**Introduction to Spark**

**Cluster Computing** – Cluster is a group of service entities that work together to provide a more scalable and usable service platform than a single service entity. In the client, a cluster is like a service entity, but in fact, a cluster is made up of a set of service entities. Compared with a single service entity, the cluster provides the following two key features:

* *Scalability* – supports dynamic addition of new service entities thereby enhancing the performance of the cluster
* *Increased Availability* – since the same service can be provided using multiple service entities, a cluster enables that the system is available all the time

**Distributed Computing** – Distributed computing involves distributed systems (located nearby or far apart) and refers to deploying different service modules in many different servers and providing services externally through remote invoking collaborative work.

The major difference between the two is that the distribution is in parallel, and the clusters are connected in series. Also, each node in the distribution can be clustered, however, clusters don't necessarily have to be distributed. We use both distributed and cluster computing in conjunction where, distributed is to shorten the execution time of a single task to improve efficiency, and clusters improve efficiency by increasing the number of tasks executed per unit time. Spark aims to offer solutions surrounding this area where we can perform operations at scale and at the same time, programmatically distribute the resources in order to achieve higher efficiency.

Spark is a platform for cluster computing. It enables distributing the data over multiple nodes (like a separate processing unit) and perform computations over clusters across the nodes. Splitting the data helps us handle large datasets and the execution time is less, since each node works only on a small independent portion of data. This enables parallel processing and computation, in turn reducing the execution time by a large extent.

**Spark Architecture**

With the advent of increasing data and applications around data, it is important for companies to scale projects to a larger extent. Traditional architectures possess problems in scaling, such as but not limited to, memory issues, iterative computing, providing solutions on upstream data, inefficient resource management etc. Spark aims to overcome these problems.

Spark is a computational engine that is responsible for scheduling, distributing, and monitoring applications consisting of many computational tasks across multiple nodes, or a computing cluster. In particular, Spark can run on Hadoop clusters and can access any Hadoop data source.

The Spark project contains multiple closely integrated components in a unified stack. The unified stack enables improvement across lower layers – thus means, when the core engine is optimized, SQL and machine learning libraries are automatically optimized. It also facilitates seamless operations over multiple different processing models – it means that one can simultaneously view results using SQL while a machine learning algorithm is being executed on the same upstream data.

**Hadoop HDFS Architecture**

Rack 1 Rack 2 Rack 3

|  |
| --- |
| Data Node 1 |
| Data Node 2 |
| … |
| … |
| … |
| Data Node k |

Switch

Switch

Switch

Core

**Name Node**

**Master**

Hadoop has a master-slave architecture where the nodes (individual systems or servers inside a cluster) of a cluster are communication with each other (in case of distributed computing). The master-slave architecture can have either:

* Star Topology – standalone system to act like a master, and rest are slaves
* Token-Ring Topology – a system is chosen to be the master; this is generally decided by the system which bears the token

Name Nodes act as a master and is connected with multiple Data Nodes (nodes inside a rack which is part of a cluster). The master is responsible for the following activities:

1. Handling requests from clients and allocating data nodes to the client
2. Maintaining Edit logs – this contains the changes and immediate logging of the real-time updates. The edit log is stored in RAM and is generally managed by a secondary name node. This is to have a dedicated node for tracking the process and also helps in hedging the risk in case the main node fails
3. Creating an FsImage – FsImage acts like a history node (contains the complete directory structure of HDFS); it is created the whenever a data node is created for the first time. FsImage keeps a track of the occupied and vacant data nodes

The secondary name node makes sure that the edit log is updated and is compiled with the FsImage (known as checkpointing).

Data node communicates with name node for:

* Acknowledgement of a command or instruction received from the name node
* Information about the action taken by the data node in order to process data (this can be read, write, execute, or any other data process) – known as block report
* Sends ‘heartbeat’ at regular intervals to notify the master that the slave is working properly. If a name node fails to receive the heartbeat from the data node, it is an indication of node failure. In this case, the task is assigned to another data node

**Process Flow:**

1. Client requests the name node to store data (initial request is made upon connection is established)
2. Name node returns IP addresses of the data nodes to the client, where data can be stored. For example, name node returns IP 1, 3, and 6 to the client where the data can be stored
3. The number of IP addresses returned by the name node depends on the ‘Replication Factor’. A replication factor determines the number of data nodes into which the data would be stored (as a copy or replica). This is to ensure that the data is safe; even if one node fails, there are other nodes from which we can access the data
4. After a client chooses an IP (let’s say 1 in this case) then the data is initially stored in the data node with IP address 1. Further, this data node (with IP 1) replicates the same into 3. Then the data node with IP 3 replicates it to another IP. This replication happens serially from one node to another
5. Upon completion of the operation, data node sends an acknowledgement to the client which then transfers it to the name node (the same block report is transferred from the data node to client, and then to name node)
6. Finally, after the name node has received the block report, it updates the edit log and creates an FsImage

The replication factor can be chosen by a user upon considering the risk associated with data. Important data can have large replication factor to avoid the risk of losing it. By default, the replication factor is 3. Hadoop stores X GB of data into parts each of 128 MB (called a block), and processing happens block wise.

Note: Even though the replication happens serially, data is read parallelly. It means that if a data node fails, the name node (having the complete structure of the directory) is able to process the data parallelly through other data nodes.

**MapReduce Model**

Count using key-value pair

Mapping – Split into 3 parts

Consider a simple text file

A B C

D D C

A D B

Text File

Text File

A B C

D D C

A D B

A 1, B 1, C 1

D 1, D 1, C 1

A 1, D 1, B 1

Reducing – counting values of like items

A 2

B 2

C 2

D 2

Shuffling – keep like items together

A 1 1, B 1 1, C 1 1, D 1 1 1

Above is the MapReduce model (example showcases counting different elements of the content of a text file) which is extended by Spark. Instead of writing a Java code for each of the individual components, Spark provides us with a packaged form of the complete model with customizations.

**Limitations of Hadoop:**

The major limitation of Hadoop is seen in terms of hardware requirements to increase the speed of execution or data access, which leads to

* failure in iterative and real-time processing (upstream processing)
* hard to customize MapReduce scripts for specific use case – complex code structure in certain cases

Although Spark overcomes these limitations, it poses certain challenges of its own:

* Costly to set-up the infrastructure
* Since Spark relies heavily on RAM for computation, it fails when the requirement exceeds RAM space
* Spark is best for iterative computation where one needs to perform same task on a large set of data. In case of serial/linear tasks, Hadoop performs the best
* Data spilling is a big issue in Spark. When the data is being mapped, if the buffer size (128 MB, where the mapped data is being stored) is less than the individual part size, it leads to data leakage, also known as spilling. Hence, finding the right buffer size is important

**Driver Programs, SparkContext, and Executors**

Every Spark application consists of a driver program that launches various parallel operations on a cluster. The driver program also contains the application’s *main()* function and defines the distributed datasets on the cluster for further operation.

Inside the driver program, the connection to Spark is represented by a SparkContext instance (representing a connection to a computing cluster), which is the entry point to any spark functionality. When we run any Spark application, a driver program starts, and the SparkContext is initiated here. The driver program then runs the operations inside the executors on worker nodes.

A SparkContext can be locally instantiated as:

sc = SparkContext(‘local’, ‘mySparkApp’)

By default, PySpark has SparkContext available as ‘sc’, and hence creation of new SparkContext won’t work. It can also be used to control various Spark configurations such as defining executor memory, setting number of cores for an executor, defining limits on cores and memory etc.

Every spark application requires 4 things:

1. Driver Memory – it is the memory where the result of any action is collected
2. Executor Memory – it specifies how much RAM is needed in order to allocate to each executor
3. Executor Cores – it defines the number of cores to be made available to each executor
4. Number of Executors – to be assigned to each job

**How Spark performs a task?**

1. First step is to define a job, which will be the final outcome which is required
2. Spark breaks the job into intermediate steps, called as stages
3. Each stage has individual task or set of tasks to perform. A task is the smallest unit of work that needs to be done
4. An executor then takes a task and/or set of tasks to complete. An executor is a program (that is a container in itself – a container is a software unit that packages the code, dependencies, and requirements needed to perform a task) which runs certain tasks
5. The executor takes in the resources to perform the task (we can specify the number of resources required for an executor to perform a task, e.g.- 1 GB memory, 2 cores etc.)

An executor can be viewed as a box consisting of cores and memory, which takes task as an input and throws data as the output.

**Job Scheduling in Spark**

**FIFO (First-in First-out)** – By default, the framework allocates the resources in FIFO manner. This means that the first defined job will get the priority for all available resources. If this first job doesn't need all resources, that's fine because other jobs can use them too. But if it's not the case, the remaining jobs must wait until the first job frees them. It can be problematic especially when the first job is a long-running one and the remaining execute much faster.

**FAIR Scheduling** – FAIR scheduling mode works in round-robin manner. This algorithm equally distributes the resources to the tasks. This means that a task doesn’t have to wait for another task to complete before it gets completed. FAIR scheduling method brings also the concept of pools. The pool is a concept used to group different jobs inside the same logical unit. Each pool can have different properties, like weight which is a kind of importance notion, minShare to define the minimum reserved capacity and schedulingMode to say whether the jobs within given pool are scheduled in FIFO or FAIR manner. Hence, pools are a great way to separate the resources between different clients. If one of the executed jobs is more important than the others, you can increase its weight and minimum capacity in order to guarantee its quick termination.

**Lazy Evaluation in Spark**

Lazy evaluation means that when we perform an operation (e.g.– calling a transformation on an RDD), the operation is not immediately performed. Instead, Spark internally records metadata to indicate that this operation has been requested. Rather than thinking of an RDD as containing specific data, it is best to think of each RDD as consisting of instructions on how to compute the data that we build up through operations. Loading data also happens in similar manner. The data is not loaded unless an action is called on the data.

**Spark RDDs (Resilient Distributed Datasets)**

RDDs are the elements that run and operate on multiple nodes to do parallel processing on a cluster. RDDs are immutable elements, which means once you create an RDD you cannot change it. RDDs are fault tolerant as well, hence in case of any failure, they recover automatically. There are two ways of applying operations on RDDs:

* Transformation – These are the operations, which are applied on a RDD to create a new RDD. Filter, groupBy and map are the examples of transformations, e.g.– map, filter, flatMap etc.
* Action – These are the operations that are applied on RDD, which instructs Spark to perform computation and send the result back to the driver, e.g.– collect

To apply any operation in PySpark, we need to create a PySpark RDD first. This can be achieved using the function *parallelize()*. The following code in a Python file creates RDD words, which stores a set of words mentioned.

*words = sc.parallelize (*

*["scala", "java", "hadoop", "spark", "akka", "spark vs hadoop", "pyspark", "pyspark and spark"]*

*)*

The *sc.parallelize()* calls the parallelize function from the SparkContext instance, denoted by *sc*. The arguments inside the function can be anything. It is used to store the data in the RDD form. We can define the number of data partitions inside the parallelize function as *sc.parallelize(data, 10)*

Spark RDDs are by default recomputed each time one runs an action on them. If we would like to reuse an RDD in multiple actions, we can persist it using rdd\_name.persist(). This will store the RDD contents in the memory (partitioned across the machines in a cluster) and reuse them in future actions. We can also persist RDDs on disk.

**Spark Transformations**

Transformations are kind of operations which will transform RDD data from one form to another. When applied on any RDD, a new RDD with transformed data is created (since, RDDs in Spark are immutable). Transformations applied on any RDD will not perform the operation immediately. It will create a DAG (Directed Acyclic Graph) using the applied operation, source RDD and function used for transformation. It will keep on building this graph using the references till we apply any action operation on the last lined up RDD (lazy evaluation).

There are two kinds of transformations:

* **Narrow** – In narrow transformation, all the elements that are required to compute the records in single partition live in single partition of parent RDD, i.e., data doesn’t have to be processed across the partitions and we retain the original partition (no modification). For example, *filter(), map(), union()* etc.
* **Wide** – In wide transformation, all the elements that are required to compute the records in single partition may live in many partitions of parent RDD. This partition may live in many partitions of parent RDD, i.e., data needs to be moved across partitions which requires intermediate result. For example, *groupbyKey(), reducebyKey(), sort()* etc.

Wide transformations will be performed in different stages of DAG, while narrow transformations are performed in single stage. This is because every wide transformation generates an intermediate output and hence, following operations have to wait for this intermediate result to get computed before the operations can be performed.

**Spark Actions**

An action is one of the ways of sending data from executor to the driver. Some example are *collect(), count()* etc.

Some common transformations and actions on RDDs are:

* **count() action** – counts the number of elements in the RDD
  + Can be called using *rdd\_name.count()*
* **collect() action** – returns the elements in the RDD
  + can be called using *words.collect()*
* **foreach(f) action** – returns only those elements which meet the condition of the function inside foreach. The function can be anything, for example, print
  + can be called using *words.foreach(f)* – where *def f(x): print(x)* – prints all the elements of *words*
* **filter(f) transformation** – a new RDD is returned containing the elements, which satisfies the function inside the filter. Again, a function is called inside filter
  + can be called using *words.filter(lambda x: 'spark' in x)* – filter for ‘spark’ in *words*
* **map(f, preservingPartitioning = False) transformation** – a new RDD is returned by applying a function to each element in the RDD. The transformed RDD us iterable of iterables (e.g.– list inside a list)
  + can be called using *words.map(lambda x: (x, 1))* – returns a tuple with 1 across every element of *words*, for example (‘scala’, 1)
* **reduce(f) action** – after performing the specified commutative and associative binary operation, the element in the RDD is returned
  + *nums = sc.parallelize([1, 2, 3, 4, 5])*
  + *nums.reduce(add)* – will return 15 (upon adding all the elements of num)
* **join(other, numPartitions = None) transformation** – it returns RDD with a pair of elements with the matching keys and all the values for that particular key
  + *x = sc.parallelize([("spark", 1), ("hadoop", 4)])*
  + *y = sc.parallelize([("spark", 2), ("hadoop", 5)])*
  + *x.join(y)* – will join x and y into a single RDD as *[(‘spark’, (1, 2)), (‘hadoop’, (4, 5))]*
* **cache() action** – persist this RDD with the default storage level (MEMORY\_ONLY). You can also check if the RDD is cached or not
  + *words.cache()* – to persist the RDD in cache
  + *words.persist().is\_cached* – to check whether the RDD has been cached or not
* **union() transformation** – to merge two RDDs if they have the same structure
  + *rdd\_1.union(rdd\_2)* – will return the RDD with combined elements
* **intersection() transformation** – gives us the common terms or objects from two RDDs which need not have the same structure
  + *rdd\_1.intersection(rdd\_2)* – returns the common element to both RDDs
* **distinct() transformation** – used to get unique elements in an RDD
  + *rdd\_name.distinct()* – returns distinct elements of the RDD
* **sample(withReplacement, fraction, seed) transformation** – used to pick sample RDD from a larger RDD. It is frequently used in ML operations. *fraction* means the percentage of total data to take the sample from
  + *rdd\_name.sample(True, 0.5, 3)* – True is for sample with replacement, 0.5 denotes the fraction, and 3 denotes the sample seed
* **flatMap() transformation** – applies changes similar to map but returns only an iterable holding entre RDD contents
* **cartesian() transformation** – to compute the cartesian product of two RDDs which will return all possible pairs of (a, b) where a is the source RDD and b is the other RDD. The cartesian product can be useful when we wish to consider the similarity between all possible pairs, such as computing every user’s expected interest in each offer. It is however, computationally very expensive
* **fold() action** – similar to reduce; key difference between fold and reduce is that, reduce throws an exception for empty collection, but fold() is defined for empty collection (used for initial call on each partition)
* **take(n) action** – returns the number of elements from RDD. It tries to cut the number of partition it accesses, so it represents a biased collection (order of elements can’t be presumed)
* **top(n) action** – extract top elements from RDD. It uses default ordering of data
* **countByValue() action** – returns number of times each element occurs in RDD in the form of tuple, for example (‘a’, 1), (‘b’, 3) etc.

Key-Value Pair and Partitioning RDDs

**Spark Configurations**

Spark configurations can be provided in 3 ways:

1. Before starting the application (spark-submit command in command line)
2. While defining SparkContext (inside the code – using *sc.config()*)
3. Defining a configuration file (using Key:Value pairs)

**Application Properties Configurations**

It indicates how shall the resources behave in the Spark environment. Some of the important configurations are:

1. spark.driver.name – providing name to a Spark application
2. spark.driver.cores – to allocate the amount of CPU cores to a driver (default value is 2)

There are 3 types of resources in Spark – CPU, memory (RAM), and disk space. The driver negotiates for resources. For example, 2 cores/executors. Note that one can define the number of cores only when Spark is running in ‘Client’ mode because something outside YARN can’t be controlled using YARN functionalities.

1. spark.driver.maxResultSize – defines the maximum size of the result which is returned to the driver. By default, it is 1 GB i.e. if the result exceeds 1 GB, the job fails
2. spark.driver.memory – defines the RAM size

This is again defined in cluster mode. When the Spark is running in Client mode, one can define the memory in command line in the following manner

--driver-memory 5g <filename>.py

We can’t define the memory inside the code in client mode because once a code is submitted in client mode, JVM (Java Virtual Machine) is started and it can’t be changed during execution. Whereas in cluster mode, JVM is initiated after SparkContext is called.

1. spark.driver.memoryOverhead – when a memory is full, the overhead memory is defined so that the extra data is spilled into the disk. Defining the above configuration facilitates the amount of memory that can be spilled in the disk (for example 5 GB of memory overhead can be spilled into the disk). Note that this can only be defined in cluster mode
2. spark.local.dir – this is used in conjunction with the above configuration to define the location where the spilled data goes into the disk
3. spark.master – defines the type of master – local, standalone, YARN, or Mesos

Standalone mode has no supporting systems; it’s just Spark running on its own

1. spark.submit.deployMode – to define how to run the application – cluster/client mode

**Memory Configurations**

In Spark, both execution and storage is done in RAM and execution is preferred over storage. If the memory doesn’t have enough space to undergo execution, some storage memory is fired by spilling data into disk. Spark handles such situations by performing ***Dynamic Allocation***. In this case, the memory scales up and down on the basis of job it has at hand.

Generally, 60% of space is shared by execution and storage. The remaining 40% of space is used for storage of data structures and reversal (in case of fallback).

1. spark.memory.fraction – sets the above fraction split

Out of the 60%, memory is split equally for execution and storage (i.e. 50% each). Since, data stored in storage can’t be evicted (spilled to disk in this case), execution is forced to run slow when storage memory is more.

1. spark.memory.storageFraction – sets the split of the execution + storage memory
2. spark.memory.ofHeap.enabled – flag to enable data spill into disk
3. spark.memory.ofHeap.size – used to set the maximum of data spill. If the spill increases the specified limit, job fails
4. spark.cleaner.periodicGC.interval – specifies the period after which garbage collection will be performed (defined in milli seconds)

When the memory is full because of excessive garbage, executor runs slowly, and 2 things are possible – executor is failed or speculation.

**Speculation** – If a task is running slowly than the expected time, Spark relaunches the task either with the same executor or different executor. This expected time is calculated as 1.5\*median time of the previously ran tasks. We take median instead of average to avoid skewness (average can be skewed towards the time taken by the lowest or highest number of cores for an executor).

Speculation happens only after a specified time (or threshold), i.e., after a set number of tasks are completed so that the median value is available to calculate the expected time.

**Scheduling Configurations**

A Spark has two scheduling modes – FIFO and FAIR. Some configurations are:

1. spark.scheduler.mode – can be set to either FIFO or FAIR mode
2. spark.speculation – can be set to TRUE to enable speculation
3. spark.speculation.interval – defined as period in milli seconds, i.e. after the every period speculation is assessed
4. spark.task.cups – set to define the number of cores each task is provided to run (optimization at task level). Default value is 2
5. spark.task.maxFailures – every time a task fails, it is retried these many number of times (default value is 4)

Reasons when a task runs slowly:

* Excessive garbage
* Resource problem where resources are not available to execute a task

**Executor Behavior Configurations**

These configurations are used to manage executors. Some important configurations are defined below:

1. spark.executor.cores – defines the number of cores to be allocated to a single executor
2. spark.executor,instances – number of executors to be started for an application
3. spark.executor.heartbeatInterval – defines the period of heartbeat in milli seconds (alive signal from executor to driver). Default value is 100 ms
4. spark.default.parallelism – defines the number of partitions to have in a resultant RDD. If this config is not given, default is number of cores, it means each single task is assigned to a core. Since, partition translates to number of tasks, we can also call it during partitionby() and repartition

**Deploying Modes in Spark**

* Local
* Standalone
* YARN
* Mesos

Standalone mode can be defined as follows –

1. First start the master
   * *spark-class org.apache.spark.deploy.master.Master*
2. Set up worker nodes
   * *spark-class org.apache.spark.deploy.worker.Worker <ip of the master>*
   * The above will assign worker nodes to the master with the specified IP
3. In order to run an application, we use *spark-submit*. In case of client mode, first the job is submitted to master, and then the job is performed on worker nodes.

**How to choose number of executors?**

Consider a scenario where, a job needs to be run, and we have two options:

* 1 Executor – 16 cores
* 4 Executors – 4 cores each

In this case, the 2nd option is better as it provides fault tolerance.