Simplifying parallel implementation of algorithms on Hadoop with Pig Latin

Conference Paper · April 2015

CITATIONS
5

READS
228

5 authors, including:

Eftim Zdravevski
Ss. Cyril and Methodius University in Skopje
157 PUBLICATIONS **1,437** CITATIONS
SEE PROFILE

Petre Lameski
Ss. Cyril and Methodius University in Skopje
102 PUBLICATIONS **930** CITATIONS
SEE PROFILE

Andrea Kulakov
Ss. Cyril and Methodius University in Skopje Macedonia
85 PUBLICATIONS **763** CITATIONS
SEE PROFILE

Sonja Filiposka
Ss. Cyril and Methodius University in Skopje
139 PUBLICATIONS **745** CITATIONS
SEE PROFILE

Simplifying parallel implementation of algorithms on Hadoop with Pig Latin

Eftim Zdravevski
Faculty of Computer Science and Engineering
Ss.Cyril and Methodius University, Skopje, Macedonia
Email: eftim.zdravevski@finki.ukim.mk

Petre Lameski
Faculty of Computer Science and Engineering
Ss.Cyril and Methodius University, Skopje, Macedonia
Email: petre.lameski@finki.ukim.mk

Andrea Kulakov
Faculty of Computer Science and Engineering
Ss.Cyril and Methodius University, Skopje, Macedonia
Email: andrea.kulakov@finki.ukim.mk

Sonja Filiposka
Faculty of Computer Science and Engineering
Ss.Cyril and Methodius University, Skopje, Macedonia
Email: sonja.filiposka@finki.ukim.mk

Dimitar Trajanov
Faculty of Computer Science and Engineering
Ss.Cyril and Methodius University, Skopje, Macedonia
Email: dimitar.trajanov@finki.ukim.mk

***Abstract*—In this paper we present a general technique for parallelizing regular algorithms with the tools the Hadoop ecosystem offers: MapReduce, HDFS, HBase and Pig. This framework can be applied for parallelizing algorithms for feature selection, clustering, machine learning etc. It consists of several steps: load the datasets in HDFS, apply some transformations if they are needed, store the datasets in HBase, and implement the algorithm in Pig with the help of User Defined Functions.**

***Keywords*—Hadoop, MapReduce, HBase, Pig, parallel algorithms, distributed algorithms**

I. Introduction

In the recent years companies, organizations and governments collect, process and analyze enormous volumes of data. For most of them the data is not only generated from their normal work, rather it a prerequisite for their success. As a result many companies have followed different ideas on how cope with the Big Data challenge. One idea was to scale-up hardware so it has more processing power that can handles the larger volumes of data, and it has proven to work up to a certain point. However, after this point is reached, this idea can not work. That lead to the other idea of distributing the computation and data storage to clusters. Even though this is not so new idea in general, it was not until about ten years that it started to gain popularity. Inspired by Google's approach described in the 2004 MapReduce [1] and 2006 Big Table [2], many other companies and open-source projects followed similar pathways developing different distributed systems. One of the most popular such systems is Apache Hadoop. It is open-source software that contains a set of algorithms for distributed processing, scheduling and storage of large datasets on computer clusters. It is well established framework and Hadoop Wiki [3] lists some of its prominent users like Yahoo, Facebook, Ebay, Adobe etc.

This work was partially financed by the Faculty of Computer Science and Engineering at the Ss.Cyril and Methodius University, Skopje, Macedonia

The MapReduce programming paradigm [1] [4] is essential to the distributed computation and storage that Hadoop achieves. It consists of two phases: map and reduce. The first phase, map, threats the data processing problems as embarrassingly parallel by splitting the data into distinct subsets that can be processed in parallel. The reduce phase is second and final aggregates the output from the map phase and produces the final result. In other words, the map procedure can perform variety of operations like: reading, projecting, filtering and sorting data. The output from this phase is an intermediate result usually comprised of a list of keys and values. These are mandatory for the reduce phase. Hadoop makes sure that the output gets to the reduce procedures in proper order so it can perform some summary or aggregate operation. Even though the MapReduce model is fairly restricted, its simplicity is making it very suitable and efficient for extremely large-scale implementations across thousands of nodes.

Hadoop with its different services schedules, distributes, orchestrates and monitors communications, data transfers, while providing redundancy and fault tolerance. There are many services (i.e. subsystems) in Hadoop that aid accomplishing the previous goals, but three of them are most notable: YARN (MapReduce2), HDFS and HBase [5] [6] [7].

The fundamental idea of YARN (i.e. Yet Another Resource Negotiator) [8] is to take care of resource management and job scheduling/monitoring, by splitting these responsibilities into separate daemons: a global ResourceManager and per-application ApplicationMaster. The ResourceManager is the ultimate authority that arbitrates resources among all the applications in the system. The per-application ApplicationMaster is responsible for negotiating resources from the ResourceManager and working with the NodeManagers to execute and monitor the component tasks. In other words, YARN is responsible for allocating resources to the MapReduce jobs, distributing them to the most appropriate nodes, etc.

Hadoop Distributed File System (HDFS) [9] is a file

system that provides scalable, fault-tolerant, distributed storage system that works closely with MapReduce which was designed to span large clusters of commodity servers. The combined resources of the servers within the cluster can easily grow with the demand. An HDFS cluster is comprised of a NameNode which manages the cluster metadata and DataNodes that store the data. The file content is split into large blocks (typically 128 megabytes), and each block of the file is independently replicated at multiple DataNodes. The blocks are stored on the local file system on the DataNodes.

HBase is an open source, non-relational, distributed database modeled after Google's BigTable. It runs on top of HDFS (Hadoop Distributed Filesystem), providing BigTable-like capabilities for Hadoop [10] [11] [12]. In other words, it provides a fault-tolerant way of storing large quantities of sparse data. HBase is a NoSQL (Not Only SQL) database and is not a direct replacement for a classic Relational SQL databases [13]. Unlike traditional databases where normalization of data and splitting it into related tables is the substance of the design, designing HBase tables takes a different approach which analyzes the usage patterns. Motivations for this approach include simplicity of design, horizontal scaling, and finer control over availability. The data structures used by NoSQL databases, including HBase, differ from those used in relational databases, making some operations faster in NoSQL and others faster in relational databases. The particular suitability of a given NoSQL database depends on the problem it must solve. Tables in HBase can serve as the input and output for MapReduce jobs run in Hadoop. In the parlance of Eric Brewers CAP theorem, HBase is a CP type system (i.e. Consistent and Partition tolerant) [14].

Because of its simplicity the MapReduce programming model has become popular. By some users this model is preferred over the traditional SQL which is a high-level declarative approach. Be that as it may, the extreme simplicity of MapReduce leads to much low-level hacking to deal with the many-step, branching dataflows that arise in practice. Furthermore, users must repeatedly code standard operations such as join by hand. These practices increase development time, introduce bugs, harm readability, and may obstruct optimizations [15]. A group at Yahoo motivated by these repeatable tasks on daily basis, has developed a scripting language called Pig Latin. Pig is a high-level dataflow system that is a compromise between SQL and MapReduce. Pig offers constructs for data manipulation similar to SQL, which can be integrated in an explicit dataflow. Pig programs are compiled into sequences of MapReduce jobs, and executed in the Hadoop MapReduce environment [16].

II. Related work

This section describes some of the most recent work on parallelizing different algorithms with MapReduce.

The authors in [18] address the problem of efficient feature evaluation for logistic regression on very large data sets. Here they present a new forward feature selection heuristic that ranks features by their estimated effect on the resulting model's performance. They test the method on already available datasets from UCI, but also generate artificial datasets for which they know the logistic regression coefficients. They use that to evaluate the selected features.

By using the MapReduce paradigm in [21] a data intensive parallel feature selection method is proposed. In each map node, a method is used to calculate the mutual information and combinatory contribution degree is used to determine the number of selected features.

In [26] an implementation based on the MapReduce programming model of Naive Bayes is proposed. During the map phase all counts needed for calculating the conditional probabilities are emitted, and during the reduce phase they are aggregated.

A parallel implementation of the SVM algorithm for scalable spam filtering using MaprReduce is proposed in [22]. By distributing, processing and optimizing the subsets of the training data across multiple participating nodes, the distributed SVM reduces the training time significantly.

In [23] the authors propose a method for reducing the dataset to a small but representative subset that can be later on used for faster machine learning. Also here the speedup is being calculated against a cluster with 1 node or with minimal number of nodes so that the computation finishes in reasonable time.

In [24] an approach based on MapReduce for distributed column subset selection is proposed. In this approach each node has access to a random subset of features.

A wrapper approach for parallel feature selection is proposed in [25]. Here with features are added to the selected set if after their addition, the performance of the classifier does not degrade. Then in a second phase from the subset obtained in the previous step, features are removed if their discarding does not degrade the classifier performance.

III. Framework design

Writing parallel computer programs is more difficult than writing sequential ones, because parallelization introduces several new types of potential software bugs of which most common are race conditions, communication and synchronization between the different subtasks. Choosing Hadoop as environment for parallelization of algorithms overcomes many of those challenges without needing the programmer to put much effort for solving those kinds of challenges.

The framework that we propose in this paper consists of several phases, as shown on Fig. 1, and each of them is described in the following subsections.

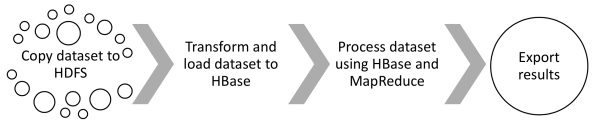

Fig. 1. Data flow phases based on MapReduce and Hadoop

A. Load data into HDFS

This is the first and most simple phase. This phase should be performed once or multiple times, depending on how the dataset is structured. The most common formats for datasets are:

- CSV (comma separated values). This format is usually used to store dense datasets.
- ARFF (Attribute-Relation File Format). Also used to store dense datasets.
- EAV (Entity Attribute Value). Used to store sparse matrices that have a lot of zeros and some non-zero elements.

If the dataset is only one file then this it will be copied from the Linux File System to HDFS using a simple command. This means that for this step cannot have parallelism. However if the dataset is dispersed into multiple files, then all of them can be copied simultaneously to HDFS. Be that as it may, this step usually is very fast compared to the following steps for machine learning, so its parallelization is not necessary at all.

B. Transformation and loading data into HBase

After the previous step III-A is finished the dataset files reside on HDFS. As it is extensively described in [9], each file in HDFS is replicated across several nodes for reliability. A typical file in HDFS is gigabytes to terabytes in size, splitted in blocks of 128 MB by default. If the files are too small than that could degrade the performance of the system and limit the level of parallelism. Map tasks usually process a block of input at a time. If the file is very small and there are a lot of them, then each map task processes very little input, and there are a lot more map tasks, each of which imposes extra bookkeeping overhead. Ideally the dataset that we have loaded to HDFS is one large file dispersed on multiple blocks so while we load it, transform it and store it in HBase we can have greater parallelism. Nevertheless, this step again is usually very fast especially compared to the step that performs the actual machine learning algorithm, so we do not recommend to spend too much time on optimizing the file sizes for better parallelism.

Even though we can achieve parallelism while processing files stored on HDFS, the control of degree of parallelism is difficult, more involved and at very low-level. On the other hand, HBase offers many other services built on top of HDFS, among which is a much better control of the degree of parallelism. This is due to the fact that the data in HBase is stored in a structured manner, while having various mechanisms that simplify random reads and writes from rows and columns. Namely, HBase tables are divided into potentially many regions, while one or more regions are serviced by a region server. The tables can be horizontally and vertically segmented while they are physically stored in HBase. Because many machine learning applications access the data by rows, in this paper we will continue to discuss only horizontal segmentation. As HBase was designed with very large tables in mind, a common use case is the following. A table at creation has only one region, which is serviced by one region server (a physical node in the Hadoop cluster). When this table is loaded with data it gets bigger and at some point it will become too big, so HBase will split its region into two regions. Then the new region will be assigned to the same region server or can be moved to another region server. The default splitting threshold is 10 GB. There are numerous reasons why HBase was designed that way, and we will not go into details about that. From parallelization perspective, this can pose a challenge, because for the automatic splits there are no guarantees that every region will contain equal amount of data, when are the splits going to occur exactly, are the regions going to be served by different region servers (nodes) etc. Further more, if one is using Hadoop for research purposes only then the dataset may not be that large, thus never overcoming the threshold for splitting. To overcome this challenge we can pre-split the tables on creation. This in turn means that the table can be configured at creation time to be stored on as many-regions as needed. Usually the number of regions is a multiple of the number of HBase region servers. The logic for having more region servers than acutal nodes is because the nodes are multi-core machines, so different threads on the same node can service different regions.

Before loading the dataset in HBase, we need to define the table structure and create it. Column names and data types are provided when storing data in each row, so at creation time we need to only specify a table name and a column family. There are some advanced configuration features that can be specified, but they are not topic of this discussion. Be that as it may, there is one very important decision that we need to make before loading data in the table. Because HBase tables, unlike SQL tables, cannot have secondary indexes, the primary key (row key) needs to be designed according to the usage patterns of the table. There are many considerations when designing the row key and they are very important for production use of HBase tables. However, for scientific use and for parallelizing machine learning algorithms, we need a simple design that allows uniform data distribution across nodes. In most scientific datasets the data instances (i.e. rows) do not have ids for their instances, or if they do they are not used for the actual machine learning. Nevertheless, in order to store a row in a HBase table, it needs a row key. For flat flies like CSV, ARFF the row key can be the line number of the instance. However, sequential row keys are very bad choice for HBase tables because the inserts will always be on the last region, therefore having no parallelism during the load, a problem called Region Server hotspotting. There are multiple ways of overcoming this problem, and one of them is a technique called salting. With this technique each sequential id is salts row keys with a prefix. The prefix is usually the modulo number between the original sequential id and the number of regions. Even though, this is very important topic, the step of loading the datasets in HBase is not the primary field of interest in this paper.

Once the dataset files are loaded into HDFS we need to transform them if needed and store them in HBase. If we have totally N rows in the dataset, and M regions, then we would like to distribute the data uniformly so each region gets N/M rows. This in turn means that we need to specify $M-1$ split points when creating the table. If we use sequential ids for the row key (like the line number in the file), than these split points would be: $N/M, 2N/M, 3N/M, ..., (M-1)N/M$. If we use a more sophisticated row key design, then the split points should reflect that design. For instance, if we take the modulo number of the id and the number of regions, then each region would get almost the same number of rows. This design of the row key allows fast random reads and writes, and additionally it facilitates addition of new data to the table at a later time without needing to redesign the table for equally dispersed load across regions. The following example shows

how a table can be pre-splitted on creation. The row key design is described with the function in listing 1. It returns a tuple in which the first element is the padded modulo number and the second part is the padded sequential id.

```
(pad(seq_id % num_regions), pad(seq_id))
```

Listing 1. Row key design

The numbers are padded with zeros so that they are lexicographically sorted. For instance if the ids vary from 1 to 100000 and we have 5 regions, then the Id 123 would be encoded to the following row key: $123\%5 = 3, pad(3) = 3, pad(123) = 000123 => row\_key = (3, 000123)$. The create statement for a table that has one column family 'r' and has 4 split points is shown in listing 2.

```
create 'dataset5', {NAME=>'r',
    COMPRESSION=>'gz', VERSIONS=>'1'},
{SPLITS=>[ "(1,", "(2,", "(3,", "(4," ]}
```

Listing 2. Crating HBase table with pre-split regions

Once the HBase table that will contain the dataset is created with appropriate split points for even data distribution across the cluster, the data can be loaded. One can write pure MapReduce jobs in Java or Python. If we choose that path, then we need to write a separate map and reduce function for each task. However, by using a scripting language called Pig Latin [15] we can write scripts from a higher-level perspective. These Pig scripts generate MapReduce tasks in the background so the programming effort is simplified and the development time is reduced. The downside of using Pig is that when Pig scripts are compiled into MapReduce jobs, there is some overhead but for longer running MapReduce tasks is insignificant because it adds up to 1 minute to the executing time. The listing 3 shows how we can load EAV files with Pig Latin. All variables starting with $ are parameters that are passed to the script on execution time.

```
register '$udf_path' using jython as
    paddingUDF;
eav_data = LOAD '$hdfs_data_path' USING
    PigStorage(',') as (id:int, feature:int,
    value:int);
eav_data_pad = FOREACH eav_data GENERATE
  paddingUDF.generate_rowkey(id,
      $padding_digits_id, $modulo_number) as
      idPad,
  paddingUDF.pad_number(feature,
      $padding_digits_feature) as featurePad,
  value;
eav_data_final = FOREACH eav_data_pad GENERATE
  idPad,
  [featurePad,value] as values;
STORE eav_data_final INTO '$table_dataset'
    USING
org.apache.pig.backend.hadoop.
    hbase.HBaseStorage('r:*');
```

Listing 3. Loading EAV files in HBase with Pig

The above script when compiled into MapReduce jobs would have only a map phase which will read the data from the HDFS file and store it in HBase. Because there is no grouping of keys needed, a reduce phase will not be generated. It uses two user-defined functions (UDFs) that are written in Python, but in order to be compatible with Pig and the rest of Hadoop, are compiled into Java byte code using Jython. The *generate_rowkey* function, shown in listing 4 calculates the row key according to the design shown in listing 1. With the function *pad_number*, shown in listing 5, the numbers are padded with zeros.

```
@outputSchema("padded:chararray")
def pad_number(number, numZeros):
    f = '%0' + str(int(numZeros)) + 'd'
    paddedNumber = f % int(number)
    return paddedNumber
```

Listing 4. Python UDF for generating row keys

```
@outputSchema("rowkey:(id_mod:chararray ,
    id:int)")
def generate_rowkey(id, id_num_digits,
    mod_number, mod_number_digits=3):
    prefix = int(id) % int(mod_number)
    return (pad_number(prefix,
        mod_number_digits), pad_number(id,
        id_num_digits))
```

Listing 5. Python UDF for padding numbers with zeros

During this step we can add various methods for data preprocessing like discretization, transformation and other methods that rely only on the values. But if the transformations need something like normalization that need the mean value for the whole dataset, then a separate step would be needed.

C. Processing HBase tables

After the dataset is loaded in a HBase table we can continue implementing machine learning algorithms. In general, this phase can be comprised of several substeps of data processing, depending on the nature of the algorithm that is being implemented. In this subsection we show how the mean value of each feature can be calculated. This is a simple task, but nevertheless, it illustrates this step of the framework. Listing 6 shows the Pig Latin script that calculates mean values. All parameters that start with *$* can be passed to Pig script when invoking it. Such parameters are the table names, number of features, index of the class value, number of padding digits etc. Lines 2 through 6 load the data from the table '$table_dataset'. In line 3 *'r:*'* denotes that all columns in the column family *r* will be loaded and line 6 denotes that they will be available as a dictionary (map) in the Pig script. Then at line 8 the UDF *expandFeatures* is invoked which accepts these arguments: *r* -the dictionary (map) of pairs *(featureIndexPadded, featureValue)*, the number of features in the dataset *$num_features*, maximum number of padding digits *$num_features_digits*, and the index of the class value *$label*. The UDF code is shown in listing 7. It will process the passed dictionary of feature indexes and values and will generate a list of triplets *(featureIndex, featureValue, class)*. The great thing about HBase is that it doesn't need to store the empty features from the dataset and each row can have different number of

columns. However when a row is loaded, like in the UDF *expandFeatures*, we can generate the zero-valued cells so they can be used for calculation of the mean value of the feature. In lines 10 and 11 the rows are grouped by feature index and we calculate the average of their values. Lines 15 through 17 store the mean values in a table that can be created in a similar manner as explained in III-A. We could easily export the calculated data in CSV files to HDFS.

What happens in the background for the script 6, is very peculiar. Pig will determine the number of regions of the table '$table_dataset' and it will start that number of map tasks. The number of reduce tasks is by default 1, but this can be also manually specified and does not depend of the table structures. During the map phase happen these statements: loading of data (lines 2-6), expanding each row to a list of tuples, merging (union) the list of tuples generated from each row into a final list (the *FLATTEN* operator in line 8) and it will emit the grouping key *featureIndexPadded* in line 10. Then in the reduce phase the rows will be grouped by the *featureIndexPadded* key, the mean value will be calculated (lines 12-14) and finally the result will be stored in the table *$table_feature_mean_value*. In this table the row key is *featureIndexPadded* and the mean value is stored in column family *r* and column *featureValueMean*. We can easily change this statement to export the results in a CSV file to HDFS.

```
1  register '$udf_path' using jython as
       paddingUDF;
2  dataset = LOAD '$table_dataset' USING
       org.apache.pig.backend.hadoop.
3  hbase.HBaseStorage('r:*',
4         '-loadKey=true') AS
5       (rowkey:tuple(prefix_padded:chararray,
           id_padded:chararray, id:int),
6        r:map[]);
7  dataset_expanded = FOREACH dataset GENERATE
8    FLATTEN(paddingUDF.expandFeatures(r,
         $num_features, $num_features_digits,
         '$label'));
9
10 feature_value_class_group = GROUP
       dataset_expanded BY (featureIndexPadded);
11
12 feature_value_class_mean = FOREACH
       feature_value_class_group GENERATE
13   group as featureIndexPadded,
14   AVG(dataset_expanded.featureValue) as
         featureValueMean:double;
15 STORE feature_value_class_counts INTO
       '$table_feature_mean_value' USING
16      org.apache.pig.backend.hadoop.
17      hbase.HBaseStorage('r:featureValueMean');
```

Listing 6. Pig script for calculating mean value of features

The code in listing 7 shows how UDF *expandFeatures* is implemented. Lines 1 and 2 describe the output format of the data that is being returned. In this the function will return a bag (Pig Latin equivalent of a unordered list) of tuples. Each tuple is a triplet containing the featureIndex (padded with zeros), the featureValue and the class. For the calculation of the mean values of the features the class is not needed, but for many other algorithms and metrics it will be. Line 5 calls a function for decoding the input arguments from byte representation to appropriate types - string and dictionary. The in the loop in lines 8 through 11 from the dictionary of pairs feature index and feature value, we generate a list of triplets, which the function returns at line 12. The interesting thing that this function does is that from a row of the HBase table it generates a list of tuples, so in a way it's transposing a row into multiple rows with one composite column (i.e. the triplets). Then in the Pig script in listing 6, with the FLATTEN operator merges those rows into a common set of rows of triplets, thus transposing the whole dataset. Having all features, values and class as triplets in separate rows is useful and allows us to leverage the SQL-like capabilities of Pig Latin. Further on in separate scripts we can normalize the values of the dataset by joining to the set of mean values and doing appropriate normalizations.

```
1  @outputSchema("feature_value_class:bag{
2  t:(featureIndexPadded:chararray ,
       featureValue:double, class:chararray)}")
3  def expandFeatures(featureValuePairs,
       numFeatures, numDigits, classKey):
4      currentClass =
           featureValuePairs.get(classKey, 0)
5      (currentClassFinal,
           featureValuePairsFinal) =
           decodeBytes(currentClass,
           featureValuePairs)
6
7      feature_value_class = []
8      for featureIndex in range(1,
           int(numFeatures) + 1):
9          featureIndexPadded =
               pad_number(featureIndex, numDigits)
10         featureValue =
               featureValuePairsFinal.get(
               featureIndexPadded, float(0))
11         feature_value_class.append(
               (featureIndexPadded, featureValue,
               currentClassFinal))
12     return feature_value_class
```

Listing 7. UDF for expanding a HBase row into a list of tuples

D. Exporting results

After the main work is performed during the previous step III-C, the results need to be exported. Using Pig Latin, the output from the Pig scripts can be stored in HDFS files or HBase tables. In the Hadoop ecosystem there are advanced services like Flume or Sqoop that facilitate connectivity with RESTful web services, various SQL databases etc. The most simple way is however to export the results in HDFS files in a common format like CSV, and then to export the HDFS files to the Linux file system.

IV. Conclusion and Future Work

In this paper we have proposed a framework for parallelization of machine learning algorithms by using the Apache Hadoop platform including its services HDFS, Yarn MapReduce and HBase, and Pig Latin as a scripting language. We have demonstrated how can we manually set the degree of parallelism by pre-splitting the HBase tables so they have

optimal number of regions and even data distribution across regions. We have also provided exemplary user-defined functions written in Python for transforming and formatting the data while it's being loaded, and also a way to efficiently decode the sparse matrix for which we only keep the non-zero elements. We have provided an exemplary script that calculates the mean values of each feature.

In order to affirm the proposed framework, we will implement various machine learning algorithms with it, measure the impact of the parallelization with different cluster configurations on different datasets. The performance should be evaluated in terms of speedup vs the sequential versions of the algorithms.

REFERENCES

[1] 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*, ser. OSDI'04. Berkeley, CA, USA: USENIX Association, 2004, pp. 10–10. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251254.1251264

[2] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," in *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, ser. OSDI '06. Berkeley, CA, USA: USENIX Association, 2006, pp. 15–15. [Online]. Available: http://dl.acm.org/citation.cfm?id=1267308.1267323

[3] "Hadoop wiki: List of institutions that are using hadoop for educational or production uses, howpublished = https://wiki.apache.org/hadoop/poweredby, note = Accessed: 2015-01-29."

[4] D. Miner, *MapReduce design patterns*. Sebastopol, CA: O'Reilly, 2013. ISBN 9781449327170

[5] C. Lam, *Hadoop in action*. Greenwich, Conn: Manning Publications, 2011. ISBN 9781935182191

[6] A. Holmes, *Hadoop in practice*. Shelter Island, NY: Manning, 2012. ISBN 9781617290237 1617290238

[7] T. White, *Hadoop: the definitive guide*, 3rd ed. Beijing: O'Reilly, 2012. ISBN 9781449311520

[8] "Apache hadoop nextgen mapreduce (yarn), howpublished = http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/yarn.html, note = Accessed: 2015-01-29."

[9] "Hdfs architecture guide, howpublished = http://hadoop.apache.org/docs/r1.2.1/hdfs_design.html, note = Accessed: 2015-01-29."

[10] L. George, *HBase the definitive guide*. Sebastopol, CA: O'Reilly, 2011. ISBN 9781449315771 1449315771. [Online]. Available: http://public.eblib.com/choice/publicfullrecord.aspx?p=769368

[11] Y. Jiang, *HBase administration cookbook master HBase configuration and administration for optimum database performance*. Birmingham: Packt Publishing, 2012. ISBN 9781849517157 1849517150 1849517142 9781849517140. [Online]. Available: http://site.ebrary.com/id/10598980

[12] N. Dimiduk and A. Khurana, *HBase in action*. Shelter Island, NY: Manning, 2013. ISBN 1617290521 9781617290527

[13] A. Awadallah and D. Graham, *Hadoop and the Data Warehouse: When to Use Which*. Coudera, Teradata, 2011. [Online]. Available: http://www.teradata.com/When-to-Use-Hadoop

[14] E. A. Brewer, "Towards robust distributed systems (abstract)," in *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC '00. New York, NY, USA: ACM, 2000. doi: 10.1145/343477.343502. ISBN 1-58113-183-6 pp. 7–. [Online]. Available: http://doi.acm.org/10.1145/343477.343502

[15] A. F. Gates, O. Natkovich, S. Chopra, P. Kamath, S. M. Narayanamurthy, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava, "Building a high-level dataflow system on top of map-reduce: The pig experience," *Proc. VLDB Endow.*, vol. 2, no. 2, pp. 1414–1425, Aug. 2009. doi: 10.14778/1687553.1687568. [Online]. Available: http://dx.doi.org/10.14778/1687553.1687568

[16] A. Gates, *Programming Pig*. Sebastopol: O'Reilly Media, 2011. ISBN 9781449317690 1449317693 9781449317683 1449317685. [Online]. Available: http://public.eblib.com/choice/publicfullrecord.aspx?p=801461

[17] E. Zdravevski, P. Lameski, A. Kulakov, and D. Gjorgjevikj, "Feature selection and allocation to diverse subsets for multi-label learning problems with large datasets," in *Computer Science and Information Systems (FedCSIS), 2014 Federated Conference on*, Sept 2014. doi: 10.15439/2014F500 pp. 387–394.

[18] S. Singh, J. Kubica, S. Larsen, and D. Sorokina, "Parallel large scale feature selection for logistic regression." in *SDM*. SIAM, 2009, pp. 1172–1183.

[19] Z. Zhao, R. Zhang, J. Cox, D. Duling, and W. Sarle, "Massively parallel feature selection: an approach based on variance preservation," *Machine Learning*, vol. 92, no. 1, pp. 195–220, 2013. doi: 10.1007/s10994-013-5373-4. [Online]. Available: http://dx.doi.org/10.1007/s10994-013-5373-4

[20] "Mpi: A message-passing interface standard. version 3.0, howpublished = http://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf, note = Accessed: 2015-01-29."

[21] Z. Sun and Z. Li, "Data intensive parallel feature selection method study," in *Neural Networks (IJCNN), 2014 International Joint Conference on*, July 2014. doi: 10.1109/IJCNN.2014.6889409 pp. 2256–2262.

[22] G. Caruana, M. Li, and M. Qi, "A mapreduce based parallel svm for large scale spam filtering," in *Fuzzy Systems and Knowledge Discovery (FSKD), 2011 Eighth International Conference on*, vol. 4, July 2011. doi: 10.1109/FSKD.2011.6020074 pp. 2659–2662.

[23] I. Triguero, D. Peralta, J. Bacardit, S. García, and F. Herrera, "Mrpr: A mapreduce solution for prototype reduction in big data classification," *Neurocomputing*, vol. 150, pp. 331–345, 2015.

[24] A. K. Farahat, A. Elgohary, A. Ghodsi, and M. S. Kamel, "Distributed column subset selection on mapreduce," in *Data Mining (ICDM), 2013 IEEE 13th International Conference on*. IEEE, 2013, pp. 171–180.

[25] A. Guilln, A. Sorjamaa, Y. Miche, A. Lendasse, and I. Rojas, "Efficient parallel feature selection for steganography problems," in *Bio-Inspired Systems: Computational and Ambient Intelligence*, ser. Lecture Notes in Computer Science, J. Cabestany, F. Sandoval, A. Prieto, and J. Corchado, Eds. Springer Berlin Heidelberg, 2009, vol. 5517, pp. 1224–1231. ISBN 978-3-642-02477-1. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-02478-8_153

[26] L. Zhou, H. Wang, and W. Wang, "Parallel implementation of classification algorithms based on cloud computing environment," *TELKOMNIKA Indonesian Journal of Electrical Engineering*, vol. 10, no. 5, pp. 1087–1092, 2012.