Spark: Past, Present and Future

Baris Can Vural

I. Introduction

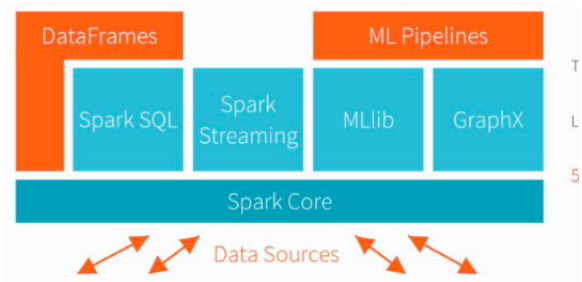
The term ”Big data” is often associated with 3 Vs: *Volume*, *Variety* and *Velocity*. *Volume* refers to the massive size of the data, *Velocity* refers to the speed at which the data is coming in and going out and lastly *Variety* refers to the wide scope of data formats and sources. These characteristics of big data makes it crucial that parallel computing be utilized if valuable information from this data is to be extracted. The volume of data to be handled cannot fit in one computer, and the computational power required to process the data is simply too large for one computer to handle. However, writing parallel programming algorithms is a cumbersome task. One must consider the challenges that come with parallel programming: Task distribution, data distribution, load balancing, fault tolerance, using data locality, etc... The task requires high technical knowledge and one could argue the cost of building these systems could outweigh the benefit one would get by extracting valuable information from big data.

With the famous map reduce paper being published in 2008[1], programmers became familiarized with the notion of automated parallelism. Two programming abstractions were introduced; *Map* and *Reduce*. In *Map*, programmers define a function that takes an input as data to be processed and produces an output that are key value pairs. In *Reduce*, programmers can use these key value pairs to generate smaller number of values. The parallelism is handled automatically by the system, which is fault tolerant, reliable and is one which exploits data locality where possible. Open source implementation of the *MapReduce* concept followed in 2011 with Apache Hadoop MapReduce[2].

In this literature study we are going to take a look at the a software called Spark, an open source cluster computing framework. [3]. Apache Spark grew to include more pieces within its ecosystem which made it the go-to cluster computing framework for big-data processing today.

II. Spark ecosystem at a glance

A large number of problems can be expressed by the *MapReduce* abstraction. Some examples include *word count* (counting how many times each word appear) *Distributed grep* (returning the lines that match a given pattern) *Inverted index* (Returning a list of document IDs in which a specific word appears). These examples can be multiplied. The way these programs are parallelized behind the scenes is the use of acyclic data flow models in which the data goes through a set of operations. There are limitations to the performance of the implementation of this concept such as loading input from disk for every individual *MapReduce* job. This creates a major bottleneck for real world problems that include iterative machine learning tasks and interactive querying of the intermediate data[3] New programming concepts were introduced by the Spark framework since its inception in 2009. At first, it appeared to tackle the performance drawbacks of Hadoop MapReduce in performing the *mapreduce* abstraction; but the Spark ecosystem grew to include a SQL module for processing intermediate data using SQL syntax, a machine learning library to construct powerful ML pipelines for analysing data and a graphx graph processing engine which leverages optimizations that pre-existing graph processing engines could not enjoy. In this section we take a glance at the Spark ecosystem as it stands today.



Spark ecosystem consists of Spark core, Spark SQL, Spark streaming engine, MLlib machine learning library, GraphX graph processing engine

A. *Resillient Distributed Datasets*

The main programming abstraction provided by the Spark framework is Resillient Distributed Datasets. A resilient distributed dataset (RDD) is a read-only collection of objects partitioned across a set of machines that can be rebuilt if a partition is lost[3]. RDDs can be viewed as a distributed shared memory concept with limited possible actions that can be performed over them. As far as the programmer is concerned, they can be regarded as global variables with limitations, which reside in the driver program. There are four ways of constructing RDDs:

- *From a shared file system such as GFS or HDFS:* Google file system (GFS) is a distributed, block-structured, fault-tolerant file system which is managed by a single master node[4]. HDFS is the open source implementation of this concept. Spark can utilize these systems to read data into the memory of the cluster nodes.
- *Parallelization of a collection:* A collection of items(arrays, lists, or other collection objects in various programming languages which are accepted by Spark) can be distributed among the nodes of the cluster by calling the parallelize() method.
- *Transforming an existing RDD:*

RDDs can be *transformed* into new RDDs by passing the elements of an RDD through a user-defined function[3]. Operations such as *flatMap*, *map* or *filter* can be applied.

- *By changing the persistence of an RDD:*
  By design, RDDs are designed as lazy structures. They are not calculated until they are used in a parallel operation[3] Programmers can call *cache()* and *save()* methods to change the persistence of these RDDs. By calling *cache()* method programmer advises Spark that the RDD in question will be used in future operations and that it should be kept in memory. However, if memory is not available, Spark will spill the RDD into disks. By calling *save()* method programmer can save an RDD in the distributed file system.

The main motivation behind RDDs is improving performance on iterative machine learning algorithms and interactive querying of the intermediate data[5]. The performance drawbacks of Hadoop MapReduce become more apparent in these kind of tasks since every iteration must be defined as a seperate MapReduce job and before every MapReduce job, input for the job must be loaded from the disks. The same goes for the output or the intermediate results, which have to be written to disks. Keeping data in memory can provide major performance improvement over reading the data from disks; as much as 10x according to [3].

The main challenge behind coming up with a distributed shared memory scheme is fault-tolerance. In order to achieve this, one must replicate the data to multiple machines and updates to data must be broadcasted to the replicates in order to ensure coherence. This approach however, does not scale well with data-intensive applications. RDDs achieve fault-tolerance by the concept of *lineage*. The operations performed on the RDDs are recorded and should any of the nodes that carry a part of the RDDs fail, only the lost part of the RDD partition is reconstructed by following the series of operations in RDD's lineage. However, only *coarse-grained* operations are allowed on RDDs, removing the possibility of having small updates being broadcasted to replicates and creating a performance bottleneck. By limiting the possible actions that can be performed, RDDs are made to scale well. Although it may seem RDDs are limited with respect to the kind of operations that can be performed; it is the case that many parallel applications are naturally defined as applying the same operation to multiple data items[5]. Another benefit of RDDs over distributed shared memory systems is that since it only allows coarse-grained operations, runtime scheduling can be done in a way that exploits data locality which reduces communication overhead among nodes and improve performance.

Spark allows a range of actions on RDDs. It makes a distinction between *transformations* and *actions*. *Transformations* create new RDDs or modify existing RDDs to produce new RDDs. *Actions* refer to operations which aim to reduce/collect the values in the RDD which is distributed among cluster nodes. Some of the more frequently used *transformations* include:

- *Map()*
  Creates a new RDD by passing each element in the source RDD through a user defined function.
- *flatmap()*
  The difference between this method and *map()* is that this function may map items to 0 or more output items. This function should return an array of values. (list in python)
- *filter()*
  Creates a new RDD by passing each element in the source RDD through a user defined function which may return true or false.

These transformations are similar to Hadoop MapReduce programming model. However, it has the added benefit of being able to create cyclic data flows out of the intermediate RDDs which are created. Changing the persistence of these intermediate results, which are results of user-defined functions which get passed to the transformations, one can significantly improve performance in comparison with Hadoop MapReduce. Hadoop MapReduce would have to do disk I/O, serialization, and data replication after each job which incur significant performance issues.

Some of the more frequently used *actions* include:

- *reduce()*
  This action aggregates the elements of an RDD in the driver program. The function passed to it takes two parameters and must return one value. The operation performed within the user defined function must have commutative and associative properties.
- *collect()* Returns elements of an RDD as a list(python term) to the driver program. The caveat is that the programmer should call this function after applying a filter in order to avoid huge volumes of data to be flooded into the driver program.
- *count()*
  Returns the number of elements in an RDD, useful in examples such as line counting.
- *take()*
  Returns a list of elements from an RDD. Useful to call before calling reduce() if the size of the RDD is relatively large.
- *saveAsTextFile()*
  Writes the elements of the RDD to a textfile, or to a distributed file system such as HDFS.

Shared Variables Spark allows the use of restricted, globally shared variable types. These serve as helpers in creating common usage patterns. The shared variables are:

- *Broadcast Variables:*
  Broadcast variables are useful when the parallel worker nodes need to access to large amounts of read-only data such as lookup tables. Spark ensures that the read-only data is copied to each worker node only once.
- *Accumulators:*
  These "add-only" variables are used to do counting in parallel.

In [6], it is argued that the implementation of broadcast variables in Spark is not made to scale well. This is due to the fact that the broadcast variables are serialized and written by the sender to a distributed file system such as HDFS. All the receivers of the broadcast variable have to read from HDFS and deserialize it to construct the variable. This creates a bottleneck at HDFS if the system is scaled up. The systems that rely heavily on broadcasting variables should come up with better strategies than using the built-in shared variables.

B. *SparkSQL*

Hadoop MapReduce offers only two programming abstractions which are *map* and *reduce*. However, many data analytics tasks can be expressed as a set of MapReduce jobs. These include SQL query, data mining, machine learning and graph processing. Hadoop is used to run ad-hoc queries on large datasets, through SQL interfaces such as Pig and Hive [3] The idea is to load the datasets to memory and query them. This creates a performance bottleneck for Hadoop MapReduce since every query has to run as a seperate mapreduce job and in the beginning of every job the data has to be loaded from the disks.

Spark's RDD abstraction helps programmers achieve great performance improvements over Hadoop MapReduce when it comes to interactive analysis tasks. The reason for this is the fact that RDDs are kept in memory and accessed much faster than disks. In-memory computing is very important in large scale data analytics for two reasons: Firstly, many machine learning algorithms and graph algorithms are iterative; they go over the data multiple times which means that in-memory access will yield much better performane. Secondly, even the traditional SQL workloads show strong temporal and spatial locality: A study of Facebook's Hive warehouse and Microsoft's Bing analytics cluster show that over 95% of queries in both systems could be served out of memory using just 64 GB/node as cache even though each system manages more than 100 PB of total data [7]

Shark[7], is a data anaysis system that does query processing in data analysis tasks. It achieves significant performance improvements over Apache Hive, a SQL querying tool for Hadoop MapReduce. It achieves this performance upgrade by implementing in-memory columnar storage and columnar compression which reduces the data size and the processing time as much as 5x over traditional Spark programs. Furthermore, there are query optimizations in place which partially executes sql queries and dynamically changing query execution strategies based on statistics. Lastly, Shark also takes advantage of the control over data partitioning which is provided by Spark.

SparkSQL[8] is built on the experience of the creators of Shark. Programmers can leverage the benefits of relational processing, and machine learning algorithms. It has two main additions two the previous systems: *a) DataFrame API,* which is a tight integration between relational and procedural processing. Programmers are able to combine relational processing provided by the API with the procedural Spark code. *b) Catalyst:* A highly extensible (e.g. easy to add data sources, optimization rules, and data types for machine learning) query optimizer leveraging the features of Scala programming language.

Authors of SparkSQL claim that it is 10x faster and more memory-efficient than naive Spark code in computations which are expressible in SQL. SparkSQL is seen as an evolution of both SQL-on-Spark and Spark itself.

Now we are going to look at two major features that seperate SparkSQL from its predecessors:

1) ***DataFrame API****:* A dataframe is a distributed collection of rows with a homogoneous schema[8] Dataframes are the main programming abstraction provided within SparkSQL. A dataframe is a *table* in traditional relational database terms. They are made to work with the RDDs, which help programmers integrate their existing Spark programs to dataframes and benefit from its performance benefits.

Programmers may create dataframes from a data source or existing RDDs. Dataframes essentially can be seen as RDD of row objects and therefore RDD transformations such as *map* can be performed on them. Dataframes are lazy structures: Dataframes represent a logical plan to compute an RDD, but execution does not take place until an output operation such as *saveToTextFile()* or *reduce()* is called. This allows for optimizations.

Basic usage of dataframes is as follows:

```
ctx = new HiveContext()
users = ctx.table("users")
young = users.where(user("age") < 21)
println(young.count())
```

In this example, users and young variables are dataframes. The user("age") expression refers to the "age" column in user table. Logical operations such as "¡" can be performed and in this example it is used as a filter in a where clause which is standard SQL syntax. The dataframes merely represent logical execution plans, i.e. they are not computed until the last line which includes count() - an output operation. There are possible optimizations behind the scenes such as only scanning the users column if the data storage is columnar[8]

SparkSQL makes use of a wide range of data sources. It supports major data types within SQL such as boolean, integer, double, decimal, string, date, and timestamp. It also allows for complex user-defined data types such as JSON or native Java/Python objects.

SparkSQL offers some benefits over working with a traditional relational database systems. Since it is integrated in a programming language, programmers may use the structures within the language to break up SQL statements, add control flow structures, or pass dataframes between functions. It also provides eager error reporting: It checks small errors such as column name typos in advance and provides a faster error reporting as opposed to running big queries in traditional relational database systems.

Another benefit of Spark SQL is that it offers even better in-memory caching than standard Spark. Columnar compression

schemes are used which result in performance upgrades by an order of magnitude[8].

Lastly, SparkSQL allows programmers to use User Defined Functions or UDFs. User may specify their custom function and use that function in the SQL queries.

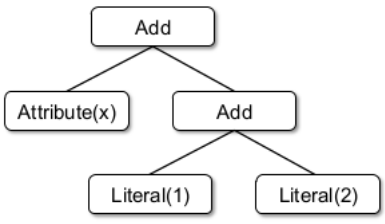
```
def squared(s):
  return s * s
sqlContext.udf.register("
    squaredWithPython", squared)

#We may use the function in queries:

from pyspark.sql.functions import udf
squared_udf = udf(squared, LongType())
df = sqlContext.table("test")
display(df.select("id", squared_udf("id")
    .alias("id_squared")))
```

In this example[9] we define and register a function called squared which squares two numbers. After this point, we can refer to this function in our SQL queries. In the dataframe called "test", we search for ids and display the squared values of those ids. Note that this custom function was written in the programmer's chosen programming language and not in a domain specific language like one would in a relational database system. This makes it straightforward to customize queries in SparkSQL.

*2) **Catalyst optimizer**:* Catalyst optimizer is the SQL optimization part of SparkSQL. It has an extensible design with two purposes: Make it easy to add new optimization techniques and features, and make it easy to extend the optimizer by specifying data-specific rules. Catalyst supports both rule-based and cost-based optimization.

Catalyst provides the possibility of extension of optimizer by exploiting the features of the Scala programming language such as pattern matching. This helps reduce the learning curve required to come up with optimization rules. The systems which require domain-specific language to generate optimization code suffer from the learning curve. SparkSQL keeps things simple in this regard.

In catalyst optimizer, main data type is a *tree*, with nodes being the name of the operations, or the values that go inside these operations. This representation is used to help achieve optimization. An example tree looks like:



The tree above is the representation of the expression: x+(1+2). In code, this would be represented as Add(x, Add(1,2))

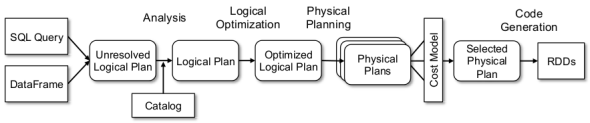
Rules are functions that take as input a tree and returns a transformed tree. These rules can be arbitrary operations on the tree but mainly they are defined as patterns that transform a subtree to another. In scala language an example is:

```
tree.transform{
case Add(Literal(c1),Literal(c2))=>
    Literal(c1+c2)
}
```

Applying this rule to the tree above, we end up with a tree of structure (x+3). The rule in this function is a partial function. A partial function of type PartialFunction[A, B] is a unary function where the domain does not necessarily include all values of type A[10]. Catalyst will look only at the subtrees that a given rule is applicable to. This allows for adding one rule after another without modifying the previously added rules.

Catalyst executes *rules* or tree transformations repetitively until a fixed point. Rules sometimes modify the trees and this makes the parts of the tree applicable to other rules. One set of rules may assign types to variables while other set of rules may transform the tree in a different way.

Catalyst optimization is consisted of 4 stages: Analysis, logical optimization, physical planning and code generation.



Analysis

In analyis, Catalyst starts with a dataframe and resolves the references and relations mentioned in the SQL statement. It resolves the types of the variables mentioned in the SQL and checks the relations by looking up the tables in question. It finally constructs a logical plan to execute the SQL statement so that further optimizations based on rules and costs can be applied.

Logical Optimization In logical optimization, rule based optimizations such as predicate pushdown, constant folding, boolean expression simplification and null-propagation are performed

Physical Planning In this phase the optimizer generates one or more physical plans out of the logical plans. Physical plan is expressing the logical plan in terms of the Spark's execution engine. The optimizer selects a physical plan based on a cost model. Rule based optimizations are also performed in this phase; such as pipelining projections or filters into one Spark map operation.

Code Generation In this phase the bytecode to be run on all cluster nodes is generated. The designers went for code generation because they wanted to speed up execution since spark works on in-memory data which is CPU bounded. In code generation, the "quasiquotes" feature in Scala language is leveraged[11].

C. Spark Streaming

So far, we only looked at the batch processing features of Spark. As of 2012, Apache Spark framework also provides a module for processing streaming data.

Processing streaming data pertains to *Velocity* of Big Data; the speed at which the data arrives for processing. There are well-known systems that process data such as TimeStream, MapReduce Online and Storm. These systems can be used for various tasks such as:

- *Provide site statistics*
  A distributed near-time system that provides statistics about visitors' activity such as clicking on ads. Facebook's *Puma* is one exmample.
- *Cluster monitoring*
  A distributed system is error prone by its nature. There are systems where log file of large distributed systems are processed as streaming data. Apache Flume is one example.[12]
- *Machine learning in real time*
  Predictive statistical models can be run on streaming data to get results as the data is coming in. One useful example is real-time spam detection. Twitter for example, can identify spam/scam tweets in real time using stream processing.

The stream data processing systems, when scaled up, face two major problems: *Fault recovery* and *stragglers(slow nodes)*. Overcoming these problems is crucial for real-time systems as slow recovery or slow nodes might cost the system its time window to make a key decision. The stream data processing systems such as TimeStream, MapReduce Online and Storm are based on *continuous operator model*, which implies the following: There are operators that receive incoming stream data. They process the data, update their internal state and they output the modified stream to another operator. Although this model design seems natural when one thinks about processing streaming data, this model makes it difficult for the system to recover from failures or respond well to slow nodes. These systems recover from errors by following two approaches:*Replication*, where there are two copies of each operator node, or *upstream backup* where nodes store every sent message and send them to a newly created recovery node. Neither approach is preferable as *replication* means doubling the hardware costs and *upstream backup* incurs large recovery time. Also, neither approach handles the case of slow nodes.

In Spark, the approach to streaming data is quite different than the pre-existing systems. The concept of *discretized systems* or *D-streams* is introduced. It essentially means taking batches of streaming input data and processing them as batch jobs in one time-step. The idea is to exploit the fault-tolerant, easily recovering nature of RDDs in stream processing. In each time-step the input streaming data is taken as RDD and this provides two major benefits: 1- Stateless, deterministic computation in each time-step; which can only be achieved in the *continuous operator model* systems with the help of synchronization algorithms.(wait until nodes synchronize) 2-easier analysis of dependencies between RDDs, allowing optimizations such as starting a job in the next time interval before the current time interval is not complete.

There are two challenges to D-streams. Firstly, discretizing the input stream incurs a minimum latency. However, in [13] it is argued that this minimum latency is acceptable in real-life scenarios. Secondly, recovering from faults is made easy by recovering in paralell. Here, Spark's existing recovery mechanism is leveraged. This is difficult to achive in *continuous operator model* systems because of the complex synchronization algorithms to ensure determinism.

Spark streaming is said to be a performance upgrade over *Storm*[14] by 2x to 5x[13]. It also has the benefit of being able to combine batch jobs and streaming jobs as in Spark system both task types are using RDDs.

An example of using strema processing in Spark is as follows:

```
pageViews = readStream("http://...","1s")
ones = pageViews.map(event=>(event.url,1)
    )
counts = ones.runningReduce((a,b)=>a+b)
```

In the example, pageViews is an RDD created by the input stream and the second parameter to the readStream function is the time interval for each discretized step. "ones" is an RDD which are (url, 1) pairs and the last line runs a reduce operation on keys, which finally yields the (url, number of visits) pairs.

Spark streaming allows programmers to define input streams from outside sources or loading it from a file system such as HDFS. It also allows *stateless* transformations on the data such as *map*, *reduce*, *groupBy*. These operations are stateless in the sense that their processing does not remember what happened in the previous discretized step. Spark streaming also allows for stateful transformations across time-steps such as ***1-Windowing:*** Programmers can define time intervals to access the RDDs of the specified time interval;***2-Incremental aggregation:*** where group operations such as reduce can be performed per time interval provided; ***3-State tracking:*** where programmers would like to get notified when certain events take place such as socket connection from client and socket close by the client if, for example, the programmer wants to keep track of a user's statistics when they watch a video.

D. MLlib

The combinination of machine learning algorithms in big data analytics has become very important. In its evolution, spark ecosystem grew also to have its own machine learning library called MLlib. This library has features such as[15]:

- **ML Algorithms:** common learning algorithms such as classification, regression, clustering, and collaborative filtering
- **Featurization:** feature extraction, transformation, dimensionality reduction, and selection
- **Pipelines:** tools for constructing, evaluating, and tuning ML Pipelines

- **Persistence:** saving and load algorithms, models, and Pipelines
- **Utilities:** linear algebra, statistics, data handling, etc.

The MLlib library provides many benefits to the developers. It exploits the fact that Spark keeps the data in memory; the iterative nature of machine learning algorithms result in huge performance boosts because of this. The library also provides sophisticated implementations of the most popular learning algorithms; the complexity of having to implement these algorithms and potential errors that could arise is avoided if one uses this library.

The MLlib library is huge and it would take too long to mention all of its features here. However, to demonstrate how easy to use it is, here is an example of a logistic regression example in spark:

```
data = [LabeledPoint(1.0,[1.0,0.0]),
    LabeledPoint(1.0,[1.0,0.0]]
lrm = LogisticRegressionWithLBFGS.train(
    sc.parallelize(data),iterations=10)
lrm.predict([1.0,0.0])#returns 1
lrm.predict([0.0,1.0])#returns 0
```

In the example above, ”data” represents our training data set. Our training data consists of labeled points which are of structure: outcome,feature vector. Next, we can train our logistic regression model using this data. We have to pass the model an RDD which we create by parallelizing the array we have in the driver program. Spark also supports reading training data in other formats such as libsvm. Next and finally we can ask our model the prediction outcome of a given feature vector. Since we ask exactly what we gave in the training data, the model returns the expected values. This is a trivial example to show the user-friendlienss of the library.

MLlib library reduces the developer's development time greatly by providing very sophisticated algorithms like the one mentioned above. Normally, if one is to implement logistic regression, one would go about doing this using parameter learning using *gradient descent* since this is the most standard and easy way to solve the problem. In the example above, the algorithm uses the technique called *LBFGS*, which is a great deal more sophisticated than gradient descent and has better performance, and it is available for Spark developers without them knowing how it exactly works.

MLlib used to use RDDs as its standard data abstraction but now, the standard is to use dataframes instead. This allows for SQL optimizations on the data and allows *pipelining* which is a popular concept in machine learning workflows. Here is an example of dataframes and pipelines in Apache Spark MLlib:

```
sentenceData = spark.createDataFrame([
    (0.0,"Hi␣I␣heard␣about␣Spark"),
    (0.0,"I␣wish␣Java␣could␣use␣case␣
        classes"),
    (1.0,"Logistic␣regression␣models␣are␣
        neat")
],["label","sentence"])

tokenizer=Tokenizer(inputCol="sentence",
    outputCol="words")
wordsData=tokenizer.transform(
    sentenceData)

hashingTF=HashingTF(inputCol="words",
    outputCol="rawFeatures",numFeatures
    =20)
featurizedData = hashingTF.transform(
    wordsData)

idf=IDF(inputCol="rawFeatures",outputCol=
    "features")
idfModel=idf.fit(featurizedData)
rescaledData=idfModel.transform(
    featurizedData)

rescaledData.select("label","features").
    show()
```

The above example computes IDF values in a given corpus. ”sentenceData” is a dataframe with columns ”label” and ”sentence”. Next, we can use MLlib's tokenizer and give it an input and output column. By giving the exact column names as we use in our dataframe, we construct our pipeline. The functions IDF and HashingTF follow similar structure. One output column is followed by the next MLlib library function's input column. Using this structure, one can construct powerful machine learning data flows.

E. GraphX

In the past, the graph processing systems like Pregel [16] or PowerGraph [17], were preferred over distributed dataflow frameworks such as Hadoop MapReduce to increase performance. These systems provided graph programming abstractions which accelerated iterative graph algorithms. Early on, distributed dataflow frameworks relied on single computation and disk storage, which limited the performance of graph algorithms. Also, these frameworks did not allow fine-grained control over the partitioned distributed data.

The graph processing systems provided optimizations that the distributed dataflow frameworks could not because they had a narrower focus. Implementing a graph processing system using operations such as map, reduce, groupby proved to be challenging and these operations did not take into considerations the optimizations that could be derived from the graph structure. Furthermore, most of the time graph processing is only a part of a data analytics task. Data analytics task requires graph computation as well as working with structured or semi-structured data. This complexity resulted in performance drawbacks.

GraphX, is the graph processing system within Spark. It is a variant of the graph processing system Pregel[16]. Developments in the Spark framework made it advantageous for Spark

to have its own graph processing engine. These developments include control over the partitioning of the data in memory, and low cost fault recovery through the concept of RDDs. The existing graph processing systems relied on recovery based on snapshotting: saving graph state as a checkpoint.

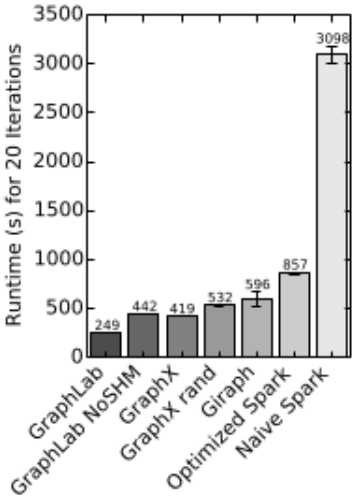
Graph processing systems represent graph structured data as a *property graph*, which associates user-defined properties with each vertex and edge[18]. The term *triplet* refers to the Vertex-edge-Vertex structure in graphs. These are represented as collections in Spark, which makes room for optimizations that are not available in graph processing systems.

GraphX provides some programming abstractions to Spark in addition to the existing dataflow operators. Some of these are:

- **mrTriplets** "Map reduce triplets" is the operator which comprises map and a groupby operations on the triplets. The map function is applied to each triplet and the result is aggregated to the destination vertex.
- **edges** Returns the edges in a graph as a collection
- **vertices** Returns the vertices in a graph as a collection
- **triplets** Returns the triplets in a graph as a collection
- **mapV** Applies a map function to all vertices and returns a graph
- **mapE** Applies a map function to all edges and returns a graph

Essentially graphX treats graph representations as horizontally partitioned collections and it performs the join-map-groupBy operations to achieve typical graph processing operations.

GraphX scales well against other graph processing engines. In [18], graphx is benchmarked against other graph processing engines such as giraph.



We can observe that a graph processing engine such as giraph outperforms naive spark and optimized spark in this pagerank algorithm. However, graphx engine seems to outperform giraph with a little margin.

III. Evolution of Spark and What is next

Spark was built in 2009 at UC Berkeley and open-sourced in 2014[19]. It quickly grew from a research project to the world's biggest open source project in cluster computing. Spark has a large ecosystem today: SQL querying, machine learning, graph analytics, stream analytics... Spark's rich language support as well as these tools make it a very good choice for big-data analysis.

Early on, Spark's main attraction was RDDs. RDDs are fault-tolerant collection of replicated data that sits in-memory of the computers in the cluster. Data engineers, however, required more tools for data analytics such as SQL or iterative machine learning algorithms.

Spark introduced the concept of dataframes within its integration with the SQL engines for interactive analysis of data. Machine learning algorithm library MLlib was added to make sense of this data with standardized methods. D-streams allow the use of all of Spark's features from an input stream. Graphx engine likewise can be utilized to process graphs in a fault-tolerant manner

Spark offers many different features under one framework. Big-data analytics engineers do not have to switch to other programs. This helps with productivity and uniformity among other professionals.

In spark 1.6, the concept of datasets was introduced as an experimental api:

Datasets The Apache Spark Dataset API provides a type-safe, object-oriented programming interface. In other words, in Spark 2.0 DataFrame and Datasets are unified[20]. It provides compile-time type safety while preserving the optimizations of the Catalyst optimizer.

Datasets are an etenstion to the dataframe API. They are created through the sqlcontext object just like dataframes. Datasets make use of Tungsten's[1] in-memory encoding. Datasets api uses the dataframe query planner to achieve more optimizations. This is achieved by converting expressions and fields into java bytecode on the fly.

A small example of datasets looks like this(In scala):

```
val df=sqlContext.read.json("people.json")
case class Person(name:String, age:Long)
val ds: Dataset[Person]=df.as[Person]
```

Datasets make sense in type-safe languages like scala or java. In python, variable types are inferred therefore datasets are not supported in python.

IV. Conclusions

Spark has major performance benefits over other distributed dataflow frameworks. The concept of RDDs as in-memory data sets distributed over cluster nodes speeds up big-data computation especially in two scenarios:

[1]Tungsten helps Spark keep objects out of memory. It also makes use of columnar storage to save space.

- *Iterative jobs* where the intermediate results are being reused such as in some machine learning operations such as gradient descent. In Hadoop MapReduce, each intermediate result would have to be written to disk and reread in the beginnning of the next MapReduce job. This incurs a huge performance bottleneck.
- *Interactive analytics* where a dataset of huge volume can be loaded into the memory of the cluster nodes to be queried in reasonable response times. The RDD abstraction

At first spark's main selling point was RDDs, but the ecosystem grew to include other common data-analytics task libaries and now these structures are started to being replaced by concepts such as dataframes and datasets. Using Spark, one can query big-data interactively in-memory using SQL syntax, do graph computations, apply machine learning algorithms, and achieve all of this using streams as well as distributed file systems such as hdfs.

Spark ecosystem is constantly evolving to include the commonplace data analytics task of data engineers. Judging by its evolution in only 2 years, it is safe to say that it will grow bigger and be better suited to the needs of its users.

REFERENCES

[1] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[2] "Apache hadoop," https://en.wikipedia.org/wiki/Apache$_H adoop$.

[3] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets." *HotCloud*, vol. 10, no. 10-10, p. 95, 2010.

[4] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *ACM SIGOPS operating systems review*, vol. 37, no. 5. ACM, 2003, pp. 29–43.

[5] 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.

[6] M. Chowdhury, "Performance and scalability of broadcast in spark," 2014.

[7] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica, "Shark: Sql and rich analytics at scale," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of data*. ACM, 2013, pp. 13–24.

[8] 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.

[9] "Call the udf in spark sql," https://docs.databricks.com/spark/latest/spark-sql/udf-in-python.html.

[10] "Partial functions," https://www.scala-lang.org/api/current/scala/PartialFunction.html.

[11] "Quasiquotes," http://docs.scala-lang.org/overviews/quasiquotes/intro.html.

[12] "Apache flume," https://flume.apache.org/.

[13] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: Fault-tolerant streaming computation at scale," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 2013, pp. 423–438.

[14] A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham *et al.*, "Storm@ twitter," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM, 2014, pp. 147–156.

[15] "Machine learning library," http://spark.apache.org/docs/latest/ml-guide.html.

[16] 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 *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 2010, pp. 135–146.

[17] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs." in *OSDI*, vol. 12, no. 1, 2012, p. 2.

[18] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "Graphx: Graph processing in a distributed dataflow framework." in *OSDI*, vol. 14, 2014, pp. 599–613.

[19] "Apache spark," https://en.wikipedia.org/wiki/Apache$_S park$.

[20] "Datasets," https://databricks.com/product/getting-started-guide/datasets.