Data-Mining Seminar: Mining of Massive Datasets

Steven Lang
steven.lang.mz@gmail.com

ABSTRACT

This report is orientated towards the book *Mining of Massive Datasets* [6] and describes two novel frameworks for efficient massive data analysis on compute clusters. A short introduction to compute clusters and a distributed file system will build the base for MapReduce. After showing a few examples of applications and optimizations, the disadvantages are listed and lead to the second framework, called Spark, which captures on these drawbacks. The report shows that Spark outperforms the MapReduce implementation Hadoop at iterative tasks with working sets.

1. INTRODUCTION

Human generated data is growing at an exponential rate. Fig. 1 shows the hours of YouTube upload-content and the number of tweets per minute of the past three years. This ever growing rate has lead to massive amounts of data in the past. The wish to analyse those datasets brought the necessity of frameworks for compute clusters, which concentrate on efficiently parallelizing operations applied on the data. In this context efficiency will mostly be achieved through minimizing the amount of data which is read and written during the whole analysis process.

Managing those immense amounts of regular data quickly has lead to a novel approaches of data-mining. Two well known examples are

- Ranking web-pages by importance (iterated matrix-vector multiplication with huge dimensionality)
- Querying social-networks (graphs with hundreds of millions of nodes and many billion edges)

These modern data-mining applications have evolved a new software stack which will be described in the following.

2. COMPUTE CLUSTER

ACM ISBN .
DOI:

	2012	2014	2015
YouTube Uploads	48h	72h	300h
Tweets	100k	277k	350k

Figure 1: 60 seconds in the web. [5]

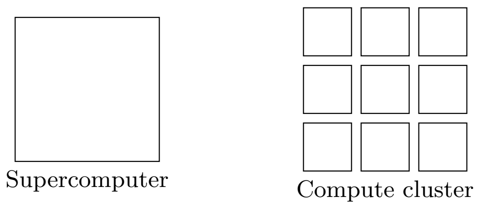

Figure 2: Supercomputing vs. Clustercomputing

Instead of scaling up with supercomputers (high costs), the new trend is to scale out and build compute clusters (Fig. 2) with large collections of commodity hardware (low cost) serving as compute nodes connected in a network.

A new file system combined with new frameworks especially written for the purpose of cluster computing provide the ability to deal with problems coming along, as well as making the most benefit out of it.

Cluster systems need those elements to solve problems which appear by node failures like the consistent availability of information/data which is solved by the new file system DFS and the modularity of tasks which is addressed by a programming model called MapReduce.

Compute nodes are stored on racks with a typical number of 8 up to 64 each, which are connected by Ethernet and switches (Fig. 3). As interrack-connection will be a bottleneck of the cluster, it is higher than the intrarack-connectivity which is usually gigabit Ethernet.

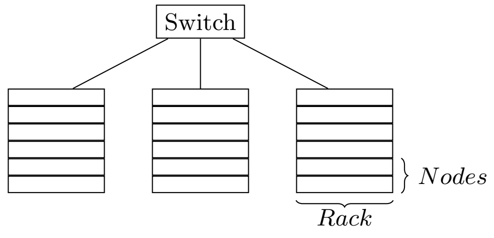

Figure 3: An example cluster with nodes and racks connected by a switch

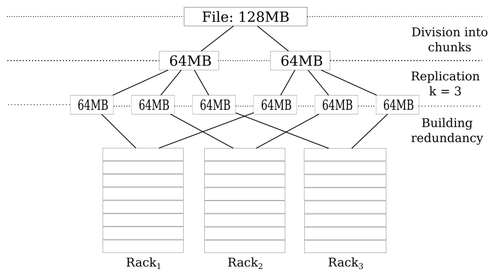

Figure 4: Data-redundancy in DFS

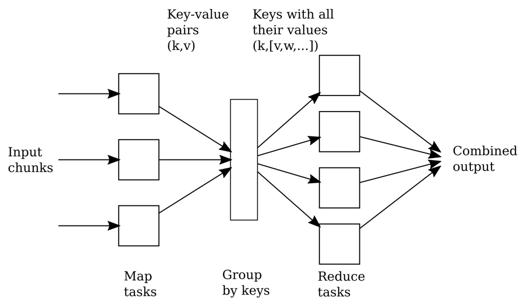

Figure 5: Schematic of a MapReduce computation

3. DISTRIBUTED FILE SYSTEM (DFS)

Files which are stored on the DFS need to fulfill certain conditions, such as they need to be large. Another criterion is that they are rarely updated, since long computations take place on the same data.

The distributed file system solves the problem of consistent availability of data by generating redundancy on purpose. Files are divided into chunks of a fixed size (e.g. 64mb). These chunks are then replicated k-times (typically $k = 3$) at different compute nodes. To avoid loss of redundancy by rack failures these k compute nodes should be located on k different racks (Fig. 4). Which chunk is stored on which node can be found on a map on the master node.

4. MAPREDUCE

4.1 Introduction

MapReduce is a programming model allowing scalability across a complete compute cluster. It takes care of parallelism, task-coordination and node failures. The only functions which need to be implemented are *Map* and *Reduce*

- map: $(key_1, value_1) \mapsto [(key_2, value_2)]$
- reduce: $(key_2, [value_2]) \mapsto [(key_3, value_3)]$

Map takes a key-value pair and produces several key-value pairs which can be of arbitrary type. The master-controller collects, groups and sorts those by key. Each key with its list of values will then be assigned to a Reduce-Task. Reduce outputs a new list of key-value pairs (Fig. 5).

The following terminology will be used:

- Mapper: The map-function
- Reducer: The reduce-function
- Map-Task: A process - takes one or more chunks of input data and executes the mapper

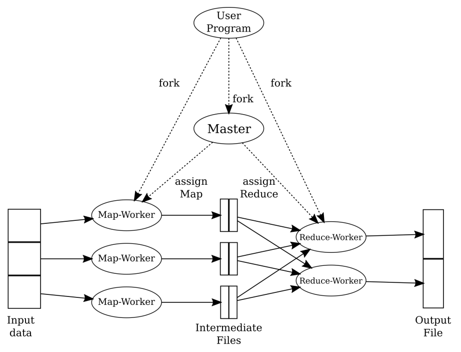

Figure 6: Execution of a MapReduce program

- Reduce-Task: A process - takes one or more key-value pairs and executes the reducer

4.2 Illustrating MapReduce with a Common Example: WordCount

The goal is to count words in a large repository of documents. The Map-Task gets a document as input and splits it up into a list of words $w_1, ..., w_2$. For each word w_i, it outputs a pair $(w_i, 1)$. The master-controller now groups the pairs by key which means n occurrences of $(w_i, 1)$ are turned into a pair (w_i, L), where L is a list containing n-times the number *1*. The keys are hashed into r (user defined parameter for the number of Reduce-Tasks) buckets. Each bucket is assigned to one Reduce-Task. The Reduce-Task simply reduces the list of values for each key by summing them up and emits a new pair (w_i, v_i) where v_i is the sum of the list L.

For some specific tasks the reducer is associative and commutative. This means that the order of combination of the values does not change the result. Furthermore one can then move some of the reducer-logic to the mapper. Applied on the WordCount example, this means each Map-Task can count the occurrences of a word w_i and if it has finished all of its input, emit a pair (w_i, m_i) where m is the number of occurrences of w_i.

4.3 Further Details on MapReduce

The user forks the master-controller process and a bunch of Worker processes. The master-controller assigns those Workers either Map-Tasks (Map-Worker) or Reduce-Tasks (Reduce-Worker) (Fig. 6).

Since each Map-Task creates an intermediate file for each Reduce-Task, limiting the number of Reduce-Tasks is necessary to avoid storage overflows. These files from the Map-Tasks are stored at the node where the Map-Worker is running. Reduce-Tasks are informed of the location of its input files by the master controller.

A Tasks status can be either idle, executing at a particular Worker, or completed. This status is continuously reported to the master controller. When a task at a worker finishes, the master schedules a new Task for that process.

4.4 Coping with Node Failures

The worst-case scenario for cluster-computing is a failure of the compute node at which the master controller is executing. This leads to a restart of the whole MapReduce job.

Less tragic node failures are those of compute nodes, at which a Map-Worker is located at. If this happens, it is necessary to restart all of the Map-Workers Map-Tasks which were executing there, since their output was stored on the failed node, which means that some Reduce-Tasks will have lost their input data. The status of these Map-Tasks will be set to idle and the master controller schedules them to the next free Map-Worker. The Reduce-Tasks which correspond to the lost input are informed about its new location

The simplest case is the failure of a compute node which is running a Reduce-Worker. The master controller sets the Reduce-Task's status to idle and reschedules it on another Reduce-Worker.

4.5 Application of MapReduce: Matrix-Vector-Multiplication

The main intention of Google creating an implementation of MapReduce was the need for large matrix-vector multiplications for their search-engine algorithm PageRank.

The input is a $n \times n$ matrix M and a vector V of length n. The standard matrix-vector multiplication is denoted as:

$$x_i = \sum_{j=0}^{n} m_{ij} v_i$$

4.5.1 *V fits into Main Memory*

Lets first assume that the magnitude of n is as such limited that the vector V fits into the compute nodes main memory.

The entry m_{ij} in the i-th row and the j-th column of the matrix M will be stored as a triple (i, j, m_{ij}) on the DFS. Analogously will the j-th element v_j of the vector V be stored as a tuple (j, v_j).

A Map-Task loads the vector V into its memory. Given a chunk of the matrix M it applies the mapper to each triple (i, j, m_{ij}) and emits the tuple $(i, m_{ij}v_j)$.

The reducer gets the input $(i, [m_{i0}v_0, ..., m_{in}v_n,])$ which contains all terms of the sum of the i-th value x_i of the result vector X. The output will be the sum of the given term (i, x_i).

4.5.2 *V does not fit into the Main Memory*

If V does not fit into the main memory, a possible solution for the matrix-vector multiplication would be to split up the matrix into vertical stripes of equal width and the vector into horizontal stripes of equal height (Fig. 7). The number of stripes need to be adjusted so that each stripe can fit into main memory. Each Map-Task gets a whole stripe of the vector and a chunk of the corresponding stripe of the matrix. The mapper and reducer do not need to be changed and behave like in the first example.

4.6 Relational Algebra Operations

The model of MapReduce easily allows to implement standard relational algebra operations such as selection, projection, union, intersection, difference natural join and grouping and aggregation. An example for the selection is show below and the natural join will be shown in Section 4.7.1.

Selections filter a relation R and return each tuple that fulfills a certain condition C. In relational algebra this is denoted with $\sigma_C(R)$.

- The mapper tests for each tuple t from R if it fulfills the condition C. If this is the case, it emits a tuple (t, t).

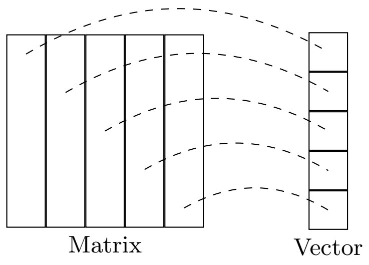

Figure 7: Striped matrix-vector-multiplication

- The reducer simply applies the identity function: it gets $(t, [t])$ as input and emits $(t, [t])$.

4.7 Communication Cost

To achieve decent performance one needs to address the bottleneck of the system. In most cases this refers to moving data between nodes in the cluster (communication).

The size of the input to a MapReduce job is defined as its communication cost. It can either be measured in bytes or in number of parts such as the number of input tuples of a join-operation.

It is possible to concatenate MapReduce jobs, so the reduce output of one job will be the map input for the next job. Hence looking at a network of MapReduce jobs, the communication cost of such a network is the sum of the input of each job. The output of a job will not be counted into the costs since it is always the input for another job which will already be counted in the above mentioned sum (except for the last output which will be the final result).

4.7.1 *Example: Natural Join*

Lets assume one wants to join a relation R of size r with attributes A and B with another relation S with attributes B and C of size s. Chunks of the file of R and S are the input for the Map-Task, hence the sum of all inputs is $r + s$.

The mapper gets a tuple $(R, (a, b))$ or $(S, (b, c))$ which will be mapped into $(b, (R, a))$ or $(b, (S, c))$ respectively. Usually the Reduce-Tasks are executed on another node, so the input of the Reduce-Tasks will be as large as the input for the Map-Tasks was.

The reducer gets a tuple (b, L) where L is a list of (R, a)'s and (S, c)'s which agreed on b. The reducer builds all combinations of (R, a) and (S, c) of this list and outputs for each combination $(k, (a, b, c))$ (where the key is irrelevant). Usually this output can be much larger than $r + s$ but one can assume for large outputs some form of postprocessing such as grouping or aggregation.

Counting the resulting computation costs leads to an estimation of $\mathcal{O}(r + s)$. This performance measure can e.g. help to optimize the order of multiway-joins.

4.7.2 *Example: Cascaded Two-Way Joins*

We now shall do two two-way joins $R(A, B) \bowtie S(B, C) \bowtie T(C, D)$ (with sizes r, s, t) as an example.

Let p be the probability that a tuple from R and a tuple from S agree on B, aswell as a tuple from S and a tuple from T agree on C.

If R and S are joined first, the communication costs will be $\mathcal{O}(r+s)$. The output size of $R \bowtie S$ is prs. Joining this result with T leads to communication costs of $\mathcal{O}(t + prs)$. Hence

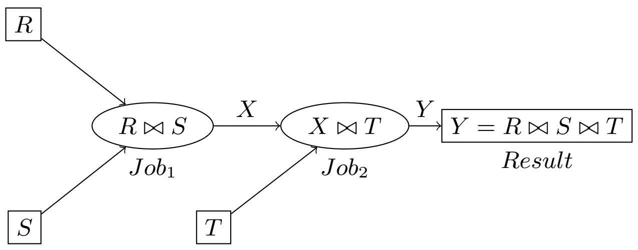

Figure 8: Schematic of cascaded two-way joins

the entire costs are $\mathcal{O}(r + s + t + prs)$. Another way would have been joining S and T first and R afterwards, resulting in communication costs of $\mathcal{O}(r+s+t+pst)$. Depending on r and t one can now optimize the communication costs of the multi-join and thus improve its performance.

4.8 Complexity Theory for MapReduce

Even though communication cost is a bottleneck of a MapReduce cluster, it is not the only parameter which needs to be optimized. If the hard drive of a node would be capable of storing the whole dataset, one could simply execute all Map-Tasks and Reduce-Tasks on the same node, resulting in zero communication costs but a large wallclock time[1].

4.8.1 Reducer Size and Replication Rate

To describe the performance of a MapReduce algorithm further, two more characteristics can be defined. The *reducer size* q is an upper bound on the size of the list which comes along with a key k as input to a reducer. Ensuring a low reducer size results on one side in a high degree of parallelism, minimizing the wallclock time. On the other side, choosing q small enough allows the reducer to execute completely in main memory and thus avoiding swapping between main memory and disk.

The *replication rate* r is the number of all mapper-output key-value pairs divided by the number of all mapper-input key-value pairs.

4.8.2 Example: Similarity Joins

Given a set of one million images with a size of one megabyte each, it could be interesting to determine the similarity $s(x, y)$ of two images x and y. Solving this problem with a mindless MapReduce approach would look like the following.

For each input (i, P_i) the mapper produces 999,999 outputs of the form

$$(\{i, j\}, P_i), \forall j \in \{0, 1, .., 999.999\} \backslash \{i\}$$

The reducer then executes the similarity function on its input $(\{i, j\}, [P_i, P_j])$.

This algorithm will not be applicable for an obvious reason. Even though the reducer size q is only 2MB (since each reducer gets two images to compare), the replication rate is unbearable. The amount of data is 1,000,000MB (picture size) times 999,999 (replication rate) times 1,000,000 (number of input pictures) resulting in 10^{18} bytes to send along the cluster which takes approx. 300 years on gigabit-Ethernet.

To overcome this issue one can group pictures into g groups with $10^6/g$ pictures each. The mapper takes (i, P_i) and

[1]time it takes a parallel algorithm to finish

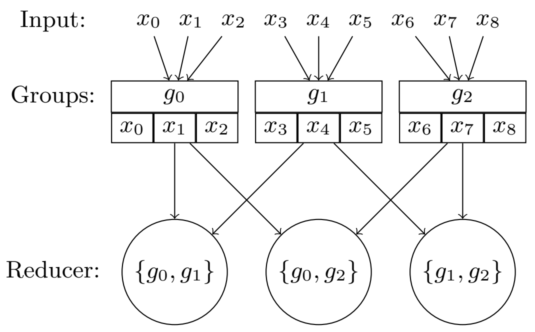

Figure 9: Grouping approach for similarity joins with nine inputs and three groups

transforms it into *g-1* (replication rate) key-value pairs of $(\{u, v\}, (i, P_i))$, where u is the group of i and v is one of the other *g-1* groups (Fig. 9).

The reducer gets the key $\{u, v\}$ with a list of $2 \times 10^6/g$ (reducer size) (k, P_k). For each pair (i, P_i) and (j, P_j) where i and j are from different groups, the reducer applies $s(P_i, P_j)$. For pictures from the same group u, only the reducer with the key $\{u, (u + 1) \mod g\}$ needs to apply the similarity function.

With $g = 1000$ each list in the reducer is of size 2GB which perfectly fits into standard cluster node main memory. On the other hand the communication is now reduced to 1,000,000MB (picture size) times 999 (replication rate) times 1,000,000 (number of input pictures) resulting in 10^{15} bytes which is 1000 times less than in the mindless approach.

4.9 Disadvantages of MapReduce

Although MapReduce performs very well on acyclic workflows, quite a number of disadvantages have become visible in the past. On the one hand it has a very strict workflow. More complex workflows hence need to be translated into a cascading MapReduce-Jobs. On the other hand the redundancy introduced by DFS results in a high I/O load which further reduces the performance of the actual MR-Job. Furthermore, each algorithm which is going to be implemented in MapReduce needs to be mapped to a map-function and a reduce-function.

For jobs which use working sets of data, namely iterative machine learning algorithms or data analysis tools, MapReduce will generate a lot overhead by continuously reading the input data for each iteration again and again.

In Section 5 a new framework will be introduced which captures the above mentioned disadvantages.

5. SPARK: CLUSTER COMPUTING WITH WORKING SETS

Spark [9] is a new framework introducing a concept which optimizes the distribution of datasets under the aspect of parallelism.

Iterative jobs in MapReduce will reload the whole data for each job, resulting in high communication costs. Therefore Spark introduces an abstraction called RDDs, with which Spark performs very well on iterative jobs and interactive analysis as shown in Section 5.8.

It is implemented in Scala, a general purpose programming language and makes use of Scala's full support for functional programming.

5.1 Programming Model

The Spark workflow is controlled by a so called driver program. It defines RDDs, transformations and parallel operations on these RDDs and collects their results.

5.2 Resilient Distributed Datasets (RDDs)

Resilient distributed datasets are immutable collections of objects which are partitioned across different nodes in the cluster. If a partition is lost, it can be rebuilt from other partitions. Furthermore a RDD does not need to be persistent on the storage explicitly, but only needs to know how it can be computed from other RDDs. Hence RDDs implement the principle of lazy evaluation: computing the data only when it is necessary.

RDDs can be constructed in four different ways:

- By loading a file from a shared file system
- By making a Scala collection available on different nodes (Section5.3)
- By transformation of a RDD. Spark provides several operations such as *flatMap* which executes a user-provided function that maps each element of type A to list of elements of type B. Other operations are map ($A \mapsto B$) or filter ($A \mapsto \{true, false\}$)
- By enabling or disabling the persistence of a RDD. By default partitions of a RDD are discarded after they have been used in an operation just as in MapReduce. Though the user can make the RDD persistent if they it again after the operation has finished (such as in iterative tasks). This persistence can be established on two different ways:
  - By caching the RDD. This keeps the lazy evaluation, but tries to keep the RDD in memory after its first computation, since the user knows he needs it at a later point again.
  - By storing the computed dataset on the current file system.

5.3 Parallel Operations

Next to RDDs, Sparks second main abstraction are parallel operations which can be applied on RDDs:

- reduce can meld two dataset instances to one single and yield its result at the driver program. Ex. $(x, y) \mapsto x+y$ would reduce two instances x and y to its sum. In addition, the function which is passed to reduce needs to be associative.
- collect returns all instances of the dataset to the driver program
- foreach sends each instance through a user defined function

5.4 Shared Variables

One way Spark makes use of Scala's functional programming is when allowing the user to pass *closures* to parallel operations, such as *map*, *filter* and *reduce*. A property of closures is that variables which were in the same scope as where the closure was created, are referable from within the closure at any time. This means when the user passes a closure to a parallel operation, all its accessible variables will be copied to the node at which the operation is executed. To enhance the usage of these, Spark introduced two types of shared variables:

- To avoid unnecessary communication between nodes when the same operation is called twice, the user shall use *broadcast variables*. This ensures that a variable is only copied once to a node, even when a operation is invoked multiple times with the same closure.
- Accumulators are shared variables on which the worker nodes can only apply an associative "add" operation. The type of accumulators need to support an "add" operation and a "zero" element to be associative. The value of the accumulator can only be read at the driver.

5.5 Text Search with Spark

```
val file = spark.textFile("hdfs://...")
val errs = file.filter(_.contains("ERROR
  "))
val ones = errs.map(_ => 1)
val count = ones.reduce(_+_)
```

1. Reads a text file from the HDFS into a RDD *file*. Each element in *file* is a line in the actual file.
2. Forwards each instance of *file* which contains the substring 'ERROR' into a new RDD *errs*
3. Maps each instance of *errs* to the value *1*
4. Reduces all instances to their sum

Since Spark's RDDs are lazy, *errs* and *ones* are not evaluated until *reduce* is called on each node. This means when *reduce* is invoked, each node gets a chunk of the input, evaluates *ones* and returns their sum to the driver which further reduces the nodes outputs to their sum.

A big advantage over MapReduce is that Spark allows intermediate results to be persistent throughout multiple operations. The line

```
val cachedErrs = errs.cache()
```

allows the reuse of the RDD in parallel operations, without recomputing it a second time. After *cachedErrs* is evaluated once, the node tries to cache its partition of *cachedErrs* in memory which results in great performance improvements.

5.6 WordCount with Spark [2]

```
val file = spark.textFile("hdfs://...")
val words = file.flatMap(_ => _.split("
  ")).map(_ => (_, 1))
val counts = words.reduceByKey(_ + _)
```

1. Reads a text file from the HDFS into a RDD *file*. Each element in *file* is a line in the actual file.
2. Maps each line to a list of its words and maps these words to key-value pairs with the word as key and the value one
3. Groups the above generated key-value pairs by key and aggregates all values with the same key as their sum

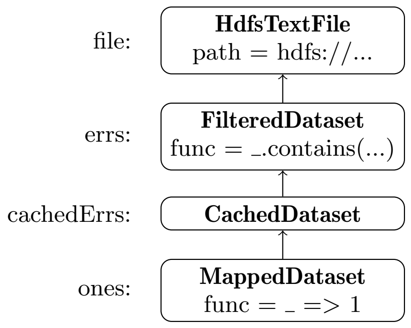

Figure 10: Lineage chain for the RDDs in Section 5.5 (cached errors included)

5.7 RDD Implementation

Lets take Section 5.5 as an example to explain the implementation of RDDs. A RDD is a object which describes the transformation of a parent RDD. They can thus be represented as a lineage chain, shown in figure 10. A single RDD has a pointer to its parent and information about the transformation. If a compute node fails, this means the partitions can be reconstructed from their parents RDDs. Thus Spark's fault tolerance is achieved through the abstraction of resilient distributed datasets.

A RDD object implements an interface containing the following three operations:

- *getPartitions*: Returns a list of the partition IDs
- *getIterator(partition)*: Returns an iterator for the given partition
- *getPreferredLocations(partition)*: Returns information about where to schedule a task to achieve data locality

RDD types differ in the way they implement the above described interface. In example 5.5 there are three main types of RDDs. The *HdfsTextFile* uses the HDFS block IDs as partition IDs. The preffered partition location is the actual block location and *getIterator* on a partition opens a stream to that block. A *MappedDataset* has the same partitions and preferred locations as its parent, but applies the map-function while iterating on the parents elements. The *CachedDataset* has equal partitions and on creation equal preferred partition locations as its parent. If the cache operation as described in Section 5.2 is called on the RDD, the preferred locations are updated to the nodes local location.

5.8 Performance Results

The improvements in iterative jobs are shown in an example of logistic regression [9] used on a 29 GB dataset. Hadoop [1] (a MapReduce implementation from Apache) needs 127s for each iteration. Spark takes 174s for the first iteration and only 6s for all following iterations (see Fig. 11). This is due to Spark's advantage of cached datasets. The first iteration is in the same magnitude as Hadoop which is dominated by hard-drive reading time. The following iterations can access and reuse cached data from the first iteration, whereas in Hadoop each iteration is a closed MapReduce job, thus reading the input data again and again in each iteration.

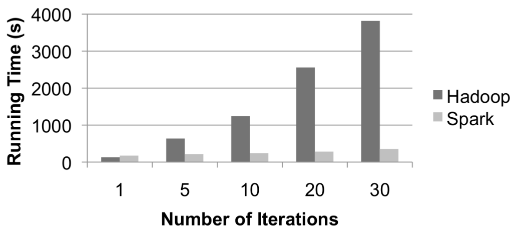

Figure 11: Logistic regression performance in Hadoop and Spark

6. RELATED WORK

Pig [4] and Hive [7] are two common frameworks built on top of Hadoop which implement a declarative style for SQL-like queries.

Another approach for the analysis of massive datasets is data stream mining in real time as used in the MOA [3] workbench. Evaluating algorithms can be done in the Scavenger [8] framework which allows efficient evaluation of dynamic and modular algorithms.

7. REFERENCES

[1] Apache Hadoop. http://hadoop.apache.org/. Accessed: 2016-07-14.

[2] Apache Spark Examples. http://spark.apache.org/examples.html. Accessed: 2016-06-14.

[3] A. Bifet, G. Holmes, R. Kirkby, and B. Pfahringer. Moa: Massive online analysis. volume 11, pages 1601–1604. JMLR.org, Aug. 2010.

[4] 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. volume 2, pages 1414–1425. VLDB Endowment, Aug. 2009.

[5] A. Hildebrandt. Big Data: Algorithms, methods and techniques for analysing massive datasets. JGU Mainz.

[6] A. Rajaraman and J. D. Ullman. *Mining of Massive Datasets*. Cambridge University Press, New York, NY, USA, 2011.

[7] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Anthony, H. Liu, and R. Murthy. Hive - a petabyte scale data warehouse using Hadoop. In F. Li, M. M. Moro, S. Ghandeharizadeh, J. R. Haritsa, G. Weikum, M. J. Carey, F. Casati, E. Y. Chang, I. Manolescu, S. Mehrotra, U. Dayal, and V. J. Tsotras, editors, *ICDE*, pages 996–1005. IEEE, 2010.

[8] A. Tyukin, S. Kramer, and J. Wicker. Scavenger - A Framework for the Efficient Evaluation of Dynamic and Modular Algorithms. In A. Bifet, M. May, B. Zadrozny, R. Gavalda, D. Pedreschi, J. Cardoso, and M. Spiliopoulou, editors, *Machine Learning and Knowledge Discovery in Databases*, volume 9286 of *Lecture Notes in Computer Science*, pages 325–328. Springer International Publishing, 2015.

[9] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster Computing with Working Sets. In *Proceedings of the 2Nd USENIX Conference on*

Hot Topics in Cloud Computing, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.