Difference betn Apache Spark & Apache Hadoop

Apache Hadoop

② No built in iterative mode

③ for data processing Hadoop uses batch engine

④

⑤ Doesn't have any abstraction.

⑦ Needs Higher Storage.

Apache Spark

① Spark easy to program

② has iterative built in mode

③ Uses streaming, machine learning & batch processing.

④ Spark completes job processing jobs about 10 to 100 times faster.

⑤ Uses Abstraction called RDD. which makes spark feature rich.

⑥ Uses low latency.

⑦ needs Higher RAM

Spark Architecture.

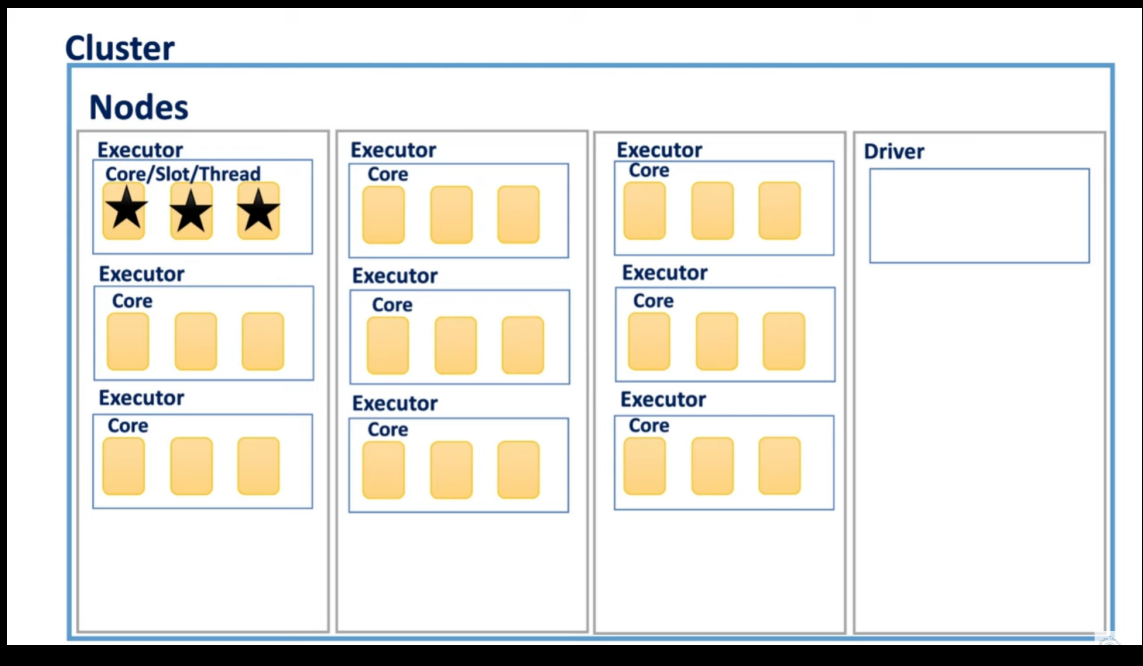
Driver - Is responsible to give instruction runs on his own JVM.

Executors - Who to execution.

Dataset - chunks of data.

job - Number of stages

cluster - collection of one or more node.



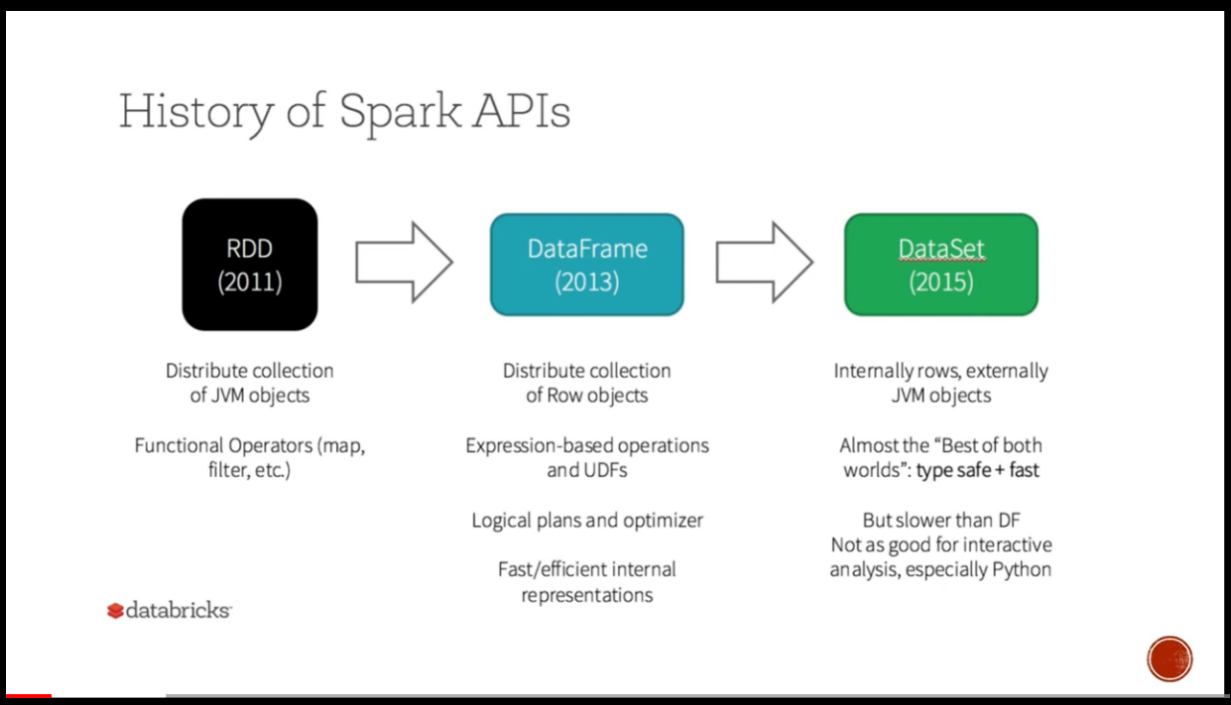
Spark API :-

three types of API

RDD (2011)

Dataframe (2013)

Dataset (2015)



RDD -

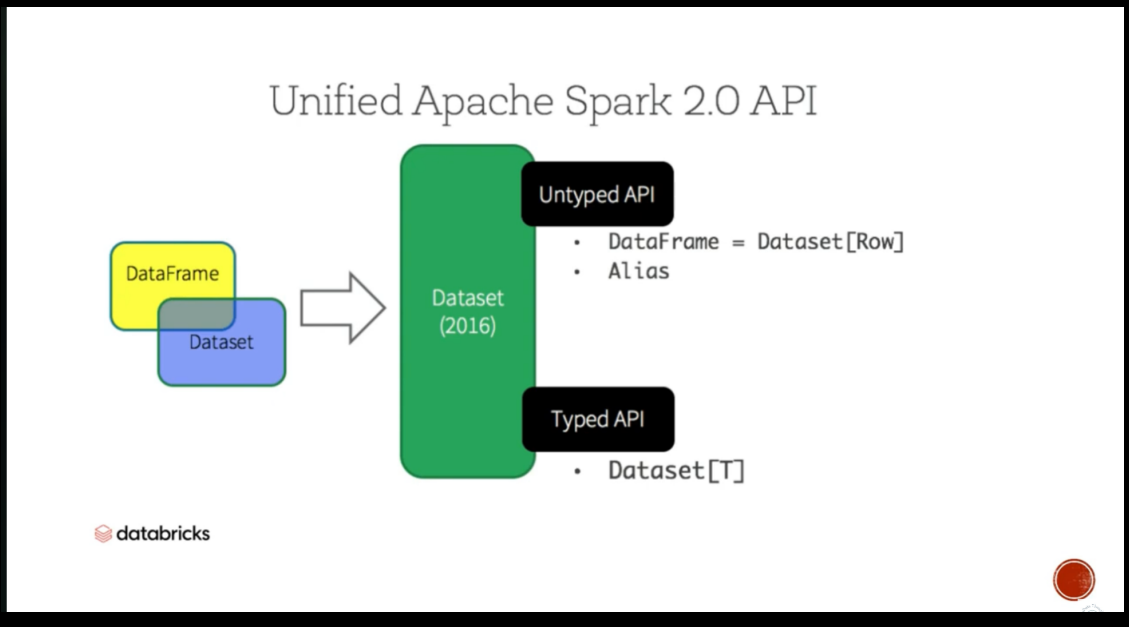
Resilient distributed dataset

RDD

- Resilient distributed dataset
- Fundamental abstraction on which spark was built
- Low level API
- Offers control and flexibility – 'How '

Resilient
Immutable
Compile time type safe
Lazy Evaluation – materialized in a lazy manner after a action happens
Doesn't infer schema
When we do transformation, we get one RDD from another like a lineage

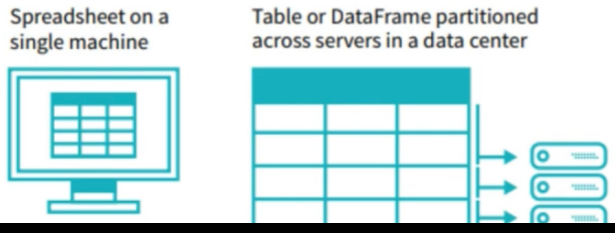
RDD -> T -> RDD-> T->RDD



untyped - python
typed - Java.

DATAFRAME

- High level API
- Immutable distributed collection of data.
- Data is organized into named columns, like a table in a relational database
- DataFrame allows developers to impose a structure onto a distributed collection of data,

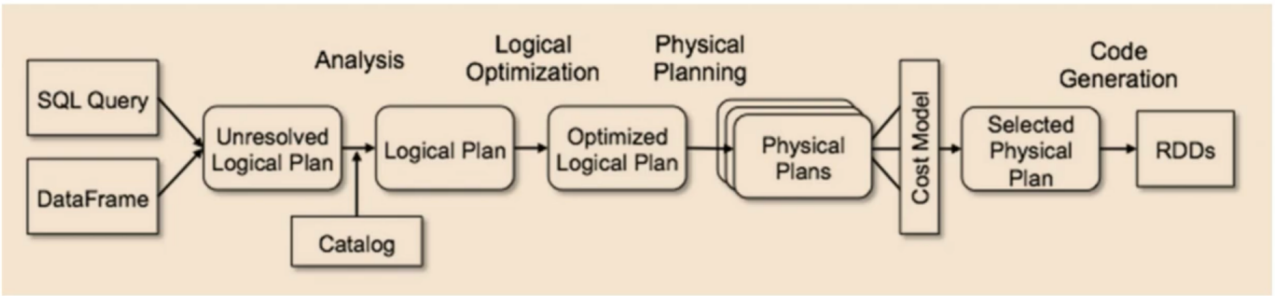


DATASET

- High level API
- The Apache Spark Dataset API provides a type-safe, object-oriented programming interface.
- Datasets provide compile-time type safety—which means that production applications can be checked for errors before they are run

WHEN TO USE DATAFRAME AND DATASET

- High-level abstractions, and domain specific APIs
- Processing demands high-level expressions, filters, maps, aggregation, averages, sum, SQL queries, columnar access and use of lambda functions on semi-structured data,
- Higher degree of type-safety at compile time - Dataset
- Unification and simplification of APIs across Spark Libraries
- R user - DataFrames.
- Python user - DataFrames

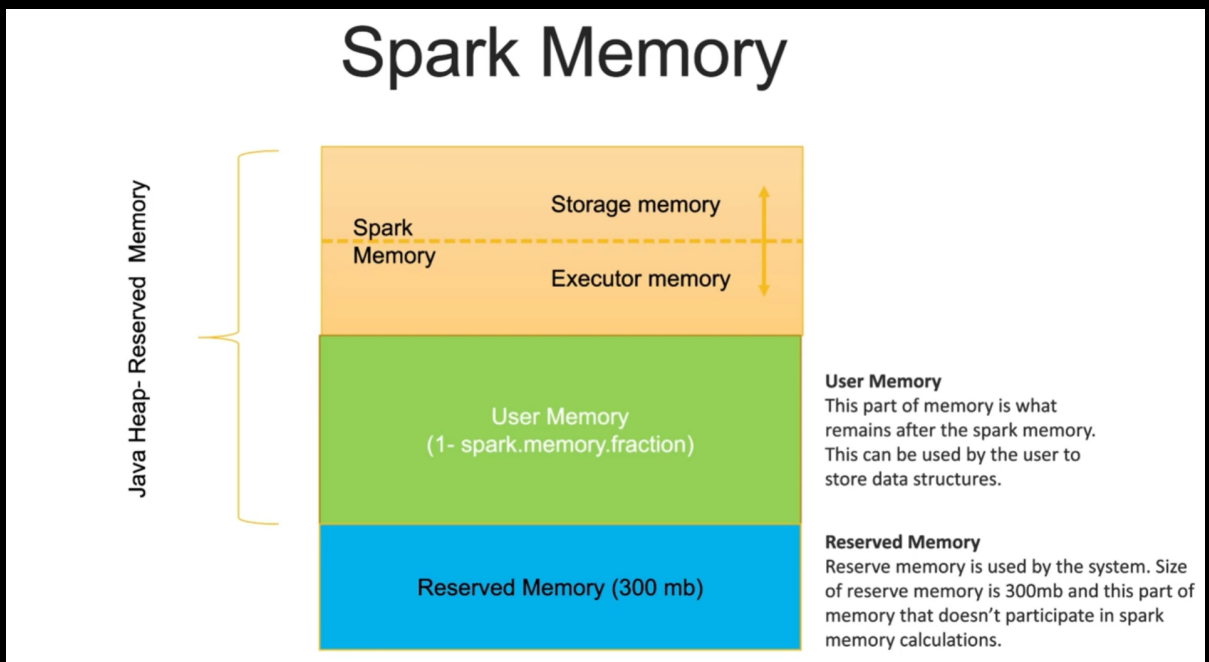


Memory management

why ?

— memory critical for spark

more = Higher performance



Memory Types

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

Execution Memory

Storage memory refers to that used for caching and propagating internal data across the cluster

Storage Memory

Execution and storage share a unified region

previously memory was static memory allocation.

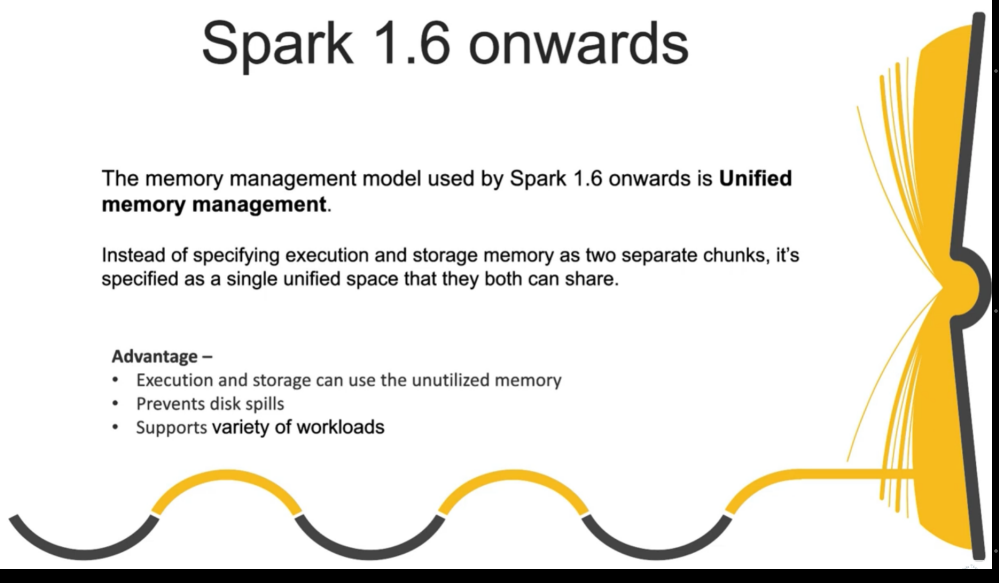
Spark 1.6 onwards

The memory management model used by Spark 1.6 onwards is **Unified memory management**.

Instead of specifying execution and storage memory as two separate chunks, it's specified as a single unified space that they both can share.

Advantage –

- Execution and storage can use the unutilized memory
- Prevents disk spills
- Supports variety of workloads



Nowadays unified memory management is used.

— It specifies execution and storage as single chunk and only specified reserved memory.

Determining Memory Consumption

Memory consumption of a dataset

Create an RDD, put it into cache, and look at the "Storage" page in the web UI. The page will tell how much memory the RDD is occupying.

Memory consumption of a particular object

Use SizeEstimator estimate method

We can see the consumption of dataset
— create a RDD put that into cache.
go into web UI and check the
consumption on the storage.

Tuning Considerations

Find out the Memory consumption of a dataset/object

Serialize RDD storage

Tune Garbage collection

Level of Parallelism

Broadcasting

RDD from slides

Resilient Distributed Dataset

- collection of in-memory objects divided into multiple parts which resides on different machine

Basically Architecture of Spark

- RDD exposes two types of operations

* Transform operations :-
  - map
  - filter
  - reduceByKey

* Action operations :-
  - collect
  - count

RDD objects are immutable

Deploy a Spark Program

- following event happens
  - driver process started on the same JVM
    - → Driver will initiate the object to keep track of all configuration of Appl^n
    - → Driver process spawns a set of Executors via a cluster manager.

SparkObject initiated by Spark Program
↓
Driver started
↓
Driver initiates the object
↓
Driver spawns a set of Executors

Driver process is also responsible for job scheduling
- FIFO scheduler
- fair scheduler

Driver process job
- Construct the DAG from Spark job
- generate the execution plan outlining the Spark Tasks to be executed.
- Invoke the task scheduler to schedule tasks to run on selected executors.

---

following commands for **configuration of cluster**

num-executors → Number of executors

executor-cores → Number of cores used per executor

executor-memory → Memory size used per executor

One of the following **cluster Manager** is used according to the value.
- Spark Native cluster manager (master = spark://host:port)
- YARN (master = yarn)
- Mesos (master = mesos://host:port)

Spark RDD programming

Setup of Deployment Environment.
- create a Maven project with intellij
  - Add Spark dependencies to POM.xml
    - Spark core

- Spark SQL
- Spark ML
- etc.

Main types of RDD objects
- Java RDD
- JavaPair RDD

1. Java RDD
   - Collection of partitioned distributed objects
   - Created via SparkContext object
   - Applicable Transform operations
     - map
     - filter
     - distinct

- JavaPair RDD
  - Collection of partitioned distributed key-value pair objects
  - created by invoking the mapToPair operation on a JavaRDD object
  - Applicable Transform operations
    - reduceByKey
    - groupByKey
    - sortByKey
    - join

Common Action operations on both RDD types
- Return a RDD object to the driver program
  - count(), collect(), first(), reduce()
  - foreach()

- Persist a RDD object in a file
  - saveAsTextFile()

Common Transform operations on both RDD types

- repartition(), coalesce()

repartition() ← To increase decrease the no. of partition in RDD
coalesce() ← To only decrease the no. of partitions in efficient way

very expensive operations as they shuffle the data across many partitions hence to minimize them.

spark.sparkContext.parallelize (range(0,20),6) ← RDD partitions
rdd1.repartition(4) from 6 to 4

Programming pattern

- SparkContext.parallelize()
- SparkContext.textFile()

RDD automatically recover from node failures.

Spark supports two types of shared variables

① broadcast variables
↳ Used to cache a value in memory on all nodes

② accumulators
↳ type of variables that can only be added to such as counters and sums.

Parallelized collections are created by calling SparkContext's parallelize method on an existing iterables or collection in your driver program.

RDD Operations :-

Two types of operations:

① Transformations
– Which creates a new dataset from an existing one

② Actions

- Which returns a value to the driver program after running a computation on the dataset.

Example :-

Map is transformation that passes element through a function and returns a new RDD representing the results.

All transformation on Spark are lazy

- GroupBykey()
- Join()
- ReduceBykey()
- SortBykey()

→ They just remember the transformation applied to some base dataset. transformation only computed when action requires results to be returned to the driver.

By default, each transformed RDD may Recomputed each time you run an action on it.

However, You can also persist an RDD in memory using the persist (or cache) method, in which case Spark will keep the elements around on the cluster for much faster access the next time you query it.

Shuffle :- Spark mechanism for re-distributing data so that's grouped differently across partitions.

from youtube

* In Spark we can decide what size of container should be also we can choose How many container we want.

- Spark Dynamic execution allowcation for allowcation for data and it will automatically kill itself after completion.
- Data locality is not 100% guranteed.
- Block size in Hadoop 128MB.

In Spark More partition = more speed
Executor is JVM 1 cores is required to manage it.
More partition = more processing power you can get.

* In Spark If you request 10GB memory it will only give you 54%.
because 10% goes for system calls
remaining goes for JVM & garbase collection
So you only get the remaining memory 6GBs

- After tranformation for example we filtered the data then we run coalesce() it wont show the data because it's lazy transformation which means it wont show it without using Action.
Once processing done Pipeline will be empty.

DAG - Directed acyclic graph.

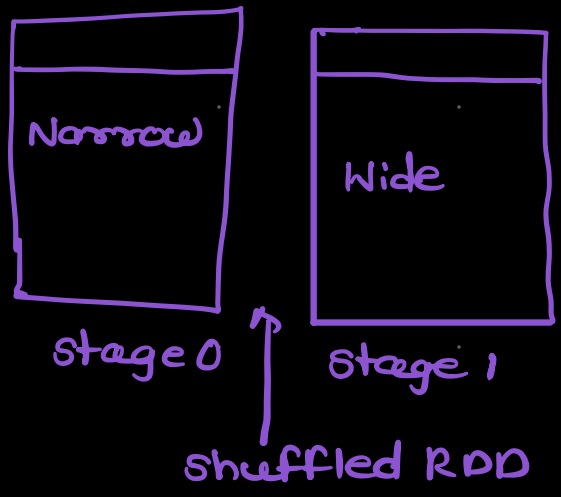
SparkSQL is much more optimized = Dataframe.


Spark is in memory processing doesn't mean it will use ram everytime. It will use ram if its available.

* Job, Stages and Task

- Transformation has two types
  ① Narrow (Where is in no communication needed betn two task)
  ex. filter, map.

  ② Wide (Where communication betn two task required
  ex. join()
  GroupByKey()



after successful compilition executors will be deleted automaticcally but driver will still be there. Which takes around 1GB of resources so they have to be stopped manually.

pyspark2 --master local[3]

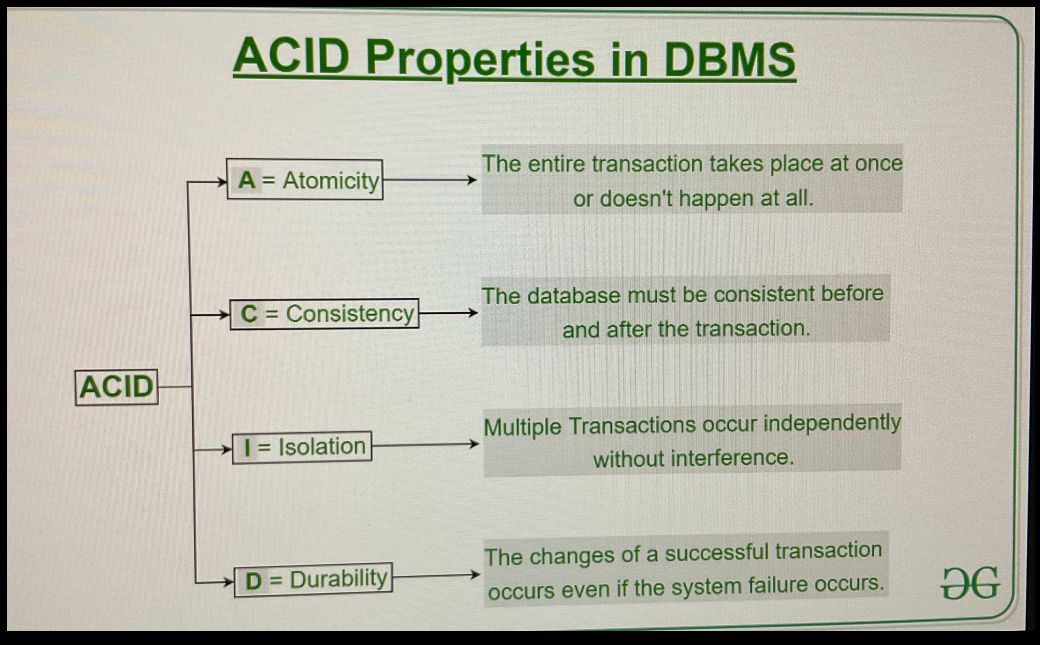
[3] = Number of cores to use
[*] = use whatever cores available.

ACID

properties & MCQ

Why to use?

In order to maintain consistancy in database, before and after the transaction.



Automacity - either the transaction takes place at once doesn't

40% ① 20 words match the column

- map reduce
- HDFS
- Spark
- Sharding
- mongodb
- Shard key

30% ② 2 parts

1. – Which Spark method used for partition
   - output txt what changed
   repartition

2. Which method for k:V pair.
   - map?
   - flatmap
   - map design?

– Explain the same?

30% ③ Numerical

Map reduce based

- 50 nodes 256GB

Yarn container How many?

$$\#\text{ of containers} = \frac{\min(2 \times \text{cores}, 1.8 \times \text{Disk}(\text{total available RAM}))}{\text{min\_container size}}$$

$=$

Reserved memory = Reserved for stack memory + Reserved for HBase

ex. 12 CPU cores, 48GB RAM & 12 disks

Reserved memory = 6 GB for system + 8GB for HBase

Min Container Size = 2 GB

$$= \min\left[12, 1.8 \times 12, \left(\frac{48}{2}\right)\right]$$

RAM = 48 - 6 (removing reserved)

$$\min\left(2 \times 12, 1.8 \times 12, \left(\frac{48-6}{2}\right)\right)$$

$= \min(24, 21.6, 21)$

$= 21$?

RAM-per container ?

RAM-per container = Max (min_container_size, (total available RAM) / containers)

$$= \text{Max}\left(2, \left(\frac{48-6}{21}\right)\right.$$

$$= \max(2, 2) = 2$$

Container = 2 x cores, 1.8 x disk, $\frac{\text{RAM-reserved}}{\text{min container}}$

$$= \frac{48-6}{2}$$

(21) ✓

RAM per container $\frac{48-6}{21}$

# of containers required ?

50 datanodes

256 GB

= 50 x 256 GB

= 12.5 TB

2GB container size we will require

$\frac{256}{2} =$

256GB size will require containers?

8/16/32

for Blaze engine

$(n*2)+3$

$=(50*2)+3$

103 container require

could be wrong

Just theory from 2016

When you ask for Yarn yarn.scheduler.

minimum Allocation-mb make sure to ask for exact no. for ex. 4GB if you round that off to 4.5GB yarn will give the closest number which is 8GB.

offheap Java — 10 - 25%. for example garbage collection.

— So in total you need the requested on Heap Java memory plus 10~25%. for offheap otherwise YARN will simply shoot down the container due to

Launcher is biger than the memory limit.
for this container.
Container could be 4-8-16-32
if you request anything smaller
or anything offnumber you get next closest.

→ repartitions(), coalesce()

output after operation

from local = 5

parallelize = 6

Textfile = 10

we have 10 text files

using spark.Contextspark.parallelize(Range(0,20),6)
Which will distribute RDD into 6 partitions

p1
p2
p3
p4
p5
p6

using repartition()

→ rdd1.repartition(4)

with using this command output file will be reduced to 4 files

P1
P2
P3
P4

} This will be done in shuffle manner which is an expensive to do.

→ impact of saveastext file → sheffeld output

P1: 1 2 3
P2: 4 5 6
P3: 7 8 9
P4: 10 11 12
P5: 13 14 15
P5: 16 17

impact of repartition(4)

P1: 1 8 9 12 15
P2: 2 6 10 14
P3: 3 5 11 16
P4: 4 7 13 17

→ RDD coalesce()

Is used to only reduce the no. of partitions This is optimized or improved version of repartition()

- P1:
  P2:
  P3:
  P4:

  P5:
  P6:

What coalesce() do here is combine any files together based on requirment

for Here:- coalesce(4)

This will combine any two partitions into one to reduce from 6 to 4

= rdd1.coalesce(4)

P1:
P2:
P4: → P3+P4
P5: P5+P6

final verdict

repartition() — increase decrease + shuffle
coalesce() → decrease but in indexed

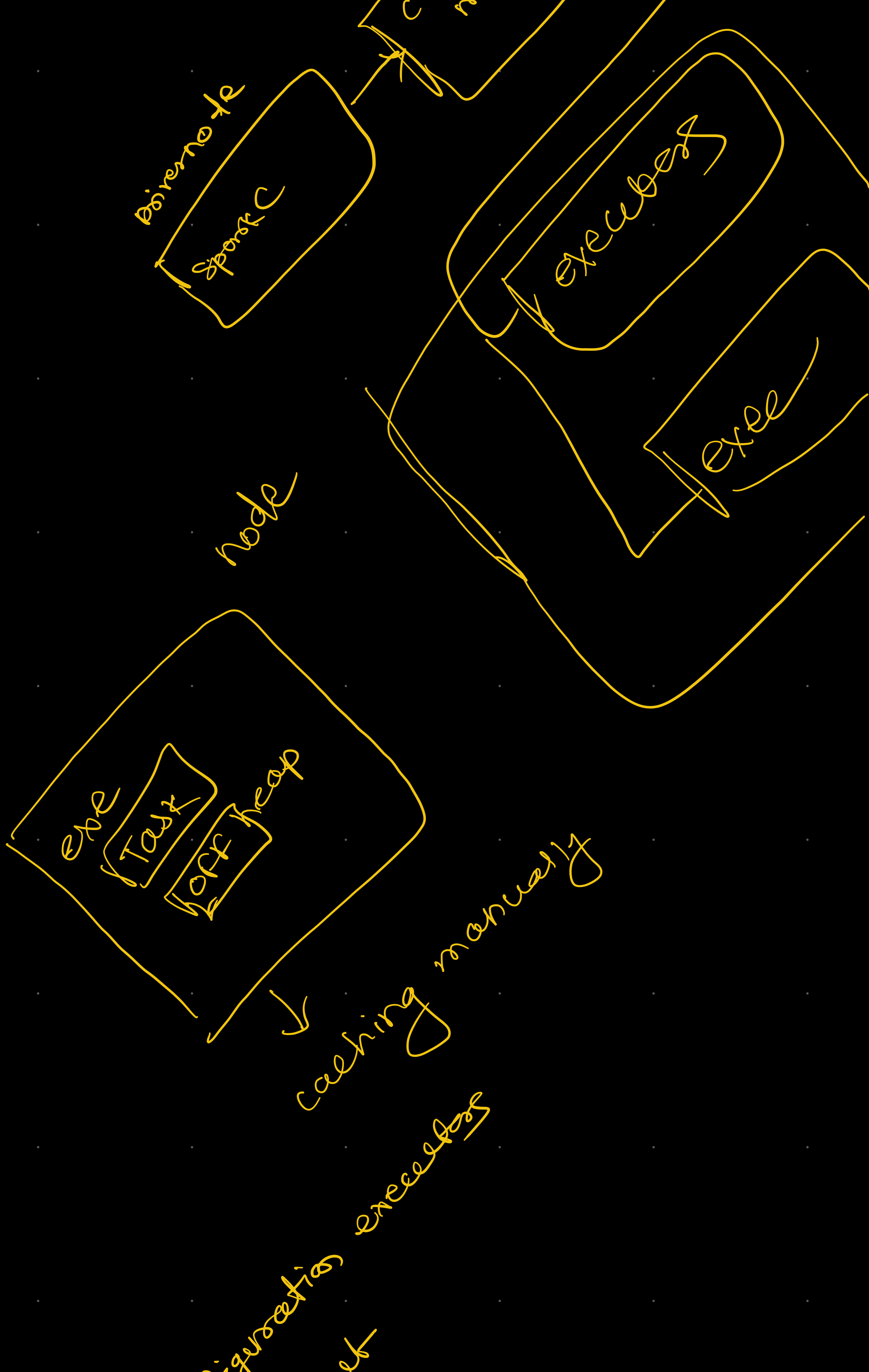
textfile() → Read multiple files & output is single RDDfile(string)

Wholetextfiles() → Read multiple & o/p is single RDD file (Tuples (string, string)

→ emptyRDD()

To create emptyRDD() function.

Tuesd
we ne


drivernode
sparkC
executors
exec
node
exe
Task
off heap
caching manually
executors

Spark config

① num-executors

② executor-cores

rep(4)

~~Sheeple~~ shuffle file

P1: 01

P2: 2

P3:

P4:

P

mapToPair() ×

K:V

reduce

JavaPairRDD ×

by key

K:V pair

shuffle when
repartition
coalesce
groupByKey
reduceByKey(_)
cogroup(_)
join
countByKey

persistence
— cache
persist