upGrad

Optimising Spark for Large-Scale Data Processing

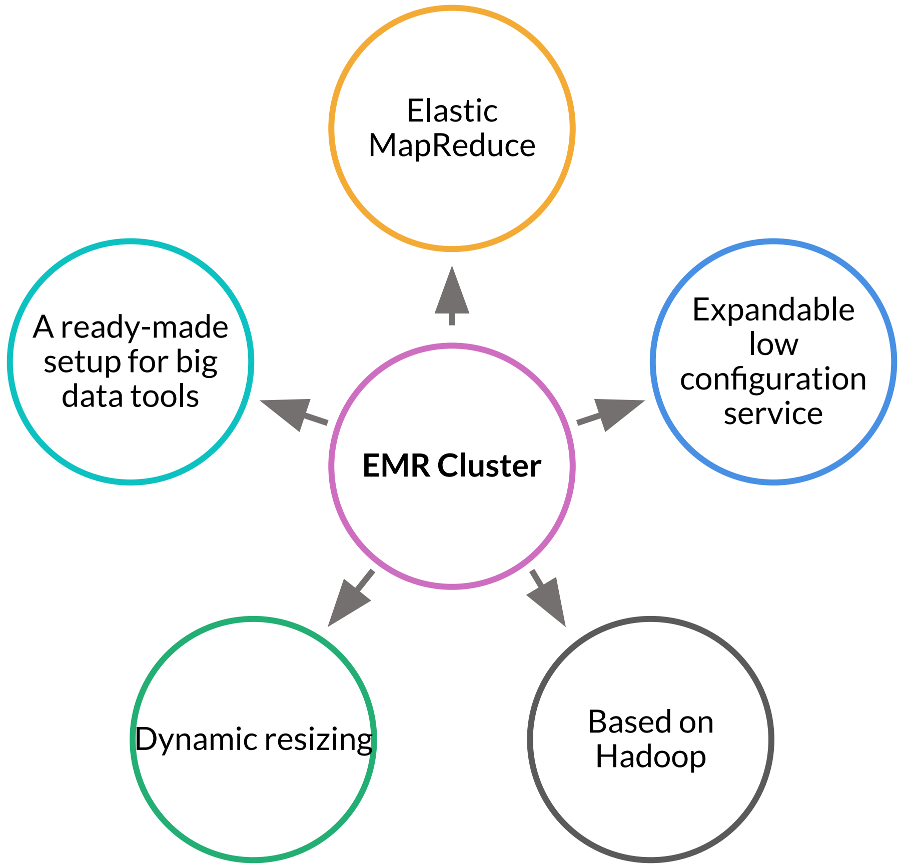
upGrad

Course: Data Engineering - II

Lecture On: Optimising Disk I/O for Spark

Instructor: Vishwa Mohan

EMR CLUSTER AT A GLANCE

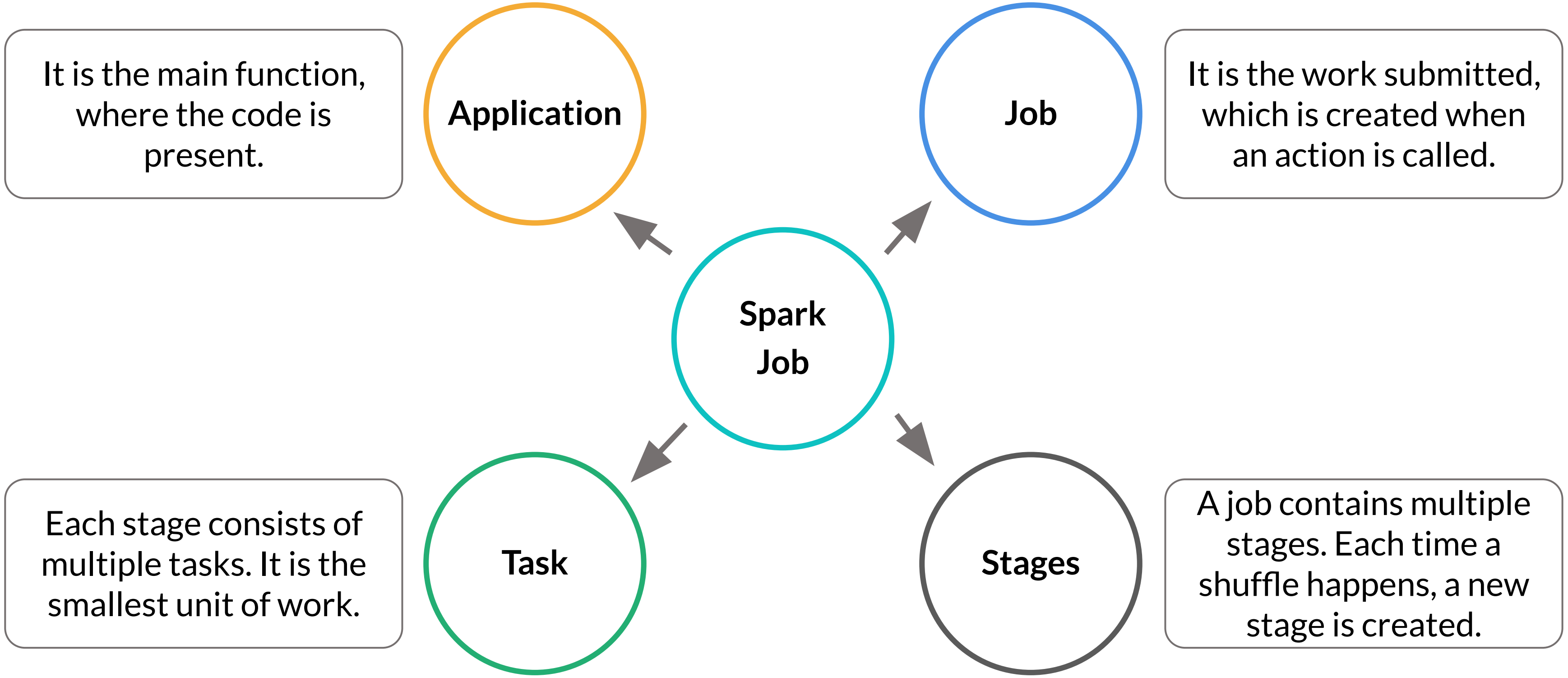

EMR Cluster
Elastic MapReduce
Expandable low configuration service
Based on Hadoop
Dynamic resizing
A ready-made setup for big data tools

Amazon EMR is a managed cluster platform that simplifies running big data frameworks, such as **Apache Hadoop** and **Apache Spark**, on AWS to process and analyse vast amounts of data.

EMR CLUSTER

Advantage	Disadvantage
Provides automatic scaling and ensures minimal loss of HDFS data and low costs, as spot instances can be used	Has no management console similar to that of Cloudera Manager, which makes it difficult to manage and monitor services
Enables dynamic orchestration of a new cluster on-demand and easy termination once the work is done to optimise costs	Does not ensure high availability of the cluster's master node, which makes it a single point of failure
Enables direct access to data on S3 from or through Hive tables	Does not allow shutting down of the clusters and is restricted to being directly terminated
Enables highly availability of slave nodes due to constant monitoring and replaces unhealthy nodes with new nodes	Although automatic replacement of unhealthy nodes offers a maintenance advantage, this poses a risk of data loss

COMPONENTS OF A SPARK JOB



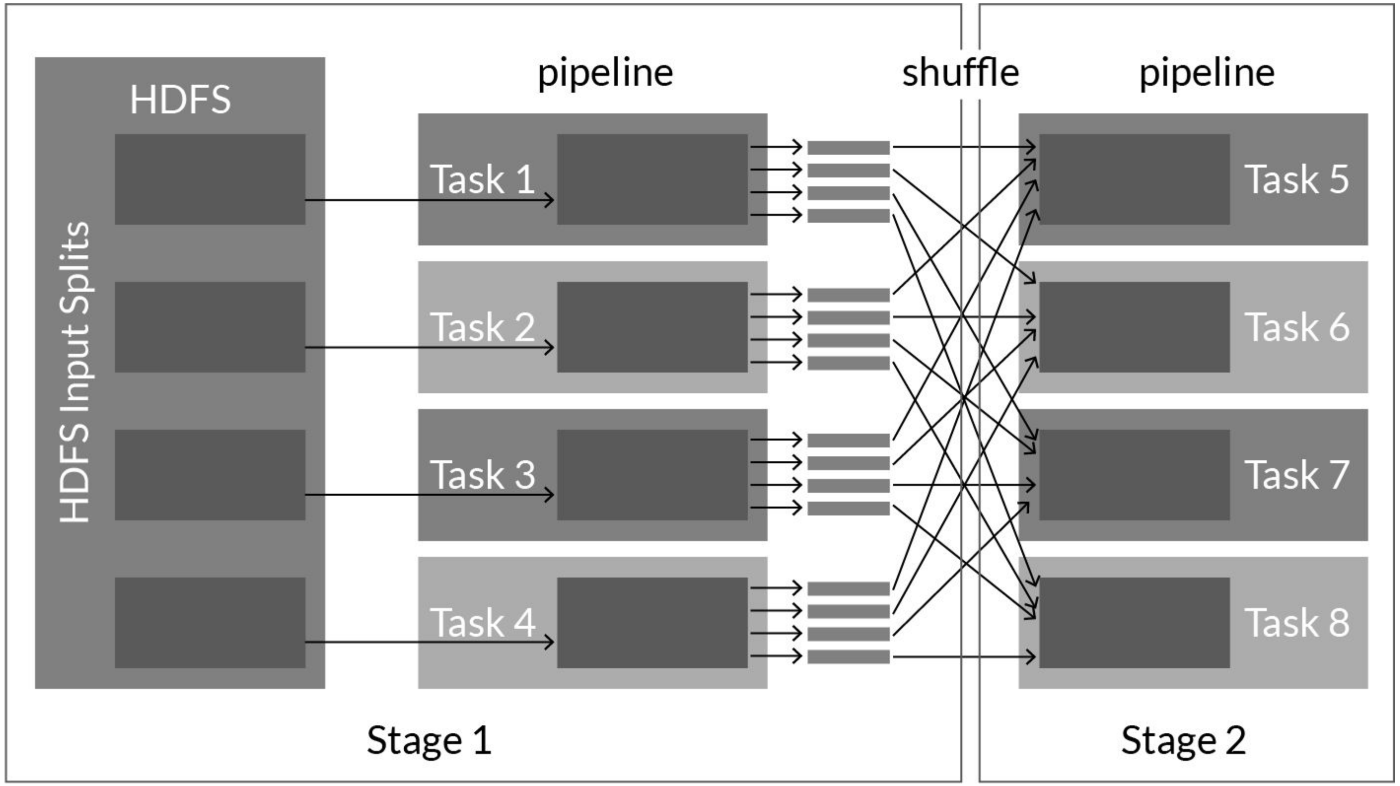
It is the main function, where the code is present.

It is the work submitted, which is created when an action is called.

Each stage consists of multiple tasks. It is the smallest unit of work.

A job contains multiple stages. Each time a shuffle happens, a new stage is created.

COMPONENTS OF A SPARK JOB



WHY OPTIMISE A SPARK JOB?

01 From the execution time perspective

02 From the resource utilisation perspective

03 From the scalability point of view

04 From the maintainability point of view

KEY PERFORMANCE METRICS IN A SPARK JOB

01 Stages and tasks

02 RDD - memory imprint and usage

03 Spark environment information

04 Detailed information about running executors

JOB OPTIMISATION

Code-Level Optimisation

For example, deciding on the right number of partitions and the partitioning size for the entire data set, which APIs to use to handle the data, which methods to use.

Cluster-Level Optimisation

For example, deciding on the number of machines, how to optimise the utilisation of a cluster, how to prevent underutilisation.

CODE OPTIMISATION

01 Reduce Disk I/O

02 Reduce Network I/O

REDUCING DISK I/O

01 Proper serialisation

02 Memory tuning

03 Storing data in more optimised file formats

04 Using a caching strategy

REDUCING NETWORK I/O

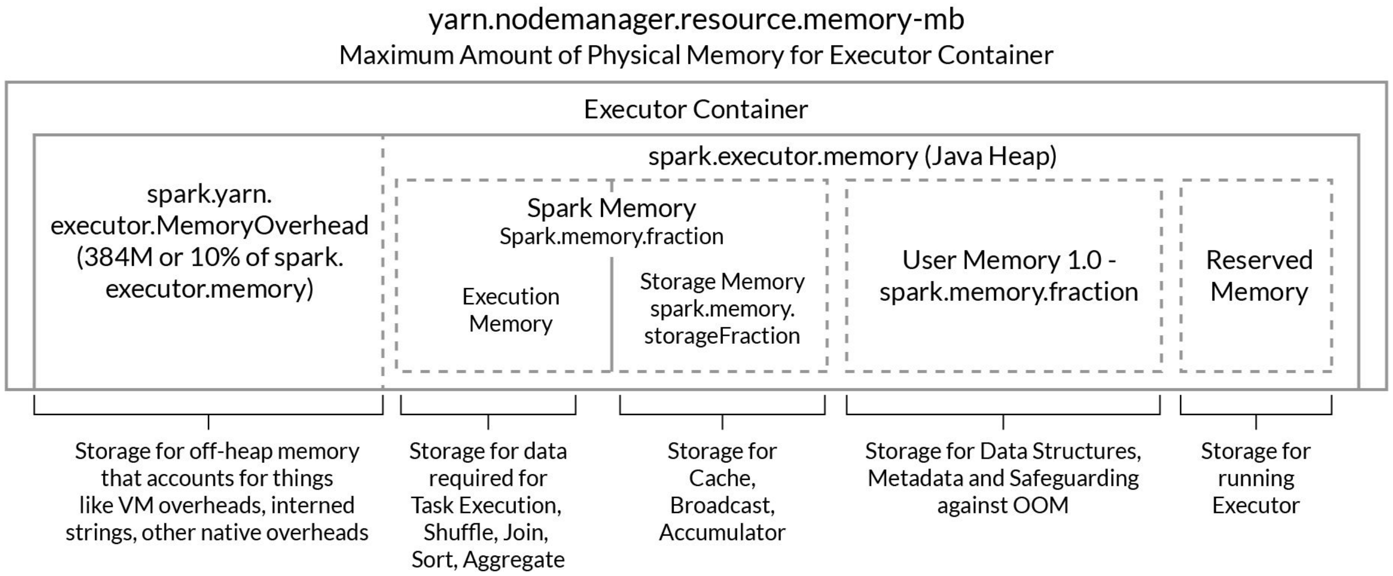
01 Data locality

02 Partitioning

03 Shuffle and sort

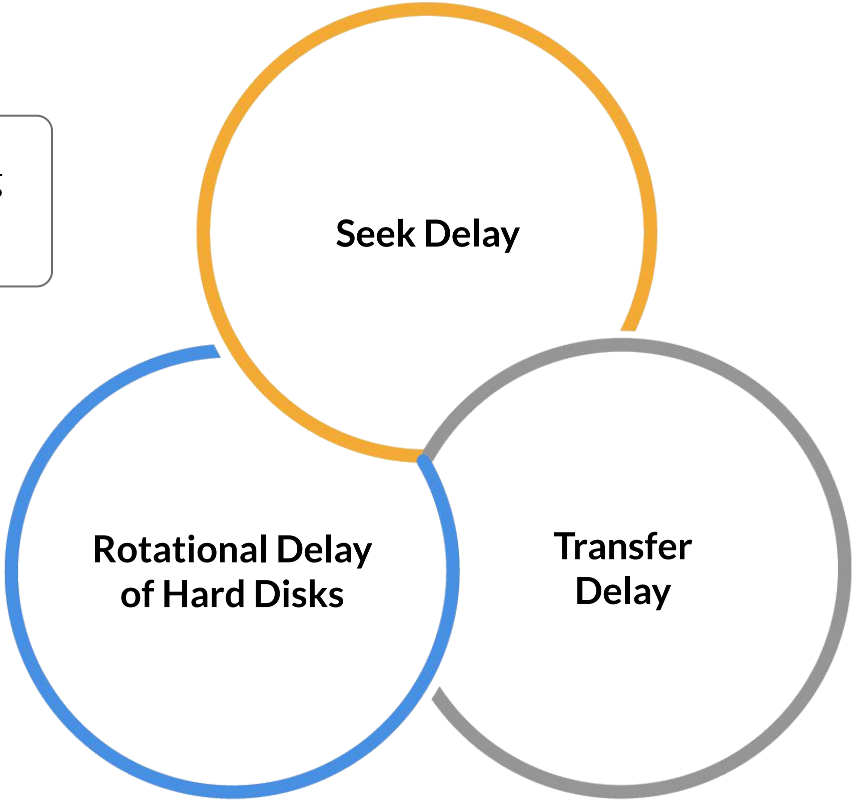
04 Joins

CLUSTER OPTIMISATION


yarn.nodemanager.resource.memory-mb
Maximum Amount of Physical Memory for Executor Container
Executor Container
spark.executor.memory (Java Heap)
spark.yarn. executor.MemoryOverhead (384M or 10% of spark. executor.memory)
Spark Memory
Spark.memory.fraction
Execution Memory
Storage Memory spark.memory. storageFraction
User Memory 1.0 - spark.memory.fraction
Reserved Memory
Storage for off-heap memory that accounts for things like VM overheads, interned strings, other native overheads
Storage for data required for Task Execution, Shuffle, Join, Sort, Aggregate
Storage for Cache, Broadcast, Accumulator
Storage for Data Structures, Metadata and Safeguarding against OOM
Storage for running Executor

UNDERSTANDING DISK I/O IN SPARK

Disk I/O
Reading/Writing data from hard disks



Total Delay = Seek Delay + Rotational Delay of HDD + Transfer Delay

TECHNIQUES TO REDUCE DISK I/O

Q1 Avoid shuffling as much as possible

A: Shuffling leads to stages. At the stage boundary, data is stored in the disk to be fault-tolerant

Q2 File formats: Parquet and ORC

A: Compress the size by close to 70%, which leads to less storage, and use columnar storage for better compression

Q3 Serialisation and deserialisation

A: Uses serialised memory storage and cached data in a serialised form using the Kryo algorithm

USING VARIOUS FILE FORMATS IN SPARK

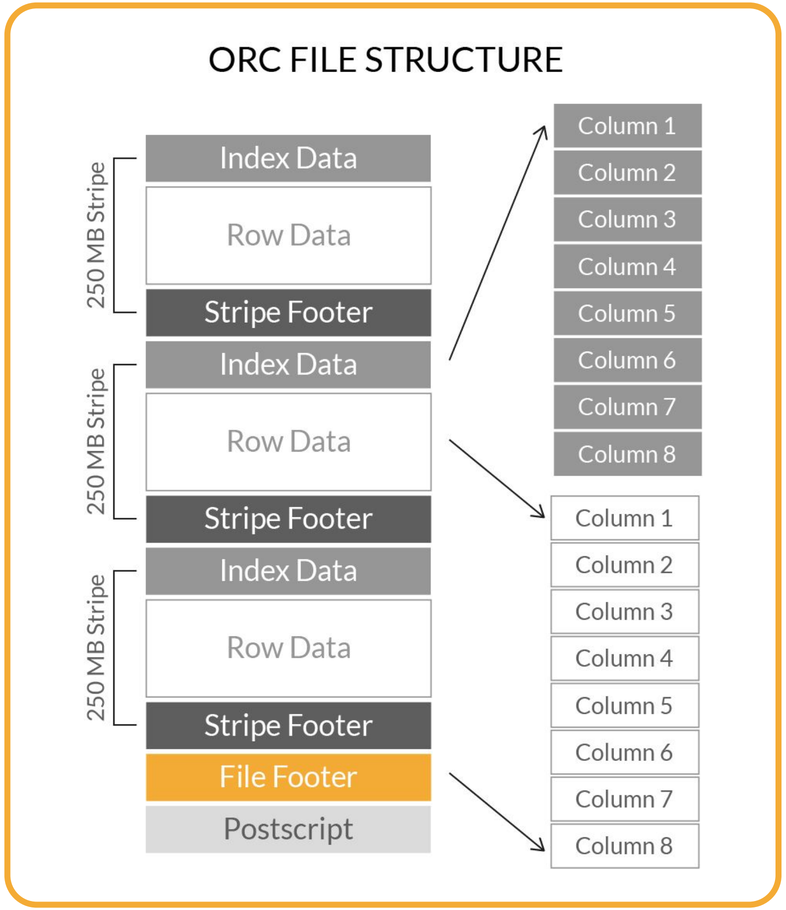
01 .txt

02 .csv

03 .json Parquet

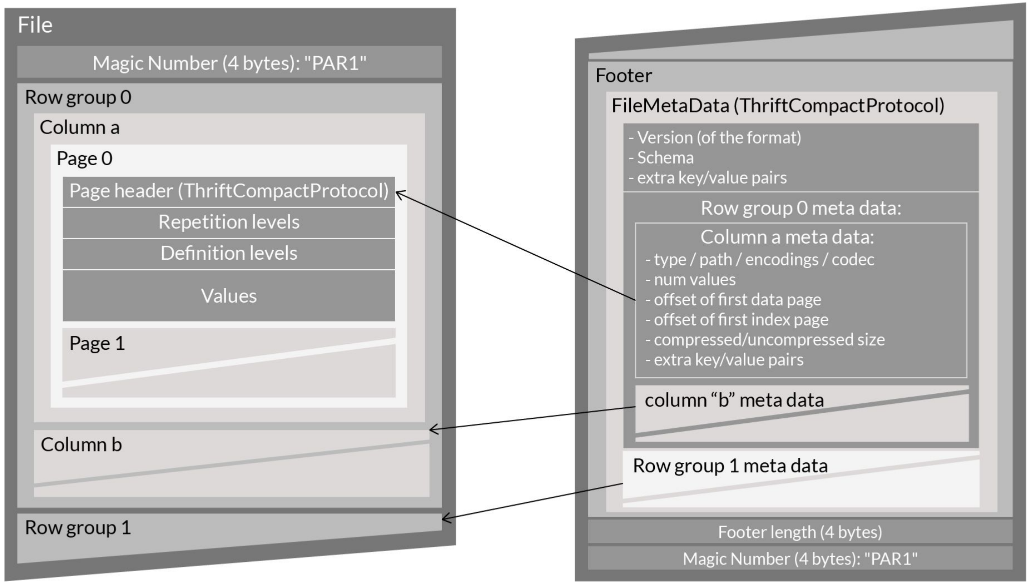
04 .avro ORC

ORC FILE FORMAT



- Used for both compressed and uncompressed storage
- Stores a collection of rows in a file, and within a collection, it stores row data in a columnar format
- Fast response time
- Better when the original data is flat
- Supports light weight index

PARQUET FILE FORMAT


File
Magic Number (4 bytes): "PAR1"
Row group 0
Column a
Page 0
Page header (ThriftCompactProtocol)
Repetition levels
Definition levels
Values
Page 1
Column b
Row group 1
Footer
FileMetaData (ThriftCompactProtocol)
- Version (of the format)
- Schema
- extra key/value pairs
Row group 0 meta data:
Column a meta data:
- type / path / encodings / codec
- num values
- offset of first data page
- offset of first index page
- compressed/uncompressed size
- extra key/value pairs
column “b” meta data
Row group 1 meta data
Footer length (4 bytes)
Magic Number (4 bytes): "PAR1"

- Columnar storage - efficient compression storage
- Metadata at the end of the file
- Supported by all Apache big data products
- Used in the case of nested data

IMPACT OF CHOOSING A FILE FORMAT

01 Faster read time

02 Faster write time

03 Splittable files

04 Schema evolution support
Advanced compression support

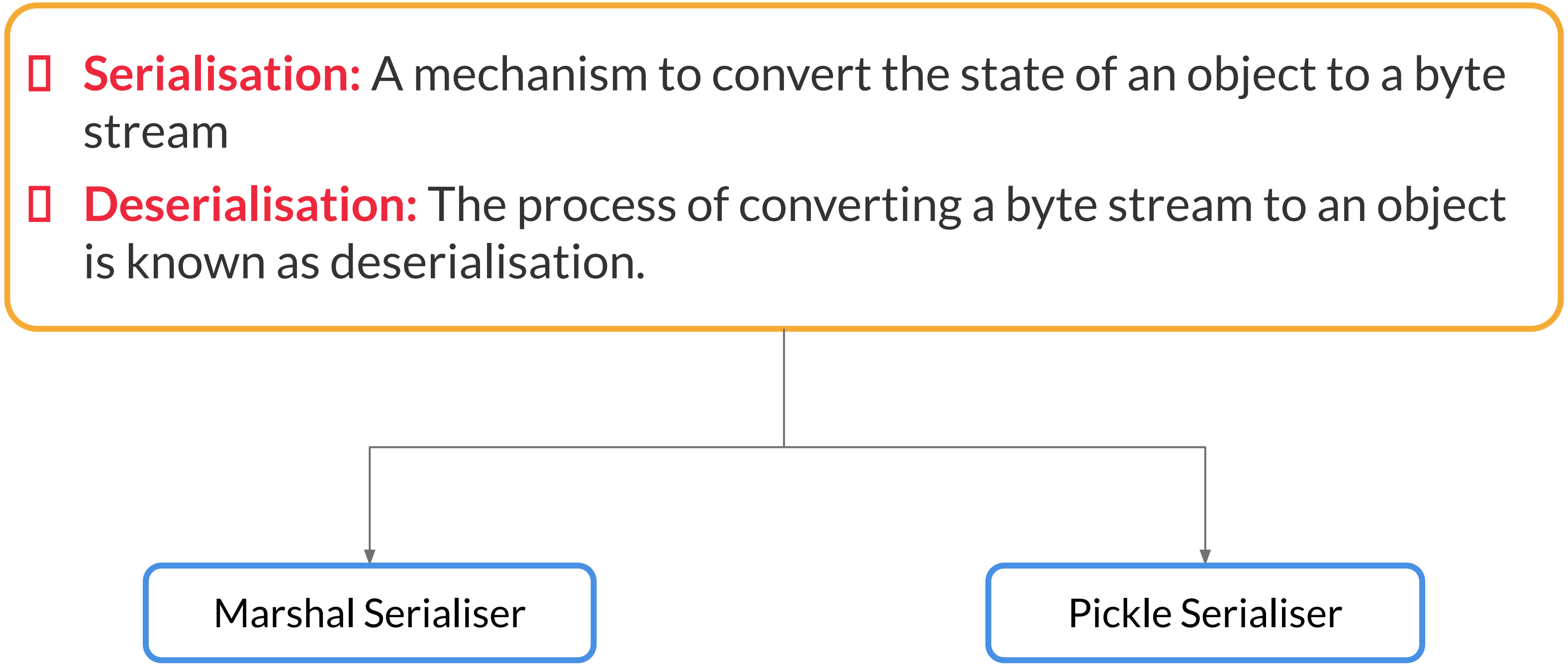
BENEFITS OF USING A COLUMNAR FILE FORMAT

01 Better compression, as the data is more homogenous

02 I/O will be reduced, as you will scan only a subset of a column

03 You can use an encoding better suited to modern processors (as the data in a column is of the same type).

WHAT IS SERIALISATION?



- **Serialisation:** A mechanism to convert the state of an object to a byte stream
- **Deserialisation:** The process of converting a byte stream to an object is known as deserialisation.

WHY SERIALISATION?

- Serialisation is implemented for maintaining performance in distributed systems.
- Serialisation is helpful when you want to save objects to a disk or send them over networks.
- For example, RDDs may be serialised to:
    - Decrease memory usage when stored in a serialised form
    - Reduce network bottleneck in processes such as shuffling
    - Tune Performance
- Another example is that objects can be serialised so they can be sent to the Spark worker nodes.

SERIALISERS ARE SET DURING THE CREATION OF SPARKCONTEXT.

Marshal Serialiser	Pickle Serialiser
Faster	Comparatively slower
Supports fewer data types	Supports nearly every Python object type

SPARK MEMORY MANAGEMENT PARAMETERS
MEMORY LEVELS SUPPORTED BY SPARK

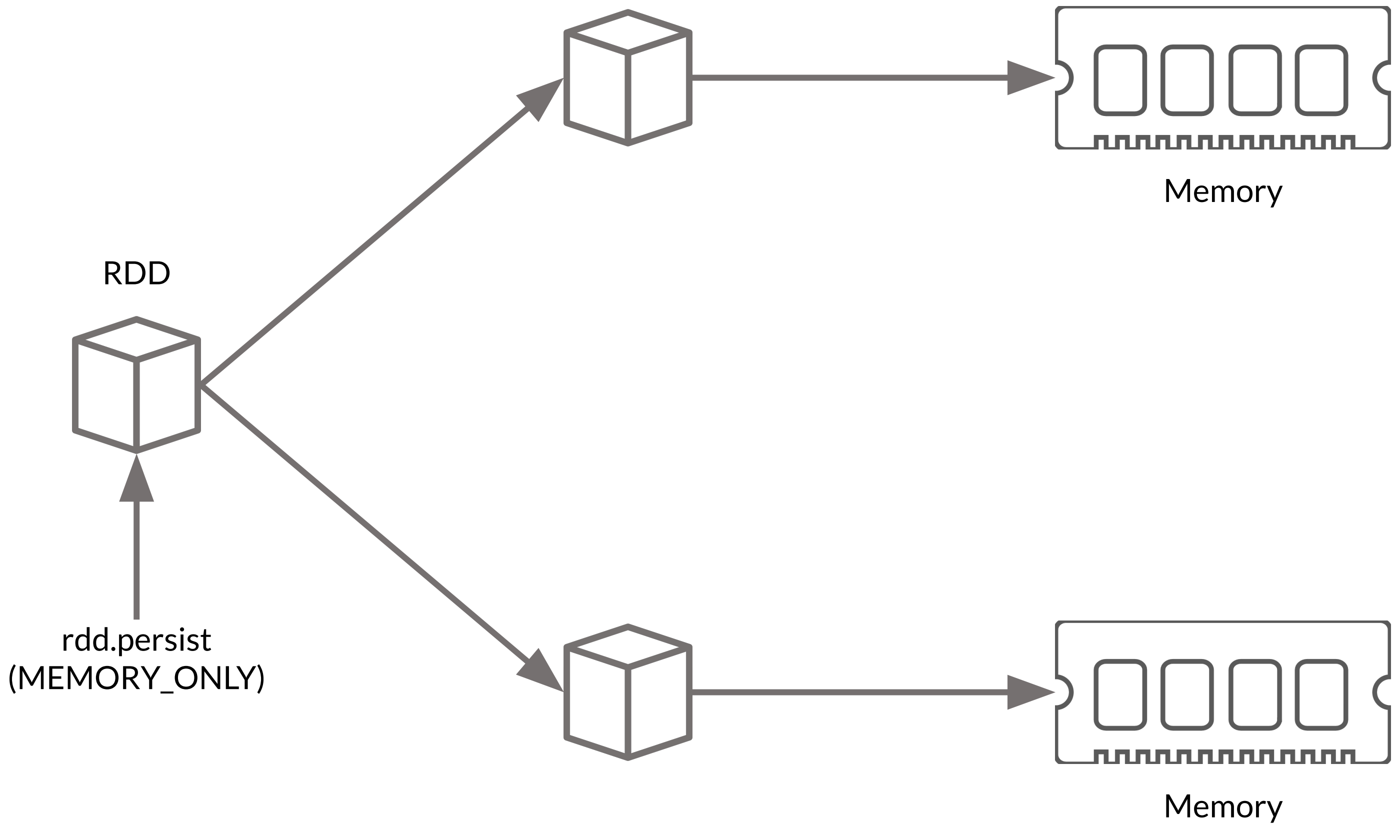
01 MEMORY_ONLY

02 MEMORY_AND_DISK

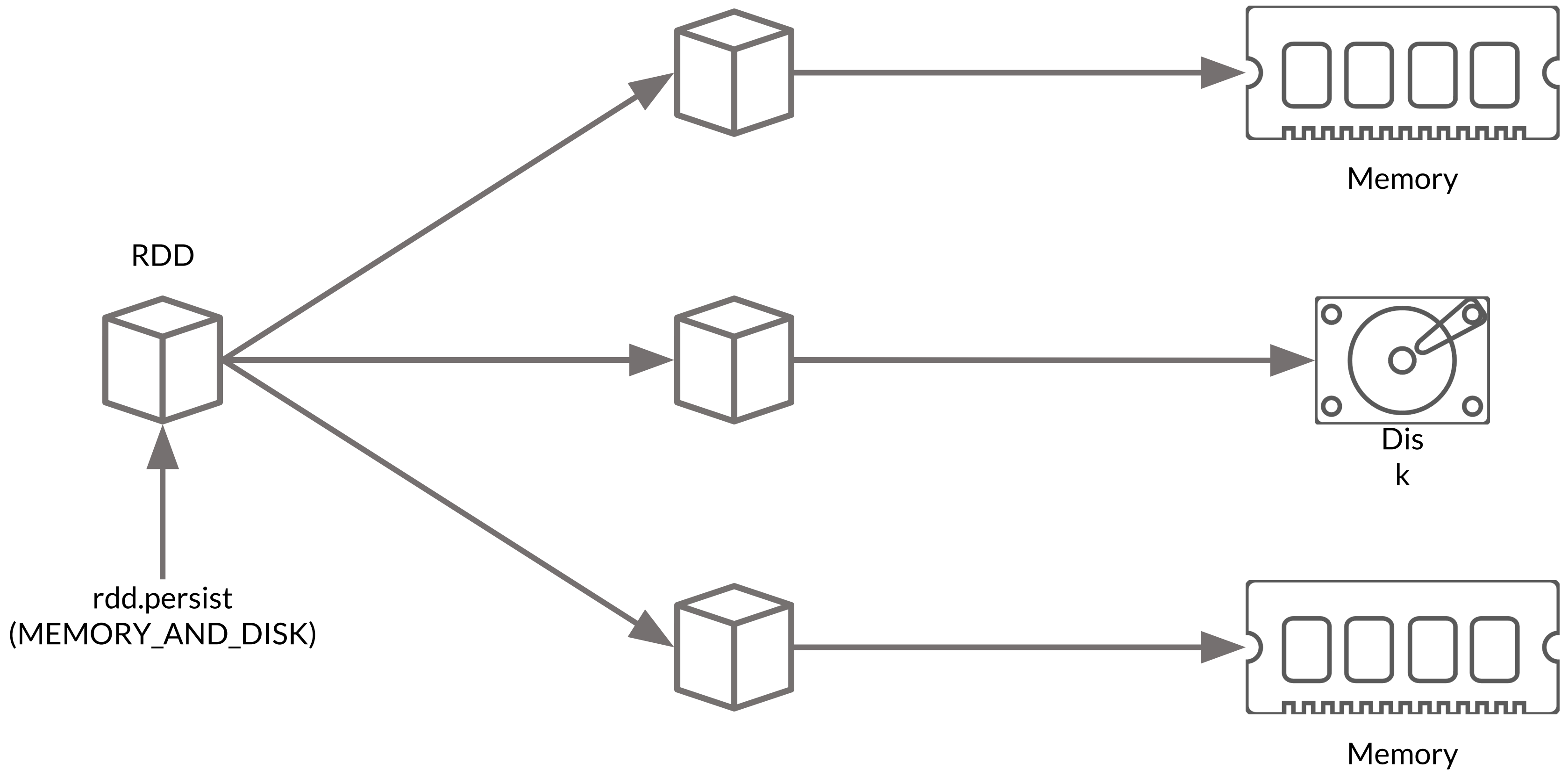
03 MEMORY_ONLY_SER
DISK_ONLY

04 MEMORY_AND_DISK_SER
MEMORY_ONLY_2 and MEMORY_AND_DISK_2

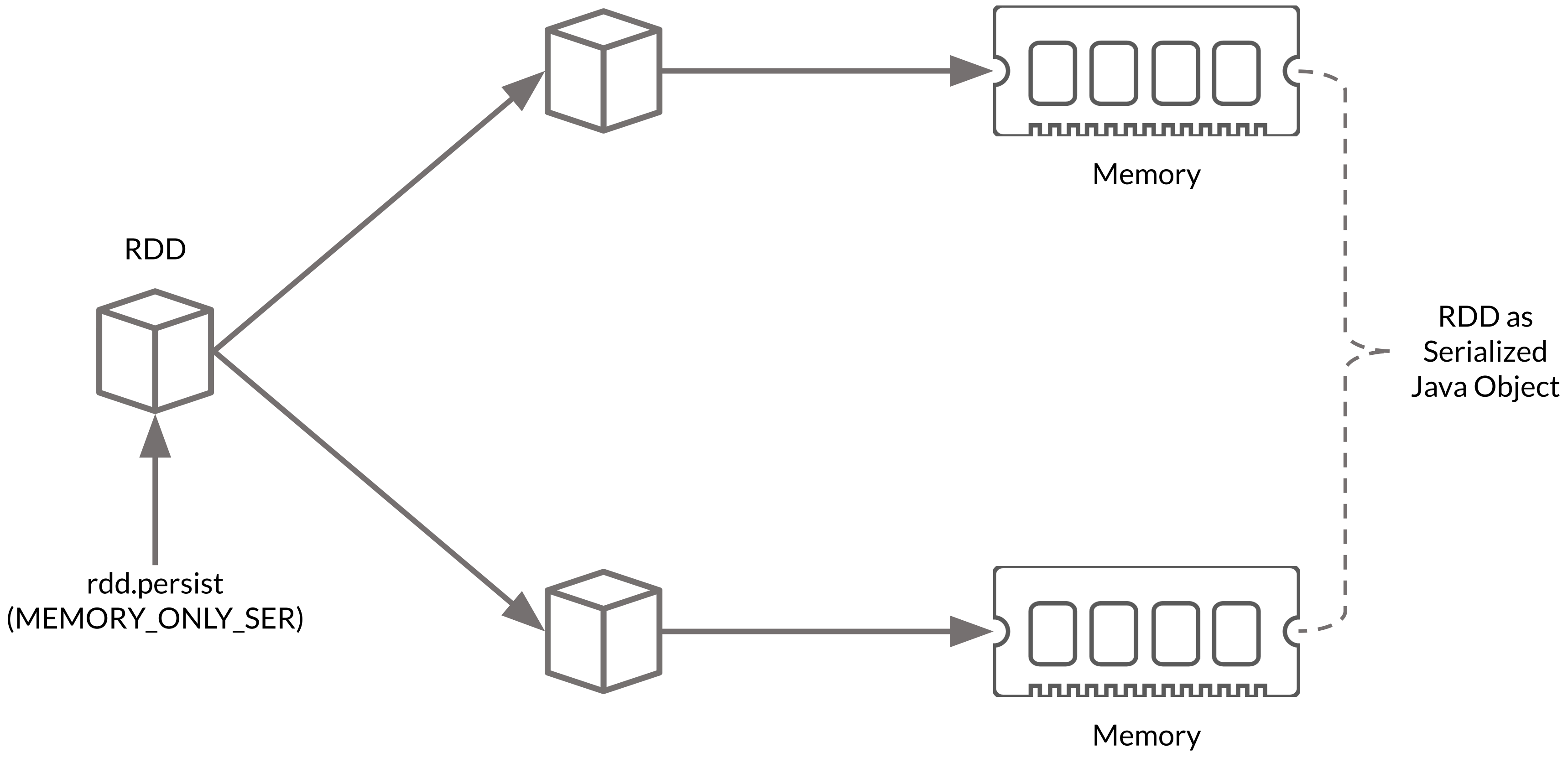
RDD PERSISTENCE: MEMORY ONLY



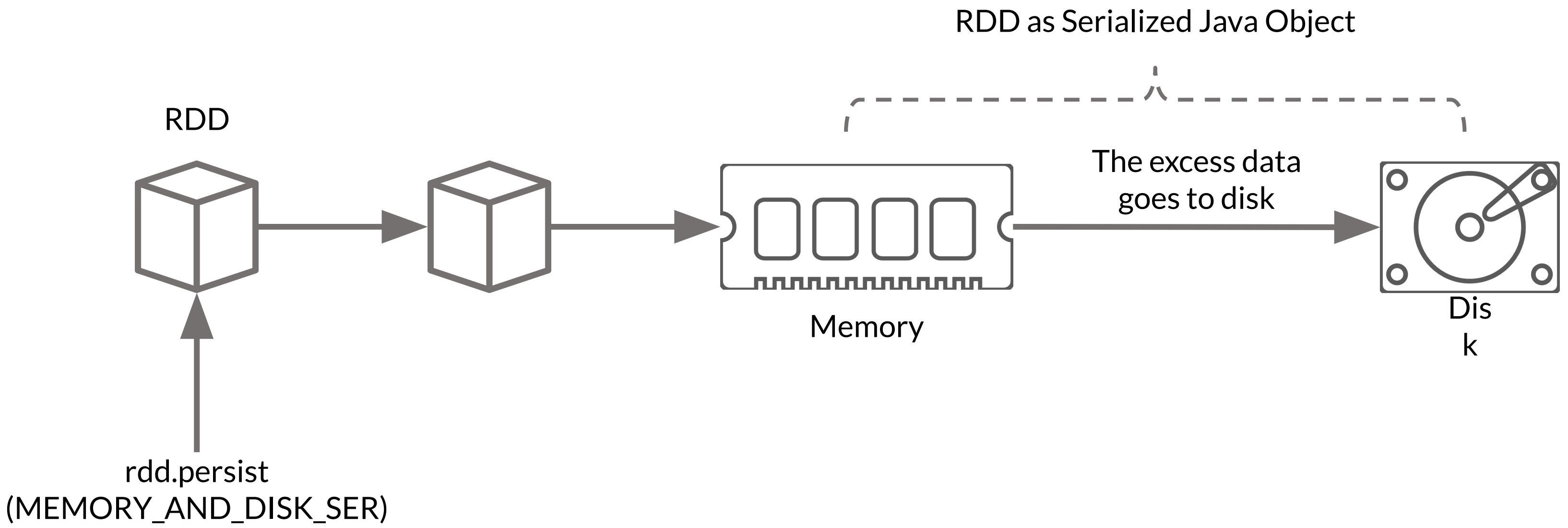
RDD PERSISTENCE: MEMORY & DISK



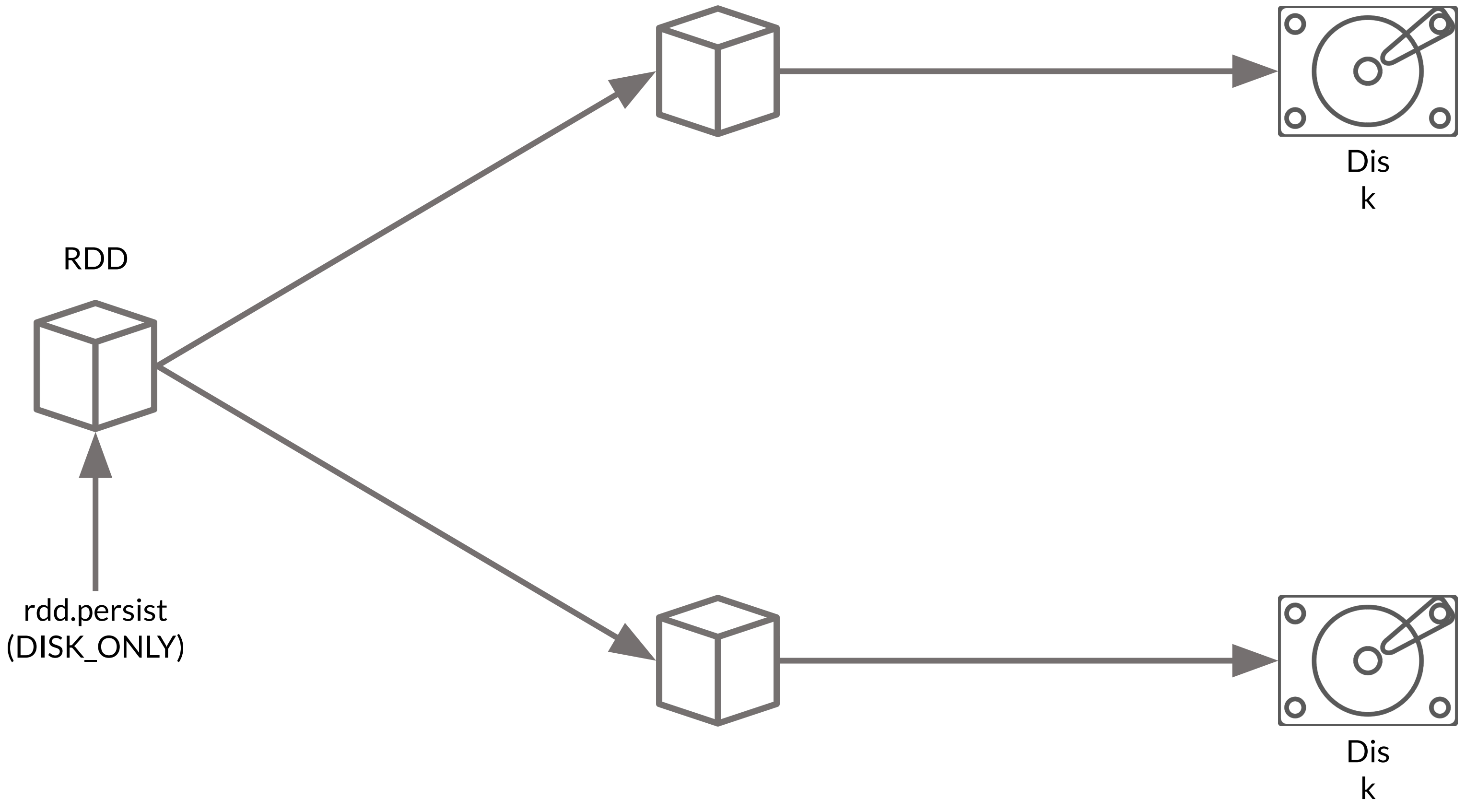
RDD PERSISTENCE: MEMORY ONLY SERIALIZED



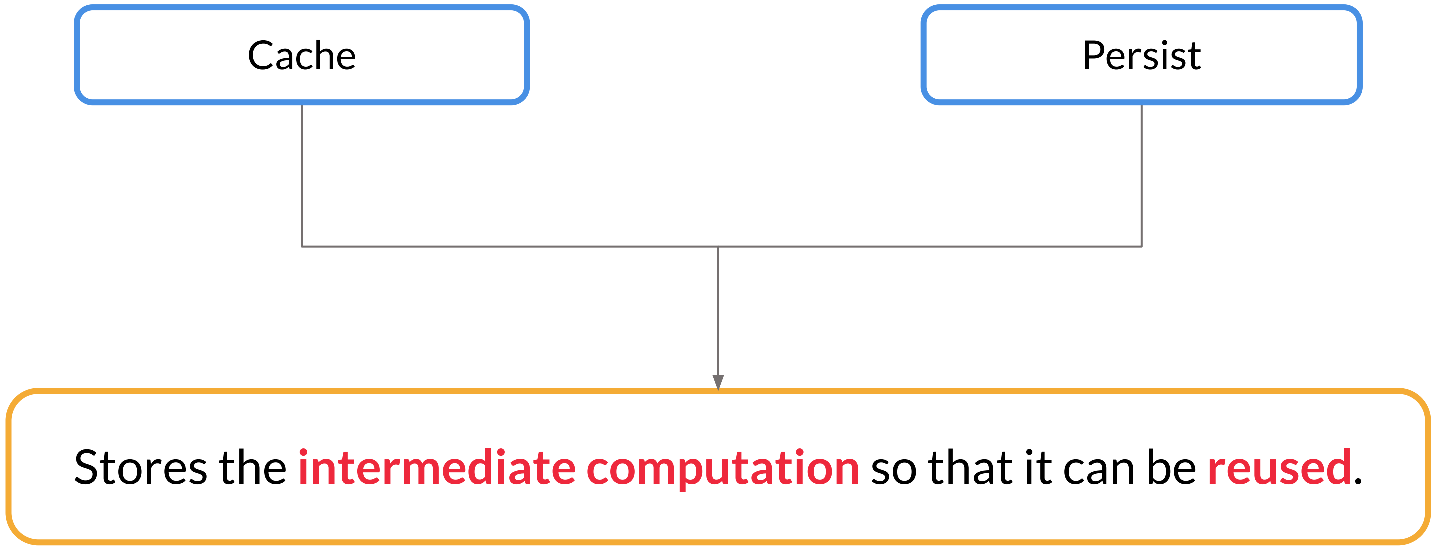
RDD PERSISTENCE: MEMORY & DISK SERIALIZED



RDD PERSISTENCE: DISK ONLY



SPARK MEMORY MANAGEMENT PARAMETERS



Stores the **intermediate computation** so that it can be **reused**.

SPARK MEMORY MANAGEMENT PARAMETERS

Cache

- **Dataframe** - Memory and disk
- **RDD** - Only memory
- Lazy operation

Persist

- **Memory levels supported** - All levels are supported
- Lazy operation

SPARK MEMORY MANAGEMENT PARAMETERS

Checkpoint

- **Breaks the lineage** as compared with Cache and Persist, which maintain the lineage
- Computes **separately** from other jobs
- Checkpoint data is **persistent** and is **not removed after SparkContext is destroyed**

SPARK MEMORY MANAGEMENT PARAMETERS

Unpersist

- **Manually (on demand) remove** the rdd from the cache
- Cache is evicted automatically following LRU eviction principle if not called manually