GEORG-AUGUST-UNIVERSITÄT
GÖTTINGEN

What’s new with Spark 3

M.Inf.1237.Mp: Seminar Newest Trends in High-Performance Data Analytics

30. Sept 2022

By

Abdul Rafay

Matriculation number

18413875

Supervisor

Prof. Dr Julian Kunkel and Patrick Michaelis

Abstract

As the world is emerging day by day, new smart devices and IoT devices have been seen growing drastically. The increase in devices and usage to digital platform creates a lot of data in many formats and with a huge velocity. To handle, analyze and benefit from such big data, several big data frameworks have been introduced like Hadoop, Apache Storm, Apache Flink and Apache Spark. Out of many big data frameworks available in the market, apache spark is the most famous and widely adopted framework due to its lightning fast processing of data by performing all the operations in memory. As the data is growing at a very high speed with many varieties in data, the big data frameworks also needs to be improved with time. Spark 3 is major release and successor of Spark 3 with a lot of new features and with a major focus on enhancing the performance for interactive, batch, streaming, and inaudible workloads. In this report, we discuss the major improvements and new features introduced in spark 3. Furthermore, we derive some performance based statistics with new features in spark 3 tested on sample dataset. We also present the performance statistics comparison between spark 3 and spark 2.

Table of contents

1 Introduction to Apache Spark: A unified Analytics Engine

When we consider size, we can't help but consider Google's search engine's prowess at swiftly indexing and searching all of the material available on the internet. Scale and Google are mutually exclusive terms. The scale at which Google planned to generate and search the indexed information on the internet was too great for traditional storage systems like relational database management systems (RDBMSs) or imperative programming techniques. The Google File System (GFS), MapReduce (MR), and Bigtable were developed as a consequence of the need for fresh ideas. Bigtable enabled scalable storage of structured data over GFS, whereas GFS supplied a fault-tolerant and distributed filesystem across multiple commodity hardware servers in a cluster farm. For the large-scale processing of data distributed through Google File System and Bigtable, MapReduce created a new parallel programming paradigm built on functional programming.

With a project they termed Spark, researchers at UC Berkeley who had previously worked on Hadoop MapReduce took on this task. They were aware that MR was ineffective (or unsolvable) for interactive or iterative computing tasks and was a difficult framework to master, so they embraced the concept of making Spark quicker, simpler, and easier from the start. This project began in 2009 at the RAD Lab, which eventually changed its name to the AMPLab (and now is known as the RISELab). Early Spark publications showed that, for some tasks, MapReduce from Hadoop was 10 to 20 times quicker. In recent times, Spark performs much faster than Hadoop's MapReduce, can be think of greater than 50 times for large workloads [2].

Apache spark is not a storage engine as compared to Hadoop Distributed File System (HDFS) and is not built to replace Hadoop in any manner. Spark can act as a storage engine to cache immediate results into memory. Mainly, Spark complements Hadoop by integrating to HDFS and process large data stored on Hadoop cluster in very fast way by computing all the data in memory instead of MapReduce which process data on hard disk and runs many Input/ Output operations. Spark comes with variety of libraries/ packages which can perform batch/ stream processing, machine learning, and graph processing which makes spark as a unified engine that can perform almost all the operations that a Data Engineer or Data Scientist needs to perform in daily tasks.

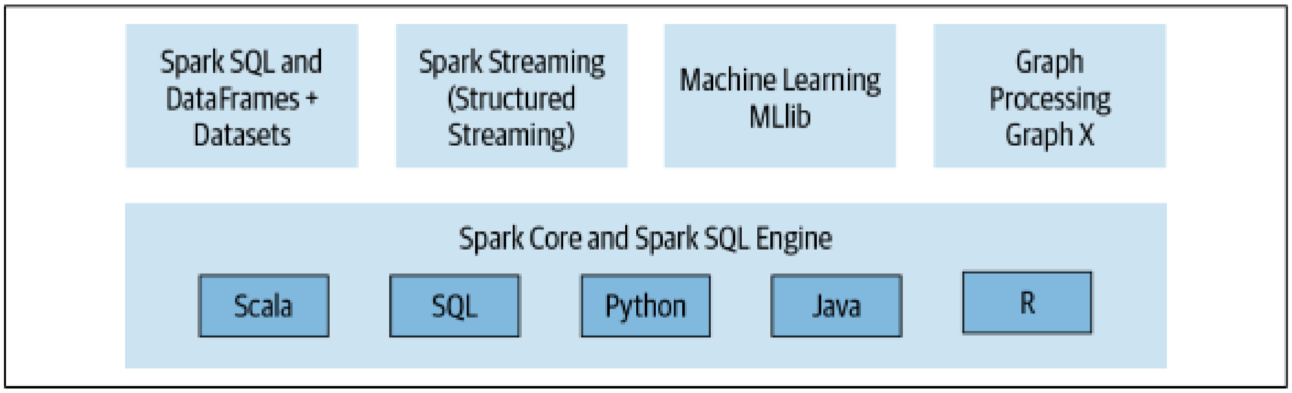
1.a What is Apache Spark

For on-premises servers, or in the cloud, large-scale distributed data processing, Apache Spark is a unified engine that can be deployed on both. As opposed to Hadoop MapReduce, Spark offers in-memory storage for intermediate operations that does not needs to be spilled to the hard disk. It includes machine learning libraries (MLlib), Spark SQL for running interactive queries on data, stream processing libraries (Structured Streaming) for dealing with real-time continuous unbounded streams of data, and graph processing libraries (GraphX). Spark jobs and libraries can be implemented in a variety of programming languages. Fig. 1 depicts the Spark components.

Apache Spark development motivations includes the following key characteristics:

- Extensibility
- Modularity
- Easy adoption
- Speed

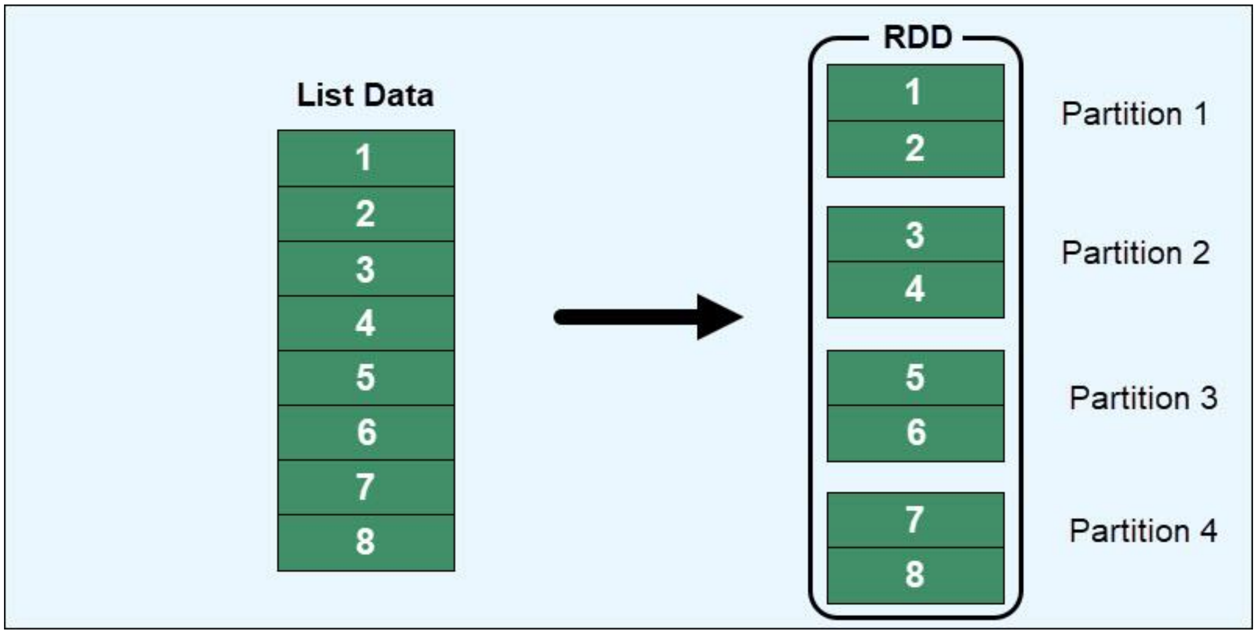
Figure 1. Components and APIs in Apache Spark [1]


Spark SQL and DataFrames + Datasets
Spark Streaming (Structured Streaming)
Machine Learning MLlib
Graph Processing Graph X
Spark Core and Spark SQL Engine
Scala
SQL
Python
Java
R

1.b Spark: What's underneath an RDD

Spark operated data in Resilient Distributed Datasets (RDD) The primary logical data units in Spark are RDDs. They are a dispersed group of things that are kept in the memory or on the disks of various machines in a cluster if and only if there is not enough memory to reside the data then data is going to spill on the disk. Multiple logical partitions can be created from a single RDD so that these partitions can be processed and stored on many cluster computers. RDDs have a read-only (immutable) nature, and once created they cannot be changed. Original RDDs cannot be modified, but they can be transformed or subjected to other coarse-grained procedures to produce new RDDs. Fig.2 illustrates the RDD in action.

Figure 2. How RDDs maintain data in spark [2]



Following are the features present in RDDs:

Resilience: When RDDs fail, they automatically restore lost data by tracking data lineage information. Another name for it is fault tolerance.

Lazy evaluation: Even though you specify data, it does not load into an RDD. When you call an operation, like count or collect, or when you save the result to a file system, transformations are really computed.

Immutability: Data saved in an RDD is read-only; it is not possible to alter the information inside. But by applying modifications to the current RDDs, you may produce new RDDs.

Distributed: An RDD's data is spread across several worker nodes. It is dispersed across many cluster nodes.

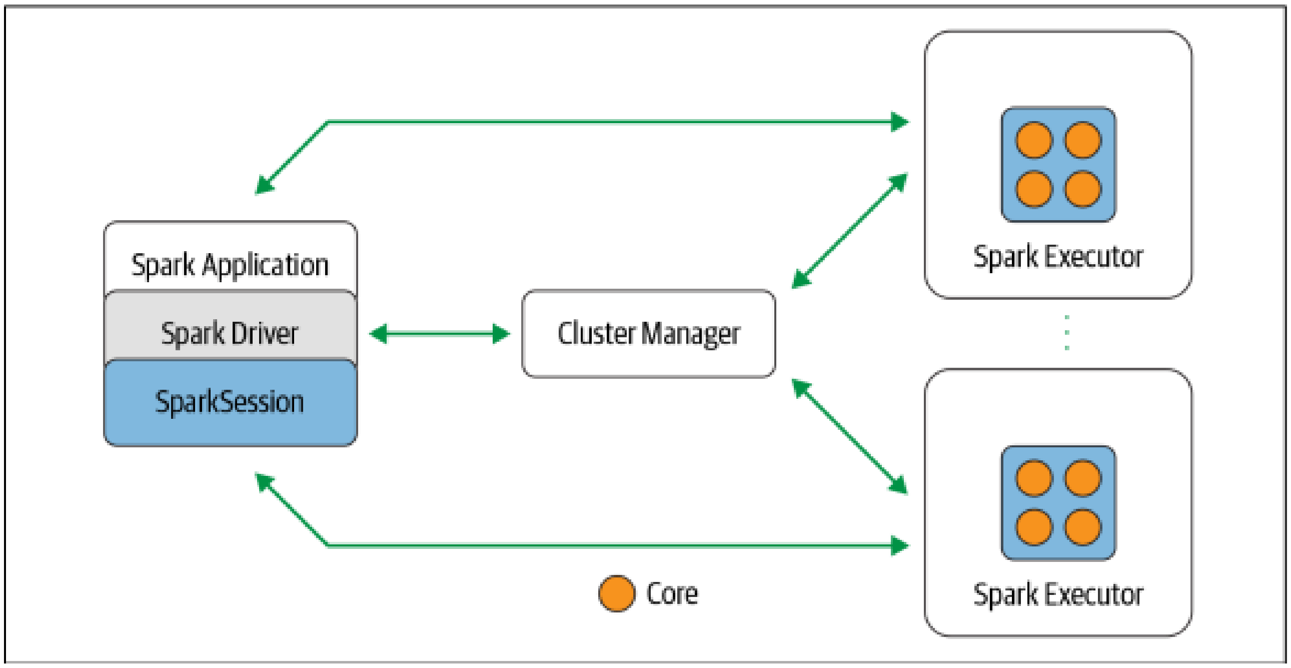
In-memory computing: An RDD saves any immediately created data in RAM rather than the disk to enable speedier access.

Partitioning: Any RDD that already exists can be partitioned to generate modifiable logical sections. This may be done by performing transformations on the current partitions.

2 Spark Deconstructed

Spark follows a Master-Slave architecture and runs everything in a cluster. A driver software that manages parallel activities on the Spark cluster makes up a Spark application at the highest level of the Spark architecture. Through a SparkSession, the driver connects to the cluster's distributed components, including the cluster manager and Spark executors which actually executes the jobs on RDDs assigned by a driver program as can be seen in Fig. 3.

Figure 3. Architecture of Spark [3]



Following are the components in spark architecture:

Driver: It serves as the brain of a Spark application and keeps track of all user data during the program's lifespan.

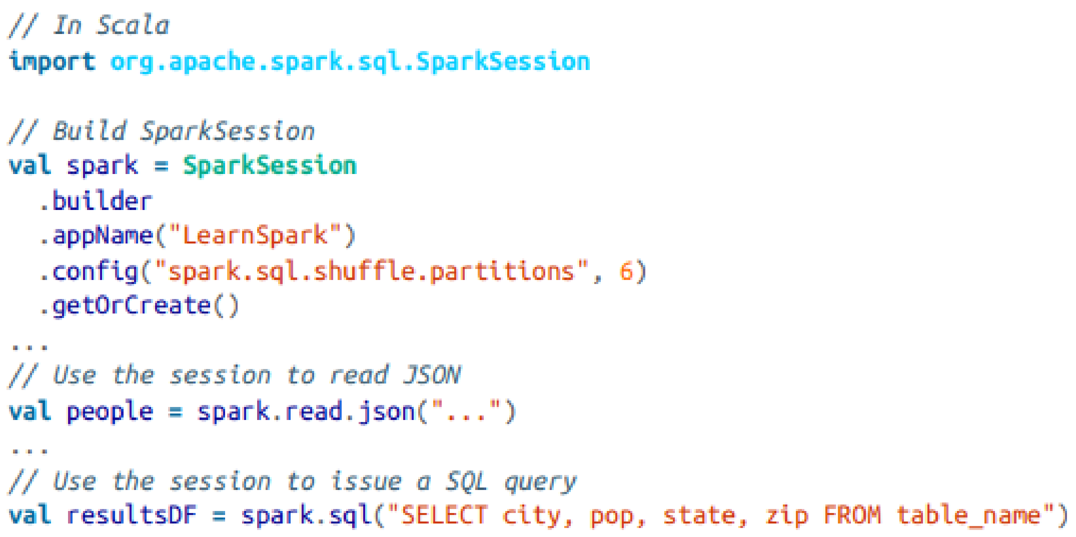
Executor: The executors are accountable for completing the tasks the driver gives them and reporting back to the driver the status of the computation on that executor.

SparkSession: This is the entry point of the spark job. In previous version of spark the conduit was separated into SparkContext, SparkConf and more.

Cluster manager: The cluster of nodes on which your Spark application runs must be managed and its resources allocated by the cluster manager. The built-in standalone cluster manager, Apache Hadoop YARN, Apache Mesos, and Kubernetes are the four cluster managers that Spark currently supports.

Spark 3 unifies the conduit in SparkSession unlike in the previous version of spark. Fig. 4 shows an example of the creation of spark job through SparkSession.

Figure 4. Example writing code in Spark 3 [4]



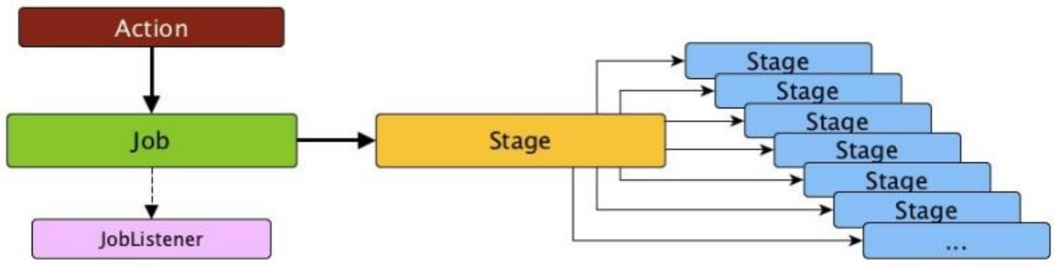
```
// In Scala
import org.apache.spark.sql.SparkSession

// Build SparkSession
val spark = SparkSession
  .builder
  .appName("LearnSpark")
  .config("spark.sql.shuffle.partitions", 6)
  .getOrCreate()
...
// Use the session to read JSON
val people = spark.read.json("...")
...
// Use the session to issue a SQL query
val resultsDF = spark.sql("SELECT city, pop, state, zip FROM table_name")
```

2.a Spark Jobs and Stages

Spark has two types of operations, one is transformation and another one is action. Transformation operation makes lineage graph whereas action triggers the job and perform all the transformations. A job is a series of steps that are initiated by an operation like count, show, read, or save function that are called actions, each parallelized action is referred to as a Job. The Executor program delivers the outcomes of each Job (parallelized/distributed activity on partitions) to the Driver program. Jobs are dependent on the workload and code, multiple jobs might be created.

A Stage is a group of Tasks that can be carried performed sequentially, or in parallel, without shuffling. As an illustration, utilizing ".read" to read a file from disk, followed by the execution of ".filter" may be done without shuffling, allowing it to fit in a single stage. Dataset's Partition count will also affect how many Tasks are in each Stage. Fig. 5 represents the sketch of a job and stage in spark. Furthermore Fig. 6 depicts the task in spark.

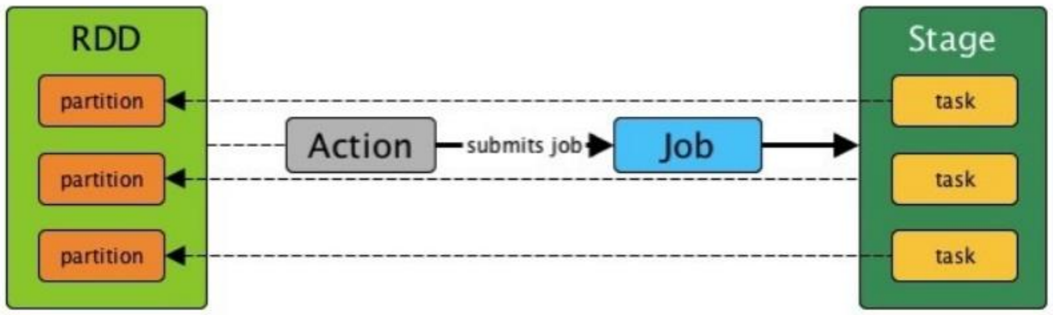
Figure 5. Sketch of Job and Stage in spark



As shown in the Fig. 6 an executor is given a task when they are given a piece of work. There are a few tasks at each step, one for each division. The identical process is carried out over many RDD partitions. Simply, each step in a stage is a task.

2.b Tasks

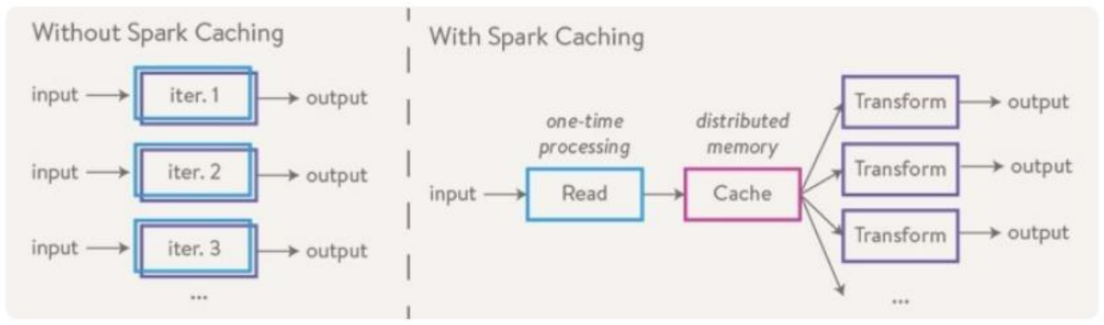
Figure 6. Sketch of Task in spark



One of the most helpful improvements in applications that repeatedly utilize the same datasets is caching. Caching will speed up subsequent reads by storing a DataFrame or DataSet or Table across all of your cluster's executors in temporary storage. This data will be residing in the memory of each executor which is having respective partitions of data. As can be seen in Fig. 7, caching can make reads faster.

2.c Caching and Shuffling

Figure 7. Sketch of caching in spark [5]

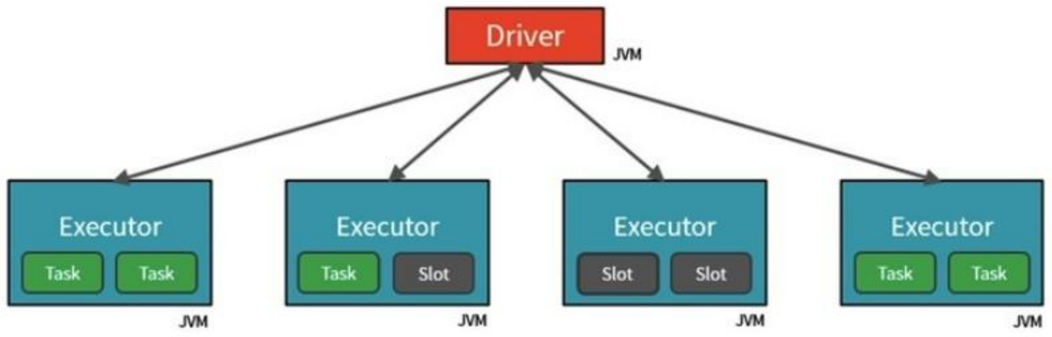


When data has to travel across executors for aggregation or comparision, this procedure is referred to as a "shuffle," where data is re-partitioned throughout a Cluster. join and any action with the last character ByKey will cause a Shuffle. Because so much data may be transferred across the network, it is an expensive procedure. Shuffle is considered a heavy operation in spark because there is a lot of network overhead. Moreover, shuffle is defined by a wide transformation which requires data from multiple partitions, on contrast there are also narrow transformations which does not cause shuffle and are operations like filter etc.

2.d Slots/ Threads in Spark

Spark has two levels of parallelization. One is assigning tasks to different executors. The slot is the other. Numerous positions are available to each executor. A Task may be assigned to each slot. This two level of parallelization makes spark to work much faster. As we have many cores on modern CPU architecture, thus park makes use of multiple cores by just assigning each task to each core and those are also known as slots. Fig. 8 depicts the Task and Slot assignment.

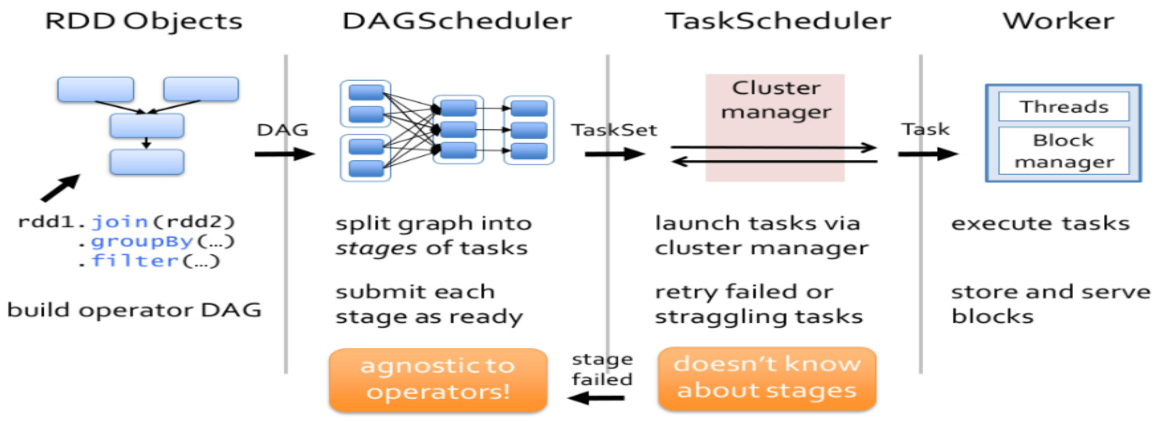
Figure 8. Sketch of slots and tasks in spark. [4]



2.e Directed Acyclic Graph

The RDDs are represented by the vertices of the Directed Acyclic Graph (DAG) in Apache Spark, and the operations that will be performed on the RDDs are represented by the edges. Stage-oriented scheduling is implemented by Apache Spark's DAGScheduler scheduling layer. A logical execution plan is changed into a physical execution plan (using stages). Following the execution of an action, SparkContext provides DAGScheduler with a logical plan, which the latter then converts into a collection of stages and submits as a set of tasks for execution. Jobs and stages are the essential ideas of DAGScheduler, which it tracks using internal registries and counters. Fig. 9 illustrates the DAG and work of DAGScheduler in spark

Figure 9. Sketch of DAG and DAGScheduler in spark. [6]



2.f Spark User Interface

The web interface of a running Spark application, also known as Application UI, webUI, or Spark UI, allows users to see and examine Spark job executions via a web browser. Each SparkContext creates a unique instance of the Web UI, which is accessible by default at http://[driver]:4040 (you may modify the port using the spark.ui.port parameter), and which scales up if this port is already in use (until an open port is found).

The spark user interface contains the following tabs which contains information regarding the runtime job statistics.

Jobs: Contains the information related to job and nested jobs.

Stages: Depicts the information and statistics regarding the stages in the job.

Storage: This tab shows the partitions data size and cached items.

Environment: It contains the configuration settings of the spark app.

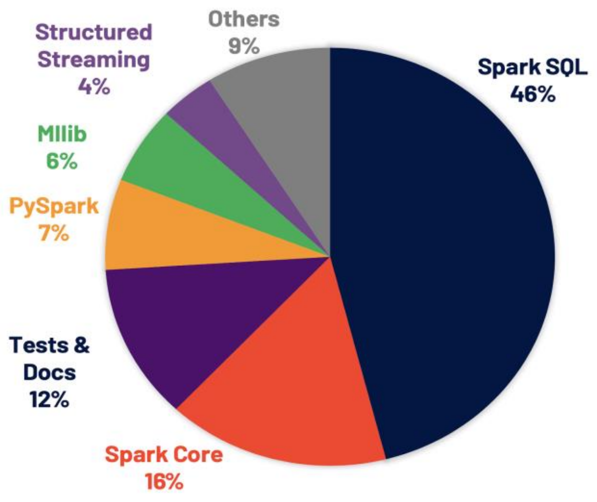
Executors: contains the information for all the worker nodes eg (CPU, RAM).

SQL: contains the query performance indicators and query plans.

3 Spark 3 Features

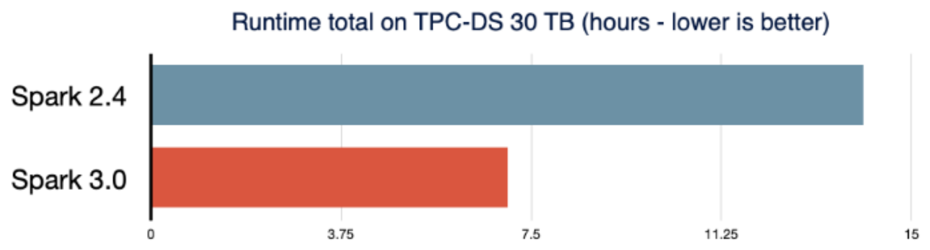
Spark is intended to become quicker, simpler, and smarter with each new version. Spark 3.0, a significant release, expands its reach by resolving more than 3000 JIRAs. These improvements help all higher-level libraries, with 46% of Spark SQL problems fixed [1]. Many new features and performance improvements were included in the Spark 3.x version. Fig. 10 depicts the distribution of Spark's ticket resolved for performance improvements and bugs resolution.

Figure 10. Spark 3 distribution of resolved tickets. [1]



As can be seen from Fig. 10, most of the improvements are don on Spark SQL, and that is because Spark's APIs are built on top of SQL only, means that all of the code on programming language are going to be translated to SQL command only. There are many new improvements in Spark 3, but in this paper we discuss the major new features introduced in Spark 3.0. Moreover Databricks, the creator of Spark claims that the new version of Spark is roughly 2 times better than the previous version. Fig. 11 represents the performance improvement in spark 3 tested on a TPC-DS 30 Terabyte of data.

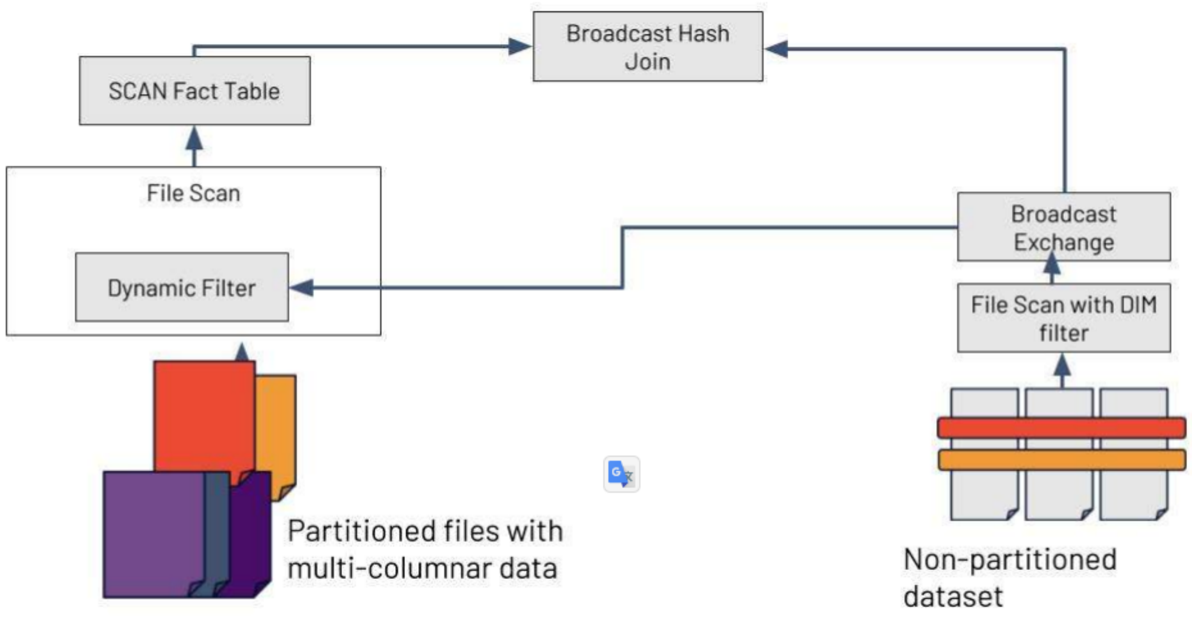
Figure 10. Spark 3 performance improvement as compared to previous version on test data of 30TB. [1]



3.a Dynamic Partition Pruning

Dynamic Partition Pruning (DPP) aims to restrict data reading to what is absolutely necessary. DPP has the ability to automatically improve queries' performance. Dynamic partition trimming is demonstrated in Fig. 11 along with the mentioned processes. The figure's right-hand dimension table has been searched and filtered. As part of the filter query, a hash table is created. Spark creates a broadcast variable using the outcome of this query and a hash table. Then, Spark's physical plan is modified to apply the dynamic filter to the fact table during runtime by broadcasting the filter to each executor. The inquiries can be significantly sped up using DPP. DPP is disabled by default in spark and can be enabled by setting the following *spark.sql.optimizer.dynamicPartitionPruning.enabled* property to true in the configuration file [3].

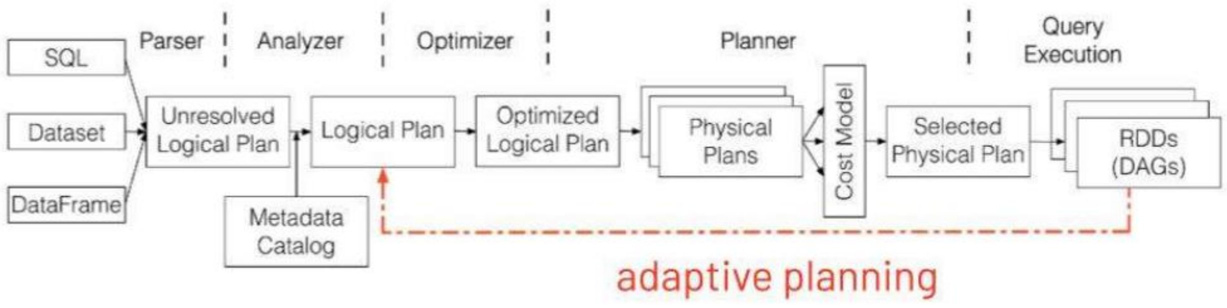
Figure 11. DPP in action. [1]



3.b Adaptive Query Execution (AQE)

Using runtime information gathered while a query is being executed, AQE enables Spark to re-optimize and modify query plans. When AQE is enabled, Spark will provide feedback about the quantity of the data in the shuffle files while the code is being performed, allowing it to dynamically alter join techniques, coalesce the number of shuffle partitions, or improve skew joins for the logical plan stage that follows. Fig. 12 illustrates that where AQE resides in the spark execution model. There is a major performance improvement achieved in spark 3 as depicted in the Fig. 16 and Fig. 17 when Spark is ran on TPC-DS 30 TB of data.

Figure 12. AQE planning in the spark execution model. [1]



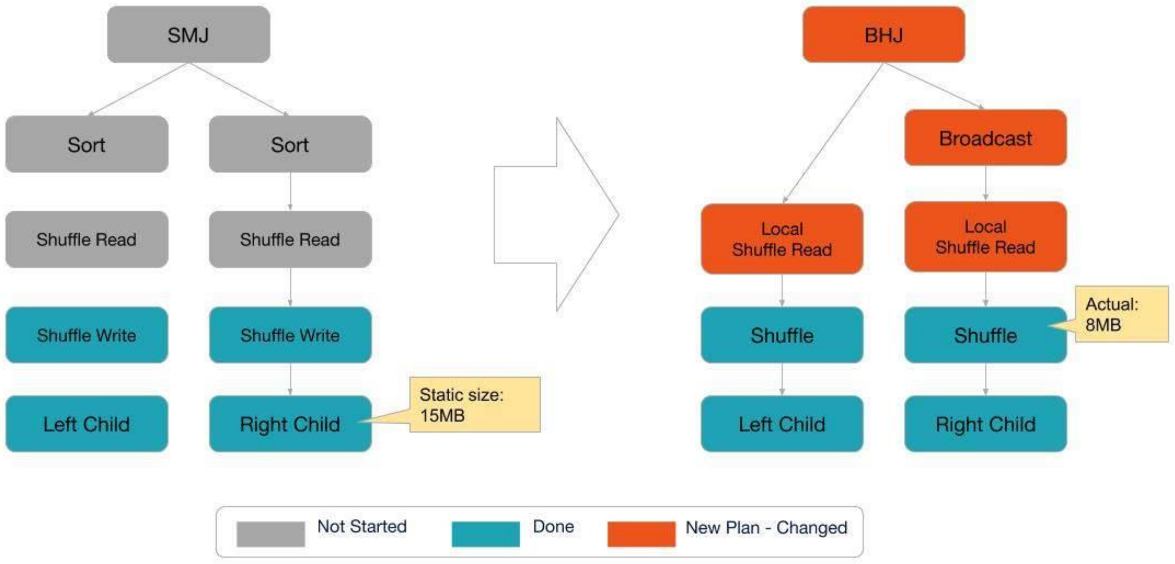
In a nutshell, AQE performs the following strategies:

- Switch Join Strategy
- Coalesce shuffle partitions
- Optimize skew joins

Switch Join Strategy

If the projected size of a join relation is less than the broadcast-size threshold, existing rule-based optimizations include preparing a broadcast hash join. If it may be used without, Broadcast Hash Join is one of the most effective join techniques provided by Spark. This is dependent on a data estimation depending on file size. However, a variety of factors, such as the presence of a particularly selective filter or the join relation being a sequence of sophisticated operators instead than just a scan, might cause this calculation to be incorrect. To address this issue, AQE now adjusts the join technique depending on the most precise join relation size during runtime. The right side of the connection is discovered to be significantly smaller than the estimate, as seen below. In reality, it can be broadcasted since it is tiny enough, allowing AQE to redesign the statically designed sort-merge join and change it to a broadcast hash join. In the Fig. 13, the tree on the left shows an example plan where Spark has selected a Sort-Merge Join. Before any data has actually been read, Spark can read that the file size is over 15MB, which is above the default threshold for using a broadcast join. In Spark 2.x, this is the strategy that will be used to carry out the query. The re-optimized plan created with AQE is displayed in the tree on the right. We can see that the query really reads in a file that is 8MB in size. Keep in mind that Spark is also attempting to capture just the relevant data from the file by pushing filters down to the data source. The end result is an 8MB file, which is less than the standard limit for a broadcast join. Spark may dynamically swap join techniques to employ the faster broadcast-hash join thanks to AQE [3].

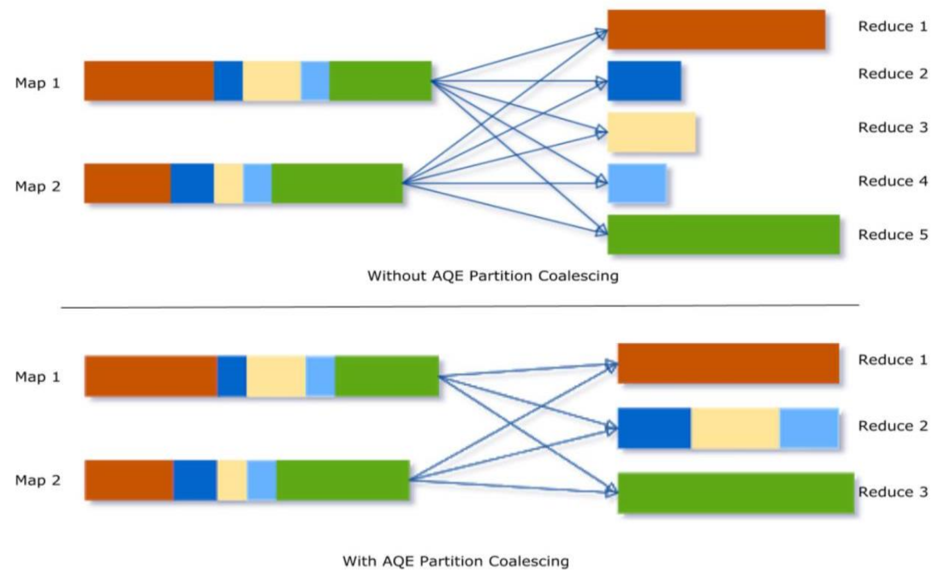
Figure 13. Switch join strategy illustration in exectuion. [1]



Coalesce shuffle partitions

The default number of shuffle partitions in spark is 200 and this can be tunable depending on the size of data and its partitions. Data size determines the optimal number of divisions, however as data sizes can vary greatly from stage to stage, this number can be challenging to optimize. Each partition's data size may be quite high if there aren't enough of them, which would slow down the query if processing these huge partitions necessitates spilling data to disk (for example, when sorting or aggregating is required). The query may be slowed down by the inefficient I/O pattern if there are too many partitions since the data size of each partition may be extremely little and there will be numerous small network data requests to read the shuffle blocks. Additionally, having many jobs increases the workload on the Spark task scheduler. If the input data table is tiny, as in Figure 14, there are only two divisions before grouping. After local grouping, the partially grouped data is shuffled into five partitions because the initial shuffle partition number is set to five. Spark will launch five jobs to complete the final aggregate if AQE is not present. It would be unnecessary to start a different job for each of the three little divisions present in this area. The final aggregation now only has to complete three jobs as opposed to five since AQE combines these three tiny partitions into one (as shown in the image below).

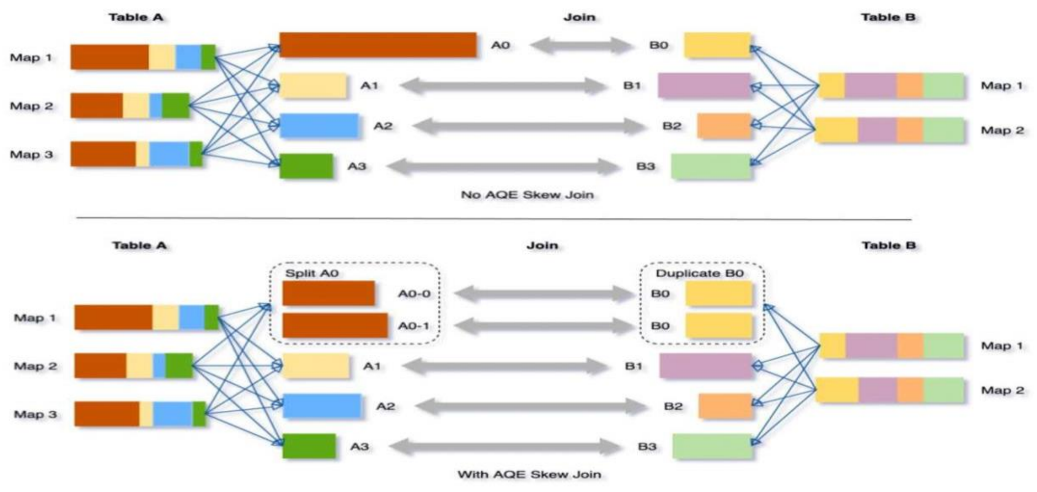
Figure 14. coalesce shuffle partitions illustration in exectuion. [1]



Optimize Skew Joins

When data are spread erratically among cluster partitions, data skew results. Extreme skew can dramatically reduce query speed, particularly when joining tables. Such skew is immediately found by AQE skew join optimization using shuffle file statistics. The skewed partitions are then divided into smaller sub partitions, each of which will be connected to the equivalent partition from the other side. Note that partition A0 in table A is noticeably larger than the other partitions, as illustrated in Fig. 15. In order to join each of the two sub partitions of partition A0 to its corresponding partition B0 of table B, the skew join optimizer will break partition A0 into two.

Figure 15. optimizing skey joins illustration in exectuion. [1]



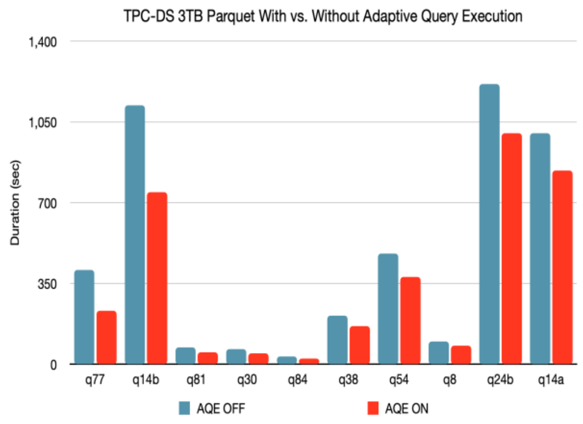


Figure 16. Performance statistics when AQE is ON and OFF in spark 3. [1]

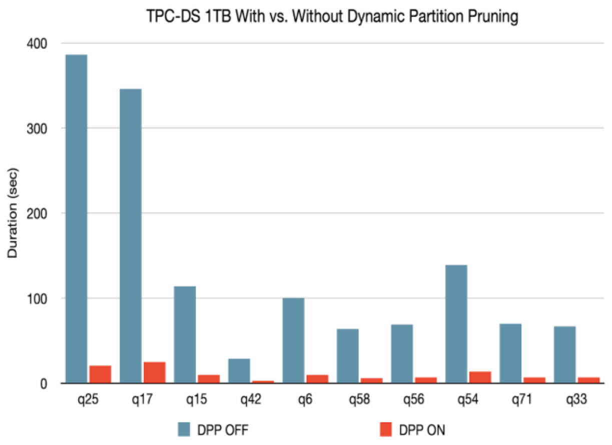
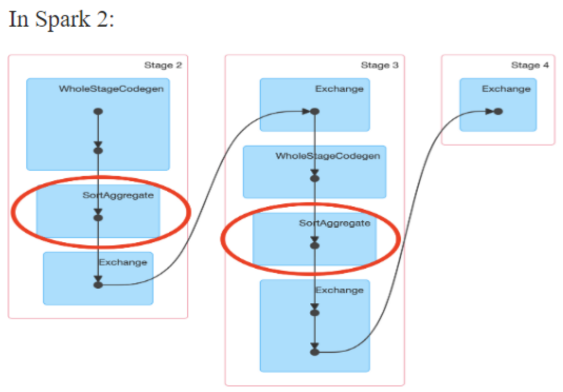
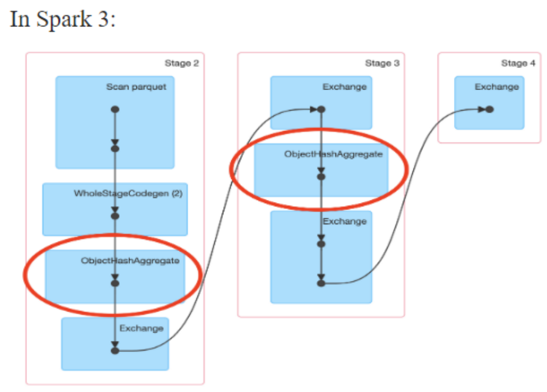


Figure 17. Performance statistics when DPP is ON and OFF in spark 3. [1]

Figure 20. Performance statistics comparison





As can be seen in Fig. 20, while aggregating, no use of serializing and deserializing of tuples. It reduces the pressure on Garbage collection and provides approximate 15% speedup. Also, SortAggregate is replaced by ObjectHashAggregate in Spark 3 query plan. It saves to perform a sort step and thus can save approximate 20% time in each stage.

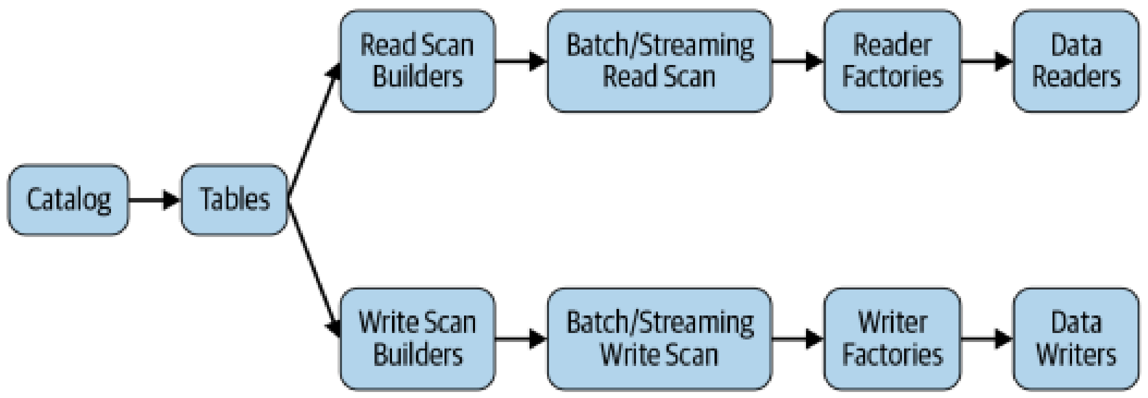
3.c SQL Join Hints

The compiler in smart has no guarantee to make optimal decision by choosing the join strategy so now in spark 3 join strategy can be provided in the job, thus spark can perform the optimal operation. Users can use join hints to persuade the optimizer to choose a better plan when the compiler is unable to make the optimal decision. Join algorithm selection is based on statistics and heuristics. The new join hints SHUFFLE MERGE, SHUFFLE HASH, and SHUFFLE REPLICATE NL enhance the current join hints.

3.d DataSouce V2 and Catalog Plugin API

The DataSourceV2 API in Spark 3.0 expands the Spark ecosystem and offers programmers three fundamental features. It permits connecting an external data source for catalog and table administration, to be more precise. With supporting file formats including ORC, Parquet, Apache Kafka, Cassandra, and Delta Lake, it also provides predicate pushdown to other data sources. Last but not least, it offers integrated APIs for batch and streaming processing of data sources for sinks and sources as shown in Fig. 18.

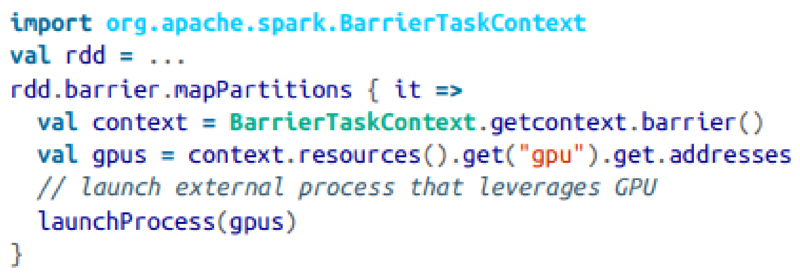
Figure 18. DataSource V2 lineage in action [2]



3.e Accelerator Aware Scheduler

The three main objectives of Project Hydrogen, a spark community project to integrate artificial intelligence (AI) and big data, are to establish barrier execution mode, schedule tasks that take use of accelerators, and optimize data sharing. Apache Spark 2.4 provided a simple barrier execution mode implementation. With the release of Spark 3.0, a simple scheduler has been added to target systems where Spark is deployed in standalone mode, YARN, or Kubernetes, allowing them to benefit from hardware accelerators like GPUs. Hardware resources must be provided in the spark configuration for Spark to utilize these GPUs for particular tasks in an orderly manner.
spark.worker.resource.gpu.discoveryScript=/path/to/script.sh needs to be set in order to utilize the GPU hardware resources [2].

Figure 19. Example code for leveraging GPUs on spark cluster [2]



```
import org.apache.spark.BarrierTaskContext
val rdd = ...
rdd.barrier.mapPartitions { it =>
  val context = BarrierTaskContext.getcontext.barrier()
  val gpus = context.resources().get("gpu").get.addresses
  // launch external process that leverages GPU
  launchProcess(gpus)
}
```

3.f ACID Transactions with Delta Lake

Spark is indeed not a storage engine rather it is a processing engine, but writing or saving the data in the data lakes with structured partitioning and bucketing can make spark to use as a storage engine as well. In contrast to conventional databases, which utilize complicated sharding, a data lake is a distributed storage solution that operates on commodity hardware and expands out horizontally with ease. The distributed storage system and the distributed computation system are separated by the data lake design, in contrast to database architecture. As a result, each system may expand as necessary to handle the burden. Additionally, the data is stored as open-format files that may be read and written by any processing engine using standard APIs. The Apache Hadoop project's Hadoop File System (HDFS), which was released in the late 2000s, helped make this concept more well-known. But the capabilities of database like ACID (Atomicity, Consistency, Isolation and Durability), schema enforcement, upsert and delete operations were missing in the data lake. The delta lake release with the spark 3.0 version now have all the capabilities of database. It is often known as LakeHouse concept.

Built by the original developers of Apache Spark, Delta Lake is an open source project managed by the Linux Foundation. Similar to the others, it is an open data storage framework that supports schema enforcement and evolution and offers transactional assurances. It also offers a number of other intriguing features, some of which are exclusive. Delta Lake encourages [2]:

- Utilizing Structured Streaming sources and sinks to stream reading from and writing to tables.
- Even in Java, Scala, and Python APIs, update, delete, and merge (for upserts) actions are supported.
- Time travel enables you to look for a specific table snapshot using its ID or timestamp.
- To fix mistakes, revert to earlier versions.

- Any SQL, batch, or streaming actions are performed amongst several concurrent writers in a serializable manner.

3.g Improved Pandas UDFs and Arrow Integration

In the past, one of the main problems with utilizing PySpark UDFs was that they performed less quickly than Scala UDFs. This was due to the PySpark UDFs' high requirement for data transfer between the JVM and Python. Pandas UDFs, sometimes referred to as vectorized User Defined Functions, were introduced as part of Apache Spark 2 version to address this issue. A Pandas UDF processes the data using Pandas while transferring the data uses Apache Arrow. The term pandas udf can be used to encapsulate a function or declare a Pandas UDF as its decorator. The data does not need to be serialized or pickled once it is in Apache Arrow format because it is already in a form that the Python process can use [3].

It may now be used with a Pandas Series or DataFrame instead of working on individual inputs row by row which in a nutshell is a vectorized execution. Pandas UDFs and Pandas Function APIs were separated into two API groups starting with Spark 3 version.

Pandas UDFs: Now it supports Python type hints.

Pandas Function APIs: It applies a local function of Python to a DataFrame in PySpark.

4 Discussion

Spark 3 is a major release in spark history and comes with a lot of new features that makes spark 3 fast, modular and extendible. Adaptive Query Execution (AQE) and Dynamic Partition Pruning (DPP) along with the Delta lakes support are the main highlight features of spark 3 which provides a lot more power to spark in terms of speed, efficiency and power. Although many minor versions are also release with time, but that only resolves a minor bugs available in spark and not release a major performance improvement. With the continuous development and support for spark 3 makes it much stronger to remain widely adopted big data framework in the market and also among the developers community.

5 References

[1] Databricks, Introducing Apache Spark 3.0. [Online]. Available: https://www.databricks.com/blog/2020/06/18/introducing-apache-spark-3-0-now-available-in-databricks-runtime.

[2] Jules S. Damji, Brooke Wenig, Tathagata Das & Denny Lee, Learning Spark, O.REILLY 2nd Edition.

[3] Databricks, Deep Dive into the New Features of Apache Spark 3.0 Download Slides [Online]. Available: https://www.databricks.com/session_na20/deep-dive-into-the-new-features-of-apache-spark-3-0

[4] Deepak Rajak, Just Enough Spark! Core Concepts Revisited [Online]. Available: https://www.linkedin.com/pulse/just-enough-spark-core-concepts-revisited-deepak-rajak/

[5] Vishal Loharkar, Just Enough Spark! Core Concepts Revisited [Online]. Available: https://www.linkedin.com/pulse/caching-spark-vishal-loharkar/?trk=read_related_article-card_title

[6] Yuval Itz, What happens when an executor is lost? [Online]. Available: https://stackoverflow.com/questions/37377512/what-happens-when-an-executor-is-lost