Open Source Big Data Analytics Frameworks Written in Scala

John A. Miller, Casey Bowman, Vishnu Gowda Harish and Shannon Quinn
Department of Computer Science
University of Georgia
Athens, GA, USA
{*jam@cs., bowman99@, vishnu.gowdahari25@, squinn@cs.*}*uga.edu*

***Abstract*—Frameworks for big data arguably began with Google's use of MapReduce. Since then, a huge amount of progress has been made in the development of big data frameworks, many of which have been released as open source. Further to increase portability and ease of set-up, many are coded in a Java Virtual Machine (JVM) based language, e.g., Java or Scala. In addition, processing of big data involves the flow of data, and of course, the processing of data as it flows. This computational paradigm is a natural for functional programming. Furthermore, the map, reduce and combiner have analogs in functional programming. There has been a trend in the last few years toward developing open source big data frameworks written in Scala to support big data analytics. Scala is a modern JVM language that supports both object-oriented and functional programming paradigms.**

***Keywords*-big data; analytics; frameworks; functional programming; scala**

I. Introduction

With the rapidly increasing amount of available data and parallel and distributed computing resources, Big Data Analytics has emerged as essential for business, engineering and science. Frameworks for Big Data Analytics are the great facilitators of the paradigm shift to Big Data. They handle the storage and access to massive volumes of data and well as application of numerous analytics techniques to make sense of the massive data.

The purpose of a framework supporting big data is four-fold: First, it should obviously support the storage and rapid access to large amounts of data. Second, it should support the application of parallel and distributed processing techniques to speed up the execution of analytics techniques. Third, it should hide the complexity involved in the second aspect, i.e., the APIs provided should be relatively simple and not hard to grasp, e.g., like MapReduce. Details of location, partitioning of data, creation of threads, scheduling, fault tolerance, recovery and synchronization should be hidden as much as possible. Fourth, an extensible library of analytics techniques (e.g., from statistics, data mining and machine learning) should be provided or at least mechanisms should be provided to facilitate this. Others have focused on issues such as scalability, streaming vs. batch, data integration, data compression, security and privacy [1], [2].

Of recent, several frameworks have been developed using the Scala Programming Language [3]. Scala supports both the object-oriented and functional programming paradigms, and is built on top of Java (can call and be called by Java). The Scala compiler produces JVM byte-code and runs nearly as fast as Java (although a little slower than C, both are faster than Python). Like Python, the code can be very concise and still readable, as well as very natural for expressing mathematically oriented algorithms. Furthermore, the Scala-based ecosystem for big data is large and growing. It includes Spark [4], Kafka [5], Samza [6] and Flink Scala API [7], as well as smaller projects like our SCALATION [8] project.

In this paper, we review the evolution of Big Data Frameworks and then focus on frameworks written in Scala, with illustrations from our SCALATION project. The rest of this paper is organized as follow: Section II gives an overview of the evolution of Frameworks that support Big Data Analytics, emphasizing those that are open source. In Section III, the focus narrows to Open Source Frameworks that are written in Scala. Section IV examines the support, direct or indirect, provided by frameworks for data storage and access (e.g., distributed file systems, high-performance database systems). Finally, conclusions are given in Section V.

II. Evolution of Frameworks for Big Data Analytics

One of the earliest frameworks used to implement big data analytics over distributed clusters was the Message Passing Interface (MPI) [9]. Even today, MPI remains one of the most performant big data frameworks by virtue of its extremely low overhead: nearly every aspect of a distributed program must be implemented by the user. MPI exposes language-independent primitives to the user in for synchronizing and communicating between processes. Most MPI applications are coded in C, C++, or Fortran. While MPI provides very little out-of-the-box user-facing functionality such as fault tolerance, it is still the framework of choice for users whose primary concern is speed.

978-1-5090-2622-7/16 $31.00 © 2016 IEEE
DOI 10.1109/BigDataCongress.2016.61

The MapReduce paradigm, popularized by Google from its use in the company as a proprietary technology [10] and then incorporated into the Apache Hadoop [11] project, was a major step toward simplifying the parallelization of distributed tasks. For big data problems that are "embarrassingly parallel" and have few discrete, well-defined phases of computations, the MapReduce paradigm provides a means for highly parallel execution on extremely large clusters. Some classical examples include word counting, statistics computations such as means and variances, and log parsing.

The Apache Hadoop project includes an open source MapReduce implementation, thereby putting such capabilities within the hands of many programmers. Unlike MPI, only a limited amount of specialized training is needed to create MapReduce applications that can run on Hadoop. Furthermore, Hadoop includes fault-tolerant distributed data structures out-of-the-box, making programming using Hadoop easier.

The MapReduce paradigm consists of three primary phases: map, shuffle, and reduce. In typical Hadoop MapReduce workflows, data in key-value format is read from the Hadoop Distributed File System (HDFS) by mappers and written back to HDFS prior to the shuffle phase. Once all mappers have finished, the data is deserialized and sorted over the network before being serialized back to HDFS, prior to the reduce phase. Reducers then read the sorted key-value pairs from HDFS, perform their agglomerative operations, and serialize the final results back to HDFS before returning control to the user. A full map-shuffle-reduce triad represents a single "pass" over the data; it is common for more sophisticated tasks to require multiple passes.

While MapReduce and its open source implementation in Apache Hadoop provided users with a powerful abstraction for creating scalable distributed applications with built-in fault tolerance, the earliest versions and even later versions of Hadoop possessed drawbacks. Most prevalent are the structural weaknesses inherent to the MapReduce paradigm of distributed computing: embarrassingly parallel batch algorithms can be implemented very efficiently, but iterative algorithms with complex data interdependencies create serious performance bottlenecks. For example, running PageRank until convergence on a large graph requires an unspecified number of passes over the full dataset. This bottleneck is further exacerbated in Hadoop MapReduce by the constant serialization-deserialization steps in between each phase; most of the execution time of an iterative algorithm in Hadoop MapReduce is spent performing disk I/O, or in the network shuffle. A recent overhaul of the Hadoop architecture into Hadoop 2.0 saw the introduction of Yet Another Resource Negotiator (YARN) to improve resource utilization across a cluster for a variety of workloads. Nevertheless, the fundamental limitations of MapReduce remain.

In response to these limitations, many additional next-generation distributed computing frameworks have appeared. Apache Storm [12] is similar to Hadoop, but focuses on more efficient stream processing, allowing data to be sent directly from one worker to another and is capable of processing millions of incoming tuples per second per node. Like Hadoop, it has built-in fault tolerance, and guarantees "exactly-once" processing of incoming data.

An alternative to dividing computations into mappers and reducers for iterative algorithms, is to divide computations into a series of supersteps that involve receiving input messages, performing computations and sending output messages. Synchronization is system-level, since a task must wait for all subtasks within a superstep to complete before synchronizing and moving to the next superstep. This approach is referred to as the Bulk Synchronous Parallel (BSP) [13] model of distributed computing. In cases where the number of supersteps is not too large and work is well balanced across tasks, BSP can be quite useful for implementing graph algorithms.

A special form of BSP, called vertex-centric, has become popular for big data graph analytics. In this programming model, each vertex of the graph is an independent computing unit, or task. Each vertex initially knows only about its own status and its outgoing edges. In each step, vertices can exchange messages with neighbors through successive supersteps to learn about each other. When a vertex believes that it has accomplished its tasks, it votes to halt and goes to inactive mode. When all vertices vote to halt, the algorithm terminates. Several frameworks support this style of programming, including Pregel [14], GPS [15], early GraphLab [16] architectures, and Apache Giraph [17].

One of the most popular open source big data frameworks is Apache Spark [18]. Spark provides powerful functional primitives beyond map and reduce, which coupled with in-memory serialization of intermediate data results in significant performance gains over MapReduce. Furthermore, Spark provides built-in fault tolerance in its core distributed data abstraction by tracking the lineage of computations used to derive each partition, preserving a recipe for recomputing any partitions lost to hardware failure. Apache Flink (formerly Stratosphere) is a similar framework in that it provides users with powerful functional primitives in Scala for operating on distributed data, but unlike Spark, Flink's core processing engine is streaming (like Storm). Furthermore, all operations in Flink are evaluated lazily; workflows are only executed once an underlying optimizer has established the most efficient pipeline for the data, avoiding extraneous operations. This is particularly attractive for highly iterative algorithms. This optimizer effectively decouples the user-facing API from the eventual computation.

III. Scala-Based Open Source Frameworks

As mentioned, there are a growing number of big data frameworks and associated packages implemented in Scala.

We provide an overview of a few of the more popular ones below as well as discuss our framework called SCALATION.

A. Spark

Spark is an open-source Big Data Framework [4] which was originally created at the University of California, Berkeley, and later donated to the Apache Software Foundation. The foundation for Spark is Spark Core, which is built in Scala. In addition to Scala and Java, Spark has Python and R domain specific languages (DSL) for developers to use for dispatching, scheduling, and basic I/O functions in compute tasks.

The basic abstraction for a distributed dataset in Spark is a resilient distributed dataset, or RDD [18]. Core operations, such as `map`, `reduce`, and `groupByKey` can be performed on the elements of the RDD and are either evaluated lazily (transformations) or eagerly (actions). A key property of RDDs is that they are immutable; transformations actually create new RDDs when they are executed. In this way, the lineage of each RDD is maintained throughout the life of the data structure. This lineage provides a "recipe" for recomputing each partition of the RDD, providing intrinsic fault-tolerance in the event of hardware failure.

In addition to Spark Core, there are several additional packages shipped with Spark that build on the basic RDD abstraction and provide out-of-the-box support for many different use-cases: Spark SQL, Spark Streaming, MLlib, and GraphX.

Spark SQL allows for relational processing to be done on RDDs and on external datasets [19]. The fundamental structure in Spark SQL is the DataFrame, which can be created from an RDD or from external data, but which has a much richer functional interface than a standard RDD. As of Spark 2.0, the DataFrames and Datasets APIs have been unified under the Scala and Java DSLs.

Spark Streaming [20] provides a framework using RDDs and DataFrames to handle stateful computations, which is the maintenance of a system state in the presence of streaming data. The incoming data stream is divided into mini-batches, defined using some time interval. Each mini-batch is processed as an RDD, with all the advantages and functionality an RDD provides.

MLlib is Spark's machine learning library [21]. It can be used to process RDDs or the higher level DataFrame. It includes distributed implementations of many machine learning algorithms, such as classification and clustering, as well as optimization algorithms and tools for dimensionality reduction and statistical analysis. It can be integrated with all of the other tools provided in the Spark package.

GraphX is Spark's graph computation framework [22]. GraphX exposes a variant of the Pregel API [14]; at a high level, it implements the BSP [13] graph processing model, in which vertices in the graph conduct computations in parallel before sending synchronized messages to their neighbors. Iteration halts when all vertices have signaled they have no more computations to perform. GraphX uses a variant of RDDs that relies on high level vertex and edge abstractions, though recent work has included the graph analog for DataFrames–the GraphFrame–as a key data abstraction in GraphX.

B. Kafka and Samza

Kafka and Samza were originally developed by LinkedIn, and subsequently donated to the Apache Software Foundation. Samza [6] caters to distributed processing of real time data. In Samza, computations, called jobs, take in input streams and deliver output streams. For parallel computation, each input stream is divided into partitions and a job into tasks that are assigned to machines. Each task consumes data from a partition. Thus, a job operates across multiple machines and can be highly parallel. The messaging system layer for Samza is by default Kafka.

Kafka [5] defines the notion of a topic, which is a stream of messages of a specific type, which producers create and to which consumers subscribe. The servers which are used to maintain messages related to topics are called brokers, and any number of brokers can be added at any time. There is no master broker, so adding new brokers is easy. The topics themselves can be partitioned, and brokers can hold one or more of the partitions, but partitions of a topic can be distributed among more than one broker.

The general flow of the system is that producers create messages which are pushed to brokers and assigned to topics. Consumers subscribe to topics, and pull the messages from the brokers as they are written. Consumers can also belong to consumer groups, and when a consumer group is subscribed to a topic which has a new message, only one consumer from the group consumes it. Kafka is built on the concept of the transaction log, and appends messages to the end of a partition's log. In this way, Kafka allows consumers to rewind the message stream since previous messages will still be on the log.

C. SCALATION

SCALATION is a modeling and analytics framework that supports predictive analytics [23], graph analytics [24] and simulation modeling [8]. A recent application combining the two utilizes SCALATION to analyze traffic based on large amounts of sensor data and microscopic simulation of traffic flow. Beyond modeling and analysis, this project also involves the use of optimization (also supported by SCALATION) to improve traffic flow. This is sometimes referred to as prescriptive analytics.

The framework for SCALATION current support for big data focuses on parallel and out-of-core support for big data as well as tight integration with two NoSQL database systems provided as part of SCALATION: a columnar database

system and a graph database system. For out-of-core computation, it uses memory mapped files for arrays, vectors, matrices and columnar relations (see below).

For parallel processing, SCALATION builds on features and capabilities provided by Scala [25]. The easiest way to add parallelism to sequential code by using the `par` method on ranges, as shown below in the vector multiplcation (`x * y`) method.

```
def * (b: VectoD): VectorD =
{
    val c = new VectorD (dim)
    for (i <- range.par) c.v(i) = v(i) * b(i)
    c
} // *
```

The range variable is `0 until dim` and `par` will cause the loop to be divided and threads assigned to do computations on subranges (see [25] for details).

In the case of the `dot` method that computes the inner (or dot) product, simply adding `par` to the range leads to race conditions causing indeterminate values for the result (i.e., possibly correct, but usually not).

```
def dot (b: VectoD): VectorD =
{
    var sum = 0.0
    for (i <- range) sum += v(i) * b(i)
    sum
} // dot
```

The obvious solution of synchronizing access to the `sum` variable may make the parallel implementation slower than the sequential version. An improvement would be to use Java's `DoubleAdder` for atomic updates. Even better would be support for Hardware Transactional Memory (HTN) that deals with race conditions at the hardware level. Currently, SCALATION divides the computation into subranges and assigns a `Thread` to compute subtotals and then adds these subtotals at the end.

```
def dotp (b: VectorD): Double =
{
    val nn = dim / PAR_LEVEL
    val dt = makeThreads ()
    for (i <- dt.indices) {
        dt(i) = new VecThread (this, b,
                i * nn, (i+1)*nn min dim, dot)
        dt(i).start ()
    } // for
    await ()
    dt.foldLeft (0.0)((s, x) => s + x.result)
} // dotp
```

IV. SUPPORT FOR DATA STORAGE AND ACCESS

Big data is typically stored in a distributed file system, e.g., the Hadoop Distributed File System (HDFS), or a high-performance database system. Recently, NoSQL databases [26], as an alternative to traditional relational databases, have emerged as practical ways to store big data for applications requiring greater storage and performance. NoSQL databases include key-value stores, document-oriented databases, graph databases and columnar databases [26]. In addition to providing direct built-in support for the last two, SCALATION provides some support for reading and write JavaScript Object Notation (JSON) object for interfacing with MongoDB databases.

A. Columnar Databases

In the NoSQL domain, the closest to traditional relation databases are columnar databases, e.g., C-Store [27] and HP Vertica. For analytics, storing relations in columns as opposed the ordinary row storage provides several benefits. Columns contain data of the same type, columns are large (much larger than rows) and are likely to have repetitive values. This opens up the possibility of compressing the columns. Several compression techniques are viable [27] and two have been included in SCALATION, run-length encoded columns and sparse columns. It also makes it easy for vectors and matrices to be spun off of relations and passed into various analytic techniques. For example, in the the `prodSales` relation, columns 0 to 10 are used to form a design matrix `x`, while column 11 is used to form the response vector `y`. These are passed into a General Linear Model (GLM) which includes multiple linear regression, after which the `train` method is called to produce a least squares fit.

```
val (x, y) = prodSales.toMatriDD (0 to 10, 11)
val rg = GLM (x, y)
rg.train ()
```

SCALATION provides operations of select, project, join, union, etc. so that data can be easily and efficiently extracted as desired for analysis.

B. Graph Databases

Graph databases [28] have existed for some time, (e.g., Neo4j [29] and OrientDB [30]). SCALATION provides extensive support for efficient pattern matching (e.g., Tight Simulation, DualIso for subgraph isomorphism [24]) on large graphs. We are currently in the process of adding more practical querying capabilities to these core capabilities.

V. CONCLUSIONS

This paper discusses the evolution of big data frameworks and then hightlights the recent trend toward using the Scala programming language to develop such frameworks. Overviews of popular Scala based frameworks, Spark, Samza and Kafka, are given along with our own system called SCALATION.

REFERENCES

[1] H. Hu, Y. Wen, T.-S. Chua, and X. Li, "Toward scalable systems for big data analytics: a technology tutorial," *Access, IEEE*, vol. 2, pp. 652–687, 2014.

[2] A. N. Richter, T. M. Khoshgoftaar, S. Landset, and T. Hasanin, "A multi-dimensional comparison of toolkits for machine learning with big data," in *Information Reuse and Integration (IRI), 2015 IEEE International Conference on*. IEEE, 2015, pp. 1–8.

[3] M. Odersky, L. Spoon, and B. Venners, *Programming in Scala*. Artima Inc, 2008.

[4] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," *HotCloud*, vol. 10, pp. 10–10, 2010.

[5] J. Kreps, N. Narkhede, J. Rao *et al.*, "Kafka: A distributed messaging system for log processing," in *2011 NetDB Workshop*. NetDB, 2011.

[6] M. Kleppmann and J. Kreps, "Kafka, Samza and the Unix philosophy of distributed data," *Bulletin of the IEEE CS Technical Committee on Data Engineering*, 2015.

[7] V. Markl, "Breaking the chains: On declarative data analysis and data independence in the big data era," *Proceedings of the VLDB Endowment*, vol. 7, no. 13, pp. 1730–1733, 2014.

[8] J. A. Miller, J. Han, and M. Hybinette, "Using domain specific language for modeling and simulation: Scalation as a case study," in *Simulation Conference (WSC), Proceedings of the 2010 Winter*. IEEE, 2010, pp. 741–752.

[9] W. Gropp, E. Lusk, N. Dossb, and A. Skjellumb, "A high-performance, portable implementation of the MPI message passing interface standard," *Parallel Computing*, vol. 22, no. 6, pp. 789–828, 1996.

[10] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *OSDI*, 2004.

[11] A. Bialecki, M. Cafarella, D. Cutting, and O. OMalley, "Hadoop: A framework for running applications on large clusters built of commodity hardware," Wiki at http://lucene. apache. org/hadoop, Tech. Rep., 2005.

[12] M. H. Iqbal and T. R. Soomro, "Big data analysis: Apache Storm perspective," *International Journal of Computer Trends and Technology*, pp. 9–14, Jan 2015.

[13] L. G. Valiant, "A bridging model for parallel computation," *Communications ACM*, vol. 33, no. 8, pp. 103–111, 1990.

[14] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *SIGMOD*, 2010, pp. 135–146.

[15] S. Salihoglu and J. Widom, "GPS: A graph processing system," in *SSDBM*. New York, NY, USA: ACM, 2013, pp. 22:1–22:12.

[16] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed GraphLab: a framework for machine learning and data mining in the cloud," *Proceedings of the VLDB Endowment*, vol. 5, no. 8, pp. 716–727, 2012.

[17] C. Avery, "Giraph: Large-scale graph processing infrastructure on Hadoop," *Proceedings of the Hadoop Summit. Santa Clara*, 2011.

[18] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2012, pp. 2–2.

[19] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi *et al.*, "Spark SQL: Relational data processing in Spark," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 2015, pp. 1383–1394.

[20] M. Zaharia, T. Das, H. Li, S. Shenker, and I. Stoica, "Discretized streams: An efficient and fault-tolerant model for stream processing on large clusters," in *Fourth USENIX Workshop on Hot Topics in Cloud Computing*, 2012.

[21] X. Meng, J. Bradley, B. Yavuz, E. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. Tsai, M. Amde, S. Owen *et al.*, "MLlib: Machine learning in Apache Spark," *arXiv preprint arXiv:1505.06807*, 2015.

[22] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica, "GraphX: A resilient distributed graph system on Spark," in *First International Workshop on Graph Data Management Experiences and Systems*. ACM, 2013, p. 2.

[23] J. A. Miller, M. E. Cotterell, and S. J. Buckley, "Supporting a modeling continuum in scalation: from predictive analytics to simulation modeling," in *Proceedings of the 2013 Winter Simulation Conference: Simulation: Making Decisions in a Complex World*. IEEE Press, 2013, pp. 1191–1202.

[24] J. A. Miller, L. Ramaswamy, K. J. Kochut, and A. Fard, "Research directions for big data graph analytics," in *Big Data (BigData Congress), 2015 IEEE International Congress on*. IEEE, 2015, pp. 785–794.

[25] A. Prokopec, P. Bagwell, T. Rompf, and M. Odersky, "A generic parallel collection framework," in *Euro-Par 2011 Parallel Processing*. Springer, 2011, pp. 136–147.

[26] A. Moniruzzaman and S. A. Hossain, "NoSql database: New era of databases for big data analytics-classification, characteristics and comparison," *arXiv preprint arXiv:1307.0191*, 2013.

[27] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil *et al.*, "C-Store: a column-oriented DBMS," in *Proceedings of the 31st international conference on Very large data bases*. VLDB Endowment, 2005, pp. 553–564.

[28] I. Robinson, J. Webber, and E. Eifrem, *Graph databases*. "O'Reilly Media, Inc.", 2013.

[29] J. Webber, "A programmatic introduction to NEO4j," in *Systems, Programming, and Applications: Software for Humanity*. ACM, 2012, pp. 217–218.

[30] C. Tesoriero, *Getting Started with OrientDB*. Packt Publishing Ltd, 2013.