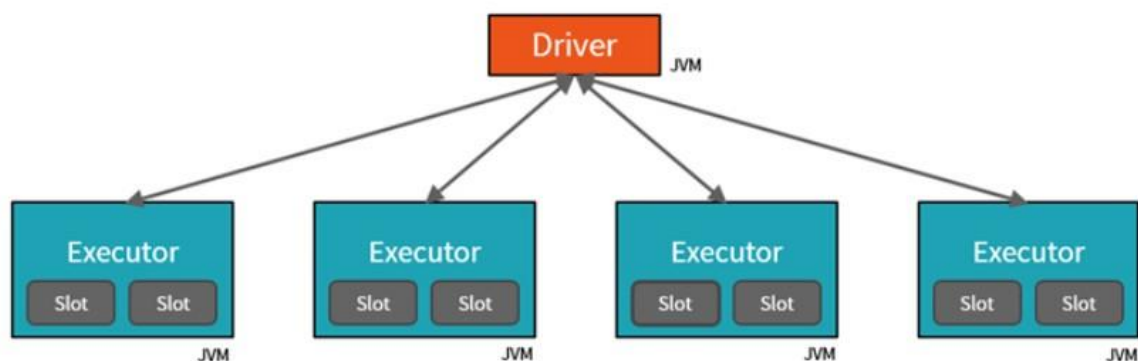


Interview Questions Spark

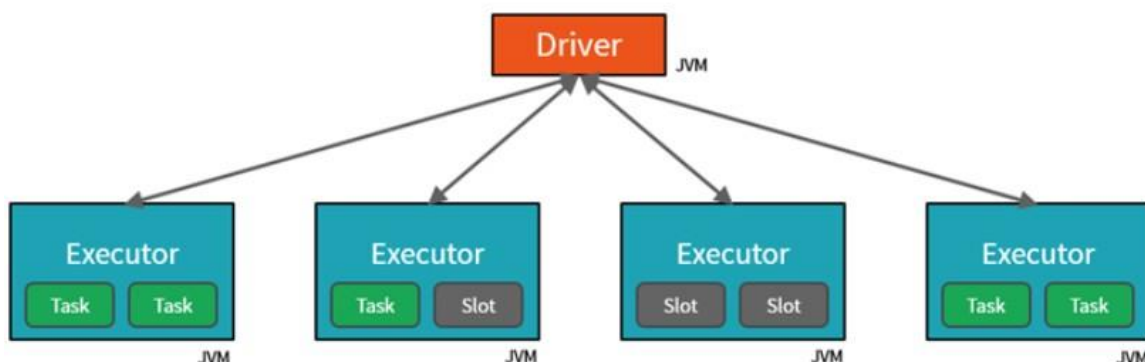
1. Explain Spark Clusters Architecture And Process?

The diagram below shows an example Spark cluster, basically there exists a **Driver node** that communicates with **executor nodes**. Each of these executor **nodes** have **slots** which are logically like execution **cores**.

Spark Physical Cluster

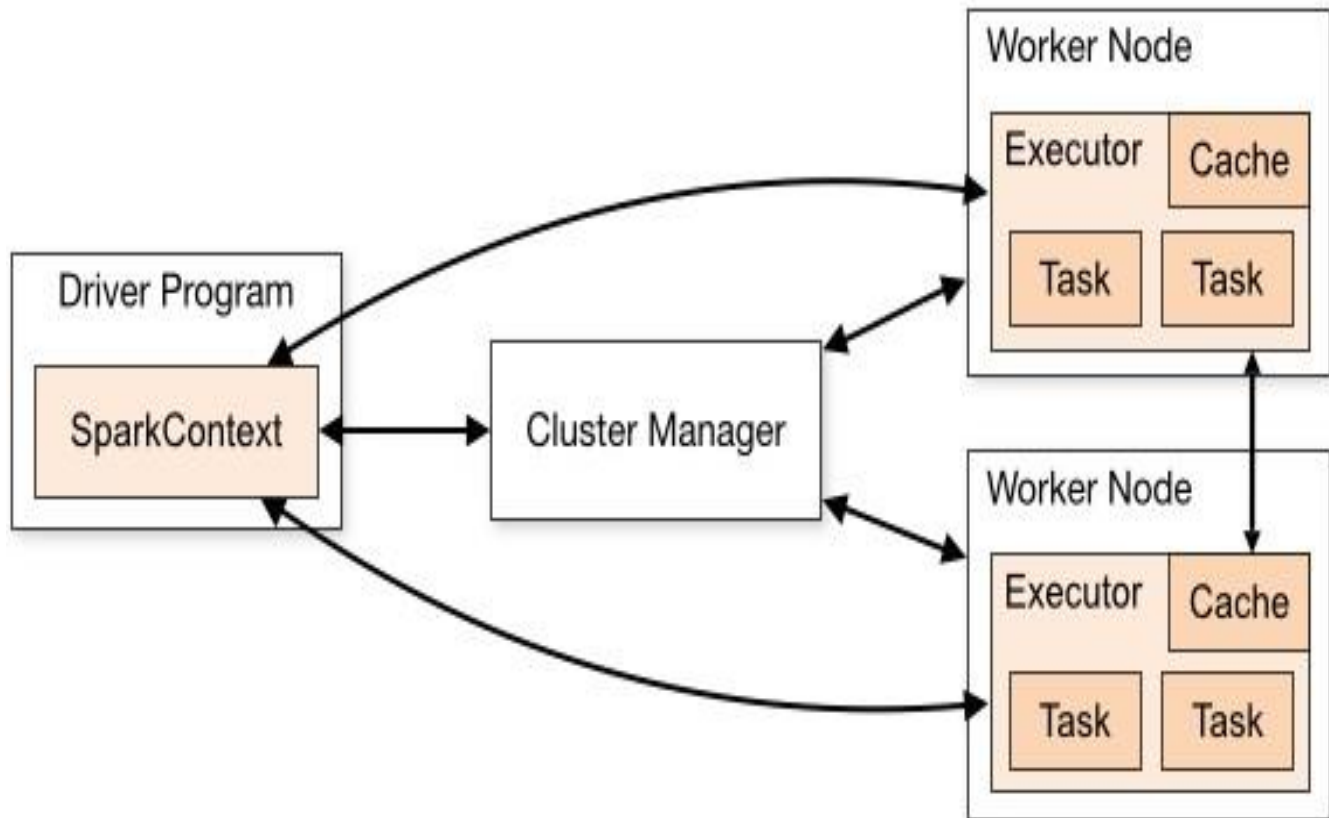


The Driver sends Tasks to the empty slots on the Executors when work has to be done:



At a high level, every Apache Spark application consists of a **driver program** that launches various parallel operations on **executor Java Virtual Machines (JVMs)** running either in a cluster or locally on the same machine. In Databricks, the notebook interface is the **driver program**. This driver program contains the main loop for the program and creates distributed datasets on the cluster, then applies operations (transformations & actions) to those datasets.

Driver programs access Apache Spark through a **SparkSession** object regardless of deployment location.



2. Spark Job Execution process and involved steps?

At high level, when any action is called on the RDD, Spark creates the DAG and submits it to the DAG scheduler.

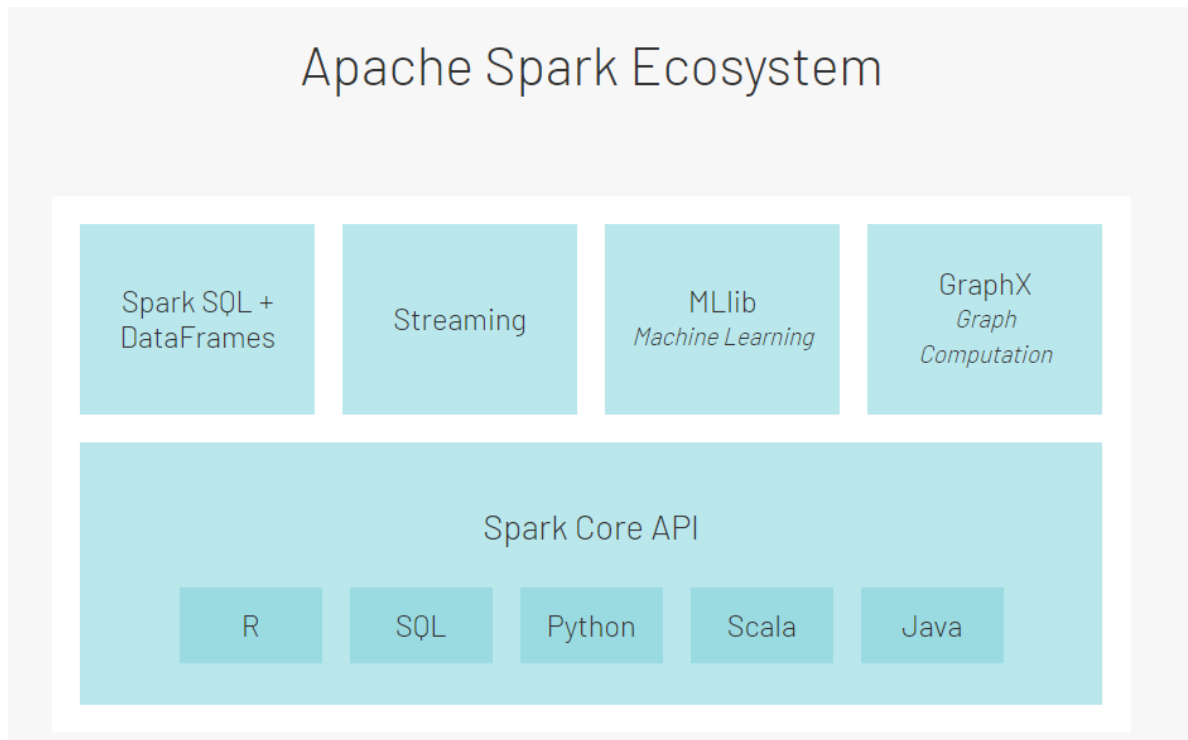
- ❑ The DAG scheduler divides operators into stages of tasks. A stage is comprised of tasks based on partitions of the input data. The DAG scheduler pipelines operators together. For e.g. Many map operators can be scheduled in a single stage. The final result of a DAG scheduler is a set of stages.
- ❑ The Stages are passed on to the Task Scheduler. The task scheduler launches tasks via cluster manager (Spark Standalone/Yarn/Mesos). The task scheduler doesn't know about dependencies of the stages.
- ❑ The Worker executes the tasks on the Slave.

- Job: A piece of code which reads some input from HDFS or local, performs some computation on the data and writes some output data.
- Stages: Jobs are divided into stages. Stages are classified as a Map or reduce stages (Its easier to understand if you have worked on Hadoop and want to correlate). Stages are divided based on computational boundaries, all computations (operators) cannot be Updated in a single Stage. It happens over many stages.
- Tasks: Each stage has some tasks, one task per partition. One task is executed on one partition of data on one executor (machine).
- DAG: DAG stands for Directed Acyclic Graph, in the present context its a DAG of operators.
- Executor: The process responsible for executing a task.
- Master: The machine on which the Driver program runs
- Slave: The machine on which the Executor program runs

3. Differences between Hadoop Map reduce And Spark?

HADOOP MAPREDUCE	SPARK
The data is stored in the disc.	The data is stored in-memory.
Computing is based on the disc.	Computing relies on RAM.
Fault tolerance is done through replication.	Fault tolerance is done through RDD.
Hard to work with real-time data.	Easy to work with real-time data.
Less costly in comparison to spark.	More costly.
For batch processing only.	Supports interactive query.

4. What are the components of Spark?



1) **Structured Data: Spark SQL**

Many data scientists, analysts, and general business intelligence users rely on interactive SQL queries for exploring data. Spark SQL is a Spark module for structured data processing. It provides a programming abstraction called DataFrame and can also act as distributed SQL query engine. It enables unmodified Hadoop Hive queries to run up to 100xfaster on existing deployments and data. It also provides powerful integration with the rest of the Spark ecosystem (e.g., integrating SQL query processing with machine learning).

2) **Streaming Analytics: Spark Streaming**

Many applications need the ability to process and analyse not only batch data, but also streams of new data in real-time. Running on top of Spark, Spark Streaming enables powerful interactive and analytical applications across both streaming and historical data, while inheriting Spark's ease of use and fault tolerance characteristics. It readily integrates with a wide variety of popular data sources, including HDFS, Flume, Kafka, and Twitter.

3) **Machine Learning: MLlib**

Machine learning has quickly emerged as a critical piece in mining Big Data for actionable insights. Built on top of Spark, MLlib is a scalable machine learning library that delivers both high-quality algorithms (e.g., multiple iterations to increase accuracy) and blazing speed (upto 100x faster than MapReduce). The library is usable in Java, Scala, and Python as part of Spark applications, so that you can include it in complete workflows.

4) **Graph Computation: GraphX**

GraphX is a graph computation engine built on top of Spark that enables users to interactively build, transform and reason about graph structured data at scale. It comes complete with a library of common algorithms.

5) **General Execution: Spark Core**

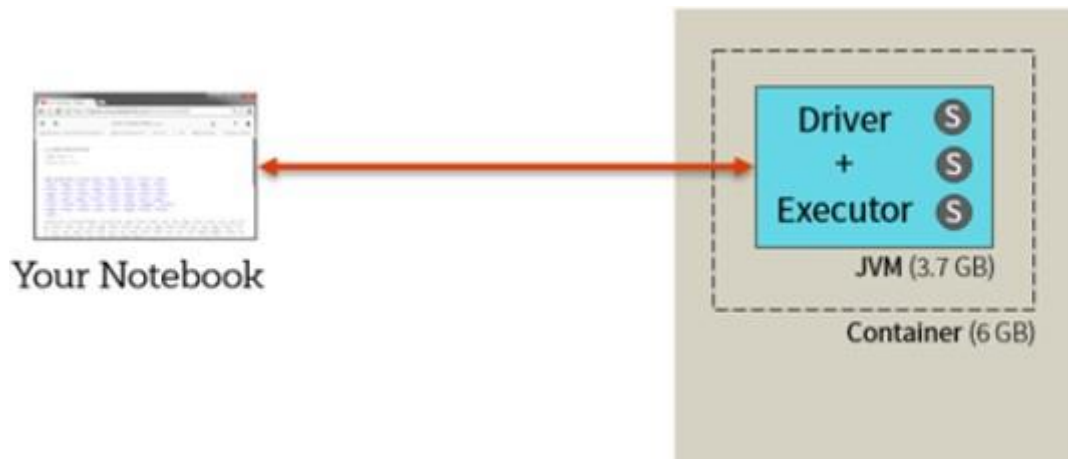
Spark Core is the underlying general execution engine for the Spark platform that all other functionality is built on top of. It provides in-memory computing capabilities to deliver speed, a generalized execution model to support a wide variety of applications, and **Java**, **Scala**, and **Python** APIs for ease of development.

5. What is Single Node Cluster (Local Mode) in Spark?

A Single Node cluster is a cluster consisting of a Spark driver and no Spark workers. Such clusters support Spark jobs and all Spark data sources, including Delta Lake. In contrast, Standard clusters require at least one Spark worker to run Spark jobs.

Single Node clusters are helpful in the following situations:

- ❑ Running single node machine learning workloads that need Spark to load and save data
- ❑ Lightweight exploratory data analysis (EDA)



Single Node cluster properties

A Single Node cluster has the following properties:

- Runs Spark locally with as many executor threads as logical cores on the cluster (thenumber of cores on driver - 1).
- Has 0 workers, with the driver node acting as both master and worker.
- The executor stderr, stdout, and log4j logs are in the driver log.
- Cannot be converted to a Standard cluster. Instead, create a new cluster with the modeset to Standard

6. Why RDD resilient?

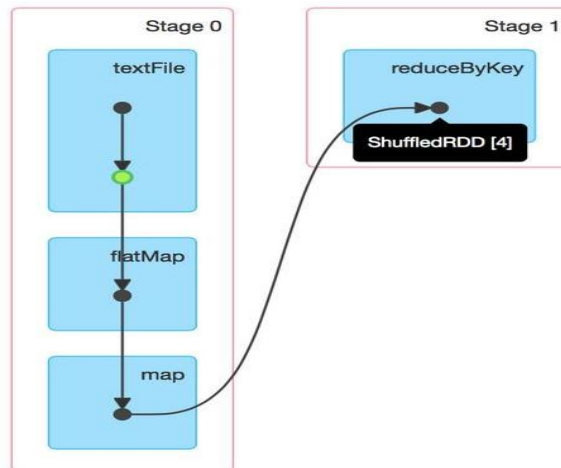
Apache spark RDD is Resilient Distributed Dataset. Resilient means self-auto recovery from failures. That's called **fault tolerance**. Fault Tolerance property means RDD, has a capability of handling if any loss occurs. It can recover the failure itself, here fault refers to failure. If any bug or loss found, RDD has the capability to recover the loss. We need a redundant element to redeem the lost data. Redundant data plays important role in a self-recovery process. Ultimately, we can recover lost data by redundant data.

In Spark, a job is associated with a chain of RDD dependencies organized in a directed acyclic graph (DAG) that looks like the following

Details for Job 0

Status: SUCCEEDED
Completed Stages: 2

- ▶ Event Timeline
- ▼ DAG Visualization



RDD Lineage (like *RDD operator graph* or *RDD dependency graph*) is a graph of all the parent RDDs of a RDD. It is built as a result of applying transformations to the RDD and creates a logical execution plan.

7. Difference between persist and cache?

Cache() and **persist()** both the methods are used to improve performance of spark computation. These methods help to save intermediate results so they can be reused in subsequent stages.

The only difference between **cache()** and **persist()** is ,

Cache technique we can save intermediate results in memory only

Persist () we can save the intermediate results in 5 storage levels(MEMORY_ONLY, MEMORY_AND_DISK, MEMORY_ONLY_SER, MEMORY_AND_DISK_SER, DISK_ONLY).

Spark Caching & Persistence

Spark caching can be used to pull data sets into a cluster-wide in-memory cache. This is very useful for accessing repeated data, such as querying a small “hot” dataset or when running an iterative algorithm

There are two ways to persist RDDs in Spark:

1. `cache()`
2. `persist()`

There are some advantages of RDD caching and persistence mechanism in spark.

- Time efficient
- Cost efficient
- Lesser the execution time.

STORAGE TYPES:

- 1) MEMORY_ONLY
- 2) MEMORY_AND_DISK
- 3) DISK_ONLY
- 4) MEMORY_ONLY_SER
- 5) MEMORY_AND_DISK_SER

8. What is narrow and wide transformation?

At high level, there are two transformations that can be applied onto the RDDs, namely narrow transformation and wide transformation. Wide transformations basically result in stage boundaries.

Narrow transformation - doesn't require the data to be shuffled across the partitions. for example, Map, filter etc..

wide transformation - requires the data to be shuffled for example, reduceByKey etc..



vs

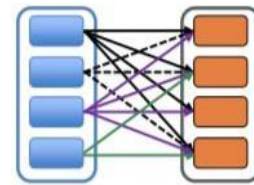
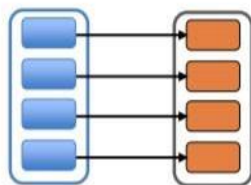


narrow

wide

each partition of the parent RDD is used by
at most one partition of the child RDD

multiple child RDD partitions may depend
on a single parent RDD partition

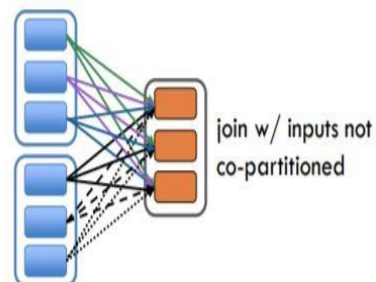
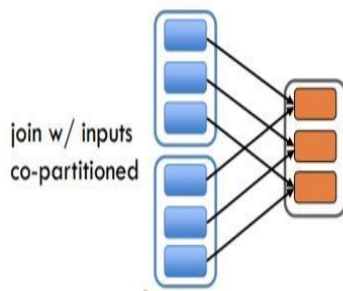
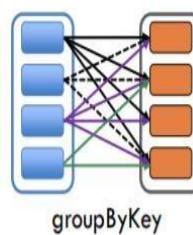
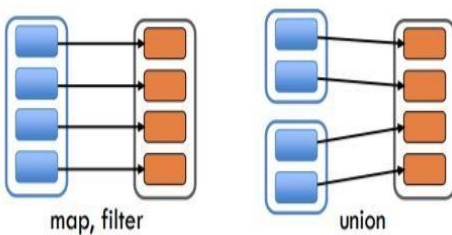


narrow

wide

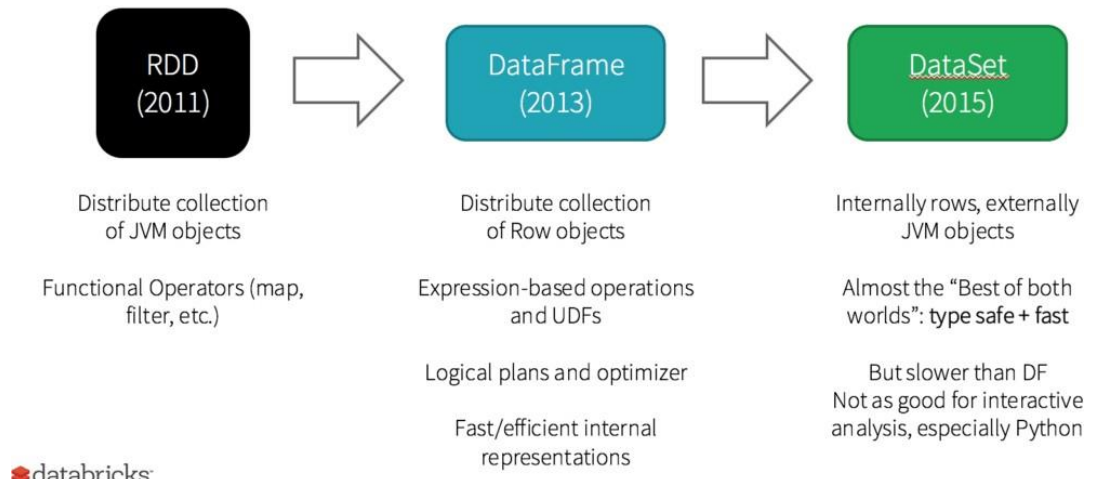
each partition of the parent RDD is used by
at most one partition of the child RDD

multiple child RDD partitions may depend
on a single parent RDD partition



9. Differences between RDD , Dataframe And DataSet?

History of Spark APIs



Differences Between RDD & DataFrame & DataSet

RDD Features:-

- 1) Distributed collection
- 2) Immutable
- 3) Fault tolerant
- 4) Lazy evaluations
- 5) Functional transformations => Transformations and Actions
- 6) Data processing formats => structured as well as unstructured data
- 7) Programming Languages supported => Java, Scala, Python and R.

DataFrame Features:

- 1) Distributed collection of Row Object
- 2) Data Processing
- 3) Optimization using catalyst optimizer
- 4) Hive Compatibility
- 5) Tungsten
- 6) Programming Languages supported => Java, Scala, Python and R.

DataSet Features:

- 1) Provides best of both RDD and Dataframe
- 2) Encoders
- 3) Programming Languages supported => Java, Scala
- 4) Type Safety

10. What are shared variables and it uses?

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.

Syntax:

```
# Create Broadcast Variable using SparkContext.broadcast (LIST)
```

```
broadcast_v = sc.broadcast([1, 2, 3])
```

```
# Print the broadcast variable value using .value
```

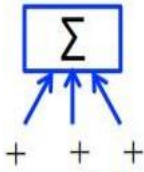
```
broadcast_v.value
```

- **Broadcast variables** are created from a variable `v` by calling `SparkContext.broadcast(v)`.
- The broadcast variable is a wrapper around `v`, and its value can be accessed by calling the `.value` method.

```
broadcast_v.unpersist()
```

```
unpersist(self, blocking=False)
```

Delete cached copies of this broadcast on the executors. If the broadcast is used after this is called, it will need to be re-sent to each executor.



Accumulators

- Variables that can only be “added” to by associative op
- Used to efficiently implement parallel counters and sums
- Only driver can read an accumulator’s value, not tasks

```
>>> accum = sc.accumulator(0)
>>> rdd = sc.parallelize([1, 2, 3, 4])
>>> def f(x):
>>>     global accum
>>>     accum += x

>>> rdd.foreach(f)
>>> accum.value
Value: 10
```

11. how to create UDF in Pyspark? User-defined functions – Python

Python user-defined function (UDF) examples. It shows how to register UDFs, how to invoke UDFs, and caveats regarding evaluation order of subexpressions in Spark SQL.

Register a function as a UDF

Python

```
def squared(s):  
    return s * s  
spark.udf.register("squaredWithPython", squared)
```

You can optionally set the return type of your UDF. The default return type is StringType.

Python

```
from pyspark.sql.types import LongType  
def squared_typed(s):  
    return s * s  
spark.udf.register("squaredWithPython", squared_typed, LongType())
```


Call the UDF in Spark SQL

Python

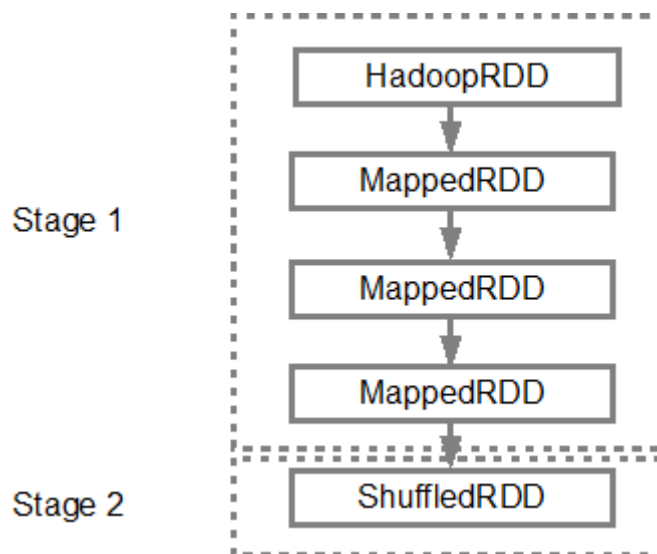
```
spark.range(1, 20).createOrReplaceTempView("test")
```

SQL

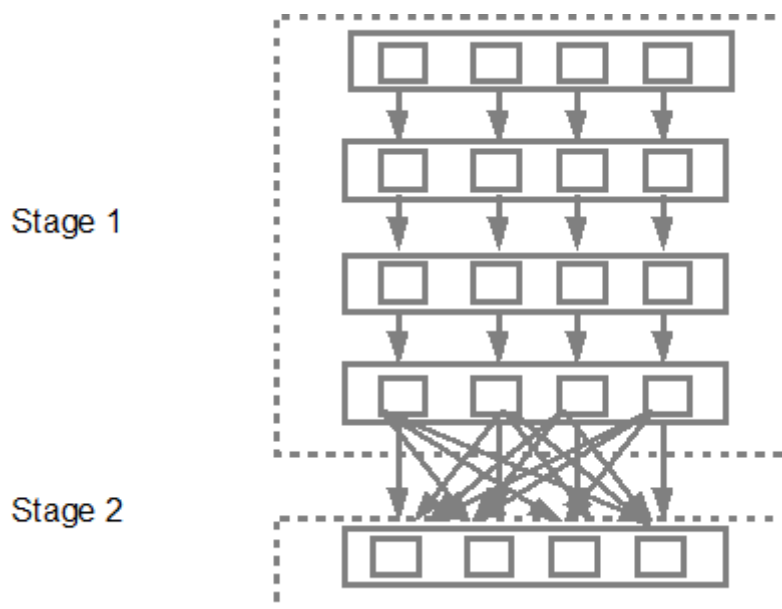
```
%sql select id, squaredWithPython(id) as id_squared from test
```

12. Explain Stages and Tasks creation in Spark?

Once the DAG is build, the Spark scheduler creates a physical execution plan. As mentioned above, the DAG scheduler splits the graph into multiple stages, the stages are created based on the transformations. The narrow transformations will be grouped (pipe-lined) together into a single stage. So for our example, Spark will create two stage execution as follows:



The DAG scheduler will then submit the stages into the task scheduler. The number of tasks submitted depends on the number of partitions present in the textFile. For example, consider we have 4 partitions in this example, then there will be 4 sets of tasks created and submitted in parallel provided there are enough slaves/cores. Below diagram illustrates this in more detail:



13. difference between coalesce and repartition

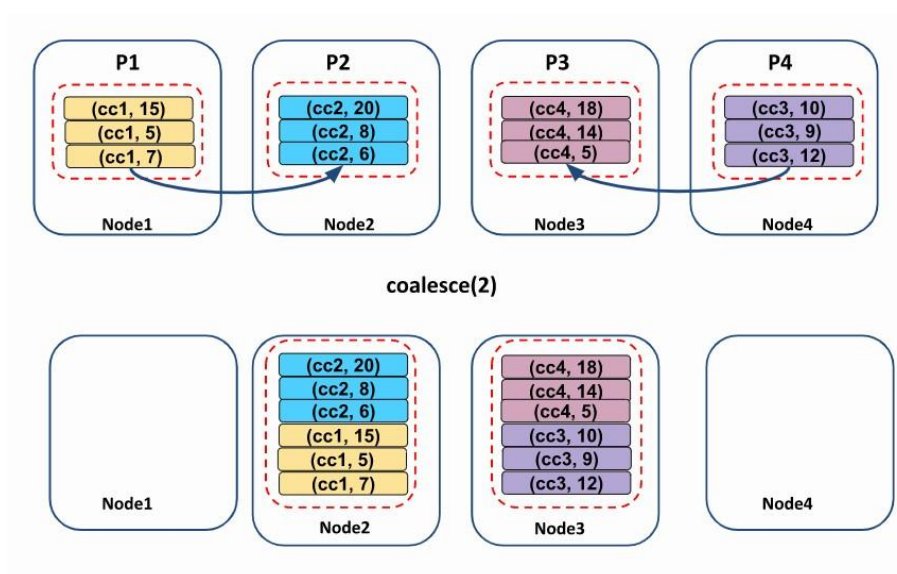
The **coalesce** method reduces the number of partitions in a DataFrame. Coalesce avoids full shuffle, instead of creating new partitions, it shuffles

the data using Hash Partitioner (Default), and adjusts into existing partitions, this means it can only decrease the number of partitions.

Unlike **repartition**, **coalesce doesn't perform a shuffle** to create the partitions. The repartition method can be used to either increase or decrease the number of partitions in a DataFrame. Repartition is a full Shuffle operation, whole data is taken out from existing partitions and equally distributed into newly formed partitions.

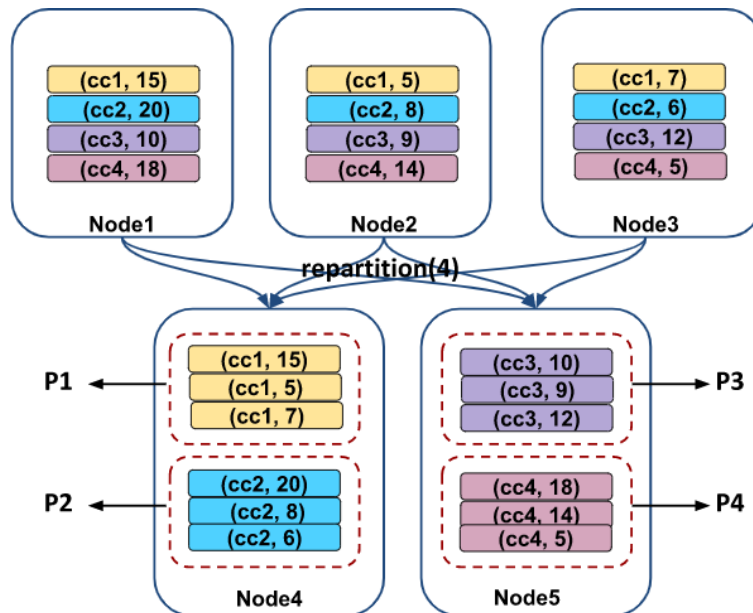
Coalesce

1. Coalesce is used to reduce the number of partitions
2. Coalesce will not trigger partition
3. Here an RDD with 4 partitions is reduced to 2 partitions (`coalesce(2)`)
4. Partition from Node1 is moved to Node2
5. Partition from Node4 is moved to Node3



Repartition

1. Repartition is used to increase the number of partitions
2. Repartition triggers shuffling
3. Here an RDD with 3 partitions is repartitioned into 4 partitions (`repartition(4)`)



Source: <https://pixipanda.com/>

14. What is data skew?

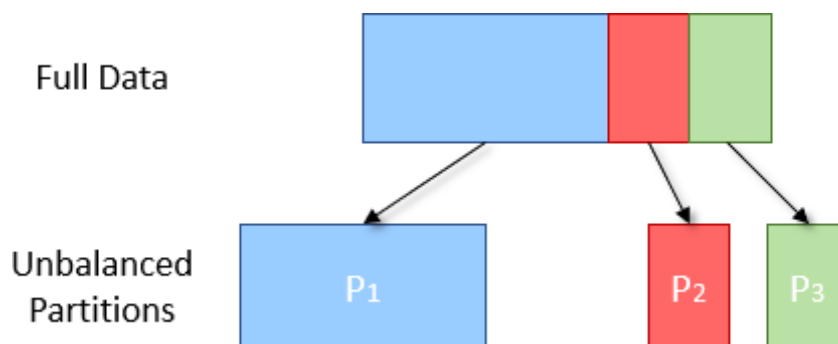
Data Skew

Data skew is a condition in which a table's data is unevenly distributed among partitions in the cluster. Data skew can severely downgrade performance of queries, especially those with joins. Joins between big tables require shuffling data and the skew can lead to an extreme imbalance of work in the cluster. It's likely that data skew is affecting a query if a query appears to be stuck finishing very few tasks (for example, the last 3 tasks out of 200).

Often the data is split into partitions based on a key, for instance the first letter of a name. If values are not evenly distributed throughout this key then more data will be placed in one partition than another. An example would be:

```
{Adam, Alex, Anja, Beth, Claire}
-> A: {Adam, Alex, Anja}
-> B: {Beth}
-> C: {Claire}
```

Here the A partition is 3 times larger than the other two, and therefore will take approximately 3 times as long to compute. As the next stage of processing cannot begin until all three partitions are evaluated, the overall results from the stage will be delayed.



15. How to fix data skew issues?

A ***skew hint*** must contain at least the name of the relation with skew. A relation is a table, view, or a subquery. All joins with this relation then use skew join optimization.

```
SQL

-- table with skew
SELECT /*+ SKEW('orders') */ * FROM orders, customers WHERE c_custId = o_custId

-- subquery with skew
SELECT /*+ SKEW('C1') */ *
  FROM (SELECT * FROM customers WHERE c_custId < 100) C1, orders
 WHERE C1.c_custId = o_custId
```

Relation and columns

There might be multiple joins on a relation and only some of them will suffer from skew. Skew join optimization has some overhead so it is better to use it only when needed. For this purpose, the *skew hint* accepts column names. Only joins with these columns use skewjoin optimization.

```
SQL

-- single column
SELECT /*+ SKEW('orders', 'o_custId') */ *
  FROM orders, customers
 WHERE o_custId = c_custId

-- multiple columns
SELECT /*+ SKEW('orders', ('o_custId', 'o_storeRegionId')) */ *
  FROM orders, customers
 WHERE o_custId = c_custId AND o_storeRegionId = c_regionId
```

Relation, columns, and skew values

You can also specify skew values in the hint. Depending on the query and data, the skewvalues might be known (for example, because they never change) or might be easy to findout. Doing this reduces the overhead of skew join optimization. Otherwise, Delta Lake detects them automatically.

SQL

```
-- single column, single skew value
SELECT /*+ SKEW('orders', 'o_custId', 0) */ *
  FROM orders, customers
 WHERE o_custId = c_custId

-- single column, multiple skew values
SELECT /*+ SKEW('orders', 'o_custId', (0, 1, 2)) */ *
  FROM orders, customers
 WHERE o_custId = c_custId

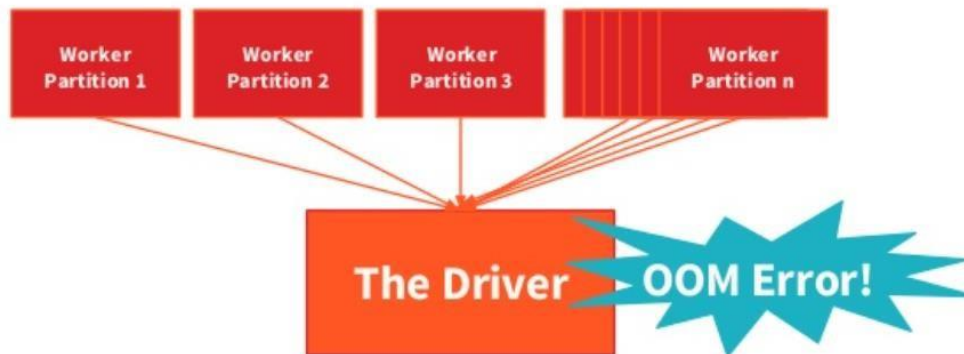
-- multiple columns, multiple skew values
SELECT /*+ SKEW('orders', ('o_custId', 'o_storeRegionId'), ((0, 1001), (1, 1002))) */ *
  FROM orders, customers
 WHERE o_custId = c_custId AND o_storeRegionId = c_regionId
```

16. What happens when use collect() action on DF or RDD?

Don't use **collect()** on large datasets. If we call collect() on large datasets it will collect all data from all workers nodes and it will send to driver node. It may cause out of memory exception. To avoid this type of errors use take() or head to identify the same data.

What happens when calling collect()

collect() sends all the partitions to the single driver



collect() on a large RDD can trigger a OOM error

Don't call collect() on a large RDD



Be cautious with all actions that may return *unbounded output*.

Option 1: Choose actions that return a bounded output per partition, such as **count()** or **take(N)**.

Option 2: Choose actions that outputs directly from the workers such as **saveAsTextFile()**.

17. what is pair rdd? When to use them?

Key/value RDDs are commonly used to perform aggregations, and often we will do some initial ETL (extract, transform, and load) to get our data into a key/value format. Key/value RDDs expose new operations (e.g., counting up reviews for each product, grouping together data with the same key, and grouping together two different RDDs).

Spark provides special operations on RDDs containing key/value pairs. These RDDs are called pair RDDs. Pair RDDs are a useful building block in many programs, as they expose operations that allow you to act on each key in parallel or regroup data across the network. For example, pair RDDs have a `reduceByKey()` method that can aggregate data separately for each key, and a `join()` method that can merge two RDDs together by grouping elements with the same key. It is common to extract fields from an RDD (representing, for instance, an event time, customer ID, or other identifier) and use those fields as keys in pair RDD operations.

Transformations on

one pair RDD `PairRDD`:

`{{(1, 2), (3, 4), (3, 6)}}`

reduceByKey(func) Combine values with the same key.

groupByKey() Group values with the same key.

mapValues(func) Apply a function to each value of a pair RDD without changing the key.

flatMapValues(func) Apply a function to each value of a pair RDD without changing the key.

keys() Return an RDD of just the keys.

values() Return an RDD of just the values

sortByKey() Return an RDD sorted by the key.

subtractByKey remove elements with a key present in the other rdd.

join : perform an inner join

between two rdd's.

rightouterjoin: perform a join

between two RDD's

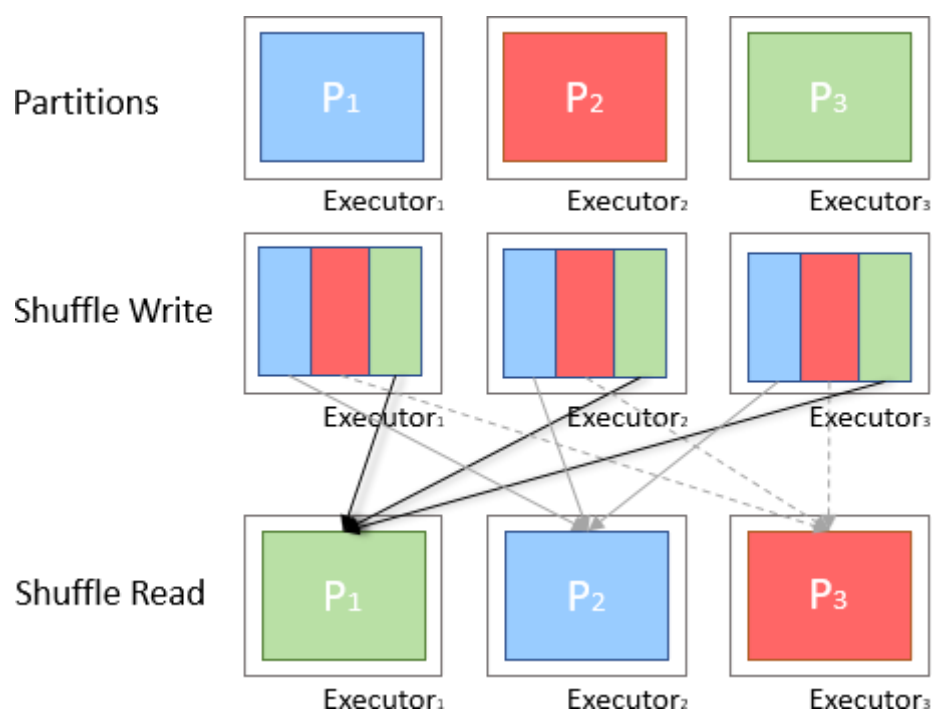
leftouterjoin: perform a join

between two rdd's

cogroup: group data from both rdd's sharing the same key.

18. What is Shuffling?

A **shuffle** occurs when data is rearranged between partitions. This is required when a transformation requires information from other partitions, such as summing all the values in a column. Spark will gather the required data from each partition and combine it into a new partition, likely on a different executor.

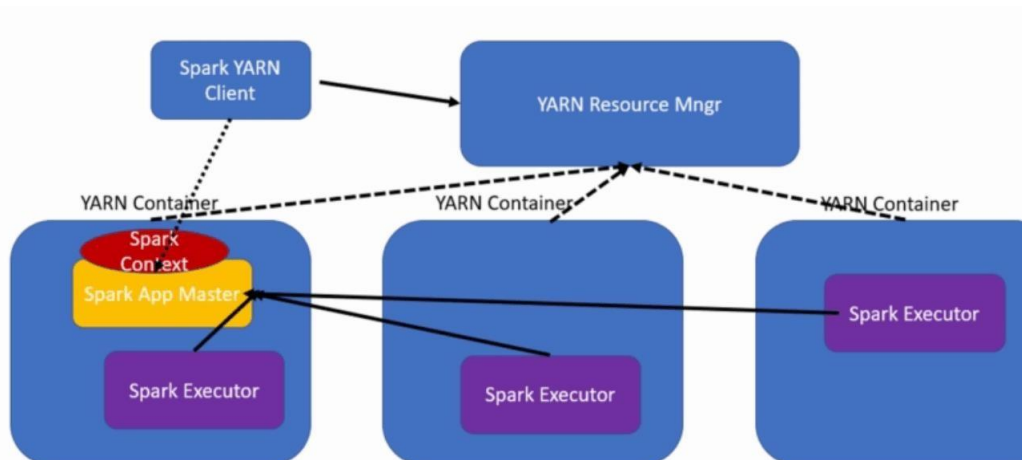


During a shuffle, data is written to disk and transferred across the network, halting Spark's ability to do processing in-memory and causing a performance bottleneck. Consequently we want to try to reduce the number of shuffles being done or reduce the amount of data being shuffled.

19. difference between cluster and client mode

Cluster Mode

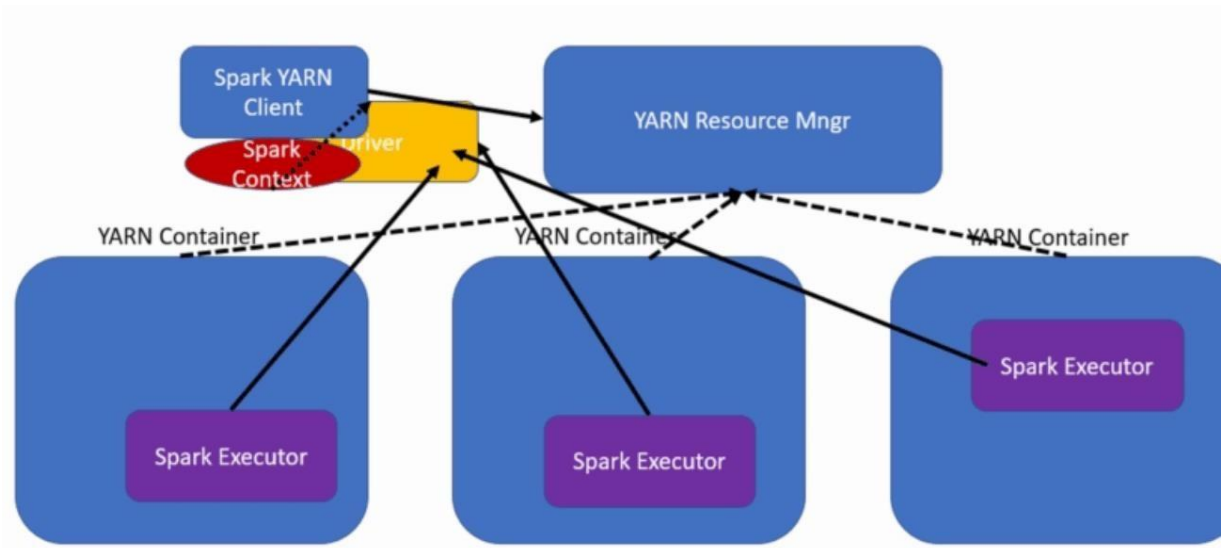
In the cluster mode, the Spark driver or spark application master will get started in any of the worker machines. So, the client who is submitting the application can submit the application and the client can go away after initiating the application or can continue with some other work. So, it works with the concept of Fire and Forgets.



The question is: when to use Cluster-Mode? If we submit an application from a machine that is far from the worker machines, for instance, submitting locally from our laptop, then it is common to use cluster mode to minimize network latency between the drivers and the executors. In any case, if the job is going to run for a long period time and we don't want to wait for the result then we can submit the job using cluster mode so once the job submitted client doesn't need to be online.

Client Mode

In the client mode, the client who is submitting the spark application will start the driver and it will maintain the spark context. So, till the particular job execution gets over, the management of the task will be done by the driver. Also, the client should be in touch with the cluster. The client will have to be online until that particular job gets completed.



In this mode, the client can keep getting the information in terms of what is the status and what are the changes happening on a particular job. So, in case if we want to keep monitoring the status of that particular job, we can submit the job in client mode. In this mode, the entire application is dependent on the Local machine since the Driver resides in here. In case of any issue in the local machine, the driver will go off. Subsequently, the entire application will go off. Hence this mode is not suitable for Production use cases. However, it is good for debugging or testing since we can throw the outputs on the driver terminal which is a Local machine.

20. What types of file format using in big data and those differences?

Big Data File Types.



Apache
orc™



Parquet

row-store

ID	Name	City	Country	Order_no
----	------	------	---------	----------



+ easy to add/modify a record

- might read in unnecessary data

column-store

ID	Name	City	Country	Order_no
----	------	------	---------	----------



+ only need to read in relevant data

- tuple writes require multiple accesses

=> suitable for read-mostly, read-intensive, large data repositories

BIG DATA FORMATS COMPARISON

	Avro	Parquet	ORC
Schema Evolution Support			
Compression			
Splitability			
Most Compatible Platforms	Kafka, Druid	Impala, Arrow Drill, Spark	Hive, Presto
Row or Column	Row	Column	Column
Read or Write	Write	Read	Read

ORC or Optimized Row Columnar file format

- ORC stands for Optimized Row Columnar (ORC) file format.
- This is a columnar file format and divided into header, body and footer.
- File Header with ORC text
- The header will always have the ORC text to let applications know what kind of files they are processing.
- Column oriented storage format.
- Schema is with the data, but as a part of footer.
- Data is stored as row groups and stripes.
- Each stripe maintains indexes and stats about data it stores.
- support for complex types including DateTime and complex and semi-structured types.
- up to 70% compression.
- indexes every 10,000 rows, which allow skipping rows.
- a significant drop in run-time execution.

Parquet

- Parquet, an open-source file format for Hadoop stores nested data structures in a flat columnar format.
- Parquet is a columnar format. Columnar formats work well where only few columns are required in query/ analysis.
- Only required columns would be fetched / read, it reduces the disk I/O.
- Parquet is well suited for data warehouse kind of solutions where aggregations are required on certain column over a huge set of data.
- Parquet provides very good compression upto 75% when used with compression formats like snappy.
- Parquet can be read and write using Avro API and Avro Schema.
- It also provides predicate pushdown, thus reducing further disk I/O cost.

Avro

- It is row major format.
- It is used for Serialization to convert objects into binary format but with some rules & that rule is AVRO Schema (JSON schema+ binary data) .
- Its primary design goal was schema evolution.
- Avro stores the schema in header of file so data is self-describing.
- Avro formatted files are splittable and compressible and hence it's a good candidate for data storage in Hadoop ecosystem.
- Schema Evolution – Schema used to read a Avro file need not be same as schema which was used to write the files. This makes it possible to add new fields.

21. what happens if a worker node is dead?

The master considers the worker to be failure if it didn't receive the heartbeat message for past 60 sec (according to spark.worker.timeout). In that case the partition is assigned to another worker (remember partitioned RDD can be reconstructed even if its lost).

1. An RDD is an immutable, deterministically re-computable, distributed dataset. Each RDD remembers the lineage of deterministic operations that were used on a fault-tolerant input dataset to create it.
2. If any partition of an RDD is lost due to a worker node failure, then that partition can be re-computed from the original fault-tolerant dataset using the lineage of operations.

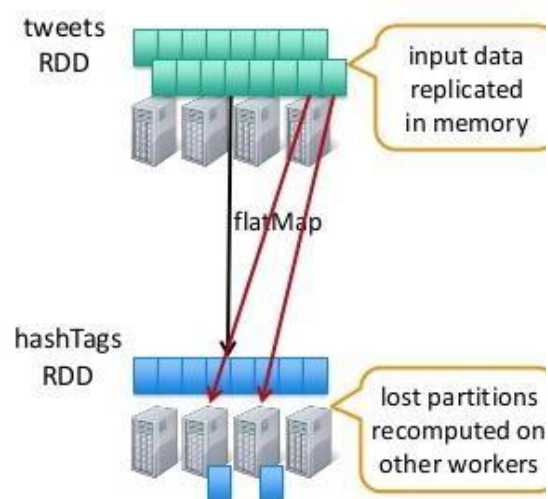
3. Assuming that all of the RDD transformations are deterministic, the data in the finaltransformed RDD will always be the same irrespective of failures in the Spark cluster.

Spark operates on data in fault-tolerant file systems like HDFS or S3. Hence, all of the RDDs generated from the fault-tolerant data are also fault-tolerant. However, this is not the case for Spark Streaming as the data in most cases is received over the network (except when FileStream is used). To achieve the same fault-tolerance properties for all of the generated RDDs, the received data is replicated among multiple Spark executors in workernodes in the cluster (default replication factor is 2). This leads to two kinds of data in the system that need to be recovered in the event of failures:

1. **Data received and replicated** - This data survives failure of a single worker node as a copy of it exists on one of the other nodes.
2. **Data received but buffered for replication** - Since this is not replicated, the only way to recover this data is to get it again from the source.
Furthermore, there are two kinds of failures that we should be concerned about:
3. **Failure of a Worker Node** - Any of the worker nodes running executors can fail, and all in-memory data on those nodes will be lost. If any receivers were running on failed nodes, then their buffered data will be lost.
4. **Failure of the Driver Node** - If the driver node running the Spark Streaming application fails, then obviously the SparkContext is lost, and all executors with their in-memory data are lost.
With this basic knowledge, let us understand the fault-tolerance semantics of Spark Streaming.

Fault-tolerance

- RDDs remember the operations that created them
- Batches of input data are replicated in memory for fault-tolerance
- Data lost due to worker failure, can be recomputed from replicated input data
- Therefore, all transformed data is fault-tolerant



22. Difference between `reduceByKey()` and `groupByKey()`?

Both of these transformations operate on pair RDDs. A pair RDD is an RDD where each element is a pair tuple (key, value). For example, `sc.parallelize([('a', 1), ('a', 2), ('b', 1)])` would create a pair RDD where the keys are 'a', 'a', 'b' and the values are 1, 2, 1.

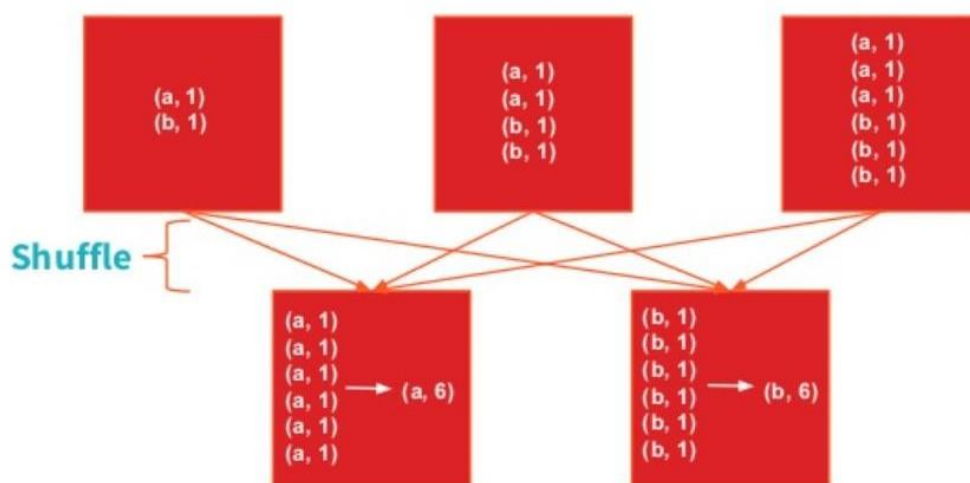
The `reduceByKey()` transformation gathers together pairs that have the same key and applies a function to two associated values at a time. `reduceByKey()` operates by applying the function first within each partition on a per-key basis and then across the partitions.

While both the `groupByKey()` and `reduceByKey()` transformations can often be used to solve the same problem and will produce the same answer, the `reduceByKey()` transformation works much better for large distributed datasets. This is because Spark knows it can combine output with a common key on each

partition before shuffling (redistributing) the data across nodes. Only use `groupByKey()` if the operation would not benefit from reducing the data before the shuffle occurs.

groupByKey() is just to group your dataset based on a key. It will result in data shuffling when RDD is not already partitioned.

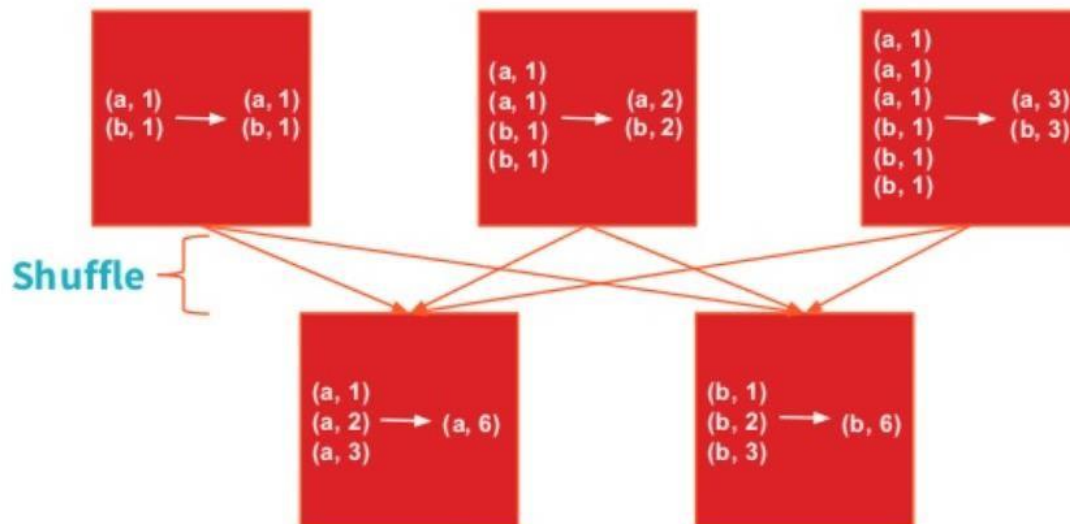
GroupByKey: Shuffle Step



With GroupByKey, all the data is wastefully sent over the network and collected on the reduce workers.

reduceByKey() is something like grouping + aggregation. We can say `reduceByKey()` equivalent to `dataset.group(...).reduce(...)`. It will shuffle less data unlike `groupByKey()`.

ReduceByKey: Shuffle Step



With ReduceByKey, data is combined so each partition outputs at most one value for each key to send over the network.

Here are more functions to prefer over groupByKey:

- `combineByKey` can be used when you are combining elements but your return type differs from your input value type.
- `foldByKey` merges the values for each key using an associative function and a neutral "zero value".

?

Prefer `ReduceByKey` over `GroupByKey`

Caveat: Not all problems that can be solved by *groupByKey* can be calculated with *reduceByKey*.

`ReduceByKey` requires combining all your values into another value with the exact same type.

`reduceByKey`, `aggregateByKey`, `foldByKey`, and `combineByKey`, preferred over `groupByKey`

23. Lazy evaluation in Spark and its benefits? Lazy Evaluation:

1. Laziness means not computing transformation till it's need
2. Once, any action is performed then the actual computation starts
3. A DAG (Directed acyclic graph) will be created for the tasks
4. Catalyst Engine is used to optimize the tasks & queries
5. It helps reduce the number of passes

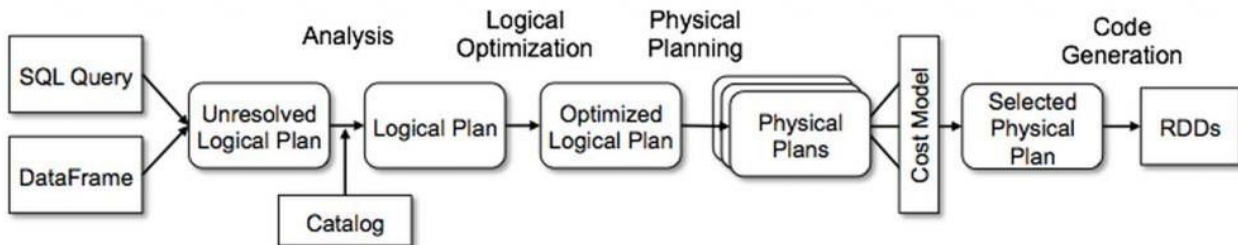
24. What is Catalyst Optimizer And Explain End to End Process?

Spark SQL is one of the most technically involved components of Apache Spark. It powers both SQL queries and the DataFrame API. At the core of Spark SQL is the Catalyst optimizer, which leverages advanced programming language features (e.g. Scala's pattern matching and quasiquotes) in a novel way to build an extensible query optimizer.

Catalyst is based on functional programming constructs in Scala and designed with these key two purposes:

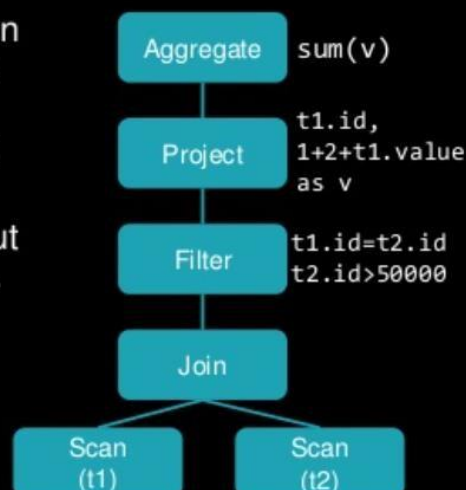
- ❑ Easily add new optimization techniques and features to Spark SQL
- ❑ Enable external developers to extend the optimizer (e.g. adding data source specific rules, support for new data types, etc.)

As well, Catalyst supports both rule-based and cost-based optimization.



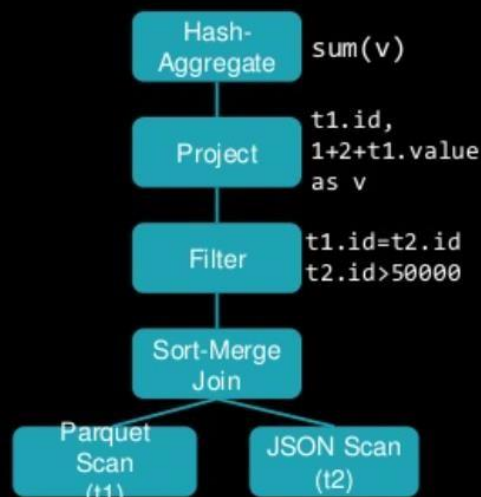
Logical Plan

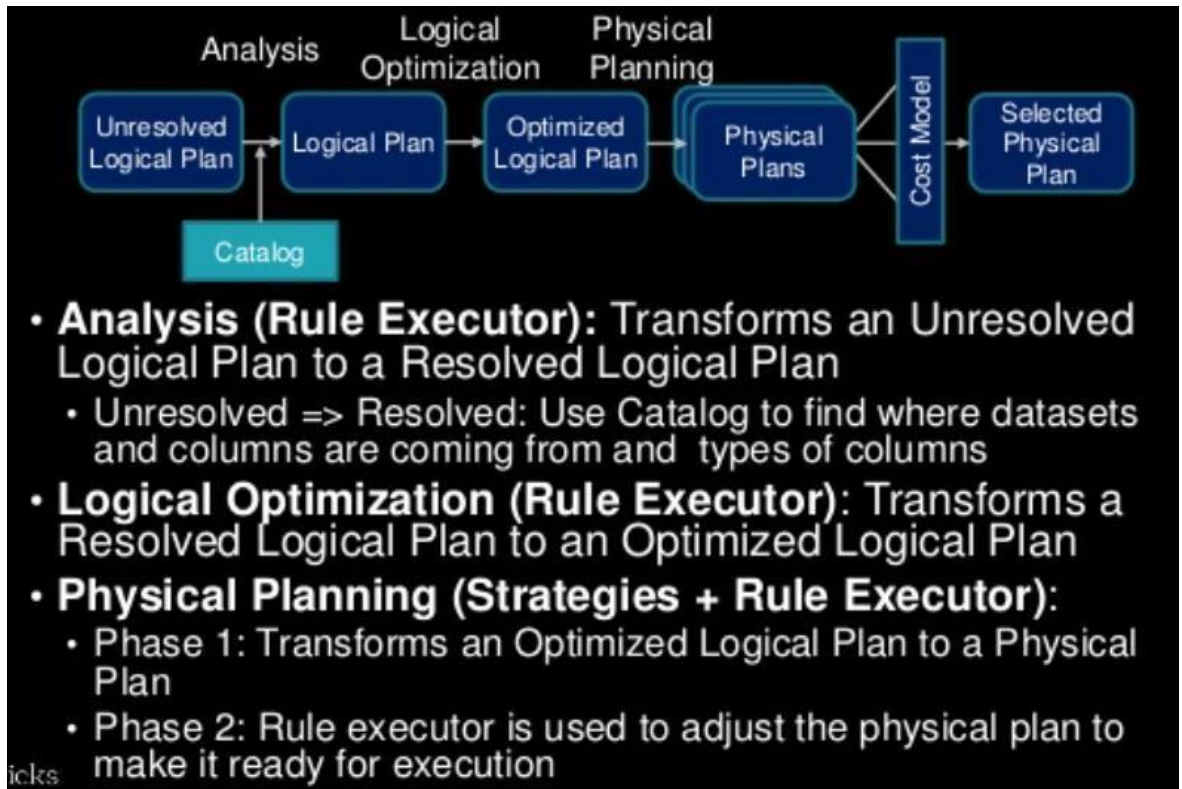
- A Logical Plan describes computation on datasets **without** defining how to conduct the computation
- **output**: a list of attributes generated by this Logical Plan, e.g. [id, v]
- **constraints**: a set of invariants about the rows generated by this plan, e.g. $t2.id > 50000$
- **statistics**: size of the plan in rows/bytes. Per column stats (min/max/ndv/nulls).



Physical Plan

- A Physical Plan describes computation on datasets with specific definitions on how to conduct the computation
- A Physical Plan is executable





25. Difference between ShuffledHashJoin And BroadcastHashjoin?

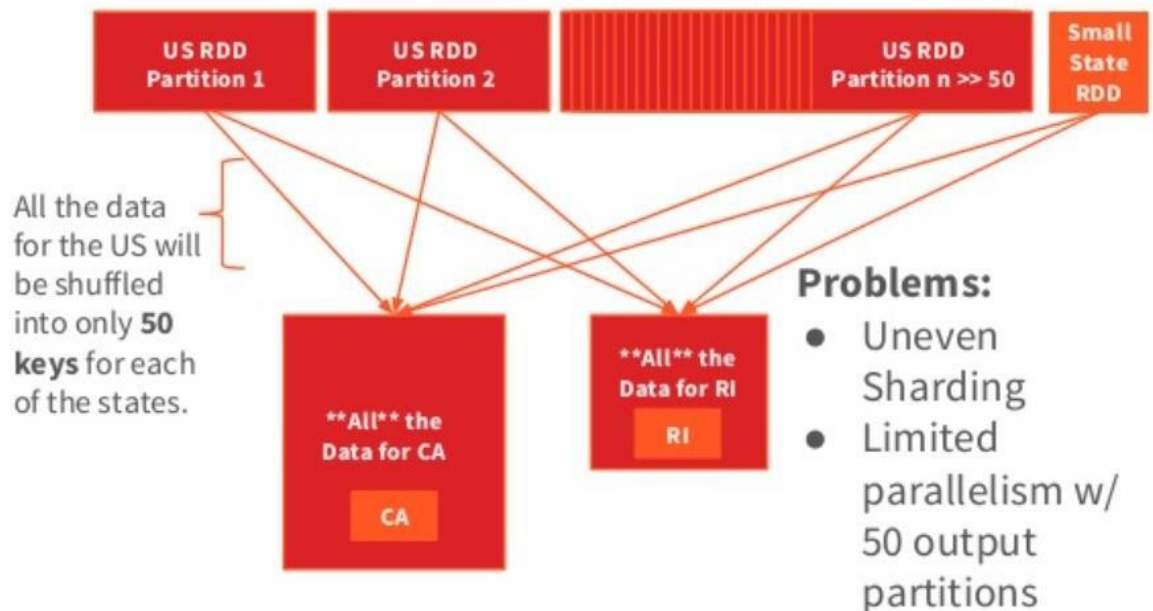
Shuffle Hash Join (SHJ)

When the side of the table is relatively small, we choose to broadcast it out to avoid shuffle, improve performance. But because the broadcast table is first to collect to the driver segment, and then distributed to each executor redundant, so when the table is relatively large, the use of broadcast progress will be the driver and executor side caused greater pressure.

But because Spark is a distributed computing engine, you can partition the large number of data can be divided into n smaller data sets for parallel computing. This idea is applied to the Join is Shuffle Hash Join. Spark SQL will be larger table join and rule, the first table is divided into n partitions,

and then the corresponding data in the two tables were Hash Join,so that is to a certain extent, the same time, Reducing the pressure on the side of the driverbroadcast side, but also reduce the executor to take the entire broadcast by the memory ofthe table.

ShuffledHashJoin



Even a larger Spark cluster will not solve these problems!

Broadcast HashJoin (BHJ)

As we all know, in the database common model (such as star model or snowflake model), the table is generally divided into two types: **fact** table and **dimension** table. Dimension tables (small tables) generally refer to fixed, less variable tables, such as contacts, items, etc., the general data is limited. The fact table generally records water, such as sales lists, etc., usually with the growth of time constantly expanding.... means large tables

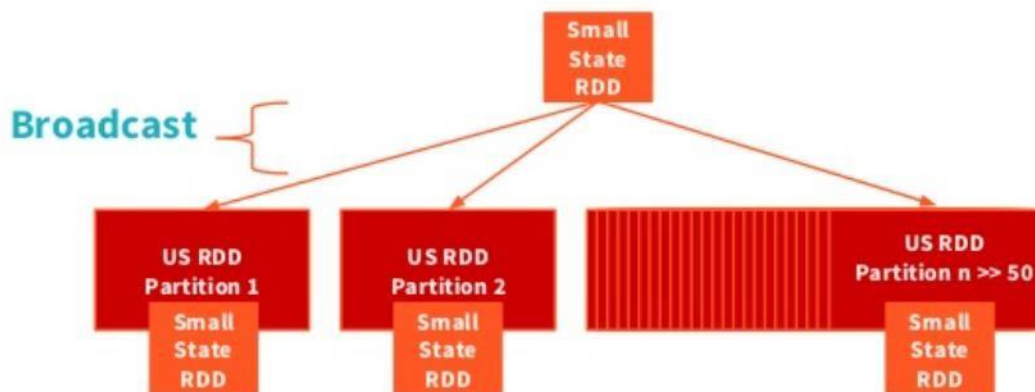
Because the Join operation is the two tables in the same key value of the record to connect, in SparkSQL, the two tables to do Join the most direct way is based on the key partition, and then in each partition the key value of the same record Come out to do the connection operation. But this will inevitably involve shuffle, and shuffle in Spark is a more time-consuming operation, we should try to design Spark application to avoid a lot of shuffle.

When the dimension table and the fact table for the Join operation, in order to avoid shuffle, we can be limited size of the dimension table of all the data distributed to each node for the fact table to use. executor all the data

stored in the dimension table, to a certain extent, sacrifice the space, in exchange for shuffle operation a lot of time-consuming, which in SparkSQL called Broadcast Join

BroadcastHashJoin











Solution: Broadcast the Small RDD to all worker nodes.



Parallelism of the large RDD is maintained (n output partitions), and shuffle is not even needed.

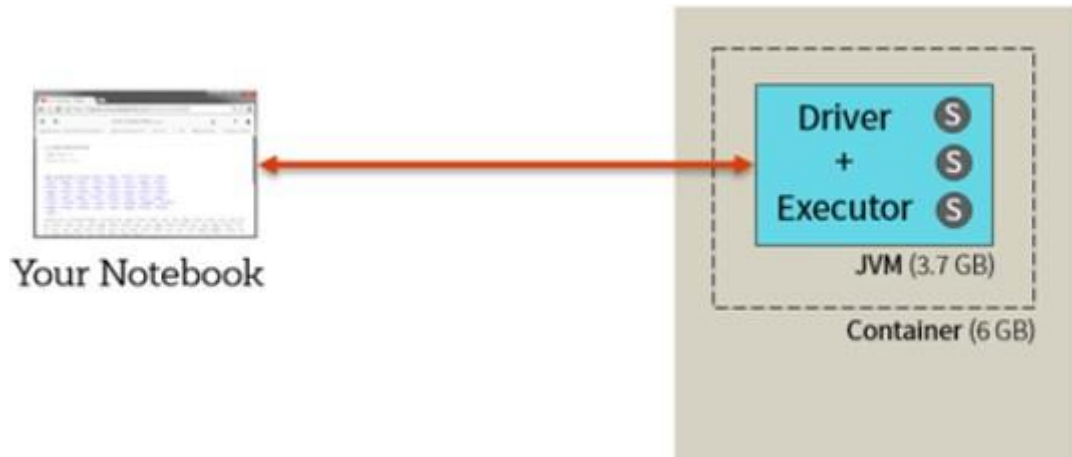
26. How many modes are there for spark execution?

WAYS TO RUN SPARK

-  - Local 
-   - Standalone Scheduler 
-   - YARN 
-  - Mesos 

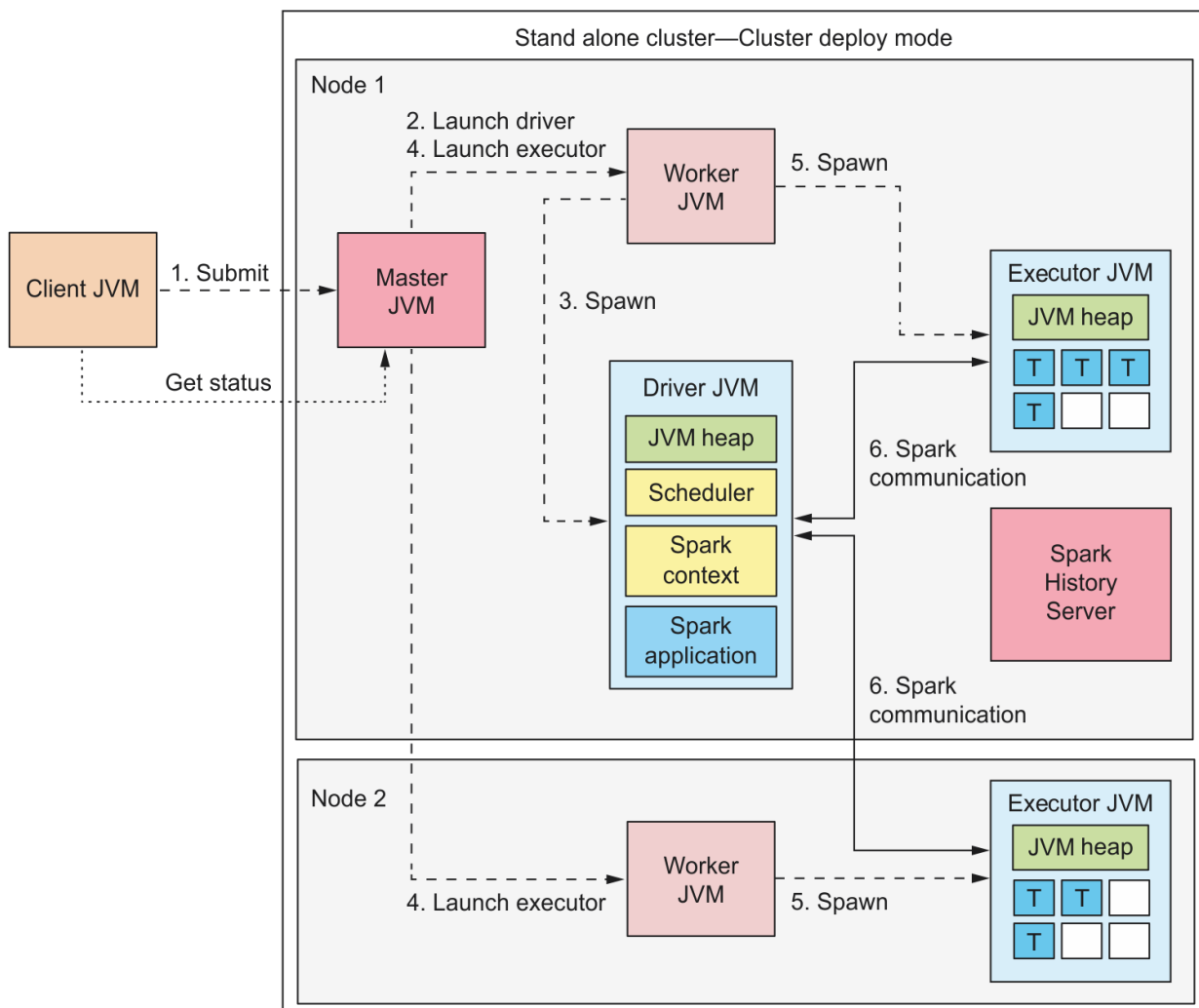
Local Mode

Spark local mode is special case of standalone cluster mode in a way that the
_master &
_worker run on same machine.



Standalone Cluster Mode

As the name suggests, it's a standalone cluster with only spark specific components. It doesn't have any dependencies on hadoop components and **Spark driver** acts as clustermanager.



Hadoop YARN/ Mesos

Apache Spark runs on Mesos or YARN (Yet another Resource Navigator, one of the key features in the second-generation Hadoop) without any root-access or pre-installation. It integrates Spark on top Hadoop stack that is already present on the system.

