

What is spark in big data?

Spark uses Micro-batching for real-time streaming.

Apache **Spark** is open source, general-purpose distributed computing engine **used** for processing and analyzing a large amount of data.

Just like Hadoop MapReduce, it also works with the system to distribute data across the cluster and process the data in parallel.

How is spark different from Hadoop?

Need of Apache Spark

In the industry, there is a need for general purpose cluster computing tool as:

Hadoop MapReduce can only perform batch processing.

Apache Storm / S4 can only perform stream processing.

Apache Impala / Apache Tez can only perform interactive processing

Neo4j / Apache Giraph can only perform to graph processing

Apache Spark Ecosystem

Following are 6 components in Apache Spark Ecosystem which empower to Apache Spark- Spark Core, Spark SQL, Spark Streaming, Spark MLlib, Spark Graphics , and Spark R.

Working of Apache Spark

Apache Spark is open source, general-purpose distributed computing engine used for processing and analyzing a large amount of data.

Just like Hadoop [MapReduce](#), it also works with the system to distribute data across the cluster and process the data in parallel.

Spark uses master/slave architecture i.e. one central coordinator and many distributed workers.

The central coordinator is called the driver. The driver runs in its own java process.

These drivers communicate with a potentially large number of distributed workers called executors.

Each executor is a separate java process.

A Spark Application is a combination of driver and its own executors. With the help of cluster manager

Standalone Cluster Manager is the default built in cluster manager of Spark. Apart from its built-in cluster manager, Spark works with some open source cluster manager like Hadoop Yarn, Apache Mesos etc.

Web UI — Spark Application's Web Console:

Web UI (aka **Application UI** or **webUI** or **Spark UI**) is the web interface of a Spark application to monitor and inspect Spark job executions in a web browser.

The screenshot displays the Spark Web UI interface. At the top, there's a navigation bar with tabs for Jobs, Stages, Storage, Environment, Executors, and SQL. The 'Jobs' tab is selected. Below the navigation bar, the 'Spark Jobs (?)' section shows summary statistics: User: jacek, Total Uptime: 35 s, Scheduling Mode: FIFO, Active Jobs: 1, Completed Jobs: 1, and Failed Jobs: 1. A link for 'Event Timeline' is also present. The 'Active Jobs (1)' section contains a table with one job (ID 2) in progress. The 'Completed Jobs (1)' section contains a table with one job (ID 0) completed. The 'Failed Jobs (1)' section contains a table with one job (ID 1) that failed.

Spark Jobs (?)

User: jacek
Total Uptime: 35 s
Scheduling Mode: FIFO
Active Jobs: 1
Completed Jobs: 1
Failed Jobs: 1
[Event Timeline](#)

Active Jobs (1)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
2	show at <console>:24	2016/09/29 14:01:20	5 s	0/1	0/1

Completed Jobs (1)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
0	show at <console>:24	2016/09/29 14:01:07	0.3 s	1/1	1/1

Failed Jobs (1)


Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
1	show at <console>:24	2016/09/29 14:01:14	87 ms	0/1 (1 failed)	0/1 (1 failed)

web UI comes with the following tabs (which may not all be visible immediately, but only after the respective modules are in use, e.g. the SQL or Streaming tabs):

1. Jobs
2. Stages
3. Storage
4. Environment
5. Executors

Jobs Tab:

Jobs tab in [web UI](#) shows [status of all Spark jobs](#) in a Spark application (i.e. a [SparkContext](#)).

 2.1.0-SNAPSHOT

Jobs

Stages

Storage

Environment

Executors

SQL

Spark shell application UI

Spark Jobs (?)

User: jacek
Total Uptime: 35 s
Scheduling Mode: FIFO
Active Jobs: 1
Completed Jobs: 1
Failed Jobs: 1
[▶ Event Timeline](#)

Active Jobs (1)

Job Id ▾	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
2	show at <console>:24	2016/09/29 14:01:20	5 s	0/1	0/1

Completed Jobs (1)

Job Id ▾	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
0	show at <console>:24	2016/09/29 14:01:07	0.3 s	1/1	1/1

Failed Jobs (1)

Job Id ▾	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
1	show at <console>:24	2016/09/29 14:01:14	87 ms	0/1 (1 failed)	0/1 (1 failed)

Stages Tab

Stages tab in [web UI](#) shows... FIXME

3 Fair Scheduler Pools

Pool Name	Minimum Share	Pool Weight	Active Stages	Running Tasks	SchedulingMode
production	2	1	0	0	FAIR
test	3	2	0	0	FIFO
default	0	1	1	1	FIFO

Active Stages (1)

Stage Id	Pool Name	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
2	default	map at <console>:29 +details (kill)	2016/06/02 20:56:36	2 s	<div><div>7/8</div></div>	168.0 B			414.0 B

Pending Stages (1)

Stage Id	Pool Name	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
3		count at <console>:29 +details	Unknown	Unknown	<div><div>0/8</div></div>				


Completed Stages (1)

Stage Id	Pool Name	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
1	default	count at <console>:29 +details	2016/06/02 20:56:05	0.1 s	<div><div>8/8</div></div>	192.0 B			

Failed Stages (1)

Stage Id	Pool Name	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write	Failure Reason
0	default	count at <console>:29 +details	2016/06/02 20:55:45	0.2 s	<div><div>7/8 (1 failed)</div></div>					Job aborted due to stage failure: Task 1 in stage 0.0 failed 1 times, most recent failure: Lost task 1.0 in stage 0.0 (TID 1, localhost): java.lang.Exception: failed +details

Executors Tab: Executors tab in web UI shows... FIXME

 2.3.1-SNAPSHOT

JobsStagesStorageEnvironmentExecutorsSQL

Spark shell application UI

Executors

Summary

	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write	Blacklisted
Active(1)	8	1 KB / 384.1 MB	0.0 B	8	0	0	9	9	2 s (72 ms)	0.0 B	460 B	460 B	0
Dead(0)	0	0.0 B / 0.0 B	0.0 B	0	0	0	0	0	0 ms (0 ms)	0.0 B	0.0 B	0.0 B	0
Total(1)	8	1 KB / 384.1 MB	0.0 B	8	0	0	9	9	2 s (72 ms)	0.0 B	460 B	460 B	0

Executors

Show 20 entries

Search:

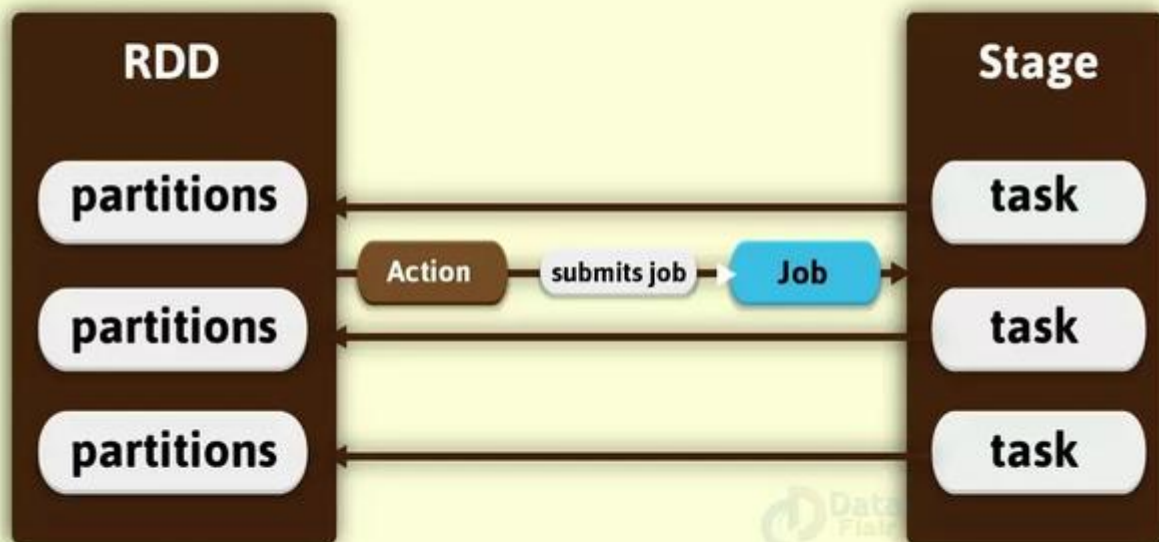
Executor ID	Address	Status	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write	Thread Dump
driver	192.168.0.185:62559	Active	8	1 KB / 384.1 MB	0.0 B	8	0	0	9	9	2 s (72 ms)	0.0 B	460 B	460 B	Thread Dump

Showing 1 to 1 of 1 entries

Previous1Next

Introduction to Stages in Spark

Tasks and submitting a job



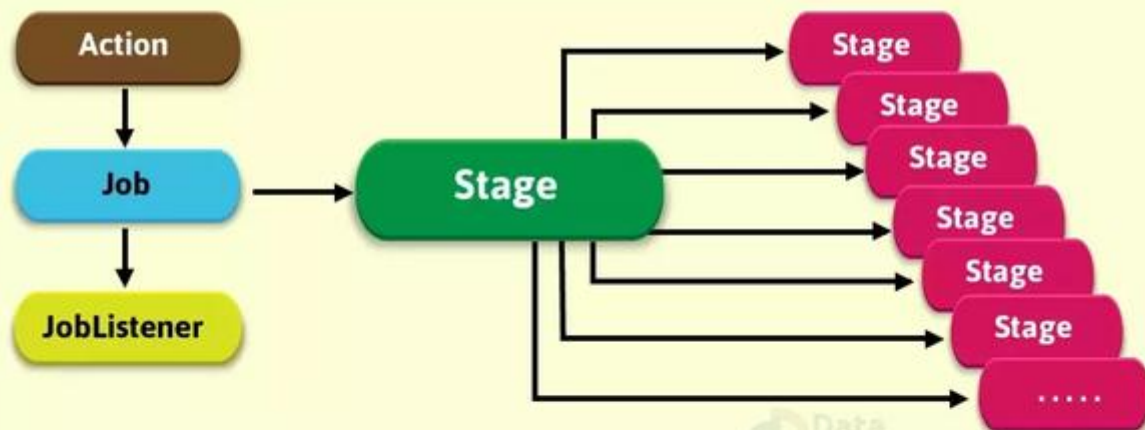
Submitting a job triggers execution of the stage and its parent Spark stages

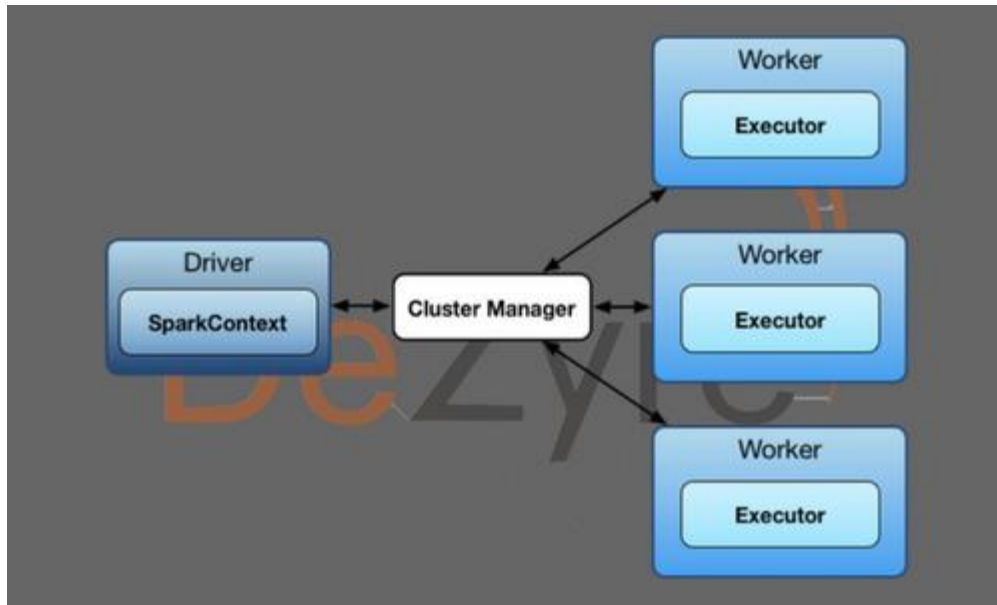
There are two types of Spark Stage

Basically, stages in Apache spark are two categories

- a. ShuffleMapStage in Spark
- b. ResultStage in Spark

Submitting a job triggers execution of the stage and its parent stages





Spark Architecture Overview

Apache Spark follows a master/slave architecture with two main daemons and a cluster manager –

1. Master Daemon – (Master/Driver Process)
2. Worker Daemon –(Slave Process)

A spark cluster has a single Master and any number of Slaves/Workers. The driver and the executors run their individual Java processes.

Role of Driver in Spark Architecture

Spark Driver – Master Node of a Spark Application

It is the central point and the entry point of the Spark Shell (Scala, Python, and R).

The driver program runs the main () function of the application and is the place where the Spark Context is created.

Spark Driver contains various components – DAGScheduler, TaskScheduler, BackendScheduler and BlockManager responsible for the translation of spark user code into actual spark jobs executed on the cluster.

- The driver program that runs on the master node of the spark cluster schedules the job execution and negotiates with the cluster manager.

- It translates the RDD's into the execution graph and splits the graph into multiple stages.
- Driver stores the metadata about all the Resilient Distributed Databases and their partitions.
- Cockpits of Jobs and Tasks Execution -Driver program converts a user application into smaller execution units known as tasks. Tasks are then executed by the executors i.e. the worker processes which run individual tasks.
- Driver exposes the information about the running spark application through a Web UI at port 4040.

Role of Executor in Spark Architecture

Executor is a distributed agent responsible for the execution of tasks.

Every spark applications has its own executor process. Executors usually run for the entire lifetime of a Spark application and this phenomenon is known as "Static Allocation of Executors".

However, users can also opt for dynamic allocations of executors wherein they can add or remove spark executors dynamically to match with the overall workload.

- Executor performs all the data processing.
- Reads from and Writes data to external sources.
- Executor stores the computation results data in-memory, cache or on hard disk drives.
- Interacts with the storage systems.

Role of Cluster Manager in Spark Architecture

An external service responsible for acquiring resources on the spark cluster and allocating them to a spark job. There are 3 different types of cluster managers a Spark application can leverage for the allocation and deallocation of various physical resources such as memory for client spark jobs, CPU memory, etc. Hadoop YARN, Apache Mesos or the simple standalone spark cluster manager either of them can be launched on-premise or in the cloud for a spark application to run.

Choosing a cluster manager for any spark application depends on the goals of the application because all cluster managers provide different set of scheduling capabilities. To get started with apache spark, the standalone cluster manager is the easiest one to use when developing a new spark application.

Understanding the Run Time Architecture of a Spark Application

What happens when a Spark Job is submitted?

When a client submits a spark user application code, the driver implicitly converts the code containing transformations and actions into a logical directed acyclic graph (DAG).

At this stage, the driver program also performs certain optimizations like pipelining transformations and then it converts the logical

DAG into physical execution plan with set of stages. After creating the physical execution plan, it creates small physical execution units referred to as tasks under each stage.

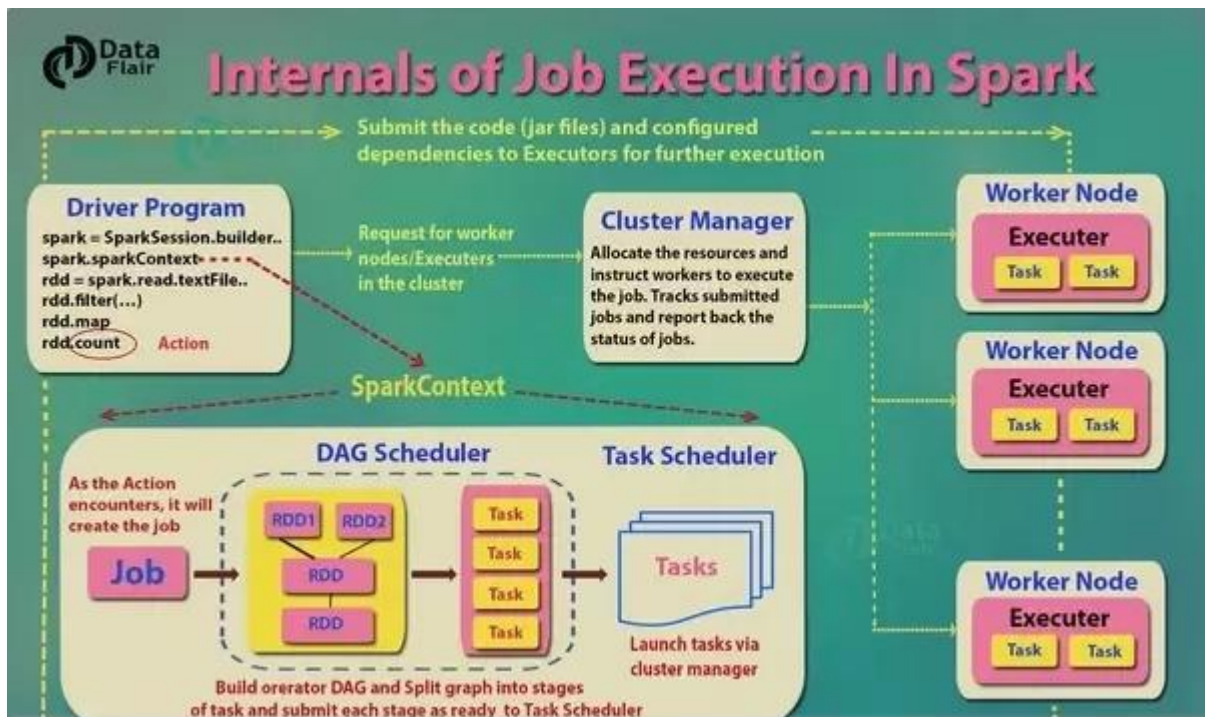
Then tasks are bundled to be sent to the Spark Cluster.

The driver program then talks to the cluster manager and negotiates for resources. The cluster manager then launches executors on the worker nodes on behalf of the driver.

At this point the driver sends tasks to the cluster manager based on data placement. Before executors begin execution, they register themselves with the driver program so that the driver has holistic view of all the executors. Now executors start executing the various tasks assigned by the driver program.

At any point of time when the spark application is running, the driver program will monitor the set of executors that run. Driver program in the spark architecture also schedules future tasks based on data placement by tracking the location of cached data. When driver programs main () method exits or when it call the stop () method of the Spark Context, it will terminate all the executors and release the resources from the cluster manager.

The structure of a Spark program at higher level is - RDD's are created from the input data and new RDD's are derived from the existing RDD's using different transformations, after which an action is performed on the data. In any spark program, the DAG operations are created by default and whenever the driver runs the Spark DAG will be converted into a physical execution plan.



RDD:

At a high level, every Spark application consists of a *driver program* that runs the user's main function and executes various *parallel operations* on a cluster. The main abstraction Spark provides is a *resilient distributed dataset* (RDD), which is a collection of elements partitioned across the nodes of the cluster that can be operated on in parallel. RDDs are created by starting with a file in the Hadoop file system (or any other Hadoop-supported file system), or an existing Scala collection in the driver program, and transforming it. Users may also ask Spark to *persist* an RDD in memory, allowing it to be reused efficiently across parallel operations. Finally, RDDs automatically recover from node failures.

SparkContext — Entry Point to Spark Core

`SparkContext` (aka **Spark context**) is the heart of a Spark application.

You could also assume that a `SparkContext` instance *is* a Spark application.

Once a `SparkContext` is created you can use it to [create RDDs](#), [accumulators](#) and [broadcast variables](#), access Spark services and [run jobs](#) (until `SparkContext` is [stopped](#))

A Spark context is essentially a client of Spark's execution environment and acts as the *master of your Spark application*

Creating SparkContext Instance

You can create a `SparkContext` instance with or without creating a [SparkConf](#) object first.

Getting Existing or Creating New SparkContext — getOrCreate Methods

`getOrCreate(): SparkContext`

`getOrCreate(conf: SparkConf): SparkContext`

`getOrCreate` methods allow you to get the existing `SparkContext` or create a new one.

```
import org.apache.spark.SparkContext
val sc = SparkContext.getOrCreate()
```

```
// Using an explicit SparkConf object
import org.apache.spark.SparkConf
val conf = new SparkConf()
  .setMaster("local[*]")
  .setAppName("SparkMe App")
val sc = SparkContext.getOrCreate(conf)
```

```
SparkContext()
SparkContext(conf: SparkConf)
SparkContext(master: String, appName: String, conf: SparkConf)
SparkContext(
  master: String,
  appName: String,
  sparkHome: String = null,
  jars: Seq[String] = Nil,
  environment: Map[String, String] = Map())
```

Units of Physical Execution

Jobs: Work required to compute RDD in runJob.

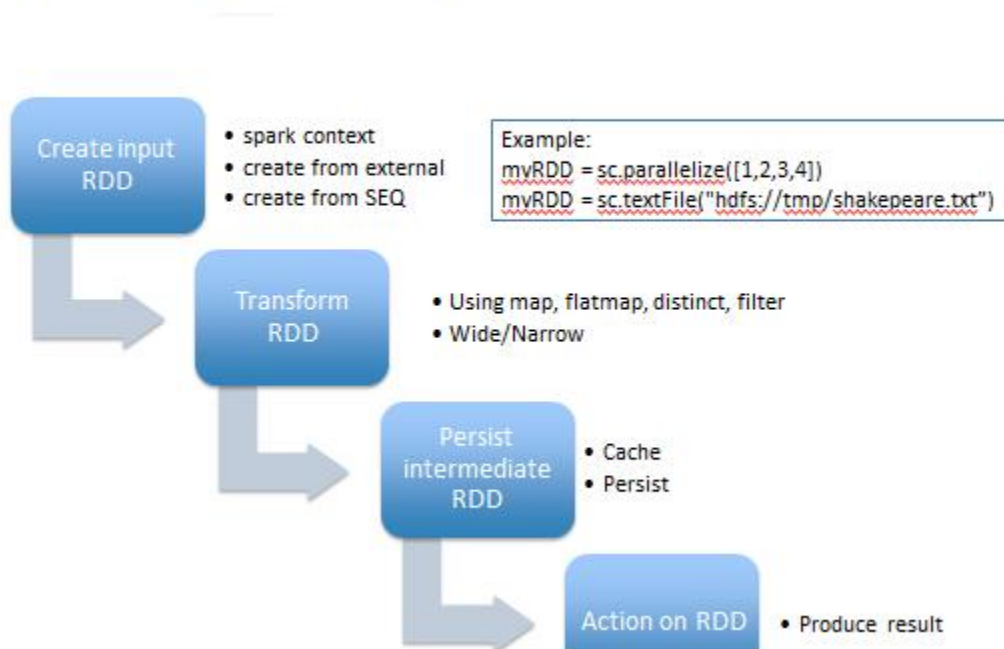
Stages: A wave of work within a job, corresponding to one or more pipelined RDD's.

Tasks: A unit of work within a stage, corresponding to one RDD partition.

Shuffle: The transfer of data between stages.

DAG(Directed Acyclic Graph) in [Apache Spark](#) is a set of Vertices and Edges, where *vertices* represent the RDDs and the *edges* represent the Operation to be applied on RDD. In Spark DAG, every edge directs from earlier to later in the sequence. On the calling of *Action*, the created DAG submits to DAG Scheduler which further splits the graph into the stages of the task

Spark Program Flow by RDD



The difference between `cache` and `persist` operations is purely syntactic. `cache` is a synonym of `persist` or `persist(MEMORY_ONLY)`, i.e. `cache` is merely `persist` with the default storage level `MEMORY_ONLY`

Caching or persistence are optimization techniques for (iterative and interactive) Spark computations. They help saving interim partial results so they can be reused in subsequent stages. These interim results as `RDDs` are thus kept in memory (default) or more solid storage like disk and/or replicated. `RDDs` can be cached using `cache` operation. They can also be persisted using `persist` operation.

`persist`, `cache`

These functions can be used to adjust the storage level of a `RDD`. When freeing up memory, Spark will use the storage level identifier to decide which partitions should be kept. The parameter less variants `persist()` and `cache()` are just abbreviations for `persist(StorageLevel.MEMORY_ONLY)`.

Warning: Once the storage level has been changed, it cannot be changed again!

Warning -Cache judiciously... see ([\(Why\) do we need to call cache or persist on a RDD](#))

Just because you can cache a `RDD` in memory doesn't mean you should blindly do so. Depending on how many times the dataset is accessed and the amount of work involved in doing so, recomputation can be faster than the price paid by the increased memory pressure.

It should go without saying that if you only read a dataset once there is no point in caching it, it will actually make your job slower. The size of cached datasets can be seen from the Spark Shell..

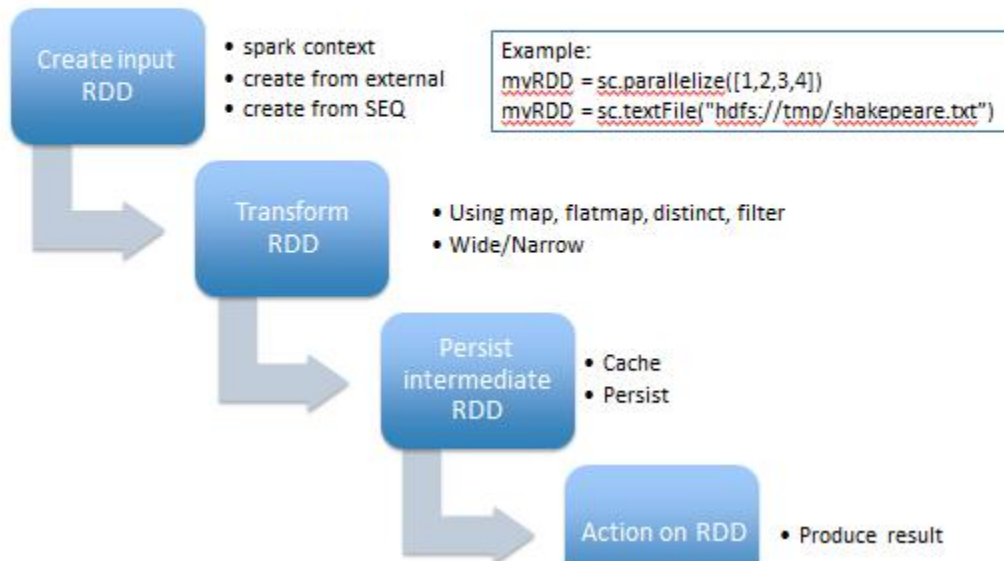
Listing Variants...

```
def cache(): RDD[T] def persist(): RDD[T] def persist(newLevel: StorageLevel): RDD[T]
```

***See below example : ***

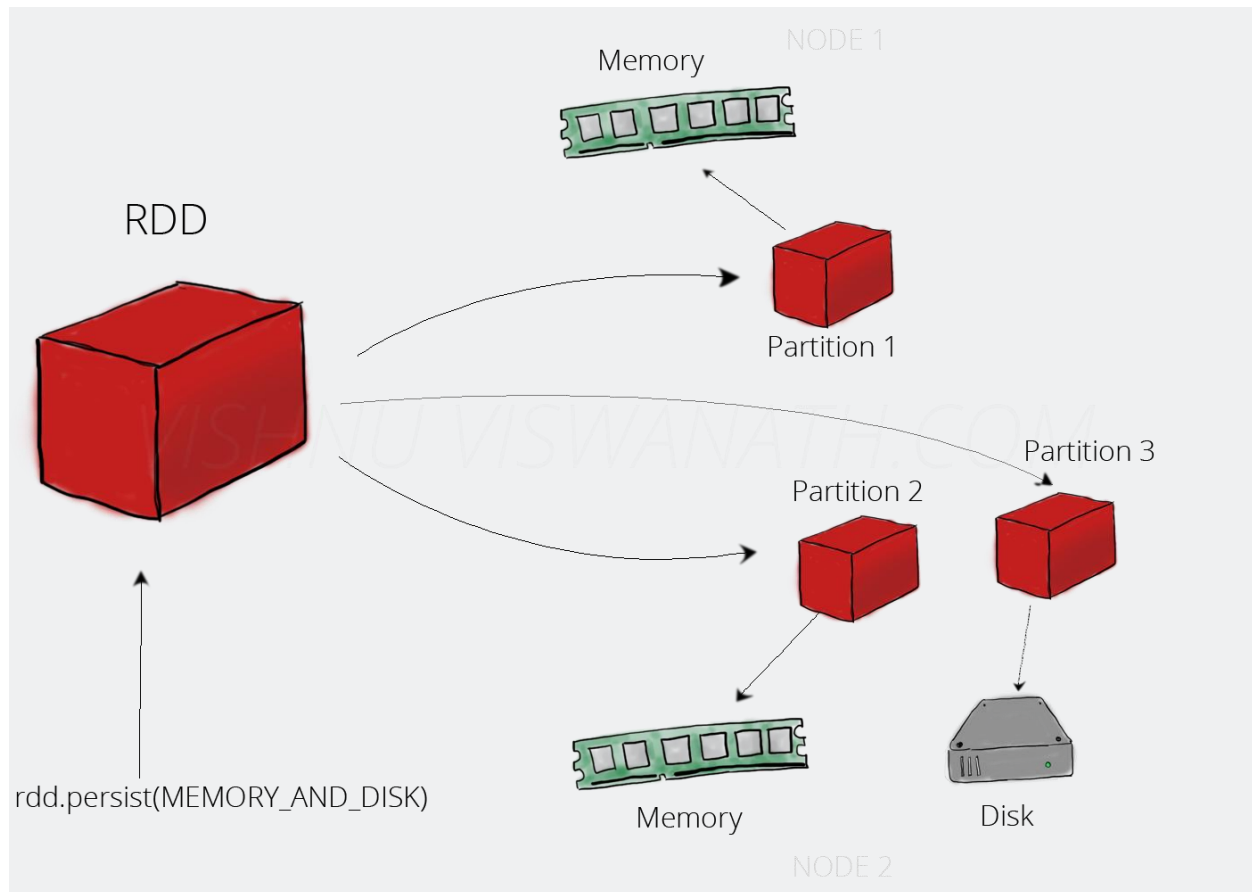
```
val c = sc.parallelize(List("Gnu", "Cat", "Rat", "Dog", "Gnu", "Rat"), 2)
c.getStorageLevel res0: org.apache.spark.storage.StorageLevel =
StorageLevel(false, false, false, false, 1) c.cache c.getStorageLevel res2:
org.apache.spark.storage.StorageLevel = StorageLevel(false, true, false, true,
1)
```

Spark Program Flow by RDD



The difference between `cache` and `persist` operations is purely syntactic. `cache` is a synonym of `persist` or `persist(MEMORY_ONLY)`, i.e. `cache` is merely `persist` with the default storage level `MEMORY_ONLY`

Persist in memory and disk:



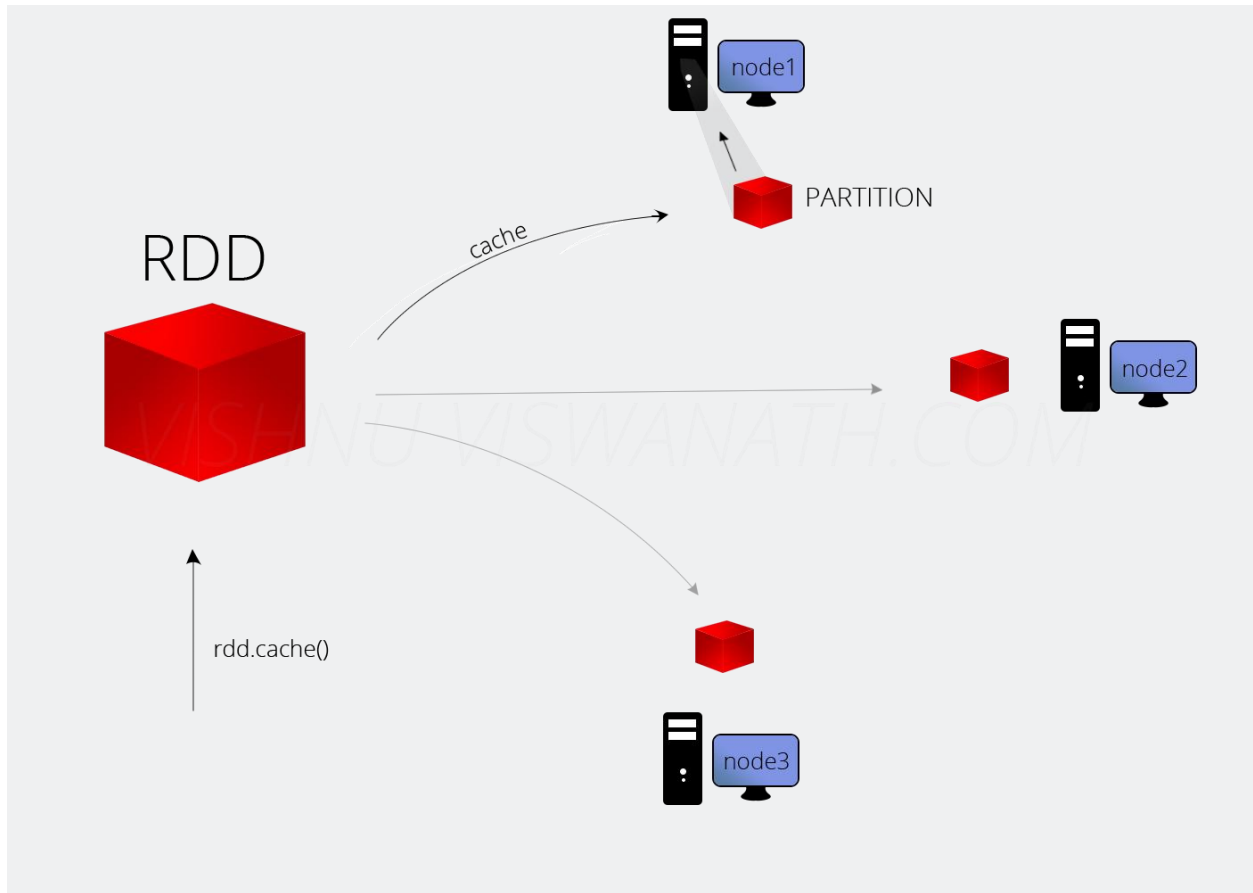
Spark gives 5 types of Storage level

- `MEMORY_ONLY`
- `MEMORY_ONLY_SER`
- `MEMORY_AND_DISK`
- `MEMORY_AND_DISK_SER`
- `DISK_ONLY`

`cache()` will use `MEMORY_ONLY`. If you want to use something else, use `persist(StorageLevel.<type*>)`

Cache

Caching can improve the performance of your application to a great extent.



1. Broadcast Variables
2. Accumulators

2.1. Broadcast Variables in Spark

Broadcast Variables despite shipping a copy of it with tasks. We can use them, for example, to give a copy of a large input dataset in an efficient manner to every node. In [Spark](#), by using efficient algorithms it is possible to distribute broadcast variables. It helps to reduce communication cost.

Spark can broadcast the common data automatically, needed by tasks within each stage. The data broadcasted this way then cached in serialized form and also deserialized before running each task. Hence, creating broadcast

variables explicitly is useful in some cases, like while tasks across multiple stages need the same data. While caching the data in the deserialized form is important.

It is also very important that no modification can take place on the object *v* after it is broadcast. It will help ensure that all nodes get the same value of the broadcast variable.

```
1. scala> val broadcastVar1 = sc.broadcast(Array(1, 2, 3))
2. broadcastVar1:
   org.apache.spark.broadcast.Broadcast[Array[Int]] =
   Broadcast(0)
3.
4. scala> broadcastVar1.value
5. res0: Array[Int] = Array(1, 2, 3)
```

2.2. Accumulators

The variables which are only “added” through a commutative and associative operation. Also, can efficiently support in parallel. We can use *Accumulators* to implement counters or sums. Spark natively supports programmers for new types and accumulators of numeric types.

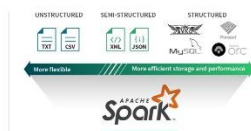
By calling `SparkContext.longAccumulator()`, we can create a numeric accumulator and by `SparkContext.doubleAccumulator()`, we can accumulate values of type long or double.

Learning is a continuous process. Though I am using Spark from quite a long time now, I never noted down my practice exercise. With this repo, I am documenting it!

I have used databricks free community cloud for this excercises,
link: <https://community.cloud.databricks.com/login.html>



Spark Structured Streaming



```
#1. create stream DF
dfStream = spark.readStream
  .option("maxFilesPerTrigger", 1) \ max number files to consider per trigger
  .format("csv") \ txt, csv, json, parquet
  .option("header", "true") \
  .schema(staticSchema) \
  .load("/FileStore/tables/*.csv") \
```

```
dfStream.isStream
Out[40]: True
```

[illegible]

```
#2. some code to process streaming data  
from pyspark.sql.functions import col, window  
dfNew = dfStream.selectExpr("CustomerId"  
                             , "(UnitPrice * Quantity)"  
                             , "InvoiceDate")\n                        .groupBy( col("CustomerId"),  
                                sum("total cost"))
```

```
dfNew.writeStream\
  .format("memory")\
  .queryName("test")\
```

```
Out[34]: <pyspark.sql.streaming.StreamingQuery at 0x7fd40b8a7d50>
```

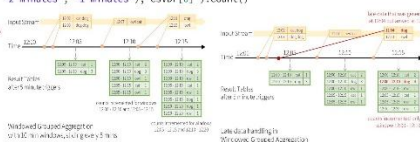
```
Out[34]: <pyspark.sql.streaming.StreamingQuery at 0x7fd40b8a7d50>
```

④

groupBy: aggregation in every 2 minutes window, sliding every 1 minute

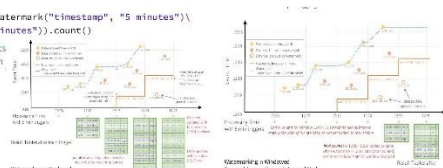
```
windowedCount = csvDF.groupBy(window(csvDF[0], "2 minutes", "1 minutes"), csvDF[0]).count()
```

```
runningStreamWindow = windowedCount.writeStream()
    .format("memory")
    .queryName("myResult")
    .outputMode("complete")
    .start();
```



```
runningStreamWindowWatermark = csvDF.withWatermark("timestamp", "5 minutes")\
  .groupBy(window(csvDF[0], "2 minutes" "1 minutes")) count()
```

in Spark 2.1, we have introduced watermarking, which lets the engine automatically track the current event time in the data and attempt to clean up old state accordingly.



(3)

```
runningStream = csvDF.writeStream(
  .format("memory")\
  .queryName("myResult")\
  .outputMode("update"))\
.start()
#ERROR
Caution: Complete output mode not supported
(//FileStore/table//).start()
```

```

// Complete output mode not supported when there are no streaming aggregations on streaming DataFrames/Datasets;;\nFileSource
(//FileStore/tables/)''.start()

```

```
start writing stream to memory,
w.writeStream(
  mat("memory"),
  ryName("test"),
  putNode("complete")
);
rt();
```

- Complete Mode**
 - The entire updated Result Table will be written to the external storage.
- Append Mode**
 - Only the new rows appended in the Result Table since the last trigger will be written to the external storage.
- Update Mode**
 - Only the rows that are updated in the Result Table since the last trigger will be written to the external storage.

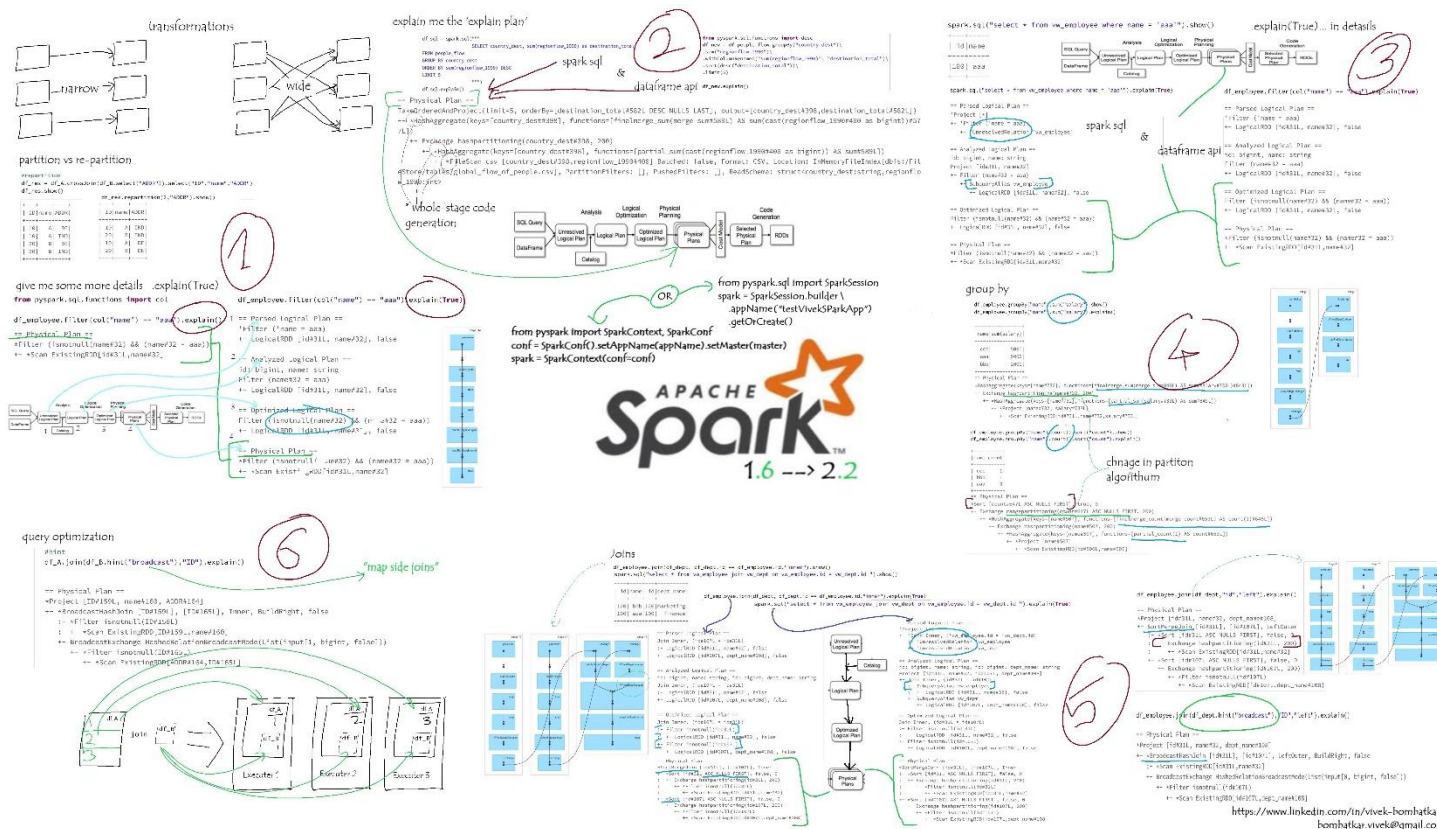
test (id: df228a3d-9c7b-40ee-af3e-832b4d740b) Last updated: 22 hours ago

```
Out[34]: <pyspark.sql.streaming.StreamingQuery at 0x7
```

```
spark.sql("select * from test").show()
```

```
-----+-----+-----+
| CustomerID | window | sum(total_cost) |
|-----+-----+-----+
```

[illegible]



spark_explain_plan

spark_explain_plan notebook

DAG and explain plan

<https://www.tutorialkart.com/apache-spark/dag-and-physical-execution-plan/>

***How Apache Spark builds a DAG and Physical Execution Plan ? ***

1. User submits a spark application to the Apache Spark.
2. Driver is the module that takes in the application from Spark side.
3. Driver identifies transformations and actions present in the spark application. These identifications are the tasks.
4. Based on the flow of program, these tasks are arranged in a graph like structure with directed flow of execution from task to task forming no loops in the graph (also called DAG). DAG is pure logical.
5. This logical DAG is converted to Physical Execution Plan. Physical Execution Plan contains stages.

6. Some of the subsequent tasks in DAG could be combined together in a single stage. Based on the nature of transformations, Driver sets stage boundaries.
7. There are two transformations, namely
 - a. narrow transformations : Transformations like Map and Filter that does not require the data to be shuffled across the partitions.
 - b. wide transformations : Transformations like ReduceByKey that does require the data to be shuffled across the partitions.
8. Transformation that requires data shuffling between partitions, i.e., a wide transformation results in stage boundary.
9. DAG Scheduler creates a Physical Execution Plan from the logical DAG. Physical Execution Plan contains tasks and are bundled to be sent to nodes of cluster.

Catalyst optimizer

<http://data-informed.com/6-steps-to-get-top-performance-from-the-changes-in-spark-2-0/>

What is Catalyst? Catalyst is the name of Spark's integral query optimizer and execution planner for Dataset/DataFrame.

Catalyst is where most of the "magic" happens to improve the execution speed of your code. But in any complex system, "magic" is unfortunately not good enough to always guarantee optimal performance. Just as with relational databases, it is valuable to learn a bit about exactly how the optimizer works in order to understand its planning and tune your applications.

In particular, Catalyst can perform sophisticated refactors of complex queries. However, almost all of its optimizations are qualitative and rule-based rather than quantitative and statistics-based. For example, Spark knows how and when to do things like combine filters, or move filters before joins. Spark 2.0 even allows you to define, add, and test out your own additional optimization rules at runtime. [1][2]

On the other hand, Catalyst is not designed to perform many of the common optimizations that RDBMSs have performed for decades, and that takes some understanding and getting used to.

For example, Spark doesn't "own" any storage, so it does not build on-disk indexes, B-Trees, etc. (although its parquet file support, if used well, can get you some related features). Spark has been optimized for the volume, variety, etc. of big data – so, traditionally, it has not been designed to maintain and use statistics about a stable dataset. E.g., where an RDBMS might know that a specific filter will eliminate most

records, and apply it early in the query, Spark 2.0 does not know this fact and won't perform that optimization

Catalyst, the optimizer and Tungsten, the execution engine!

<https://db-blog.web.cern.ch/blog/luca-canali/2016-09-spark-20-performance-improvements-investigated-flame-graphs>

*** Note in particular the steps marked with (*), they are optimized with whole-stage code generation

Code generation is the key The key to understand the improved performance is with the new features in Spark 2.0 for whole-stage code generation.

Deep dive into the new Tungsten execution engine

<https://databricks.com/blog/2016/05/23/apache-spark-as-a-compiler-joining-a-billion-rows-per-second-on-a-laptop.html>

1. The `explain()` function in the expression below has been extended for whole-stage code generation. In the explain output, when an operator has a star around it (*), whole-stage code generation is enabled. In the following case, Range, Filter, and the two Aggregates are both running with whole-stage code generation. Exchange, however, does not implement whole-stage code generation because it is sending data across the network.

```
spark.range(1000).filter("id > 100").selectExpr("sum(id)").explain()
```

```
== Physical Plan ==
*Aggregate(functions=[sum(id#201L)])
+- Exchange SinglePartition, None
   +- *Aggregate(functions=[sum(id#201L)])
      +- *Filter (id#201L > 100)
         +- *Range 0, 1, 3, 1000, [id#201L]
```

2. Vectorization The idea here is that instead of processing data one row at a time, the engine batches multiples rows together in a columnar format, and each operator uses simple loops to iterate over data within a batch. Each `next()` call would thus return a batch of tuples, amortizing the cost of virtual function dispatches. These simple loops would also enable compilers and CPUs to execute more efficiently with the benefits mentioned earlier.

Catalyst Optimizer

<https://data-flair.training/blogs/spark-sql-optimization-catalyst-optimizer/>

1. Fundamentals of Catalyst Optimizer

In the depth, Catalyst contains the tree and the set of rules to manipulate the tree.

Trees

A tree is the main data type in the catalyst. A tree contains node object. For each node, there is a node

Rules

We can manipulate tree using rules. We can define rules as a function from one tree to another tree.

2.

a. Analysis - Spark SQL Optimization starts from relation to be computed. It is computed either from abstract syntax tree (AST) returned by SQL parser or dataframe object created using API.

b. Logical Optimization - In this phase of Spark SQL optimization, the standard rule-based optimization is applied to the logical plan. It includes constant folding, predicate pushdown, projection pruning and other rules.

c. In this phase, one or more physical plan is formed from the logical plan, using physical operator matches the Spark execution engine. And it selects the plan using the cost model.

d. Code Generation - It involves generating Java bytecode to run on each machine. Catalyst uses the special feature of Scala language, "Quasiquotes" to make code generation easier because it is very tough to build code generation engines.

cost based optimization

<https://databricks.com/blog/2017/08/31/cost-based-optimizer-in-apache-spark-2-2.html>

Query Benchmark and Analysis We took a non-intrusive approach while adding these cost-based optimizations to Spark by adding a global config `spark.sql.cbo.enabled` to enable/disable this feature. In Spark 2.2, this parameter is set to false by default.

1. At its core, Spark's Catalyst optimizer is a general library for representing query plans as trees and sequentially applying a number of optimization rules to manipulate them.
2. A majority of these optimization rules are based on heuristics, i.e., they only account for a query's structure and ignore the properties of the data being processed,

3. ANALYZE TABLE command

CBO relies on detailed statistics to optimize a query plan. To collect these statistics, users can issue these new SQL commands described below:

ANALYZE TABLE table_name COMPUTE STATISTICS

sigmod_spark_sql

http://people.csail.mit.edu/matei/papers/2015/sigmod_spark_sql.pdf

working-with-udfs-in-apache-spark

<https://blog.cloudera.com/blog/2017/02/working-with-udfs-in-apache-spark/>

- It's important to understand the performance implications of Apache Spark's UDF features. Python UDFs for example (such as our CTOF function) result in data being serialized between the executor JVM and the Python interpreter running the UDF logic – this significantly reduces performance as compared to UDF implementations in Java or Scala. Potential solutions to alleviate this serialization bottleneck include:
- Accessing a Hive UDF from PySpark as discussed in the previous section. The Java UDF implementation is accessible directly by the executor JVM. Note again that this approach only provides access to the UDF from the Apache Spark's SQL query language. Making use of the approach also shown to access UDFs implemented in Java or Scala from PySpark, as we demonstrated using the previously defined Scala UDAF example.
- Another important component of Spark SQL to be aware of is the Catalyst query optimizer. Its capabilities are expanding with every release and can often provide dramatic performance improvements to Spark SQL queries; however, arbitrary UDF implementation code may not be well understood by Catalyst (although future features[3] which analyze bytecode are being considered to address this). As such, using Apache Spark's built-in SQL query functions will often lead to the best performance and should be the first approach considered whenever introducing a UDF can be avoided

spark-functions-vs-udf-performance

<https://stackoverflow.com/questions/38296609/spark-functions-vs-udf-performance>

```
df_employee.show()
+-----+
| id|name|
+-----+
|100| aaa|
|120| bbb|
|150| ccc|
+-----+

df_employee.filter(col("name") == "aaa").show()
+-----+
| id|name|
+-----+
|100| aaa|
+-----+
```

from pyspark.sql.functions import col

```
df_employee.filter(col("name") == "aaa").explain()
== Physical Plan ==
*Filter (isnotnull(name#32) && (name#32 = aaa))
+- *Scan ExistingRDD[id#31L,name#32]
```

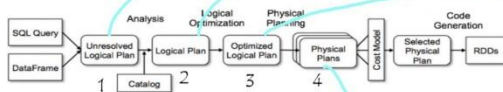
df_employee.filter(col("name") == "aaa").explain(True)

```
1 == Parsed Logical Plan ==
  'Filter ('name = aaa)
  +- LogicalRDD [id#31L, name#32], false

2 == Analyzed Logical Plan ==
  id: bigint, name: string
  Filter (name#32 = aaa)
  +- LogicalRDD [id#31L, name#32], false

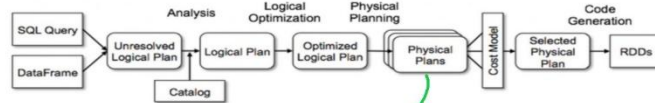
3 == Optimized Logical Plan ==
  Filter (isnotnull(name#32) && (name#32 = aaa))
  +- LogicalRDD [id#31L, name#32], false

4 == Physical Plan ==
  *Filter (isnotnull(name#32) && (name#32 = aaa))
  +- *Scan ExistingRDD[id#31L,name#32]
```



```
spark.sql("select * from vw_employee where name = 'aaa']").show()
```

```
+-----+
| id|name|
+-----+
|100| aaa|
+-----+
```



```
spark.sql("select * from vw_employee where name = 'aaa']").explain(True)
```

```
== Parsed Logical Plan ==
'Project [*]
+- 'Filter ('name = aaa)
   +- UnresolvedRelation 'vw_employee'

== Analyzed Logical Plan ==
id: bigint, name: string
Project [id#31L, name#32]
+- Filter (name#32 = aaa)
   +- SubqueryAlias vw_employee
      +- LogicalRDD [id#31L, name#32], false

== Optimized Logical Plan ==
Filter (isnotnull(name#32) && (name#32 = aaa))
+- LogicalRDD [id#31L, name#32], false

== Physical Plan ==
*Filter (isnotnull(name#32) && (name#32 = aaa))
+- *Scan ExistingRDD[id#31L,name#32]
```

df_employee.filter(col("name") == "aaa").explain(True)

```
== Parsed Logical Plan ==
'Filter ('name = aaa)
+- LogicalRDD [id#31L, name#32], false

== Analyzed Logical Plan ==
id: bigint, name: string
Filter (name#32 = aaa)
+- LogicalRDD [id#31L, name#32], false

== Optimized Logical Plan ==
Filter (isnotnull(name#32) && (name#32 = aaa))
+- LogicalRDD [id#31L, name#32], false

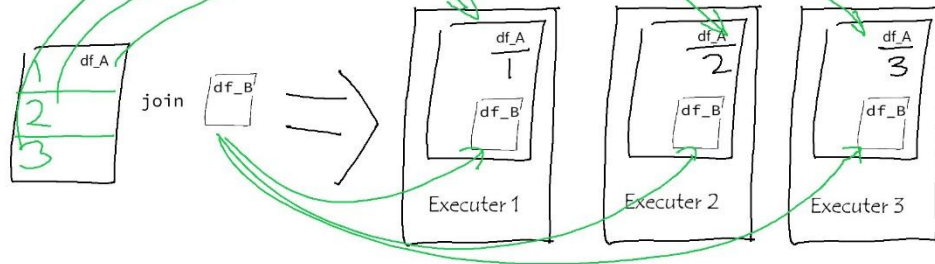
== Physical Plan ==
*Filter (isnotnull(name#32) && (name#32 = aaa))
+- *Scan ExistingRDD[id#31L,name#32]
```



```
#hint
df_A.join(df_B.hint("broadcast"), "ID").explain()
```

"map side joins"

```
== Physical Plan ==
*Project [ID#159L, name#160, ADDR#164]
+- *BroadcastHashJoin [ID#159L], [ID#165L], Inner, BuildRight, false
  :- *Filter isnotnull(ID#159L)
  : +- *Scan ExistingRDD[ID#159L,name#160]
  +- BroadcastExchange HashedRelationBroadcastMode(List(input[1, bigint, false]))
    +- *Filter isnotnull(ID#165L)
    +- *Scan ExistingRDD[ADDR#164,ID#165L]
```



```
df_employee.join(df_dept, df_dept.id == df_employee.id, "inner").show()
spark.sql("select * from vw_employee join vw_dept on vw_employee.id = vw_dept.id ").show()
```

```
-----+-----+
| id|name| id|dept_name|
-----+-----+
|120|bbb|120|marketing|
|100|aaa|100|finance|
-----+-----+
```

```
df_employee.join(df_dept, df_dept.id == df_employee.id, "inner").explain(True)
```

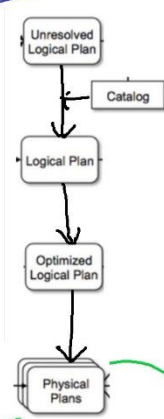
```
spark.sql("select * from vw_employee join vw_dept on vw_employee.id = vw_dept.id ").explain(True)
```

```
== Parsed Logical Plan ==
Join Inner, (id#107L = id#31L)
:- LogicalRDD [id#31L, name#32], false
+- LogicalRDD [id#107L, dept_name#108], false

== Analyzed Logical Plan ==
id: bigint, name: string, id: bigint, dept_name: string
Join Inner, (id#107L = id#31L)
:- LogicalRDD [id#31L, name#32], false
+- LogicalRDD [id#107L, dept_name#108], false

== Optimized Logical Plan ==
Join Inner, (id#107L = id#31L)
:- Filter isnotnull(id#31L)
  :- LogicalRDD [id#31L, name#32], false
  +- Filter isnotnull(id#107L)
  +- LogicalRDD [id#107L, dept_name#108], false

== Physical Plan ==
*SortMergeJoin [id#31L], [id#107L], Inner
:- *Sort [id#31L ASC NULLS FIRST], false, 0
  :- Exchange hashpartitioning(id#31L, 200)
  :- *Filter isnotnull(id#31L)
  :- *Scan ExistingRDD[id#31L,name#32]
+- *Sort [id#107L ASC NULLS FIRST], false, 0
  :- Exchange hashpartitioning(id#107L, 200)
  :- *Filter isnotnull(id#107L)
  +- *Scan ExistingRDD[id#107L,dept_name#108]
```



```
== Parsed Logical Plan ==
'Project [id#31L, name#32, id#107L, dept_name#108]
+- 'Join Inner, ('vw_employee.id = 'vw_dept.id')
  :- 'UnresolvedRelation 'vw_employee'
  :- 'UnresolvedRelation 'vw_dept'

== Analyzed Logical Plan ==
id: bigint, name: string, id: bigint, dept_name: string
Project [id#31L, name#32, id#107L, dept_name#108]
+- Join Inner, (id#31L = id#107L)
  :- SubqueryAlias vw_employee
  :- LogicalRDD [id#31L, name#32], false
  :- SubqueryAlias vw_dept
  :- LogicalRDD [id#107L, dept_name#108], false

== Optimized Logical Plan ==
Join Inner, (id#31L = id#107L)
:- Filter isnotnull(id#31L)
  :- LogicalRDD [id#31L, name#32], false
  :- Filter isnotnull(id#107L)
  +- LogicalRDD [id#107L, dept_name#108], false

== Physical Plan ==
*SortMergeJoin [id#31L], [id#107L], Inner
:- *Sort [id#31L ASC NULLS FIRST], false, 0
  :- Exchange hashpartitioning(id#31L, 200)
  :- *Filter isnotnull(id#31L)
  :- *Scan ExistingRDD[id#31L,name#32]
+- *Sort [id#107L ASC NULLS FIRST], false, 0
  :- Exchange hashpartitioning(id#107L, 200)
  :- *Filter isnotnull(id#107L)
  +- *Scan ExistingRDD[id#107L,dept_name#108]
```

```
df_employee.join(df_dept,"id","left").explain()

== Physical Plan ==
*Project [id#31L, name#32, dept_name#108]
+- SortMergeJoin [id#31L], [id#107L], LeftOuter
  :- *Sort [id#31L ASC NULLS FIRST], false, 0
  : +- Exchange hashpartitioning(id#31L, 200)
  :   +- *Scan ExistingRDD[id#31L,name#32]
  +- *Sort [id#107L ASC NULLS FIRST], false, 0
    +- Exchange hashpartitioning(id#107L, 200)
      +- *Filter isnotnull(id#107L)
        +- *Scan ExistingRDD[id#107L,dept_name#108]
```

```
df_employee.join(df_dept.hint("broadcast"),"ID","left").explain()

== Physical Plan ==
*Project [id#31L, name#32, dept_name#108]
+- *BroadcastHashJoin [id#31L], [id#107L], LeftOuter, BuildRight, false
  :- *Scan ExistingRDD[id#31L,name#32]
  +- BroadcastExchange HashedRelationBroadcastMode(List(input[0, bigint, false]))
    +- *Filter isnotnull(id#107L)
      +- *Scan ExistingRDD[id#107L,dept_name#108]
```

```
df_employee.groupBy("name").sum("salary").show()
df_employee.groupBy("name").sum("salary").explain()
```

```
-----+
|name|sum(salary)|
-----+
|ccc|      5000|
|aaa|      9000|
|bbb|      2000|
-----+

== Physical Plan ==
*HashAggregate(keys=[name#732], functions=[finalmerge_sum(merge sum#849L) AS sum(salary#733L)#843L])
+- Exchange hashpartitioning(name#732, 200)
  +- *HashAggregate(keys=[name#732], functions=[partial_sum(salary#733L) AS sum#849L])
    +- *Project [name#732, salary#733L]
      +- *Scan ExistingRDD[id#731L,name#732,salary#733L]
```

```
df_employee.groupBy("name").count().sort("count").show()
df_employee.groupBy("name").count().sort("count").explain()
```

```
-----+
|name|count|
-----+
|ccc|    1|
|bbb|    1|
|aaa|    3|
-----+

== Physical Plan ==
*Sort [count#647L ASC NULLS FIRST], true, 0
+- Exchange rangepartitioning(count#647L ASC NULLS FIRST, 200)
  +- *HashAggregate(keys=[name#507], functions=[finalmerge_count(merge count#653L) AS count(1)#646L])
    +- Exchange hashpartitioning(name#507, 200)
      +- *HashAggregate(keys=[name#507], functions=[partial_count(1) AS count#653L])
        +- *Project [name#507]
          +- *Scan ExistingRDD[id#506L,name#507]
```

```
#repartition
df_res = df_A.crossJoin(df_B.select("ADDR")).select("ID","name","ADDR")
df_res.show()
```

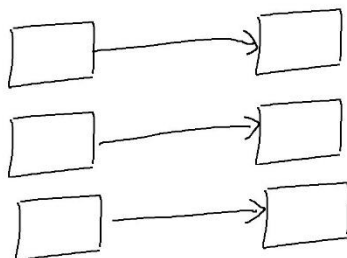
```
+---+---+---+
| ID|name|ADDR|
+---+---+---+
| 10|  A|  DE|
| 10|  A| IND|
| 20|  B|  DE|
| 20|  B| IND|
+---+---+---+
```

```
df_res.repartition(2,"ADDR").show()
```

```
+---+---+---+
| ID|name|ADDR|
+---+---+---+
| 10|  A| IND|
| 20|  B| IND|
| 10|  A|  DE|
| 20|  B|  DE|
+---+---+---+
```

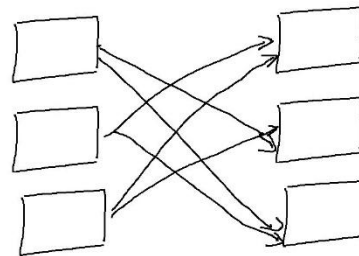
```
df_res.repartition("ADDR").show()
```

```
+---+---+---+
| ID|name|ADDR|
+---+---+---+
| 10|  A|  DE|
| 20|  B|  DE|
| 10|  A| IND|
| 20|  B| IND|
+---+---+---+
```



narrow

transformations



wide

spark-submit

<https://spark.apache.org/docs/2.2.0/submitting-applications.html> https://www.cloudera.com/documentation/enterprise/5-4-x/topics/cdh_ig_running_spark_on_yarn.html <https://jaceklaskowski.gitbooks.io/masterin>

g-apache-spark/yarn/ <https://blog.cloudera.com/blog/2014/05/apache-spark-resource-management-and-yarn-app-models/>

```
./bin/spark-submit \
  --class <main-class> \
  --master <master-url> \
  --deploy-mode <deploy-mode> \
  --conf <key>=<value> \
  ... # other options
  <application-jar> \
  [application-arguments]
```

--class: The entry point for your application (e.g. org.apache.spark.examples.SparkPi)
--master: The master URL for the cluster (e.g. spark://23.195.26.187:7077)
--deploy-mode: Whether to deploy your driver on the worker nodes (cluster) or locally as an external client (client) (default: client) †
--conf: Arbitrary Spark configuration property in key=value format. For values that contain spaces wrap “key=value” in quotes (as shown).
application-jar: Path to a bundled jar including your application and all dependencies. The URL must be globally visible inside of your cluster, for instance, an hdfs:// path or a file:// path that is present on all nodes.
application-arguments: Arguments passed to the main method of your main class, if any

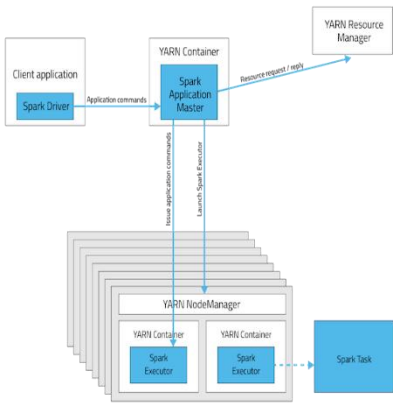
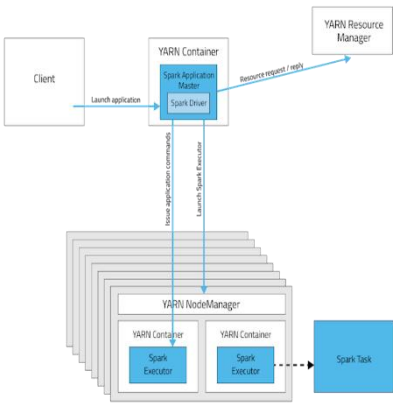
java vs python code execution

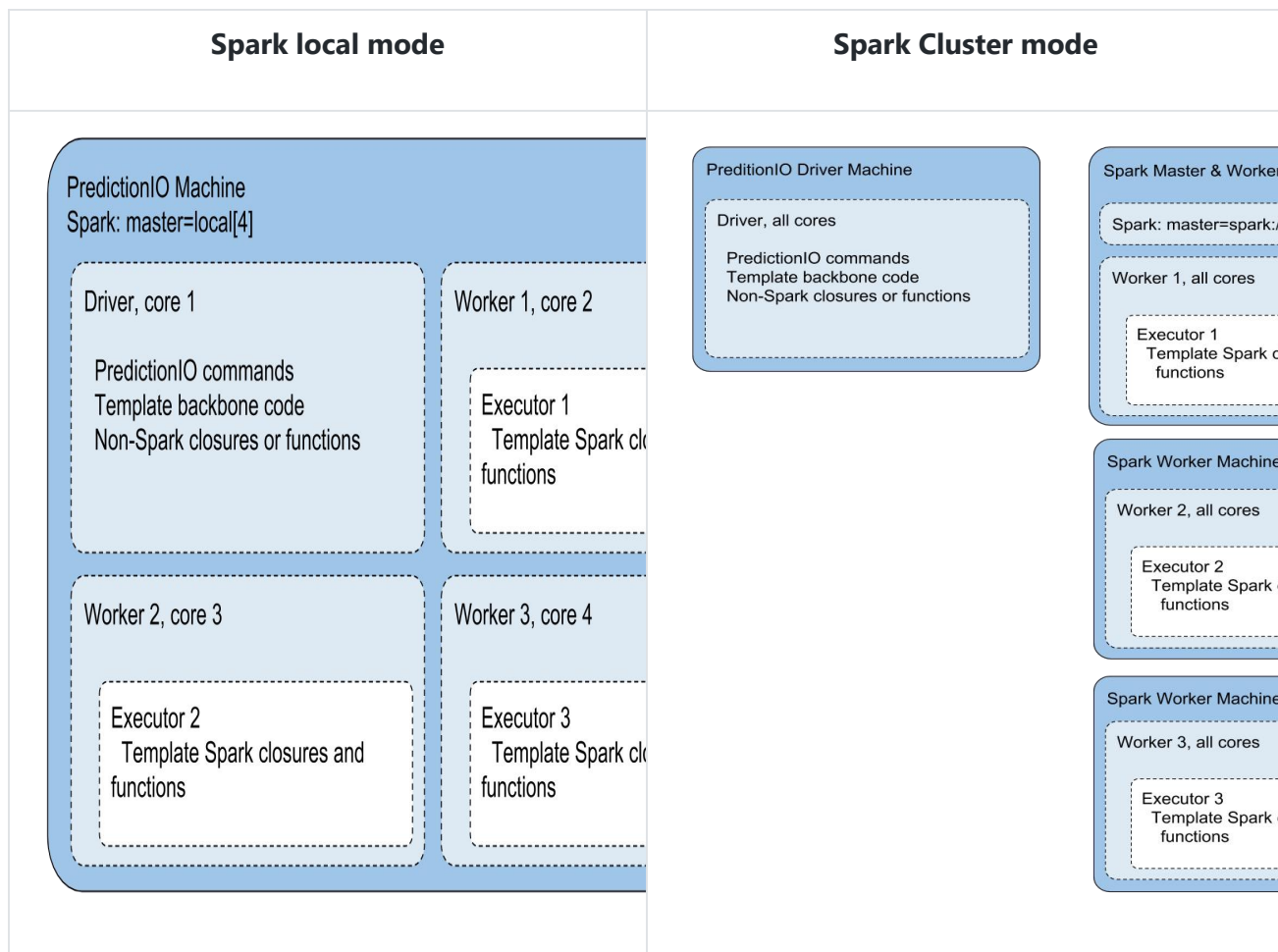
java class		python script	
--class 'class path of java main application'		(at the end of spark-submit) 'fully qualified path of the main python script'	
ex. --class com.abc.project1.Main		/opt/src/project1/module1/main.py 'pass the parameters'	
--jars 'assembly jar (or “uber” jar) containing your code and its dependencies, to be distributed with your application'		--py-files 'add .py, .zip or .egg files to be distributed with your application.'	
local	local[n]	local[n,f]	yarn
Run locally with one worker thread, no	Run locally with K worker threads , set this to the number of cores.	Run Spark locally with n worker threads and F	Connect to a YARN cluster in client or cluster mode depending on the value of --deploy-mode. The cluster location will be found based

local	local[n]	local[n,f]	yarn
parallelism	local[*] Run with as many worker threads as logical cores	maxFailures	on the HADOOP_CONF_DIR or YARN_CONF_DIR variable

YARN client vs cluster

Deploy modes are all about where the Spark driver runs.

YARN client	YARN cluster
driver runs on the host where the job is submitted	the driver runs in the ApplicationMaster on a cluster host chosen by YARN.
client that launches the application needs to be alive	client doesn't need to continue running for the entire lifetime of the application
	
Spark local mode	Spark Cluster mode



	YARN Cluster	YARN Client	Spark Standalone
Driver runs in:	Application Master	Client	Client
Who requests resources?	Application Master	Application Master	Client
Who starts executor processes?	YARN NodeManager	YARN NodeManager	Spark Slave
Persistent services	YARN ResourceManager and NodeManagers	YARN ResourceManager and NodeManagers	Spark Master and Workers
Supports Spark Shell?	No	Yes	Yes

Drivers and Executors

IMP Concepts	
Application	single job, a sequence of jobs, a long-running service issuing new commands as needed or an interactive exploration session.
Spark Driver	driver is the process running the spark context. This driver is responsible for converting the application to a directed graph of individual steps to execute on the cluster. There is one driver per application.
Spark Application Master	responsible for negotiating resource requests made by the driver with YARN and finding a suitable set of hosts/containers in which to run the Spark applications. There is one Application Master per application.
Spark Executor	A single JVM instance on a node that serves a single Spark application. An executor runs multiple tasks over its lifetime, and multiple tasks concurrently. A node may have several Spark executors and there are many nodes running Spark Executors for each client application.
Spark Task	represents a unit of work on a partition of a distributed dataset.

Dataframe operation on multiple columns

<https://medium.com/@mrpowers/performing-operations-on-multiple-columns-in-a-pyspark-dataframe-36e97896c378>

'Parsed Logical Plan' --> 'Analyzed Logical Plan' --> 'Optimized Logical Plan' --> 'Physical Plan'

Spark is smart enough to optimized (in Physical Plan) the multiple operation done in for kind of loop on dataframe

Below 2 code snippets will produce similar Physical Plan

```
for col in data_frame.columns:
    df_res= data_frame.withColumn() \
                      .withColumn()
```

```
df_res= data_frame.select(*(when(col(c) ... ,...).otherwise(col(c)).alias(c) for c in
data_frame.columns ))
```

Spark job monitoring

<https://databricks.com/blog/2015/06/22/understanding-your-spark-application-through-visualization.html>

Spark History Server web UI

a. Event timeline of spark events

The ability to view Spark events in a timeline is useful for identifying the bottlenecks in an application.

- Event timeline available in three levels
 - across all jobs
 - within one job
 - within one stage.

b. DAG