Introduction to Apache Spark

Apache Spark is a well-known Big Data analysis framework. However, without a grasp of the mechanisms, I was frequently perplexed in the beginning. The transition from executing code on a single machine to employing clusters, together with the increase in the size of processed data from MB to GB (and even TB), motivated me to begin learning Spark. In this series, I will explain my Exploration Of Spark Performance Optimization, beginning with my first introduction post. This post will discuss some of the fundamental principles, APIs, and tools associated with Spark.

Without further ado, let us begin!

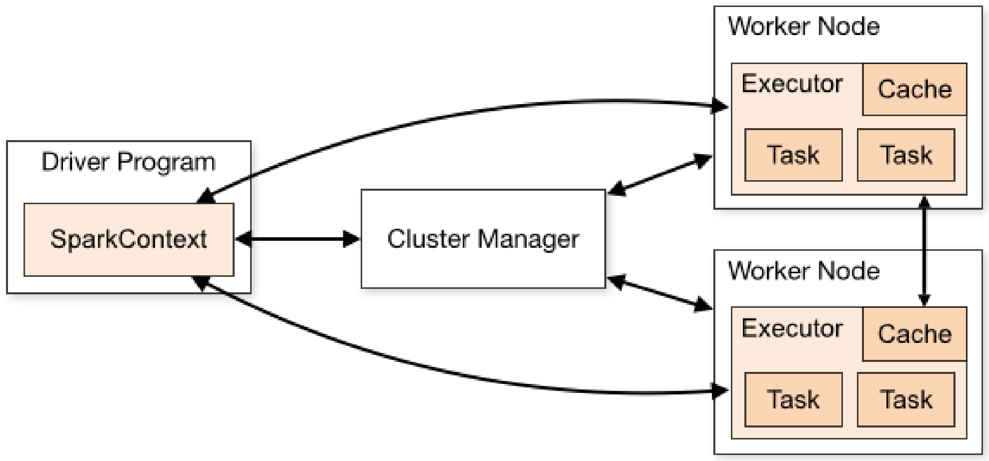
Spark is a software platform for distributed cluster computing. It provides simple APIs for computing enormous amounts of data, with end users scarcely needing to understand how Spark manages tasks and resources among machines.

Distributed Computing

Distributed computing needs the administration of resources and tasks across a cluster of devices. While resource management is concerned with getting the computers necessary to complete the current task, task management is concerned with coordinating the code and data throughout the cluster.

A Spark application is made up of a driver software that performs parallel computations on a cluster. To launch a Spark application, the host machine's driver programme must first create a SparkContext object. This SparkContext object will interface with a cluster manager to acquire resources for this application. The cluster manager can be Spark's standalone cluster manager, Mesos, YARN, or Kubernetes. The SparkContext object then distributes the application's code and duties to the worker nodes.

A worker node may have several executors for a single application, depending on the number of CPUs available on the worker node. Each executor maintains data in memory or on disc storage and executes tasks during an application's computation. This way, executors are segregated from one another, and tasks associated with the same application perform concurrently.



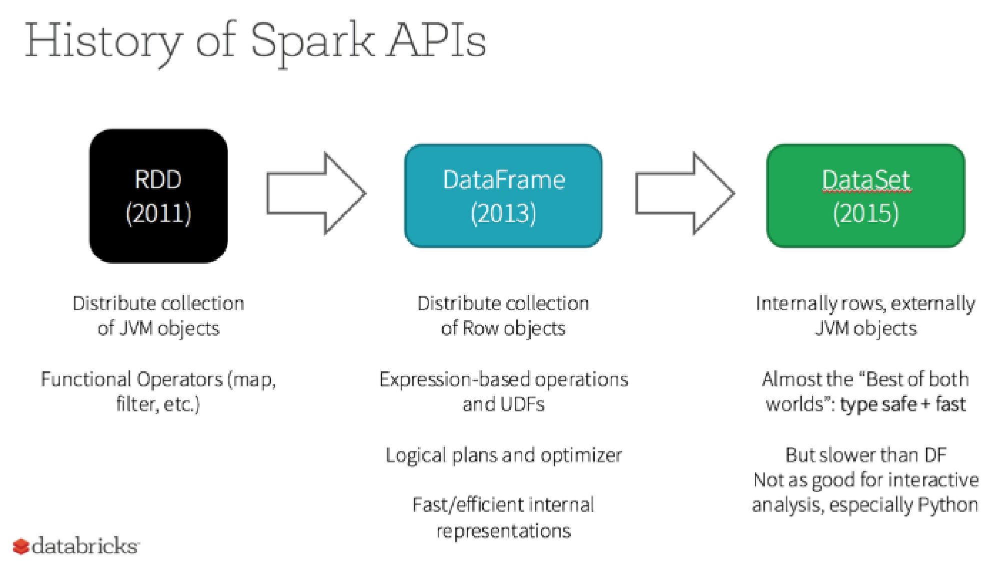
2.Resilient Distributed Dataset (RDD)

RDD, which stands for Resilient Distributed Dataset, is a fundamental abstraction in Spark. It permits the partitioning of big data sets into smaller data sets that fit on each machine, allowing for parallel computation on numerous machines. Additionally, RDDs recover automatically from node failures, ensuring storage resiliency.

HDFS (Hadoop Distributed File System) is another critical idea that I frequently encounter while working with Spark. While both RDD and HDFS are concerned with resilient distributed storage, they are intended to address distinct concerns. The robust perspective of RDD refers to the way in which computations fail automatically. While HDFS is primarily concerned with storage management, it is also built to deal with storage failure.

3. Spark APIs

Three APIs are available in Spark: RDD, DataFrames, and Datasets. While all three of these APIs enable distributed, robust data processing, they are each optimised for a certain application situation.


History of Spark APIs
RDD (2011)
DataFrame (2013)
DataSet (2015)
Distribute collection of JVM objects
Functional Operators (map, filter, etc.)
Distribute collection of Row objects
Expression-based operations and UDFs
Logical plans and optimizer
Fast/efficient internal representations
Internally rows, externally JVM objects
Almost the "Best of both worlds": type safe + fast
But slower than DF
Not as good for interactive analysis, especially Python
databricks

RDD is Spark's fundamental low-level API for manipulating unstructured or semi-structured data. Utilizing RDD is analogous to instructing Spark on how to complete a task, as opposed to simply instructing Spark on what task to perform. As a result, of the three APIs, RDD provides the most coding flexibility and data control. At the same time, without using Spark's intrinsic optimizations, a skilled RDD master imposes higher expectations on programmers' experience.

Spark includes a high-level DataFrames API in addition to the low-level RDD API. DataFrames place a premium on data structure. Thus, DataFrames are amenable to programmers with prior knowledge with relational databases. When working with DataFrames, the experience is quite similar to that of working with Pandas DataFrames or Excel Spreadsheets, but with Spark managing the cluster computing behind the scenes.

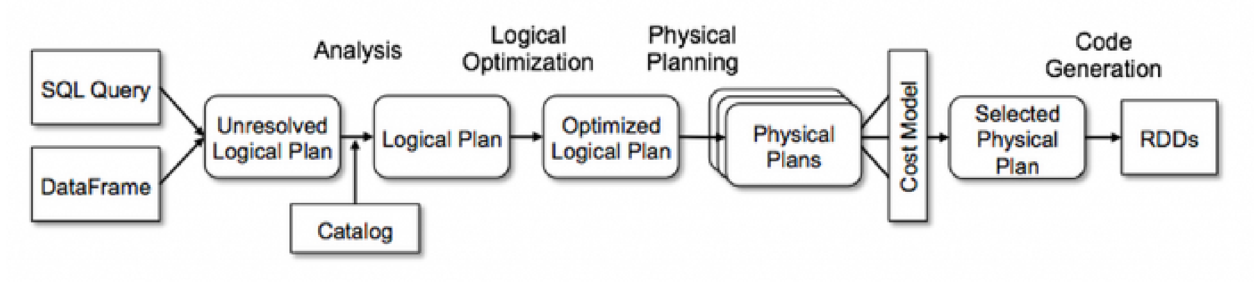
If we consider RDD and DataFrames APIs to be on opposite sides of a coin, with RDD on the side of flexible low-level control and DataFrames on the side of simple high-level code, then Datasets API is in the middle of the two. Datasets API imposes type safety on top of DataFrames API to avoid run-time errors.

Both the RDD and Datasets APIs require type safety and are currently only available in Java and Scala. The DataFrame API, on the other hand, allows dynamically typed languages like Python and R.

Jules Damji has an excellent blog devoted to RDDs, DataFrames, and Datasets. Don't forget to check it out if you're interested.

4. SQL Spark

Spark SQL is a Spark module that allows you to interact with structured data.
Spark SQL supports two types of structured APIs: Datasets and DataFrames. By leveraging the schema information contained in Datasets or DataFrames, Spark SQL code instructs Spark in a declarative manner, in contrast to the way Spark is instructed when utilising the low-level RDD API. Thus, Spark SQL code benefits from Spark's catalyst, which optimises performance. Thus, using Spark SQL in conjunction with structured APIs simplifies the process of writing performant programmes.



Utilizing Spark SQL in conjunction with the DataFrames API is analogous to running a SQL query on a relational database. Spark SQL supports all commonly used SQL functions, including filter, join, aggregation, and window. Spark SQL and the DataFrames API are language-agnostic, supporting Python, R, Scala, and Java.

While Spark SQL, Presto, and Hive all offer SQL-based querying of enormous amounts of data stored in distributed storage, they are utilised for quite different scenarios.

Spark SQL is the Spark core module, whereas Presto is part of the Hadoop environment. Spark SQL places a premium on calculation and is frequently used in large-scale ETA and pipeline applications. Presto, on the other hand, places a premium on querying and is hence more frequently employed for ad-hoc analysis. Spark SQL and Presto both do calculations in memory. When memory becomes scarce, Spark SQL allows for overflowing into the disc, whereas Presto will experience OOM difficulties. Spark SQL considers fault tolerance as well, but not Presto.

Hive is a Hadoop-based data warehouse that manages enormous amounts of structured data. Spark or MapReduce can be used to execute Hive queries. Hive includes its own SQL engine, HiveQL. As discussed previously, Spark SQL is the module for working with structured data in Spark. Similarly, Hadoop's Hive module is used to work with structured data.

Summary

I explained some fundamental aspects of Spark in this post. Although the combination of the high-level DataFrame API and Spark SQL simplifies the process of writing performant code, understanding how Spark works enables further speed improvements. In the following piece, I'll use YARN as an example and show how to use the YARN web UI to understand Spark resource and task management.