AN EFFICIENT LOAD BALANCING DATA PARTITIONING IN FREQUENT ITEMSET MINING ON SPARK RDD FRAMEWORK

[1] Jitha Janardhanan [2] Dr.E.Mary Shyla
[1] M.Phil. Research Scholar, [2] Assistant Professor
[1] Sri Ramakrishna College of Arts and Science for Women, Coimbatore, India

Abstract : Frequent Item mining is the significance of data mining techniques to define patterns from the Big datasets. Frequent Itemset Mining is one of the predictable data mining problems in most of the data mining applications. It comprises very large reckonings and Input/output traffic capacity. Also resources like single processor's memory and CPU are very incomplete, which lowers the functioning of algorithm. In this exploration broadsheet aims to present a EFPGSID (Enhanced Frequent Pattern Growth Skewed intermediate data blocks), a parallel Frequent item mining algorithm based on the Spark RDD (Resilient Distributed Datasets) framework—a specially-considered in-memory parallel computing framework to backing load blanching algorithms and interactive data mining. The outcomes shown that the performance of the new system is effective compared with other FiDoop-DP mining algorithms. As the Investigational results show, Enhanced-FP-Growth with load balancing strategy clearly outperforms FP-Growth and FiDoop-DP. It is faster than FP-Growth and is not expensive like FP-tree.

IndexTerms **- Frequent itemset mining, parallel data mining, data partitioning, Apache Spark model, hadoop cluster**

I. INTRODUCTION

Data mining is a method to recognize and exchange raw data into valuable information, is increasingly being used in a mixture of fields like business intelligence, marketing, scientific discoveries, biotechnology, multimedia and Internet searches. Data mining is an interdisciplinary field merging concepts from machine learning, statistics and natural language processing.

Advances in computing and networking technologies have resulted in many distributed computing environments. The several distributed data sets allow large-scale data-driven knowledge discovery to be used in science, business, and medicine. Data mining in such environments requires a utilization of the available resources. Conventional data mining algorithms are developed with the assumption that data is memory resident, making them unable to cope with the exponentially increasing size of data sets. Therefore, the use of parallel and distributed systems has gained significance.

Generally, parallel data mining algorithms work on tightly coupled custom-made shared memory systems or distributed-memory systems with fast interconnects. The main differences between such algorithms are scale, communication costs; interconnect speed, and data distribution. MapReduce is an emerging programming model to write applications that run on distributed environments. Several implementations such Apache Hadoop are currently used on clusters of tens of thousands of nodes [1]. This thesis focuses on Spark with Hadoop MapReduce design and the implementation of two new data mining techniques relating to enhanced frequent item mining and load balancing in Spark computing environment. This trend to use distributed, complex, heterogeneous computing environments has given rise to a range of new data mining research challenges. This work explores the different methods and trade-offs when designing and implementing distributed data mining algorithms. On the whole, it deliberates data partition/replication and workload dispersion and data formats. Also, this work objects to investigate the hardware utilization when running Spark MapReduce algorithms on the infrastructure. This helps to study the behavior of algorithms on simulated large clusters. This helps rapid optimizing and rapid developing efficient algorithms that use the spark load balancing framework.

Apache Spark is an open source cluster computing environment that supports in-memory distributed datasets enhancing iterative process runs. Spark was developed at the University of California Berkeley, Algorithms Machines and People Lab to build large-scale and low-latency data analytics applications. [2]

Spark is employed in the Scala language. Spark and Scala are tightly assimilated, that makes Scala the proficiency to easily deploy distributed datasets as local objects. Spark was intended for a specific type of jobs in cluster computing that reprocess a working set of data across the parallel operations. As an optimization for these types of jobs, Spark developers introduced the concept of in-memory cluster computing, where it is possible to cache the datasets in memory to reduce their latency of access [2].

Spark established an abstraction named resilient distributed datasets (RDD). Those datasets are read-only object groups that are allocated across the nodes. One can create RDDs by applying operations called transformations, such as map, filter and groupBy, to the data in a stable storage system, such as the Hadoop Distributed File System (HDFS). As an example the following Spark code counts the words in a text file:

```
JavaRDD<String> file = spark.textFile("hdfs://...");
JavaRDD<String> words = file.flatMap(new FlatMapFunction<String, String>()
public Iterable<String> call(String s) { return Arrays.asList(s.split(" "));} });
JavaPairRDD<String, Integer> pairs = words.map(new PairFunction<String, String, Integer>()
public Tuple2<String, Integer> call(String s) { return new Tuple2<String, Integer>(s, 1);} });
JavaPairRDD<String, Integer> counts = pairs.reduceByKey(new Function2<Integer, Integer>()
public Integer call(Integer a, Integer b) { return a + b; } });
counts.saveAsTextFile("hdfs://...");
```

II. RELATED WORK

In [3] authors conferred a data mining has been widely distinguished as a powerful tool to reconnoiter added value from across-the-board databases. One of data mining techniques, generalized association rule mining with taxonomy, is potential to determine more useful knowledge than ordinary flat association rule mining by taking application specific information into account. As discussed the pattern growth mining paradigm based FP-tax algorithm, which make use of a tree structure to compress the database. Two methods to traverse the tree structure are examined: Bottom-Up and Top-Down.

In [4] authors provided a MapReduce is a programming model used for processing and producing large data sets. Users specify a map function that processes a key/value pair to generate a set of intermediate key/value pairs, and a reduce function that bring together all intermediate values associated with the same intermediate key. Many real world chores are expressible in this model, as revealed in the paper. Programs scripted in this functional style are instinctively parallelized and implemented on a large cluster of commodity machines. The run-time system ensure the details of segregating the input data, slating the program's performance across a set of machines, overseeing machine letdowns, and handling the requisite inter-machine communication. This tolerates programmers without any proficiency with parallel and distributed systems to easily exploit the resources of a large distributed system. The execution of MapReduce runs on a large cluster of commodity machines and is highly scalable. A distinctive MapReduce reckoning processes many terabytes of data on thousands of machines.

In [5] authors proposed an inclusive survey for a family of methodologies and mechanisms of large scale data processing mechanisms that have been implemented based on the original idea of the MapReduce framework and are currently gaining a lot of drive in both research and industrial communities. It also conceals a set of introduced systems that have been applied to provide declarative programming interfaces on top of the MapReduce framework. At the same time, authors assessed several outsized data processing systems that bear a resemblance to some of the ideas of the MapReduce framework for different purposes and application scenarios. Lastly, they conversed some of the upcoming research directions for implementing the next generation of MapReduce-like solutions.

In [6] authors provided Many parallelization techniques have been projected to enrich the performance of the Apriori-like frequent itemset mining algorithms, described by both map and reduce functions. MapReduce has shape up and shines in the mining of datasets of terabyte scale or larger in either homogeneous or heterogeneous clusters. Minimizing the scheduling overhead of each map-reduce phase and maximizing the deployment of nodes in each phase are roots to successful MapReduce implementations. In investigation report, authors suggested three algorithms, named SPC, FPC, and DPC, to explore efficient implementations of the Apriori algorithm in the MapReduce framework. DPC showcases in dynamically combining candidates of various lengths and overtakes both the straight-forward algorithm SPC and the fixed passes combined counting algorithm FPC. Wide spread assessment outcomes also show that all the three algorithms scale up linearly with respect to dataset sizes and cluster sizes.

In [7] authors considered a Traditional Association Rules algorithm has computing power scarcity in dealing with enormous datasets. So as to overcome these problems, a distributed association rules algorithm grounded on MapReduce programming model named MR-Apriori is suggested. In investigation report, authors bring together the MapReduce programming model of Hadoop platform and Apriori algorithm of data mining, which recommends the detailed procedure of MR-Apriori algorithm. Speculative and experimental results show MR-Apriori algorithm make a sharp increase in efficiency.

In [8] authors proposed Frequent itemset mining (FIM) plays an vital role in mining associations, correlations and many other important data mining tasks. Regrettably, as the size of dataset gets larger day by day, most of the FIM algorithms in literature become frutile due to either too enormous resource requirement or too much communication cost. In investigation report, authors suggested a balanced parallel FP-Growth algorithm BPFP, established on the PFP algorithm, which parallelizes FP-Growth in the MapReduce approach. BPFP

adds into PFP load balance feature, which enhances parallelization and thereby improves execution. From end to end experimental study, BPFP overtook the PFP which uses some simple grouping strategy.

III. RESEARCH METHODOLOGY

In experimentation report, we have proposed a new Enhanced FP-Growth with load balancing strategy for frequent item mining in large scale data. The proposed workflow accepts the simulation parameters as input which contains the SPARK 1.6.0 with HADOOP 2.6 version where the optimal Enhanced FP-Growth with load balancing strategy is applied to the IBM Quest Data Set. This overall proposed architecture in figure 1 follows a procedure form start to end state.

3.1.Data Cleaning

Data cleaning or data preprocessing is a data mining technique that accomplishes transforming raw data into a reasonable format. The real-world data is frequently partial, conflicting, and/or lacking in confident behaviors or trends, and is expected to contain many errors. Data preprocessing is a recognized method of ascertain such issues. Data preprocessing formulates raw data for further processing.

Preprocessing alters the data into an arrangement that will be more without difficulty and efficiently processed for the point of the user. In the real world, data are forever accompanied by noisy, incomplete or inconsistent problems which would be switched in data cleaning process. Data may be missing because of missing group, replica records or problematic equipment. Missing data can be overlooked, filled in physically or automatically with a inclusive constant. Noisy data refers to data with unsystematic error or variance in a calculated variable. It may be caused during the data collection or transportation step or because of dissimilar position or alternative spellings. To resolve this problem, the binning method is which means categorization data and separation them in equivalent frequently bins and then can smooth bins by means, median or smooth boundaries.

Identify outliers and smooth noisy data

- **Clustering:** Cluster feature values in clusters then detect and remove outliers;
- **Binning:** sort the attribute values and partition them into bins;
- **Regression:** smooth data by using regression functions.

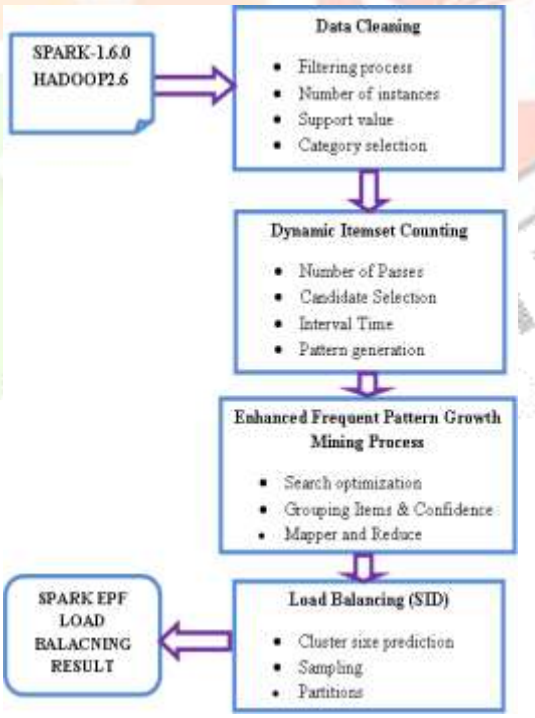

Fig. 1: Proposed Architecture

3.2.Dynamic Itemset Counting

Dynamic Itemset Counting (DIC) is to hustle up the detection of frequent itemsets in a big database. DIC divides the database into a number of partitions marked by initial points. Then, it determines the supports of all itemsets counted so far, dynamically adding new candidate itemsets whenever their subsets are ascertained to be frequent, even if their subsets have not yet been seen at all transactions.

The main dissimilarity between DIC and frequent pattern mining is that whenever a candidate itemset reaches the support during a particular scan, DIC starts producing additional candidate itemsets based on it, without waiting to complete the scan as FP-Growth does. To accomplish the dynamic candidate itemsets generation, DIC employs a prefix-tree where each item counted so far is associated with a node.

DIC deliberated numerous ways of addressing this problem:

- **Virtually randomize the data:** That is, call the file in a random order while creation certain that each pass is in the alike order. This can warrant a high seek cost, predominantly if the data is on tape. In this case, it may be adequate to bound to a new position every few thousand transactions or so.
- **Loosen the support value:** First, start with a support threshold (mean and trial error method) drastically lower than the given one. Then, deliberately boost the threshold to the obligatory level. This way, the algorithm starts fairly conventional and then becomes more positive as more data is collected. However, perhaps more careful control of the slack or a dissimilar dataset would make this a useful technique.
- One thing to note is that if the data is simultaneous with its position in the input file, it may be constructive to notice this and report it. This is possible if a "neighbourhood" counter is kept along with each itemset which process the count of the present interval. At the end of each interval it can be checked for considerable discrepancies with its overall support in the whole data set.

3.3.Enhanced Fp-Growth Mining Process

The EFP-growth uses three Spark MapReduce phases to parallelize FP-Growth.

Step 1: Dividing Database (DB) into succeeding parts (sub-datsets), every part identical size according to the transaction numbers, and accumulating the parts on *N* different computers nodes. Use Parallel Counting algorithm to calculate the support values of all items which is appear in *DB*, and then each sub-datasets will be entered in a particular mapper. The end result is stored in Enhanced Frequent list (EF-List), Algorithm 1 shows the Parallel Counting algorithm, For each item, $a \in T$, the a key-value pair will be the results of the mapper (*key* =*a* , *value*= 1). After all mapper samples have completed, for each key produced by the mappers, the MapReduce framework collects the set of parallel values and feed the reducers with key-value pairs (key, List (key)).

Algorithm 1: Parallel Counting
Procedure: Spark Mapper (key, value = T_i)
For each item a_i in T_i do
Call output (<key = a_i, value ='1');
End for
End Procedure
Procedure: Spark Reducer (key= a_i, value=list(a_i))
C← 0;
For each item '1' in list(a_i) do
C ←C+1;
End for
Call Output (key = *null*, value=a_i+*C*>);
End Procedure

The reducer thus basically outputs (key = *null*, value= key + sum (List (key))). It is simple to observe that key is an item and value is sup (key).
Step 2: Clustering Items: Partitioning whole items (*I*) into groups (*G*) which are positioned in *EF-List*. The concluding obtained list of groups is called Grouplist (*G*-list), where every group is known an Index EF-list (Id). As EF-list and G-list are both little and the time complexity is O(|I|), this step can inclusive on a single computer in few seconds.
Step 3: Enhanced FP-Growth process work process collecting one MapReduce passes, where the map and reduce phases are executed in different significant functions:
Spark Mapper – Each spark mapper samples are placed with a sub-datasets created in Step 1. Ahead of it procedures transactions in the sub-datasets individual by individual, it interprets the *G*-list through step 2. The mapper algorithm results will be one or more key-value pairs, where each key includes an *Id* and its corresponding value respected a generated group-dependent transaction.
Spark Reducer – While all mapper samples have completed their work, the MapReduce framework repeatedly clusters all parallel group-dependent transactions, for every *Id*, into a sub datasets of group-dependent transactions. Each reducer sample is allocated to process one or more group-dependent sub-datasets one by one. For every sub-datasets, the reducer sample constructs a local EFP-tree and EFP-growth its provisional FP-trees recursively. During the recursive process, it may output discovered patterns

Step 4: collecting the results which are produced in Step.3 as last Enhanced frequent itemsets.

3.4.Load Balancing Skewed Intermediate Data Blocks Algorithm

The load balancing skewed intermediate data blocks model (i.e., splitting and combination), for quantifying the sizes of the clusters received by a bucket with considering the effect of data skew, some initial and intermediate objects with their relationships can be formalized as follows. As cluster is the collection of key/value tuples with a same key, the overall clusters can be formalized as a set *C* in:

$$C = \{C_1, C_2, \ldots, C_i, \ldots, C_m\},\ 1 \le i \le m \quad \text{eqn. (1)}$$

where *m* is the number of clusters. C_i is a structure, which can be formalized as $C_i = \{order, SC\}$, where $C_i.order$ records the initial order number of this cluster, and *SC* can be expressed as a separate sequence in:

$$SC = \{SC_1, SC_2, \ldots, SC_i, \ldots, SC_m\},\ 1 \le i \le m \quad \text{eqn. (2)}$$

where SC_i is an integer which denotes the data size of a specific cluster.

And current buckets in the system can be formalized as a set B in Eq. (3):

$$B = \{B_1, B_2, \ldots, B_k, \ldots, B_n\},\ 1 \le k \le n \quad \text{eqn. (3)}$$

where *n* is the number of the buckets. To record the number of key/value tuples in each cluster C_i, this model proposes a set *SC* to simplify this problem, which is shown in Equation (2).

In this proposed framework presents a reservoir sampling method for higher accuracy to estimate the inner structure of large input data in the Spark framework. As a typical random sampling method, with the random data replacement policy in the sampling zone, this algorithm can achieve a much better approximation to the distribution of intermediate data. For sampling is with a small percentage of the input data, this experimentation report prioritizes the execution of sampling job over the normal map tasks in order to achieve the distribution statistics.

IV. SIMULATION RESULTS

The simulation studies work has been evaluated the performance of Enhanced FP-Growth mining with load balancing strategy in-house Spark 1.6.0 Hadoop 2.6 cluster equipped with data nodes. Each node has an Intel I5-6500 series 3.20 GHz 4 core processor, 8GB main memory, and runs on the Windows operating system, on which Java JDK 1.8.0_20 and Spark1.6.0 with inbuild Hadoop 2.6 are installed. The hard disk of NameNode is configured to 100 GB; and the capacity of disks in each Data- Node is 1 TB.

To evaluate the performance of the proposed EFPGSID (Enhanced Frequent Pattern Growth Skewed intermediate data blocks), We generate synthetic datasets using the IBM Quest Market-Basket Synthetic Data Generator [9], which can be flexibly configured to create a wide range of data sets to meet the needs of various test requirements. The parameters' traits of our dataset are recapped in Table 1.

Table1: Input IBM Quest Market-Basket Synthetic Data

Parameters	Average Length	#Items	Avg.Size/Transaction
T10I4D	10	4000	17.5B
T40I10D	40	10000	31.5B
T60I10D	60	10000	43.6B
T85I10D	85	10000	63.7B

In this experiment study, table 2 compare the performance of Running Time of the testing datasets.

$$\text{Running Time} = \frac{Proces\ End\ time}{Process\ Start\ time} * 1000 \quad \text{eqn.(4)}$$

Table 2: Running time comparison repercussion of the number of pivots on EFPGSID, FiDoop-DP and Pfp.

Methods	20	60	100	140	180
Pfp	18	14	20	22	30
FiDoop-DP	14	13	16	18	26
EFPGSID	**12**	**11.5**	**15**	**16**	**24**

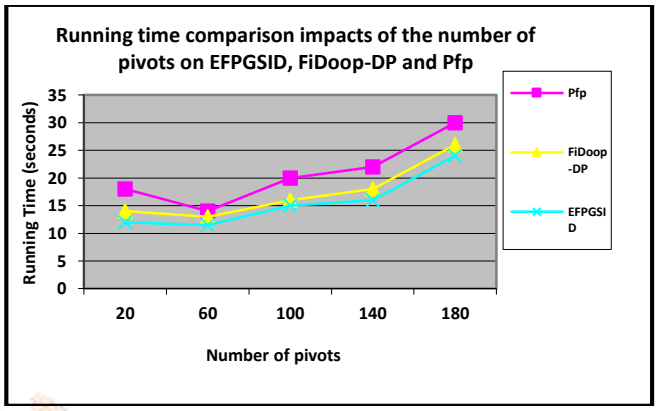

Figure 2: Running time Repercussion of the number of pivots on EFPGSID, FiDoop-DP and Pfp.

In table 3 represents the performance of Mining Cost of the testing datasets.

$$\text{Mining Cost} = \frac{Spark\ passes\ End\ time}{Spark\ passes\ Start\ time} * 1000 \quad \text{eqn.(5)}$$

The research work compared the performance of EFPGSID, FiDoop-DP and Pfp [10] when the number k of pivots varies from 20 to 180. Please note that k in EFPGSID corresponds to the number of groups in FiDoop-DP. The performance measures reveals the running time, shuffling cost, and mining cost of EFPGSID, FiDoop-DP and Pfp processing the 4G 61-block T40I10D dataset on the 8-node cluster.

Table 3: Mining cost comparison repercussion of the number of pivots on EFPGSID, FiDoop-DP and Pfp

Methods	20	60	100	140	180
Pfp	8.8	8.4	9	11.8	14
FiDoop-DP	8.4	8	8.8	10.2	12.4
EFPGSID	**8.2**	**7.6**	**8**	**9.8**	**11.6**

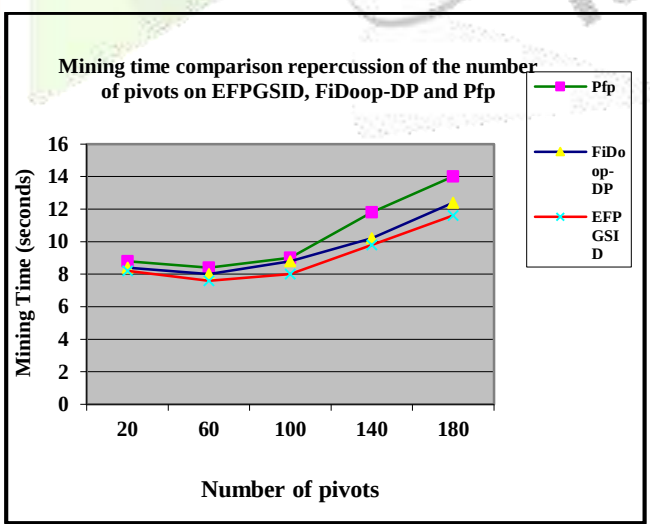

Figure 3: Mining time repercussion of the number of pivots on EFPGSID, FiDoop-DP and Pfp

Shuffle cost: The expected shuffle cost of the parallel clustering approach is a function of the number of reducer's r to receive the data and the amount of data to be shuffled s, which is given by:

$$\text{Shuffling Cost}\ (r, s) = \frac{s \cdot D_r}{r} \cdot \frac{1}{N_s} \quad \text{eqn. (6)}$$

The majority of the shuffling cost is related to shipping data between distinct machines through the network. Whenever possible, spark MapReduce minimizes the cost by assigning reduce tasks to the machines that already have required data in local disks. D_r is the ratio of data actually shipped between distinct machines relative to the total amount of data processed. Thus, the total amount of data be shipped is $s.D_r$ bytes. The data will be received in paralleled by r reducers, each one receiving in average N_s byters per second.

In table 4 represents the performance of Shuffling Cost of the testing datasets.

Table 4: Shuffling cost comparison repercussion of the number of pivots on EFPGSID, FiDoop-DP and Pfp

Methods	20	60	100	140	180
Pfp	60.2	57	63	72.4	74.6
FiDoop-DP	55.6	45.8	55.2	64.2	70
EFPGSID	**54.3**	**42.8**	**52**	**63.4**	**68**

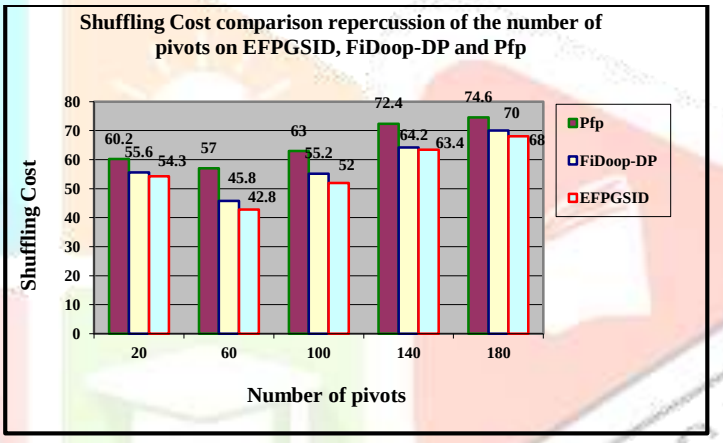

Figure 4: Shuffling Cost repercussion of the number of pivots on EFPGSID, FiDoop-DP and Pfp.

V. CONCLSUION

In this experimentation report, presents an enhanced method of Enhanced Frequent Pattern Growth Skewed intermediate data blocks (EFPGSID), which combines Spark framework and load balancing strategy to solve the optimization problem during Big data processing. In the EFPGSID model, a novel EFP-Growth with load balancing of splitting and combination scheme that achieves cost rate proportionality, while maximizing the total capacity is obtained. The proposed technique that iteratively computes the optimal solution with the help of a big dataset program solves. Our scheme is implemented using Spark 1.6 with Hadoop 2.6 platform under goes MapReduce framework. According to the EFP-Growth strategy it can work with the large transaction database for finding the mining frequent itemsets

REFERENCES

[1] O.O. Malley and A.C. Murthy, ?Winning a 60 Second Dash with a Yellow Elephant Hadoop implementation, March 2009, URL http://sortbenchmark.org/Yahoo2009.pdf.
[2] Spark, an alternative for fast data analytics, http://www.ibm.com/developerworks/library/os-spark/ [04.11.2014]
[3] I. Pramudiono and M. Kitsuregawa, "Fp-tax: Tree structure based generalized association rule mining," in Proc. 9th ACM SIGMOD Workshop Res. Issues Data Mining Knowl. Discovery, 2004, pp. 60–63.
[4] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," ACMCommun, vol. 51, no. 1, pp. 107–113, 2008.

[5] S. Sakr, A. Liu, and A. G. Fayoumi, "The family of mapreduce and large-scale data processing systems," ACM Comput. Surveys, vol. 46, no. 1, p. 11, 2013.
[6] M.-Y. Lin, P.-Y. Lee, and S.-C. Hsueh, "Apriori-based frequent itemset mining algorithms on mapreduce," in Proc. 6th Int. Conf. Ubiquitous Inform. Manag. Commun., 2012, pp. 76:1–76:8.
[7] X. Lin, "Mr-apriori: Association rules algorithm based on mapreduce," in Proc. IEEE 5th Int. Conf. Softw. Eng. Serv. Sci., 2014, pp. 141–144.
[8] L. Zhou, Z. Zhong, J. Chang, J. Li, J. Huang, and S. Feng, "Balanced parallel FP-growth with mapreduce," in Proc. IEEE Youth Conf. Inform. Comput. Telecommun., 2010, pp. 243–246.
[9] L. Cristofor, "ARtool: Association rule mining algorithms and tools," 2006
[10] MapReduce, http://searchcloudcomputing.techtarget.com/definition/MapReduce [04.11.2014]