HOW TO PREPARE TO GET APACHE SPARK 3.0 CERTIFICATION

2021/2022

by Rodrigo, Thomaz and Nelio

CONTENTS

CERTIFICATION PREP FOR APACHE SPARK EXAM **6**
- EXPECTED 6
- NOT EXPECTED 6
- TOPICS ASSESSED ON THE EXAM 6
- GENERAL PREPARATION STRATEGIES 6
- QUESTIONS OVERVIEW 7
- STUDY RESOURCES 7
- SPARK ARCHITECTURE BASICS 8
- SPARK ARCHITECTURE APPLICATION 8
- SPARK DATAFRAME API BASICS 8
- THE MINIMALLY-QUALIFIED CANDIDATE 9
- SELF-ASSESSMENT ON SPARK ARCHITECTURE 9

SPARK ARCHITECTURE **10**
- SPARK'S BASIC ARCHITECTURE 10
- GLOSSARY 11
- SPARK EXECUTION MODES 12
  - SPARK LOCAL MODE 12
  - CLUSTER MANAGER 13
    - STANDALONE CLUSTER 14
    - YARN CLUSTER 15
    - MESOS CLUSTER 15
- SPARK APPLICATIONS 16
- POPULAR SPARK USE CASES 16
- WHAT IS A JVM? 16
- SPARK COMPONENTS 18
    - CLIENT PROCESS 20
      - SPARK-SUBMIT OPTIONS 20
    - SPARK SESSION 20
    - DRIVER 21
    - EXECUTORS 21
    - JOB 21
    - STAGE 22
    - TASK 23
    - SLOTS 24
    - SHUFFLE 24
    - PARTITION 25
    - DAG 25
    - DAGScheduler 25
    - CATALOG 26
- SPARK CONFIG 27
- SPARK UI 27
  - SPARK JOBS TAB 28
    - SCHEDULING MODE 28

NUMBER OF SPARK JOBS 29
NUMBER OF STAGES 29
DESCRIPTION 29
STAGES TAB 29
STAGE DETAIL 30
TASKS TAB 31
STORAGE TAB 31
ENVIRONMENT TAB 31
EXECUTORS TAB 32
SQL TAB 32
TRANSFORMATION, ACTIONS AND EXECUTION 33
LAZY EVALUATION 33
TRANSFORMATIONS 34
ACTIONS 36
PIPELINING 37
CATALYST OPTIMIZER 37
SQL QUERY AND DATAFRAME 38
UNRESOLVED LOGICAL PLAN 38
ANALYSIS 38
LOGICAL OPTIMIZATION 39
PHYSICAL PLANNING 39
COST MODEL 39
CODE GENERATION 40
VIEWING THE METADATA 40
CACHING 40
UNCACHE TABLE 41
RDD PERSISTENCE 41
PERSIST OPTIONS 41
WHICH STORAGE LEVEL TO CHOOSE? 42
DYNAMIC PARTITION PRUNING 43
ADAPTIVE QUERY EXECUTION 43
DYNAMICALLY COALESCING SHUFFLE PARTITIONS 44
DYNAMICALLY SWITCHING JOIN STRATEGIES 46
DYNAMICALLY OPTIMIZING SKEW JOINS 46
BROADCAST VARIABLES 46
REPARTITION 47
COALESCE 47
REPARTITION VS COALESCE 48
ACCUMULATORS 48

PERFORMANCE TUNING CONFIG OPTIONS 49
Performance Tuning considerations 49
IMPROVE SPARK PERFORMANCE 49

TUNING IN SPARK 52
NEED OF TUNING 52
AREAS OF TUNING 53
DATA SERIALIZATION 53

MEMORY TUNING 53
MEMORY MANAGEMENT IN SPARK 53
SLOW READS AND WRITES 54
DETERMINING MEMORY USAGE 55
MEASURING IMPACT OF GC 55
COMPONENTS OF MEMORY IN JVM 55
GOAL OF GC 56
STEPS TO AVOID FULL GC 56
DATA LOCALITY 56

DATAFRAME 56
READ/WRITE 56
GENERIC FILE OPTIONS 56
READ 57
WRITE 57
CSV 57
JSON 58
TEXT 58
PARQUET 58
ORC 58
DATABASE 59
FUNCTIONS 59
CREATEDATAFRAME 59
PRINTSCHEMA 60
CREATEORREPLACETEMPVIEW 60
SCHEMA 60
DROP 60
DROPDUPLICATES 61
DROPNA 61
FILLNA 62
DISTINCT 63
SPLIT 63
ALIAS 64
EXPLODE 64
WHERE / FILTER 64
COLLECT 65
TOLOCALITERATOR 65
WITHCOLUMN 65
WITHCOLUMNRENAMED 66
ARRAY_CONTAINS 66
SORT / ORDERBY 67
SORT_ARRAY 68
ASC_NULLS_FIRST 68
DESC_NULLS_FIRST 68
ASC_NULLS_LAST 69
DESC_NULLS_LAST 69
BETWEEN 69

SIZE 70
SAMPLE 70
ISIN 70
CONTAINS 71
MONOTONICALLY_INCREASING_ID() 71
DATEDIFF 71
ARRAY FUNCTIONS 71
WINDOW FUNCTIONS 74
AGGREGATE FUNCTIONS 76
AGG 76
COUNT 76
PIVOT 77
JOINS 77
JOIN 77
INNER JOIN 78
REFERENCE FOR JOINS 78
JOIN STRATEGIES 82
BROADCAST JOINS 83
BROADCAST HASH JOIN 84
SHUFFLE HASH JOIN 85
SHUFFLE SORT-MERGE JOIN 85
USER-DEFINED FUNCTIONS 86
SPARK SQL UDFS 87
EVALUATION ORDER AND NULL CHECKING IN SPARK SQL 87
UDF IN PYTHON, JAVA OR SCALA? 88

CERTIFICATION PREP FOR APACHE SPARK EXAM

EXPECTED

- Understanding of the basics of the Apache Spark architecture
- Ability to perform basic data manipulations using the Apache Spark DataFrame API
- Ability to read and write non-streaming data using Apache Spark
- Ability to apply basic scaling and debugging mechanisms for Apache Spark clusters

NOT EXPECTED

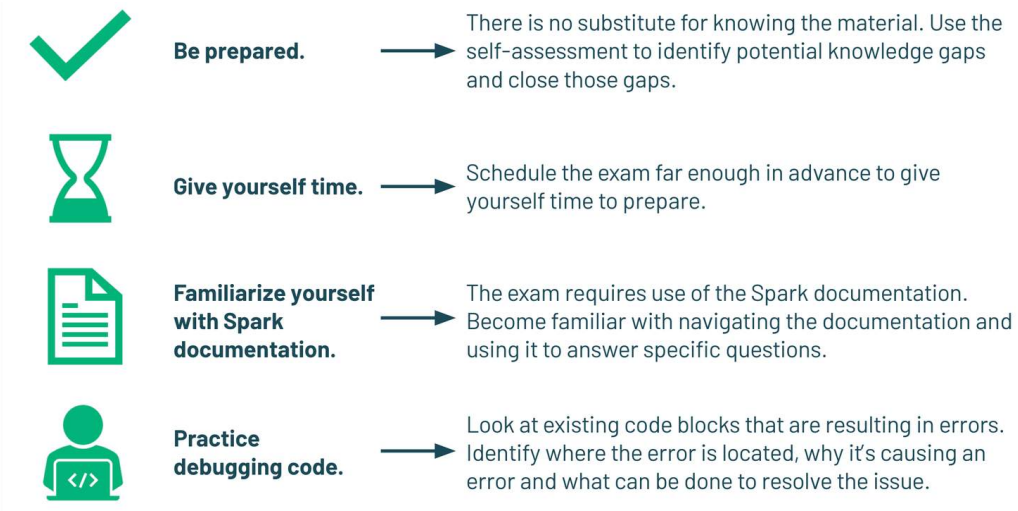
- Ability to tune Apache Spark jobs
- Memorization of the Apache Spark APIs
- Ability to create data visualizations
- Ability to build, evaluate, deploy, and manage machine learning models
- Understanding of data engineering and machine learning pipelines
- Ability to set up real-time data streams

TOPICS ASSESSED ON THE EXAM

1. Basics of the Apache Spark Architecture: The architecture of Apache Spark is detailed by its nature as a cluster-computing framework. It describes how data is partitioned, processed, etc.

2. Basics of the Apache Spark DataFrame API: The Spark DataFrame is the fundamental user-facing data structure of Apache Spark. Its API is used to manipulate data using common data manipulation terminology.

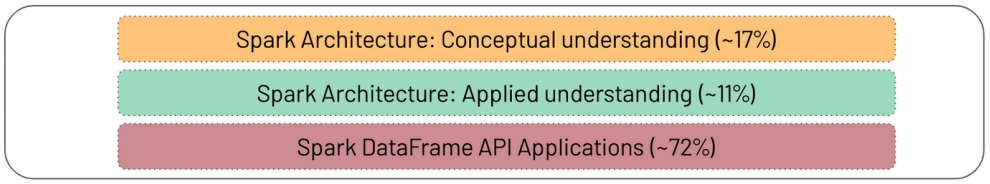
GENERAL PREPARATION STRATEGIES



QUESTIONS OVERVIEW

There are **60 total** questions on this exam. All of the questions are multiple-choice questions with five options - one correct answer and four distractors.

Exam questions are distributed into three categories:



STUDY RESOURCES

Exam:

- [ ] Databricks Certified Associate Developer for Apache Spark 3.0 - Databricks

Databricks Academy:

- [ ] Quick Reference: Spark Architecture
- [ ] Apache Spark Programming with Databricks

Books:

- [ ] Spark - The Definitive Guide - Sections I, II, and IV
- [ ] Learning Spark: Lightning-Fast Data Analytics: Damji, Jules S., Wenig, Brooke, Das, Tathagata, Lee, Denny: 9781492050049

Simulates:

- [ ] Apache Spark 3 - Databricks Certification Practice (PySpark)
- [ ] Databricks Certified Apache Spark 3.0 TESTS (Scala & Python)
- [ ] Databricks Spark Developer 3.0 Exam Questions 2021

Articles:

- [ ] https://docs.databricks.com/spark/latest/spark-sql/aqe.html
- [ ] https://spark.apache.org/docs/latest/sql-performance-tuning.html
- [ ] https://databricks.com/blog/2020/05/29/adaptive-query-execution-speeding-up-spark-sql-at-runtime.html
- [ ] https://calonsca.medium.com/guide-for-databricks-certified-associate-developer-for-apache-spark-3-0-1638bdb89883
- [ ] https://www.educba.com/data-science/data-science-tutorials/spark-tutorial/

Youtube Channels:

- [ ] Databricks: https://www.youtube.com/channel/UC3q8O3Bh2Le8Rj1-Q-_UUbA
- [ ] Learning Journal: https://www.youtube.com/channel/UC8OU1Tc1kxiI37uXBAbTX7A
- [ ] TechWithViresh: https://www.youtube.com/channel/UCZqHmLZxX0KC6PiJHETflOg
- [ ] Advancing Analytics: https://www.youtube.com/channel/UCmRI-X6XoeH2dQE4BShRU9Q

SPARK ARCHITECTURE BASICS

As for the basics of the Spark architecture, the following concepts are assessed by this exam:

- Cluster architecture: nodes, drivers, workers, executors, slots, etc.
- Spark execution hierarchy: applications, jobs, stages, tasks, etc.
- Shuffling
- Partitioning
- Lazy evaluation
- Transformations vs. actions
- Narrow vs. wide transformations

SPARK ARCHITECTURE APPLICATION

In addition, candidates are asked to apply their knowledge of the following to make optimal decisions when working with Spark. It should be able to interpret how these topics affect a Spark session and how they can use them to improve performance.

- Execution deployment modes
- Stability
- Garbage collection
- Out-of-memory errors
- Storage levels
- Repartitioning
- Coalescing
- Broadcasting
- DataFrames

SPARK DATAFRAME API BASICS

As for the basics of the Spark DataFrame API, candidates will be assessed in their ability to apply the DataFrame API to complete the following tasks:

- Subsetting DataFrames (select, filter, etc.)
- Column manipulation (casting, creating columns, manipulating existing columns, complex column types)
- String manipulation (Splitting strings, regular expressions)
- Performance-based operations (repartitioning, shuffle partitions, caching)
- Combining DataFrames (joins, broadcasting, unions, etc.)
- Reading/writing DataFrames (schemas, overwriting)
- Working with dates (extraction, formatting, etc.)
- Aggregations
- Miscellaneous (sorting, missing values, typed UDFs, value extraction, sampling)

THE MINIMALLY-QUALIFIED CANDIDATE

Some of the tasks the minimally-qualified candidate should be able to perform include:

- Selecting, renaming and manipulating columns
- Filtering, dropping, sorting and aggregating rows
- Joining, reading, writing and partitioning DataFrames
- Working with UDFs and Spark SQL functions

It is expected that data professionals that have been using Spark and its DataFrame API for six months or more should be able to pass the exam.

SELF-ASSESSMENT ON SPARK ARCHITECTURE

- ☐ Describe the difference between a Spark driver and Spark executor.
- ☐ Form a hierarchy of Spark jobs, tasks, stages, and applications.
- ☐ Describe what causes data to shuffle.
- ☐ Describe the difference between local and cluster execution modes.
- ☐ Determine what happens to the Spark application if the driver shuts down.
- ☐ Describe garbage collection strategies in Spark.

SELF-ASSESSMENT ON SPARK DATAFRAME API

- ☐ Select a subset of columns from a DataFrame.
- ☐ Filter a subset of rows from a DataFrame based on two logical filtering criteria.
- ☐ Cast a column from a numeric type to a string type.
- ☐ Create a new DataFrame column by mathematically combining two existing columns.
- ☐ Split a string DataFrame column into two columns based on a regular expression.
- ☐ Cache a DataFrame to a specific storage level.
- ☐ Aggregate data to find the mean of a column by group.
- ☐ Write a DataFrame to disk.
- ☐ Extract the month from a DataFrame column of date type.
- ☐ Create a UDF to use in a Spark SQL statement.

SPARK ARCHITECTURE

SPARK'S BASIC ARCHITECTURE

Apache Spark is a sophisticated distributed computation framework for executing code in parallel across many different machines.

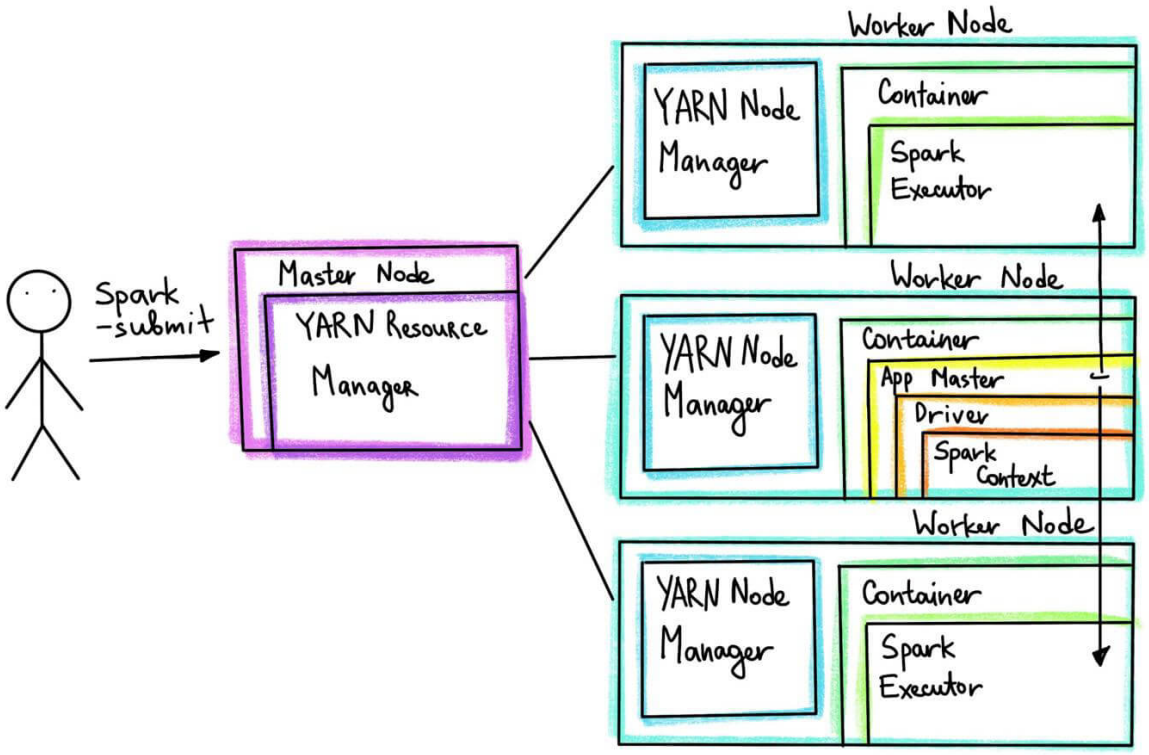
Spark uses clusters of machines to process big data by breaking a large task into smaller ones and distributing the work among several machines.

A **cluster**, or group of machines, pools the resources of many machines together allowing us to use all the cumulative resources as if they were one. Now a group of machines sitting somewhere alone is not powerful, you need a framework to coordinate work across them.

The cluster of machines that Spark will leverage to execute tasks will be managed by a **cluster manager** like:

- Spark's Standalone cluster manager
- YARN - Yet Another Resource Negotiator
- Mesos

We then submit Spark Applications to these cluster managers which will grant resources to our application so that we can complete our work.



GLOSSARY

The following table summarizes terms you'll see used to refer to cluster concepts:

Term	Meaning
Application	User program built on Spark. Consists of a *driver program* and *executors* on the cluster.
Application jar	A jar containing the user's Spark application. In some cases users will want to create an "uber jar" containing their application along with its dependencies. The user's jar should never include Hadoop or Spark libraries, however, these will be added at runtime.
Driver program	The process running the main() function of the application and creating the SparkContext
Cluster manager	An external service for acquiring resources on the cluster (e.g. standalone manager, Mesos, YARN)
Deploy mode	Distinguishes where the driver process runs. In "cluster" mode, the framework launches the driver inside of the cluster. In "client" mode, the submitter launches the driver outside of the cluster.
Worker node	Any node that can run application code in the cluster
Executor	A process launched for an application on a worker node, that runs tasks and keeps data in memory or disk storage across them. Each application has its own executors.
Task	A unit of work that will be sent to one executor
Job	A parallel computation consisting of multiple tasks that gets spawned in response to a Spark action (e.g. save, collect); you'll see this term used in the driver's logs.
Stage	Each job gets divided into smaller sets of tasks called *stages* that depend on each other (similar to the map and reduce stages in MapReduce); you'll see this term used in the driver's logs.

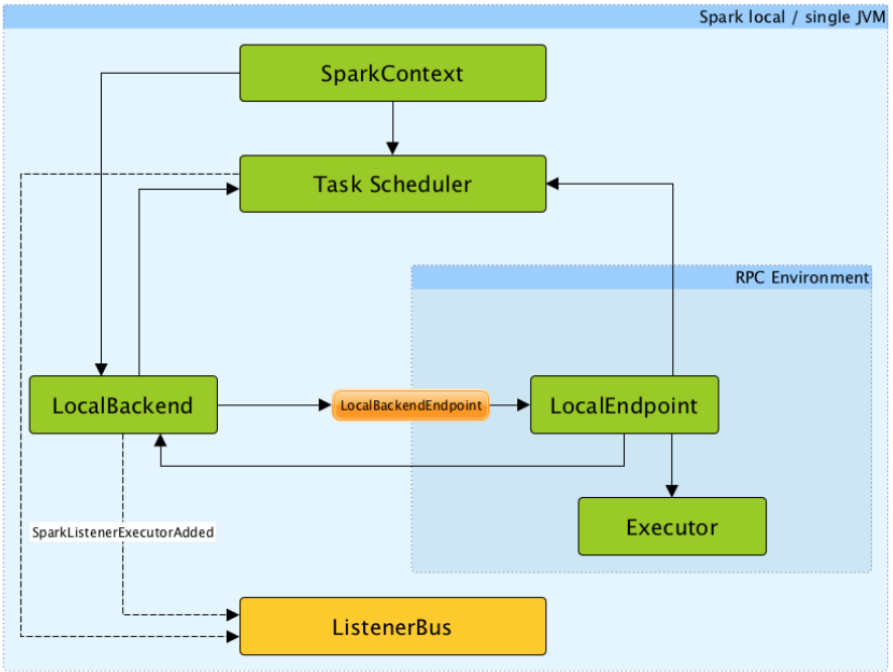
SPARK EXECUTION MODES

Spark can run in **local mode** and inside **Spark standalone, YARN, and Mesos clusters**. Although Spark runs on all of them, one might be more applicable for your environment and use cases. In this section, you'll find the pros and cons of each cluster type.

Mode	Spark driver	Spark executor	Cluster manager
Local	Runs on a single JVM, like a laptop or single node	Runs on the same JVM as the driver	Runs on the same host
Standalone	Can run on any node in the cluster	Each node in the cluster will launch its own executor JVM	Can be allocated arbitrarily to any host in the cluster
YARN (client)	Runs on a client, not part of the cluster	YARN's NodeManager's container	YARN's Resource Manager works with YARN's Application Master to allocate the containers on NodeManagers for executors
YARN (cluster)	Runs with the YARN Application Master	Same as YARN client mode	Same as YARN client mode
Kubernetes	Runs in a Kubernetes pod	Each worker runs within its own pod	Kubernetes Master

SPARK LOCAL MODE

- In this non-distributed single-JVM deployment mode.
- Spark spawns all the execution components (driver, executor, LocalSchedulerBackend and master) **in the same single JVM**
- The default parallelism is the number of threads as specified in the master URL.



CLUSTER MANAGER

Keep track of resources available. Spark runs in local mode or inside a cluster.

Spark applications are run as independent sets of processes, coordinated by a SparkContext in a driver program.

The context connects to the cluster manager which allocates resources

Each worker in the cluster is managed by an executor.

The executor manages computation as well as storage and caching on each machine

The application code is sent from the driver to the executors, and the executors specify the context and the various tasks to be run.

The driver program must listen for and accept incoming connections from its executors throughout its lifetime.

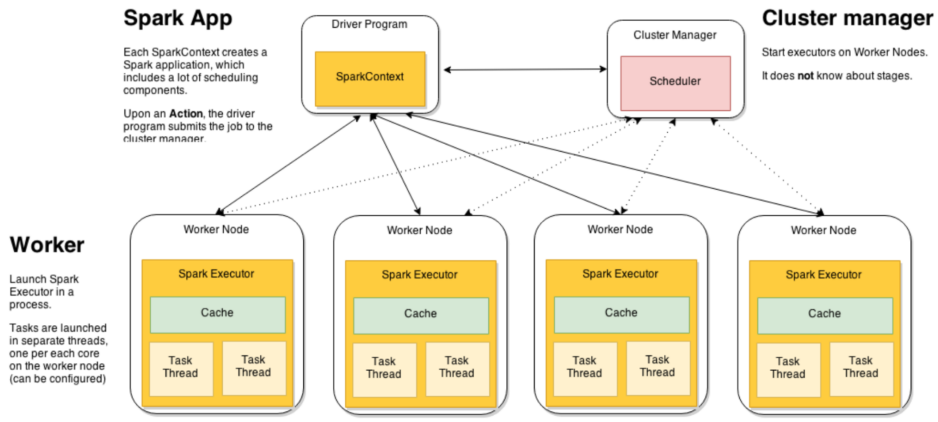
Spark has below cluster managers:

- **Standalone**
    - Most basic and default
    - Can run only spark applications

- **YARN**
    - Hadoop's resource manager
    - Can run spark and other java applications

- **Mesos**
    - Scalable and fault tolerant.
    - Can run spark as well several python,java and other applications

- **Kubernetes**
    - an open-source system for automating deployment, scaling, and management of containerized applications



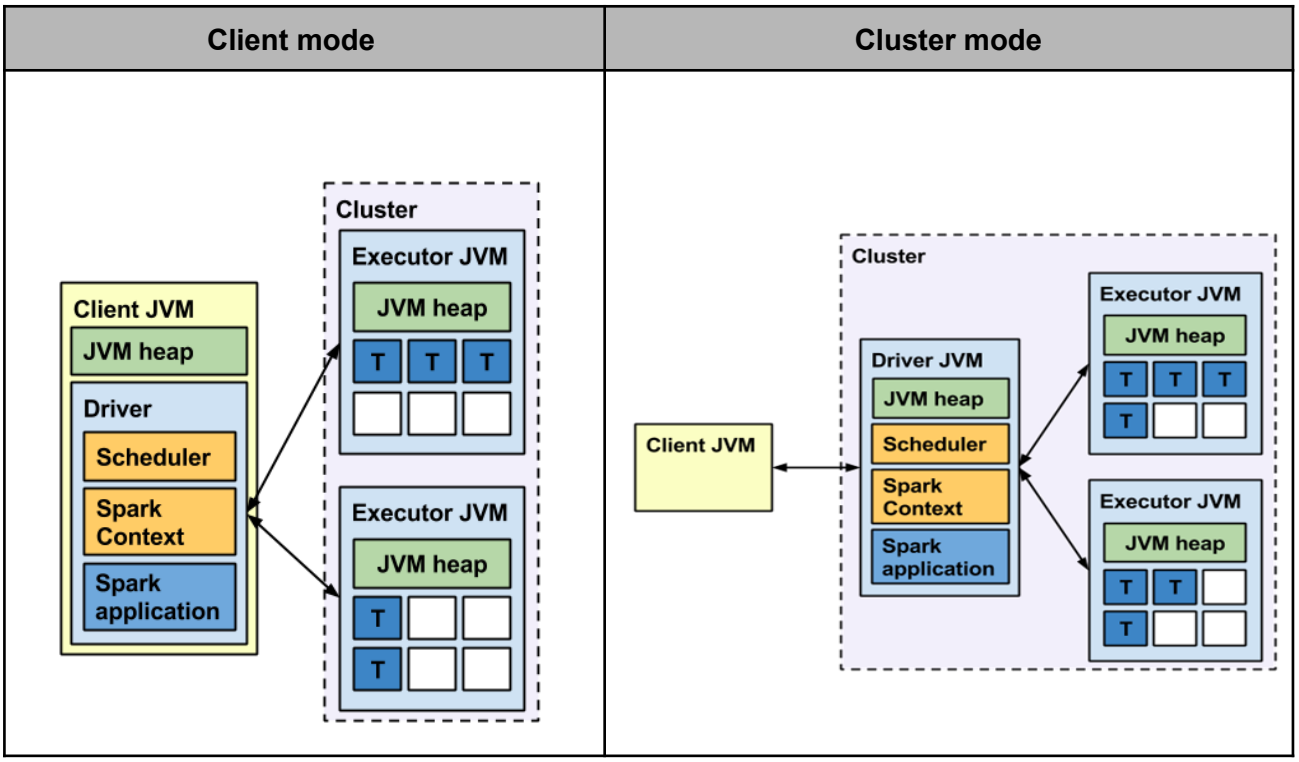
A Spark standalone cluster is a Spark-specific cluster. Because a standalone cluster's built specifically for Spark applications, it doesn't support communication with an HDFS secured with Kerberos authentication protocol. If you need that kind of security, use YARN for running Spark. A Spark standalone cluster, but provides faster job startup than those jobs running on YARN.

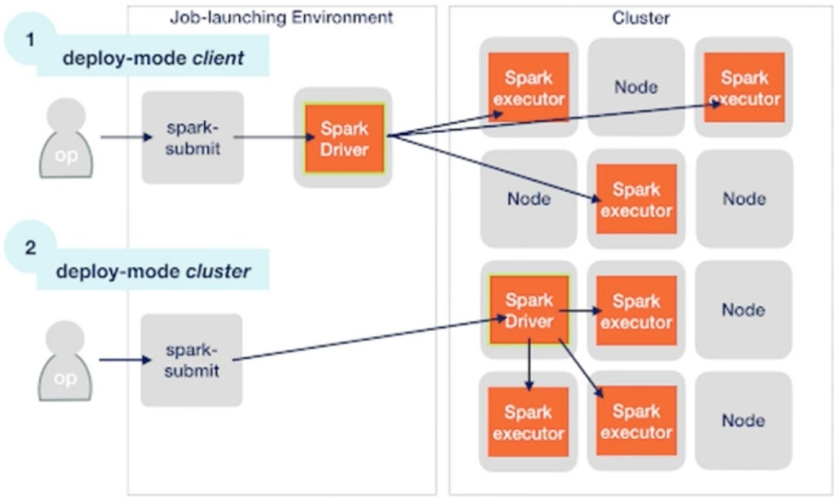
For standalone clusters it supports two deploy modes. They distinguish where the driver process runs:

- **Client mode (by default):** the driver is launched in the same process as the client that submits the application
- **Cluster mode:** the driver is launched from one of the Worker processes inside the cluster

The client process exits as soon as it fulfils its responsibility of submitting the application without waiting for the application to finish.

Note: Currently, the standalone mode does not support cluster mode for Python applications.





YARN CLUSTER

YARN in Hadoop's resource manager and execution system. It's also known as MapReduce 2 because it superseded the MapReduce engine in Hadoop 1 that supported only MapReduce jobs.

Running Spark on YARN has several advantages:

- Many organizations already have YARN clusters of a significant size, along with the technical know-how, tools, and procedures for managing and monitoring them
- Furthermore, YARN lets you run different types of Java applications, not only Spark, and you can mix legacy Hadoop and Spark applications with ease
- YARN also provides methods for isolating and prioritizing applications among users and organizations, a functionality the standalone cluster doesn't have
- It's the only cluster type that supports Kerberos-secured HDFS
- Another advantage of YARN over the standalone clusters that you don't have to install Spark on every node in the cluster.

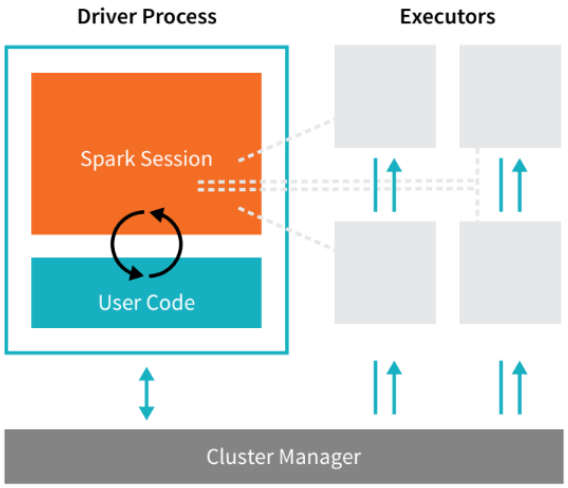
MESOS CLUSTER

Mesos is a scalable and fault-tolerant "distributed systems kernel" written in C++. Running Spark in a Mesos cluster also has its advantages. Unlike YARN, Mesos also supports C++ and Python applications, and unlike YARN and a standalone Spark cluster that only schedules memory, Mesos provides scheduling of other types of resources (for example, CPU, disk space and ports), although these additional resources aren't used by Spark currently. Mesos has some additional options for job scheduling that other cluster types don't have (for example, fine-grained mode).

And, Mesos is a "*scheduler of scheduler frameworks*" because of its two-level scheduling architecture. The jury's still out on which is better: YARN or Mesos; but now, with the Myriad project (https://github.com/mesos/myriad), you can run YARN on top of Mesos to solve the dilemma.

SPARK APPLICATIONS

Spark Applications consist of a driver process and a set of executor processes. In the illustration we see our **driver is on the left and four executors on the right.**



POPULAR SPARK USE CASES

Whether you are a data engineer, data scientist, or machine learning engineer, you'll find Spark useful for the following use cases:

- Processing in parallel large data sets distributed across a cluster
- Performing ad hoc or interactive queries to explore and visualize data sets
- Building, training, and evaluating machine learning models using MLlib
- Implementing end-to-end data pipelines from myriad streams of data
- Analyzing graph data sets and social networks

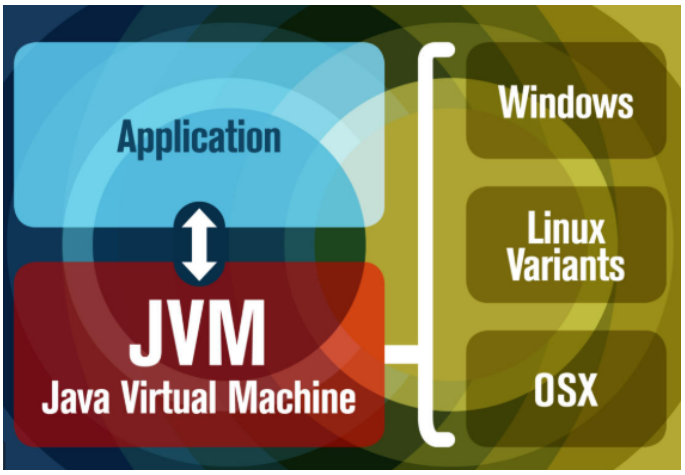
WHAT IS A JVM?

The JVM manages system memory and provides a portable execution environment for Java-based applications

- **Technical definition:** The JVM is the specification for a software program that executes code and provides the runtime environment for that code

- **Everyday definition:** The JVM is how we run our Java programs. We configure the JVM's settings and then rely on it to manage program resources during execution.

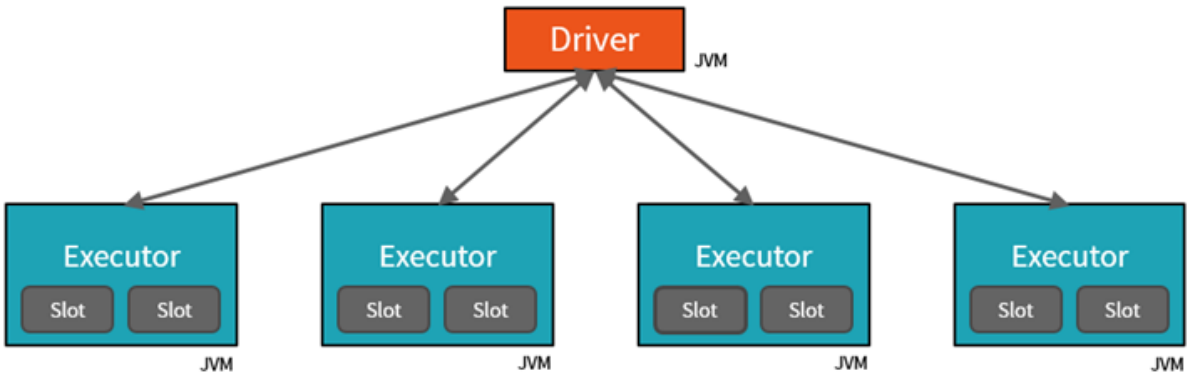
The Java Virtual Machine (JVM) is a program whose purpose is to execute other programs.

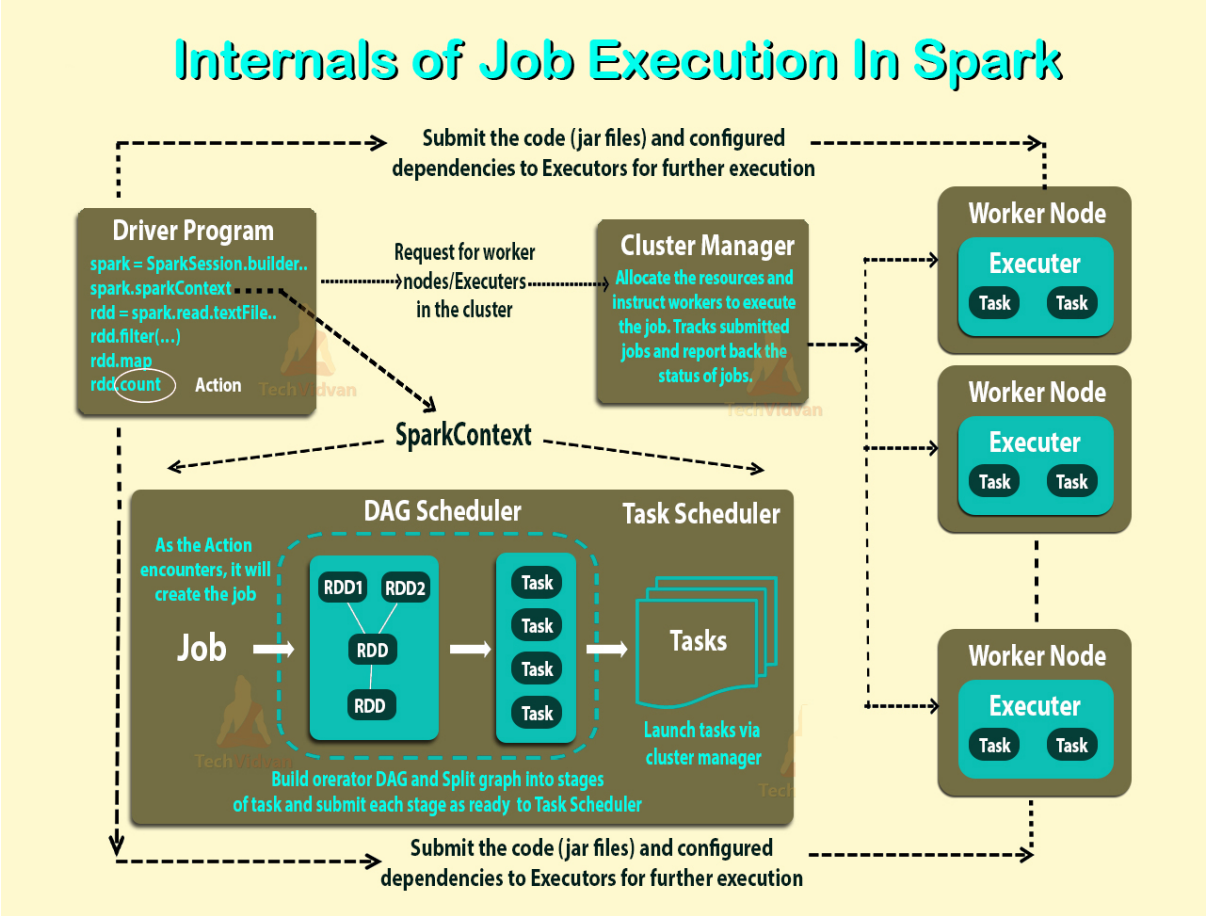


The JVM has two primary functions:

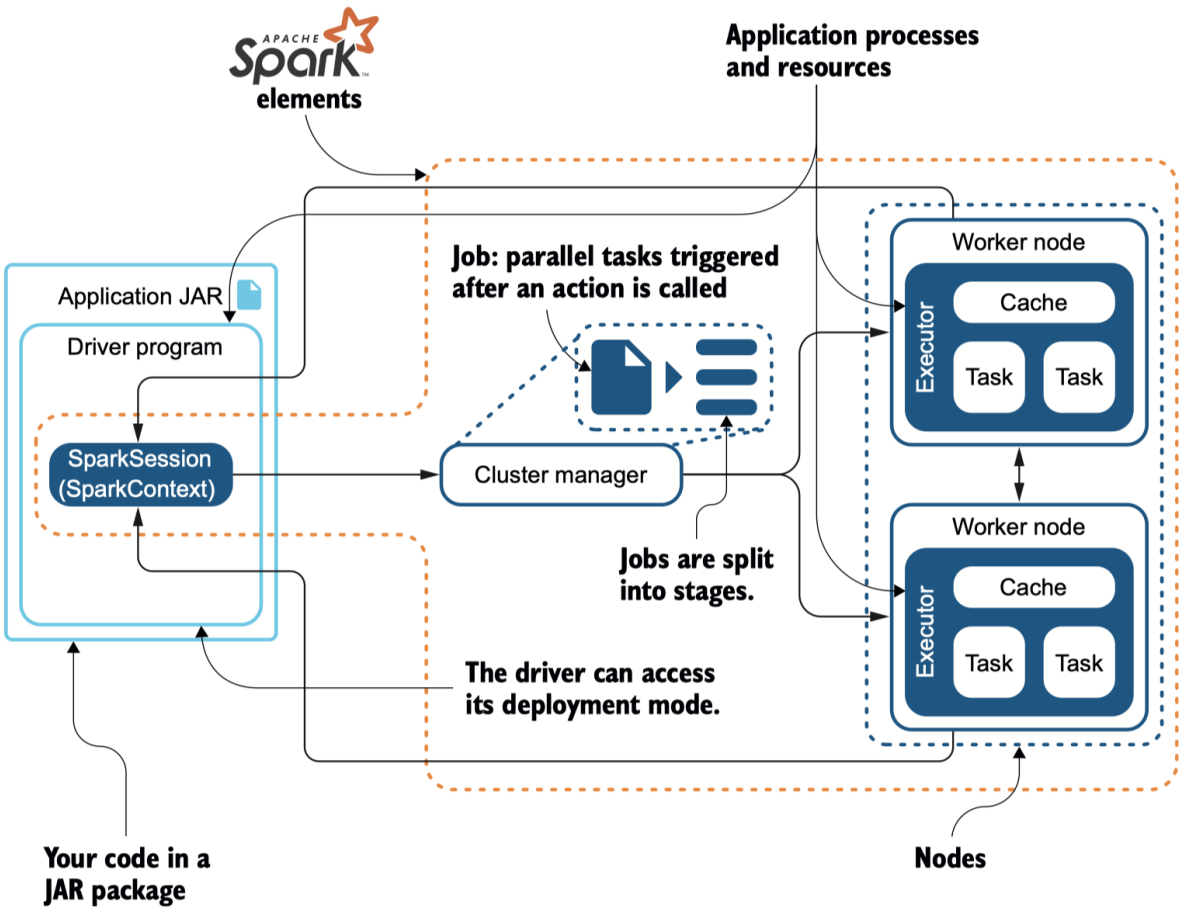
- To allow Java programs to run on any device or operating system (known as the "Write once, run anywhere" principle)
- To manage and optimize program memory

The Spark runtime architecture leverages JVMs:

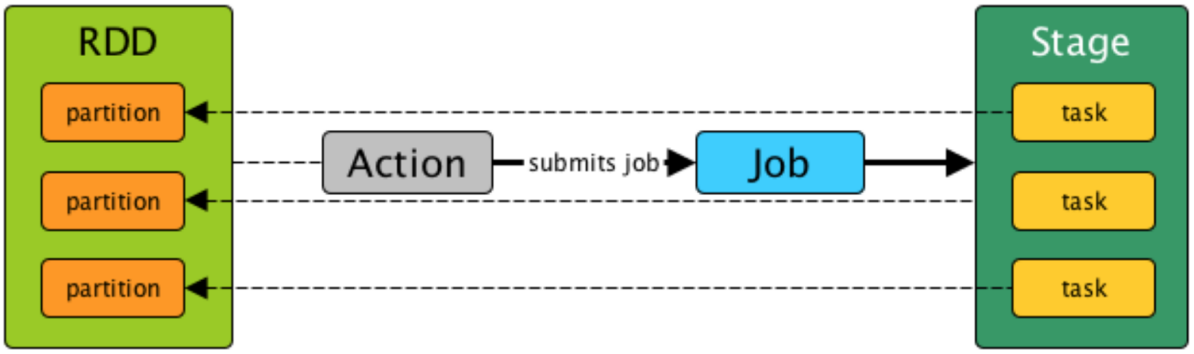




SPARK COMPONENTS



- **Driver:** is one of the nodes in the Cluster
- **Executor:** are JVMs that run on Worker nodes
- **Job:** set of tasks executed as a result of an action
- **Stage:** set of tasks in a job that can be executed in parallel – at partition level
- **Task:** individual unit of work sent to one executor over a sequences of partitions
- **RDD:** Parallel dataset with partitions
- **DAG:** Logical Graph of RDD operations



CLIENT PROCESS

The client process starts the **driver program**. For example, the client process can be a **spark-submit** script for running applications, a **spark-shell** script, or a custom application using **Spark API**. The client process prepares the classpath and all configuration options for the Spark application. It also passes application arguments, if any, to the application running inside the driver.

SPARK-SUBMIT OPTIONS

- **master**: determines how to run the job:
  - **local** - Run Spark locally with one worker thread (i.e. no parallelism at all);
  - **local[K]** - Run Spark locally with K worker threads (ideally, set this to the number of cores on your machine);
  - **local[K,F]** - Run Spark locally with K worker threads and F maxFailures (see spark.task.maxFailures for an explanation of this variable);
  - **local[*]** - Run Spark locally with as many worker threads as logical cores on your machine;
  - **local[*,F]** - Run Spark locally with as many worker threads as logical cores on your machine and F maxFailures;
  - **spark://HOST:PORT** - Connect to the given Spark standalone cluster master. The port must be whichever one your master is configured to use, which is 7077 by default;
  - **spark://HOST1:PORT1,HOST2:PORT2** - Connect to the given Spark standalone cluster with standby masters with Zookeeper. Port :7077 by default;
  - **mesos://HOST:PORT** - Connect to the given Mesos cluster;
  - **Yarn** - Connect to a YARN cluster in client or cluster mode depending on the value of --deploy-mode.
- **driver-memory**: amount memory available for the driver process.
- **executor-memory**: amount of memory allocated to the executor process
- **executor-cores**: total number of cores allocated to the executor process
- **total-executor-cores**: Total number of cores available for all executors

See: https://spark.apache.org/docs/latest/submitting-applications.html

SPARK SESSION

Spark session instance is the handle through which spark executes spark user defined manipulations across clusters.

There is 1-1 between spark session and spark application.

Gives access to low level configs and contexts.

subsume previous entry points to the Spark like the SparkContext, SQLContext, HiveContext, SparkConf, and StreamingContext.

Spark Session allows you to create JVM runtime parameters, define DataFrames and Datasets, read from data sources, access catalog metadata, and issue Spark SQL queries.

DRIVER

The driver is the machine in which the application runs. The driver **orchestrates and monitors** execution of a Spark application. There's always **one driver per Spark application**. You can think of the driver as a wrapper around the application. The driver and its subcomponents – the Spark Context and Scheduler – are responsible for:

- maintaining information about the Spark Application
- responding to the user's program
- analyzing, distributing and scheduling work across the executors
- requesting memory and CPU resources from cluster managers
- breaking application logic into stages and tasks
- sending tasks to executors
- collecting the results

Note: In a single Databricks cluster, there will only be one driver, regardless of the number of executors

Two basic ways the driver program can be run are:

- Cluster deploy mode: In this mode, the driver process runs as a separate JVM process inside a cluster, and the cluster manages its resources (mostly JVM heap memory).
- Client deploy mode: In this mode, the driver's running inside the client's JVM process and communicates with the executors managed by the cluster.

The deploy mode you choose affects how you configure Spark and the resource requirements of the client JVM.

EXECUTORS

Each executor will hold a chunk of the data to be processed. This chunk is called a Spark **partition**. It is a collection of rows that sits on one physical machine in the cluster.

Note: this is completely separate from hard disk partitions, which have to do with the storage space on a hard drive.

The executors are responsible for carrying out the work assigned by the driver. Each executor is responsible for two things:

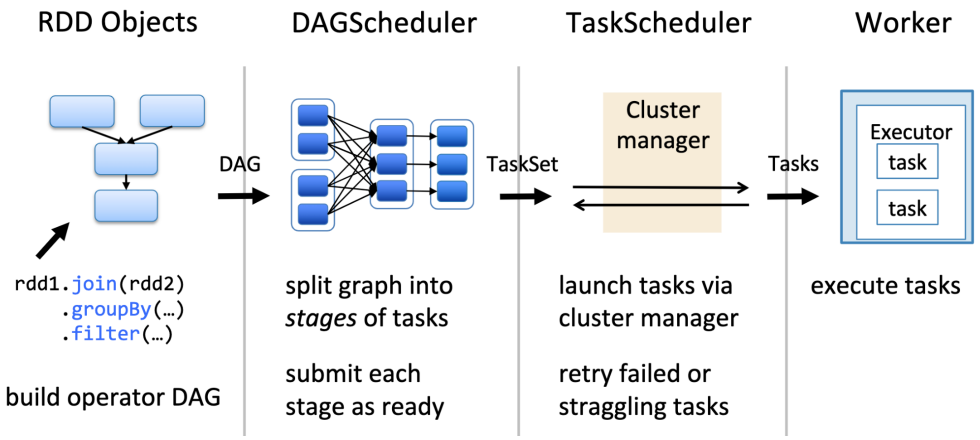
- Execute code assigned by the driver
- Report the state of the computation back to the driver

Driver	Responsible for assigning work that will be completed in parallel
Executor	Reports the state of some computation back to a central system
Spark Partition	A chunk of data (collection of rows) that sit on a single machine in a cluster

JOB

A Job is a sequence of Stages, triggered by an Action such as .count(), foreachRdd(), collect(), read() or write().

- Highest element of Spark's execution hierarchy.
- Each Spark job corresponds to one Action

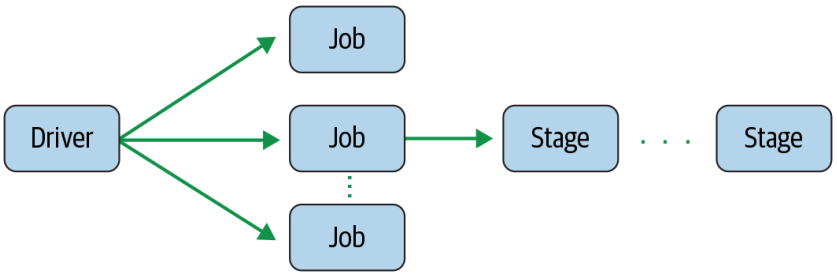


STAGE

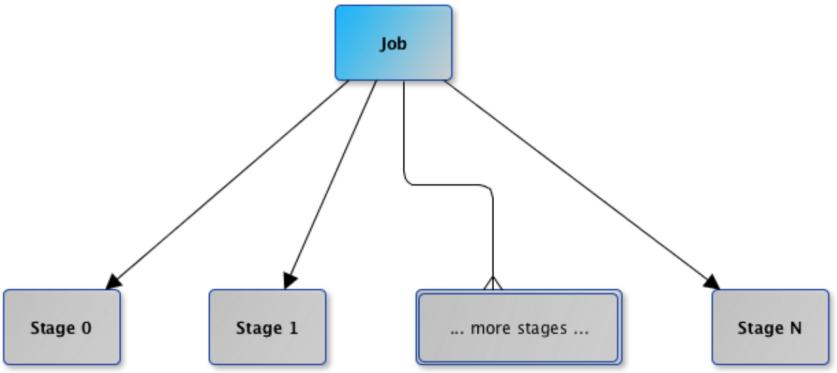
A Stage is a sequence of Tasks that can all be run together, in parallel, without a shuffle.

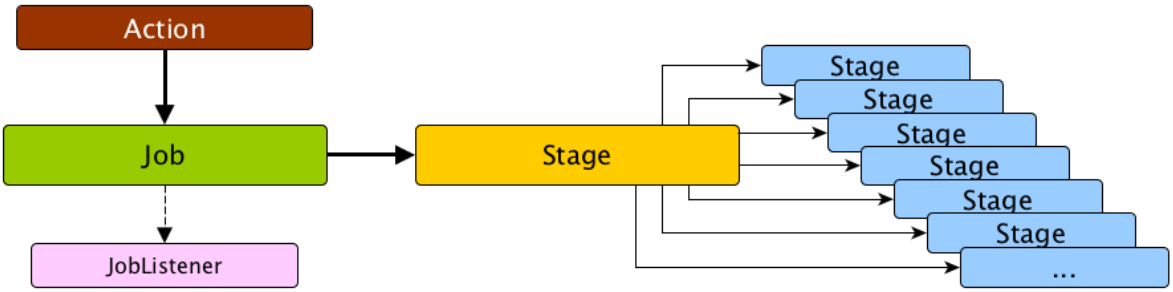
For example: using .read to read a file from disk, then running .map and .filter can all be done without a shuffle, so it can fit in a single stage.

As part of the DAG nodes, **stages are created based on what operations can be performed serially or in parallel.** Not all Spark operations can happen in a single stage, so they may be divided into multiple stages. Often stages are delineated on the operator's computation boundaries, where they dictate data transfer among Spark executors.



- a job is defined by calling an action.
- the action may include several transformations, which break down jobs into stages.
- several transformations with narrow dependencies can be grouped into one stage
- it is possible to execute stages in parallel if they are used to compute different RDDs
- wide transformations that are needed to compute one RDD have to be computed in sequence
- one stage can be computed without moving data across the partitions
- Within one stage, the tasks are the unit of work done for each partition of the data



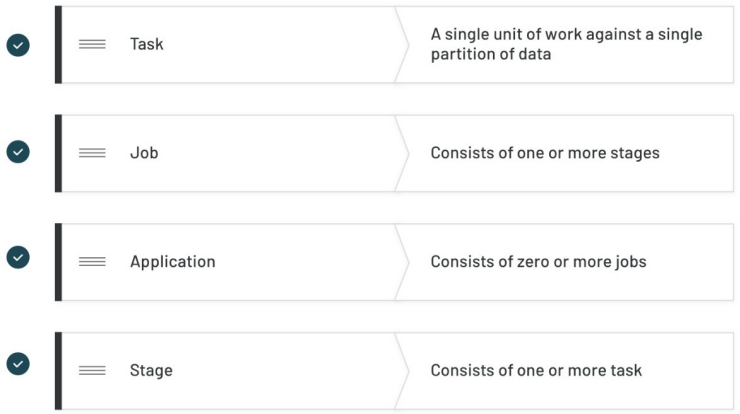


TASK

Tasks are created by the driver and assigned a **partition** of data to process. A partition is a collection of rows that sit on one physical machine in your cluster. Then, tasks are assigned to **slots** for parallel execution. Once started, each task will fetch its assigned partition from the original data source.

- A stage consists of tasks
- The task is the smallest unit in the execution hierarchy
- Each task can represent one local computation

- One task cannot be executed on more than one executor
- However, each executor has a dynamically allocated number of slots for running tasks
- The number of tasks per stage corresponds to the number of partitions in the output RDD of that stage



SLOTS

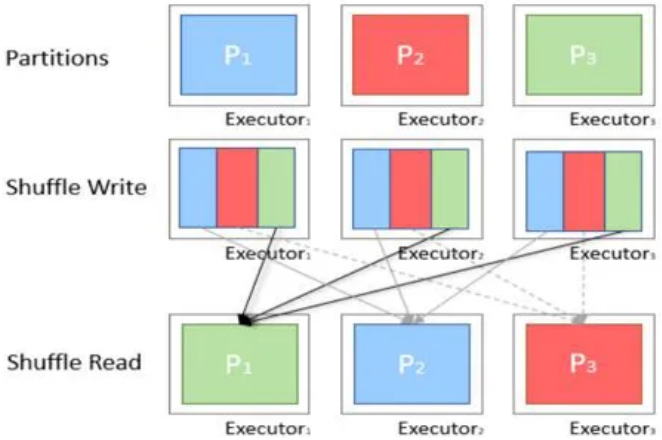
Spark parallelizes at two levels. One is the splitting the work among **executors**. The other is the slot. Each executor has a number of **slots**. Each slot can be assigned a **task**.

SHUFFLE

A Shuffle refers to an operation where data is re-partitioned across a Cluster.

join() and any operation that ends with ByKey will trigger a Shuffle.

It is a costly operation because a lot of data can be sent via the network.



In RDD, the below are a few operations and examples of shuffle:

- subtractByKey
- groupBy

- foldByKey
- reduceByKey
- aggregateByKey
- transformations of a join of any type
- distinct
- cogroup

Important points to be noted about Shuffle in Spark:

1. Spark Shuffle partitions have a static number of shuffle partitions.
2. Shuffle Spark partitions do not change with the size of data.
3. 200 is an overkill for small data, which will lead to lowering the processing due to the schedule overheads.
4. 200 is smaller for large data, and it does not use all the resources effectively present in the cluster.

And to overcome such problems, the shuffling partitions in spark should be done dynamically.

PARTITION

A Partition is a logical chunk of your RDD/Dataset.

Data is split into Partitions so that each Executor can operate on a single part, enabling parallelization.

It can be processed by a single Executor core.

For example: If you have 4 data partitions and you have 4 executor cores, you can process each Stage in parallel, in a single pass.

DAG

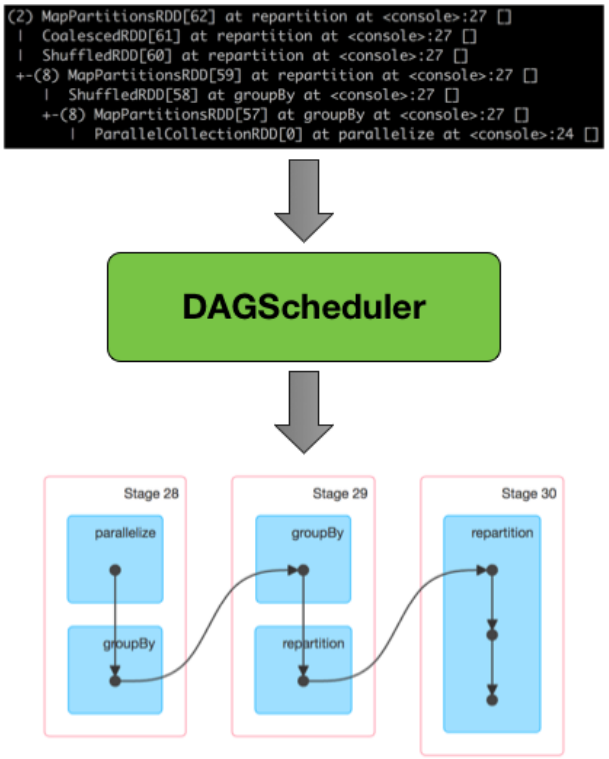
DAG stands for directed acyclic graph.

Has no directed circles.

Used to construct execution plans for spark code, form different stages and optimize on the same.

DAGScheduler

DAGScheduler is the scheduling layer of Apache Spark that implements stage-oriented scheduling. It transforms a logical execution plan to a physical execution plan (using stages).



After an action (see below) has been called, SparkContext hands over a logical plan to DAGScheduler that it in turn translates to a set of stages that are submitted as a set of tasks for execution.

The fundamental concepts of DAGScheduler are jobs and stages that it tracks through internal registries and counters.

DAGScheduler is the scheduling layer of Apache Spark that implements stage-oriented scheduling:

- It transforms a logical execution plan to a physical execution plan (using stages).
- **Uses a fifo scheduler**
- Fair scheduler runs jobs in round robin fashion
- To enable fair scheduler set **spark.scheduler.mode**= Fair

CATALOG

Catalog is the interface for managing a metastore (aka metadata catalog) of relational entities (e.g. database(s), tables, functions, table columns and temporary views).

Catalog is available using **SparkSession.catalog** property.

- The catalog is an Interface through which the user may create, drop, alter or query underlying databases, tables, functions, etc.

SPARK CONFIG

- **spark.executor.instances:** Number of executors for the spark application.

- **spark.executor.memory:** Amount of memory to use for each executor that runs the task.
- **spark.executor.cores:** Number of concurrent tasks an executor can run.
- **spark.driver.memory:** Amount of memory to use for the driver.
- **spark.driver.cores:** Number of virtual cores to use for the driver process.
- **spark.sql.shuffle.partitions:** Number of partitions to use when shuffling data for joins or aggregations.
- **spark.default.parallelism:** Default number of partitions in resilient distributed datasets (RDDs) returned by transformations like join and aggregations.

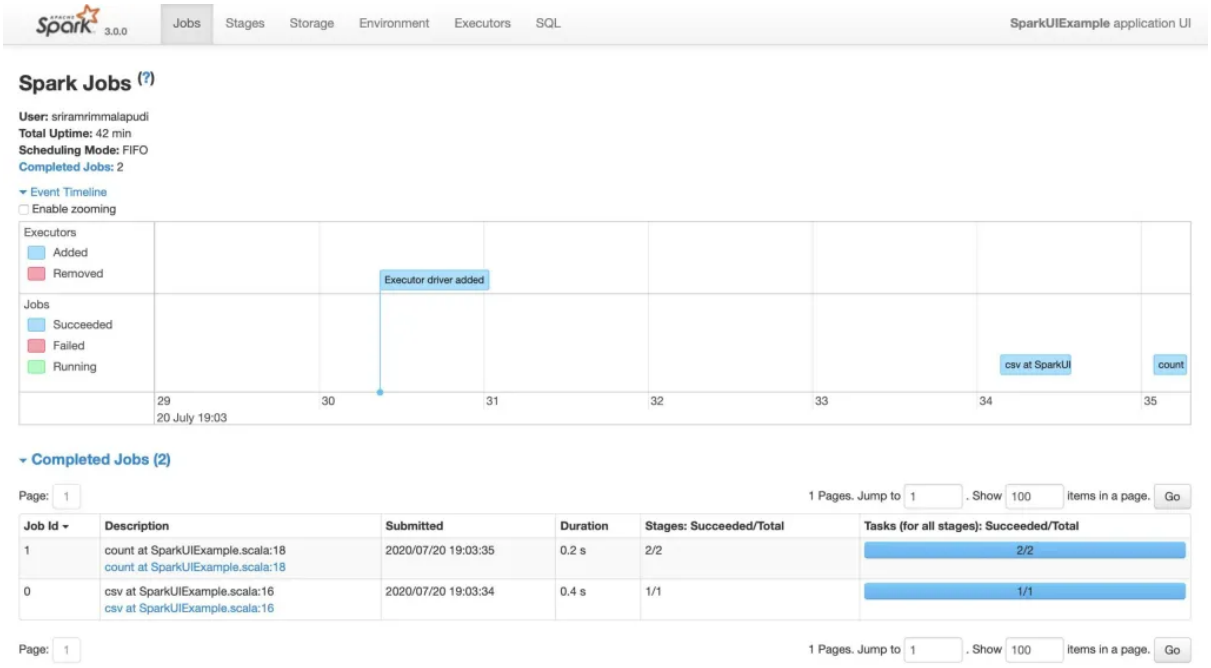
Yarn has different configuration settings which can be looked at as well

SPARK UI

Spark UI is separated into below tabs.

- Spark Jobs
- Stages
- Tasks
- Storage
- Environment
- Executors
- SQL

If you are running the Spark application locally, Spark UI can be accessed using the **http://localhost:4040/** . Spark UI by default runs on port 4040 and below are some of the additional UI's that would be helpful to track Spark application.



- Spark Application UI: http://localhost:4040/
- Resource Manager: http://localhost:9870
- Spark JobTracker: http://localhost:8088/
- Node Specific Info: http://localhost:8042/

Note: To access these URLs, Spark application should in running state. If you wanted to access this URL regardless of your Spark application status and wanted to access Spark UI all the time, you would need to start Spark History server.

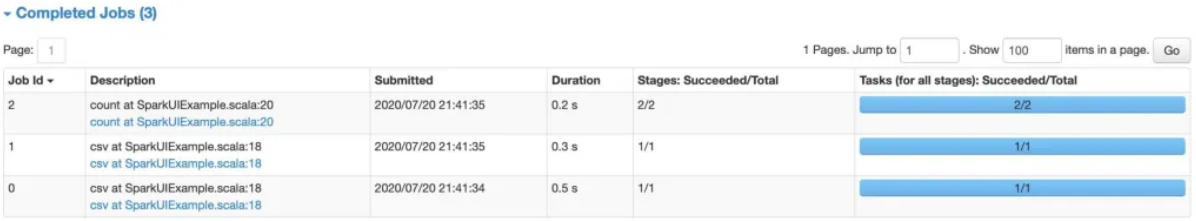
SPARK JOBS TAB

The details that I want you to be aware of under the jobs section are Scheduling mode, the number of Spark Jobs, the number of stages it has, and Description in your spark job.

SCHEDULING MODE

We have three Scheduling modes.

- Standalone mode
- YARN mode
- Mesos



NUMBER OF SPARK JOBS

Always keep in mind, the number of Spark jobs is equal to the number of actions in the application and each Spark job should have at least one Stage.

In our above application, we have performed 3 Spark jobs (0,1,2)

- Job 0. read the CSV file.
- Job 1. Inferschema from the file.
- Job 2. Count Check

So if we look at the fig it clearly shows 3 Spark jobs result of 3 actions.

NUMBER OF STAGES

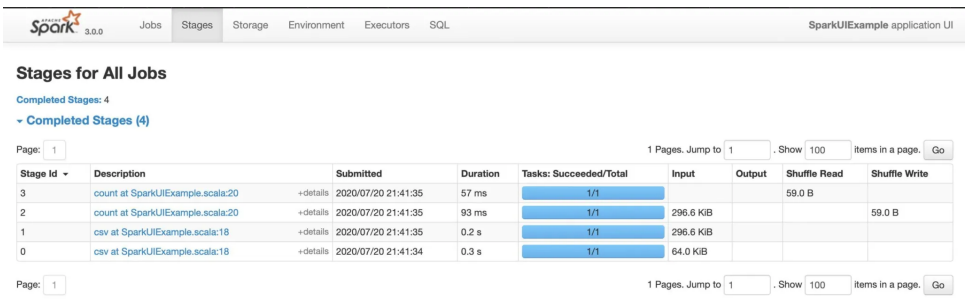
Each Wide Transformation results in a separate Number of Stages.

In our case, Spark job0 and Spark job1 have individual single stages but when it comes to Spark job 3 we can see two stages that are because of the partition of data. Data is partitioned into two files by default.

DESCRIPTION

Description links the complete details of the associated SparkJob like Spark Job Status, DAG Visualization, Completed Stages

STAGES TAB



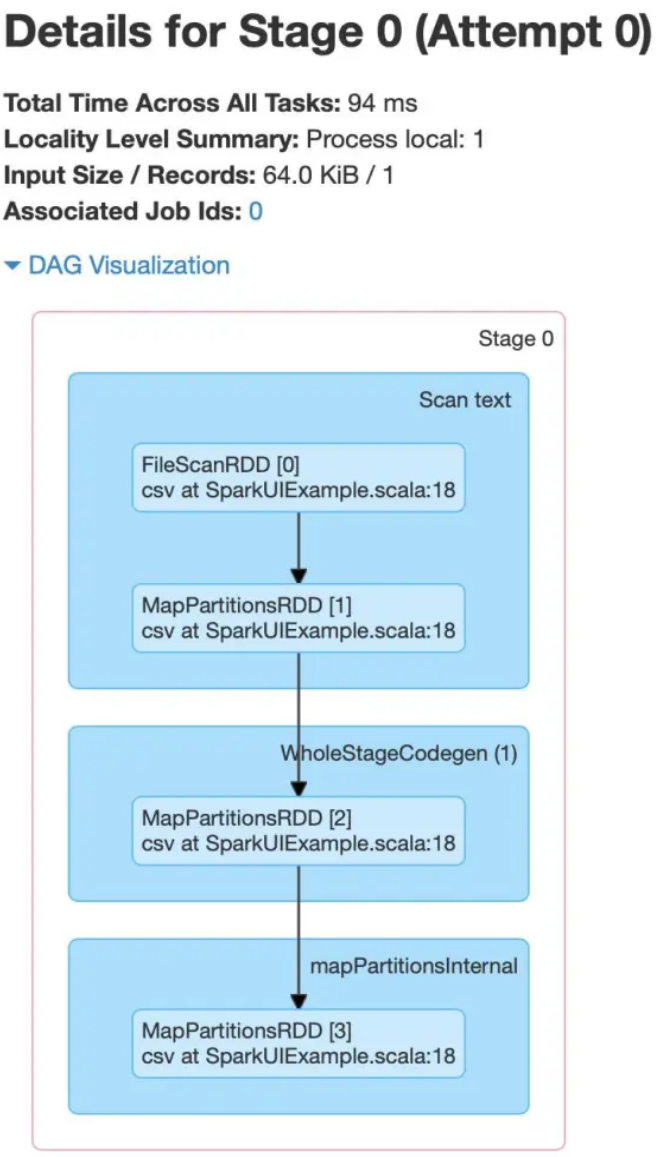
We can navigate into Stage Tab in two ways.

- Select the Description of the respective Spark job (Shows stages only for the Spark job opted)

- On the top of Spark Job tab select Stages option (Shows all stages in Application)
- In our application, we have a total of 4 Stages.

The Stage tab displays a summary page that shows the current state of all stages of all Spark jobs in the spark application

The number of tasks you could see in each stage is the number of partitions that spark is going to work on and each task inside a stage is the same work that will be done by spark but on a different partition of data.



STAGE DETAIL

Details of stage showcase Directed Acyclic Graph (DAG) of this stage, where vertices represent the RDDs or DataFrame and edges represent an operation to be applied.

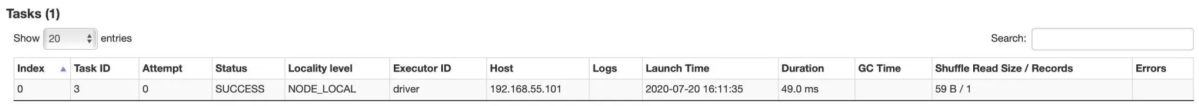
let us analyze operations in Stages

Operations in Stage0 are:

- 1.FileScanRDD
    - FileScan represents reading the data from a file.
    - It is given FilePartitions that are custom RDD partitions with PartitionedFiles (file blocks)

- 2.MapPartitionsRDD
    - MapPartitionsRDD will be created when you use map Partition transformation

TASKS TAB



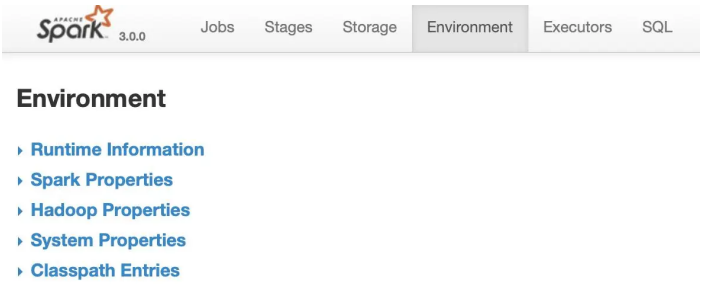
Tasks are located at the bottom space in the respective stage.
Key things to look task page are:

- 1. Input Size – Input for the Stage
- 2. Shuffle Write-Output is the stage written.

STORAGE TAB

The Storage tab displays the persisted RDDs and DataFrames, if any, in the application. The summary page shows the storage levels, sizes and partitions of all RDDs, and the details page shows the sizes and using executors for all partitions in an RDD or DataFrame.

ENVIRONMENT TAB



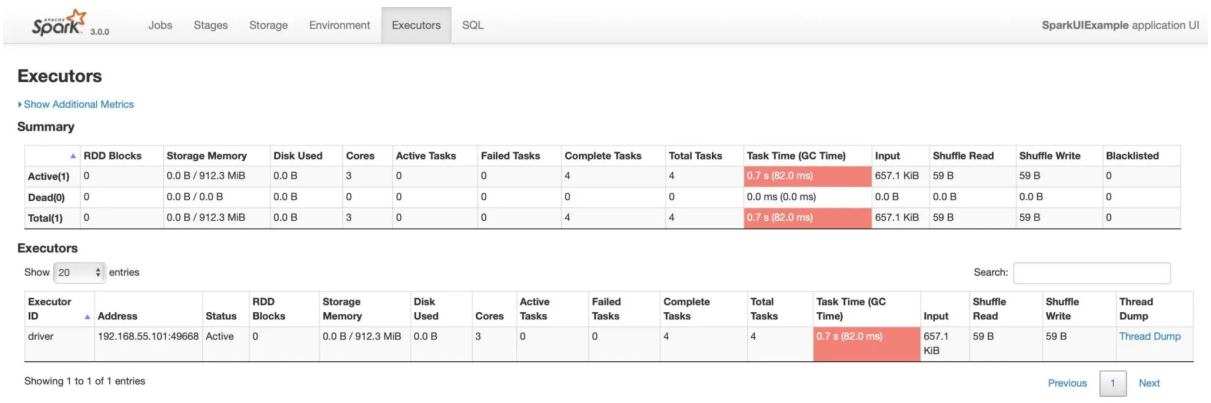
This environment page has five parts. It is a useful place to check whether your properties have been set correctly.

- **Runtime Information:** simply contains the runtime properties like versions of Java and Scala.
- **Spark Properties:** lists the application properties like ‘spark.app.name’ and ‘spark.driver.memory’.

- **Hadoop Properties:** displays properties relative to Hadoop and YARN. Note: Properties like ‘spark.hadoop’ are shown not in this part but in ‘Spark Properties’.
- **System Properties:** shows more details about the JVM.
- **Classpath Entries:** lists the classes loaded from different sources, which is very useful to resolve class conflicts.

The Environment tab displays the values for the different environment and configuration variables, including JVM, Spark, and system properties.

EXECUTORS TAB



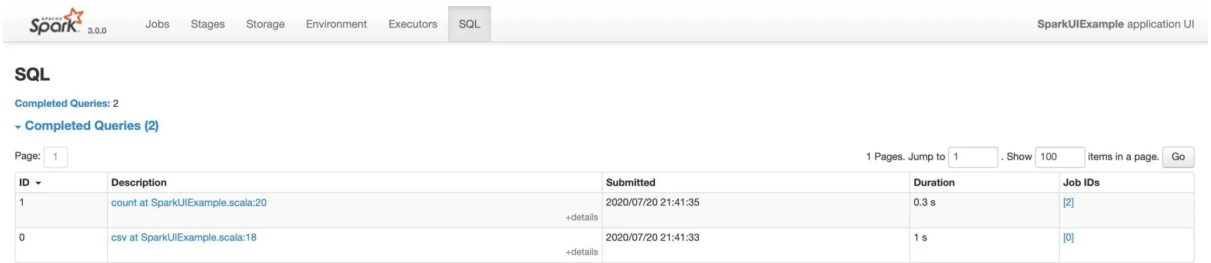
The Executors tab displays summary information about the executors that were created for the application, including memory and disk usage and task and shuffle information. The Storage Memory column shows the amount of memory used and reserved for caching data.

The Executors tab provides not only resource information like amount of memory, disk, and cores used by each executor but also performance information.

In Executors:

- Number of cores = 3 as I gave master as local with 3 threads
- Number of tasks = 4

SQL TAB



If the application executes Spark SQL queries then the SQL tab displays information, such as the duration, Spark jobs, and physical and logical plans for the queries.

In our application, we performed read and count operation on files and DataFrame. So both read and count are listed SQL Tab

TRANSFORMATION, ACTIONS AND EXECUTION

Tables are equivalent to Apache Spark DataFrames. A DataFrame is an immutable, distributed collection of data organized into named columns. In concept, a DataFrame is equivalent to a table in a relational database, except that DataFrames carry important metadata that allows Spark to optimize queries.

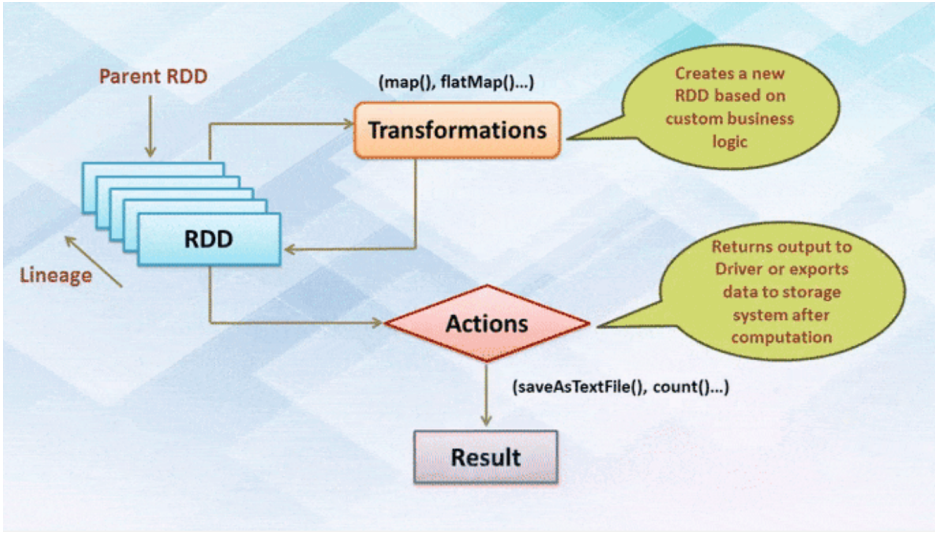
LAZY EVALUATION

Lazy Evaluation refers to the idea that Spark waits until the last moment to execute a series of operations. Instead of modifying the data immediately when you express some operation, you build up a plan of transformations that you will apply to your source data. That plan is executed when you call an action.

Lazy evaluation makes it easier to parallelize operations and allows Spark to apply various optimizations.

In general, Spark will fail only at job execution time rather than DataFrame definition time even if, for example, we point to a file that does not exist. This is due to lazy evaluation.

Lazy evaluation in Spark means that the execution will not start until an action is triggered. In Spark, the picture of lazy evaluation comes when Spark transformations occur.



There are some benefits of Lazy evaluation in Apache Spark:

- **Increases Manageability**
  - By lazy evaluation, users can organize their Apache Spark program into smaller operations. It reduces the number of passes on data by grouping operations.

- **Saves Computation and increases Speed**
    - Spark Lazy Evaluation plays a key role in saving calculation overhead. Since only necessary values get compute. It saves the trip between driver and cluster, thus speeds up the process.

- **Reduces Complexities**
    - The two main complexities of any operation are time and space complexity. Using Apache Spark lazy evaluation we can overcome both. Since we do not execute every operation, Hence, the time gets saved. It let us work with an infinite data structure. The action is triggered only when the data is required, it reduces overhead.

- **Optimization**
    - It provides optimization by reducing the number of queries.

Predicate pushdown on DataFrames:

- If we build a large Spark job but specify a filter at the end that only requires us to fetch one row from our source data, the most efficient way to execute this is to access the single record that we need. Spark will actually optimize this for us by pushing the filter down automatically.

TRANSFORMATIONS

Transformations are at the core of how you express your business logic in Spark. They are the instructions you use to modify a DataFrame to get the results that you want. We call them **lazy** because they will not be completed at the time you write and execute the code in a cell - they will only get executed once you have called an action.

There are two types of transformations: **Narrow and Wide**.

- For **narrow transformations**, the data required to compute the records in a single partition reside in at most one partition of the parent dataset.

- For **wide transformations**, the data required to compute the records in a single partition may reside in many partitions of the parent dataset.

Remember, spark partitions are collections of rows that sit on physical machines in the cluster.

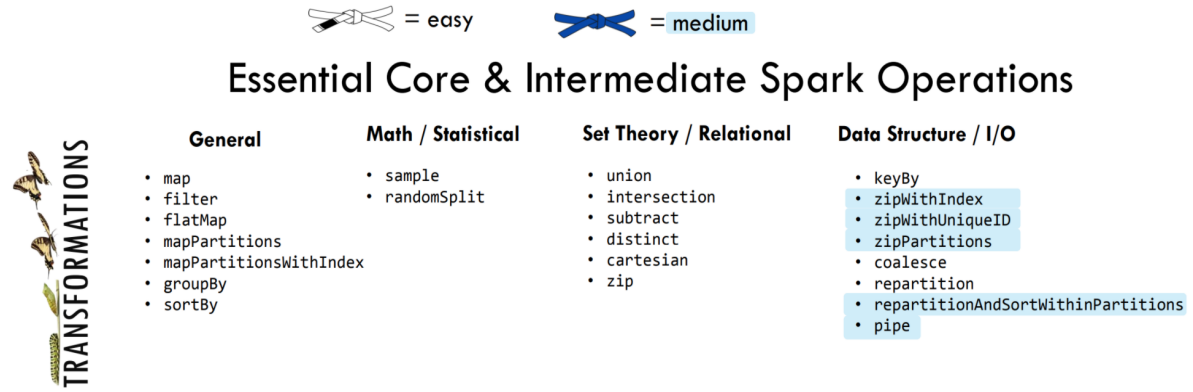
Narrow transformations mean that work can be computed and reported back to the executor without changing the way data is partitioned over the system.

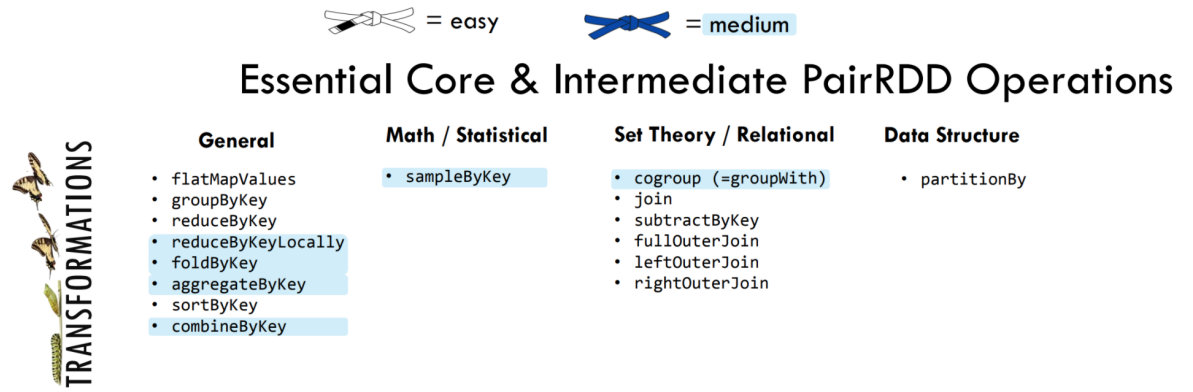
Wide transformations require that data be redistributed over the system. This is called a shuffle.

Shuffles are triggered when data needs to move between executors.

Narrow Transformation	Wide Transformation

map	intersection
flatMap	distinct
mapPartitions	reduceByKey
filter	groupByKey
sample	join
union	cartesian
mapValues	repartition
drop	groupBy
selectExpr	leftOuterJoin
withColumnRenamed	sort
coalesce	orderBy
select	
limit	
agg	





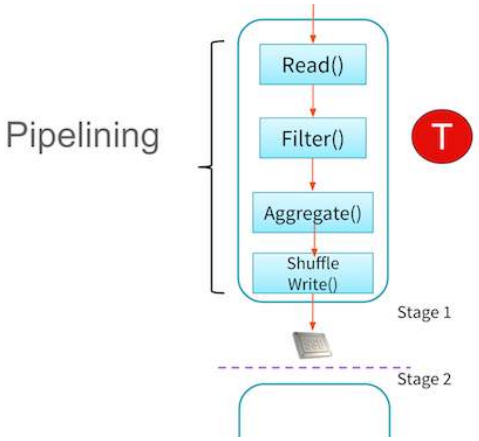
ACTIONS

Actions are statements that are computed AND executed when they are encountered in the developer's code. They are not postponed or wait for other code constructs. While transformations are lazy, actions are eager.

List of Actions functions		
show	countByValue	top
count	reduce	head
collect	fold	first
save	aggregate	
take(n)	foreach	

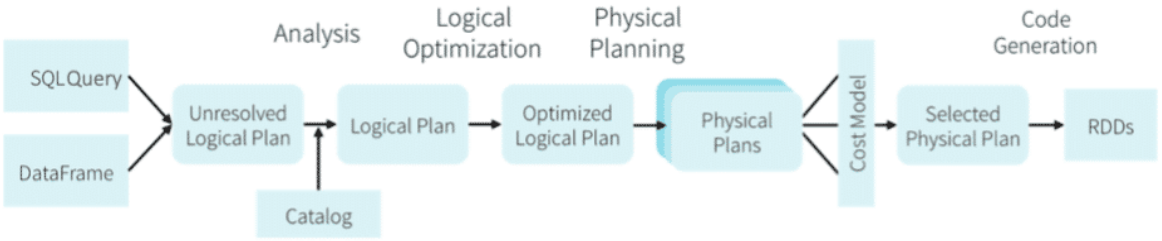
PIPELINING

Lazy evaluation allows Spark to optimize the entire pipeline of computations as opposed to the individual pieces. This makes it exceptionally fast for certain types of computation because it can perform all relevant computations at once. Technically speaking, Spark pipelines this computation, which we can see in the image below. This means that certain computations can all be performed at once (like a map and a filter), rather than having to do one operation for all pieces of data and then the following operation. Additionally, Apache Spark can keep results in-memory as opposed to other frameworks that immediately write to disk after each task.



CATALYST OPTIMIZER

The Catalyst Optimizer is at the core of Spark SQL's power and speed. It automatically finds the most efficient plan for applying your transformations and actions.



SQL QUERY AND DATAFRAME

User input: This is where you enter queries

UNRESOLVED LOGICAL PLAN

This is your logical plan for how you want to transform the data. It is called unresolved at this stage because some elements, like column names and table names for example, have not been resolved. They might not exist.

ANALYSIS

At this stage, column and table names are validated against the **catalog**. The catalog is how we refer to the metadata about the data stored in your tables. The unresolved logical plan becomes the logical plan.

Spark SQL begins with a relation to be computed, either from an abstract syntax tree (AST) returned by a SQL parser, or from a DataFrame object constructed using the API. In both cases, the relation may contain unresolved attribute references or relations: for example, in the SQL query SELECT col FROM sales, the type of col, or even whether it is a valid column name, is not known until we look up the table sales. An attribute is called unresolved if we do not know its type or have not matched it to an input table (or an alias). Spark SQL uses Catalyst rules and a Catalog object that tracks the tables in all data sources to resolve these attributes. It starts by building an “unresolved logical plan” tree with unbound attributes and data types, then applies rules that do the following:

- Looking up relations by name from the catalog.
- Mapping named attributes, such as col, to the input provided given operator’s children.
- Determining which attributes refer to the same value to give them a unique ID (which later allows optimization of expressions such as col = col).
- Propagating and coercing types through expressions: for example, we cannot know the return type of 1 + col until we have resolved col and possibly casted its subexpressions to compatible types.
- In total, the rules for the analyzer are about 1000 lines of code.

LOGICAL OPTIMIZATION

This is where the first set of optimizations take place. You query may be optimized by reordering the sequence of commands.

The logical optimization phase applies **standard rule-based optimizations** to the logical plan. (Cost-based optimization is performed by generating multiple plans using rules, and then computing their costs.) These include constant folding, predicate pushdown, projection pruning, null propagation, Boolean expression simplification, and other rules. In general, we have found it extremely simple to add rules for a wide variety of situations. For example, when we added the fixed-precision DECIMAL type to Spark SQL, we wanted to optimize aggregations such as sums and averages on DECIMALs with small precisions; it took 12 lines of code to write a rule that finds such decimals in SUM and AVG expressions, and casts them to unscaled 64-bit LONGs, does the aggregation on that, then converts the result back.

PHYSICAL PLANNING

The Catalyst Optimizer generates 1 or more physical plans that can be used to execute your query.

Each physical plan represents what the query engine will actually do after all optimizations have been applied.

In the physical planning phase, Spark SQL takes a logical plan and generates one or more physical plans, using physical operators that match the Spark execution engine. It then selects a plan using a cost model. At the moment, cost-based optimization is only used to select join algorithms: for relations that are known to be small, Spark SQL uses a broadcast join, using a peer-to-peer broadcast facility available in Spark. The framework supports broader use of cost-based optimization, however, as costs can be estimated recursively for a whole tree using a rule. We thus intend to implement richer cost-based optimization in the future.

The physical planner also performs rule-based physical optimizations, such as pipelining projections or filters into one Spark map operation. In addition, it can push operations from the logical plan into data sources that support predicate or projection pushdown. We will describe the API for these data sources in a later section.

In total, the physical planning rules are about 500 lines of code.

COST MODEL

Each physical plan is evaluated according to its cost model. The best performing model is selected. This gives us the selected physical plan.

CODE GENERATION

The selected physical plan is compiled to Java bytecode and executed.

The final phase of query optimization involves generating Java bytecode to run on each machine. Because Spark SQL often operates on in-memory datasets, where processing is CPU-bound, we wanted to support code generation to speed up execution. Nonetheless, code generation engines are often complicated to build, amounting essentially to a compiler. Catalyst relies on a special feature of the Scala language, quasiquotes, to make code generation simpler. Quasiquotes allow the programmatic construction of abstract syntax trees (ASTs) in the Scala language, which can then be fed to the Scala compiler at runtime to generate bytecode. We use Catalyst to transform a tree representing an expression in SQL to an AST for Scala code to evaluate that expression, and then compile and run the generated code.

In total, Catalyst's code generator is about 700 lines of code.

VIEWING THE METADATA

Spark manages the metadata associated with each managed or unmanaged table. This is captured in the Catalog, a high-level abstraction in Spark SQL for storing metadata. The Catalog's functionality was expanded in Spark 2.x with new public methods enabling you to examine the metadata associated with your databases, tables, and views. Spark 3.0 extends it to use external catalog.

For example, within a Spark application, after creating the SparkSession variable spark, you can access all the stored metadata through methods like these:

In Scala/Python :

```
spark.catalog.listDatabases()
spark.catalog.listTables()
spark.catalog.listColumns("us_delay_flights_tbl")
```

CACHING

- Shuffle files are by definition temporary files and will eventually be removed. However, we can cache data explicitly to accomplish the same thing that happens inadvertently with shuffle files.
- This is used for optimizing the spark query.
- LRU cache is used by default.
- Cache is eager by default, we can change this by using keyword lazy.
- **Cache the Dataframe with the default storage level (MEMORY_AND_DISK).**
- Syntax:
    - spark.catalog.cacheTable("tableName")
    - dataFrame.cache()

    - Unpersist() - used to forcibly remove rdd from head
    - spark.sql("cache lazy table table_name")

UNCACHE TABLE

- Removes the entries and associated data from the in-memory and/or on-disk cache for a given table or view.
- The underlying entries should already have been brought to cache by the previous CACHE TABLE operation.
- The UNCACHE TABLE on a non-existent table throws an Exception if IF EXISTS is not specified.
- Syntax
    - UNCACHE TABLE [IF EXISTS] table_name
- Parameters
    - var: table_name
    - def: The name of the table or view to be uncached.
- Example:
    - UNCACHE TABLE t1
    - spark.catalog.clearCache()

RDD PERSISTENCE

One of the most important capabilities in Spark is persisting (or caching) a dataset in memory across operations.

All different persistence (persist() method) storage level Spark/PySpark supports are available at **org.apache.spark.storage.StorageLevel** and pyspark.StorageLevel classes respectively

Spark persist has two signature first signature doesn't take any argument which by default saves it to **MEMORY_AND_DISK**

Syntax:

- persist() : Dataset.this.type
- persist(newLevel : org.apache.spark.storage.StorageLevel) : Dataset.this.type

PERSIST OPTIONS

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.

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, 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. This requires off-heap memory to be enabled.

WHICH STORAGE LEVEL TO CHOOSE?

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 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, recomputing 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 recomputing lost data**, but the replicated ones let you continue running tasks on the RDD without waiting to recompute a lost partition.

DYNAMIC PARTITION PRUNING

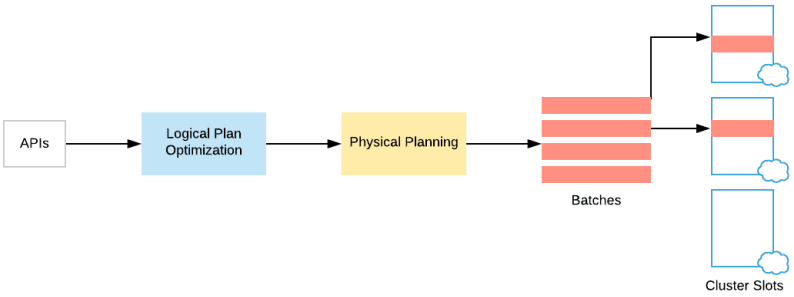
In standard database pruning means that the optimizer will avoid reading files that cannot contain the data that you are looking for. For example,

```
Select * from Students where subject = 'English';
```

In this simple query, we are trying to match and identify records in the Students table that belong to subject English.

DPP is implemented based on the **partition pruning** and **Broadcast hashing**. Star-schema tables are broadly classified into Fact and dimension tables, mostly where the dimension table is much smaller compared to the fact table. When joining these tables, DPP creates an internal subquery acquired out of the filter applied on the Dimension table, broadcasts it, makes a hash table out of it and internally applies it to the physical plan of the Fact table, before the scan phase.

- Spark's optimizes physical plan to pass a filter in scanning phase
- Partitions that doesn't contains relevant data are not read
- Significant in star-schema architecture based table queries
- Doesn't need any additional configuration to be done with spark config

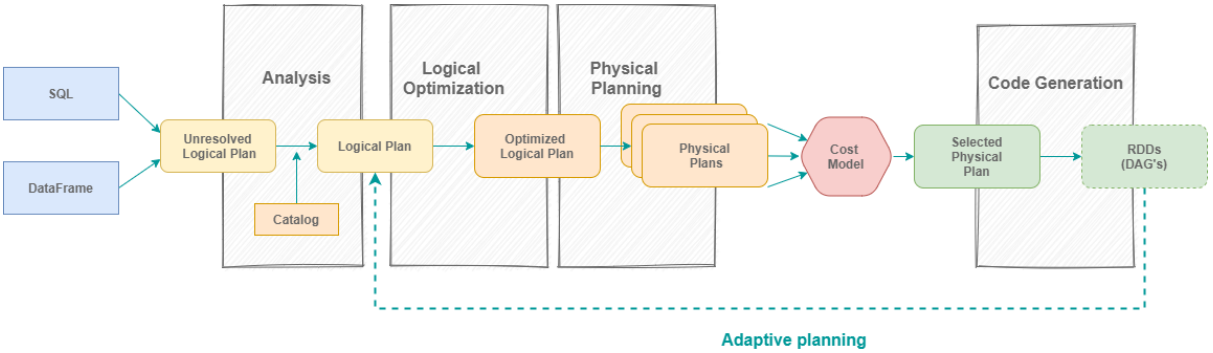


More details: [Dynamic Partition Pruning in Spark 3.0]

ADAPTIVE QUERY EXECUTION

Another way Spark 3.0 optimizes query performance is by adapting its physical execution plan at runtime. Adaptive Query Execution (AQE) reoptimizes and adjusts query plans based on runtime statistics collected in the process of query execution. It attempts to to do the following at runtime:

- Reduce the number of reducers in the shuffle stage by decreasing the number of shuffle partitions.
- Optimize the physical execution plan of the query, for example by converting a SortMergeJoin into a BroadcastHashJoin where appropriate.
- Handle data skew during a join.
- Cost optimization is done based on initial statistics which is different from runtime statistics.

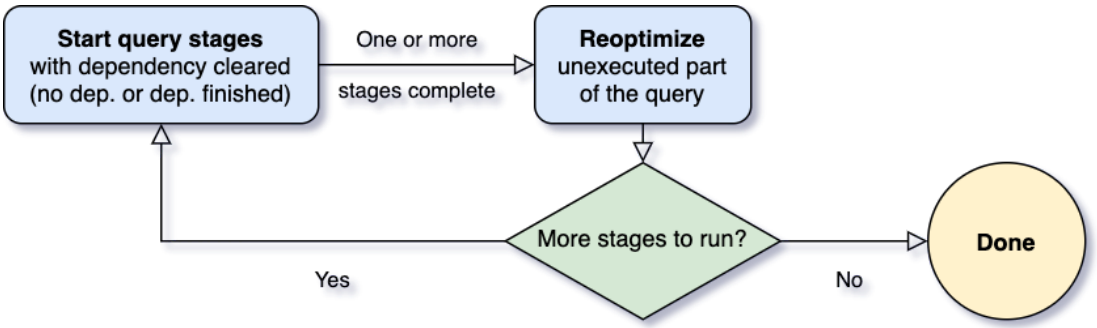


[A hitchhiker's guide to Spark's AQE — exploring dynamically coalescing shuffle partitions]

- AQE aims to optimize on the same.
- This is done at stage boundaries.
- Not applicable for streaming queries
- **spark.sql.adaptive.enabled=true** needs to be set

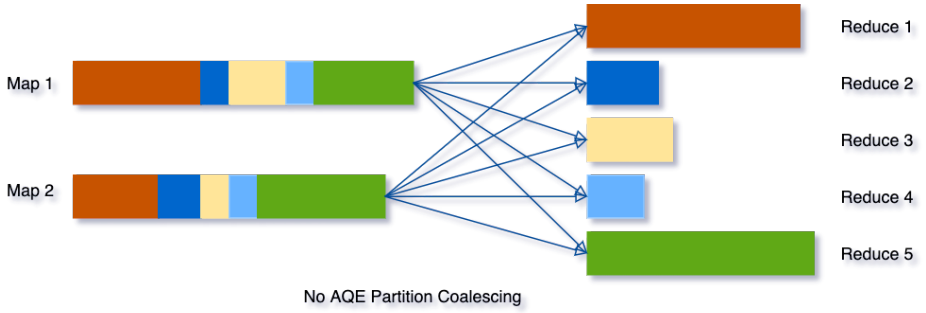
In Spark 3.0, the AQE framework is shipped with three features:

- Dynamically coalescing shuffle partitions
- Dynamically switching join strategies
- Dynamically optimizing skew joins

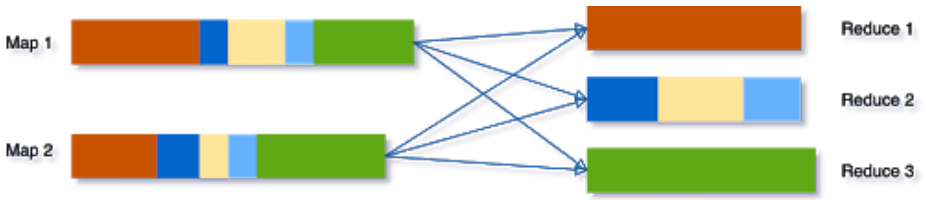


DYNAMICALLY COALESCING SHUFFLE PARTITIONS

https://databricks.com/blog/2020/05/29/adaptive-query-execution-speeding-up-spark-sql-at-runtime.html

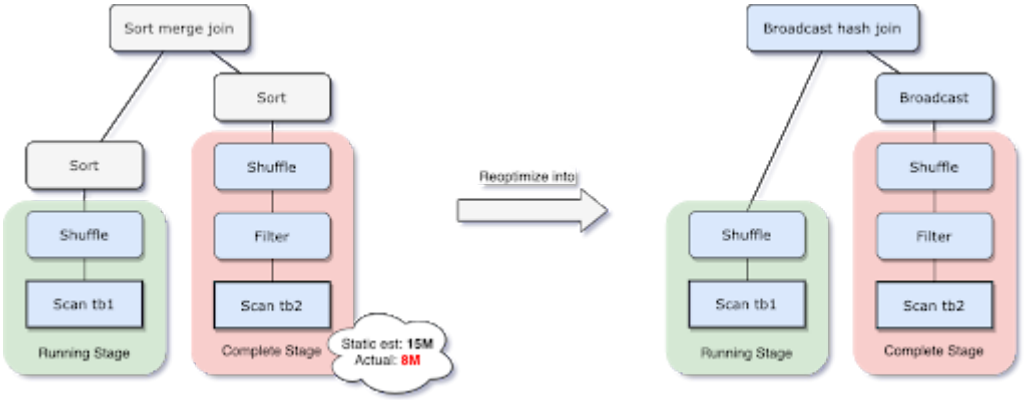


No AQE Partition Coalescing

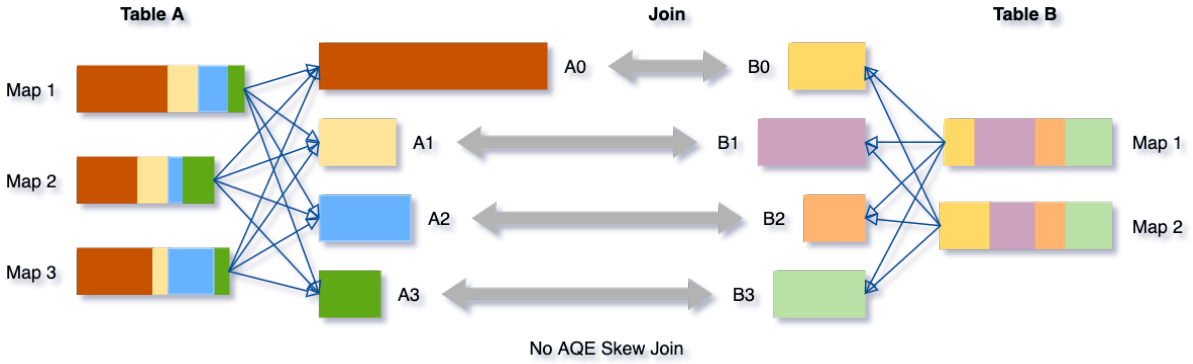


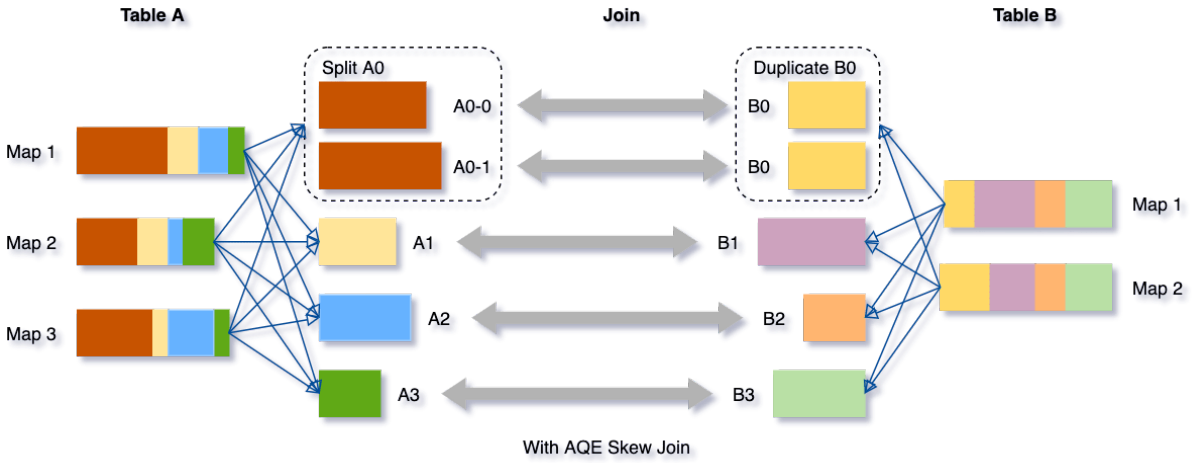
With AQE Partition Coalescing

DYNAMICALLY SWITCHING JOIN STRATEGIES



DYNAMICALLY OPTIMIZING SKEW JOINS





BROADCAST VARIABLES

Broadcast variables allow the programmer to **keep a read-only variable cached on each machine** rather than shipping a copy of it with tasks.

They can be used, for example, to give every node a copy of a large input dataset in an efficient manner.

Spark also attempts to distribute broadcast variables using efficient broadcast algorithms to reduce communication cost.

After the broadcast variable is created, it should be used in any functions run on the cluster. In addition, the object should **not be modified after it is broadcast** in order to ensure that all nodes get the same value of the broadcast variable (e.g. if the variable is shipped to a new node later).

- Shared immutable variable cached on every machine on the cluster.
- Present in the driver and sent across clusters.
- Broadcastjoin is used when we are joining 2 tables and want to broadcast a complete table **(size limit 10mb)**
- No shuffle incurred.

REPARTITION

Repartition is a method in spark which is used to perform a full shuffle on the data present and create partitions based on the user's input.

- Method 1: Repartition using Column Name: **df = df.repartition('_1')**
- Method 2: Repartition using integer value: **df = df.repartition(3)**

By default, 200 partitions are created if the number is not specified in the repartition clause.

Use Repartition only when you want to **increase or decrease** the number of partitions (or) if you want to perform a full shuffle on the data.

You can optionally specify the number of partitions you would like, too:

- `df.repartition(5, col("DEST_COUNTRY_NAME"))`

Coalesce, on the other hand, will not incur a full shuffle and will try to combine partitions. This operation will shuffle your data into five partitions based on the destination country name, and then coalesce them (without a full shuffle):

- `df.repartition(5, col("DEST_COUNTRY_NAME")).coalesce(2)`

COALESCE

Used to **reduce** the number of partitions in a dataframe.

We cannot increase the number of partitions using coalesce.

Unlike repartition, coalesce doesn't perform a shuffle to create the partitions.

Suppose there are 100 partitions with 10 records in each partition, and if the partition size is reduced to 50, it would retain 50 partitions and append the other values to these existing partitions thereby having 20 records in each partition.

REPARTITION VS COALESCE

Repartition can be used under these scenarios:

- when you want your output partitions to be of equally distributed chunks.
- ***increase*** the number of partitions.
- perform a shuffle of the data and create partitions.

Coalesce can be used under the following scenarios:

- when you want to ***decrease*** the number of partitions.

ACCUMULATORS

Accumulators are used to write data across a cluster.

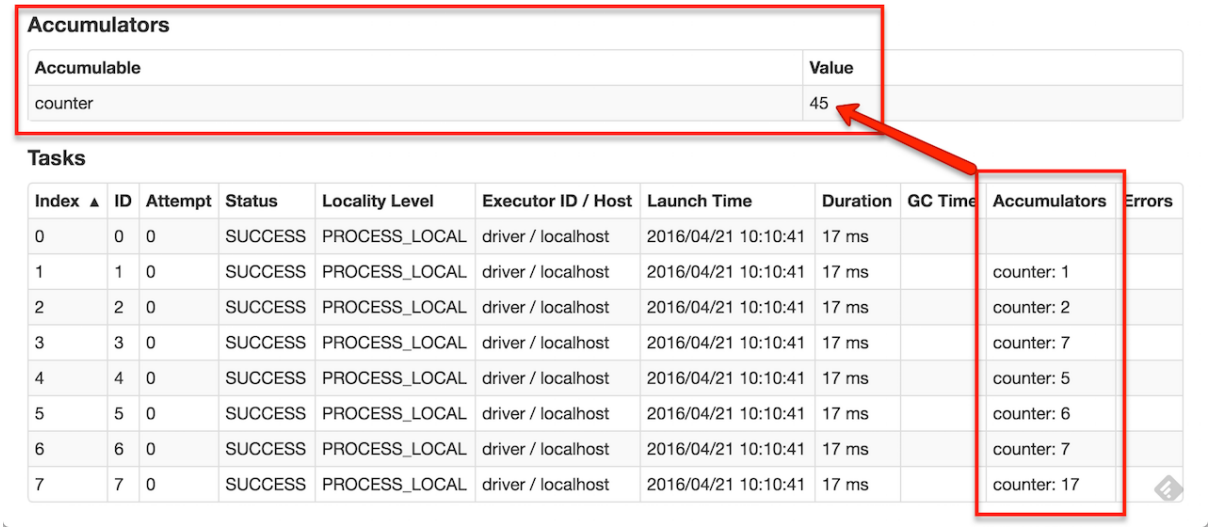
Examples:

- We are reading a file in chunks and we want to count the number of blank lines. We can use an accumulator to do this.

Accumulators are variables that are only “added” to through an associative operation and can therefore, be efficiently supported in parallel. They can be used to implement counters (as in MapReduce) or sums. Spark natively supports accumulators of numeric types, and programmers can add support for new types.

Accumulators do not change the lazy evaluation model of Spark. If they are being updated within an operation on an RDD, their value is only updated once that RDD is computed as part of an action. Consequently, accumulator updates are not guaranteed to be executed when made within a lazy transformation like map().

If accumulators are created with a name, **they will be displayed in Spark’s UI.** This can be useful for understanding the progress of running stages (**NOTE − this is not yet supported in Python**).



Accumulators

Accumulable	Value
counter	45

Tasks

Index ▲	ID	Attempt	Status	Locality Level	Executor ID / Host	Launch Time	Duration	GC Time	Accumulators	Errors
0	0	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms			
1	1	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 1	
2	2	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 2	
3	3	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 7	
4	4	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 5	
5	5	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 6	
6	6	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 7	
7	7	0	SUCCESS	PROCESS_LOCAL	driver / localhost	2016/04/21 10:10:41	17 ms		counter: 17	

An accumulator is created from an initial value v by calling `SparkContext.accumulator(v)`.

Tasks running on the cluster can then add to it using the add method or the += operator (in Scala and Python). However, they cannot read its value. Only the driver program can read the accumulator's value, using its value method.

Task failed due to some exception in code. Spark will try 4 times(default number of tries). If a task fails 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.

If a task is running slow then, Spark can launch a speculative copy of that task on another node. **Accumulator will give wrong output.**

RDD which is cached is huge and can't reside in Memory.So whenever the RDD is used it will re run the Map operation to get the RDD and again accumulator will be updated by it. **Accumulator will give wrong output.**

PERFORMANCE TUNING CONFIG OPTIONS

Performance Tuning considerations

- Overhead of garbage collection
- Amount of memory used by objects
- Cost of accessing these objects

IMPROVE SPARK PERFORMANCE

- Increase **spark.default.parallelism** and **spark.sql.shuffle.partitions** to improve spark performance;

- As the shuffle operations re-partitions the data, we can use configurations spark.default.parallelism and spark.sql.shuffle.partitions **to control** the number of partitions shuffle creates;
- **spark.default.parallelism**
    - was introduced with RDD hence this property is only applicable to RDD;
    - For distributed shuffle operations like reduceByKey and join, the largest number of partitions in a parent RDD. For operations like parallelize with no parent RDDs, it depends on the cluster manager:
        - **Local mode:** number of cores on the local machine
        - **Mesos fine grained mode:** 8
        - **Others:** total number of cores on all executor nodes or 2, whichever is larger
- **spark.sql.shuffle.partitions**
    - was introduced with DataFrame and it only works with DataFrame, the default value for this configuration set to 200;
    - The default number of partitions to use when shuffling data for joins or aggregations. Note: For structured streaming, this configuration cannot be changed between query restarts from the same checkpoint location.

Property	**Definition**	**Default**
spark.sql.inMemoryColumnar Storage.compressed	When set to true, Spark SQL will automatically select a compression codec for each column based on statistics of the data.	true
spark.sql.inMemoryColumnar Storage.batchSize	Controls the size of batches for columnar caching. Larger batch sizes can improve memory utilization and compression, but risk OOMs when caching data.	10000
spark.sql.files.maxPartitionBytes	The maximum number of bytes to pack into a single partition when reading files. This configuration is effective only when using file-based sources such as Parquet, JSON and ORC.	134217728 (128 MB)

spark.sql.files.openCostInBytes	The estimated cost to open a file, measured by the number of bytes, could be scanned at the same time. This is used when putting multiple files into a partition. It is better to over-estimated, then the partitions with small files will be faster than partitions with bigger files (which is scheduled first). This configuration is effective only when using file-based sources such as Parquet, JSON and ORC.	4194304 (4 MB)
spark.sql.files.minPartitionNum	The suggested (not guaranteed) minimum number of split file partitions. If not set, the default value is `spark.default.parallelism`. This configuration is effective only when using file-based sources such as Parquet, JSON an ORC.	Default Parallelism
spark.sql.broadcastTimeout	Timeout in seconds for the broadcast wait time in broadcast joins	300
spark.sql.autoBroadcastJoinThreshold	Configures the maximum size in bytes for a table that will be broadcast to all worker nodes **when performing a join.** By setting this value to -1 broadcasting can be disabled. Note that currently statistics are only supported for Hive Metastore tables where the command ANALYZE TABLE <tableName> COMPUTE STATISTICS noscan has been run.	10485760 (10 MB)
spark.sql.shuffle.partitions	Configures the number of partitions to use when shuffling data for joins or aggregations	200

spark.sql.sources.parallelPartitionDiscovery.threshold	Configures the threshold to enable parallel listing for job input paths. If the number of input paths is larger than this threshold, Spark will list the files by using a Spark distributed job. Otherwise, it will fallback to sequential listing. This configuration is only effective when using file-based data sources such as Parquet, ORC and JSON.	32
spark.sql.sources.parallelPartitionDiscovery.parallelism	Configures the maximum listing parallelism for job input paths. In case the number of input paths is larger than this value, it will be throttled down to use this value. Effective when using file-based data sources such as Parquet, ORC and JSON.	10000
spark.dynamicAllocation.enabled	Whether to use dynamic resource allocation, which scales the number of executors registered with this application up and down based on the workload. This requires **spark.shuffle.service.enabled** or **spark.dynamicAllocation.shuffleTracking.enabled** to be set. The following configurations are also relevant: spark.dynamicAllocation.minExecutors spark.dynamicAllocation.maxExecutors spark.dynamicAllocation.initialExecutors spark.dynamicAllocation.executorAllocationRatio	false

TUNING IN SPARK

NEED OF TUNING

- Since spark is of in-memory nature we can have bottlenecks due to resource, memory or network bandwidth.

AREAS OF TUNING

- Data serialization - crucial for good network performance and reduce memory usage.
- Memory tuning

DATA SERIALIZATION

- Covered earlier

MEMORY TUNING

- Java objects occupy a lot of memory so we need to mind memory usage and also use appropriate objects.

MEMORY MANAGEMENT IN SPARK

- Spark has two categories of memory usage:
    - Storage
        - used for storing all of the cached data, broadcast variables are also stored here. Any persist option which includes MEMORY in it, spark will store that data in this segment. Spark clears space for new cache requests by removing old cached objects based on Least Recently Used (LRU) mechanism. Once the cached data it is out of storage, it is either written to disk or recomputed based on configuration. Broadcast variables are stored in this segment with MEMORY_AND_DISK persistent level.
        - Formula:
            - Storage Memory = (Java Heap — Reserved Memory) * spark.memory.fraction * spark.memory.storageFraction
        - Calculation for 4GB:
            - Storage Memory = (4096MB — 300MB) * 0.75 * 0.5 = ~1423MB
    - Execution
        - is used by Spark for objects created during execution of a task. For example, it is used to store hash table for hash aggregation step, it is used to store shuffle intermediate buffer on the Map side in memory etc,. This pool also supports spilling on disk if not enough memory is available, but the blocks from this pool cannot be forcefully evicted by other tasks.
        - Formula:
            - Execution Memory = (Java Heap — Reserved Memory) * spark.memory.fraction * (1.0 — spark.memory.storageFraction)
        - Calculation for 4GB:
            - Execution Memory = (4096MB — 300MB) * 0.75 * (1.0 — 0.5) = ~1423MB

- Execution refers to memory used for computation in shuffles, joins, sorts, and aggregations

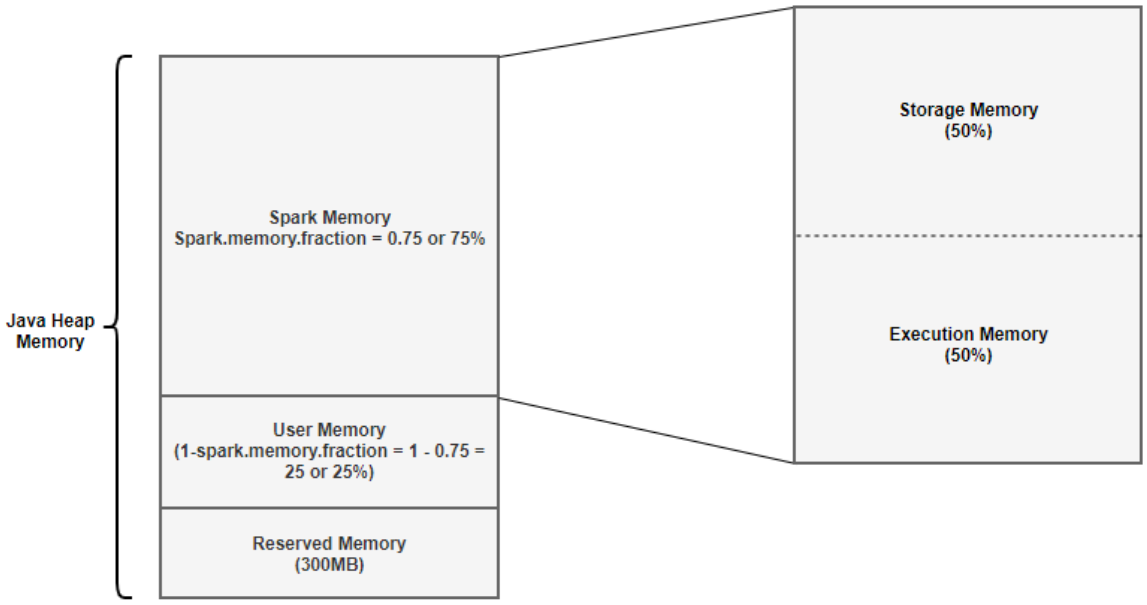
- Storage is memory used for caching partitions of data and propagating internal data structures.

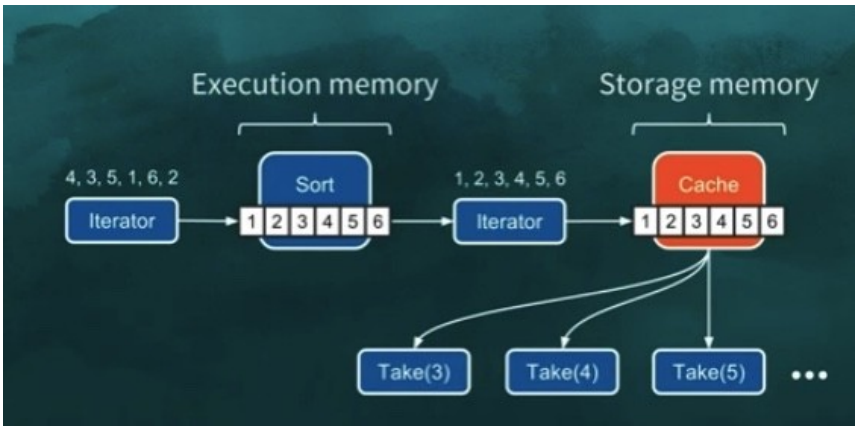
- When no execution is being done, storage can occupy all the memory.

- There is a subregion in memory which is always reserved for storage.

- **spark.memory.fraction** refers to the size of memory M
    - percentage of memory used for computation in shuffles, joins, sort and aggregations. If gc is invoked multiple times before task completed, **decrease** this so that amount of memory used for caching is reduced

- **spark.memory.storageFraction** refers to size of storage fraction as a fraction of M





More details:

- [Apache Spark Memory Management. This blog describes the concepts behind… | by Suhas NM | Analytics Vidhya]
- [Apache Spark Memory Management: Deep Dive]

SLOW READS AND WRITES

Slow I/O can be difficult to diagnose, especially with networked file systems.

Signs and symptoms

- Slow reading of data from a distributed file system or external system.
- Slow writes from network file systems or blob storage.

Potential treatments

- Turning on speculation (set **spark.speculation** to true)
    - relaunch one or more tasks if they are running slowly in a stage
    - However, it can cause duplicate data writes with some eventually consistent cloud services, such as Amazon S3, so check whether it is supported by the storage system connector you are using

DETERMINING MEMORY USAGE

- Put rdd in the cache and look at the storage tab in spark UI.
- To estimate the memory of a particular object, use sizeEstimator's estimate method.

MEASURING IMPACT OF GC

- This can be done by adding
    - `-verbose:gc`
    - `-XX:+PrintGCDetails`
    - `-XX:+PrintGCTimeStamps` to the Java options

COMPONENTS OF MEMORY IN JVM

- Java Heap space is divided into two regions: **Young and Old**. The Young generation is meant to hold short-lived objects while the Old generation is intended for objects with longer lifetimes.

- The Young generation is further divided into three regions [Eden, Survivor1, Survivor2].

- A simplified description of the garbage collection procedure:

    - When Eden is full, a minor GC is run on Eden and objects that are alive from Eden and Survivor1 are copied to Survivor2.

    - The Survivor regions are swapped. If an object is old enough or Survivor2 is full, it is moved to Old. Finally, when Old is close to full, a full GC is invoked.

GOAL OF GC

- The goal of GC tuning in Spark is to ensure that only long-lived RDDs are stored in the Old generation and that the Young generation is sufficiently sized to store short-lived objects.

STEPS TO AVOID FULL GC

- Check if there are too many garbage collections by collecting GC stats. If a full GC is invoked multiple times before a task completes, it means that there isn't enough memory available for executing tasks.

- If there are too many minor collections but not many major GCs, allocating more memory for Eden would help.

- In the GC stats that are printed, if the OldGen is close to being full, reduce the amount of memory used for caching by lowering spark.memory.fraction;

DATA LOCALITY

- Data locality is how close data is to the processing it. This affects speed of execution.

- The levels of data locality are:

    - PROCESS_LOCAL data is in the same JVM as the running code. This is the best locality possible

    - NODE_LOCAL data is on the same node. Examples might be in HDFS on the same node, or in another executor on the same node. This is a little slower than PROCESS_LOCAL because the data has to travel between processes

    - NO_PREF data is accessed equally quickly from anywhere and has no locality preference

    - RACK_LOCAL data is on the same rack of servers. Data is on a different server on the same rack so needs to be sent over the network, typically through a single switch ANY data is elsewhere on the network and not in the same rack

DATAFRAME

READ/WRITE

GENERIC FILE OPTIONS

- Ignore Corrupt Files
- Ignore Missing Files
- pathGlobFilter: include files with file names matching the pattern.

- recursiveFileLookup: used to recursively load files and it disables partition inferring. Its default value is false

Examples:

- `spark.sql("set spark.sql.files.ignoreCorruptFiles=true")`
- `spark.sql.files.ignoreMissingFiles`
- `spark.read.load("examples/src/main/resources/dir1", format="parquet", pathGlobFilter="*.parquet")`
- `recursive_loaded_df = spark.read.format("parquet").option("recursiveFileLookup", "true").load("examples/src/main/resources/dir1")`

READ

The read modes are:

- **permissive:** all fields are set to null and corrupted records are placed in a string column called ***_corrupt_record***
- **dropMalformed:** drops all rows containing corrupt records.
- **failFast:** fails when corrupt records are encountered.

If you want to control the maximum partition size while reading files:
spark.files.maxpartitionBytes

Syntax: `DataFrameReader.format(…).option("key", "value").schema(…).load()`

Examples:

- `df = spark.read.parquet('python/test_support/sql/file.parquet')`
- `df = spark.read.format('json').load('python/test_support/sql/people.json')`
- `df = spark.read.json('python/test_support/sql/people.json')`

WRITE

DataFrame writer default format is the parquet file

Save modes are:

- append: appends output data to files that already exist
- overwrite: completely overwrites any data present at the destination
- errorIfExists: Spark throws an error if data already exists at the destination
- ignore: if data exists do nothing with the dataFrame

Syntax:

```
DataFrameWriter.format(...).option(...).partitionBy(...).bucketBy(...)
.sortBy( ...).save()
```

Note: bucketBy is applicable for file-based data sources in combination with DataFrameWriter.saveAsTable().

Examples:

- `csvFile.write.format("csv").mode("overwrite").option("sep", "\t")`
  `.save("/tmp/my-tsv-file.tsv")`

CSV

- Read:
  - ```
    csvFile = spark.read.format("json").option("mode", "FAILFAST")
    .option("inferSchema","true").load("/data/flight-data/json/2010-summar
    y.js on").show(5)
    ```
- Write:
  - ```
    csvFile.write.format("csv").mode("overwrite").option("sep", "\t")
    .save("/tmp/my-tsv-file.tsv")
    ```

JSON

Spark SQL can automatically infer the schema of a JSON dataset and load it as a Dataset[Row].

Each line must contain a separate, self-contained valid JSON object

For a regular multi-line JSON file, set the multiLine option to true.

Default date format in json is **yyyy-MM-dd**

- Read:
  - ```
    spark.read.format("json") .option("mode", "FAILFAST")
    .option("inferSchema", "true")
    .load("/data/flight-data/json/2010-summary.js on").show(5)
    ```
- Write:
  - ```
    csvFile.write.format("json").mode("overwrite")
    .save("/tmp/my-json-file.json")
    ```

TEXT

When you write a text file, you need to be sure to have only one string column; otherwise, the write will fail

- Read:
  - ```
    lines = sc.textFile("/home/deepak/test1.txt")
    ```
- Write:
  - ```
    lines.coalesce(1).write.format("text")
    .option("header","false").mode("append").save("output.txt")
    ```

PARQUET

Parquet is a columnar format that is supported by many other data processing systems

When reading Parquet files, all columns are automatically converted to be nullable for compatibility reasons.

- Read:
  - ```
    spark.read.format("parquet")
    .load("/data/flight-data/parquet/summary.parq uet") .show(5)
    ```
- Write:
  - ```
    csvFile.write.format("parquet").mode("overwrite")
    .save("/tmp/my-parquet-file.parquet")
    ```

ORC

- Read:
  - `spark.read.format("orc").load("/data/flight-data/orc/2010-summary.orc").show(5)`
- Write:
  - `csvFile.write.format("orc").mode("overwrite").save("/tmp/my-json-file.orc")`

DATABASE

driver = "org.sqlite.JDBC" path = "/data/flight-data/jdbc/my-sqlite.db" url = "jdbc:sqlite:" + path tablename = "flight_info"

- Read:
  - `dbDataFrame = spark.read.format("jdbc") .option("url", url).option("dbtable", tablename) .option("driver", driver).load()`
- Write:
  - `newPath = "jdbc:sqlite://tmp/my-sqlite.db" csvFile.write.jdbc(newPath, tablename, mode="overwrite", properties=props)`

FUNCTIONS

CREATEDATAFRAME

Creates a DataFrame from an RDD, a list or a pandas.DataFrame

Package: pyspark.sql.sparkSession

Syntax:

```
createDataFrame(data, schema=None, samplingRatio=None, verifySchema=True)
```

Parameters:

- data:
  - type: dataRDD or iterable
  - def: an RDD of any kind of SQL data representation
- schema:
  - type: pyspark.sql.types.DataType, str or list, optional
  - def:
- samplingRatio:
  - type: float, optional
  - def: the sample ratio of rows used for inferring
- verifySchema:
  - type: bool, optional
  - def: verify data types of every row against schema. Enabled by default.

PRINTSCHEMA

Prints out the schema in the tree format.

Package: **pyspark.sql.DataFrame**

Syntax: `DataFrame.printSchema()`

Example:

- `df.printSchema()`

CREATEORREPLACETEMPVIEW

Creates or replaces a local temporary view with this DataFrame.

Package: **pyspark.sql.DataFrame**

Syntax: `DataFrame.createOrReplaceTempView(name)`

Example: `df.createOrReplaceTempView("people")`

SCHEMA

Returns the schema of this DataFrame as a pyspark.sql.types.StructType.

Package: **pyspark.sql.DataFrame**

Syntax: `DataFrame.schema`

Example: `df.schema`

DROP

Returns a new DataFrame that drops the specified column.

This is a no-op if the schema doesn't contain the given column name(s).

Package: **pyspark.sql. DataFrame**

Syntax: `DataFrame.drop(*cols)`

- cols:
    - type: str or :class:`Column`
    - def: a name of the column, or the Column to drop

Example:

- `df.drop('age').collect()`
- `df.drop(df.age).collect()`
- `df.join(df2, df.name == df2.name, 'inner').drop(df.name).collect()`
- `df.join(df2, df.name == df2.name, 'inner').drop(df2.name).collect()`
- `df.join(df2, 'name', 'inner').drop('age', 'height').collect()`

DROPDUPLICATES

Return a new DataFrame with duplicate rows removed, optionally only considering certain columns.

- For a static batch DataFrame, it just drops duplicate rows.
- For a streaming DataFrame, it will keep all data across triggers as an intermediate state to drop duplicate rows.

You can use *withWatermark()* to limit how late the duplicate data can be and the system will accordingly limit the state.

drop_duplicates() is an alias for dropDuplicates()

Package: **pyspark.sql.DataFrame**

Syntax: `DataFrame.dropDuplicates(subset=None)`

Example:

- `df.dropDuplicates(['name', 'height']).show()`

DROPNA

Returns a new DataFrame omitting rows with null values.

DataFrame.dropna( ) and DataFrameNaFunctions.drop() are aliases of each other.

Package: **pyspark.sql.DataFrame**

Syntax: `dataframe.dropna(how='any', thresh=None, subset=None)`

- how
  - type: str, optional
  - def: ‘any’ or ‘all’.
    - If ‘any’, drop a row if it contains any nulls.
    - If ‘all’, drop a row only if all its values are null.
- thresh:
  - type: int, optional
  - def: default None
    - If specified, drop rows that have less than thresh non-null values.
    - This overwrites the how parameter.

        - thresh=2 means that if there are any **rows** or **columns** which is having fewer than non-NULL values than thresh values then we are dropping that row or column from the Dataframe.
- subset:
    - type:str, tuple or list, optional
    - def: optional list of column names to consider.

Example:

- `df.na.drop().show()`
- `df.na.drop("any").show()`
- `df.na.drop("all").show()`
- `df.na.drop(subset=["population","type"]).show()`
- `df.dropna().show()`
- `df = df.dropna(thresh=2)`

FILLNA

DataFrame.fillna(value, subset=None)

Replace null values, alias for na.fill().

Package: **pyspark.sql.DataFrame**

Syntax: `dataframe.fillna(value, subset=None)`

Parameters:

- value: int, float, string, bool or dict
    - Value to replace null values with. If the value is a dict, then subset is ignored and value must be a mapping from column name (string) to replacement value. The replacement value must be an int, float, boolean, or string.

- subset: str, tuple or list, optional
    - optional list of column names to consider. Columns specified in subset that do not have matching data type are ignored. For example, if value is a string, and subset contains a non-string column, then the non-string column is simply ignored.

Examples:

- `df4.na.fill(50).show()`
- `df5.na.fill(False).show()`
- `df4.na.fill({'age': 50, 'name': 'unknown'}).show()`

Using the fill function, you can fill one or more columns with a set of values. This can be done by specifying a map - that is a particular value and a set of columns.

For example, to fill all null values in columns of type String, you might specify the following:

```
df.na.fill("All Null values become this string")
```

We could do the same for columns of type Integer by using `df.na.fill(5:Integer)`, or for Doubles `df.na.fill(5:Double)`. To specify columns, we just pass in an array of column names like we did in the previous example:

```
df.na.fill("all", subset=["StockCode", "InvoiceNo"])
```

We can also do this with with a Scala Map, where the key is the column name and the value is the value we would like to use to fill null values:

```
fill_cols_vals = {"StockCode": 5, "Description" : "No Value"}
df.na.fill(fill_cols_vals)
```

DISTINCT

Returns a new DataFrame containing the distinct rows in this DataFrame.

Package: **pyspark.sql.DataFrame**

Syntax: `DataFrame.distinct()`

Example:

- `df.distinct().count()`

SPLIT

Splits str around matches of the given pattern.

Package: **pyspark.sql. functions**

Syntax: `split(str, pattern, limit=- 1)`

- str
  - type: Column or str
  - def: a string expression to split
- pattern
  - type: str
  - def: a string representing a regular expression. The regex string should be a Java regular expression.
- limit
  - type: int, optional
  - def: an integer which controls the number of times a pattern is applied.
  - **limit > 0:** The resulting array's length will not be more than limit, and the resulting array's last entry will contain all input beyond the last matched pattern.
  - **limit <= 0:** pattern will be applied as many times as possible, and the resulting array can be of any size.

Example:

- `df.select(split(df.s, '[ABC]', -1).alias('s')).collect()`

ALIAS

Returns this column aliased with a new name or names.

In the case of expressions that return more than one column, such as explode).

Package: **pyspark.sql.column**

Syntax: `column.alias(*alias, **kwargs)`

- alias type: str def: desired column names (collects all positional arguments passed)

Example:

- `df.select(df.age.alias("age2")).collect()`

EXPLODE

Returns a new row for each element in the given array or map.

Uses the default column name for elements in the array and key and value for elements in the map unless specified otherwise.

Package: **pyspark.sql.functions**

Syntax: `explode(e: Column)`

Example:

- `df = spark.createDataFrame([Row(a=1, intlist=[1,2,3], mapfield={"a": "b"})])`
- `df.select(explode(df.intlist).alias("anInt")).collect()`

WHERE / FILTER

Filters rows using the given condition.

where() is an alias for filter().

Package: **pyspark.sql.DataFrame**

Syntax: `DataFrame.filter(condition)`

- Condition:
    - type: column or str

    - def: a column of types.BooleanType or a string of SQL expression.

Example:

- `df.filter(df.age > 3).collect()`
- `df.where(df.age == 2).collect()`
- `df.filter("age > 3").collect()`
- `df.where("age = 2").collect()`

COLLECT

Returns all the records as a list of Row.

Package: **pyspark.sql.DataFrame**

Syntax: `DataFrame.collect()`

Example:

- `df.collect()`
    - `[Row(age=2, name='Alice'), Row(age=5, name='Bob')]`

TOLOCALITERATOR

Returns an iterator that contains all of the rows in this DataFrame. The iterator will consume as much memory as the largest partition in this DataFrame. With prefetch it may consume up to the memory of the 2 largest partitions.

Package: **pyspark.sql.DataFrame**

Syntax: `DataFrame.toLocalIterator(prefetchPartitions=False)`

- Parameters: prefetchPartitionsbool, optional
    - If Spark should pre-fetch the next partition before it is needed.

Example:

- `list(df.toLocalIterator())`
    - `[Row(age=2, name='Alice'), Row(age=5, name='Bob')]`

WITHCOLUMN

Returns a new DataFrame by adding a column or replacing the existing column that has the same name.

The column expression must be an expression over this DataFrame; attempting to add a column from some other DataFrame will raise an error.

Package: **pyspark.sql.DataFrame**

Syntax: `DataFrame.withColumn(colName, col)`

- colName:
  - Type:str
  - def: string, name of the **new column**.
- col:
  - type: column
- def: a column expression for the new column.

Example:

- `df.withColumn('age2', df.age + 2).collect()`

WITHCOLUMNRENAMED

Returns a new DataFrame by renaming an existing column. This is a no-op if schema doesn't contain the given column name.

Package: **pyspark.sql.DataFrame**

Syntax: `DataFrame.withColumnRenamed(existing, new)`

Parameters:

- **existing**: string, name of the existing column to rename.
- **new**: string, new name of the column.

Examples:

- `df.withColumnRenamed('age', 'age2').collect()`
  - `[Row(age2=2, name='Alice'), Row(age2=5, name='Bob')]`

ARRAY_CONTAINS

Collection function: returns null if the array is null, true if the array contains the given value, and false otherwise.

Package: **pyspark.sql.Functions**

Syntax: `pyspark.sql.functions.array_contains(col, value)`

Parameters:

- colColumn or str: name of column containing array
- value: value or column to check for in array

Examples:

- `df = spark.createDataFrame([(["a", "b", "c"],), ([],)], ['data'])`
- `df.select(array_contains(df.data, "a")).collect()`
  - `[Row(array_contains(data, a)=True), Row(array_contains(data, a)=False)]`

- `df.select(array_contains(df.data, lit("a"))).collect()`
    - `[Row(array_contains(data, a)=True), Row(array_contains(data, a)=False)]`

SORT / ORDERBY

Returns a new DataFrame sorted by the specified column(s).

Package: **pyspark.sql.DataFrame**

Syntax: `DataFrame.sort(*cols, **kwargs)`

Parameters:

- cols: str, list, or Column, **optional**
- list of column or column names to sort by.
- ascending: bool or list, **optional**
    - boolean or list of boolean (default True).
    - Sort ascending vs. descending.
    - Specify list for multiple sort orders.
    - If a list is specified, length of the list must equal length of the cols.

Examples:

- `df.sort(df.age.desc()).collect()`
    - `[Row(age=5, name='Bob'), Row(age=2, name='Alice')]`
- `df.sort("age", ascending=False).collect()`
    - `[Row(age=5, name='Bob'), Row(age=2, name='Alice')]`
- `df.orderBy(df.age.desc()).collect()`
    - `[Row(age=5, name='Bob'), Row(age=2, name='Alice')]`
- `df.sort(asc("age")).collect()`
    - `[Row(age=2, name='Alice'), Row(age=5, name='Bob')]`
- `df.orderBy(desc("age"), "name").collect()`
    - `[Row(age=5, name='Bob'), Row(age=2, name='Alice')]`
- `df.orderBy(["age", "name"], ascending=[0, 1]).collect()`
    - `[Row(age=5, name='Bob'), Row(age=2, name='Alice')]`

SORT_ARRAY

Collection function: sorts the input array in ascending or descending order according to the natural ordering of the array elements. Null elements will be placed at the beginning of the returned array in ascending order or at the end of the returned array in descending order.

Package: **pyspark.sql.functions**

Syntax: `sort_array(col, asc=True)`

Parameters:

- col: column or str name of column or expression
- asc bool, optional

Examples:

- `df = spark.createDataFrame([([2, 1, None, 3],),([1],),([],)], ['data'])`
- `df.select(sort_array(df.data).alias('r')).collect()`
    - `[Row(r=[None, 1, 2, 3]), Row(r=[1]), Row(r=[])]`
- `df.select(sort_array(df.data, asc=False).alias('r')).collect()`
    - `[Row(r=[3, 2, 1, None]), Row(r=[1]), Row(r=[])]`

ASC_NULLS_FIRST

Returns a sort expression based on the ascending order of the column, and **null values return before** non-null values.

Package: **pyspark.sql.Column**

Syntax: `Column.asc_nulls_first()`

Example:

- `df.select(df.name).orderBy(df.name.asc_nulls_first()).collect()`

DESC_NULLS_FIRST

Returns a sort expression based on the descending order of the column, and **null values appear before** non-null values.

Package: **pyspark.sql.Column**

Syntax: `Column.desc_nulls_first()`

Example:

- `df.select(df.name).orderBy(df.name.desc_nulls_first()).collect()`

ASC_NULLS_LAST

Returns a sort expression based on the ascending order of the column, and **null values appear afte**r non-null values.

Package: **pyspark.sql. Column**

Syntax: `Column.asc_nulls_last()`

Example:
- `df.select(df.name).orderBy(df.name.asc_nulls_last()).collect()`

DESC_NULLS_LAST

Returns a sort expression based on the descending order of the column, and **null values appear after** non-null values.

Package: **pyspark.sql. Column**

Syntax: `Column.desc_nulls_last()`

Example:
```
df.select(df.name).orderBy(df.name.desc_nulls_last()).collect()
df.orderBy(col("column").desc_nulls_last())
```

BETWEEN

A boolean expression that is evaluated to true if the value of this expression is between the given columns.

Package: **pyspark.sql.Column**

Syntax: `Column.between(lowerBound, upperBound)`

Examples

```
df.select(df.name, df.age.between(2, 4)).show()
```

name	**((age >= 2) AND (age <= 4))**
Alice	true
Bob	false

SIZE

Collection function: returns the length of the array or map stored in the column.

Syntax: **pyspark.sql.functions.size(col)**

Parameters:

- col: Column or str - name of column or expression

Examples:

- `df = spark.createDataFrame([([1, 2, 3],),([1],),([],)], ['data'])`
- `df.select(size(df.data)).collect()`
    - `[Row(size(data)=3), Row(size(data)=1), Row(size(data)=0)]`

SAMPLE

Returns a sampled subset of this DataFrame.

Package: **pyspark.sql.DataFrame**

Syntax: `DataFrame.sample(withReplacement=None, fraction=None, seed=None)`

Parameters:

- **withReplacement**
    - type: bool, ***optional***
    - def: sample with replacement or not (**default False**).
- **fraction**
    - type: float, ***optional***
    - def: Fraction of rows to generate, range [0.0, 1.0].
    - *note that it doesn't guarantee to provide the exact number of the fraction of records.*
- **seed**
    - type: int, ***optional***
    - def: seed for sampling (default a random seed).

Examples:

- `df.sample(withReplacement=True, fraction=0.5, seed=3).count()`
- `df.sample(False, fraction=1.0)`

ISIN

A boolean expression that is evaluated to be true if the value of this expression is contained by the evaluated values of the arguments.

Package: **pyspark.sql.Column**

Syntax: `Column.isin(*cols)`

Example:

- `df[df.name.isin("Bob", "Mike")].collect()`
- `df[df.age.isin([1, 2, 3])].collect()`

CONTAINS

Contains the other element. Returns a boolean Column based on a string match.

Package: **pyspark.sql.Column**

Syntax: `Column.contains(other)`

Parameters:

- other: string in line. A value as a literal or a Column.

Examples:

- `df.filter(df.name.contains('o')).collect()`
    - `[Row(age=5, name='Bob')]`

MONOTONICALLY_INCREASING_ID()

A column that generates monotonically increasing 64-bit integers. monotonically increasing and unique, but not consecutive.

```
df.withColumn("new_id", f.monotonically_increasing_id())
```

DATEDIFF

Returns number of data from start to end

Package: **pyspark.sql.functions**

Syntax: `datediff(end, start)`

Example:

```
df = spark.createDataFrame([('2015-04-0 8','2015-05-10')], ['d1', 'd2'])

df.select(datediff(df.d2, df.d1).alias('diff')).collect()
```

ARRAY FUNCTIONS

ARRAY FUNCTION SYNTAX	ARRAY FUNCTION DESCRIPTION

array_contains(column: Column, value: Any)	Check if a value presents in an array column. Return below values. true: Returns if value presents in an array. false: When value no presents. null: when the array is null.
array_distinct(e: Column)	Return distinct values from the array after removing duplicates.
array_except(col1: Column, col2: Column)	Returns all elements from col1 array but not in col2 array.
array_intersect(col1: Column, col2: Column)	Returns all elements that are present in col1 and col2 arrays.
array_join(column: Column, delimiter: String, nullReplacement: String) array_join(column: Column, delimiter: String)	Concatenates all elments of array column with using provided delimeter. When Null values are present, they replaced with 'nullReplacement' string
array_max(e: Column)	Return maximum values in an array
array_min(e: Column)	Return minimum values in an array
array_position(column: Column, value: Any)	Returns a position/index of first occurrence of the 'value' in the given array. Returns position as long type and the position is not zero based instead starts with 1. Returns zero when value is not found. Returns null when any of the arguments are null.
array_remove(column: Column, element: Any)	Returns an array after removing all provided 'value' from the given array.
array_repeat(e: Column, count: Int)	Creates an array containing the first argument repeated the number of times given by the second argument.
array_repeat(left: Column, right: Column)	Creates an array containing the first argument repeated the number of times given by the second argument.
array_sort(e: Column)	Returns the sorted array of the given input array. All null values are placed at the end of the array.
array_union(col1: Column, col2: Column)	Returns an array of elements that are present in both arrays (all elements from both arrays) without duplicates.

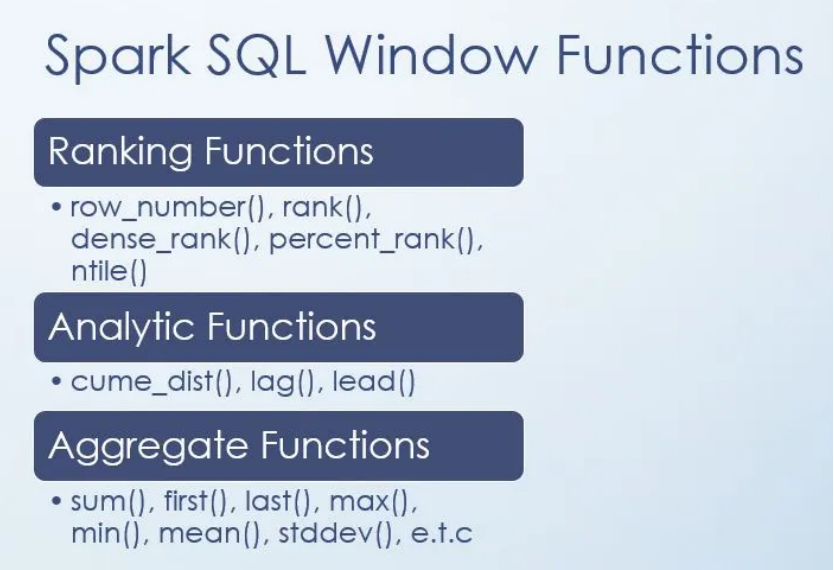
arrays_overlap(a1: Column, a2: Column)	true - if `a1` and `a2` have at least one non-null element in common false - if `a1` and `a2` have completely different elements. null - if both the arrays are non-empty and any of them contains a `null`
arrays_zip(e: Column*)	Returns a merged array of structs in which the N-th struct contains all N-th values of input
concat(exprs: Column*)	Concatenates all elements from a given columns
element_at(column: Column, value: Any)	Returns an element of an array located at the 'value' input position.
exists(column: Column, f: Column => Column)	Checks if the column presents in an array column.
explode(e: Column)	Create a row for each element in the array column
explode_outer (e : Column)	Create a row for each element in the array column. Unlike explode, if the array is null or empty, it returns null.
filter(column: Column, f: Column => Column) filter(column: Column, f: (Column, Column) => Column)	Returns an array of elements for which a predicate holds in a given array
flatten(e: Column)	Creates a single array from an array of arrays columns.
forall(column: Column, f: Column => Column)	Returns whether a predicate holds for every element in the array.
posexplode(e: Column)	Creates a row for each element in the array and creates two columns "pos' to hold the position of the array element and the 'col' to hold the actual array value.
posexplode_outer(e: Column)	Creates a row for each element in the array and create a two columns "pos' to hold the position of the array element and the 'col' to hold the actual array value. Unlike posexplode, if the array is null or empty, it returns null,null for pos and col columns.
reverse(e: Column)	Returns the array of elements in a reverse order.
sequence(start: Column, stop: Column)	Generate the sequence of numbers from start to stop number.
sequence (start : Column , stop : Column , step : Column)	Generate the sequence of numbers from start to stop number by incrementing with a given step value.

shuffle(e: Column)	Shuffle the given array
size(e: Column)	Return the length of an array.
slice(x: Column, start: Int, length: Int)	Returns an array of elements from position 'start' and the given length.
sort_array(e: Column)	Sorts the array in an ascending order. Null values are placed at the beginning.
sort_array(e: Column, asc: Boolean)	Sorts the array in an ascending or descending order based on the boolean parameter. For ascending, Null values are placed at the beginning. And for descending they are places at the end.
transform(column: Column, f: Column => Column) transform(column: Column, f: (Column, Column) => Column)	Returns an array of elments after applying transformation.
zip_with(left: Column, right: Column, f: (Column, Column) => Column)	Merges two input arrays.
aggregate(expr: Column, zero: Column, merge: (Column, Column) => Column, finish: Column => Column)	Aggregates

WINDOW FUNCTIONS

Spark Window functions operate on a group of rows (like frame, partition) and return a single value for every input row. Spark SQL supports three kinds of window functions:

- ranking functions
- analytic functions
- aggregate functions



The below table defines Ranking and Analytic functions and for aggregate functions, we can use any existing aggregate functions as a window function.

To perform an operation on a group first, we need to partition the data using `Window.partitionBy()` , and for row number and rank function we need to additionally order by on partition data using orderBy clause.

WINDOW FUNCTION USAGE & SYNTAX	**WINDOW FUNCTION DESCRIPTION**
row_number(): Column	Returns a sequential number starting from 1 within a window partition
rank(): Column	Returns the rank of rows within a window partition, with gaps.
percent_rank(): Column	Returns the percentile rank of rows within a window partition.
dense_rank(): Column	Returns the rank of rows within a window partition without any gaps. Whereas Rank() returns rank with gaps.
ntile(n: Int): Column	Returns the ntile id in a window partition
cume_dist(): Column	Returns the cumulative distribution of values within a window partition

lag(e: Column, offset: Int): Column lag(columnName: String, offset: Int): Column lag(columnName: String, offset: Int, defaultValue: Any): Column	returns the value that is `offset` rows before the current row, and `null` if there is less than `offset` rows before the current row.
lead(columnName: String, offset: Int): Column lead(columnName: String, offset: Int): Column lead(columnName: String, offset: Int, defaultValue: Any): Column	returns the value that is `offset` rows after the current row, and `null` if there is less than `offset` rows after the current row.

AGGREGATE FUNCTIONS

AGG

Compute aggregates and returns the result as a DataFrame.

The available aggregate functions can be:

- built-in aggregation functions, such as avg, max, min, sum, count
- group aggregate pandas UDFs, created with pyspark.sql.functions.pandas_udf()

Package: **pyspark.sql. GroupedData**

Syntax: `GroupedData.agg(*exprs)`

- exprs:
  - type: dict
  - def: a dict mapping from column name (string) to aggregate functions (string), or a list of columns.

Example:

- `gdf = df.groupBy(df.name)`
  - `.sorted(gdf.agg({"*":"count"}).collect())`
  - `.sorted(gdf.agg(F.min(df.age)).collect())`

COUNT

Counts the number of records for each group

package: **pyspark.sql.GroupedData**

Syntax: `GroupedData.count()`

Example:

- `sorted(df.groupBy(df.age).count ().collect())`

PIVOT

Pivots a column of the current DataFrame and performs the specified aggregation.

There are two versions of the pivot function:

- one that requires the caller to specify the list of distinct values to pivot on
- one that does not.

Package: **pyspark.sql.GroupedData**

Syntax: `GroupedData.pivot(pivot_col, values=None)`

- pivot_col
  - type: str
  - def: Name of the column to pivot.
- values:
  - def: List of values that will be translated to columns in the output DataFrame.

Examples:

- Compute the sum of earnings for each year by course with each course as a separate column:

```
df4.groupBy("year").pivot("course", ["dotNET", "Java"]).sum("earnings").collect()
```

- Or without specifying column values (less efficient)

```
df4.groupBy("year").pivot("course").sum("earnings").collect()
```

JOINS

JOIN

Joins with another DataFrame, using the given join expression.

Package: **pyspark.sql.DataFrame.join**

Syntax: `DataFrame.join(other, on=None, how=None)`

Parameters:

- other:
  - type: DataFrame
  - def: Right side of the join
- on:
  - type:str, list or Column, optional
  - def:

        - a string for the join column name, a list of column names, a join expression (Column), or a list of Columns.
        - If there is a string or a list of strings indicating the name of the join column(s), the column(s) must exist on both sides, and this performs an equi-join.
- how:
    - type: str, optional
    - def:
        - default **inner**.
        - Must be one of: **inner, cross, outer, full, fullouter, full_outer, left, leftouter, left_outer, right, rightouter, right_outer, semi, leftsemi, left_semi, anti, leftanti and left_anti.**

Examples:

- `df.join(df2, df.name == df2.name, 'outer').select(df.name, df2.height).sort(desc("name")).collect()`
- `df.join(df2, 'name', 'outer').select('name', 'height').sort(desc("name")).collect()`
- `cond = [df.name == df3.name, df.age == df3.age]`
- `df.join(df3, cond, 'outer').select(df.name, df3.age).collect()`
- `df.join(df2, 'name').select(df.name, df2.height).collect()`
- `df.join(df4, ['name', 'age']).select(df.name, df.age).collect()`

INNER JOIN

Spark Inner join is the default join and it's mostly used.

It is used to join two DataFrames/Datasets on key columns, and where keys don't match the rows get dropped from both datasets.

Example:

```
df.join(df2, df.name == df2.name, 'outer').select(df.name, df2.height)
```

REFERENCE FOR JOINS

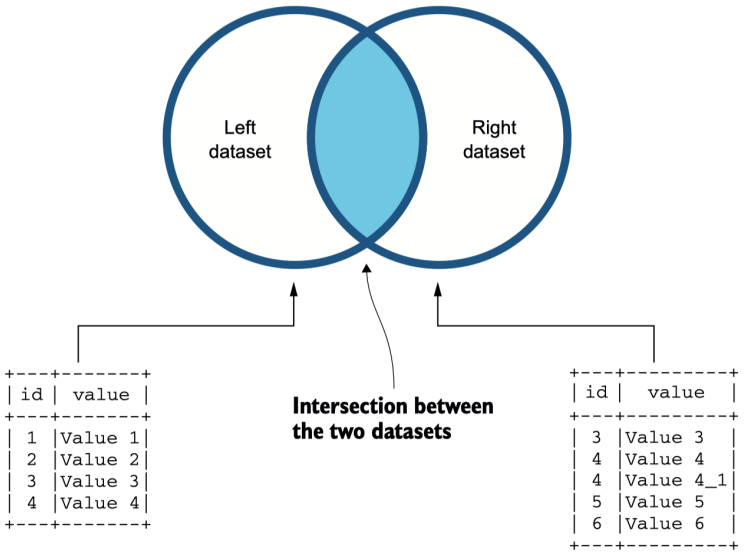
Dataset used for the left side:

Identifier	Value
1	Value 1
2	Value 2
3	Value 3
4	Value 4

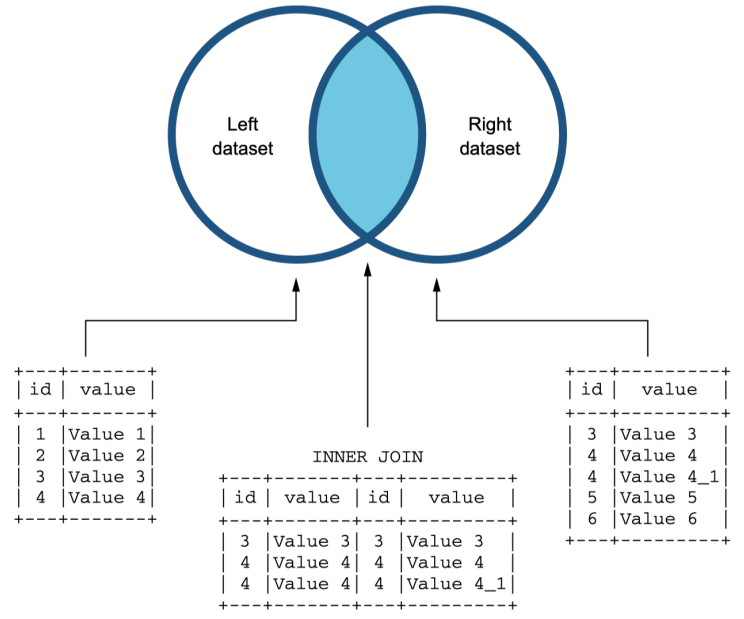
Dataset used for the right side:

Identifier	Value	Note
3	Value 3	
4	Value 3	The identifier is the same for both rows. Spark is not a relational database, where you can have unicity constraints on a column.
4	Value 4_1	
5	Value 5	
6	Value 6	

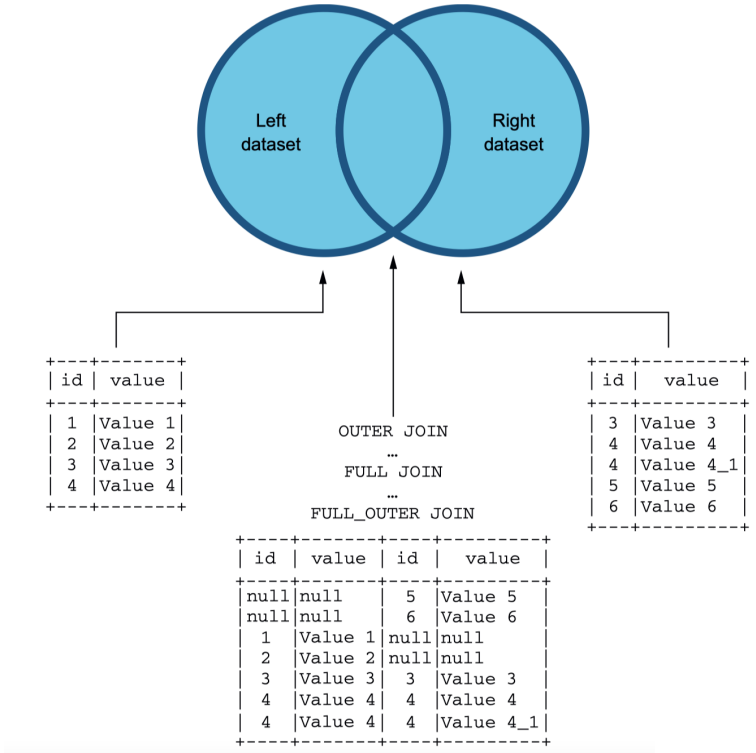
Illustration of a join operation between two datasets



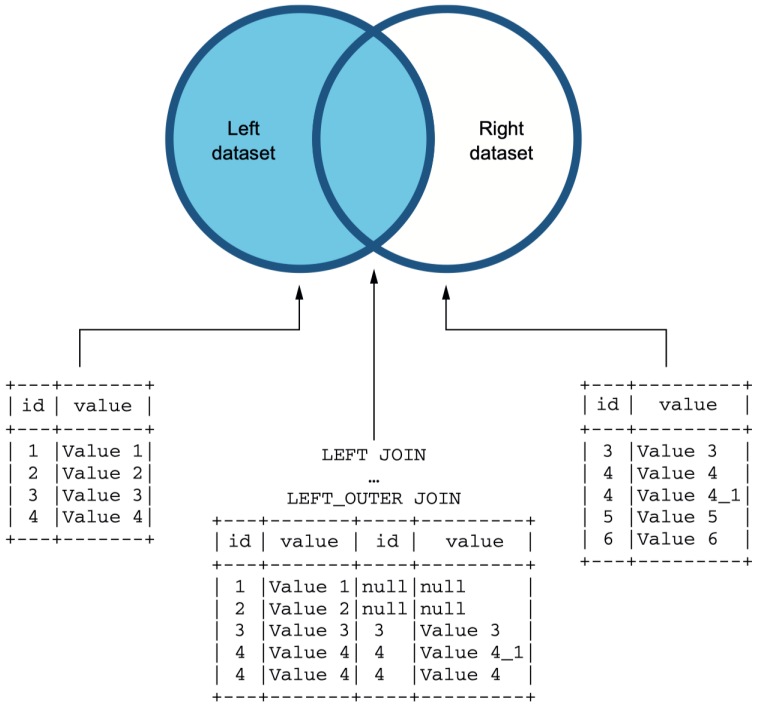
An INNER JOIN between two datasets on the id column



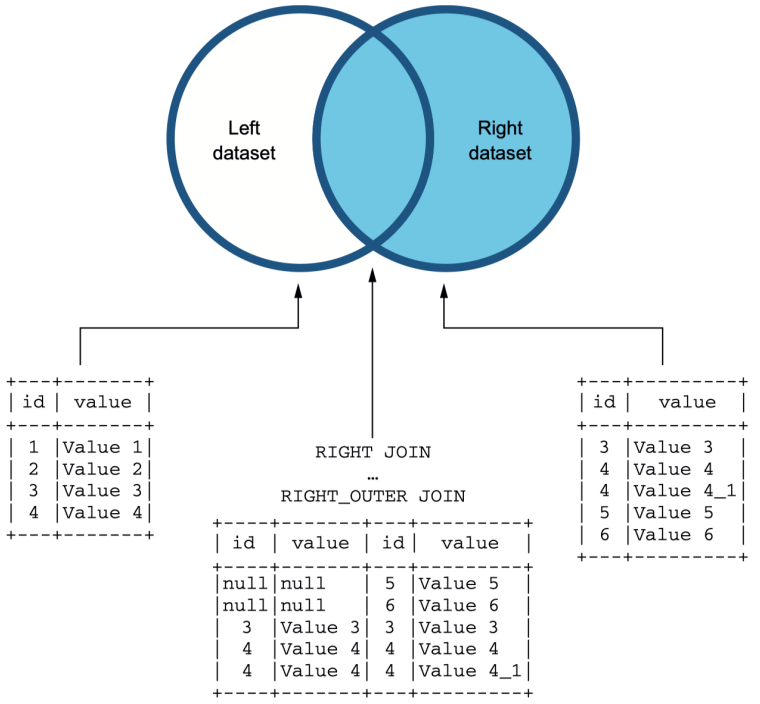
An OUTER, FULL, OR FULL_OUTER JOIN between two datasets on the id column



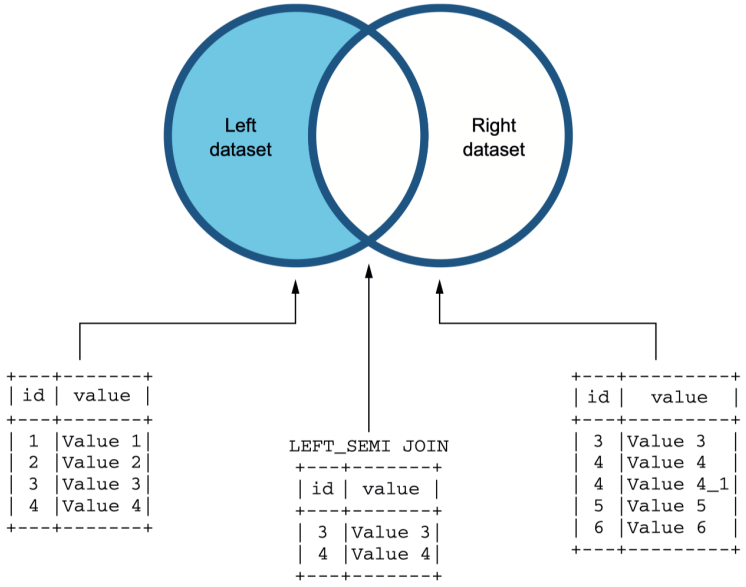
A LEFT, OR LEFT_OUTER JOIN, between two datasets on the id column



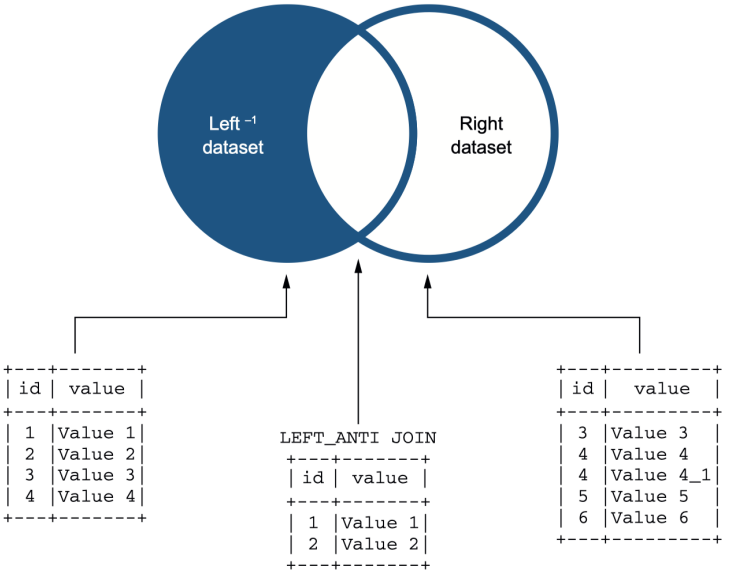
A RIGHT, OR RIGHT_OUTER JOIN between two datasets on the id column



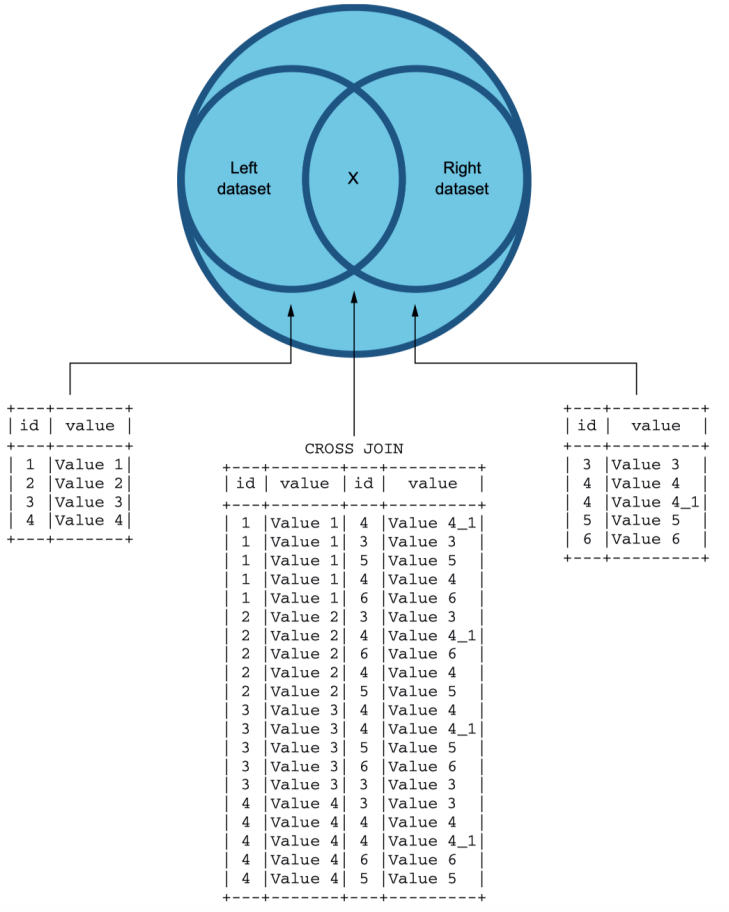
A LEFT_SEMI JOIN between two datasets on the id column



A LEFT_ANTI JOIN between two datasets on the id column



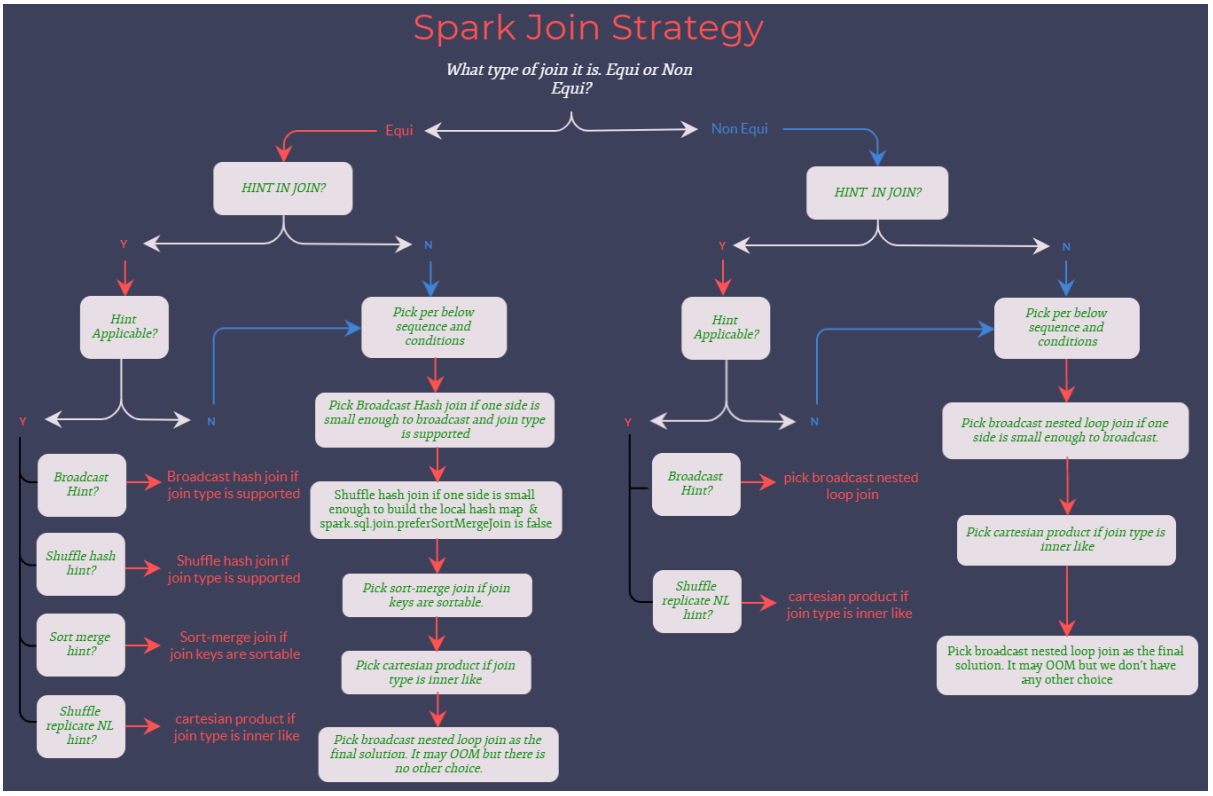
A CROSS_JOIN between two datasets on the id column



JOIN STRATEGIES

Spark selects Join strategy by considering the below:

- Type of Join
- Hint in Join



When the hints are specified on both sides of the Join, Spark selects the hint in the below order:

1. BROADCAST hint
2. MERGE hint
3. SHUFFLE_HASH hint
4. SHUFFLE_REPLICATE_NL hint
5. When BROADCAST hint or SHUFFLE_HASH hint are specified on both sides, Spark will pick up the build side based on the join type and the data size

The specified hint will not always be selected since a specific strategy may not support all the Join types.

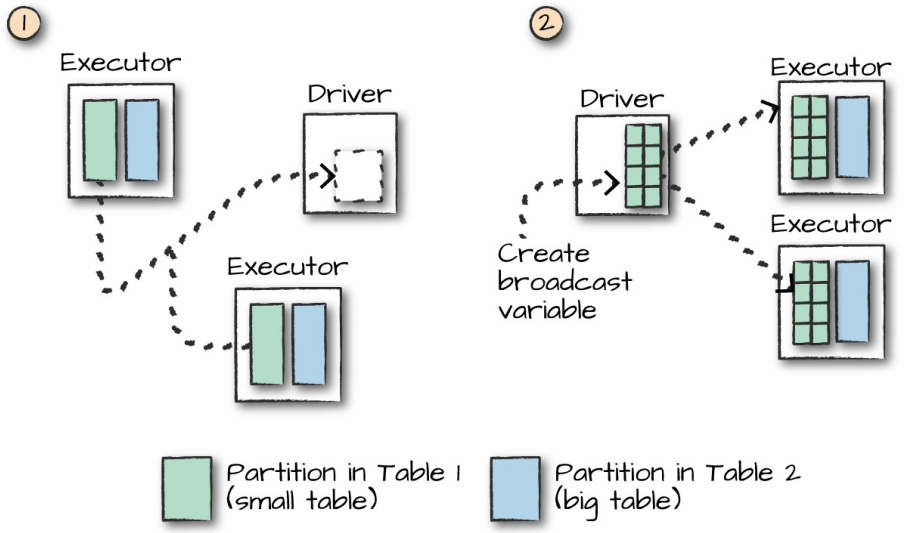
More details: https://blog.clairvoyantsoft.com/apache-spark-join-strategies-e4ebc7624b06

BROADCAST JOINS

Spark broadcast joins are perfect for joining a large DataFrame with a small DataFrame.

Broadcast joins cannot be used when joining two large DataFrames.

This post explains how to do a simple broadcast join and how the **broadcast()** function helps Spark optimize the execution plan.



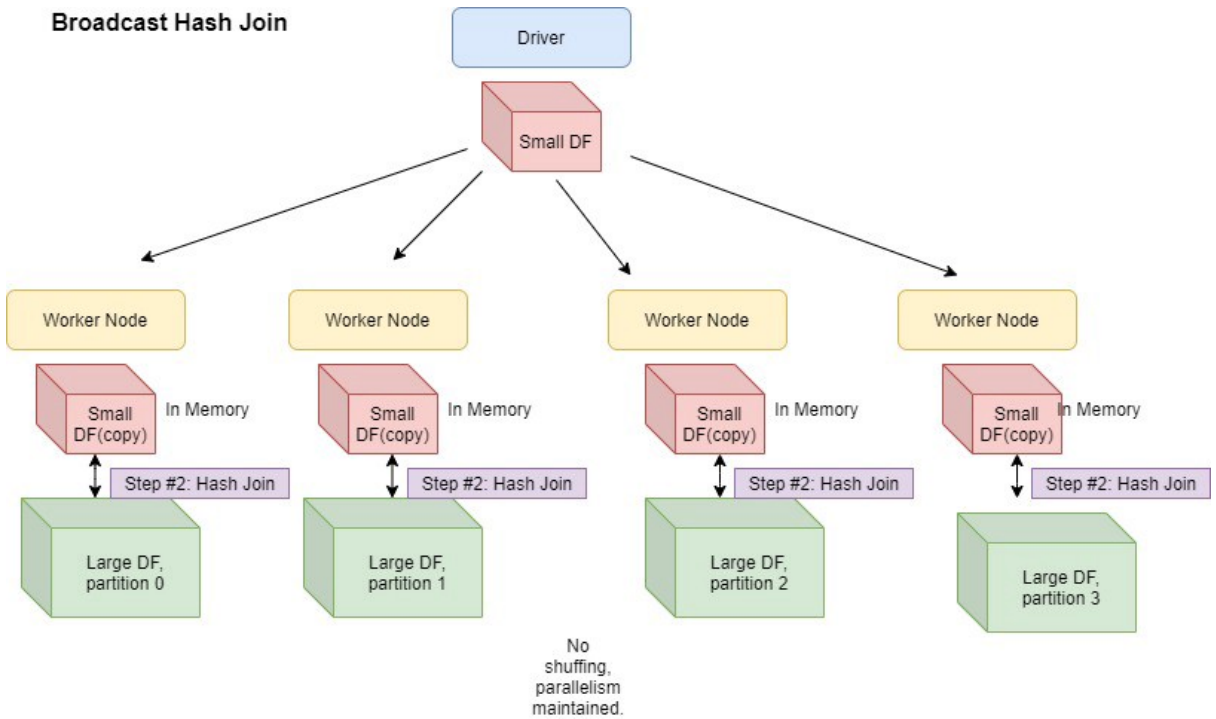
The algorithm chosen depends on the size of the tables joined and the hints provided in join

The commonly used joins in spark are:

- Broadcast Hash join
- Sort Merge Join
- Shuffle Hash Join

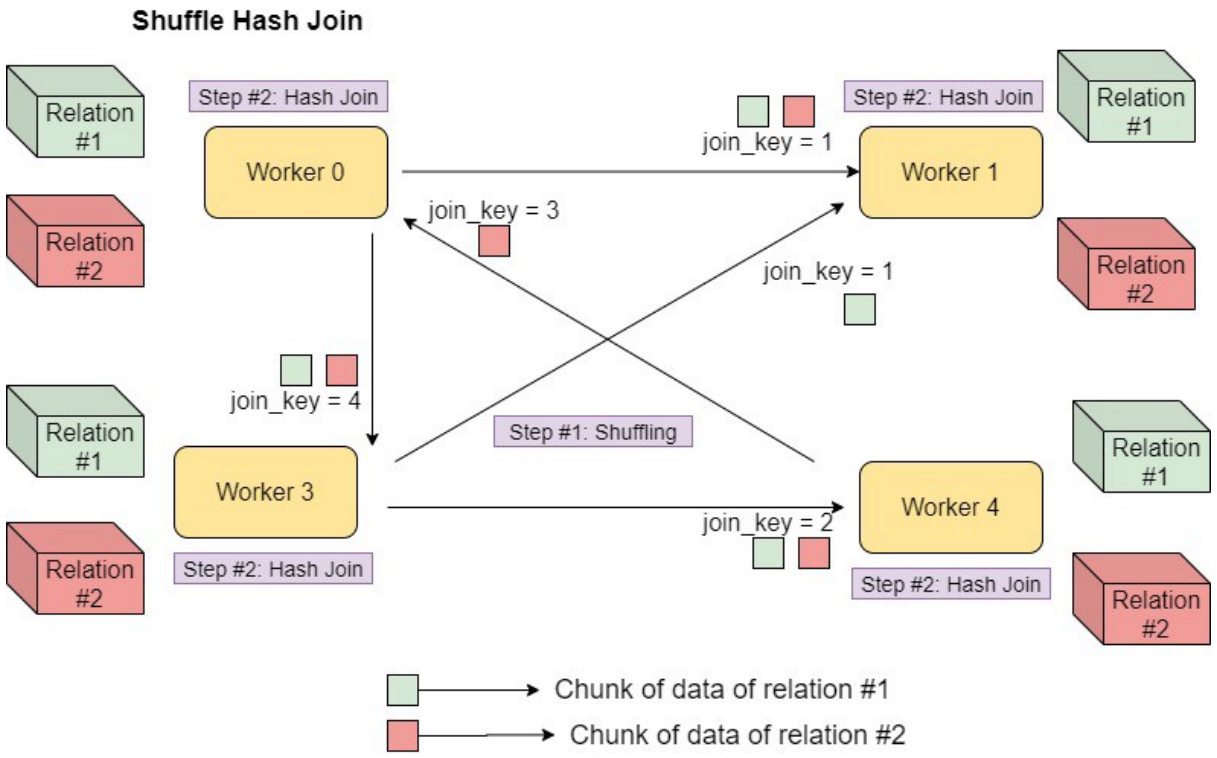
BROADCAST HASH JOIN

- Used when dataframe size is **less** than **spark.sql.autoBroadcastJoinThreshold**.
- Does not perform shuffle while doing a join.
- Broadcasting huge datasets results in a timeout.



SHUFFLE HASH JOIN

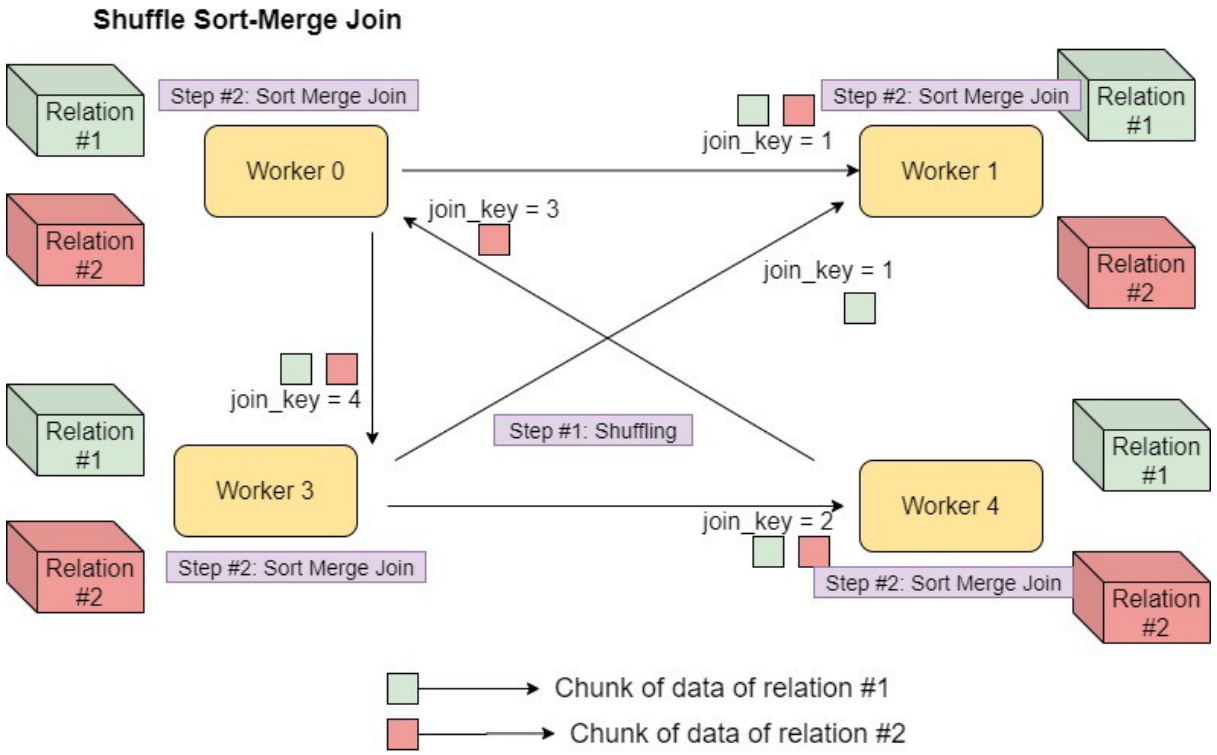
- Works on the concept of map reduce.
- Maps through a dataframe and uses values of join as output key.
- Used when one side is at least 3 times smaller than the other.
- Dataframe size is less than the broadcast threshold.
- Prefer Sort Merge Join is set to false.



SHUFFLE SORT-MERGE JOIN

- If dataframes cannot be broadcasted, spark used sort merge join.
- Uses node-to node communication strategy.
- Two steps:
    - First step exchanges and sorts datasets.
    - Second step merges the dataset.
- Better to use bucketing for optimization.
- Default since spark 2.3

Syntax: `spark.sql.join.preferSortMergejoin=true`

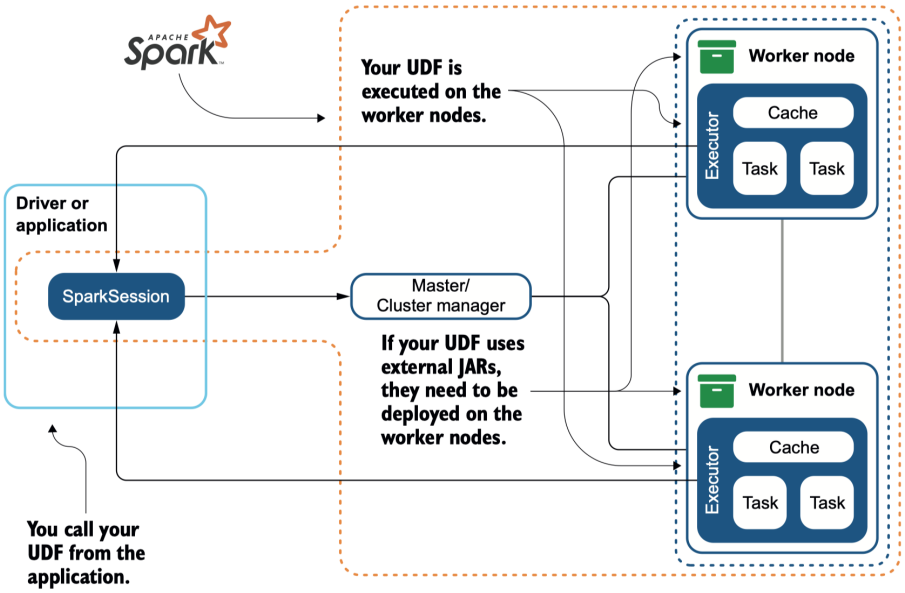


USER-DEFINED FUNCTIONS

UDFs work on columns in your dataframe. A custom function remains a function: it has arguments and a return type. They can take from 0 to 22 arguments and always return one. The returned value will be the value stored in the column.

Your UDF code must be serializable (to be transported to the node). Spark takes care of the transport part for you. If you have a UDF requiring external libraries, you must make sure that they are deployed on the worker nodes.

UDFs are called in your application (or driver), but the execution is done on the worker nodes. As a consequence, if your UDF requires external JARs, those JARs will need to be deployed on those worker nodes.



SPARK SQL UDFS

Here's a simplified example of creating a Spark SQL UDF. Note that UDFs operate per session and they will not be persisted in the underlying metastore:

In Python:

```
from pyspark.sql.types import LongType
```

Create cubed function:

```
def cubed(s):
     return s * s * s
```

Register UDF:

```
spark.udf.register("cubed", cubed, LongType())
```

Generate temporary view:

```
spark.range(1, 9).createOrReplaceTempView("udf_test")
```

Query the cubed UDF:

```
spark.sql("SELECT id, cubed(id) AS id_cubed FROM udf_test").show()
```

EVALUATION ORDER AND NULL CHECKING IN SPARK SQL

```
spark.sql("SELECT s FROM test1 WHERE s IS NOT NULL AND strlen(s) > 1")
```

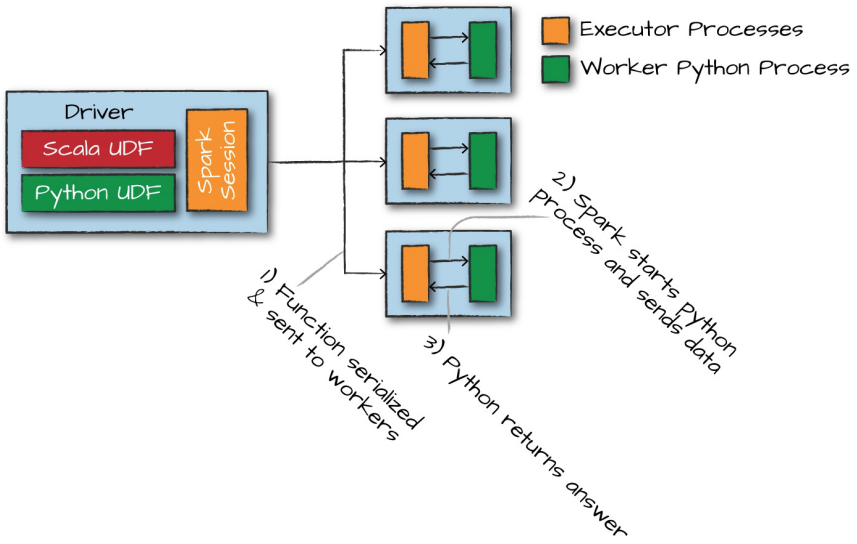
To perform proper null checking, it is recommended that you do the following:

1. Make the UDF itself null-aware and do null checking inside the UDF.

2. Use IF or CASE WHEN expressions to do the null check and invoke the UDF in a conditional branch.

UDF IN PYTHON, JAVA OR SCALA?

If the function is written in Python, something quite different happens. Spark starts a Python process on the worker, serializes all of the data to a format that Python can understand (remember, it was in the JVM earlier), executes the function row by row on that data in the Python process, and then finally returns the results of the row operations to the JVM and Spark.



Starting 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, after 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).

We recommend that you write your UDFs in Scala or Java — the small amount of time it should take you to write the function in Scala will always yield significant speed ups, and on top of that, you can still use the function from Python!