'94)*, (San Francisco, CA, USA), pp. 487–499, Morgan Kaufmann Publishers Inc., 1994.

[6] The Apache Software Foundation, "Apache Hadoop." `http://hadoop.apache.org`, 2014.

[7] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica, "Improving mapreduce performance in heterogeneous environments," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, (Berkeley, CA, USA), pp. 29–42, USENIX Association, 2008.

[8] H. Li, Y. Wang, D. Zhang, M. Zhang, and E. Y. Chang, "Pfp: parallel fp-growth for query recommendation," in *Proceedings of the 2008 ACM conference on Recommender systems*, RecSys '08, (New York, NY, USA), pp. 107–114, ACM, 2008.

[9] R. Agrawal and J. C. Shafer, "Parallel mining of association rules," *IEEE Transactions on Knowledge and Data Engineering*, vol. 8, pp. 962–969, 1996.

[10] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, (Berkeley, CA, USA), pp. 10–10, USENIX Association, 2004.

[11] M.-Y. Lin, P.-Y. Lee, and S.-C. Hsueh, "Apriori-based frequent itemset mining algorithms on mapreduce," in *Proceedings of the 6th International Conference on Ubiquitous Information Management and Communication*, ICUIMC '12, (New York, NY, USA), pp. 76:1–76:8, ACM, 2012.

[12] N. Li, L. Zeng, Q. He, and Z. Shi, "Parallel implementation of apriori algorithm based on mapreduce.," in *SNPD* (T. Hochin and R. Y. Lee, eds.), pp. 236–241, IEEE Computer Society, 2012.

[13] M. Riondato, J. A. DeBrabant, R. Fonseca, and E. Upfal, "Parma: A parallel randomized algorithm for approximate association rules mining in mapreduce," in *Proceedings of the 21st ACM International Conference on Information and Knowledge Management*, CIKM '12, (New York, NY, USA), pp. 85–94, ACM, 2012.

[14] M. Malek and H. Kadima, "Searching frequent itemsets by clustering data: Towards a parallel approach using mapreduce," in *Web Information Systems Engineering WISE 2011 and 2012 Workshops* (A. Haller, G. Huang, Z. Huang, H.-y. Paik, and Q. Sheng, eds.), vol. 7652 of *Lecture Notes in Computer Science*, pp. 251–258, Springer Berlin Heidelberg, 2013.

[15] R. Agrawal and J. C. Shafer, "Parallel mining of association rules," *IEEE Trans. on Knowl. and Data Eng.*, vol. 8, pp. 962–969, Dec. 1996.

[16] S. Moens, E. Aksehirli, and B. Goethals, "Frequent itemset mining for big data," in *SML: BigData 2013 Workshop on Scalable Machine Learning*, IEEE, 2013.

[17] F. Kovács and J. Illés, "Frequent itemset mining on hadoop," in *Proc. of the IEEE 9th International Conference on Computational Cybernetics (ICCC 2013*.

[18] "Frequent itemset mining dataset repository." `http://fimi.ua.ac.be/data/`, 2004.