Spark Applications

Spark Applications consist of a *driver* process and a set of executor processes. The driver process runs your main() function, sits on a node in the cluster, and is responsible for three things: maintaining information about the Spark Application; responding to a user’s program or input; and analyzing, distributing, and scheduling work across the executors (defined momentarily). The driver process is absolutely essential - it’s the heart of a Spark Application and maintains all relevant information during the lifetime of the application.

The *executors* are responsible for actually executing the work that the driver assigns them. This means, each executor is responsible for only two things: executing code assigned to it by the driver and reporting the state of the computation, on that executor, back to the driver node.

The cluster manager controls physical machines and allocates resources to Spark Applications. This can be one of several core cluster managers: Spark’s standalone cluster manager, YARN, or Mesos. Spark, in addition to its cluster mode, also has a *local mode where driver, executor all run under same process and used for testing*.

Spark’s language APIs allow you to run Spark code from other langauges. For the most part, Spark presents some core “concepts” in every language and these concepts are translated into Spark code that runs on the cluster of machines. Spark has two fundamental sets of APIs: the low level “Unstructured” APIs and the higher level Structured APIs. This driver process manifests itself to the user as something called the SparkSession. The *SparkSession* instance is the way Spark exeutes user-defined manipulations across the cluster.

You can express your business logic in either language and Spark will compile that logic down to an underlying plan (that we see in the explain plan) before actually executing your code. There is no performance difference between writing SQL queries or writing DataFrame code, they both “compile” to the same underlying plan that we specify in DataFrame code.

Fig 2

A *DataFrame* is the most common Structured API and simply represents a table of data with rows and columns. The list of columns and the types in those columns the *schema*. A simple analogy would be a spreadsheet with named columns. The fundamental difference is that while a spreadsheet sits on one computer in one specific location, a Spark DataFrame can span thousands of computers.

Fig 3

In order to allow every executor to perform work in parallel, Spark breaks up the data into chunks, called partitions. A *partition* is a collection of rows that sit on one physical machine in our cluster. If you have one partition, Spark will only have a parallelism of one even if you have thousands of executors. If you have many partitions, but only one executor Spark will still only have a parallelism of one because there is only one computation resource. With DataFrames, we do not (for the most part) manipulate partitions individually. We simply specify high level transformations of data in the physical partitions and Spark determines how this work will actually execute on the cluster. Lower level APIs do exist, RDD interface but difficult to write efficient code with it.

In Spark, the core data structures are *immutable* meaning they cannot be changed once created. In order to “change” a DataFrame you will have to instruct Spark how you would like to modify the DataFrame you have into the one that you want, known as Transformation.

Transformations consisting of *narrow dependencies* are those where each input partition will contribute to only one output partition. A *wide dependency* (or wide transformation) style transformation will have input partitions contributing to many output partitions. You will often hear this referred to as a *shuffle* where Spark will exchange partitions across the cluster. With narrow transformations, Spark will automatically perform an operation called pipelining on narrow dependencies, this means that if we specify multiple filters on DataFrames they’ll all be performed in-memory. The same cannot be said for shuffles. When we perform a shuffle, Spark will write the results to disk.

Fig 4 & 5

*Lazy evaulation* means that Spark will wait until the very last moment to execute the graph of computation instructions. In Spark, instead of modifying the data immediately when we express some operation, we build up a *plan* of transformations that we would like to apply to our source data. This provides immense benefits to the end user because Spark can optimize the entire data flow from end to end.

Fig 6

we can see that Spark is building up a plan for how it will execute this across the cluster by looking at the explain plan.

Transformations allow us to build up our logical transformation plan. To trigger the computation, we run an *action*. There are three kinds of actions:

• actions to view data in the console;  
• actions to collect data to native objects in the respective language;

• and actions to write to output data sources.

Explain plans can be read from top to bottom, the top being the end result and the bottom being the source(s) of data. In our case, just take a look at the first keywords. You will see “sort”, “exchange”, and “FileScan”. That’s because the sort of our data is actually a wide dependency because rows will have to be compared with one another.

Fig 8 + code page 15

The logical plan of transformations that we build up defines a lineage for the DataFrame so that at any given point in time Spark knows how to recompute any partition by performing all of the operations it had before on the same input data

flightData2015 = spark\

.read\  
.option(“inferSchema”, “true”)\  
.option(“header”, “true”)\ .csv(“/mnt/defg/flight-data/csv/2015-summary.csv”)

flightData2015.createOrReplaceTempView(“flight\_data\_2015”)

+ Page 17 -> SQL + Explain

Page 21 + 22

UDF

Page 91

Once we go to use the function, there are essentially two different things that occur. If the function is written in Scala or Java then we can use that function within the JVM. This means there will be little performance penalty aside from the fact that we can’t take advantage of code generation capabilities that Spark has for built-in functions.

If the function is written in Python, something quite different happens. Spark will start up a python process on the worker, serialize all of the data to a format that python can understand (remember it was in the JVM before), execute the function row by row on that data in the python process, before finally returning the results of the row operations to the JVM and Spark.

Fig 2

*Starting up this Python process is expensive but the real cost is in serializing the data to Python. This is costly for two reasons, it is an expensive computation but also once the data enters Python, Spark cannot manage the memory of the worker. This means that you could potentially cause a worker to fail if it becomes resource constrained (because both the JVM and python are competing for memory on the same machine).*

## **4. Cache and Persist**

* **Spark provides its own caching mechanisms like persist() and cache().**
* **cache() and persist() will store the dataset in memory.**
* **When you have a small dataset which needs be used multiple times in your program, we cache that dataset.**
* **Cache()   – Always in Memory**
* **Persist() – Memory and disks**

## **8. Level of Parallelism**

* **Parallelism plays a very important role while tuning spark jobs.**
* **Every partition ~ task requires a single core for processing.**
* **There are two ways to maintain the parallelism:**
  + **Repartition: Gives equal number of partitions with high shuffling**
  + **Coalesce: Generally reduces the number of partitions with less shuffling.**

### 2. Hive Bucketing Performance

Bucketing results with a fixed number of files as we specify the number of buckets with a bucket. Hive took the field, calculate the hash and assign a record to that particular bucket. Bucketing is more stable when the field has high cardinality, [Large Data Processing](https://www.xenonstack.com/use-cases/large-data-processing/), and records are evenly distributed among all buckets whereas partitioning works when the cardinality of the partitioning field is low. Bucketing reduces the overhead of sorting files. For Instance, if we are joining two tables that have an equal number of buckets in it, spark joins the data directly as keys already sorted buckets. The number of bucket files can be calculated as several partitions into several buckets.

### 9. Caching in Spark

Caching in [Apache Spark with GPU](https://www.xenonstack.com/insights/what-is-hadoop-with-gpu/) is the best technique for Apache Spark Optimization when we need some data again and again. But it is always not acceptable to cache data. We have to use cache () RDD and DataFrames in the following cases -

* When there is an iterative loop such as in Machine learning algorithms.
* RDD is accessed multiple times in a single job or task.
* Or, the cost to generate the RDD partitions again is higher.<l/i>

Cache () and persist (StorageLevel.MEMORY\_ONLY) can be used in place of each other. Every RDD partition which gets evicted out of the memory is required to be build again from the source that still is very expensive. One of the best solutions is to use persist (Storage level.MEMORY\_AND\_DISK\_ONLY ) that would spill the partitions of RDD to the Worker's local disk. This case only requires getting data from the Worker's local drive which is relatively fast.

<https://spark.apache.org/docs/latest/sql-performance-tuning.html>

<https://www.analyticsvidhya.com/blog/2020/10/how-can-you-optimize-your-spark-jobs-and-attain-efficiency-tips-and-tricks/>

<https://www.analyticsvidhya.com/blog/2021/05/performance-tuning-on-apache-spark/>

<https://www.protechtraining.com/blog/post/tuning-apache-spark-jobs-the-easy-way-web-ui-stage-detail-view-911>

https://github.com/LearningJournal/Spark-Programming-In-Python/blob/master/20-BucketJoinDemo/BucketJoinDemo.py

<https://medium.com/datakaresolutions/optimize-spark-sql-joins-c81b4e3ed7da>

<https://towardsdatascience.com/the-art-of-joining-in-spark-dcbd33d693c>

<https://github.com/vaquarkhan/Apache-Kafka-poc-and-notes/wiki/Apache-Spark-Join-guidelines-and-Performance-tuning>