Contents

[1. What is the difference between spark and Hadoop? 1](#_Toc474089270)

[2. What are the differences between functional and imperative languages, and why is functional programming important? 2](#_Toc474089271)

[3. What is a resilient distributed dataset (RDD), Explain showing diagrams? ​ ​ 3](#_Toc474089272)

[4. Explain transformations and actions (in the context of RDDs) 4](#_Toc474089273)

[5. What are the Spark use cases? 4](#_Toc474089274)

[6. Why do we need transformations? What is lazy evaluation and why is it useful? 4](#_Toc474089275)

[7. What is ParallelCollectionRDD? 4](#_Toc474089276)

[8. Can you use Spark to access and analyze data stored in Cassandra databases? 4](#_Toc474089277)

[9. Explain how ReduceByKey and GroupByKey works? 5](#_Toc474089278)

[10. What is the common workflow of a Spark program? 5](#_Toc474089279)

[11. Explain spark environment for driver. Ref 6](#_Toc474089280)

[12. What are the transformations and actions that you have used in Spark? 6](#_Toc474089281)

[13. How can you minimize data transfers when working with Spark? 8](#_Toc474089282)

[14. What is a lineage graph? 8](#_Toc474089283)

[15. Describe the major libraries that constitute the Spark Ecosystem 9](#_Toc474089284)

[16. What are the different file formats that can be used in SparkSql? 9](#_Toc474089285)

[17. What are Pair RDDs? 9](#_Toc474089286)

[18. What is the difference between persist() and cache() 9](#_Toc474089287)

[19. What are the various levels of persistence in Apache Spark? Ref 10](#_Toc474089288)

[20. Which Storage Level to Choose? Ref 10](#_Toc474089289)

[21. Explain advantages and drawbacks of RRD 11](#_Toc474089290)

[22. Explain why dataset is preferred over RDDs? 11](#_Toc474089291)

[23. How to share data from Spark RDD between two applications? 11](#_Toc474089292)

[24. Does Apache Spark provide check pointing? 12](#_Toc474089293)

[25. Explain the internal working of caching? 12](#_Toc474089294)

[26. What is the function of Block manager? 12](#_Toc474089295)

[27. Why Spark SQL considers the support of indexes unimportant? 12](#_Toc474089296)

[28. How to convert existing UDTFs in Hive to Scala functions and use it from Spark SQL? Explain with example Ref 12](#_Toc474089297)

[29. Why use dataframes and datasets when we have RDD? Ref Video 13](#_Toc474089298)

[30. What is a Catalyst and how does it work? Ref 14](#_Toc474089299)

[31. What are the top challenges developers faces while writing Spark applications? – Ref Video 17](#_Toc474089300)

[32. How is memory handled in Datasets? 20](#_Toc474089301)

[33. Explain Datasets and Serialization 20](#_Toc474089302)

[34. What are the limitations of dataset? 20](#_Toc474089303)

[35. Implementation difference between DataFrames and DataSet? 20](#_Toc474089304)

[36. What are the contentions with memory? 21](#_Toc474089305)

[37. Show Command to run spark in YARN client mode? 22](#_Toc474089306)

[38. Show Command to run spark in YARN cluster mode? 22](#_Toc474089307)

[39. What is Standalone and YARN mode? 22](#_Toc474089308)

[40. Explain client mode and cluster mode in spark? 22](#_Toc474089309)

[41. Which cluster managers are supported by spark? 23](#_Toc474089310)

[42. What is Executor memory? 23](#_Toc474089311)

[43. What is DStream and What is the difference between batch and Dstream in Spark streaming? 23](#_Toc474089312)

[44. How does Spark Streaming work? 23](#_Toc474089313)

[45. Difference between map() and flatMap()? 24](#_Toc474089314)

[46. What is reduce() action, Is there any difference between reduce() and reduceByKey()? 25](#_Toc474089315)

[47. What is the disadvantage of Reduce() action and how can we overcome this limitation? 25](#_Toc474089316)

[48. What are Accumulators and When are accumulators truly reliable? 25](#_Toc474089317)

[49. What is Broadcast Variables and what is the advantage? 26](#_Toc474089318)

[50. What is Piping, show an example who to create data pipeline? 26](#_Toc474089319)

[51. What is driver? 28](#_Toc474089320)

[52. What are the steps that occur when you run a Spark application on a cluster? 28](#_Toc474089321)

[53. What is a schema RDD/DataFrame? 29](#_Toc474089322)

[54. What are Row objects? 29](#_Toc474089323)

[55. How does Spark achieve fault tolerance? 29](#_Toc474089324)

[56. What does a Spark Engine do? 29](#_Toc474089325)

[57. What parameter is set if cores need to be defined across executors? 29](#_Toc474089326)

[58. Name few Spark Master system properties? 29](#_Toc474089327)

[59. Define Partitions in reference to spark implementation? 30](#_Toc474089328)

[60. Differences between how Spark and MapReduce manage cluster resources under YARN. Ref 30](#_Toc474089329)

[61. What is Hive on Spark? 34](#_Toc474089330)

[62. What is GraphX and what is PageRank? Ref 34](#_Toc474089331)

[63. What does MLlib do? Ref 40](#_Toc474089332)

[64. What is a Parquet file? 45](#_Toc474089333)

[65. Why Parquet is used for Spark SQL? Ref 45](#_Toc474089334)

[66. What is schema evolution and what is its disadvantage, explain schema merging in reference to parquet file? Ref 48](#_Toc474089335)

[67. What file systems does Spark support? 51](#_Toc474089336)

[68. Will spark replace MapReduce? 51](#_Toc474089337)

[69. What is Spark Executor? 51](#_Toc474089338)

[70. Name the different types of Cluster Managers in Spark. 51](#_Toc474089339)

[71. How many ways we can create RDD, show example? 51](#_Toc474089340)

[72. How do you flatten rows in Spark? Explain with example. Ref 52](#_Toc474089341)

[73. Explain Spark Streaming Architecture? 52](#_Toc474089342)

[74. What are the types of Transformations on DStreams? 52](#_Toc474089343)

[75. What is Receiver in Spark Streaming, can you build custom receivers? 53](#_Toc474089344)

[76. Explain the process of Live streaming and how to store DStream data to database? Ref 53](#_Toc474089345)

[78. How spark streaming is fault tolerant? 56](#_Toc474089346)

[79. Explain transform() method used in dSteam? Ref 57](#_Toc474089347)

[80. How data security is achieved in Spark? 58](#_Toc474089348)

[81. Explain Kerberos security? Ref 61](#_Toc474089349)

[82. Name various types of distributing does spark support? 62](#_Toc474089350)

[83. Consider the following SQL query: Select pName, sum(rate) from sales where (year = 2015) group by pName; Explain the functioning in Map and Reduce when the above query is executed in spark using dataSets. 62](#_Toc474089351)

[84. What are the conditions where spark driver can parallelize dataSets as RDDs? 62](#_Toc474089352)

[85. Can repartition() operation decrease number of partition? 62](#_Toc474089353)

[86. What is the drawback of repartition() and coalesce() operations? 62](#_Toc474089354)

[87. In a join operaton for example val joinVal = rddA.join(rddB) will it generate partition? 62](#_Toc474089355)

[88. Consider the following code in spark, what is the final value in fVal variable? 62](#_Toc474089356)

[89. Scala pattern matching - Show various ways code can be written? 62](#_Toc474089357)

[90. What is the return result when query is executed using Spark SQL or HIVE? Hint: RDD or dataframe/dataset? 63](#_Toc474089358)

[91. If we want to display just the schema of a dataframe/dataset what method is called? 63](#_Toc474089359)

[92. Show various implementations for the following query in spark? 63](#_Toc474089360)

[93. What are the most important factors you want to consider when you start machine learning project? 63](#_Toc474089361)

[94. As a data scientist which algorithm will you suggest if legal aspects and ease of explanation to non technical people are the main criteria? 64](#_Toc474089362)

[95. For supervised learning algorithm, what percentage of data is split between training and test dataset? 64](#_Toc474089363)

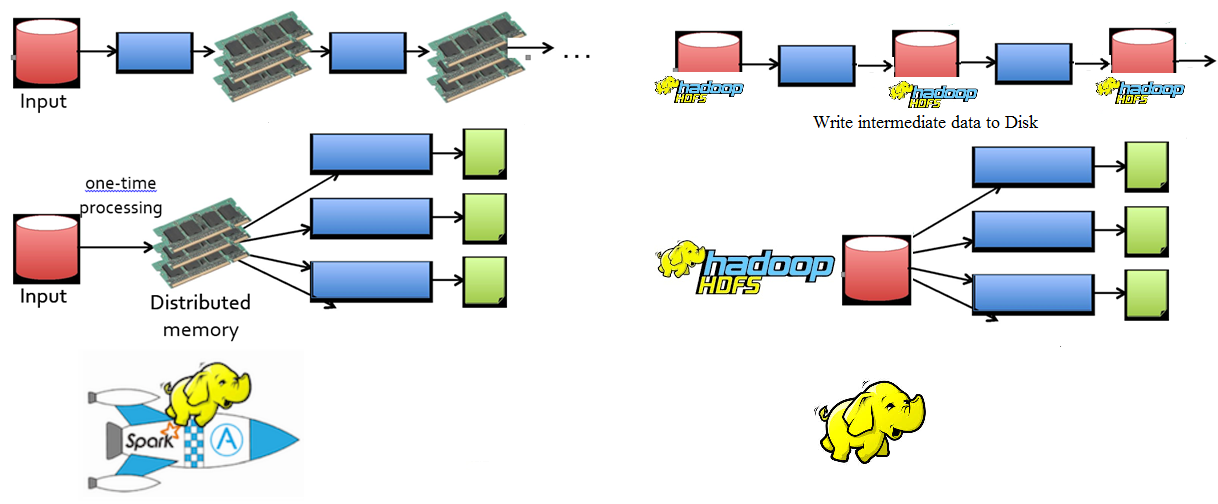
[96. Compare performance of Avro and parquet file formats, explain case while using with spark. 64](#_Toc474089364)

[97. Spark Master exposes a set of REST API’s to submit, monitor applications which data format is used for these web services? 64](#_Toc474089365)

[98. When you should not use spark? 64](#_Toc474089366)

[99. What attributes contribute for a function to achieve parallelism? 64](#_Toc474089367)

# What is the difference between spark and Hadoop?



|  |  |  |
| --- | --- | --- |
| **Features** | **SPARK** | **Hadoop** |
| **Inspiration** | * Hadoop Map-Reduce and Scala programming language, developed by UC-Berkeley's AMPLab in 2009, use Generalized computation instead of MapReduce * Real time processing capability | * Google, papers in 2004 outlining MapReduce * Batch Processing |
| **Speed** | * 100X in-memory and * 10X on Disk | * Heavy Disk read I/O intensive |
| **Easy of Use** | * Easily write application using Java, Scala, Python, R (Functional programming style) * Interactive Shell available with Scala and Python * High level simple map-reduce Operations | * Java – Imperative programming style * No shell * complex map-reduce operations |
| **Iterative Workflow** | * Great at Iterative workloads(Machine learning ..etc) | * Not ideal for iterative work |
| **Tools** | * Well integrated tools (Spark SQL, Streaming, Mlib and GraphX) to develop complex analytical application | * Loosely coupled large set of tools, but matured |
| **Deployment** | * Hadoop YARN, Mesos, Amazon-EC2 | * Usually use Oozie and Azkaban to create workflow |
| **Data Source** | * HDFS(Hadoop), HBase, Cassandra, MongoDB, Amazon-S3, RDBMS, file, socket, tweeter | * RDBMS (using sqoop), streaming using FLUME |
| **Applications** | * Spark ‘Application’ is higher level of Unit, runs multiple jobs in sequence or parallel * Application process are called executors, runs on clusters(workers) | * Hadoop ‘job’ is higher level unit process data with map reduce and writes data to storage |
| **Executors** | * Executors can run multiple tasks in a single processor | * Each mapReduce runs in its own processor |
| **Shuffle** | * Doesn’t till unless the number of partitions is above the configured threshold (200 by default) | * Always sorts its partition during shuffle |
| **Shared Variable** | * Broadcast variables: Read-only(look-up) variable, ships only once to worker * Accumulators: Workers add values and driver reads the data, and fault tolerant | * Hadoop counter have additional (system ) metric counters like ‘Map input records’ |
| **Persisting/Caching RDD** | * Cached RDDs can be used & reused in across the operation, thus increase the processing speed | * None |
| **Lazy Evaluation** | * Transformation functions execution plan bundled together and executes only with RDD action function | * None |
| **Memory Management and Compression** | * Memory is conserved, because of the compact format. Speed is improved by custom code-generation. | * Custom compression can be using AVRO, Kyro, no memory management |
| **Optimizer and Query Planning** | * Optimizer is a Rule Executor for logical plans. It uses a collection of logical plan optimizations. Generate encoders via runtime code-generation. The generated code can operate directly on the Tungsten compact format. Query is optimized – logical and physical plan (inspired by RDBMS query planning and optimization) | * None |

# What are the differences between functional and imperative languages, and why is functional programming important?

Following features of Scala makes it uniquely suitable for spark.

**Immutability** - Immutable means that you can't change your variables; you mark them as final in Java, or use the val keyword in Scala

**Higher order functions** - These are functions that take other functions as parameters, or whose result is a function. Here is a function apply which takes another function f and a value v and applies function f to v: example - def apply(f: Int => String, v: Int) = f(v)

**Lazy loading** - Lazy val is executed when it is accessed the first time else no execution.

**Pattern matching** - Scala has a built-in general pattern matching mechanism. It allows to match on any sort of data with a first-match policy

**Currying** - If we turn this into a function object that we can assign or pass around, the signature of that function looks like this: val sizeConstraintFn: IntPairPred => Int => Email => Boolean = sizeConstraint \_ Such a chain of one-parameter functions is called a curried function

**Partial application** - When applying the function, you do not pass in arguments for all of the parameters defined by the function, but only for some of them, leaving the remaining ones blank. What you get back is a new function whose parameter list only contains those parameters from the original function that were left blank.

**Monads** - Most Scala collections are monadic, and operating on them using map and flatMap operations, or using for-comprehensions is referred to as monadic-style.

**Programming approach difference**:

|  |  |  |
| --- | --- | --- |
| **Characteristic** | **Imperative approach** | **Functional approach** |
| Programmer focus | How to perform tasks (algorithms) and how to track changes in state. | What information is desired and what transformations are required. |
| State changes | Important. | Non-existent. |
| Order of execution | Important. | Low importance. |
| Primary flow control | Loops, conditionals, and function (method) calls. | Function calls, including recursion. |
| Primary manipulation unit | Instances of structures or classes. | Functions as first-class objects and data collections. |

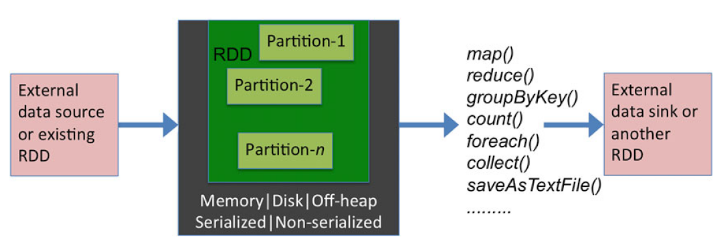
# What is a resilient distributed dataset (RDD), Explain showing diagrams? ​ ​

**Resilient distributed dataset (RDD)** is a read-only and fault-tolerant collection of objects partitioned across cluster of computers that can be operated on in parallel. There are two ways to create RDDs: parallelizing an existing collection in your driver program, or referencing a dataset in an external storage system, such as a shared filesystem, HDFS, HBase, S3, Cassandra or RDBMS.

RDDs (Resilient Distributed Datasets) are basic abstraction in Apache Spark that represent the data coming into the system in object format. RDDs are used for in-memory computations on large clusters, in a fault tolerant manner. RDDs are read-only portioned, collection of records, that are –

* Immutable – RDDs cannot be altered.
* Resilient – If a node holding the partition fails the other node takes the data.
* Lazy evaluated
* Cacheable
* Type inferred



[Ref](https://www.amazon.com/Pro-Spark-Streaming-Real-Time-Analytics/dp/1484214803)

# Explain transformations and actions (in the context of RDDs)

Transformations are functions executed on demand, to produce a new RDD. All transformations are followed by actions. Some examples of transformations include map, filter and reduceByKey.

**ReduceByKey** merges the values for each key using an associative and commutative reduce function. This will also perform the merging locally on each mapper before sending results to a reducer, similarly to a "combiner" in MapReduce.

Actions are the results of RDD computations or transformations. After an action is performed, the data from RDD moves back to the local machine. Some examples of actions include reduce, collect, first, and take.

# What are the Spark use cases?

* Data integration and ETL
* Interactive analytics or business intelligence
* High performance batch computation
* Machine learning and advanced analytics
* Real-time stream processing

Lots of people are doing data integration and ETL on MapReduce, as well as batch computation, machine learning and batch analytics. But these things are going to be much faster on Spark. Interactive analytics and BI are possible on Spark, and the same goes for real-time stream processing.

# Why do we need transformations? What is lazy evaluation and why is it useful?

As we know RDD's are immutable, thus if we have to change the format (map) or filter we need to do transformation on existing RDDs. For transformations, Spark adds them to a DAG of computation and only when driver requests some data, does this DAG actually gets executed. Advantage is that Spark can make many optimization decisions after it had a chance to look at the DAG in entirety. This would not be possible if it executed everything as soon as it got it. Lazy evaluation means that if you tell Spark to operate on a set of data, it listens to what you ask it to do, writes down some shorthand for it (blue print) so it doesn’t forget, and then does absolutely nothing. It will continue to do nothing, until you ask it for the final answer (Action). Thus saving time and processing power.

# What is ParallelCollectionRDD?

RDD is an abstract class with many subclasses and this is one of them. In general each subclass performs different action on the RDD, e.g. reading/writing/shuffling/checkpointing etc. This specific type is used when calling SparkContext.parallelize, it is used to parallelize a collection from the driver program.

# Can you use Spark to access and analyze data stored in Cassandra databases?

Yes, it is possible if you use Spark Cassandra Connector.

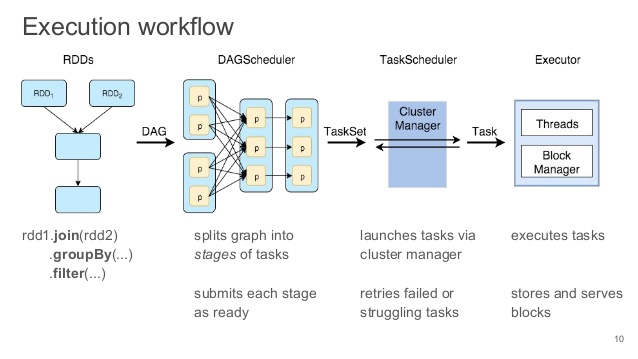
# Explain how ReduceByKey and GroupByKey works?

**ReduceByKey:** Uses CombineByKey() method behind the scene to minimize shuffle, thus efficient else they work their functioning is same. The Shuffle is an expensive operation since it involves disk I/O, data serialization, and network I/O.

|  |  |
| --- | --- |
| **ReduceByKey** | **GroupByKey** |
|  |  |

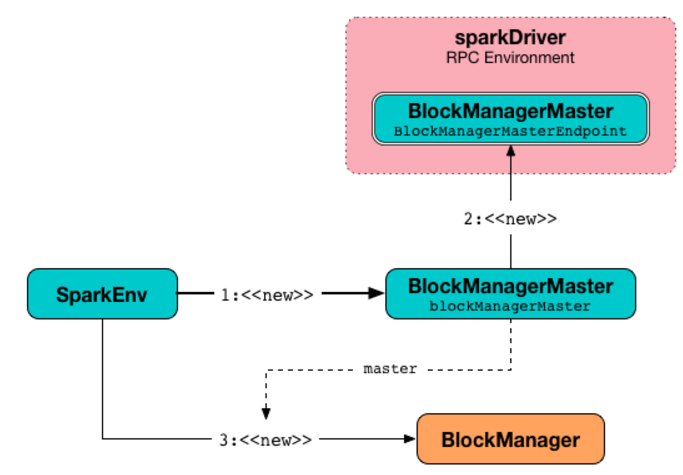
# What is the common workflow of a Spark program?

Every Spark program and shell session will work as follows:  
1. Create some input RDDs from external data.  
2. Transform them to define new RDDs using transformations like filter().  
3. Ask Spark to persist() any intermediate RDDs that will need to be reused.  
4. Launch actions such as count() and first() to kick off a parallel computation, which is then optimized and executed by Spark.

[Ref](http://www.slideshare.net/akirillov/spark-workshop-internals-architecture-and-coding-59035491)

# Explain spark environment for driver. [Ref](https://jaceklaskowski.gitbooks.io/mastering-apache-spark/content/spark-sparkenv.html)

createDriverEnv creates a SparkEnv execution environment for the driver. It accepts an instance of SparkConf, whether it runs in local mode or not, LiveListenerBus, the number of cores to use for execution in local mode or 0 otherwise, and a OutputCommitCoordinator (default: none). createDriverEnv ensures that spark.driver.host and spark.driver.port settings are defined. It then passes the call straight on to the create helper method (with driver executor id, isDriver enabled, and the input parameters).

****

# What are the transformations and actions that you have used in Spark?

|  |  |
| --- | --- |
| **Transformation** | **Meaning** |
| **map**(*func*) | Return a new distributed dataset formed by passing each element of the source through a function *func*. |
| **filter**(*func*) | Return a new dataset formed by selecting those elements of the source on which *func* returns true. |
| **flatMap**(*func*) | Similar to map, but each input item can be mapped to 0 or more output items (so *func* should return a Seq rather than a single item). |
| **mapPartitions**(*func*) | Similar to map, but runs separately on each partition (block) of the RDD, so *func* must be of type Iterator<T> => Iterator<U> when running on an RDD of type T. |
| **mapPartitionsWithIndex**(*func*) | Similar to mapPartitions, but also provides *func* with an integer value representing the index of the partition, so *func* must be of type (Int, Iterator<T>) => Iterator<U> when running on an RDD of type T. |
| **sample**(*withReplacement*, *fraction*, *seed*) | Sample a fraction *fraction* of the data, with or without replacement, using a given random number generator seed. |
| **union**(*otherDataset*) | Return a new dataset that contains the union of the elements in the source dataset and the argument. |
| **intersection**(*otherDataset*) | Return a new RDD that contains the intersection of elements in the source dataset and the argument. |
| **distinct**([*numTasks*])) | Return a new dataset that contains the distinct elements of the source dataset. |
| **groupByKey**([*numTasks*]) | When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs.  **Note:** If you are grouping in order to perform an aggregation (such as a sum or average) over each key, using reduceByKey or aggregateByKey will yield much better performance.  **Note:** By default, the level of parallelism in the output depends on the number of partitions of the parent RDD. You can pass an optional numTasks argument to set a different number of tasks. |
| **reduceByKey**(*func*, [*numTasks*]) | When called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function *func*, which must be of type (V,V) => V. Like in groupByKey, the number of reduce tasks is configurable through an optional second argument. |
| **aggregateByKey**(*zeroValue*)(*seqOp*, *combOp*, [*numTasks*]) | When called on a dataset of (K, V) pairs, returns a dataset of (K, U) pairs where the values for each key are aggregated using the given combine functions and a neutral "zero" value. Allows an aggregated value type that is different than the input value type, while avoiding unnecessary allocations. Like in groupByKey, the number of reduce tasks is configurable through an optional second argument. |
| **sortByKey**([*ascending*], [*numTasks*]) | When called on a dataset of (K, V) pairs where K implements Ordered, returns a dataset of (K, V) pairs sorted by keys in ascending or descending order, as specified in the boolean ascending argument. |
| **join**(*otherDataset*, [*numTasks*]) | When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key. Outer joins are supported through leftOuterJoin, rightOuterJoin, and fullOuterJoin. |
| **cogroup**(*otherDataset*, [*numTasks*]) | When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (Iterable<V>, Iterable<W>)) tuples. This operation is also called groupWith. |
| **cartesian**(*otherDataset*) | When called on datasets of types T and U, returns a dataset of (T, U) pairs (all pairs of elements). |
| **pipe**(*command*, *[envVars]*) | Pipe each partition of the RDD through a shell command, e.g. a Perl or bash script. RDD elements are written to the process's stdin and lines output to its stdout are returned as an RDD of strings. |
| **coalesce**(*numPartitions*) | Decrease the number of partitions in the RDD to numPartitions. Useful for running operations more efficiently after filtering down a large dataset. |
| **repartition**(*numPartitions*) | Reshuffle the data in the RDD randomly to create either more or fewer partitions and balance it across them. This always shuffles all data over the network. |
| **repartitionAndSortWithinPartitions**(*partitioner*) | Repartition the RDD according to the given partitioner and, within each resulting partition, sort records by their keys. This is more efficient than calling repartition and then sorting within each partition because it can push the sorting down into the shuffle machinery. |

|  |  |
| --- | --- |
| **Action** | **Meaning** |
| **reduce**(*func*) | Aggregate the elements of the dataset using a function *func* (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel. |
| **collect**() | Return all the elements of the dataset as an array at the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data. |
| **count**() | Return the number of elements in the dataset. |
| **first**() | Return the first element of the dataset (similar to take(1)). |
| **take**(*n*) | Return an array with the first *n* elements of the dataset. |
| **takeOrdered**(*n*, *[ordering]*) | Return the first *n* elements of the RDD using either their natural order or a custom comparator. |
| **saveAsTextFile**(*path*) | Write the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem, HDFS or any other Hadoop-supported file system. Spark will call toString on each element to convert it to a line of text in the file. |
| **saveAsSequenceFile**(*path*)  (Java and Scala) | Write the elements of the dataset as a Hadoop SequenceFile in a given path in the local filesystem, HDFS or any other Hadoop-supported file system. This is available on RDDs of key-value pairs that implement Hadoop's Writable interface. In Scala, it is also available on types that are implicitly convertible to Writable (Spark includes conversions for basic types like Int, Double, String, etc). |
| **saveAsObjectFile**(*path*)  (Java and Scala) | Write the elements of the dataset in a simple format using Java serialization, which can then be loaded usingSparkContext.objectFile(). |
| **countByKey**() | Only available on RDDs of type (K, V). Returns a hashmap of (K, Int) pairs with the count of each key. |
| **foreach**(*func*) | Run a function *func* on each element of the dataset. This is usually done for side effects such as updating an [Accumulator](http://spark.apache.org/docs/latest/programming-guide.html#accumulators) or interacting with external storage systems. |

# How can you minimize data transfers when working with Spark?

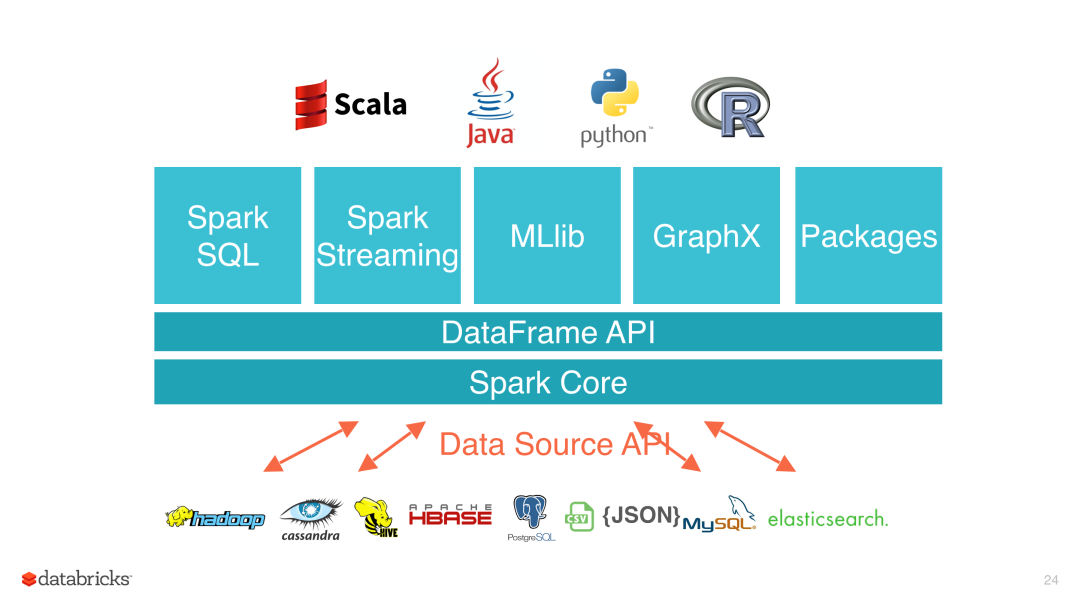
Minimizing data transfers and avoiding shuffling helps write spark programs that run in a fast and reliable manner. The various ways in which data transfers can be minimized when working with Apache Spark are:

* Using Broadcast Variable- Broadcast variable enhances the efficiency of joins between small and large RDDs.
* Using Accumulators – Accumulators help update the values of variables in parallel while executing.
* The most common way is to avoid operations ByKey, repartition or any other operations which trigger shuffles.

# What is a lineage graph?

RDD Lineage (aka RDD operator graph or RDD dependency graph) is a graph of all the parent RDDs of a RDD. It is built as a result of applying transformations to the RDD and creates a logical execution plan. The RDDs in Spark, depend on one or more other RDDs. The representation of dependencies in between RDDs is known as the lineage graph. Lineage graph information is used to compute each RDD on demand, so that whenever a part of persistent RDD is lost, the data that is lost can be recovered using the lineage graph information.

# Describe the major libraries that constitute the Spark Ecosystem



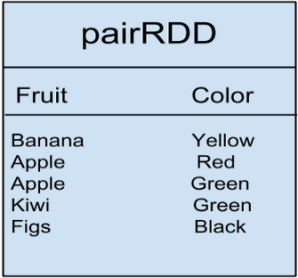
* Spark MLib- Machine learning library in Spark for commonly used learning algorithms like clustering, regression, classification, etc.
* Spark Streaming – This library is used to process real time streaming data.
* Spark GraphX – Spark API for graph parallel computations with basic operators like joinVertices, subgraph, aggregateMessages, etc.
* Spark SQL – Helps execute SQL like queries on Spark data using standard visualization or BI tools.

# What are the different file formats that can be used in SparkSql?

Parquet, text and csv, sequence files etc. Connected spark sql with RDBMS and Hive tables

# What are Pair RDDs?

Special operations can be performed on RDDs in Spark using key/value pairs and such RDDs are referred to as Pair RDDs. Pair RDDs allow users to access each key in parallel. They have a reduceByKey () method that collects data based on each key and a join () method that combines different RDDs together, based on the elements having the same key. For example, the data in the table, it represents the corresponding relationship between fruit and color:



# What is the difference between persist() and cache()

persist () allows the user to specify the storage level whereas cache () uses the default storage level.

* Cache internally uses persist API
* Persist sets a specific storage level for a given RDD
* Spark context tracks persistent RDD
* When first evaluates, partition will be put into memory by block manager

# What are the various levels of persistence in Apache Spark? [Ref](http://spark.apache.org/docs/latest/programming-guide.html)

Apache Spark automatically persists the intermediary data from various shuffle operations, however it is often suggested that users call persist () method on the RDD in case they plan to reuse it. Spark has various persistence levels to store the RDDs on disk or in memory or as a combination of both with different replication levels.

The various storage/persistence levels in Spark are -

|  |  |
| --- | --- |
| **Storage Level** | **Meaning** |
| MEMORY\_ONLY | Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, some partitions will not be cached and will be recomputed on the fly each time they're needed. This is the default level. |
| MEMORY\_AND\_DISK | Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, store the partitions that don't fit on disk, and read them from there when they're needed. |
| MEMORY\_ONLY\_SER  (Java and Scala) | Store RDD as *serialized* Java objects (one byte array per partition). This is generally more space-efficient than deserialized objects, especially when using a [fast serializer](http://spark.apache.org/docs/latest/tuning.html), but more CPU-intensive to read. |
| MEMORY\_AND\_DISK\_SER  (Java and Scala) | Similar to MEMORY\_ONLY\_SER, but spill partitions that don't fit in memory to disk instead of recomputing them on the fly each time they're needed. |
| DISK\_ONLY | Store the RDD partitions only on disk. |
| MEMORY\_ONLY\_2, MEMORY\_AND\_DISK\_2, etc. | Same as the levels above, but replicate each partition on two cluster nodes. |
| OFF\_HEAP (experimental) | Similar to MEMORY\_ONLY\_SER, but store the data in [off-heap memory](http://spark.apache.org/docs/latest/configuration.html#memory-management). This requires off-heap memory to be enabled. |

# Which Storage Level to Choose? [Ref](http://spark.apache.org/docs/latest/programming-guide.html)

Spark’s storage levels are meant to provide different trade-offs between memory usage and CPU efficiency. We recommend going through the following process to select one:

* If your RDDs fit comfortably with the default storage level (MEMORY\_ONLY), leave them that way. This is the most CPU-efficient option, allowing operations on the RDDs to run as fast as possible.
* If not, try using MEMORY\_ONLY\_SER and [selecting a fast serialization library](http://spark.apache.org/docs/latest/tuning.html) to make the objects much more space-efficient, but still reasonably fast to access. (Java and Scala)
* Don’t spill to disk unless the functions that computed your datasets are expensive, or they filter a large amount of the data. Otherwise, re-computing a partition may be as fast as reading it from disk.
* Use the replicated storage levels if you want fast fault recovery (e.g. if using Spark to serve requests from a web application). All the storage levels provide full fault tolerance by re-computing lost data, but the replicated ones let you continue running tasks on the RDD without waiting to re-compute a lost partition.

# Explain advantages and drawbacks of RRD

• Users’ functions are black boxes

• Opaque computation

• Opaque data type

• Programs built using RDD API have total control on how to execute every data operation

• Developers have to write efficient programs for different kinds of workloads

Not an easy way to write efficient programs in RDD

# Explain why dataset is preferred over RDDs?

RDDs are type-safe. However, they're also low-level, and they suffer from some problems, including:

They express the how of a solution better than the What.

They cannot be optimized by Spark.

They're slow on non-JVM languages like Python.

It's too easy to build an inefficient RDD transformation chain.

The DataFrame API provides a higher-level abstraction, allowing you to use a query language to manipulate data. In fact, you can use SQL, as well.

Much easier it is to read and simple to understand.

# How to share data from Spark RDD between two applications?

Suppose I want to generate reports on live data thus I have one job1: Spark Sliding window Streaming App, will be consuming data at regular intervals and creating RDD and dos not want to persist to storage. Now the second job Job 2: Query job that will access the same RDD created in job 1 and generate reports.

|  |  |
| --- | --- |
| You can share RDDs across different applications using Apache Ignite. Apache ignite provides an abstraction to share the RDDs through which applications can access the RDDs corresponding to different applications. In addition Ignite has the support for SQL indexes, where as native Spark doesn't. | **Spark Shared RDDs:** Apache Ignite provides an implementation of Spark RDD abstraction which allows to easily share state in memory across multiple Spark jobs, either within the same application or between different Spark applications.  IgniteRDD is implemented is as a view over a distributed Ignite cache, which may be deployed either within the Spark job executing process, or on a Spark worker, or in its own cluster.  *//Transformations*  *val sharedRdd = igniteContext.fromCache("partitioned")*    *// Store pairs of integers from 1 to 10000 into in-memory cache*  *// named "partitioned" using 10 parallel store operations.*  *sharedRdd.savePairs(sparkContext.parallelize(1 to 10000, 10).map(i => (i, i)))*  *//SQL Query*  *val sharedRdd = igniteContext.fromCache("partitioned")*    *val result = sharedRdd.sql(*  *"select \_val from Integer where val > ? and val < ?", 10, 100)* |

# Does Apache Spark provide check pointing?

Lineage graphs are always useful to recover RDDs from a failure but this is generally time consuming if the RDDs have long lineage chains. Spark has an API for check pointing i.e. a REPLICATE flag to persist. However, the decision on which data to checkpoint - is decided by the user. Checkpoints are useful when the lineage graphs are long and have wide dependencies.

# Explain the internal working of caching?

* Partition iterator checks the storage level
* If Storage level is set it calls

cacheManager.getOrCompute(partition)

* As iterator is runs for each RDD evaluation, its transparent to user

# What is the function of Block manager?

* + Handles all in memory data in spark
  + Responsible for
    - Cached Data (BlockRDD)
    - Shuffle Data
    - Broadcast data
  + Partition will be stored in Block with id (RDD.id, partition\_index)

# Why Spark SQL considers the support of indexes unimportant?

The fundamental reason why indexing over external data sources is not in the Spark scope is that Spark is not a data management system, high level data layout achieved by proper partitioning combined with columnar storage and compression can provide very efficient distributed access without an overhead of creating, storing and maintaining indexes. Refer to apache [ignite](https://ignite.apache.org/features/igniterdd.html).

# How to convert existing UDTFs in Hive to Scala functions and use it from Spark SQL? Explain with example [Ref](http://stackoverflow.com/questions/36543374/i-want-to-convert-all-my-existing-udtfs-in-hive-to-scala-functions-and-use-it-fr)

Sample Hive Table:

name id

["jack","thomas","joshua"] 151

["william","daniel","matthew"] 152

1. Creating a scala function :

def toUpper(name: Seq[String]) = (name.map(a => a.toUpperCase)).toSeq

1. Registering function as UDF : sqlContext.udf.register("toUpper",toUpper \_)
2. Calling the UDF using sqlContext and storing output as DataFrame object :

var df = sqlContext.sql("SELECT toUpper(name) FROM namelist").toDF("Name")

1. Exploding the DataFrame : df.explode(df("Name")){case org.apache.spark.sql.Row(arr: Seq[String]) => arr.toSeq.map(v => Tuple1(v))}.drop(df("Name")).withColumnRenamed("\_1","Name").show

Result:

+--------------+

| Name|

+--------------+

| JACK

THOMAS

JOSHUA

WILLIAM

DANIEL

MATTHEW|

+--------------+

# Why use dataframes and datasets when we have RDD? Ref [Video](https://www.youtube.com/watch?v=pZQsDloGB4w)

But making dataframes we've lost type safety, we do not want too may run time errors. What happens if we call the collect() action?

scala> :type df.collect()

Array[org.apache.spark.sql.Row]

Unfortunately, Row isn't typesafe. It's defined as trait Row extends Serializable. Mapping it back to something useful is ugly and error-prone:

df.collect().map { row =>

val project = row(0).asInstanceOf[String] // Yuck.

val numRequests = row(1).asInstanceOf[Long] // Yuck.

}

Main aim for the creation of dataset - We'd like to get back our compile-time type safety, without giving up the optimizations Catalyst can provide us. Datasets are:

* An extension to the DataFrame API
* Conceptually similar to RDDs. (You can use lambdas and types again.)
* Use Tungsten's fast in-memory encoding (as opposed to JVM objects or serialized objects on the heap)
* Expose expressions and fields to the DataFrame query planner, where the optimizer can use them to make decisions. (This can't happen with RDDs.)
* Interoperate more easily with the DataFrame API
* Like an RDD, Dataset has a type.

// Read a DataFrame from a JSON file

val df = sqlContext.read.json("people.json")

// Convert the data to a domain object.

case class Person(name: String, age: Long)

val ds: Dataset[Person] = df.as[Person]

* A DataFrame is just a Dataset[Row].
* Datasets: A bit of both
* With Datasets, you can still access a DataFrame-like query API. (You can also go back and forth between DataFrames and Datasets.)

**RDDs**:

val lines = sc.textFile("hdfs://path/to/some/ebook.txt")

val words = lines.flatMap(\_.split("""\s+""")).filter(\_.nonEmpty)

val counts = words.groupBy(\_.toLowerCase).map { case (w, all) => (w, all.size) }

**Datasets:**

val lines = sqlContext.read.text("hdfs://path/to/some/ebook.txt").as[String]

val words = lines.flatMap(\_.split("""\s+""")).filter(\_.nonEmpty)

val counts = words.groupBy(\_.toLowerCase).count()

// RDD

val counts = words.groupBy(\_.toLowerCase).map { case (w, all) => (w, all.size) }

// Dataset

val counts = words.groupBy(\_.toLowerCase).count()

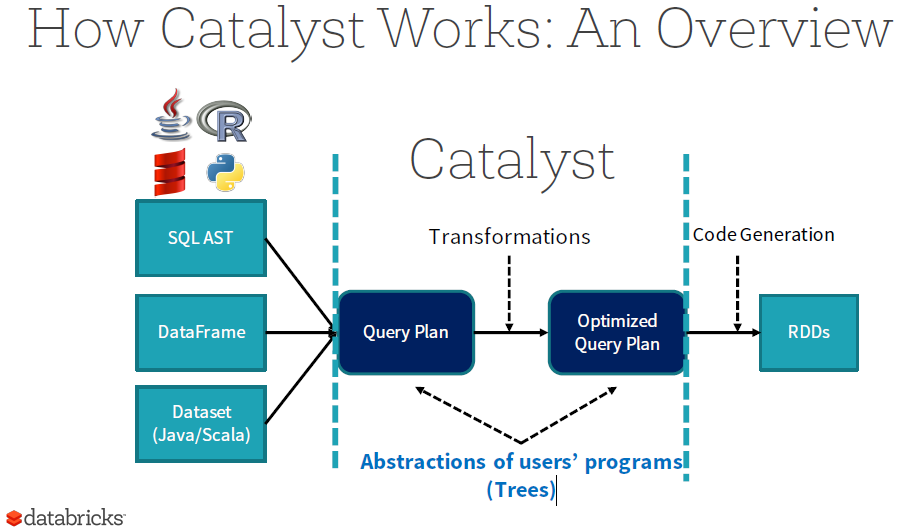
The Dataset version can use the built-in DataFrame-like count() aggregator function.

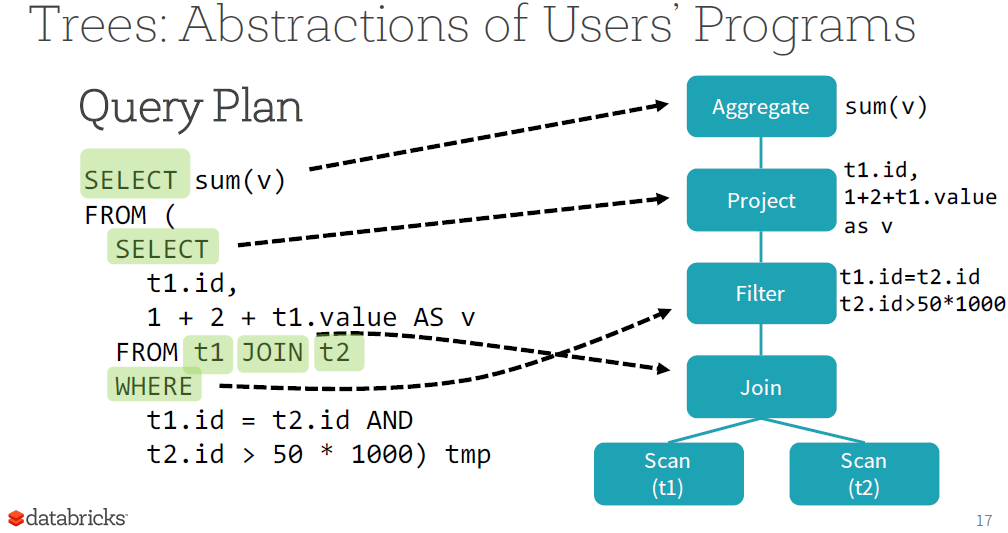
The Dataset code is more visually compact (less typing! yay!) and will tend to

execute faster than the RDD counterpart.

# What is a Catalyst and how does it work? [Ref](https://databricks.com/blog/2015/04/13/deep-dive-into-spark-sqls-catalyst-optimizer.html)

Catalyst is a collection of UDF written is scala. Catalyst contains a general library for representing trees and applying rules to manipulate them. On top of this framework, we have built libraries specific to relational query processing (e.g., expressions, logical query plans), and several sets of rules that handle different phases of query execution: analysis, logical optimization, physical planning, and code generation to compile parts of queries to Java bytecode. For the latter, we use another Scala feature, quasiquotes, that makes it easy to generate code at runtime from composable expressions. Finally, Catalyst offers several public extension points, including external data sources and user-defined types.





**Logical Plan**

• A Logical Plan describes computation on datasets without defining how to conduct the computation

• output: a list of attributes generated by this Logical Plan, e.g. [id, v]

• constraints: a set of invariants about the rows generated by this plan, e.g. t2.id > 50 \* 1000

**Physical Plan**

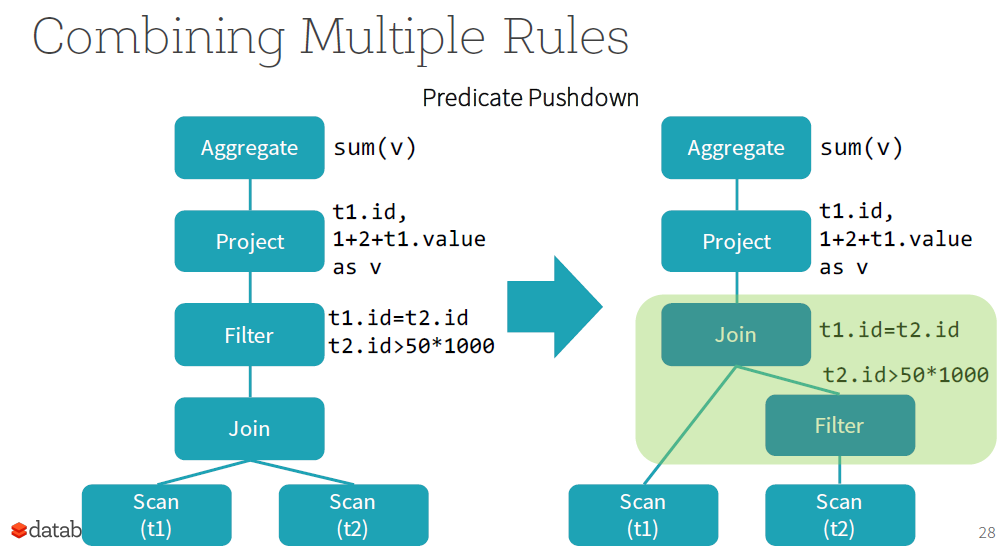
• A Physical Plan describes computation on datasets with specific definitions on how to conduct the computation

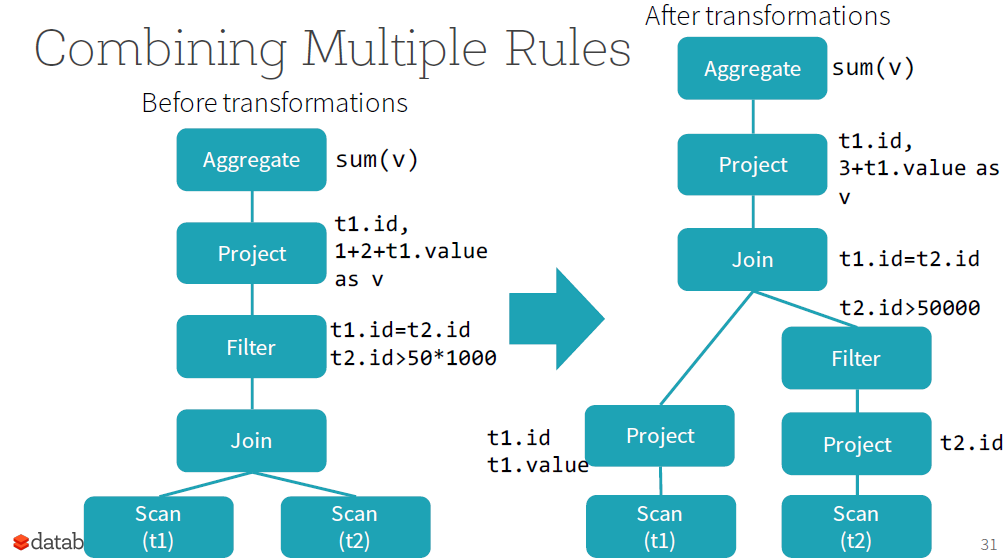
• A Physical Plan is executable

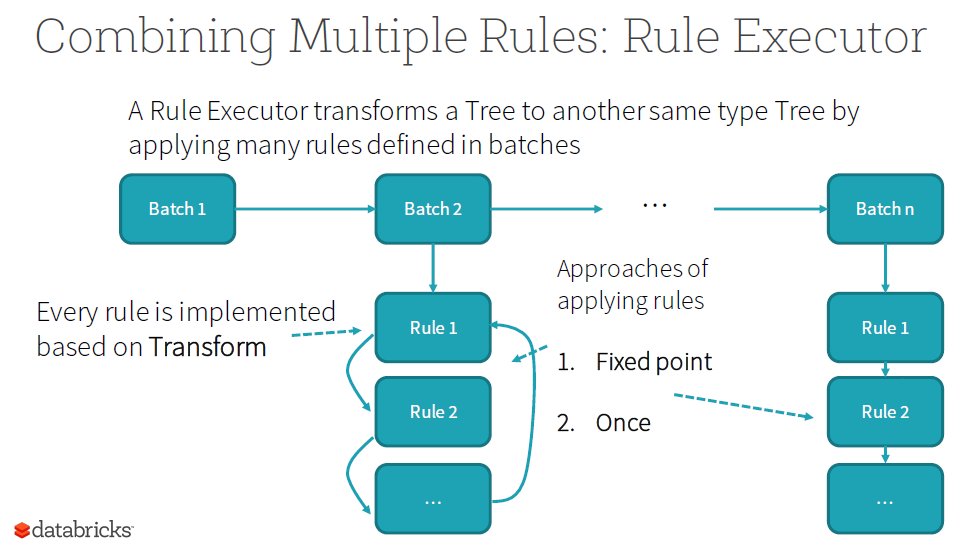
|  |  |
| --- | --- |
| **Logical Plan** | **Physical Plan** |
|  |  |

**Rules :** Trees can be manipulated using rules, which are functions from a tree to another tree. While a rule can run arbitrary code on its input tree (given that this tree is just a Scala object), the most common approach is to use a set of pattern matching functions that find and replace subtrees with a specific structure.

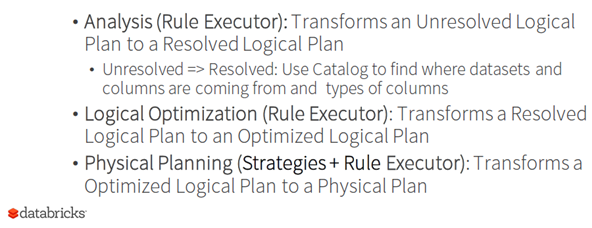
Pattern matching is a feature of many functional languages that allows extracting values from potentially nested structures of algebraic data types. In Catalyst, trees offer a transform method that applies a pattern matching function recursively on all nodes of the tree, transforming the ones that match each pattern to a result.











Spark’s Planner

• 1st Phase: Transforms the Logical Plan to the Physical Plan using Strategies

• 2nd Phase: Use a Rule Executor to make the Physical

Plan ready for execution

• Prepare Scalar sub-queries

• Ensure requirements on input rows

• Apply physical optimizations

# What are the top challenges developers faces while writing Spark applications? – Ref [Video](https://www.youtube.com/watch?v=WyfHUNnMutg)

* 1. Miss allocation of resources that is how much memory and cores should be allocated for executors which results in slow performance. As a rule of thumb use about 30% of the memory and about 1/3 of total cores for executor.
  2. One task is taking too long and other tasks finish quickly or Slow jobs on Join/Shuffle

Skewed data results in skewed partition which ultimately results in skewed RDD. What should we do?

RePartition, so that keys are distributed

* We replicate the data in the small rdd N times by creating a new key (original\_key, v) where v takes values between 0 and N. The value does not change, i.e. it is the same value that was associated to the original key.
* We take the large skewed rdd and modify the key to add some randomness (called salting) by doing (original\_key, random\_int) where random\_int takes a value between 0 and N. Note that in this case we are NOT replicating the data in the large rdd. We are simply splitting the keys so that values associated to the same original key are now split into N buckets.
* Finally, we perform the join between these datasets.
* We remove the random\_int from the key to have the final result of the join.
  1. Application failure: java.lang.IllegalArgumentException: Size exceeds Integer.MAX\_VALUE. Why because No Spark shuffle block can be greater than 2 GB. So this issue arrise when you shuffle and cache Especially problematic for Spark SQL as Default number of partitions to use when doing shuffles is 200. This low number of partitions leads to high shuffle block size specially when you join the data in spark sql. So what should we do?
     1. Increase the number of partitions Thereby, reducing the average partition size

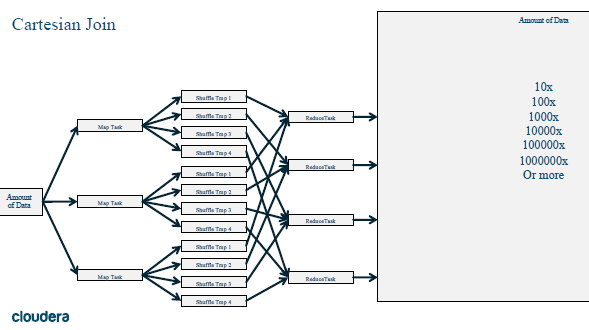
• In Spark SQL, increase the value of spark.sql.shuffle.partitions (Rule of thumb is around 128 MB per partition).If #partitions < 2000, but close, bump to just > 2000 (Spark uses a different data structure for bookkeeping during shuffles, when the number of partitions is less than 2000, vs. more than 2000.)

• In regular Spark applications, use rdd.repartition()

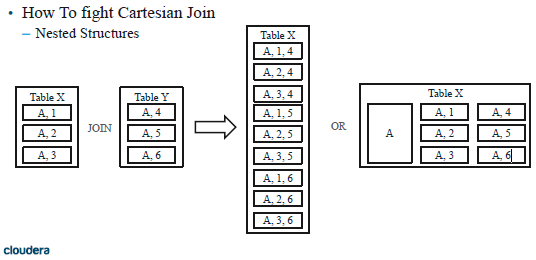
* + 1. Get rid of skew in your data. Example: If the total sales of AAPL stock is 20 of the total sales, AAPL data is skewed. Thus adding few random key helps to create distribued keys which resolves the problem of skew data. Aggregate the keys at the end say AAPL0, AAPL1, AAPL2 to AAPL.

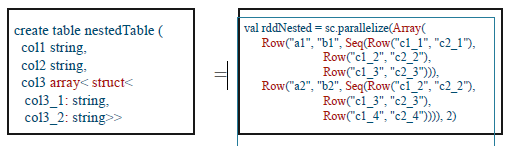
Another example join key is NULLs in one of the keys. You can do high level salting in this case by filtering Out Isolated Keys and use Map Join/Aggregate can avoid this problem.

* 1. Cartesian Join - Managing Parallelism



* + 1. Solution – Nested Structures – - Good article - [Read](http://www.congiu.com/creating-nested-data-parquet-in-spark-sql/) how to do it





* 1. Can you ever run out of memory? Can you ever have more than 20 stages? How much work is your driver doing?
     1. DAG Management mistakes

• Shuffles are to be avoided

• ReduceByKey over GroupByKey

• TreeReduce over Reduce

• Use Complex/Nested Types

Shuffles

• Map Side reduction, where possible

• Think about partitioning/bucketing ahead of time

• Do as much as possible with a single shuffle

• Only send what you have to send

• Avoid Skew and Cartesians

ReduceByKey over GroupByKey

• ReduceByKey can do almost anything that GroupByKey can do

• Aggregations

• Windowing

• Use memory

• But you have more control

• ReduceByKey has a fixed limit of Memory requirements

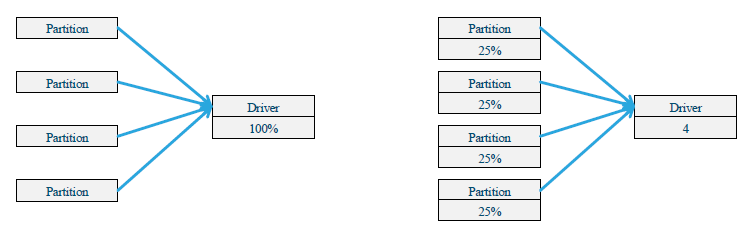
• GroupByKey is unbound and dependent on data

TreeReduce over Reduce

• TreeReduce & Reduce return some result to driver

• TreeReduce does more work on the executors

• While Reduce bring everything back to the driver



Complex Types

• Think outside of the box use objects to reduce by

• (Make something simple)

* 1. Exception in thread "main" java.lang.NoSuchMethodError: com.google.common.hash.HashFunction.hashInt(I)Lcom/google/common/hash/HashCode;

This is because you are using multiple versions of guava jars in the classpath or My protobuf version doesn’t match with Spark’s **protobuf version!** Or say you are trying to use a library outside of spark with spark

Solution: dump your classpath or fix your dependencies

**Apache Maven Shade Plugin**

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-shade-plugin</artifactId>

<version>2.2</version>

...

<relocations>

**<relocation>**

**<pattern>com.google.protobuf</pattern>**

**<shadedPattern>com.company.my.protobuf</shadedPattern>**

**</relocation>**

</relocations>

* 1. Unit Test: You do not have to test your code on your cluster, if your code works on your unit test (except very few conditions) [Video](https://www.youtube.com/watch?v=4U9Me6shpno) similar [code](https://github.com/tmalaska/spark.mergesort.example/blob/master/src/main/scala/com/cloudera/sa/spark/mergesort/example/BucketSplitter.scala)

Use code similar to this:

val sc:SparkContext = if (runLocal) {

val sparkConfig = new SparkConf()

sparkConfig.set("spark.broadcast.compress", "false")

sparkConfig.set("spark.shuffle.compress", "false")

sparkConfig.set("spark.shuffle.spill.compress", "false")

new SparkContext("local[2]", "DataGenerator", sparkConfig)

} else {

val sparkConf = new SparkConf().setAppName("DataGenerator")

new SparkContext(sparkConf)

}

# How is memory handled in Datasets?

Datasets tend to use less memory.

Spark understands the structure of data in Datasets, because they're typed.

Spark uses encoders to translate between these types ("domain objects") and

Spark's compact internal Tungsten data format.

It generates these encoders via runtime code-generation. The generated code

can operate directly on the Tungsten compact format.

Memory is conserved, because of the compact format. Speed is improved by

custom code-generation.

# Explain Datasets and Serialization

Spark has to serialize data ... a lot. Because of the efficiency of the code-generated encoders, serialization can be significantly faster than either native Java or Kryo serialization.

The resulting serialized data will often be up to 2x smaller, as well, which reduces disk use and network use.

# What are the limitations of dataset?

Dataset API lacks some aggregators (like sum()) and lacks a sortBy() function.

# Implementation difference between DataFrames and DataSet?

DataFrames produces Rows, in DataSet one can use Scala’s case classes or tuples to describe the contents of the rows. The (not so) magic gluing is done by using as on a Dataframe. (Tupels would match by position and also lack the possibility to customize naming.)

final case class Body(id: Int,

                      width: Double,

                      height: Double,

                      depth: Double,

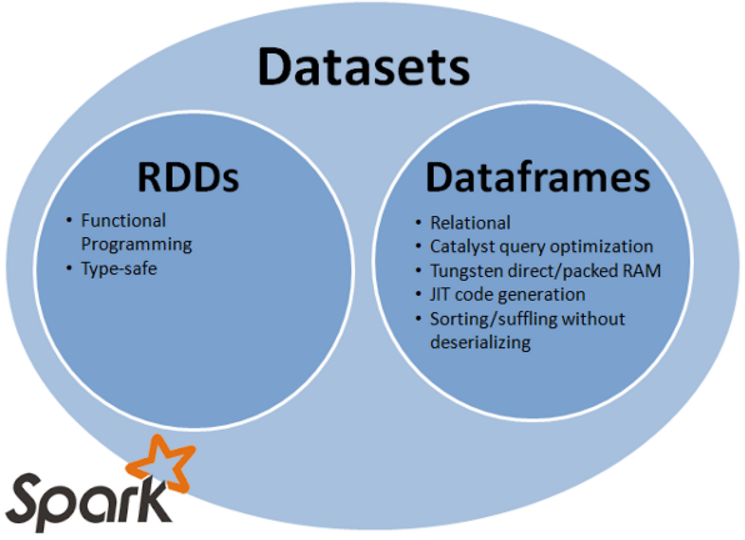
                      material: String,

                      color: String)

 val ds = df.as[Body]

The matching between the DataFrames columns and the fields of the case class is done by name and the types should match. In summary, this introduces a contract and narrows down possible sources of error. For example, one immediate benefit is that we can access fields via the dot operator and get additional IDE support

Here is the visual presentation of RDD, DataFrames and DataSet. [Ref](http://datascience-enthusiast.com/figures/RDD_DataFrames.png)



# What are the contentions with memory?

**Memory contention poses three challenges for Apache Spark**

How to arbitrate memory between execution and storage?

How to arbitrate memory across tasks running in parallel?

How to arbitrate memory across operators running within the same task?

**Two usages of memory in Apache Spark**

Execution: Memory used for shuffles, joins, sorts and aggregations

Storage: Memory used to cache data that will be reused later

|  |  |
| --- | --- |
| Execution | Storage |
|  |  |

# Show Command to run spark in YARN client mode?

spark-submit \

–class org.apache.spark.app.SparkPi \

–deploy-mode client \

–master yarn \

$SPARK\_HOME/app/lib/spark-app\_version.jar

# Show Command to run spark in YARN cluster mode?

spark-submit \

–class org.apache.spark.app.SparkPi \

–deploy-mode cluster \

–master yarn \

$SPARK\_HOME/app/lib/spark-app\_version.jar

# What is Standalone and YARN mode?

Standalone mode: Spark uses a Master daemon which coordinates the efforts of the Workers, which run the executors. Standalone mode is the default, but it cannot be used on secure clusters. When you submit an application, you can choose how much memory its executors will use, as well as the total number of cores across all executors. In YARN mode YARN ResourceManager performs the functions of the Spark Master. The functions of the Workers are performed by the YARN NodeManager daemons, which run the executors.

|  |  |
| --- | --- |
| **Standalone cluster deployment – driver running on client machine** | **Standalone cluster deployment – driver running on worker machine** |
|  |  |

# Explain client mode and cluster mode in spark?

|  |  |
| --- | --- |
| **Client Mode** | **Cluster Mode** |
| Each and every application has a driver process which coordinates its execution | Run as independent sets of processes on a cluster, coordinated by the SparkContext object in your main program (called the driver program) |
| Process can run in the foreground | Process can run in the background |
| Simple | Complex |
| spark-submit \ –class org.apache.spark.app.SparkPi \ –deploy-mode client \ –master spark//$SPARK\_MASTER\_IP:$SPARK\_MASTER\_PORT \ $SPARK\_HOME/app/lib/spark-app\_version.jar | spark-submit \ –class org.apache.spark.app.SparkPi \ –deploy-mode cluster \ –master spark//$SPARK\_MASTER\_IP:$SPARK\_MASTER\_PORT \ $SPARK\_HOME/app/lib/spark-app\_version.jar |

# Which cluster managers are supported by spark?

Currently supports three cluster managers:

Standalone – a simple cluster manager included with Spark that makes it easy to set up a cluster.

Apache Mesos – a general cluster manager that can also run Hadoop MapReduce and service applications.

Hadoop YARN – the resource manager in Hadoop 2.

# What is Executor memory?

Spark uses spark.executor.memory (Amount of memory to use per executor process (e.g. 2g, 8g). Default is 1G. For example if spark.executor.memory=2g it means that each executor on every one of your Worker nodes to have 2G memory. This setting doesn't mean "share 2G of memory between all executors", it means "give each executor 2G of memory".

# What is DStream and What is the difference between batch and Dstream in Spark streaming?

Spark supports both batch processing and real time processing.

|  |  |
| --- | --- |
| Batch processing | Real time processing |
| Uses rdd | Uses dstreams |
| Batch rate is higher | Batch rate is in sec or mili seconds |
| Example: web log processing (not critical) | Example: Tweeter feed – real time data processing |

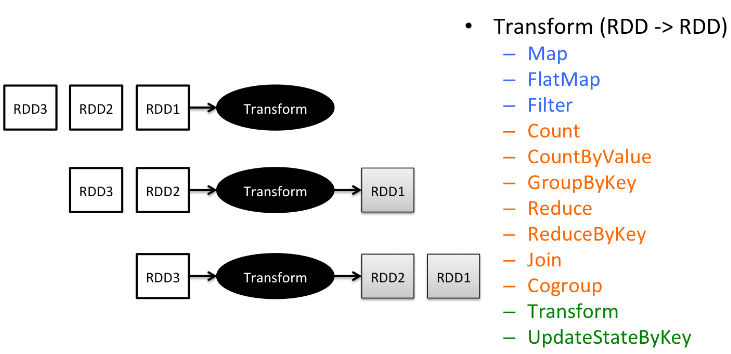
Spark’s high-level abstraction is called a DStream, Live input streams are divided into batches of data items.

DStreams can be created either from input data streams from sources such as Kafka, Flume, and Kinesis, or by applying high-level operations on other DStreams. Internally, a DStream is represented as a sequence of RDDs.

# How does Spark Streaming work?

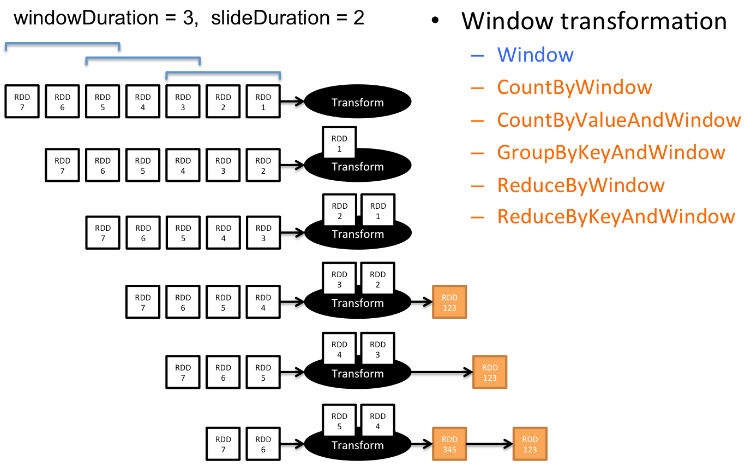
Spark Streaming receives live input data streams and divides the data into batches, which are then processed by the Spark engine to generate the final stream of results in batches. Spark Streaming provides a high-level abstraction called *discretized stream* or *DStream*, which represents a continuous stream of data. DStreams can be created either from input data streams from sources such as Kafka, Flume, or by applying high-level operations on other DStreams. Internally, a DStream is represented as a sequence of RDDs.

Here is the basic transformation where each RDD in the output DStream has a one to one correspondence with each RDD in the input DStream. [Ref](http://horicky.blogspot.com/)



Instead of performing a 1 to 1 transformation of each RDD in the DStream. Spark streaming enable a sliding window operation by defining a WINDOW which groups consecutive RDDs along the time dimension. There are 2 parameters that the window has defined.

1. **Window length**: defines how many consecutive RDDs will be combined for performing the transformation.
2. **Slide interval**: defines how many RDD will be skipped before the next transformation executes.



By providing a similar set of transformation operation for both RDD and DStream, Spark enable a unified programming paradigm across both batch and real-time processing, and hence reduce the corresponding development and maintenance cost.

# Difference between map() and flatMap()?

flatMap = map + flatten. Map expresses a one-to-one transformation that transforms each element of a collection (like an RDD) into one element of the resulting collection. flatMap expresses a one-to-many transformation that transforms each element to 0 or more elements. The map() transformation takes in a function and applies it to each element in the RDD with the result of the function being the new value of each element in the resulting RDD.

Example: [Ref](https://datamize.wordpress.com/2015/02/08/visualizing-basic-rdd-operations-through-wordcount-in-pyspark/)

|  |  |
| --- | --- |
|  | The **flatMap** function takes the input file that is returned by the sc.textFile function that returns the lines of the file. This flatMap does two things it applies the lambda function to each line, creating a list of space separated words. Then the second thing flatMap does by default is flattening the list of lists, meaning that [[w1,w2],[w2,w3]] becomes [w1,w2,w2,w3]  The **map** function is applied to the resulting RDD that is produced by flatMap. The map operation applies the lambda function provided to each element in the RDD. Here each element is a word in the list of words RDD and the map produces a pair for each word composed of the word as the key and the initial count as of that word as 1. |

# What is reduce() action, Is there any difference between reduce() and reduceByKey()?

**Reduce():** It takes a function that operates on two elements of the type in your RDD and returns a new element of the same type. It is a wide operation as it is shuffling data from multiple partitions and reduces to a single value

It accepts a Commutative (f(x, y) = f(y, x)) and Associative(f(f(x, y), z) = f(f(x, z), y) = f(f(y, z), x)) function as an argument

* The parameter function should have two arguments of the same data type
* The return type of the function also must be same as argument types

A simple example of such a function is +, which we can use to sum our RDD. With reduce(), we can easily sum the elements of our RDD, count the number of elements, and perform other types of aggregations. fold() is similar to reduce except that it takes an ‘Zero value‘ which will be used in the initial call on each Partition.

* reduceByKey() returns an RDD which is just another level/state in the DAG, therefore is a transformation.
* reduceByKey() is a function that operates on an RDD of key-value pairs.
* reduce() function is a member of RDD[T] class while reduceByKey() is a member of the PairRDDFunctions[K, V] class.

# What is the disadvantage of Reduce() action and how can we overcome this limitation?

For both reduce() & fold(), the return type should be the same as the RDD element type. aggregate() function can be used to avoid this limitation. foldByKey() is very similar to fold() except that it operates on a Pair RDD

# What are Accumulators and When are accumulators truly reliable?

Accumulator updates are sent back to the driver when a task is successfully completed. So accumulator results are guaranteed to be correct when you are certain that each task will have been executed exactly once and each task did as you expected. Thus Accumulators, provides a simple syntax for aggregating values from worker nodes back to the driver program. For example: Suppose we have a log file and we want to avoid the count of "Blank Lines", "Missing Fields", “Zero values” etc thus accumulator will update all the variables and executor relays back this info back to the driver.

Take a case when Task failed due to some exception in code. Spark will try 4 times(default number of tries).If task fail every time it will give an exception. If by chance it succeeds then Spark will continue and just update the accumulator value for successful state and failed states accumulator values are ignored. No issue with accumulator in this case. However if a task is running slow then, Spark can launch a speculative copy of that task on another node. In this case accumulator is not handled properly and Accumulator count will be wrong. RDD is cached but if it is huge and can't reside in Memory. Thus whenever the RDD is used, it will re run the Map operation to get the RDD and again accumulator will be updated, thus producing wrong accumulator output. So it may happen same function may run multiple time on same data. So Spark does not provide any guarantee for accumulator getting updated because of the Map operation. Thus accumulators are only reliable when they are present in an **Action** operation.

# What is Broadcast Variables and what is the advantage?

Spark’s second type of shared variable, Broadcast variables are designed to be shared throughout a cluster and, at the same time have to be able to fit in memory on one machine. Secondly, broadcast variables are immutable, so they cannot be changed later on (in case take a look at accumulators). Efficiency: Inside Spark, all the nodes in the cluster try to distribute the variable as quickly and efficiently as possible by downloading what they can, and uploading what they can. This makes them much faster than one node having to try and do everything and push the data to all nodes. As referenced in the Apache Spark documentation , broadcast variables are a great case for "static look up tables"

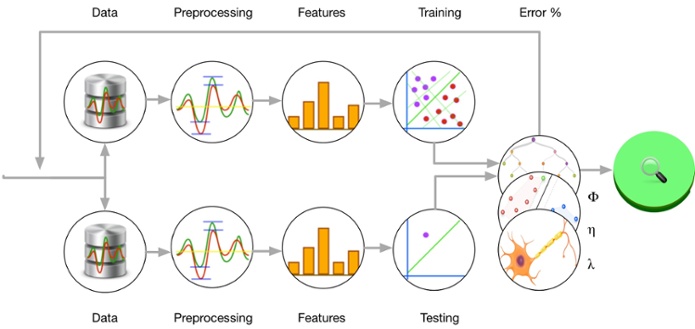
Advantage: If you have huge array that is accessed from Spark Closures, for example some reference data, this array will be shipped to each spark node with closure. For example if you have 10 nodes cluster with 100 partitions (10 partitions per node), this Array will be distributed at least 100 times (10 times to each node).

If you use broadcast it will be distributed once per node using efficient p2p protocol.

val array: Array[Int] = ??? // some huge array

val broadcasted = sc.broadcast(array)

# What is Piping, show an example who to create data pipeline?



**Diagram – Machine learning data pipeline** [**Ref**](http://cdn2.hubspot.net/hubfs/323094/spark-pipeline.jpg)

Pipe operator in Spark, allows developer to process RDD data using external applications. Sometimes in data analysis, we need to use an external library which may not be written using Java/Scala. Ex: Fortran math libraries. In that case, spark’s pipe operator allows us to send the RDD data to the external application.

Pipe in Spark

The following steps shows how to use pipe operator. To start with, we will create an RDD from in memory list.

Step 1 : Create a RDD

val data = List("hi","hello","how","are","you")

val dataRDD = sc.makeRDD(data) //sc is SparkContext

Step 2 : Create a shell script

Once we have RDD, then we will pipe it to a shell script. Let’s create a file called echo.sh, then put the following content.

#!/bin/sh

echo "Running shell script"

while read LINE; do

echo ${LINE}

done

This is a simple shell script which reads the input from stdin and output that to stdout. You can do any other shell operation in this shell script.

Step 3 : Pipe rdd data to shell script

Once we have the shell script, we can pipe the RDD through this script. Make sure that you change the scriptPath variable to match path of your file.

val scriptPath = "/home/hadoop/echo.sh"

val pipeRDD = dataRDD.pipe(scriptPath)

pipeRDD.collect()

Now you should able to see, the line printed on console with echo messages from shell script. In place of shell script, you can use any other executable. [Ref](http://blog.madhukaraphatak.com/pipe-in-spark/)

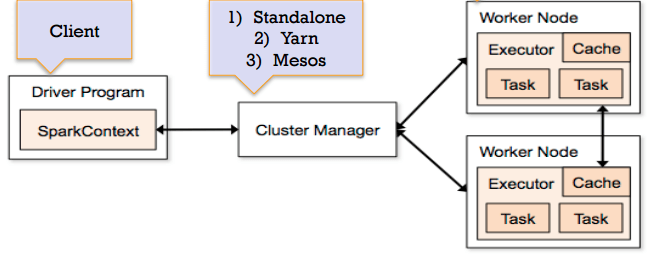
Spark provides a pipe() method on RDDs. Spark’s pipe() lets us write parts of jobs using any language we want as long as it can read and write to Unix standard streams. With pipe(), you can write a transformation of an RDD that reads each RDD element from standard input as a String, manipulates that String however you like, and then writes the result(s) as Strings to standard output.

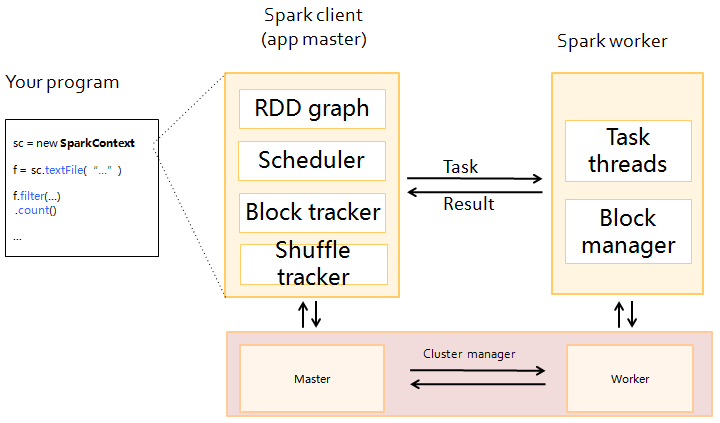
# What is driver?

The driver prepares the context and declares the operations on the data using RDD transformations and actions. It is the cockpit of jobs and tasks execution (using DAG Scheduler and Task Scheduler). It hosts Web UI for the environment. The driver submits the serialized RDD graph to the master. The master creates tasks out of it and submits them to the workers for execution. It coordinates the different job stages. The workers is where the tasks are actually executed. They should have the resources and network connectivity required to execute the operations requested on the RDDs. A driver coordinates workers and overall execution of tasks.

# What are the steps that occur when you run a Spark application on a cluster?

1. The user submits an application using spark-submit.  
2. spark-submit launches the driver program and invokes the main() method specified by the user.  
3. The driver program contacts the cluster manager to ask for resources to launch executors.  
4. The cluster manager launches executors on behalf of the driver program.  
5. The driver process runs through the user application. Based on the RDD actions and transformations in the program, the driver sends work to executors in the form of tasks.  
6. Tasks are run on executor processes to compute and save results.  
7. If the driver’s main() method exits or it calls SparkContext.stop(),it will terminate the executors and release resources from the cluster manager. Diagram [Ref](http://i2.wp.com/jennyxiaozhang.com/wp-content/uploads/2015/03/Spark-architecture1.png)

[](http://i2.wp.com/jennyxiaozhang.com/wp-content/uploads/2015/03/Spark-architecture1.png)



# What is a schema RDD/DataFrame?

A SchemaRDD is an RDD composed of Row objects with additional schema information of the types in each column. Row objects are just wrappers around arrays of basic types (e.g., integers and strings).

# What are Row objects?

Row objects represent records inside SchemaRDDs, and are simply fixed-length arrays of fields. Row objects have a number of getter functions to obtain the value of each field given its index. The standard getter, get (or apply in Scala), takes a column number and returns an Object type (or Any in Scala) that we are responsible for casting to the correct type. For Boolean, Byte, Double, Float, Int, Long, Short, and String, there is a getType() method, which returns that type. For example, get String(0) would return field 0 as a string.

# How does Spark achieve fault tolerance?

Spark stores data in-memory whereas Hadoop stores data on disk. Hadoop uses replication to achieve fault tolerance whereas Spark uses different data storage model, RDD. RDDs achieve fault tolerance through a notion of lineage: if a partition of an RDD is lost, the RDD has enough information to rebuild just that partition. This removes the need for replication to achieve fault tolerance.

# What does a Spark Engine do?

Spark Engine is responsible for scheduling, distributing and monitoring the data application across the cluster.

# What parameter is set if cores need to be defined across executors?

SPARK\_WORKER\_CORES is used to set total cores. Total number of cores to allow Spark applications to use on the machine (default: all available cores). The application will launch executors on every available node in the cluster. For a multiuser workload, you should instead ask users to cap their usage. Max core: You can cap the number of cores by setting spark.cores.max in your SparkConf.

# Name few Spark Master system properties?

SPARK\_MASTER\_OPTS supports the following system properties:

|  |  |  |
| --- | --- | --- |
| **Property Name** | **Default** | **Meaning** |
| spark.deploy.retainedApplications | 200 | The maximum number of completed applications to display. Older applications will be dropped from the UI to maintain this limit. |
| spark.deploy.retainedDrivers | 200 | The maximum number of completed drivers to display. Older drivers will be dropped from the UI to maintain this limit. |
| spark.deploy.spreadOut | true | Whether the standalone cluster manager should spread applications out across nodes or try to consolidate them onto as few nodes as possible. Spreading out is usually better for data locality in HDFS, but consolidating is more efficient for compute-intensive workloads. |
| spark.deploy.defaultCores | (infinite) | Default number of cores to give to applications in Spark's standalone mode if they don't set spark.cores.max. If not set, applications always get all available cores unless they configure spark.cores.max themselves. Set this lower on a shared cluster to prevent users from grabbing the whole cluster by default. |
| spark.deploy.maxExecutorRetries | 10 | Limit on the maximum number of back-to-back executor failures that can occur before the standalone cluster manager removes a faulty application. An application will never be removed if it has any running executors. If an application experiences more thanspark.deploy.maxExecutorRetries failures in a row, no executors successfully start running in between those failures, and the application has no running executors then the standalone cluster manager will remove the application and mark it as failed. To disable this automatic removal, set spark.deploy.maxExecutorRetries to -1. |
| spark.worker.timeout | 60 | Number of seconds after which the standalone deploy master considers a worker lost if it receives no heartbeats. |

# Define Partitions in reference to spark implementation?

A partition (aka split) is a logical chunk of a large distributed data set. Spark manages data using partitions that helps parallelize distributed data processing with minimal network traffic for sending data between executors. By default, Spark tries to read data into an RDD from the nodes that are close to it. Since Spark usually accesses distributed partitioned data, to optimize transformation operations it creates partitions to hold the data chunks. There is a one-to-one correspondence between how data is laid out in data storage like HDFS or Cassandra (it is partitioned for the same reasons).

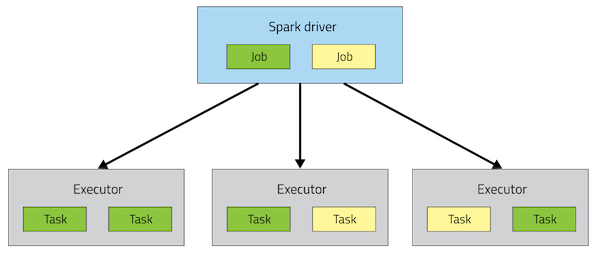
# Differences between how Spark and MapReduce manage cluster resources under YARN. [Ref](http://blog.cloudera.com/blog/2014/05/apache-spark-resource-management-and-yarn-app-models/)

The most popular Apache YARN application after MapReduce itself is Apache Spark. At Cloudera, we have worked hard to stabilize Spark-on-YARN ([SPARK-1101](https://issues.apache.org/jira/browse/SPARK-1101)), and CDH 5.0.0 added support for Spark on YARN clusters.

In this post, you’ll learn about the differences between the Spark and MapReduce architectures, why you should care, and how they run on the YARN cluster ResourceManager.

Applications

In MapReduce, the highest-level unit of computation is a job. The system loads the data, applies a map function, shuffles it, applies a reduce function, and writes it back out to persistent storage. Spark has a similar job concept (although a job can consist of more stages than just a single map and reduce), but it also has a higher-level construct called an “application,” which can run multiple jobs, in sequence or in parallel.



Spark application architecture

For those familiar with the Spark API, an application corresponds to an instance of the SparkContext class. An application can be used for a single batch job, an interactive session with multiple jobs spaced apart, or a long-lived server continually satisfying requests. Unlike MapReduce, an application will have processes, called Executors, running on the cluster on its behalf even when it’s not running any jobs. This approach enables data storage in memory for quick access, as well as lightning-fast task startup time.

**Executors**

MapReduce runs each task in its own process. When a task completes, the process goes away. In Spark, many tasks can run concurrently in a single process, and this process sticks around for the lifetime of the Spark application, even when no jobs are running.

The advantage of this model, as mentioned above, is speed: Tasks can start up very quickly and process in-memory data. The disadvantage is coarser-grained resource management. As the number of executors for an app is fixed and each executor has a fixed allotment of resources, an app takes up the same amount of resources for the full duration that it’s running. (When YARN supports [container resizing](https://issues.apache.org/jira/browse/YARN-1197), we plan to take advantage of it in Spark to acquire and give back resources dynamically.)

Active Driver

To manage the job flow and schedule tasks Spark relies on an active driver process. Typically, this driver process is the same as the client process used to initiate the job, although in YARN mode (covered later), the driver can run on the cluster. In contrast, in MapReduce, the client process can go away and the job can continue running. In Hadoop 1.x, the JobTracker was responsible for task scheduling, and in Hadoop 2.x, the MapReduce application master took over this responsibility.

Pluggable Resource Management

Spark supports pluggable cluster management. The cluster manager is responsible for starting executor processes. Spark application writers do not need to worry about what cluster manager against which Spark is running.

Spark supports YARN, Mesos, and its own “standalone” cluster manager. All three of these frameworks have two components. A central master service (the YARN ResourceManager, Mesos master, or Spark standalone master) decides which applications get to run executor processes, as well as where and when they get to run. A slave service running on every node (the YARN NodeManager, Mesos slave, or Spark standalone slave) actually starts the executor processes. It may also monitor their liveliness and resource consumption.

Why Run on YARN?

Using YARN as Spark’s cluster manager confers a few benefits over Spark standalone and Mesos:

* YARN allows you to dynamically share and centrally configure the same pool of cluster resources between all frameworks that run on YARN. You can throw your entire cluster at a MapReduce job, then use some of it on an Impala query and the rest on Spark application, without any changes in configuration.
* You can take advantage of [all the features of YARN schedulers](http://hadoop.apache.org/docs/r2.4.0/hadoop-yarn/hadoop-yarn-site/FairScheduler.html) for categorizing, isolating, and prioritizing workloads.
* Spark standalone mode requires each application to run an executor on every node in the cluster, whereas with YARN, you choose the number of executors to use.
* Finally, YARN is the only cluster manager for Spark that supports security. With YARN, Spark can run against Kerberized Hadoop clusters and uses secure authentication between its processes.

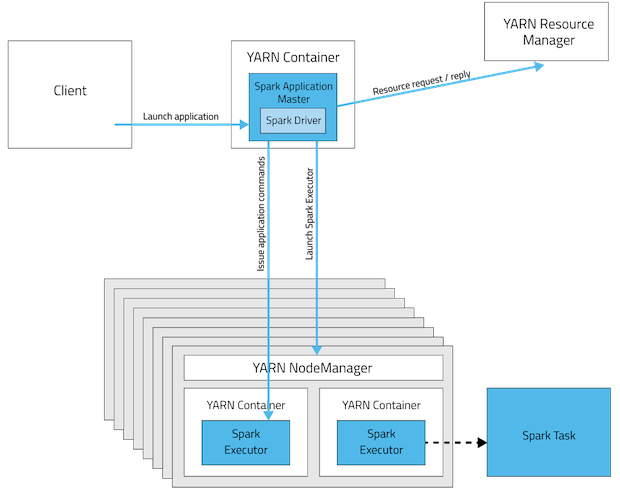
Running on YARN

When running Spark on YARN, each Spark executor runs as a YARN container. Where MapReduce schedules a container and fires up a JVM for each task, Spark hosts multiple tasks within the same container. This approach enables several orders of magnitude faster task startup time.

Spark supports two modes for running on YARN, “yarn-cluster” mode and “yarn-client” mode.  Broadly, yarn-cluster mode makes sense for production jobs, while yarn-client mode makes sense for interactive and debugging uses where you want to see your application’s output immediately.

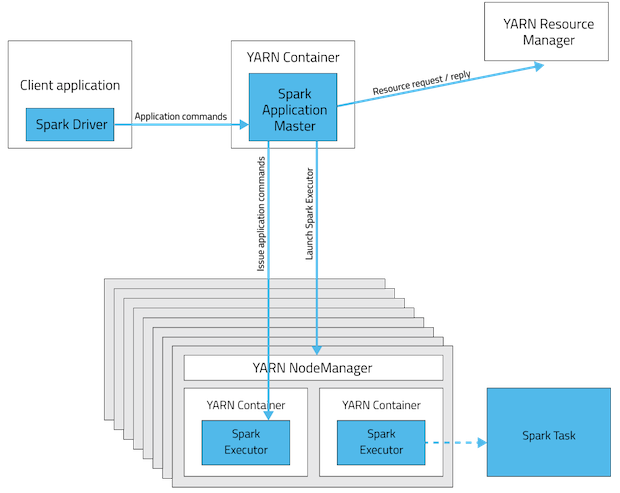
Understanding the difference requires an understanding of YARN’s Application Master concept. In YARN, each application instance has an Application Master process, which is the first container started for that application. The application is responsible for requesting resources from the ResourceManager, and, when allocated them, telling NodeManagers to start containers on its behalf. Application Masters obviate the need for an active client — the process starting the application can go away and coordination continues from a process managed by YARN running on the cluster.

In yarn-cluster mode, the driver runs in the Application Master. This means that the same process is responsible for both driving the application and requesting resources from YARN, and this process runs inside a YARN container. The client that starts the app doesn’t need to stick around for its entire lifetime.



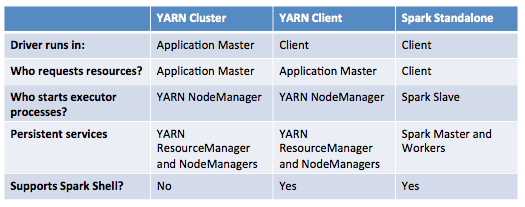
**yarn-cluster mode**

The yarn-cluster mode, however, is not well suited to using Spark interactively. Spark applications that require user input, like spark-shell and PySpark, need the Spark driver to run inside the client process that initiates the Spark application. In yarn-client mode, the Application Master is merely present to request executor containers from YARN. The client communicates with those containers to schedule work after they start:



yarn-client mode

This table offers a concise list of differences between these modes:



Key Concepts in Summary

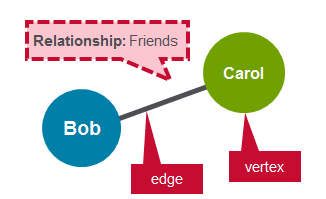
* Application: This may be a single job, a sequence of jobs, a long-running service issuing new commands as needed or an interactive exploration session.
* Spark Driver: The Spark driver is the process running the spark context (which represents the application session). This driver is responsible for converting the application to a directed graph of individual steps to execute on the cluster. There is one driver per application.
* Spark Application Master: The Spark Application Master is responsible for negotiating resource requests made by the driver with YARN and finding a suitable set of hosts/containers in which to run the Spark applications. There is one Application Master per application.
* Spark Executor: A single JVM instance on a node that serves a single Spark application. An executor runs multiple tasks over its lifetime, and multiple tasks concurrently. A node may have several Spark executors and there are many nodes running Spark Executors for each client application.
* Spark Task: A Spark Task represents a unit of work on a partition of a distributed dataset.

# What is Hive on Spark?

Hive contains significant support for Apache Spark, Hive over Spark is a Cloudera concept (Impala). Hortonworks believes in Hive over Tez. Hortonworks does not support Hive over Spark. Hive on Spark supports Spark on yarn mode by default.

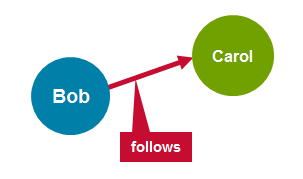
# What is GraphX and what is PageRank? [Ref](https://www.mapr.com/blog/how-get-started-using-apache-spark-graphx-scala)

A graph is a mathematical structure used to model relations between objects. A graph is made up of vertices and edges that connect them. The vertices are the objects and the edges are the relationships between them.



PageRank is the measure of each vertex in the graph. For instance, an edge from Bob to Carol represents endorsement of v’s importance by Carol. In simple terms, if a user at social media is followed massively, it will rank high on that platform.

A **directed graph** is a graph where the edges have a direction associated with them. An example of a directed graph is a Twitter follower. User Bob can follow user Carol without implying that user Carol follows user Bob.

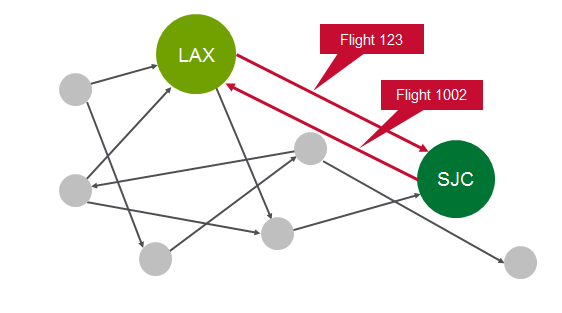


A  **regular graph** is a graph where each vertex has the same number of edges. An example of a regular graph is Facebook friends. If Bob is a friend of Carol, then Carol is also a friend of Bob.

GRAPHX PROPERTY GRAPH

GraphX extends the Spark RDD with a Resilient Distributed Property Graph.

The [property graph](http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.graphx.Graph) is a directed multigraph which can have multiple edges in parallel. Every edge and vertex has user defined properties associated with it. The parallel edges allow multiple relationships between the same vertices.



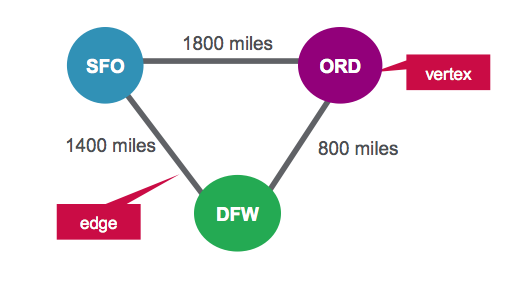
In this activity, you will use GraphX to analyze flight data.

Scenario

As a starting simple example, we will analyze three flights. For each flight, we have the following information:

| **Originating Airport** | **Destination Airport** | **Distance** |
| --- | --- | --- |
| SFO | ORD | 1800 miles |
| ORD | DFW> | 800 miles |
| DFW | SFO> | 1400 miles |

In this scenario, we are going to represent the airports as vertices and routes as edges. For our graph we will have three vertices, each representing an airport. The distance between the airports is a route property, as shown below:



VERTEX TABLE FOR AIRPORTS

| **ID** | **Property** |
| --- | --- |
| 1 | SFO |
| 2 | ORD |
| 3 | DFW |

EDGES TABLE FOR ROUTES

| **SrcId** | **DestId** | **Property** |
| --- | --- | --- |
| 1 | 2 | 1800 |
| 2 | 3 | 800 |
| 3 | 1 | 1400 |

SOFTWARE

This tutorial will run on the MapR Sandbox, which includes Spark.

* You can download the code and data to run these examples from here:
  + <https://github.com/caroljmcdonald/sparkgraphxexample>
* The examples in this post can be run in the Spark shell, after launching with the spark-shell command.
* You can also run the code as a standalone application as described in the tutorial on [Getting Started with Spark on MapR Sandbox](https://www.mapr.com/products/mapr-sandbox-hadoop/tutorials/spark-tutorial).

LAUNCH THE SPARK INTERACTIVE SHELL

Log into the MapR Sandbox, as explained in [Getting Started with Spark on MapR Sandbox](https://www.mapr.com/products/mapr-sandbox-hadoop/tutorials/spark-tutorial), using userid user01, password mapr. Start the spark shell with:

$ spark-shell

DEFINE VERTICES

First we will import the GraphX packages.

(In the code boxes, comments are in Green and output is in Blue)

import org.apache.spark.\_

import org.apache.spark.rdd.RDD

// import classes required for using GraphX

import org.apache.spark.graphx.\_

We define airports as vertices. Vertices have an Id and can have properties or attributes associated with them. Each vertex consists of :

* Vertex id → Id (Long)
* Vertex Property → name (String)

VERTEX TABLE FOR AIRPORTS

| **ID** | **Property(V)** |
| --- | --- |
| 1 | SFO |

We define an RDD with the above properties that is then used for the vertexes.

// create vertices RDD with ID and Name

val vertices=Array((1L, ("SFO")),(2L, ("ORD")),(3L,("DFW")))

val vRDD= sc.parallelize(vertices)

vRDD.take(1)

// Array((1,SFO))

// Defining a default vertex called nowhere

val nowhere = "nowhere"

DEFINE EDGES

Edges are the routes between airports. An edge must have a source, a destination, and can have properties. In our example, an edge consists of:

* Edge origin id → src (Long)
* Edge destination id → dest (Long)
* Edge Property distance → distance (Long)

EDGES TABLE FOR ROUTES

| **srcid** | **destid** | **Property(E)** |
| --- | --- | --- |
| 1 | 12 | 1800 |

We define an RDD with the above properties that is then used for the edges. The edge RDD has the form (src id, dest id, distance ).

// create routes RDD with srcid, destid, distance

val edges = Array(Edge(1L,2L,1800),Edge(2L,3L,800),Edge(3L,1L,1400))

val eRDD= sc.parallelize(edges)

eRDD.take(2)

// Array(Edge(1,2,1800), Edge(2,3,800))

CREATE PROPERTY GRAPH

To create a graph, you need to have a Vertex RDD, Edge RDD, and a Default vertex.

Create a property graph called graph.

// define the graph

val graph = Graph(vRDD,eRDD, nowhere)

// graph vertices

graph.vertices.collect.foreach(println)

// (2,ORD)

// (1,SFO)

// (3,DFW)

// graph edges

graph.edges.collect.foreach(println)

// Edge(1,2,1800)

// Edge(2,3,800)

// Edge(3,1,1400)

1. How many airports are there?

// How many airports?

val numairports = graph.numVertices

// Long = 3

2. How many routes are there?

// How many routes?

val numroutes = graph.numEdges

// Long = 3

3. which routes > 1000 miles distance?

// routes > 1000 miles distance?

graph.edges.filter { case Edge(src, dst, prop) => prop > 1000 }.collect.foreach(println)

// Edge(1,2,1800)

// Edge(3,1,1400)

4. The EdgeTriplet class extends the Edge class by adding the srcAttr and dstAttr members which contain the source and destination properties, respectively.

// triplets

graph.triplets.take(3).foreach(println)

((1,SFO),(2,ORD),1800)

((2,ORD),(3,DFW),800)

((3,DFW),(1,SFO),1400)

5. Sort and print out the longest distance routes

// print out longest routes

graph.triplets.sortBy(\_.attr, ascending=false).map(triplet =>

"Distance " + triplet.attr.toString + " from " + triplet.srcAttr + " to " + triplet.dstAttr + ".").collect.foreach(println)

Distance 1800 from SFO to ORD.

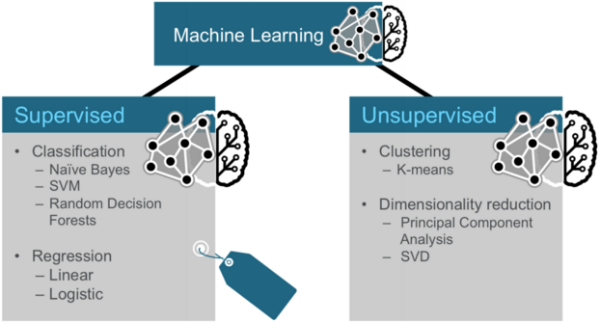
Distance 1400 from DFW to SFO.

Distance 800 from ORD to DFW.

# What does MLlib do? [Ref](https://www.mapr.com/blog/apache-spark-machine-learning-tutorial)

Overview of ML Algorithms

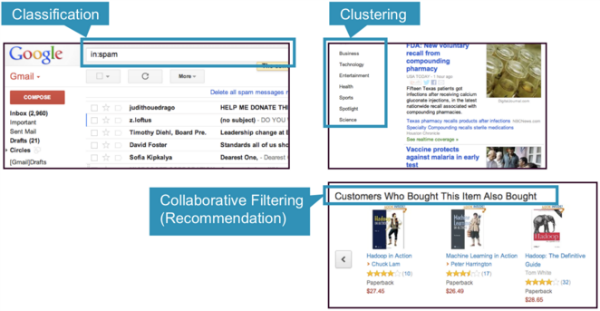
In general, machine learning may be broken down into two classes of algorithms: supervised and unsupervised.



Supervised algorithms use labeled data in which both the input and output are provided to the algorithm. Unsupervised algorithms do not have the outputs in advance. These algorithms are left to make sense of the data without labels.

Three Categories of Techniques for Machine Learning

Three common categories of machine learning techniques are Classification, Clustering and Collaborative Filtering.



* Classification: Gmail uses a machine learning technique called classification to designate if an email is spam or not, based on the data of an email: the sender, recipients, subject, and message body. Classification takes a set of data with known labels and learns how to label new records based on that information.
* Clustering: Google News uses a technique called clustering to group news articles into different categories, based on title and content. Clustering algorithms discover groupings that occur in collections of data.
* Collaborative Filtering: Amazon uses a machine learning technique called collaborative filtering (commonly referred to as recommendation), to determine which products users will like based on their history and similarity to other users.

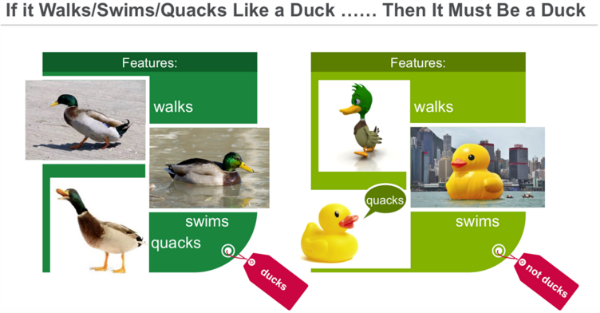
Classification

Classification is a family of supervised machine learning algorithms that designate input as belonging to one of several pre-defined classes. Some common use cases for classification include:

* credit card fraud detection
* email spam detection

Classification data is labeled, for example, as spam/non-spam or fraud/non-fraud. Machine learning assigns a label or class to new data.

You classify something based on pre-determined features. Features are the “if questions” that you ask. The label is the answer to those questions. In this example, if it walks, swims, and quacks like a duck, then the label is "duck."



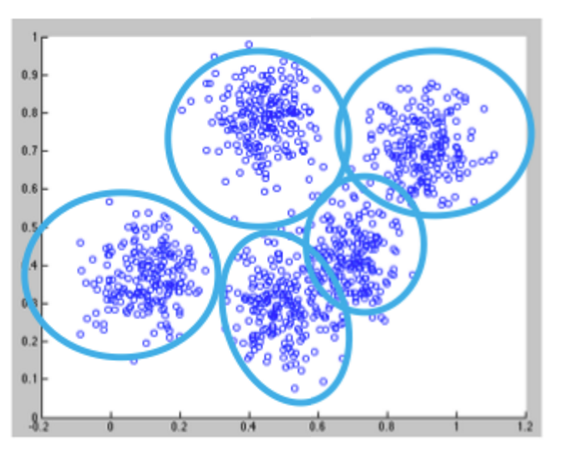
Clustering

In clustering, an algorithm groups objects into categories by analyzing similarities between input examples. Clustering uses include:

* **Search results** grouping
* **Grouping of customers**
* **Anomaly** detection
* Text categorization



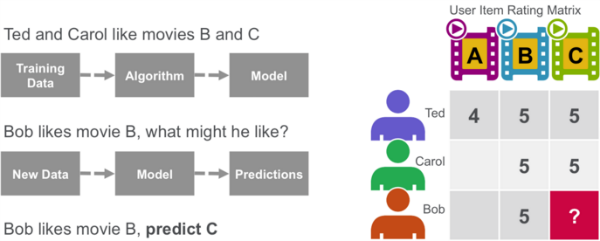
Clustering uses unsupervised algorithms, which do not have the outputs in advance.



Clustering using the K-means algorithm begins by initializing all the coordinates to centroids. With every pass of the algorithm, each point is assigned to its nearest centroid based on some distance metric, usually Euclidean distance. The centroids are then updated to be the “centers” of all the points assigned to it in that pass. This repeats until there is a minimum change in the centers.

Collaborative Filtering

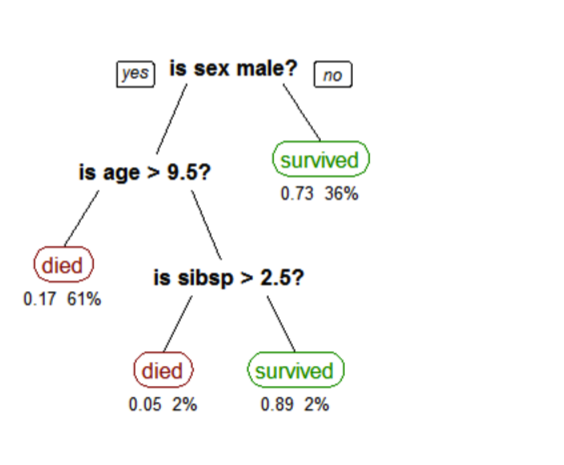
Collaborative filtering algorithms recommend items (this is the filtering part) based on preference information from many users (this is the collaborative part). The collaborative filtering approach is based on similarity; people who liked similar items in the past will like similar items in the future. The goal of a collaborative filtering algorithm is to take preferences data from users, and to create a model that can be used for recommendations or predictions. Ted likes movies A, B, and C. Carol likes movies B and C. We take this data and run it through an algorithm to build a model. Then when we have new data such as Bob likes movie B, we use the model to predict that C is a possible recommendation for Bob.



Decision Trees

Decision trees create a model that predicts the class or label based on several input features. Decision trees work by evaluating an expression containing a feature at every node and selecting a branch to the next node based on the answer. A decision tree for predicting survival on the Titanic is shown below. The feature questions are the nodes, and the answers “yes” or “no” are the branches in the tree to the child nodes.

* Q1: is sex male?
  + yes
  + Q2: is age > 9.5?
    - No
    - Is sibsp >2.5?
      * No
      * died



A tree showing survival of passengers on the [Titanic](https://en.wikipedia.org/wiki/Titanic) ("sibsp" is the number of spouses or siblings aboard). The figures under the leaves show the probability of survival and the percentage of observations in the leaf.

[Reference: tree titanic survivors by Stephen Milborrow](https://en.wikipedia.org/wiki/Decision_tree_learning)

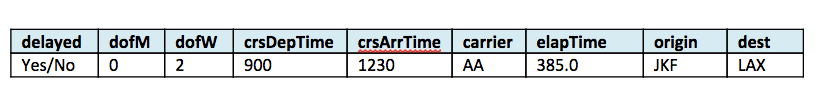
Analyze Flight Delays with Spark Machine Learning Scenario

Our data is from <http://www.transtats.bts.gov/DL_SelectFields.asp?Table_ID=236&DB_Short_Name=On-Time>. We are using flight information for January 2014. For each flight, we have the following information:



In this scenario, we will build a tree to predict the label / classification of delayed or not based on the following features:

* Label → delayed and not delayed - delayed if delay > 40 minutes
* Features → {day\_of\_month, weekday, crsdeptime, crsarrtime, carrier, crselapsedtime, origin, dest, delayed}



# What is a Parquet file?

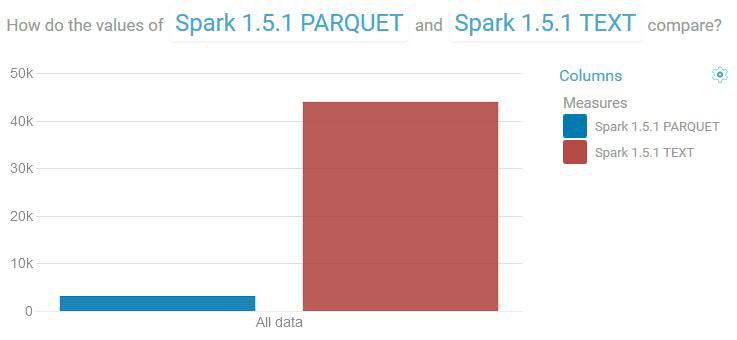
Parquet is a columnar format file supported by many other data processing systems. Spark SQL performs both read and write operations with Parquet file and consider it be one of the best big data analytics format so far.

# Why Parquet is used for Spark SQL? [Ref](https://developer.ibm.com/hadoop/2016/01/14/5-reasons-to-choose-parquet-for-spark-sql/)

It is well-known that columnar storage [saves both time and space](https://developer.ibm.com/hadoop/blog/2015/12/03/parquet-for-spark-sql/) when it comes to big data processing. Parquet, for example, is shown to boost Spark SQL performance by 10X on average compared to using text, thanks to low-level reader filters, efficient execution plans, and in Spark 1.6.0, improved scan throughput!

1. Spark SQL is much faster with Parquet!

The chart below compares the sum of all execution times of the 24 queries running in Spark 1.5.1. Queries taking about 12 hours to complete using flat CVS files vs. taking less than 1 hour to complete using Parquet, a 11X performance improvement.

[](https://i0.wp.com/developer.ibm.com/hadoop/wp-content/uploads/sites/28/2016/01/spark-parquet-vs-text-151.jpg)

*Comparing total query times in seconds between text and Parquet. Lower is better.*

2. Spark SQL works better at large-scale with Parquet

Poor choice of storage format often cause exceptions that are difficult to diagnose and fix. At 1TB scale factor for example, at least 1/3 of all runnable queries failed to complete using flat CSV files, but they all completed using Parquet files.

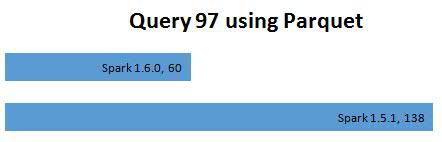
3. Less disk IO

Parquet with compression reduces your data storage by 75% on average, i.e., your 1TB scale factor data files will materialize only about 250 GB on disk. This reduces significantly input data needed for your Spark SQL applications. But in Spark 1.6.0, Parquet readers used push-down filters to further reduce disk IO. Push-down filters allow early data selection decisions to be made before data is even read into Spark

4. Higher scan throughput in Spark 1.6.0

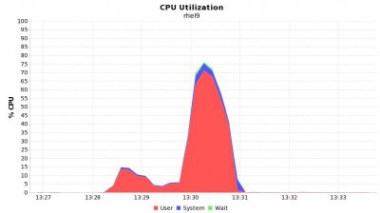
The Databricks’ Spark 1.6.0 release blog mentioned significant Parquet scan throughput because a “more optimized code path” is used. To show this in real world, we ran query 97 in Spark 1.5.1 and in 1.6.0 and captured nmon data. The improvement is very obvious.

First, the query response time is reduced by half: query 97 took 138 seconds in Spark 1.5.1 and 60 seconds in 1.6.0.

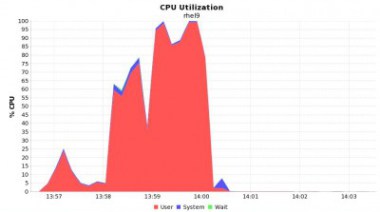
[](https://i1.wp.com/developer.ibm.com/hadoop/wp-content/uploads/sites/28/2016/01/spark-parquet-vs-text-160-151-q97.jpg)

*Query 97 times in seconds using Parquet*

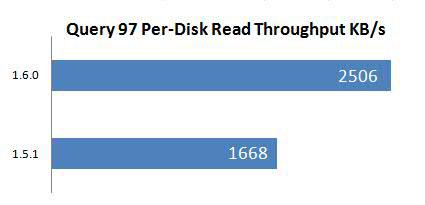
Second, CPU usage on worker nodes is much lower in Spark 1.6.0, mostly attributed by [SPARK-11787](https://issues.apache.org/jira/browse/SPARK-11787):

[](https://i2.wp.com/developer.ibm.com/hadoop/wp-content/uploads/sites/28/2016/01/spark-parquet-vs-text-160-cpu.jpg)

*Query 97 CPU usage in Spark 1.6.0, peaks at 70%*

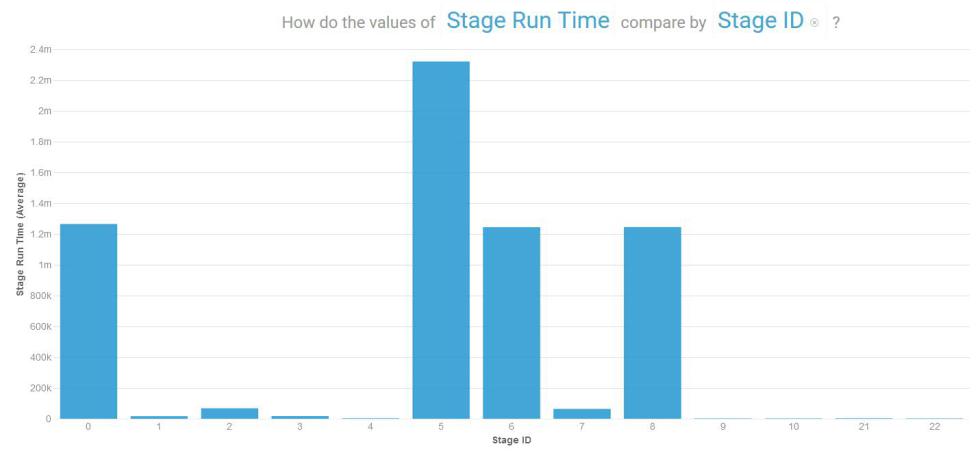
[](https://i2.wp.com/developer.ibm.com/hadoop/wp-content/uploads/sites/28/2016/01/spark-parquet-vs-text-151-cpu-e1452305854490.jpg)

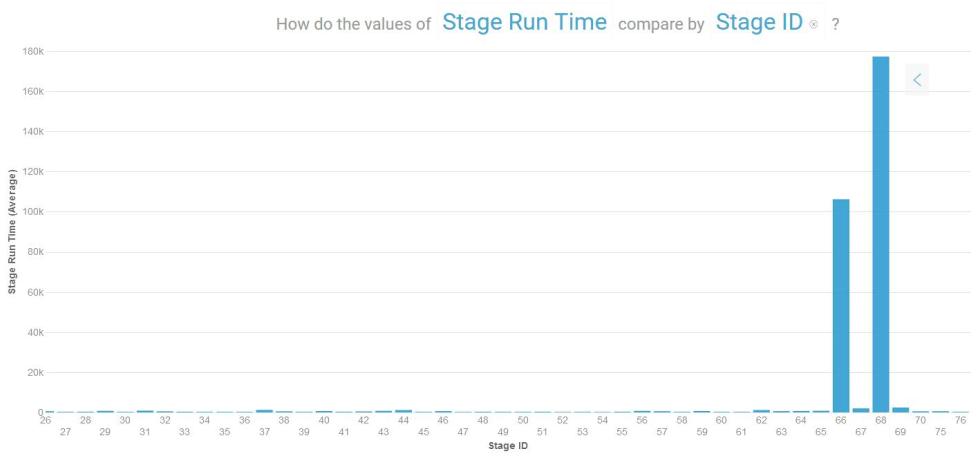
*Query 97 CPU usage in Spark 1.5.1, peaks at 100%*

Related to the above, disk read throughput is 50% higher in Spark 1.6.0:  
[](https://i2.wp.com/developer.ibm.com/hadoop/wp-content/uploads/sites/28/2016/01/spark-parquet-160-151-q97-disk.jpg)

5. Efficient Spark execution graph

In addition to smarter readers such as in Parquet, data formats also directly impact Spark execution graph because one major input to the scheduler is RDD count. In our example where we run the same query 97 on Spark 1.5.1 using text and Parquet, we got the following execution pattern for the stages.

Using text – there are many long-running stages (note the unit on y-axis is milliseconds)  
[](https://i2.wp.com/developer.ibm.com/hadoop/wp-content/uploads/sites/28/2016/01/spark-151-text-stages-q97.jpg)

Using Parquet, though more stages, jobs executed very quickly and created only two longer-running stages toward the end. This indicates cleaner parent-child stage boundaries therefore less intermediate data is needed to save to disk and/or travel via network among nodes, speeding up the end-to-end execution.  
[](https://i1.wp.com/developer.ibm.com/hadoop/wp-content/uploads/sites/28/2016/01/spark-160-text-stages-q97.jpg)

Conclusion

Parquet is great for use with Spark SQL. Not only the compression rate is desirable, it also allows reading only records of interest through selected columns only and through low-level reader filters. So it is [worth your time](https://developer.ibm.com/hadoop/blog/2015/12/03/parquet-for-spark-sql/) to encode your existing flat files even if multiple passes on data maybe required.

# What is schema evolution and what is its disadvantage, explain schema merging in reference to parquet file? [Ref](http://spark.apache.org/docs/latest/sql-programming-guide.html#schema-merging)

Like ProtocolBuffer, Avro, and Thrift, Parquet also supports schema evolution. Users can start with a simple schema, and gradually add more columns to the schema as needed. In this way, users may end up with multiple Parquet files with different but mutually compatible schemas. The Parquet data source is now able to automatically detect this case and merge schemas of all these files.

Since schema merging is a relatively expensive operation, and is not a necessity in most cases, we turned it off by default starting from 1.5.0. You may enable it by

setting data source option mergeSchema to true when reading Parquet files (as shown in the examples below), or

setting the global SQL option spark.sql.parquet.mergeSchema to true.

Code from [github](https://github.com/apache/spark/blob/master/examples/src/main/scala/org/apache/spark/examples/sql/SQLDataSourceExample.scala)

|  |
| --- |
| /\* |
|  | \* Licensed to the Apache Software Foundation (ASF) under one or more |
|  | \* contributor license agreements. See the NOTICE file distributed with |
|  | \* this work for additional information regarding copyright ownership. |
|  | \* The ASF licenses this file to You under the Apache License, Version 2.0 |
|  | \* (the "License"); you may not use this file except in compliance with |
|  | \* the License. You may obtain a copy of the License at |
|  | \* |
|  | \* http://www.apache.org/licenses/LICENSE-2.0 |
|  | \* |
|  | \* Unless required by applicable law or agreed to in writing, software |
|  | \* distributed under the License is distributed on an "AS IS" BASIS, |
|  | \* WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. |
|  | \* See the License for the specific language governing permissions and |
|  | \* limitations under the License. |
|  | \*/ |
|  | package org.apache.spark.examples.sql |
|  |  |
|  | import java.util.Properties |
|  |  |
|  | import org.apache.spark.sql.SparkSession |
|  |  |
|  | object SQLDataSourceExample { |
|  |  |
|  | case class Person(name: String, age: Long) |
|  |  |
|  | def main(args: Array[String]) { |
|  | val spark = SparkSession |
|  | .builder() |
|  | .appName("Spark SQL data sources example") |
|  | .config("spark.some.config.option", "some-value") |
|  | .getOrCreate() |
|  |  |
|  | runBasicDataSourceExample(spark) |
|  | runBasicParquetExample(spark) |
|  | runParquetSchemaMergingExample(spark) |
|  | runJsonDatasetExample(spark) |
|  | runJdbcDatasetExample(spark) |
|  |  |
|  | spark.stop() |
|  | } |
|  |  |
|  | private def runBasicDataSourceExample(spark: SparkSession): Unit = { |
|  | // $example on:generic\_load\_save\_functions$ |
|  | val usersDF = spark.read.load("examples/src/main/resources/users.parquet") |
|  | usersDF.select("name", "favorite\_color").write.save("namesAndFavColors.parquet") |
|  | // $example off:generic\_load\_save\_functions$ |
|  | // $example on:manual\_load\_options$ |
|  | val peopleDF = spark.read.format("json").load("examples/src/main/resources/people.json") |
|  | peopleDF.select("name", "age").write.format("parquet").save("namesAndAges.parquet") |
|  | // $example off:manual\_load\_options$ |
|  | // $example on:direct\_sql$ |
|  | val sqlDF = spark.sql("SELECT \* FROM parquet.`examples/src/main/resources/users.parquet`") |
|  | // $example off:direct\_sql$ |
|  | } |
|  |  |
|  | private def runBasicParquetExample(spark: SparkSession): Unit = { |
|  | // $example on:basic\_parquet\_example$ |
|  | // Encoders for most common types are automatically provided by importing spark.implicits.\_ |
|  | import spark.implicits.\_ |
|  |  |
|  | val peopleDF = spark.read.json("examples/src/main/resources/people.json") |
|  |  |
|  | // DataFrames can be saved as Parquet files, maintaining the schema information |
|  | peopleDF.write.parquet("people.parquet") |
|  |  |
|  | // Read in the parquet file created above |
|  | // Parquet files are self-describing so the schema is preserved |
|  | // The result of loading a Parquet file is also a DataFrame |
|  | val parquetFileDF = spark.read.parquet("people.parquet") |
|  |  |
|  | // Parquet files can also be used to create a temporary view and then used in SQL statements |
|  | parquetFileDF.createOrReplaceTempView("parquetFile") |
|  | val namesDF = spark.sql("SELECT name FROM parquetFile WHERE age BETWEEN 13 AND 19") |
|  | namesDF.map(attributes => "Name: " + attributes(0)).show() |
|  | // +------------+ |
|  | // | value| |
|  | // +------------+ |
|  | // |Name: Justin| |
|  | // +------------+ |
|  | // $example off:basic\_parquet\_example$ |
|  | } |
|  |  |
|  | private def runParquetSchemaMergingExample(spark: SparkSession): Unit = { |
|  | // $example on:schema\_merging$ |
|  | // This is used to implicitly convert an RDD to a DataFrame. |
|  | import spark.implicits.\_ |
|  |  |
|  | // Create a simple DataFrame, store into a partition directory |
|  | val squaresDF = spark.sparkContext.makeRDD(1 to 5).map(i => (i, i \* i)).toDF("value", "square") |
|  | squaresDF.write.parquet("data/test\_table/key=1") |
|  |  |
|  | // Create another DataFrame in a new partition directory, |
|  | // adding a new column and dropping an existing column |
|  | val cubesDF = spark.sparkContext.makeRDD(6 to 10).map(i => (i, i \* i \* i)).toDF("value", "cube") |
|  | cubesDF.write.parquet("data/test\_table/key=2") |
|  |  |
|  | // Read the partitioned table |
|  | val mergedDF = spark.read.option("mergeSchema", "true").parquet("data/test\_table") |
|  | mergedDF.printSchema() |
|  |  |
|  | // The final schema consists of all 3 columns in the Parquet files together |
|  | // with the partitioning column appeared in the partition directory paths |
|  | // root |
|  | // |-- value: int (nullable = true) |
|  | // |-- square: int (nullable = true) |
|  | // |-- cube: int (nullable = true) |
|  | // |-- key: int (nullable = true) |
|  | // $example off:schema\_merging$ |
|  | } |
|  |  |
|  | private def runJsonDatasetExample(spark: SparkSession): Unit = { |
|  | // $example on:json\_dataset$ |
|  | // A JSON dataset is pointed to by path. |
|  | // The path can be either a single text file or a directory storing text files |
|  | val path = "examples/src/main/resources/people.json" |
|  | val peopleDF = spark.read.json(path) |
|  |  |
|  | // The inferred schema can be visualized using the printSchema() method |
|  | peopleDF.printSchema() |
|  | // root |
|  | // |-- age: long (nullable = true) |
|  | // |-- name: string (nullable = true) |
|  |  |
|  | // Creates a temporary view using the DataFrame |
|  | peopleDF.createOrReplaceTempView("people") |
|  |  |
|  | // SQL statements can be run by using the sql methods provided by spark |
|  | val teenagerNamesDF = spark.sql("SELECT name FROM people WHERE age BETWEEN 13 AND 19") |
|  | teenagerNamesDF.show() |
|  | // +------+ |
|  | // | name| |
|  | // +------+ |
|  | // |Justin| |
|  | // +------+ |
|  |  |
|  | // Alternatively, a DataFrame can be created for a JSON dataset represented by |
|  | // an RDD[String] storing one JSON object per string |
|  | val otherPeopleRDD = spark.sparkContext.makeRDD( |
|  | """{"name":"Yin","address":{"city":"Columbus","state":"Ohio"}}""" :: Nil) |
|  | val otherPeople = spark.read.json(otherPeopleRDD) |
|  | otherPeople.show() |
|  | // +---------------+----+ |
|  | // | address|name| |
|  | // +---------------+----+ |
|  | // |[Columbus,Ohio]| Yin| |
|  | // +---------------+----+ |
|  | // $example off:json\_dataset$ |
|  | } |
|  |  |
|  | private def runJdbcDatasetExample(spark: SparkSession): Unit = { |
|  | // $example on:jdbc\_dataset$ |
|  | // Note: JDBC loading and saving can be achieved via either the load/save or jdbc methods |
|  | // Loading data from a JDBC source |
|  | val jdbcDF = spark.read |
|  | .format("jdbc") |
|  | .option("url", "jdbc:postgresql:dbserver") |
|  | .option("dbtable", "schema.tablename") |
|  | .option("user", "username") |
|  | .option("password", "password") |
|  | .load() |
|  |  |
|  | val connectionProperties = new Properties() |
|  | connectionProperties.put("user", "username") |
|  | connectionProperties.put("password", "password") |
|  | val jdbcDF2 = spark.read |
|  | .jdbc("jdbc:postgresql:dbserver", "schema.tablename", connectionProperties) |
|  |  |
|  | // Saving data to a JDBC source |
|  | jdbcDF.write |
|  | .format("jdbc") |
|  | .option("url", "jdbc:postgresql:dbserver") |
|  | .option("dbtable", "schema.tablename") |
|  | .option("user", "username") |
|  | .option("password", "password") |
|  | .save() |
|  |  |
|  | jdbcDF2.write |
|  | .jdbc("jdbc:postgresql:dbserver", "schema.tablename", connectionProperties) |
|  | // $example off:jdbc\_dataset$ |
|  | } |
|  | } |

# What file systems does Spark support?

• Hadoop Distributed File System (HDFS)

• Local File system

•Amazon S3

# Will spark replace MapReduce?

No for now, MapReduce is used by many big data tools including Spark. It is extremely relevant to use MapReduce when the data grows bigger. Tools like Pig and Hive convert their queries into MapReduce phases to optimize them better. Hive can we used via Tez engine or Impala.

# What is Spark Executor?

Spark executors are worker processes responsible for running the individual tasks in a given Spark job. Executors are launched once at the beginning of a Spark application and typically run for the entire lifetime of an application. Executors have two roles. First, they run the tasks that make up the application and return results to the driver. Second, they provide in-memory storage for RDDs that are cached by user programs.

# Name the different types of Cluster Managers in Spark.

The Spark framework supports three major types of Cluster Managers:

• Standalone: a basic manager to set up a cluster

• Apache Mesos: generalized/commonly-used cluster manager, also runs Hadoop MapReduce and other applications

• Yarn: responsible for resource management in Hadoop

# How many ways we can create RDD, show example?

Spark provides two methods to create RDD:

• By parallelizing a collection in your Driver program. This makes use of SparkContext’s ‘parallelize’ method

val IntellipaatData = Array(2,4,6,8,10)

val distIntellipaatData = sc.parallelize(IntellipaatData)

• By loading an external dataset from external storage like HDFS, HBase, shared file system

# How do you flatten rows in Spark? Explain with example. [Ref](http://stackoverflow.com/questions/32906613/flattening-rows-in-spark?noredirect=1&lq=1)

You can use scala explode function:

import org.apache.spark.sql.functions.explode

import org.apache.spark.sql.functions.explode

val test = sqlContext.read.json(sc.parallelize(Seq("""{"a":1,"b":[2,3]}""")))

test.printSchema

//Output

/\*

*root*

*|-- a: long (nullable = true)*

*|-- b: array (nullable = true)*

*| |-- element: long (containsNull = true)*

\*/

val flattened = test.withColumn("b", explode($"b"))

flattened.printSchema

//Output

*/\*root*

*|-- a: long (nullable = true)*

*|-- b: long (nullable = true)*

*\*/*

flattened.show

//Output

*/\**

*+---+---+*

*| a| b|*

*+---+---+*

*| 1| 2|*

*| 1| 3|*

*+---+---+*

*\*/*

# Explain Spark Streaming Architecture?

Spark Streaming uses a “micro-batch” architecture, where Spark Streaming receives data from various input sources and groups it into small batches. New batches are created at regular time intervals. At the beginning of each time interval a new batch is created, and any data that arrives during that interval gets added to that batch. At the end of the time interval the batch is done growing. The size of the time intervals is determined by a parameter called the batch interval. Each input batch forms an RDD,  and is processed using Spark jobs to create other RDDs. The processed results can then be pushed out to external systems in batches.

# What are the types of Transformations on DStreams?

**Stateless transformations:** The processing of each batch does not depend on the data of its previous batches. They include the commonRDD transformations like map(), filter(), and reduceByKey().  
**Stateful transformations**: in contrast, use data or intermediate results from previous batches to compute the results of the current batch. They include transformations based on sliding windows and on tracking state across time.

# What is Receiver in Spark Streaming, can you build custom receivers?

Every input DStream is associated with a **Receiver**object which receives the data from a source and stores it in Spark’s memory for processing. Yes custom receiver can receive streaming data from any arbitrary data source. Can be implemented in Scala or Java. Must provide implementations for onStart() and onStop()

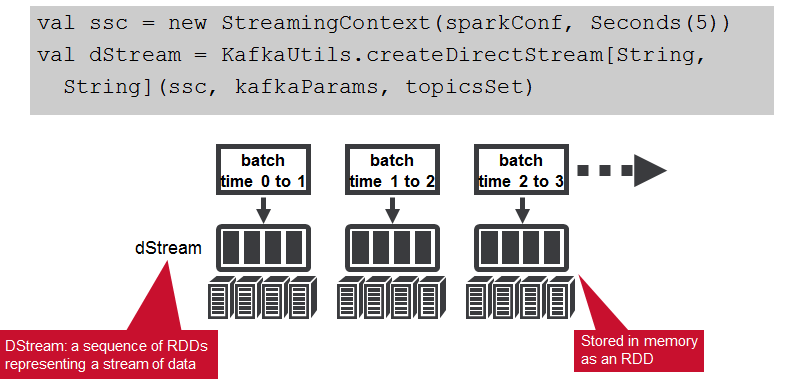
# Explain the process of Live streaming and how to store DStream data to database? [Ref](https://www.mapr.com/blog/real-time-streaming-data-pipelines-apache-apis-kafka-spark-streaming-and-hbase)

Real-Time Streaming Data Pipelines with Apache APIs: Kafka, Spark Streaming, and HBase

**Spark Streaming Code**

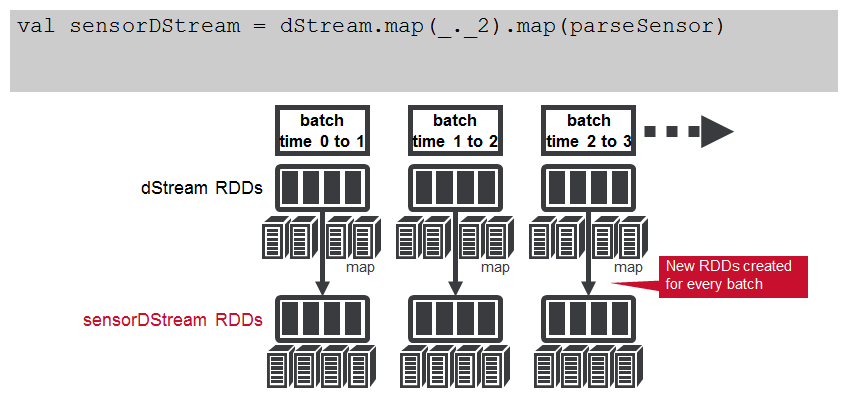
These are the basic steps for Spark Streaming code:

1. Initialize a Spark StreamingContext object. Using this context, create a DStream.
   * We use the KafkaUtils createDirectStream method to create an input stream from a Kafka or MapR Streams topic. This creates a DStream that represents the stream of incoming data, where each record is a line of text.

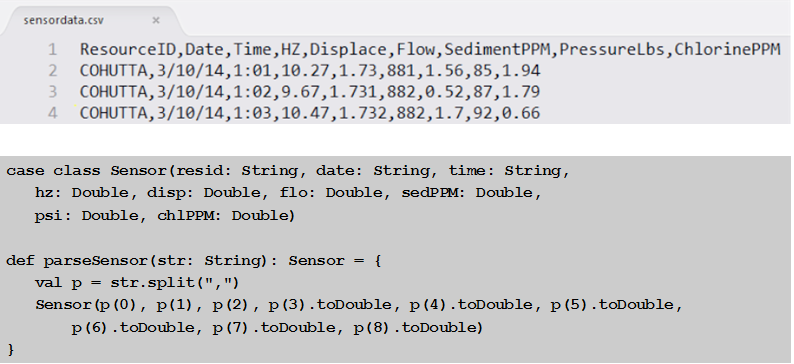


1. Apply transformations (which create new DStreams)
   * We parse the message values into Sensor objects, with the map operation on the dStream. The map operation applies the Sensor.parseSensor function on the RDDs in the dStream, resulting in RDDs of Sensor objects.

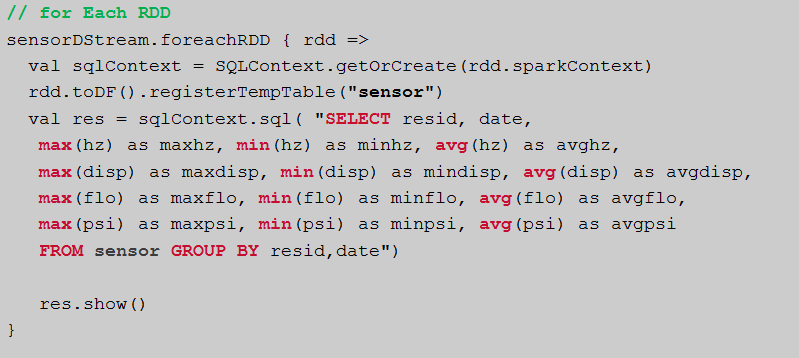
Any operation applied on a DStream translates to operations on the underlying RDDs. The map operation is applied on each RDD in the dStream to generate the sensorDStream RDDs.



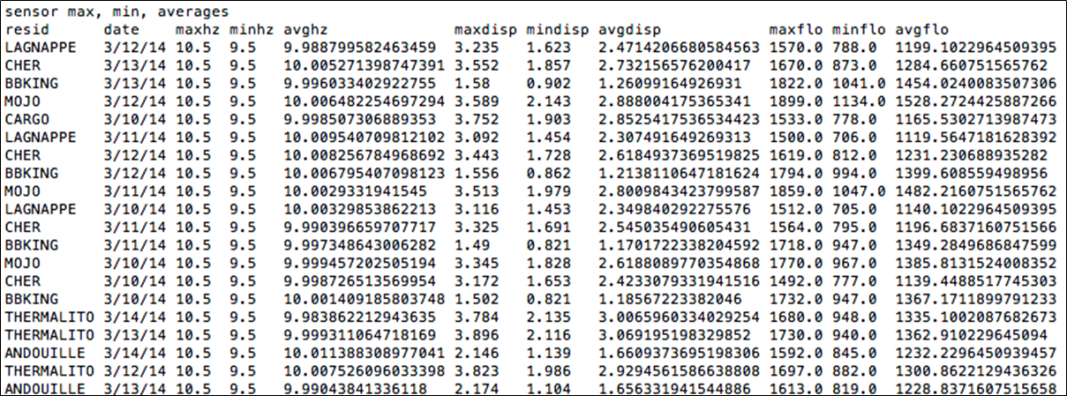
The oil pump sensor data comes in as strings of comma separated values. We use a Scala case class to define the Sensor schema corresponding to the sensor data, and a parseSensor function to parse the comma separated values into the sensor case class.



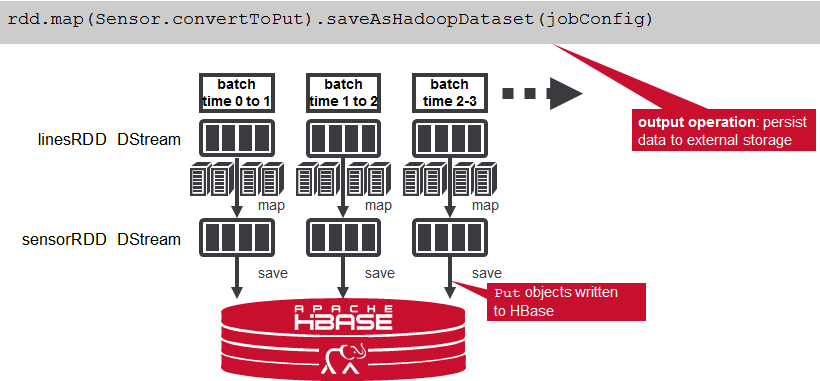
Next, we use the DStream foreachRDD method to apply processing to each RDD in this DStream. We register the DataFrame as a table, which allows us to use it in subsequent SQL statements. We use an SQL query to find the max, min, and average for the sensor attributes.



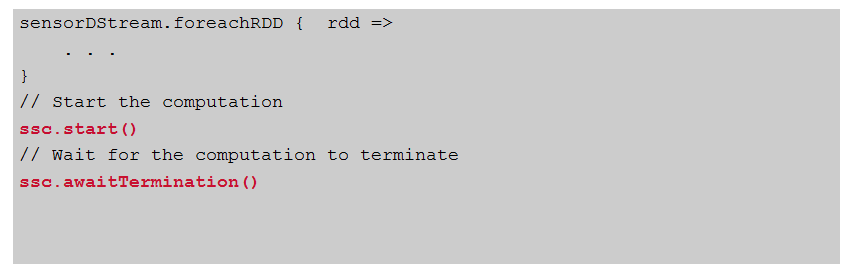
Here is example output from the query which shows the max, min, and average output from our sensors.



1. And/or Apply output operations
   * The sensorRDD objects are filtered for low psi , the sensor and alert data is converted to Put objects, and then written to HBase, using the saveAsHadoopDataset method. This outputs the RDD to any Hadoop-supported storage system using a Hadoop Configuration object for that storage system.



1. Start receiving data and processing it. Wait for the processing to be stopped.
   * To start receiving data, we must explicitly call start() on the StreamingContext, then call awaitTermination to wait for the streaming computation to finish.



**HBase Table schema**

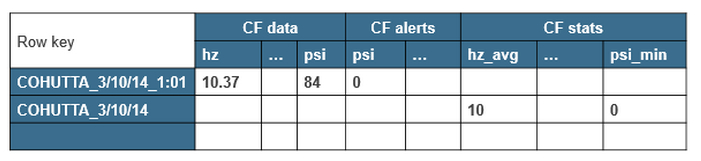
The HBase Table Schema for the streaming data is as follows:

1. Composite row key of the pump name date and time stamp

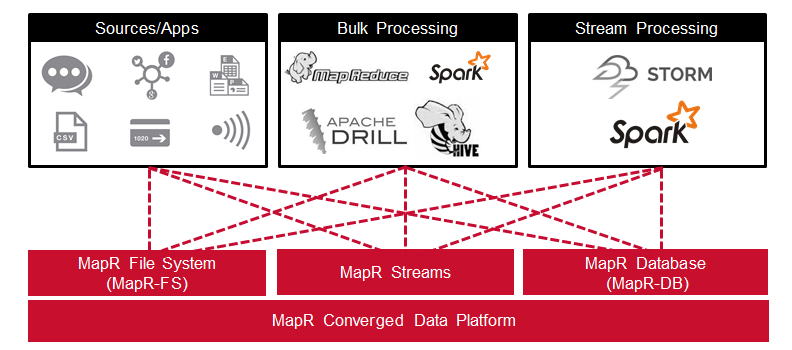
Column Family data with columns corresponding to the input data fields Column Family alerts with columns corresponding to any filters for alarming values Note that the data and alert column families could be set to expire values after a certain amount of time.

The Schema for the daily statistics summary rollups is as follows:

* Composite row key of the pump name and date
* Column Family stats
* Columns for min, max, avg.



All of the components of the use case architecture we just discussed can run on the same cluster with the MapR Converged Data Platform. There are several advantages of having MapR Streams on the same cluster as all the other components. For example, maintaining only one cluster means less infrastructure to provision, manage, and monitor. Likewise, having producers and consumers on the same cluster means fewer delays related to copying and moving data between clusters, and between applications.



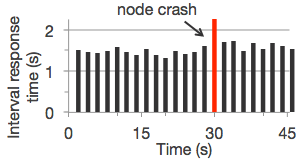
# How spark streaming is fault tolerant?

The demand of high uptimes of a Spark Streaming application require that the application also has to recover from failures of the driver process. Making the Spark driver fault-tolerant is tricky but Spark Streaming applications have an inherent structure in the computation — it runs the same Spark computation periodically on every micro-batch of data. This structure allows us to save (aka, checkpoint) the application state periodically to reliable storage and recover the state on driver restarts.

Write Ahead Logs: Write Ahead Logs (also known as a journal) are used in database and file systems to ensure the durability of any data operations. The intention of the operation is first written down into a durable log , and then the operation is applied to the data. If the system fails in the middle of applying the operation, it can recover by reading the log and reapplying the operations it had intended to do.

When write ahead logs are enabled, all the received data is also saved to log files in a fault-tolerant file system. This allows the received data to durable across any failure in Spark Streaming. Additionally, if the receiver correctly acknowledges receiving data only after the data has been to write ahead logs, the buffered but unsaved data can be resent by the source after the driver is restarted. These two together can ensure that there is zero data loss – all data is either recovered from the logs or resent by the source.

Fault Tolerance: Stateful exactly-once semantics out of the box. Spark Streaming recovers both lost work and operator state (e.g. sliding windows) out of the box, without any extra code on your part.



# Explain transform() method used in dSteam? [Ref](https://databricks.gitbooks.io/databricks-spark-reference-applications/content/logs_analyzer/chapter1/reuse.html)

DStreams have transform functions which allows you to call any arbitrary RDD to RDD functions to RDD's in the DStream. The transform functions are perfect for reusing any RDD to RDD functions that you may have written in batch code and want to port over to streaming. Let's look at some code to illustrate this point.

Let's say we have separated out a function, responseCodeCount from our batch example that can compute the response code count given the apache access logs RDD:

public static JavaPairRDD<Integer, Long> responseCodeCount(

JavaRDD<ApacheAccessLog> accessLogRDD) {

return accessLogRDD

.mapToPair(s -> new Tuple2<>(s.getResponseCode(), 1L))

.reduceByKey(SUM\_REDUCER);

}

The responseCodeCountDStream can be created by calling transformToPair with the responseCodeCount function to the accessLogDStream. Then, you can finish up by calling updateStateByKey to keep a running count of the response codes for all of time, and use forEachRDD to print the values out:

// Compute Response Code to Count.

// Notice the user transformToPair to produce the a DStream of

// response code counts, and then updateStateByKey to accumulate

// the response code counts for all of time.

JavaPairDStream<Integer, Long> responseCodeCountDStream = accessLogDStream

.transformToPair(LogAnalyzerStreamingTotalRefactored::responseCodeCount);

JavaPairDStream<Integer, Long> cumulativeResponseCodeCountDStream =

responseCodeCountDStream.updateStateByKey(COMPUTE\_RUNNING\_SUM);

cumulativeResponseCodeCountDStream.foreachRDD(rdd -> {

System.out.println("Response code counts: " + rdd.take(100));

return null;

});

It is possible to combine transform functions before and after an updateStateByKey as well:

// A DStream of ipAddresses accessed > 10 times.

JavaDStream<String> ipAddressesDStream = accessLogDStream

.transformToPair(LogAnalyzerStreamingTotalRefactored::ipAddressCount)

.updateStateByKey(COMPUTE\_RUNNING\_SUM)

.transform(LogAnalyzerStreamingTotalRefactored::filterIPAddress);

ipAddressesDStream.foreachRDD(rdd -> {

List<String> ipAddresses = rdd.take(100);

System.out.println("All IPAddresses > 10 times: " + ipAddresses);

return null;

});

# How data security is achieved in Spark?

* Set up Kerberos
* Use HDFS (or another secure filesystem)
* Use YARN!
* Configure them for security (enable auth, encryption).
* Kerberos, HDFS, and YARN provide the security backbone for Spark.

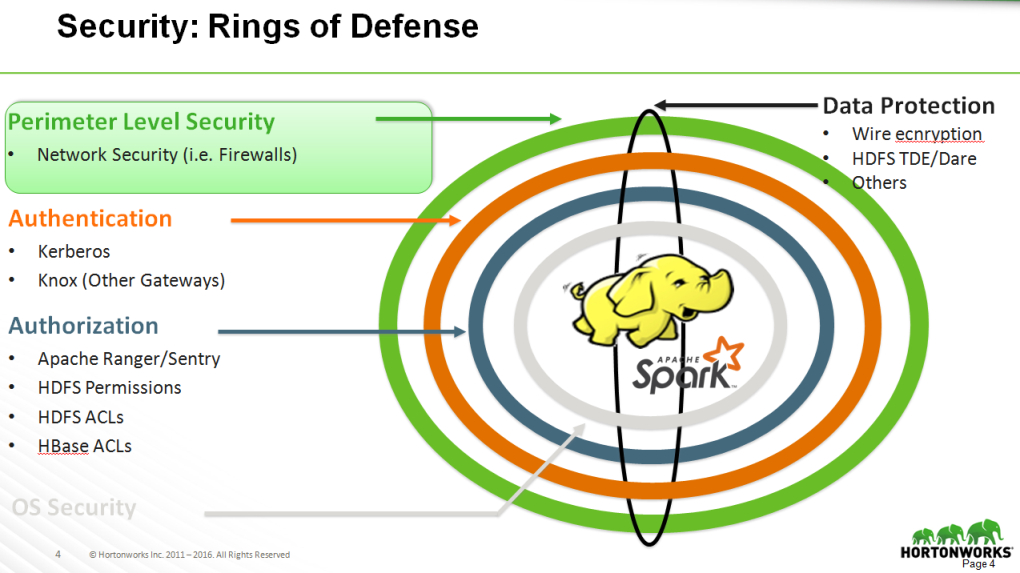
[](http://www.slideshare.net/HadoopSummit/state-of-security-apache-spark-apache-zeppelin-63959610)

Diagram [Ref](http://www.slideshare.net/HadoopSummit/state-of-security-apache-spark-apache-zeppelin-63959610)

Enable security via spark.authenticate property (defaults to false).

See org.apache.spark.SecurityManager

Enable INFO for org.apache.spark.SecurityManager to see messages regarding security in Spark.

Enable DEBUG for org.apache.spark.SecurityManager to see messages regarding SSL in Spark, namely file server and Akka.

**The application web UI at http://<driver>:4040 lists Spark properties in the “Environment” tab**

Following are the security and encryption setting in spark

**#### Security**

|  |  |  |
| --- | --- | --- |
| **Property Name** | **Default** | **Meaning** |
| spark.acls.enable | false | Whether Spark acls should be enabled. If enabled, this checks to see if the user has access permissions to view or modify the job. Note this requires the user to be known, so if the user comes across as null no checks are done. Filters can be used with the UI to authenticate and set the user. |
| spark.admin.acls | Empty | Comma separated list of users/administrators that have view and modify access to all Spark jobs. This can be used if you run on a shared cluster and have a set of administrators or devs who help debug when things do not work. Putting a "\*" in the list means any user can have the privilege of admin. |
| spark.admin.acls.groups | Empty | Comma separated list of groups that have view and modify access to all Spark jobs. This can be used if you have a set of administrators or developers who help maintain and debug the underlying infrastructure. Putting a "\*" in the list means any user in any group can have the privilege of admin. The user groups are obtained from the instance of the groups mapping provider specified by spark.user.groups.mapping. Check the entryspark.user.groups.mapping for more details. |
| spark.user.groups.mapping | org.apache.spark.security.ShellBasedGroupsMappingProvider | The list of groups for a user are determined by a group mapping service defined by the trait org.apache.spark.security.GroupMappingServiceProvider which can configured by this property. A default unix shell based implementation is provided org.apache.spark.security.ShellBasedGroupsMappingProviderwhich can be specified to resolve a list of groups for a user. *Note:* This implementation supports only a Unix/Linux based environment. Windows environment is currently **not** supported. However, a new platform/protocol can be supported by implementing the trait org.apache.spark.security.GroupMappingServiceProvider. |
| spark.authenticate | false | Whether Spark authenticates its internal connections. Seespark.authenticate.secret if not running on YARN. |
| spark.authenticate.secret | None | Set the secret key used for Spark to authenticate between components. This needs to be set if not running on YARN and authentication is enabled. |
| spark.authenticate.enableSaslEncryption | false | Enable encrypted communication when authentication is enabled. This is supported by the block transfer service and the RPC endpoints. |
| spark.network.sasl.serverAlwaysEncrypt | false | Disable unencrypted connections for services that support SASL authentication. This is currently supported by the external shuffle service. |
| spark.core.connection.ack.wait.timeout | spark.network.timeout | How long for the connection to wait for ack to occur before timing out and giving up. To avoid unwilling timeout caused by long pause like GC, you can set larger value. |
| spark.core.connection.auth.wait.timeout | 30s | How long for the connection to wait for authentication to occur before timing out and giving up. |
| spark.modify.acls | Empty | Comma separated list of users that have modify access to the Spark job. By default only the user that started the Spark job has access to modify it (kill it for example). Putting a "\*" in the list means any user can have access to modify it. |
| spark.modify.acls.groups | Empty | Comma separated list of groups that have modify access to the Spark job. This can be used if you have a set of administrators or developers from the same team to have access to control the job. Putting a "\*" in the list means any user in any group has the access to modify the Spark job. The user groups are obtained from the instance of the groups mapping provider specified byspark.user.groups.mapping. Check the entry spark.user.groups.mapping for more details. |
| spark.ui.filters | None | Comma separated list of filter class names to apply to the Spark web UI. The filter should be a standard [javax servlet Filter](http://docs.oracle.com/javaee/6/api/javax/servlet/Filter.html). Parameters to each filter can also be specified by setting a java system property of:  spark.<class name of filter>.params='param1=value1,param2=value2' For example:  -Dspark.ui.filters=com.test.filter1  -Dspark.com.test.filter1.params='param1=foo,param2=testing' |
| spark.ui.view.acls | Empty | Comma separated list of users that have view access to the Spark web ui. By default only the user that started the Spark job has view access. Putting a "\*" in the list means any user can have view access to this Spark job. |
| spark.ui.view.acls.groups | Empty | Comma separated list of groups that have view access to the Spark web ui to view the Spark Job details. This can be used if you have a set of administrators or developers or users who can monitor the Spark job submitted. Putting a "\*" in the list means any user in any group can view the Spark job details on the Spark web ui. The user groups are obtained from the instance of the groups mapping provider specified by spark.user.groups.mapping. Check the entry spark.user.groups.mapping for more details. |

**#### Encryption**

|  |  |  |
| --- | --- | --- |
| **Property Name** | **Default** | **Meaning** |
| spark.ssl.enabled | false | Whether to enable SSL connections on all supported protocols.  When spark.ssl.enabled is configured, spark.ssl.protocol is required.  All the SSL settings like spark.ssl.xxx where xxx is a particular configuration property, denote the global configuration for all the supported protocols. In order to override the global configuration for the particular protocol, the properties must be overwritten in the protocol-specific namespace.  Use spark.ssl.YYY.XXX settings to overwrite the global configuration for particular protocol denoted by YYY. Example values for YYY include fs, ui, standalone, and historyServer. See [SSL Configuration](http://spark.apache.org/docs/latest/security.html#ssl-configuration) for details on hierarchical SSL configuration for services. |
| spark.ssl.enabledAlgorithms | Empty | A comma separated list of ciphers. The specified ciphers must be supported by JVM. The reference list of protocols one can find on [this](https://blogs.oracle.com/java-platform-group/entry/diagnosing_tls_ssl_and_https) page. Note: If not set, it will use the default cipher suites of JVM. |
| spark.ssl.keyPassword | None | A password to the private key in key-store. |
| spark.ssl.keyStore | None | A path to a key-store file. The path can be absolute or relative to the directory where the component is started in. |
| spark.ssl.keyStorePassword | None | A password to the key-store. |
| spark.ssl.keyStoreType | JKS | The type of the key-store. |
| spark.ssl.protocol | None | A protocol name. The protocol must be supported by JVM. The reference list of protocols one can find on [this](https://blogs.oracle.com/java-platform-group/entry/diagnosing_tls_ssl_and_https) page. |
| spark.ssl.needClientAuth | false | Set true if SSL needs client authentication. |
| spark.ssl.trustStore | None | A path to a trust-store file. The path can be absolute or relative to the directory where the component is started in. |
| spark.ssl.trustStorePassword | None | A password to the trust-store. |
| spark.ssl.trustStoreType | JKS | The type of the trust-store. |

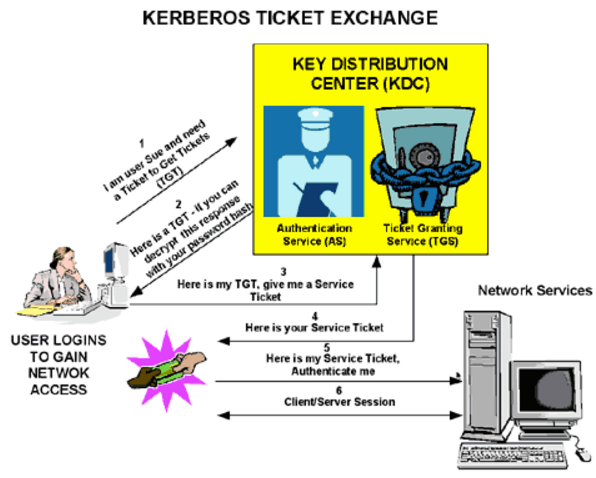
# Explain Kerberos security? [Ref](https://msdn.microsoft.com/en-us/library/bb742516.aspx)

The Kerberos protocol name is based on the three- headed dog figure from Greek mythology known as Kerberos. The three heads of Kerberos comprise the Key Distribution Center (KDC), the client user and the server with the desired service to access. The KDC is installed as part of the domain controller and performs two service functions: the Authentication Service (AS) and the Ticket-Granting Service (TGS). As exemplified in Figure below, three exchanges are involved when the client initially accesses a server resource:

AS Exchange

TGS Exchange

Client/Server (CS) Exchange



**AS Exchange**

When initially logging on to a network, users must negotiate access by providing a log-in name and password in order to be verified by the AS portion of a KDC within their domain. The KDC has access to Active Directory user account information. Once successfully authenticated, the user is granted a Ticket to Get Tickets (TGT) that is valid for the local domain. The TGT has a default lifetime of 10 hours and may be renewed throughout the user's log-on session without requiring the user to re-enter his password. The TGT is cached on the local machine in volatile memory space and used to request sessions with services throughout the network. The following is a discussion of the TGT retrieval process.

**TGS Exchange**

The user presents the TGT to the TGS portion of the KDC when desiring access to a server service. The TGS on the KDC authenticates the user's TGT and creates a ticket and session key for both the client and the remote server. This information, known as the service ticket, is then cached locally on the client machine.

The TGS receives the client's TGT and reads it using its own key. If the TGS approves of the client's request, a service ticket is generated for both the client and the target server. The client reads its portion using the TGS session key retrieved earlier from the AS reply. The client presents the server portion of the TGS reply to the target server in the client/server exchange coming next.

**Client/Server Exchange**

Once the client user has the client/server service ticket, he can establish the session with the server service. The server can decrypt the information coming indirectly from the TGS using its own long-term key with the KDC. The service ticket is then used to authenticate the client user and establish a service session between the server and client. After the ticket's lifetime is exceeded, the service ticket must be renewed to use the service.

# Name various types of distributing does spark support?

Both Big Computation that CPU intensive and Big data that is heavy disk I/O intensive.

# Consider the following SQL query: Select pName, sum(rate) from sales where (year = 2015) group by pName; Explain the functioning in Map and Reduce when the above query is executed in spark using dataSets.

The filtering that is "where" clause in SQL is performed in Map tasks and the aggregrate function "sum(rate)" is performed by reduce tasks.

# What are the conditions where spark driver can parallelize dataSets as RDDs?

* The dataset must fit into memory on the Spark driver process
* Such RDDs can be created using the parallelize() method
* Any serializable variable or data structure on the driver will be transmitted to the worker node

# Can repartition() operation decrease number of partition? More Details [Ref](https://hackernoon.com/managing-spark-partitions-with-coalesce-and-repartition-4050c57ad5c4#.dru4yixjm)

Yes, repartition() can increase or decrease the number of partition however coalesce() can only decrease the number of partitions. The repartition algorithm does a full shuffle of the data and creates equal sized partitions of data however coalesce combines existing partitions to avoid a full shuffle and may result in unequal partitions. repartition() creates equal partitions.

# What is the drawback of repartition() and coalesce() operations?

Both will trigger shuffle.

# In a join operaton for example val joinVal = rddA.join(rddB) will it generate partition?

Yes and No, if the RDDs are of different size than it will generate shuffle else it will not.

# Consider the following code in spark, what is the final value in fVal variable?

val num = spark.parallelize (Array(5, 2, 4, 3, 2, 6, 7, 2, 9))

val fVal = num.distinct.filter( \_ % 2 == 0).reduce( \_ + \_ )

Ans: Find all even numbers and find the sum for all of them. 2+4+2+6+2 = 16, thus fval will return 16

# Scala pattern matching - Show various ways code can be written?

1. def patternMatch(x: double): Double = x match {

case 0 => 1.0

case x if (x <0) => 0

case x => x \* patternMatch(x - 1)

2. val patternMatch = rdd.reduceByKey({ case ((a, aCount), (b, bCount)) => (a + b, aCount + bCount) })

3. try {

val companyUrl = new URL("http://www.company.com")

processCompanyUrl (companyUrl)

} catch {

case x: SQLException => println("Check SQL: " + x)

case x: MalformedURLException => println("Check URL: " + x)

case x: IOException => println("Check IO: " + x)

case \_ => println("Un Handled Exception: " + x)

}

# What is the return result when query is executed using Spark SQL or HIVE? Hint: RDD or dataframe/dataset?

DataFrame/DataSet reference

# If we want to display just the schema of a dataframe/dataset what method is called?

Schema can be visualized using the printSchema() method

dF. printSchema() where dF is dataframe/dataset

# Show various implementations for the following query in spark?

Table name is employee and dataframe name is dF

Query: select name, AVG(salary) from employee where country = “USA” groupby name, salary;

Method 1: Using scala code in spark:

dF.filter(“country = ‘USA’”) // Filter them

.groupBy(“name”, “salary”) // group them

.agg(dF(“name”), avg(“salary”) // Aggregate

.show() // display first 20 lines from the result

Method 2: Using sqlContext – DataFrame/DataSet implementation

dF.createOrReplaceTempView("employee") // Register the table

val empQuery = “select name, AVG(salary) from employee where country = “USA” groupby name, salary;”

sqlContext.sql(empQuery) // execute the query using sqlContext

.show() // display first 20 lines from the result

Method 3: Using hiveContext – if table exists in hive

dF.createOrReplaceTempView("employee") // Register the table

val empQuery = “select name, AVG(salary) from employee where country = “USA” groupby name, salary;”

hiveContext.sql(empQuery) // execute the query using hiveContext

.show() // display first 20 lines from the result

# What are the most important factors you want to consider when you start machine learning project?

* Clean data – garbage in garbage out
* Understanding of MLib component of spark and algorithms
* Business Domain knowledge
* Good understanding of Statics concepts

# As a data scientist which algorithm will you suggest if legal aspects and ease of explanation to non technical people are the main criteria?

Decision Tree algorithm

# For supervised learning algorithm, what percentage of data is split between training and test dataset?

Typically 70 % of data is used for training and 30 % of data is used for test.

# Compare performance of Avro and parquet file formats, explain case while using with spark.

* Avro is a row-based storage format for Hadoop.
* Parquet is a column-based storage format for Hadoop.
* If your use case typically scans or retrieves all of the fields in a row in each query, Avro is usually the best choice.
* If your dataset has many columns, and your use case typically involves working with a subset of those columns rather than entire records, Parquet is optimized for that kind of work.

Parquet showed either similar or better results on every test compared to Avro. The query-performance differences on the larger datasets in Parquet’s favor are partly due to the compression results; when querying the wide dataset, Spark had to read 3.5x less data for Parquet than Avro. Avro did not perform well when processing the entire dataset

# Spark Master exposes a set of REST API’s to submit, monitor applications which data format is used for these web services?

JSON

# When you should not use spark?

Spark was not designed as a multi-user environment. Spark users are required to know whether the memory they have access to is sufficient for a dataset.

# What attributes contribute for a function to achieve parallelism?

The function should be Commutative (f(x, y) = f(y, x)) and Associative(f(f(x, y), z) = f(f(x, z), y) = f(f(y, z), x)) so that it can be computed correctly in parallel. For example to Aggregate the elements of the dataset using a function func (which takes two arguments and returns one) should be commutative and associative. Accumulators are variables that are only “added” to through an associative and commutative operation and can therefore be efficiently supported in parallel.

# What are various types of Partitioning in Apache Spark?

* Hash Partitioning in Spark
* Range Partitioning in Spark

|  |  |
| --- | --- |
| **Hash Partition** | **Range Partition** |
|  |  |

Hash Partitioning attempts to spread the data evenly across various partitions based on the key. Object.hashCode method is used to determine the partition in Spark as partition = key.hashCode () % numPartitions.

Some Spark RDDs have keys that follow a particular ordering, for such RDDs, range partitioning is an efficient partitioning technique. In range partitioning method, tuples having keys within the same range will appear on the same machine. Keys in a range partitioner are partitioned based on the set of sorted range of keys and ordering of keys.

Spark’s range partitioning and hash partitioning techniques are ideal for various spark use cases but spark does allow users to fine tune how their RDD is partitioned, by using custom partitioner objects. Custom Spark partitioning is available only for pair RDDs i.e. RDDs with key value pairs as the elements can be grouped based on a function of each key. Spark does not provide explicit control of which key will go to which worker node but it ensures that a set of keys will appear together on some node.

Spark maintained individal partition buffers/local files for <= 200 (default). If partition is more than default per node it will switch back to a single sorted file. Thus maintains a balance or control.

# How to set partitioning for data in Apache Spark?

RDDs can be created with specific partitioning in two ways –

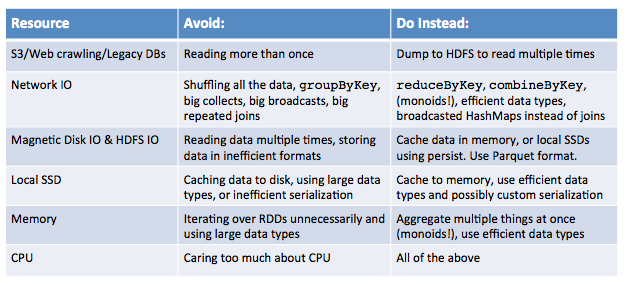
1. Providing explicit partitioner by calling partitionBy method on an RDD,
2. Applying transformations that return RDDs with specific partitioners. Some operation on RDDs that hold to and propagate a partitioner are

* Join
* LeftOuterJoin
* RightOuterJoin
* groupByKey
* reduceByKey
* foldByKey
* sort
* partitionBy
* foldByKey

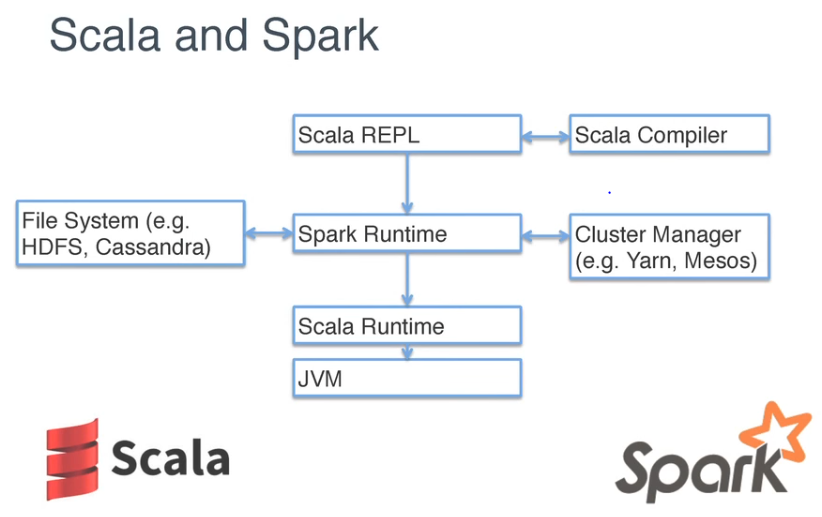
# How does mapWithState function work in cluster?

mapWithState is a shuffle stage, that means it will cause data in your cluster to move around. Each entry (key value pair) will be shuffled to a particular Executor. Upon subsequent arrivals of that same key to whichever Executor was processing it from the input stream at the given time, it will be shuffled to the node holding the in-memory map with the state of previous messages. This is done by default via the HashPartitioner which will hash the key and then send it to the proper Executor holding the state, that's why you need to choose the key carefully.

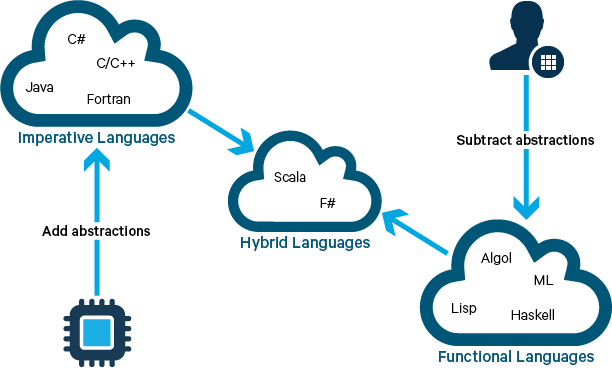
# How will you prioritize optimizations in algorithm design when designing the system? [Ref](https://blog.cloudera.com/blog/2015/08/how-apache-spark-scala-and-functional-programming-made-hard-problems-easy-at-barclays/)



# Explain How Scala and Spark fit in together?



# Explain why Scala is picked to build Spark compared to other functional Programming languages? [Ref](https://blog.cloudera.com/blog/2015/08/how-apache-spark-scala-and-functional-programming-made-hard-problems-easy-at-barclays/)



The absence of “state” and “mutation” in Scala makes reasoning about the evaluation of expressions, especially in parallel systems, much easier than imperative languages like Java or Python. This characteristic is essential for harnessing modern hardware that has more cores, more nodes, and more RAM, rather than faster clock speeds. Moore’s Law is dead, [it’s time to get functional](http://www.drdobbs.com/tools/its-time-to-get-good-at-functional-progr/212201710)!

Adding syntactic sugar to easily do lambda expressions is not sufficient to make a language functional, this is just the start. Functional languages provide many other features and tools that encourage coders to make an entire paradigm shift to a more mathematical way of thinking.

By narrowing the focus to only a small number of compose-able abstract concepts, such as sets, sequences, numbers, functions, function composition, and abstract algebra, functional programming achieves significant expressive power. For example, machine-learning algorithms can take hundreds of lines of imperative code to implement yet they can be defined in just a handful of equations.

By taking less time typing, a functional programmer can spend more time thinking—which is perhaps why the shift to Scala has gained a reputation for being difficult. Moreover, since there is less code to go wrong and the compiler can check each keystroke, bugs are reduced to the point where “debugging” (running code and inspecting its state to know what it does) is a practice that needs to be unlearned.

# Explain how efficiently data types should be used in spark?

*Premature optimization is the root of all evil.* – Donald Knuth

For big data, this statement is wrong. Knuth’s argument *against* premature optimization is that it usually makes program logic harder to debug. But given that functional programming virtually eliminates bugs, the main residual source of bugs in big data code is caused by resource issues. But identifying resource issues is hard because Spark’s powerful functional pipelining of stages means that if your code doesn’t work, it’s hard to tell *which* part of the code is to blame. So optimizing simple things like data types from the start quickly pays off.

Try to pack as much data as possible into [as few bytes as possible](http://blog.cloudera.com/blog/2015/05/working-with-apache-spark-or-how-i-learned-to-stop-worrying-and-love-the-shuffle/). Avoid Strings, index them. Pack “192.168.0.1” into an Integer. Use domain-specific knowledge. (For example, we knew the range of values for six Integers in a case class was limited, so we packed them all into a Long!) Indexing can add some time to your job, but with Spark’s zipWithIndex, it shouldn’t take much effort to code. In fact, we crush many of our datasets into HashMaps, which we can then broadcast over the cluster, eliminating joins and giving us great flexibility in our algorithm design. All these steps in the job contribute a “constant” time cost, which means that as the data scales, and in our case the number of queries scales, these constant time jobs of packing, indexing, caching, and broadcasting virtually don’t change.

More code means more tests. We used ScalaCheck’s powerful property based testing (PBT) DSL to write an exhaustive test suite. In PBT we define properties, not cases. (We just tell the framework that our data packing and unpacking functions need to compose to produce the identity function. ScalaCheck does the work of finding edge cases, counter examples, off-by-one errors, and so on.) As discussed previously, this framework also gives us the required confidence regarding near-absolute correctness.

# How can we debug query execution?

Use package org.apache.spark.sql.execution.debug which work on your Datasets and Spark Session. Now use method debug() or debugCodegen() depending on where you want to debug.