Spark - all and nothing

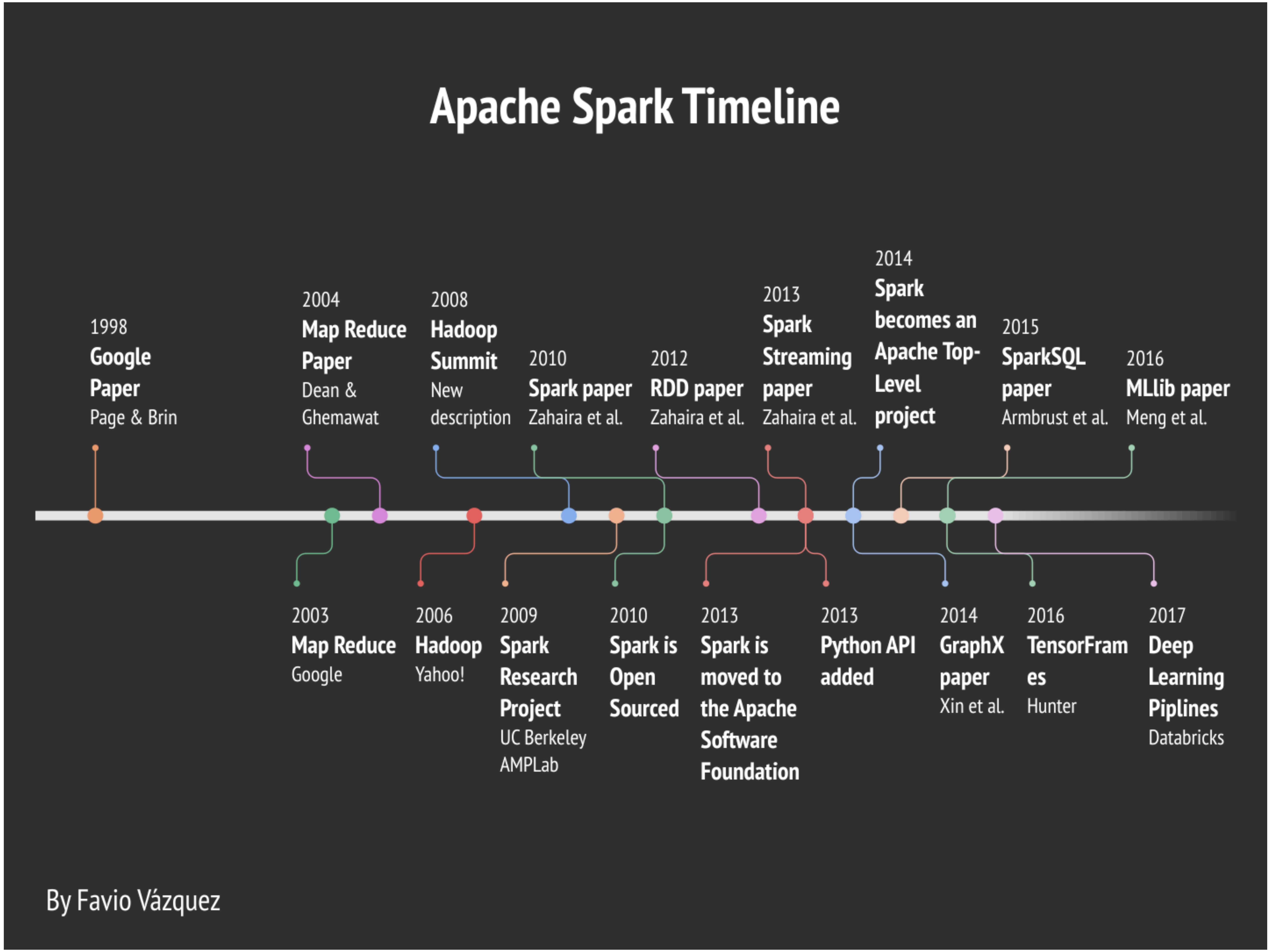
Quick overview with practical use-cases and examples

II & YB -Nov 8, 2022

About us

- Igor Ivanov: I’ve spent more than 14 years in IT. I’ve started from ground up and went through different career and professional challenges. My current goal is to build up strong, highly skilled and talented teams of people to create products with a high business value in data world. LinkedIn
- Yuriy Bukovskiy: Seasoned professional developer with more than 20 years of experience in coding and data engineering.

How it all started


Apache Spark Timeline
1998 Google Paper Page & Brin
2003 Map Reduce Google
2004 Map Reduce Paper Dean & Ghemawat
2006 Hadoop Yahoo!
2008 Hadoop Summit New description
2009 Spark Research Project UC Berkeley AMPLab
2010 Spark paper Zahaira et al.
2010 Spark is Open Sourced
2012 RDD paper Zahaira et al.
2013 Spark is moved to the Apache Software Foundation
2013 Spark Streaming paper Zahaira et al.
2013 Python API added
2014 Spark becomes an Apache Top-Level project
2014 GraphX paper Xin et al.
2015 SparkSQL paper Armbrust et al.
2016 TensorFrames Hunter
2016 MLlib paper Meng et al.
2017 Deep Learning Piplines Databricks
By Favio Vázquez

From Hadoop MR to Apache Spark

Apache Hadoop – key components:

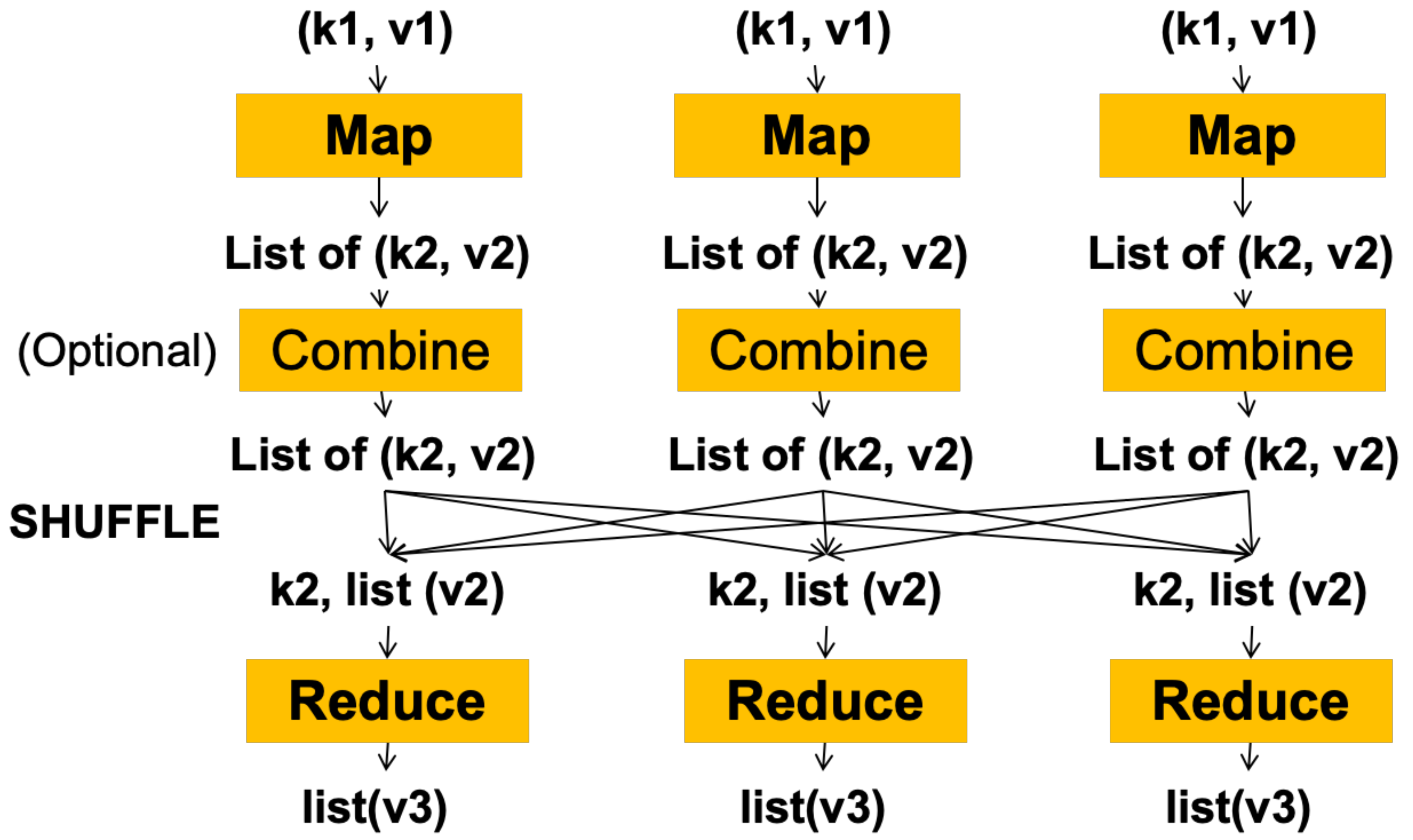
- (Storage Component) Hadoop Distributed File System (HDFS): A distributed file system that provides high-throughput access
- Many other data storage approaches also in use
- E.G., Apache Cassandra, Apache Hbase, Apache Accumulo (NSA-contributed)
- (Scheduling) Hadoop YARN: A framework for job scheduling and cluster resource management
- (Processing) Hadoop MapReduce (MR2): A YARN-based system for parallel processing of large data sets
- Other execution engines increasingly in use, e.g., Spark

Note:

All of these key components are OSS under Apache 2.0 license

Related: Ambari, Avro, Cassandra, Chukwa, Hbase, Hive, Mahout, Pig, Tez, Zookeeper

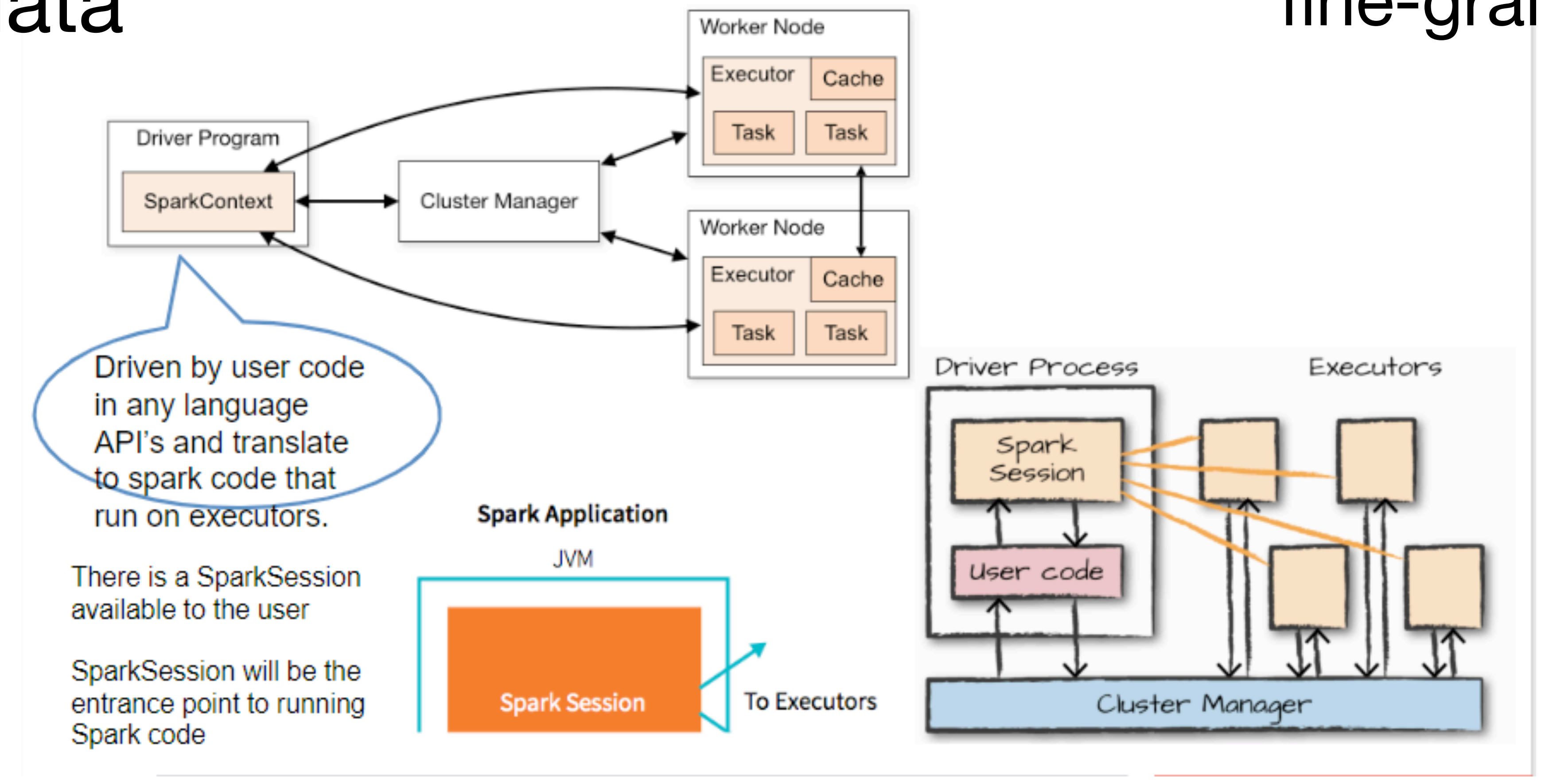
MapReduce process figure:



Apache Spark

- Processing engine; instead of just “map” and “reduce”, defines a large set of operations (transformations & actions)
- Operations can be arbitrarily combined in any order
- Open source software
- **Supports Java, Scala and Python**
- Original key construct: Resilient Distributed Dataset (RDD)
- More recently added: DataFrames & DataSets
- Different APIs for aggregate data

- **Resilient Distributed Dataset (RDD) – key Spark construct**
- RDDs represent data or transformations on data
- RDDs can be created from Hadoop InputFormats (such as HDFS files), “parallelize()” datasets, or by transforming other RDDs (you can stack RDDs)
- Actions can be applied to RDDs; actions force calculations and return values
- Lazy evaluation: Nothing computed until an action requires it
- RDDs are best suited for applications that apply the same operation to all elements of a dataset
- Less suitable for applications that make asynchronous fine-grained updates to shared state



Driven by user code in any language API’s and translate to spark code that run on executors.

There is a SparkSession available to the user

SparkSession will be the entrance point to running Spark code

Apache Spark: Dataset&DataFrame and RDD

- Spark has three sets of APIs—RDDs, DataFrames, and Datasets
- RDD – In Spark 1.0 release, “lower level”
- DataFrames – Introduced in Spark 1.3 release
- Dataset – Introduced in Spark 1.6 release
- Each with pros/cons/limitations

.

- Spark SQL
- Spark Streaming – stream processing of live datastreams
- MLlib - machine learning
- GraphX – graph manipulation
- extends Spark RDD with Graph abstraction: a directed multigraph with properties attached to each vertex and edge.

.

- DataFrame: Unlike an RDD, data organized into named columns, e.g. a table in a relational database.
- Imposes a structure onto a distributed collection of data, allowing higher-level abstraction
- Dataset: Extension of DataFrame API which provides type-safe, object-oriented programming interface (compile-time error detection)
- Both built on Spark SQL engine & use Catalyst to generate optimized logical and physical query plan; both can be converted to an RDD

.

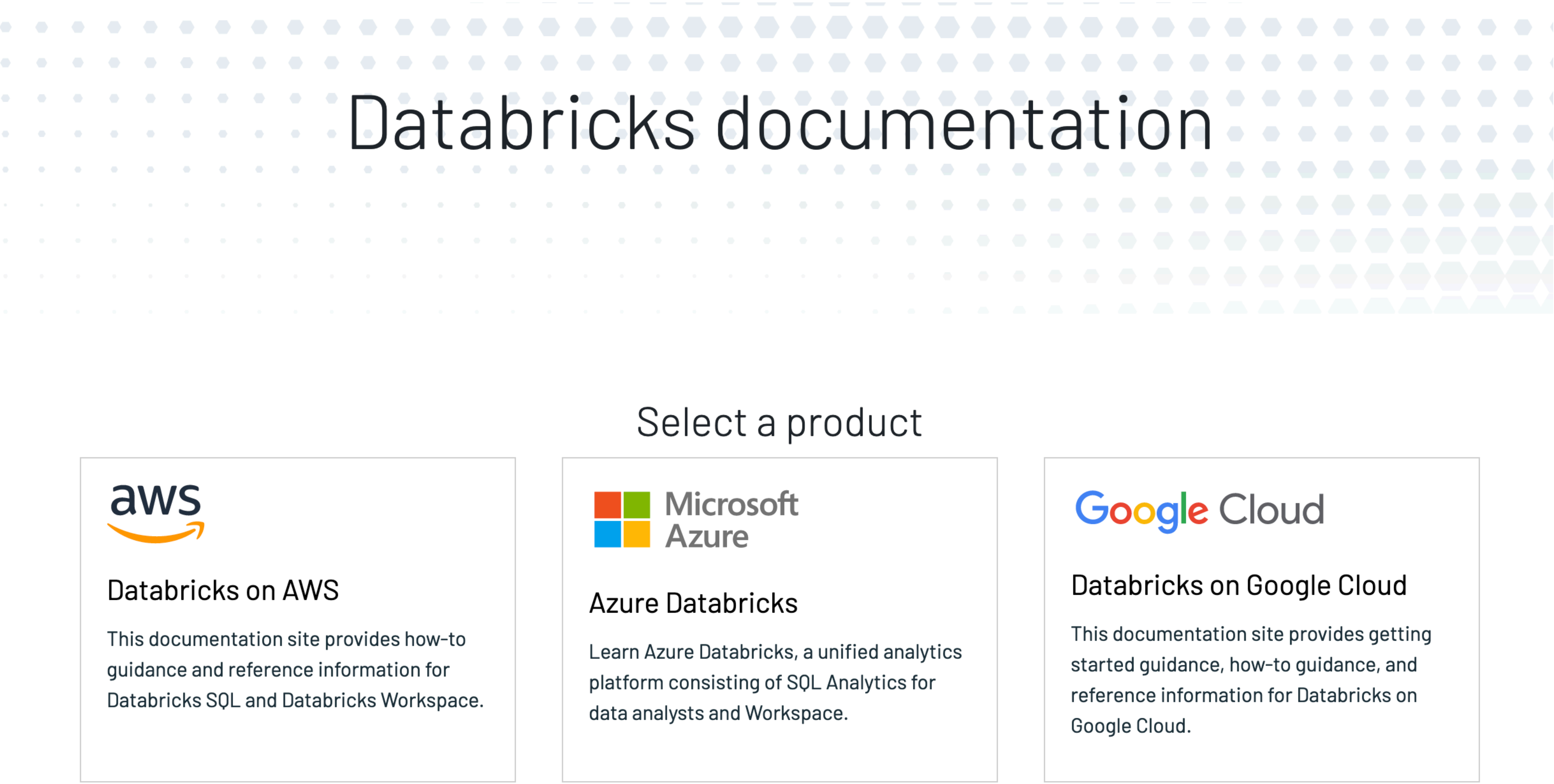
- Python & R don’t have compile-time type safety checks, so only support DataFrame
- Error detection only at runtime
- Java & Scala support compile-time type safety checks, so support both DataSet and DataFrame
- Dataset APIs are all expressed as lambda functions and JVM typed objects
- any mismatch of typed-parameters will be detected at compile time.

.

https://data-flair.training/blogs/apache-spark-rdd-vs-dataframe-vs-dataset/
https://databricks.com/blog/2016/07/14/a-tale-of-three-apache-spark-apis-rdds-dataframes-and-datasets.html

Examples and use cases by YB

Apache Spark and Databricks



Databricks documentation

Select a product

aws

Databricks on AWS

This documentation site provides how-to guidance and reference information for Databricks SQL and Databricks Workspace.

Microsoft Azure

Azure Databricks

Learn Azure Databricks, a unified analytics platform consisting of SQL Analytics for data analysts and Workspace.

Google Cloud

Databricks on Google Cloud

This documentation site provides getting started guidance, how-to guidance, and reference information for Databricks on Google Cloud.

https://databricks.com/documentation

Apache Spark Optimization (example)

Spark Supports the following clusters:

- Local Cluster
- Standalone
- Hadoop YARN
- Apache Mesos
- Kubernetes

Use --master flag to select the cluster

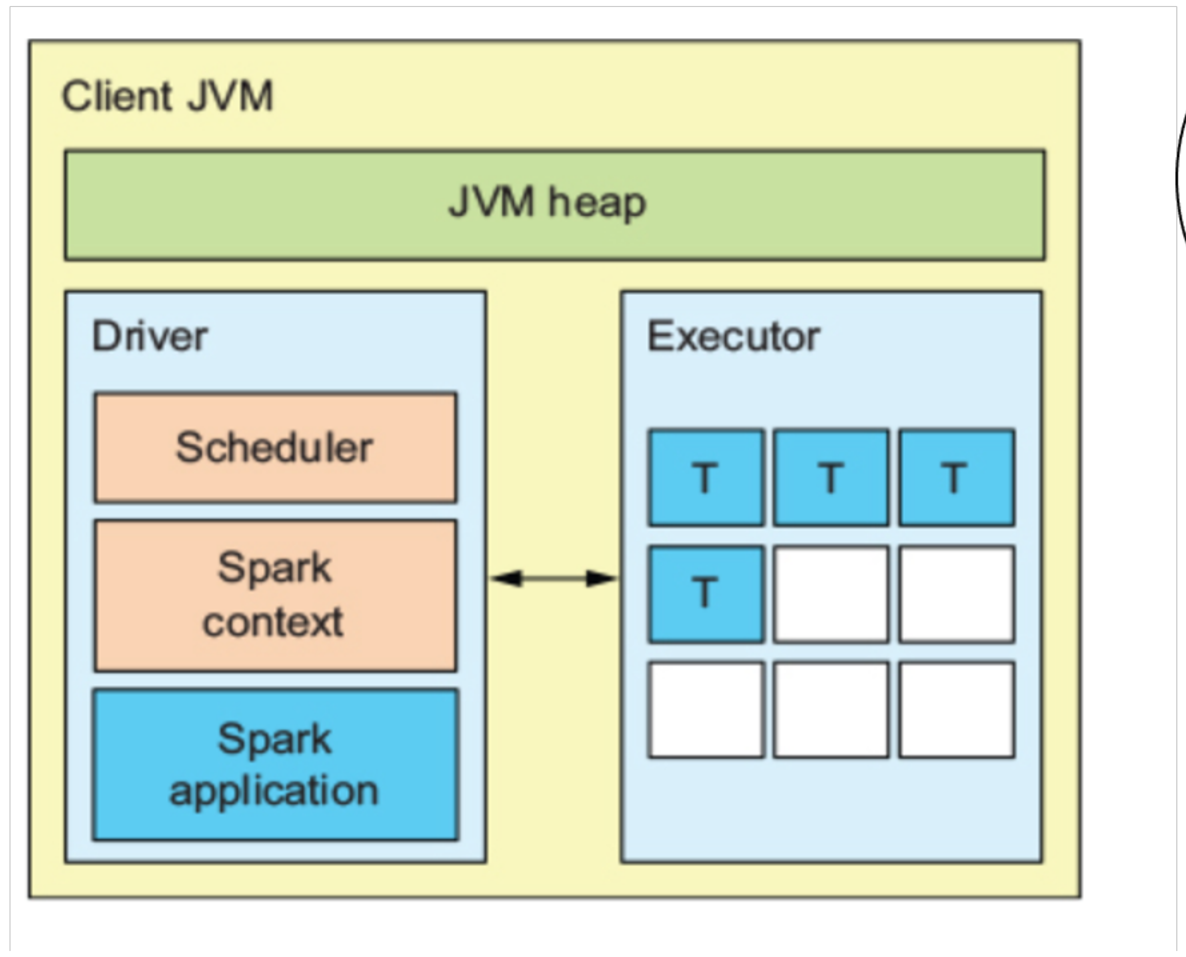
Use --master yarn for YARN

Use --master local(local[*], local[n]) to run locally
(But do you want to run locally?)

What about if you forget the –master flag?

Local mode

➢ Spark-submit/shell use --*master local(local[*], local[n])* to run locally OR if there is no –master flag.

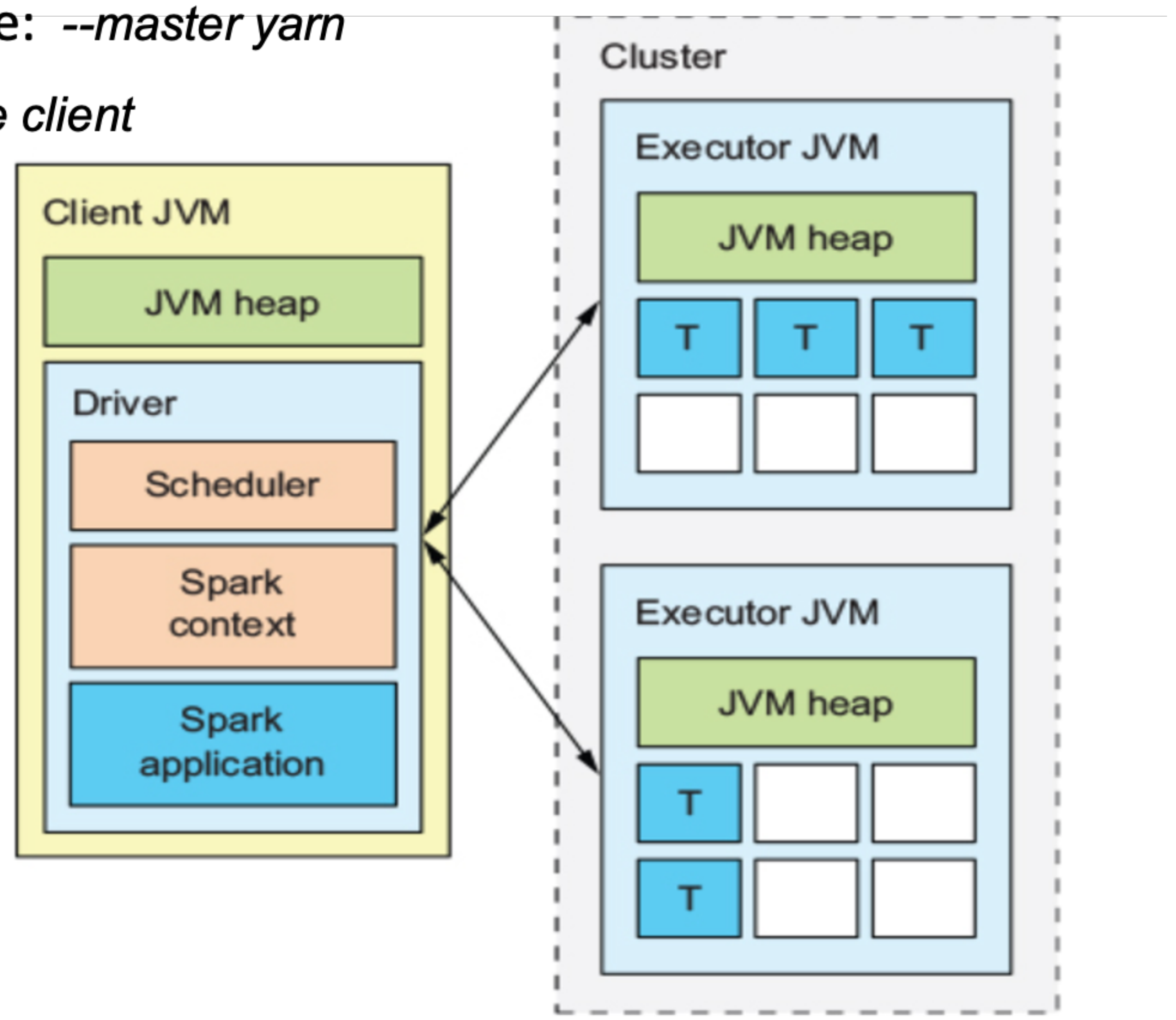

Client JVM
JVM heap
Driver
Scheduler
Spark context
Spark application
Executor
T T T
T

This is the local node you are running the command on – Edge node, laptop etc

Cluster mode

➢ Spark-submit use: *--master yarn*

--deploy-mode client

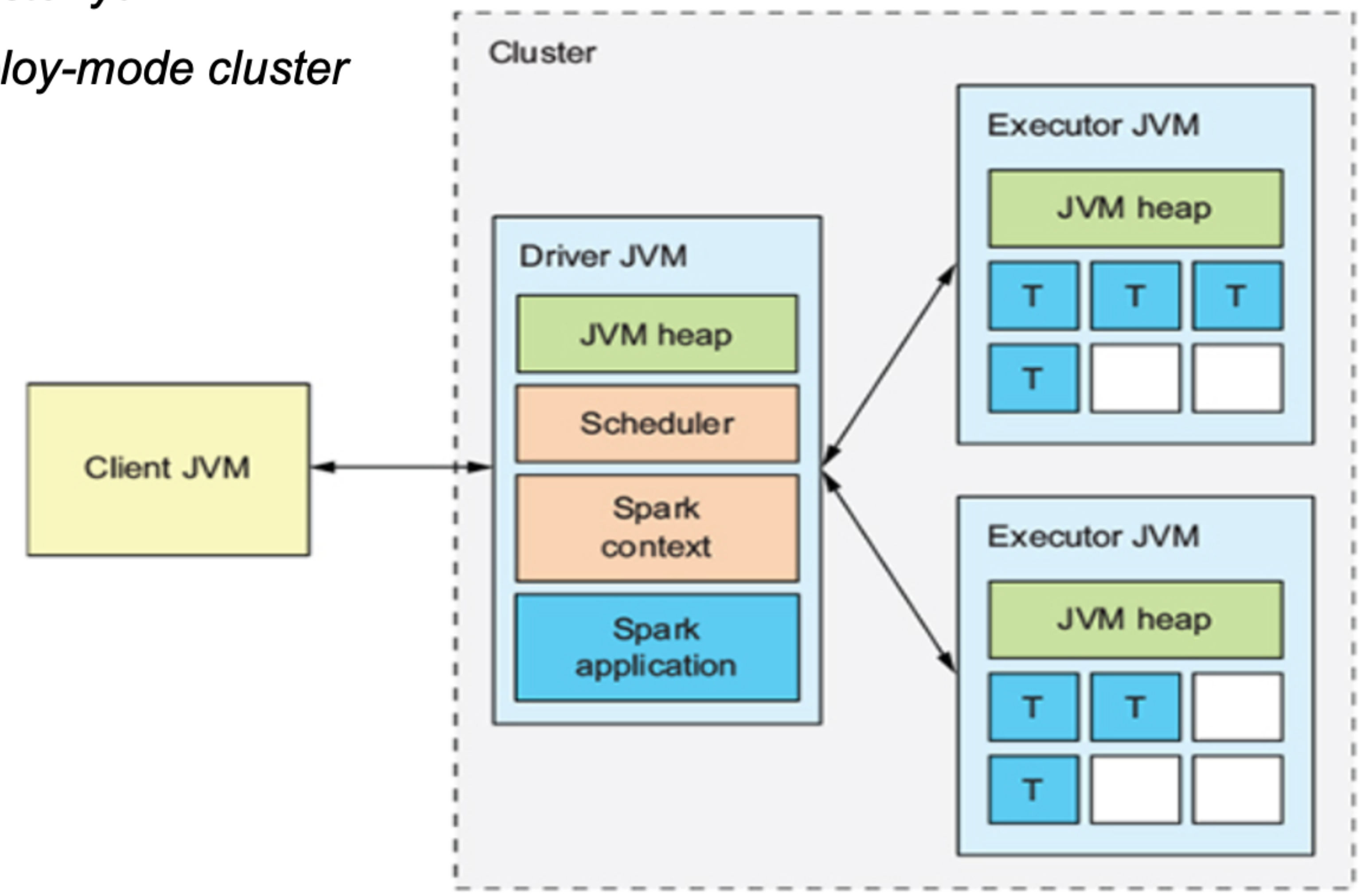

Client JVM
JVM heap
Driver
Scheduler
Spark context
Spark application
Cluster
Executor JVM
JVM heap
T T T
Executor JVM
JVM heap
T
T

Client mode

➢ Spark-submit use:

--master yarn

--deploy-mode cluster


Client JVM
Cluster
Driver JVM
JVM heap
Scheduler
Spark context
Spark application
Executor JVM
JVM heap
T T T
T
Executor JVM
JVM heap
T T
T

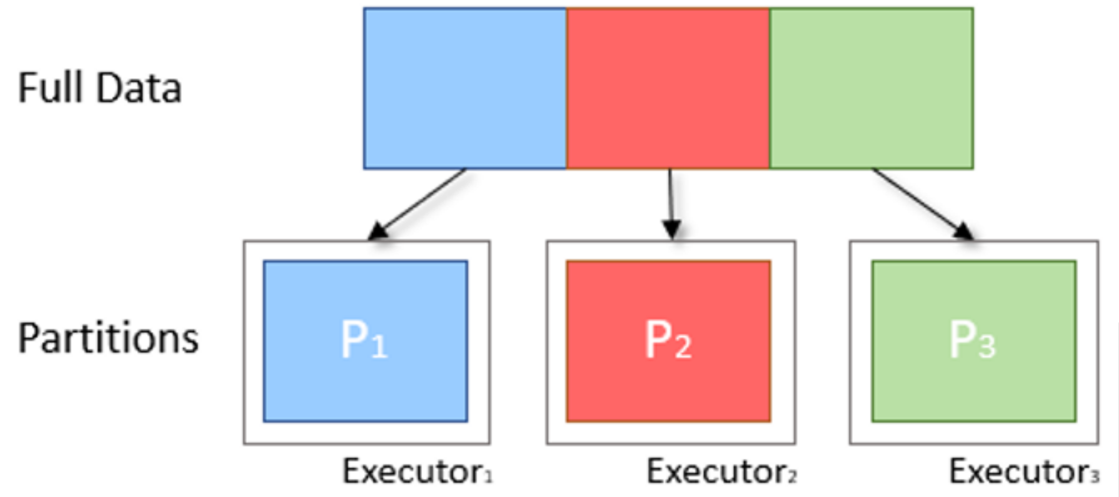
Partitions

- To distribute work across the cluster and reduce the memory requirements of each node Spark will split the data into smaller parts called **Partitions**.

- Each of these is then sent to an Executor to be processed.

- Only one partition is computed per executor **thread** at a time

- Therefore the size and quantity of partitions passed to an executor is directly proportional to the time it takes to complete.

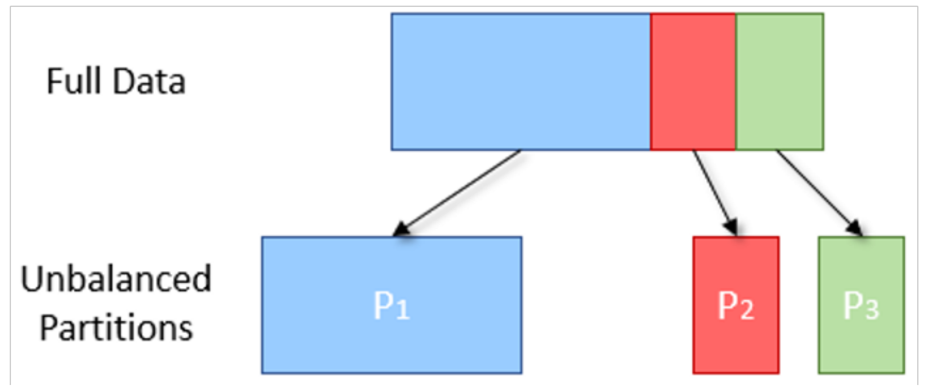


- Often the data is split into partitions based on a key

- If values are not evenly distributed throughout this key then more data will be placed in one partition than another.

- Eg. 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.

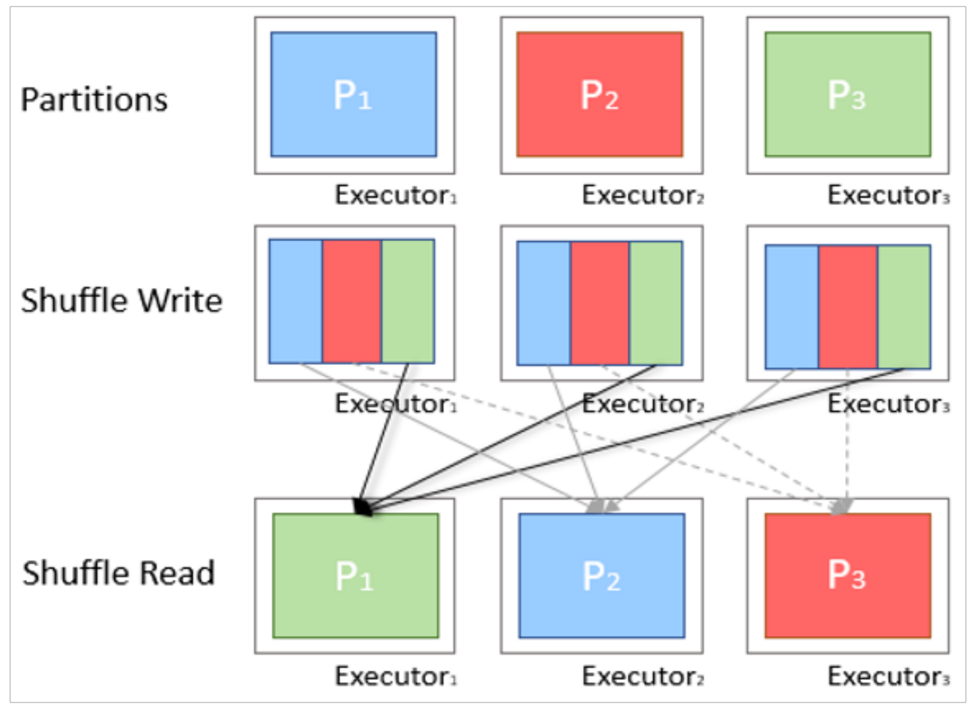


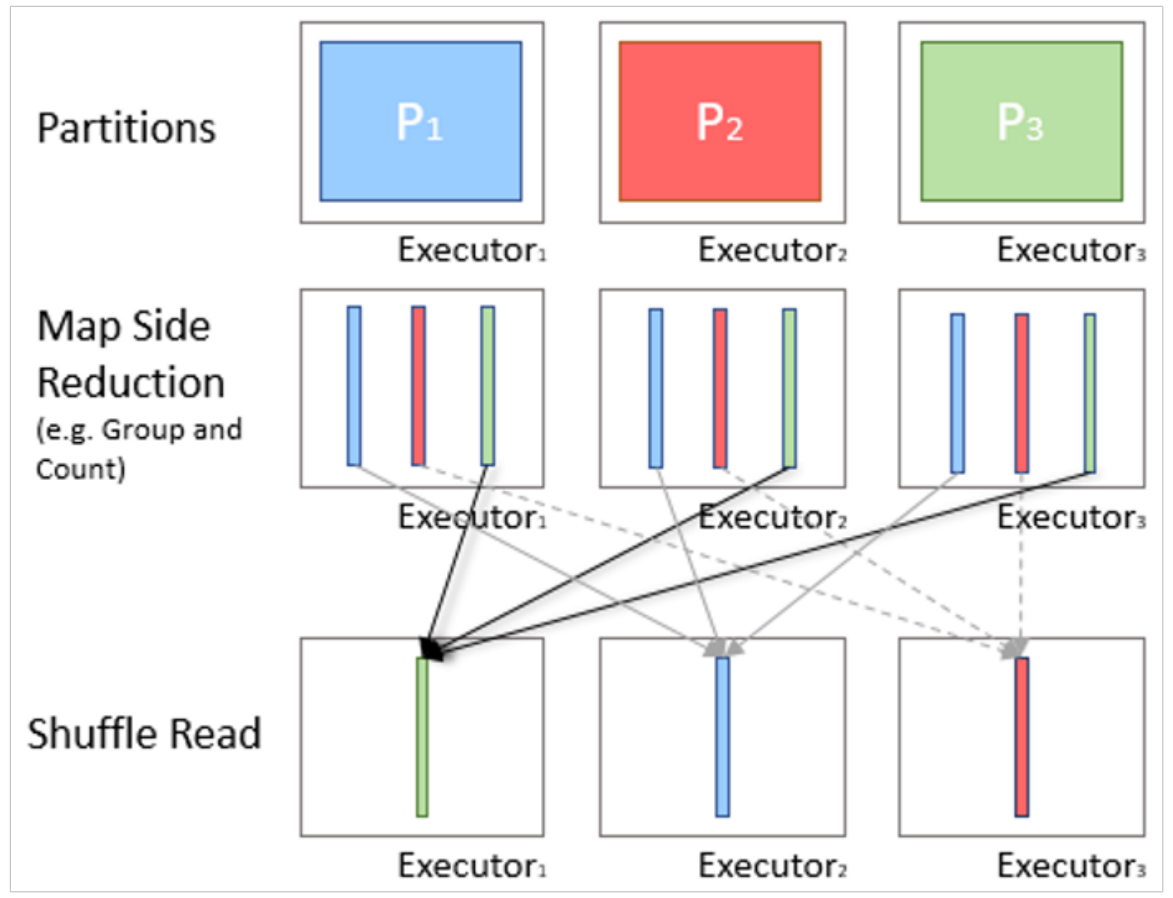
Less Shuffle is good

- A shuffle occurs when data is rearranged between partitions.(ByKey functions in rdd, Grouping, joins etc)

- 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

- During a shuffle, data might be written to disk and transferred across the network, 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.





config controls resource allocation

```
spark-submit \
  --class org.apache.spark.examples.SparkPi \
  --master yarn \
  --deploy-mode cluster \  # can be client for client mode
  --executor-memory 20G \
  --num-executors 50 \
```

spark.driver.cores	1	--driver-cores
spark.driver.memory	1g	--driver-memory
spark.executor.memory	1g	--executor-memory
spark.master	local	--master
spark.executor.cores	1	--executor-cores
spark.executor.instances		--num-executors

Conf not to be used in Yarn mode:

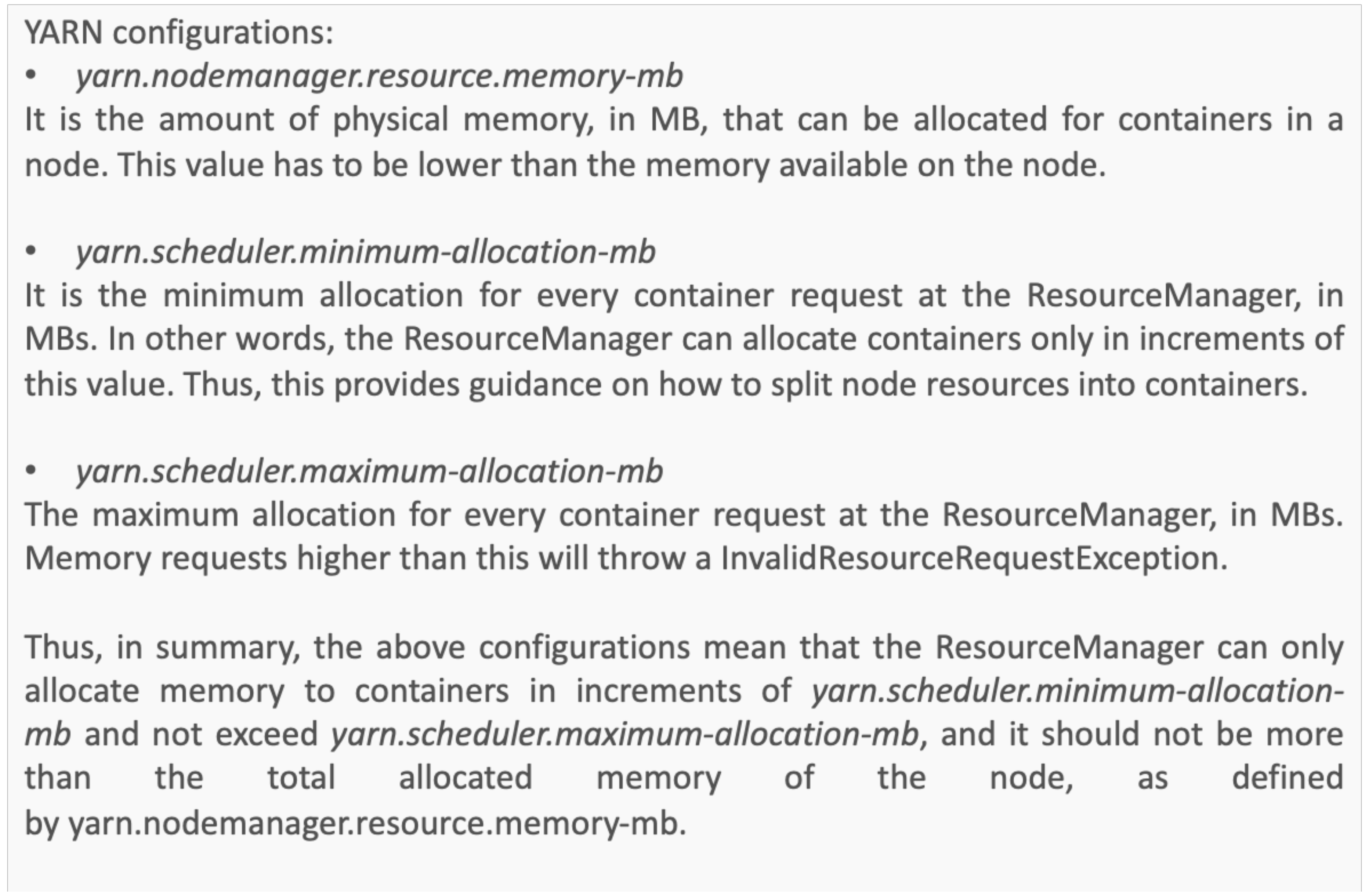
--total-executor-cores 100 (This is specifically for Standalone mode)

--master spark://207.184.161.138:7077 Master is picked from yarn-site xml.

--supervise

https://spark.apache.org/docs/2.2.0/configuration.html

Memory Management



YARN configurations:

- *yarn.nodemanager.resource.memory-mb*

It is the amount of physical memory, in MB, that can be allocated for containers in a node. This value has to be lower than the memory available on the node.

- *yarn.scheduler.minimum-allocation-mb*

It is the minimum allocation for every container request at the ResourceManager, in MBs. In other words, the ResourceManager can allocate containers only in increments of this value. Thus, this provides guidance on how to split node resources into containers.

- *yarn.scheduler.maximum-allocation-mb*

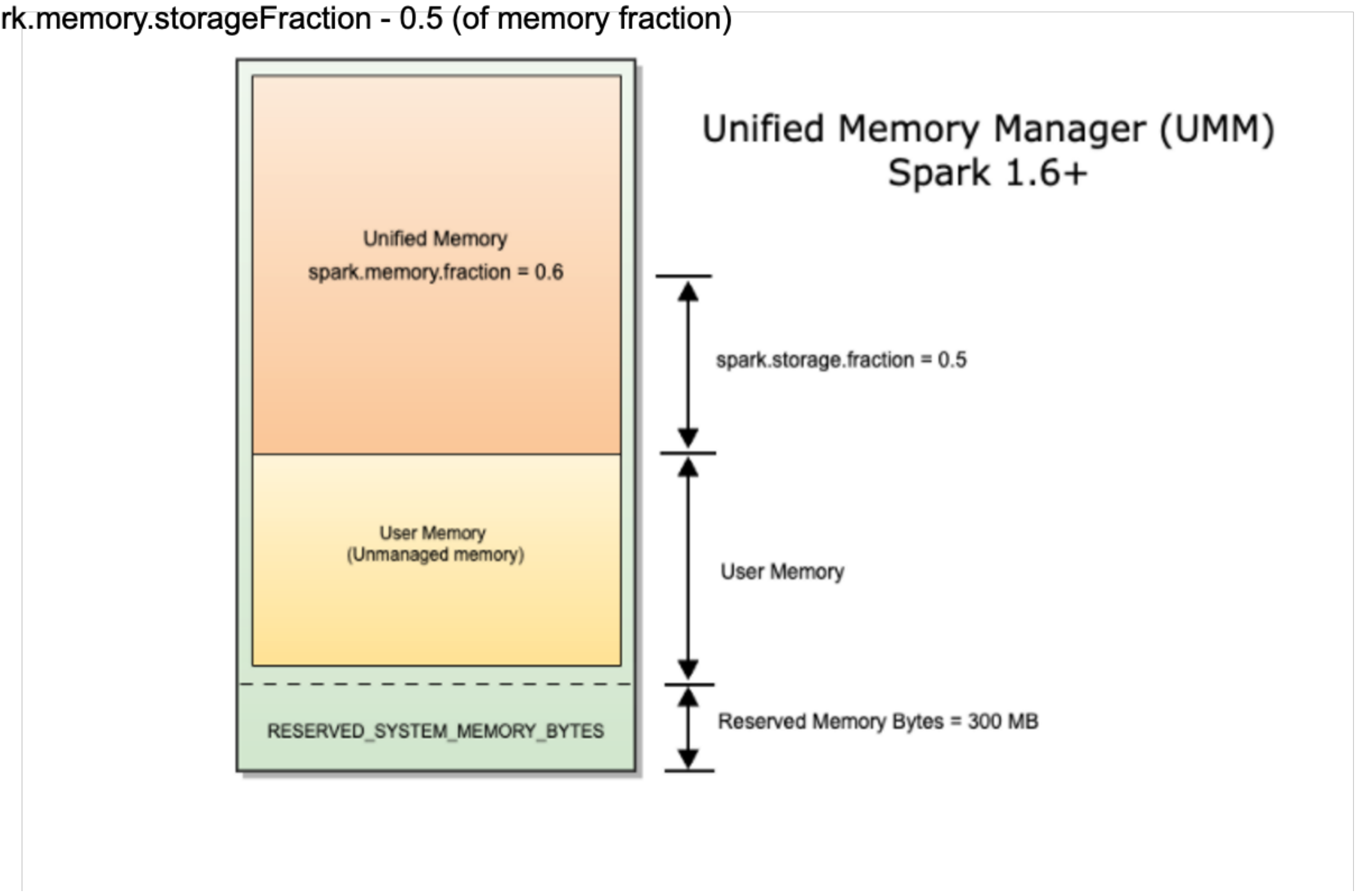
The maximum allocation for every container request at the ResourceManager, in MBs. Memory requests higher than this will throw a InvalidResourceRequestException.

Thus, in summary, the above configurations mean that the ResourceManager can only allocate memory to containers in increments of *yarn.scheduler.minimum-allocation-mb* and not exceed *yarn.scheduler.maximum-allocation-mb*, and it should not be more than the total allocated memory of the node, as defined by yarn.nodemanager.resource.memory-mb.

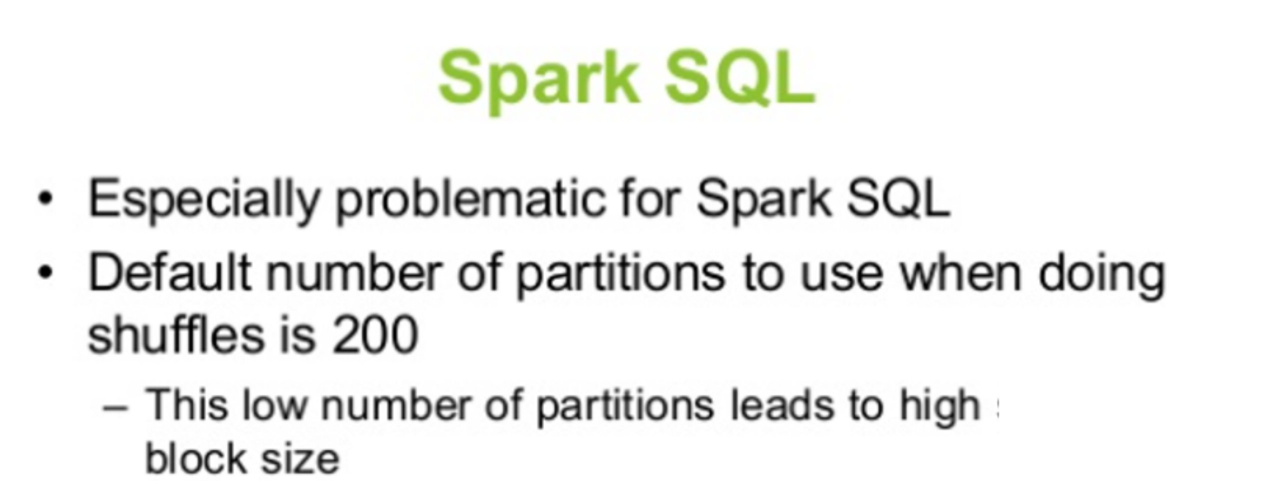
Memory request to Yarn/exec = spark.executor.memory + spark.executor.memoryOverhead

spark.memory.fraction - 0.6 (of executor memory)

spark.memory.storageFraction - 0.5 (of memory fraction)

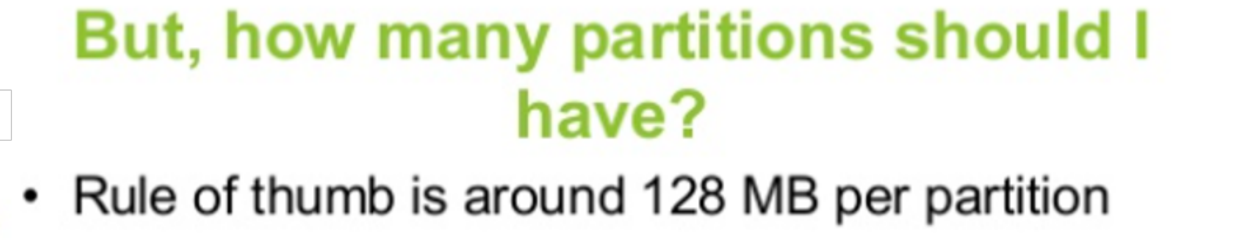

Unified Memory Manager (UMM)
Spark 1.6+
Unified Memory
spark.memory.fraction = 0.6
spark.storage.fraction = 0.5
User Memory
(Unmanaged memory)
User Memory
RESERVED_SYSTEM_MEMORY_BYTES
Reserved Memory Bytes = 300 MB

Optimizations – Shuffle partitions, DataFrames and Cache



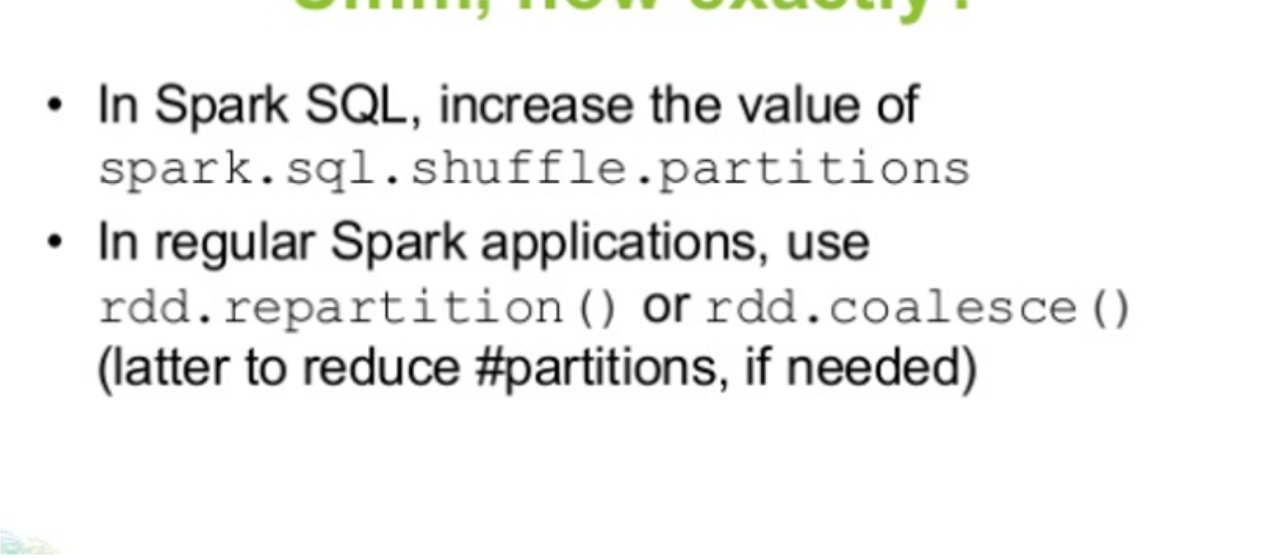
Spark SQL

- Especially problematic for Spark SQL
- Default number of partitions to use when doing shuffles is 200
  - This low number of partitions leads to high block size



But, how many partitions should I have?

- Rule of thumb is around 128 MB per partition



Umm, how exactly?

- In Spark SQL, increase the value of `spark.sql.shuffle.partitions`
- In regular Spark applications, use `rdd.repartition()` or `rdd.coalesce()` (latter to reduce #partitions, if needed)

➢ Cache/Persist:

- Reuse a dataframe with transformations
- Un-persist when done
- Without cache, dataframe is built from scratch each time

➢ Minimize Data scans:

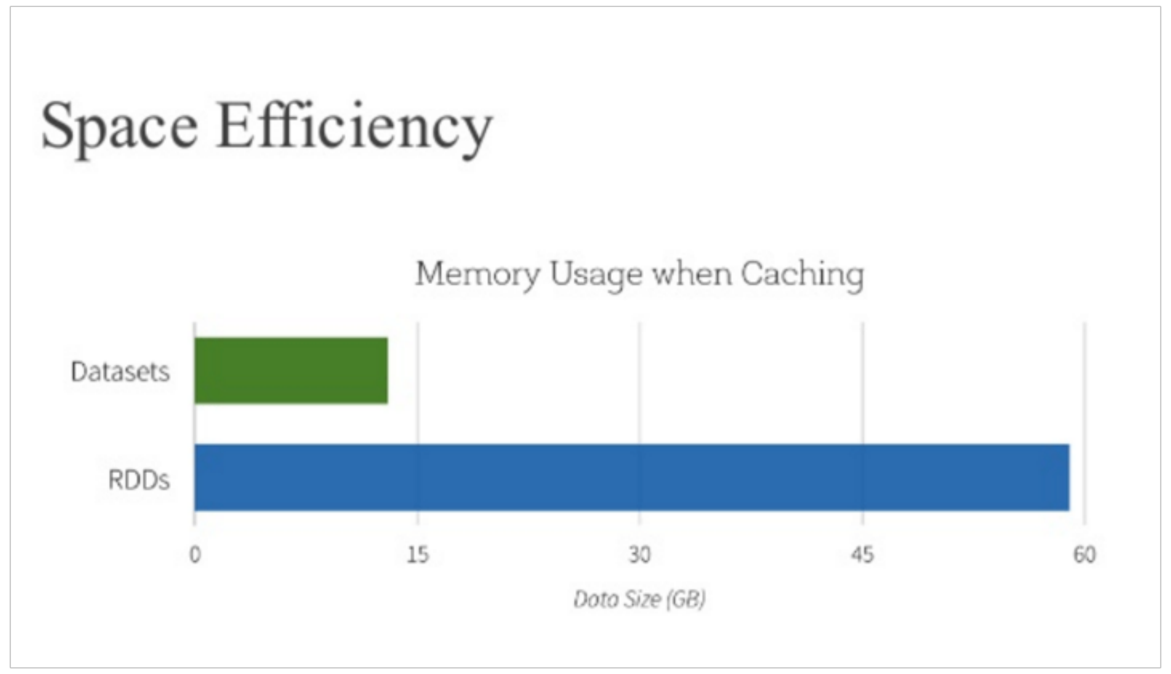
- Predicate push down
- Partition pruning
- Persistance

➢ Use DataFrame/Dataset over RDD:

➢ Spark RDD is a building block of Spark programming, even when we use DataFrame/Dataset, Spark internally uses RDD to execute operations/queries.

➢ Using RDD directly leads to performance issues as Spark doesn't know how to apply the optimization techniques and RDD serialize and de-serialize the data when it distributes across a cluster (repartition & shuffling).

➢ Serialization and de-serialization are very expensive operations for Spark applications or any distributed systems, most of our time is spent only on serialization of data rather than executing the operations hence try to avoid using RDD.



Resources and Links:

- https://www2.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-82.pdf
- https://spark.apache.org/research.html
- https://databricks.com/documentation
- Wheeler docs
- LinkedIN
- https://docs.databricks.com/delta/best-practices.html

Спасибо