Apache Spark Introduction

BIG
DATA
USE
ONE
SINCE
SOFTWARE
UBIQUITOUS
REQUIRING
BUSINESS
MOVING
ANALYTICS
SAN
RADIO-FREQUENCY
SOLID
COMPLEXITY
WIRELESS
TOLERABLE
PARALLEL
SIZE
NEEDED
QUALITIES
PROCESSING
BURIED
PETABYTES
LOGS
INTERNET
TECHNOLOGIES
ORGANIZATIONS
USED
DISTRIBUTED
CAPTURE
MAY
MANAGEMENT
MANAGE
GROW
SOCIAL
LARGE
DEFINING
STORE
EVERY
CASE
LARGER
STORAGE
WORKING
USING
TYPES
GARTNER
MASSIVELY
PERFORMANCE
RELATED
ALSO
BIOLOGICAL
HUNDREDS
CREATED
DISK
RELATIONAL
ZETTABYTES
GENOMICS
COMPUTING
TOOLS
SET
SHARED
TIME
WITHIN
COMBAT
SIGNIFICANT
PROCESS
INCLUDE
SYSTEMS
DEFINITION
BIOGEOCHEMICAL
NETWORKS
RECONSIDER
SEARCH
OPPORTUNITIES
INFORMATION
CONNECTOMICS
RECORDS
DESKTOP
CURRENTLY
FC
TENS
CAPACITY
FORMS
PRACTITIONERS
PRESENTATIONS
NOW
WORLD'S
COST
CONTINUES
TERABYTES
MPP
RESEARCH
CITATION
COMPLEX
DATABASES
INDEXING
DIFFICULTY
TARGET
ABILITY
SETS
EXAMPLES
ELAPSED
CURRENT
THOUGHT
APPLIED
SENSOR
ARCHIVES
AMOUNT
DESCRIBING

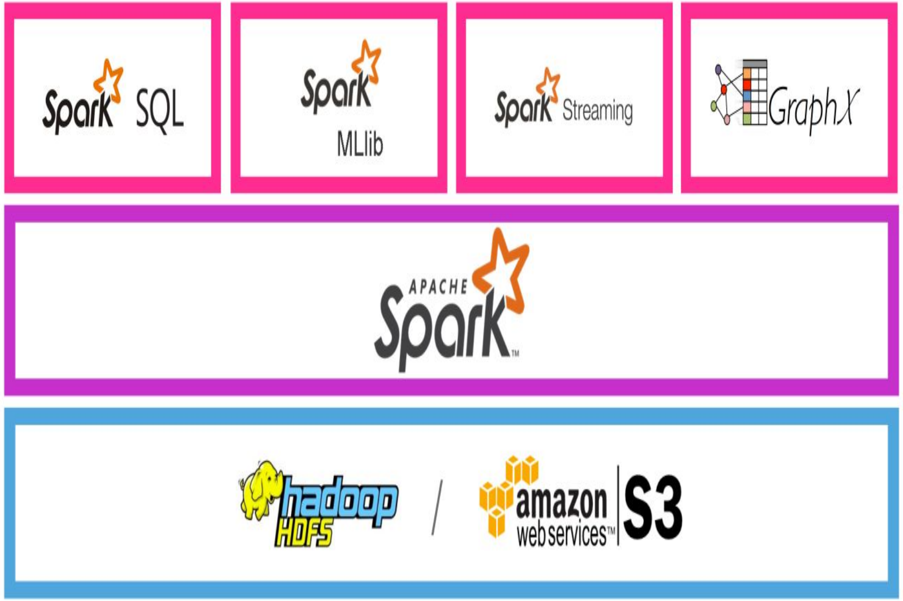
How to define Big Data?

- Volume
- Velocity
- Variety
- Variability
- Complexity

APACHE Spark™

Overview

- developed by Mateu Zaharia as a part of PhD theshis at UC Berkley 2013
- written in java
- distributed querying and processing engine
- up to 100 times faster than Apache Hadoop when data is stored in memory and up to 10 times when accessing disk

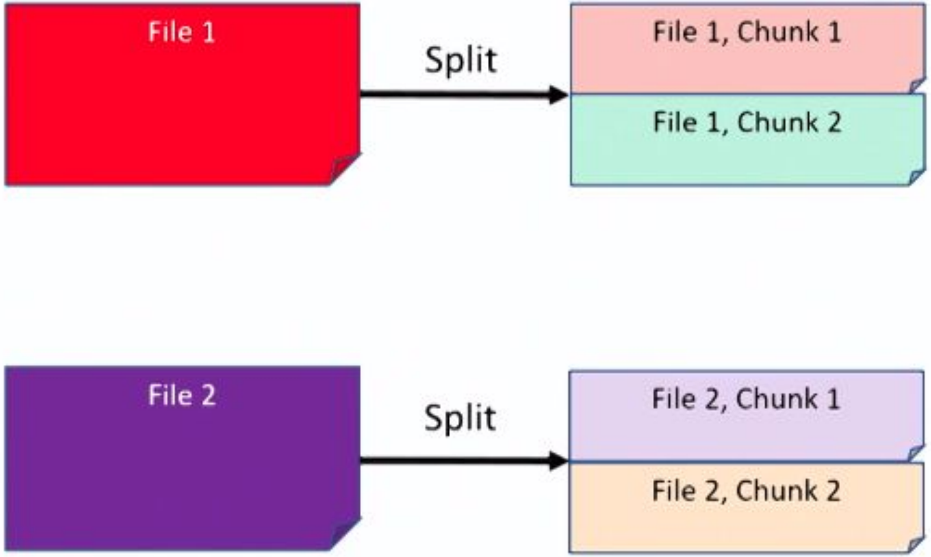


Spark ecosystem

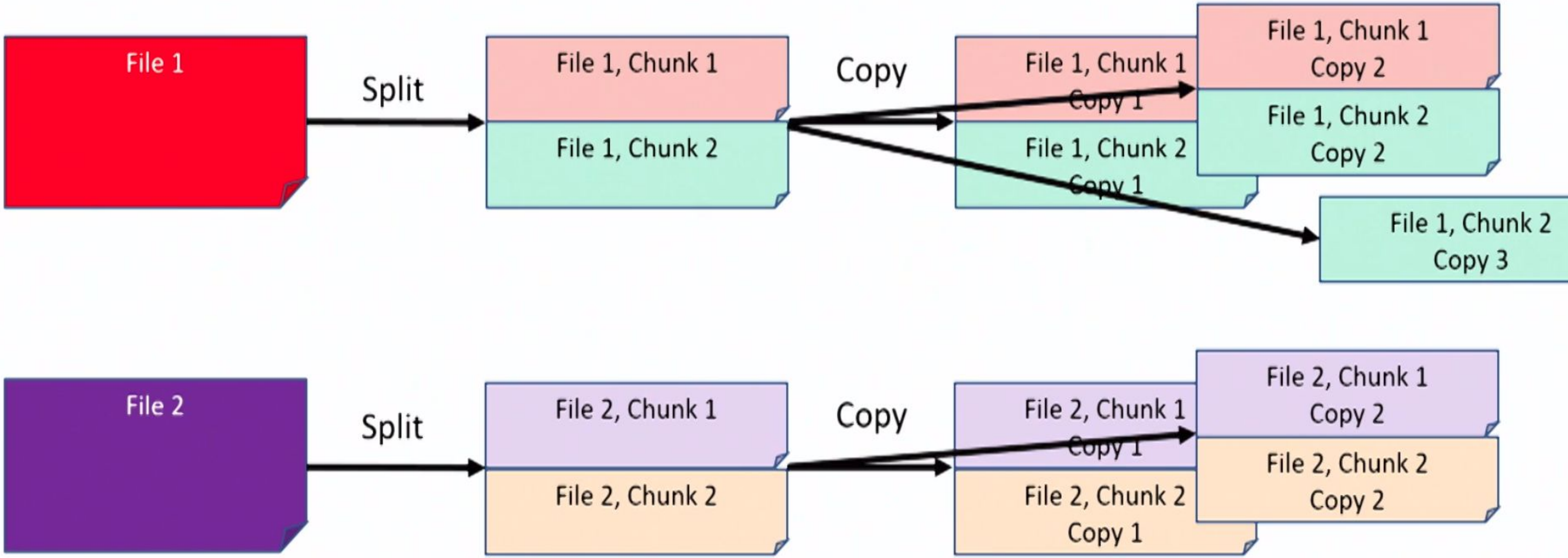
- Spark SQL
- Spark MLib
- Spark Streaming
- Graphx

1. HDFS

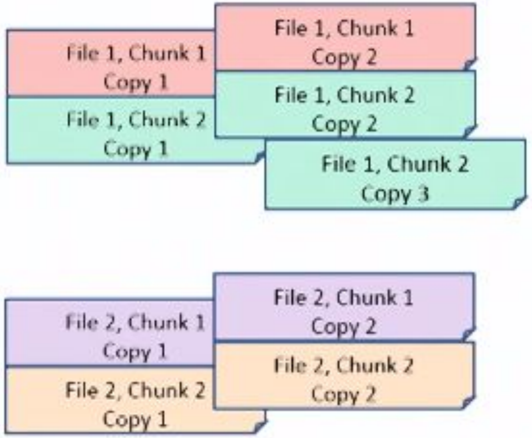
HDFS: Chunking files

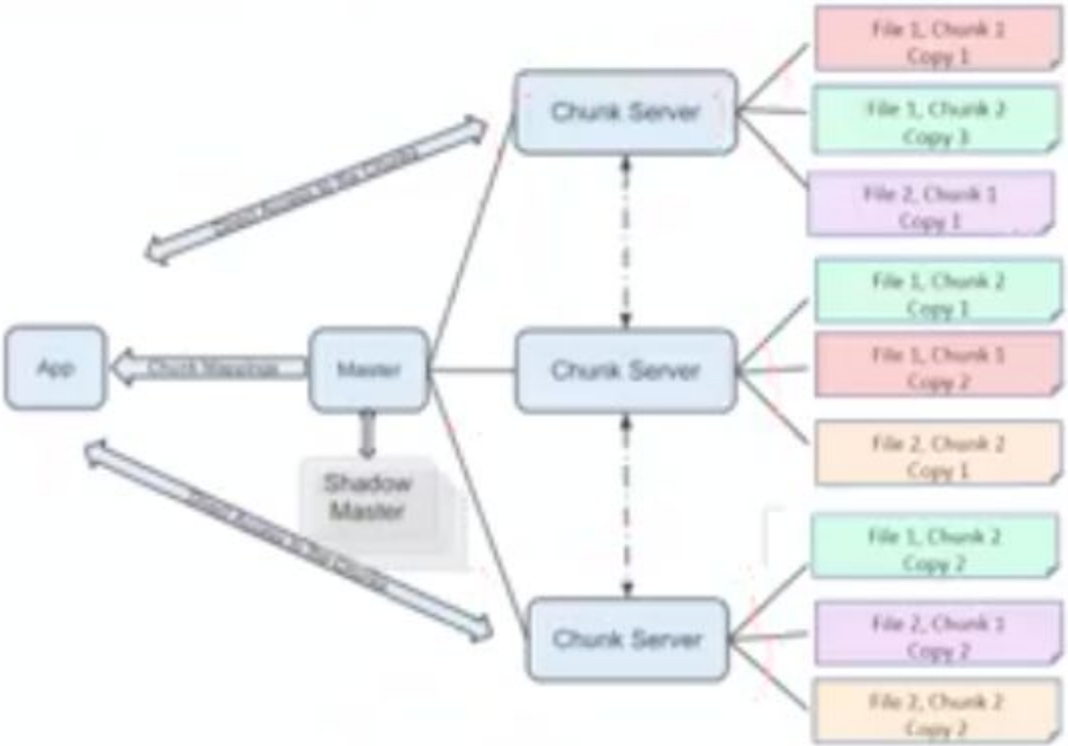


HDFS: Chunking files



HDFS: Distributing Chunks

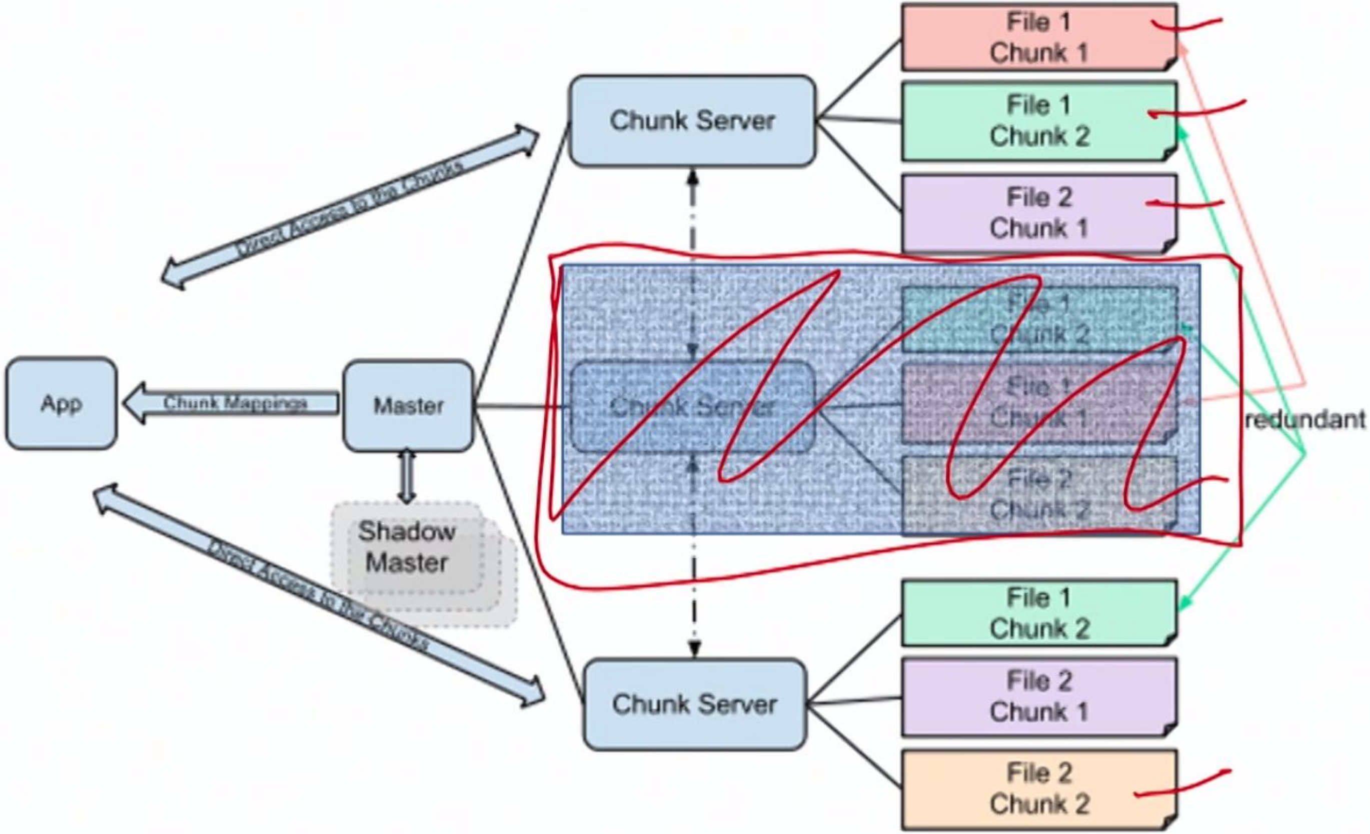




Properties of GFS/HDFS

- **Commodity Hardware** : Low cost per byte of storage
- **Locality** : data stored close to CPU
- **Redundancy** : can recorver from server failures
- **Simple abstraction** : looks to user like standard file system (files,directories,etc) Chunk mechanism is hidden

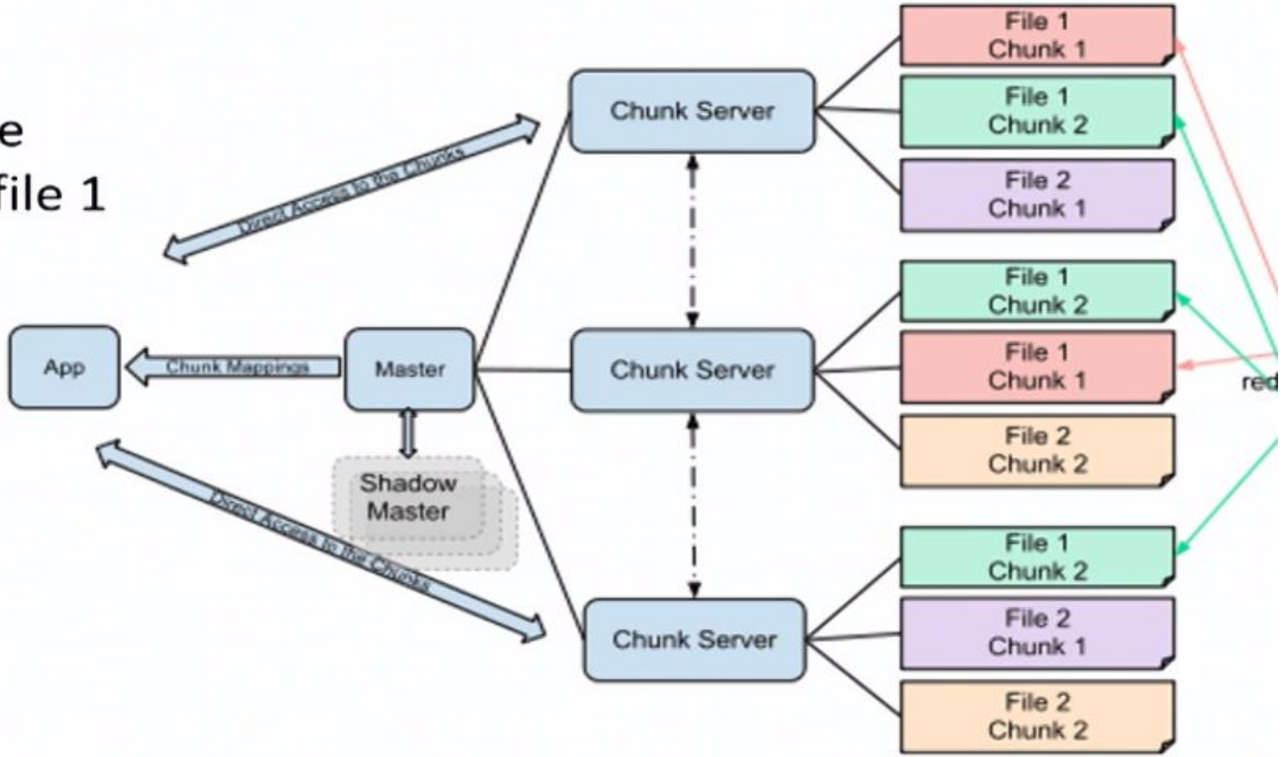
Redundancy



Chunk Server
File 1
Chunk 1
File 1
Chunk 2
File 2
Chunk 1
Direct Access to the Chunks
App
Chunk Mappings
Master
Shadow
Master
Chunk Server
File 1
Chunk 2
File 1
Chunk 1
File 2
Chunk 2
redundant
Direct Access to the Chunks
Chunk Server
File 1
Chunk 2
File 2
Chunk 1
File 2
Chunk 2

Locality

Task:
Sum all of the elements in file 1



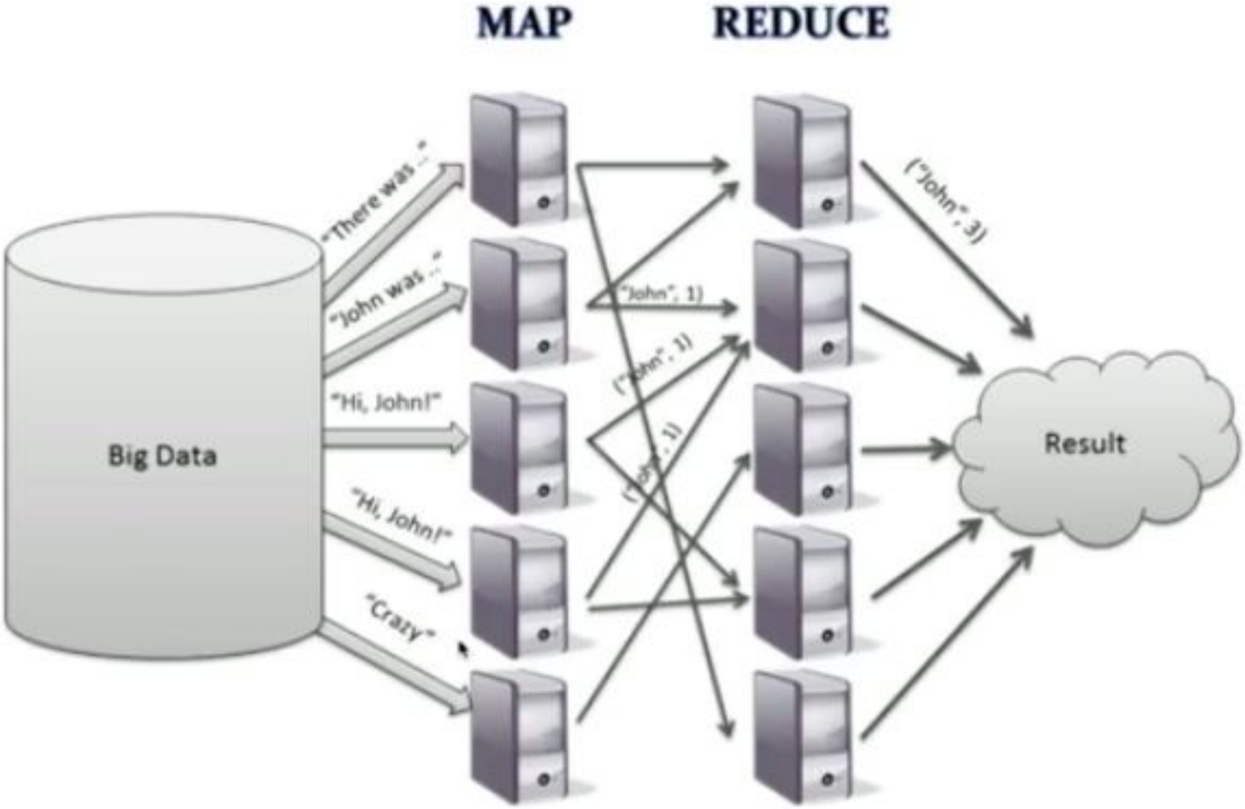
Map-Reduce

- HDFS is a **storage abstraction**
- Map-Reduce is a **computation abstraction** that works well with HDFS
- Allows programmer to specify parallel computation without knowing how hardware is organized

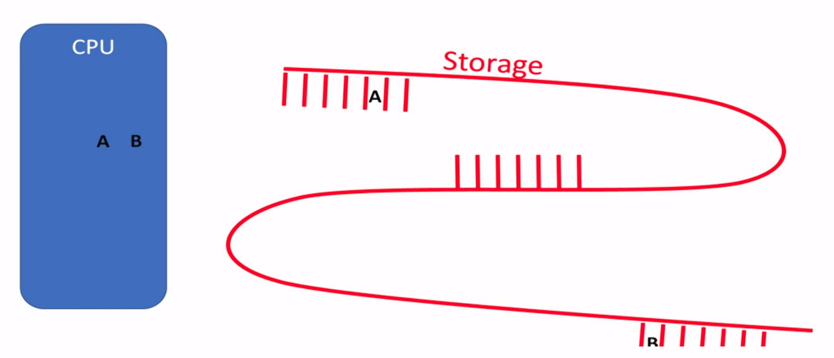
Summary

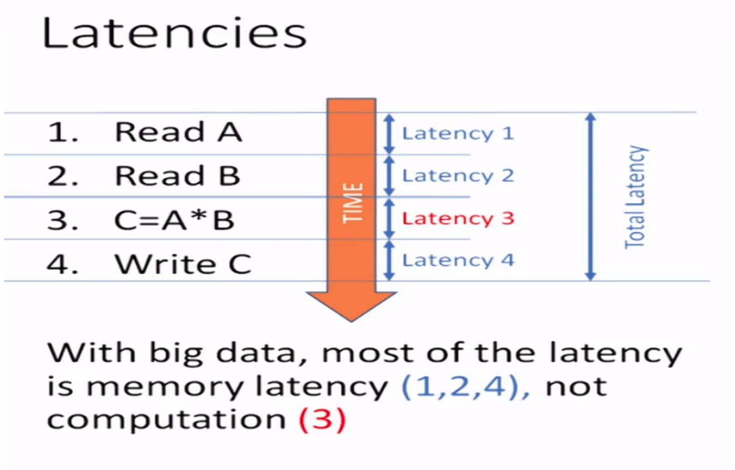
- Big data analysis is performed on large clusters of commodity computers
- HDFS (Hadoop file system) breakdown files to chunks, make copies, distribute randomly
- Hadoop Map-Reduce : a computation abstraction that works well with HDFS

2. Map-Reduce


MAP
REDUCE
Big Data
"There was .."
"John was .."
"Hi, John!"
"Hi, John!"
"Crazy"
"John", 1)
("John", 1)
("John", 1)
("John", 3)
Result

Data Latency





Latencies

1. Read A
2. Read B
3. C=A*B
4. Write C

TIME
Latency 1
Latency 2
Latency 3
Latency 4
Total Latency

With big data, most of the latency is memory latency (1,2,4), not computation (3)

- Major source of latency in data is reading and writing into storage
- Different types of storage offer different latencty, capacity and price
- Big Data analytics resolves around methods for organizing storage and computation in ways that maximize speed while minimizing cost.

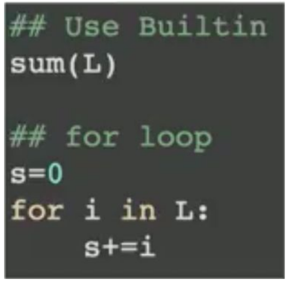
Cache computations

How to use computers clusters to efficently process large amounts of data?

Reduce : compute the sum

- list L = [3,1,5,7]
- find the sum(16)

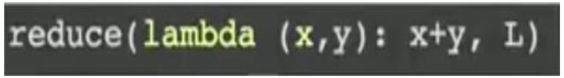
Traditional



```
## Use Builtin
sum(L)

## for loop
s=0
for i in L:
    s+=i
```

Map-Reduce

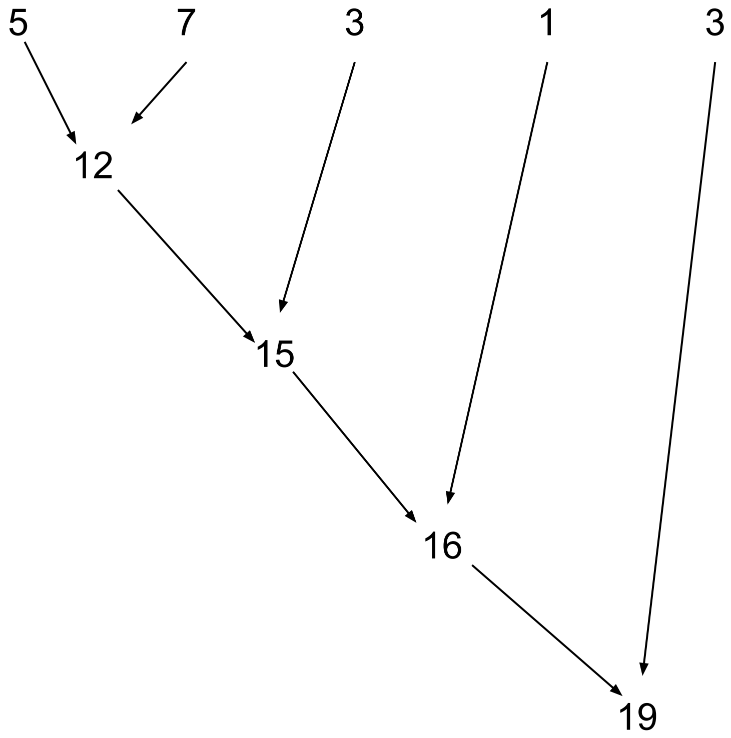


```
reduce(lambda (x,y): x+y, L)
```

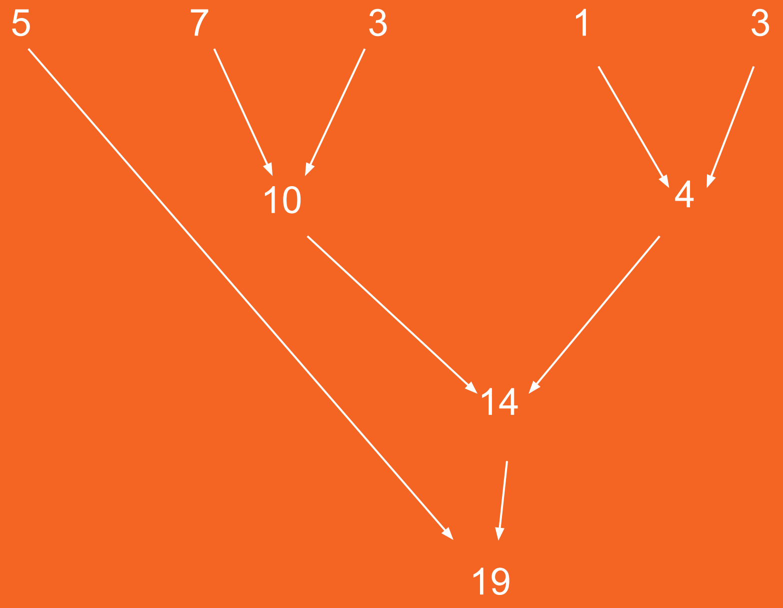
Order :

- Tradition : compute from first to last **in order**
- Map-Reduce : computation **order is not specified**

For loop order



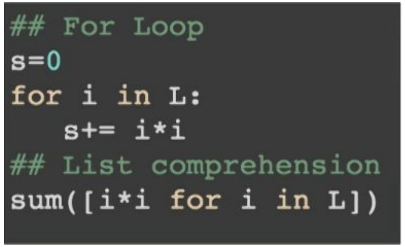
Parallel order



Map + Reduce

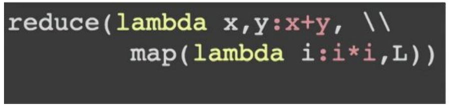
- list L = [3,1,5,7]
- compute the sum of the squares

Traditional



```
## For Loop
s=0
for i in L:
   s+= i*i
## List comprehension
sum([i*i for i in L])
```

Map-Reduce



```
reduce(lambda x,y:x+y, \\
        map(lambda i:i*i,L))
```

Order :

- Tradition : compute from first to last **in order**
- Map-Reduce : computation **order is not specified**

Execution :

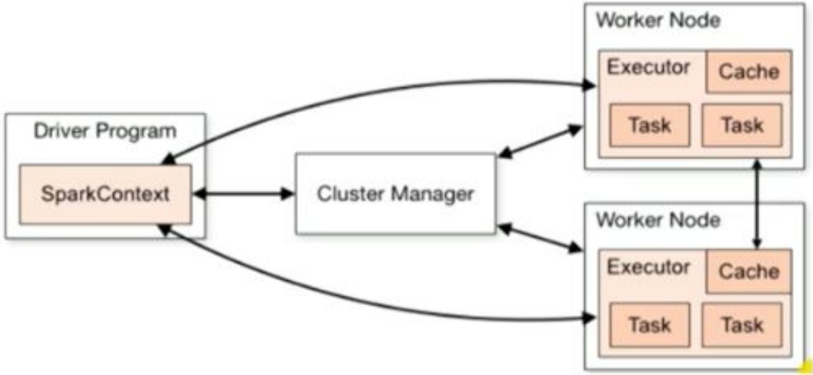
- Tradition : immediate execution
- Map-Reduce : execution plan

Map, Reduce operations should not depend on :

- Order of items in the list (**commutavity**)
- Order of operations (**assocaivity**)

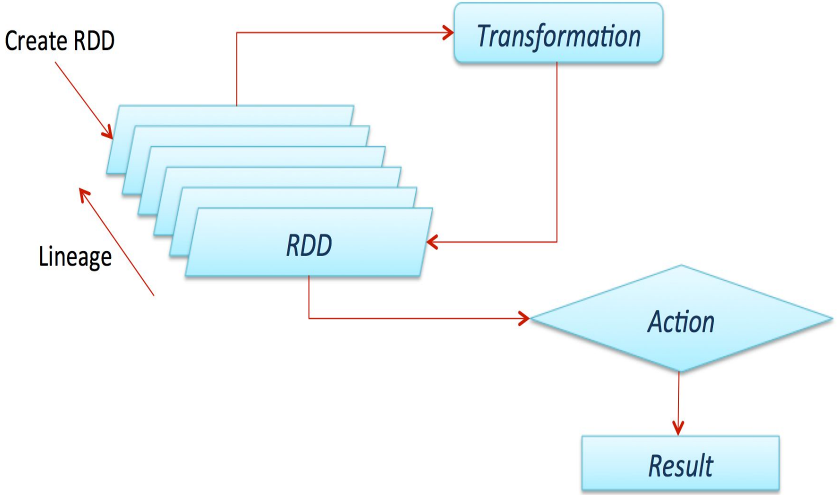
4. RDD

Spark software architecture

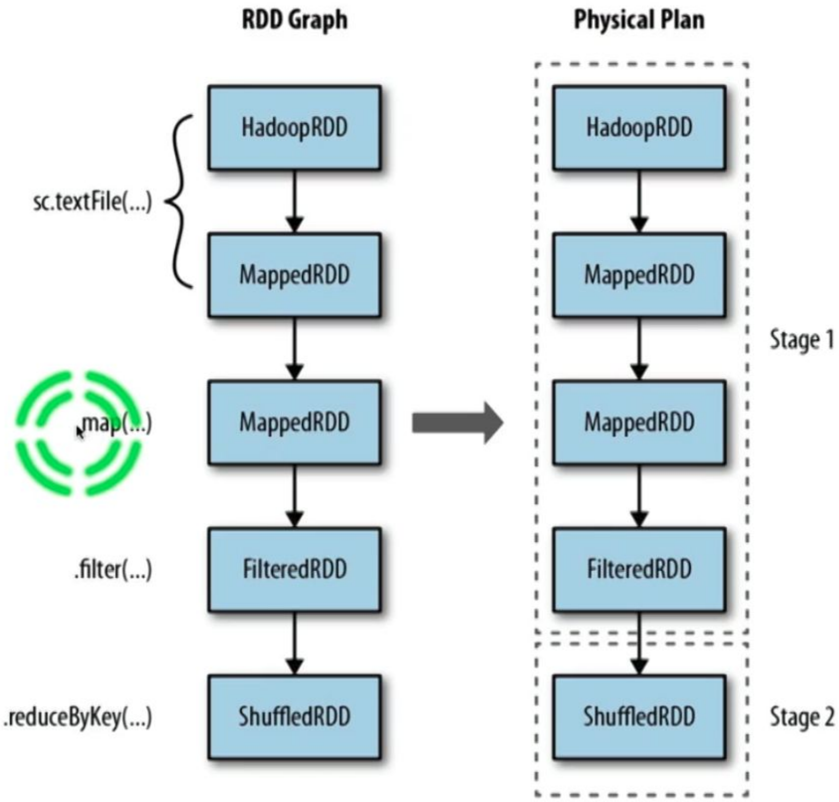


- Driver - runs on master, and execute the main of the program
- Cluster Master - manage the computational resources
- Workers - manage each single core,
  - manage partitions and executors
- Each RDD is partitioned among the workers
- Executors - execute tasks on their partition

RDD Processing



- **Transformation** -> computation of RDD from the previous one
- Action -> trigger the computation
- **Lineage ->** one RDD is computed from another computed from another...
- RDDs, on default, are not materialized
- They do materialize if cached or otherwise persisted



- RDD graph -> execution plan defined by programmer, computation in head node

- Stage1 -> set of operations that can be done before materialization. You can compute quite a lot of RDD’s because you don’t need to actually store them

- Stage 2 -> data materialized

Transformations

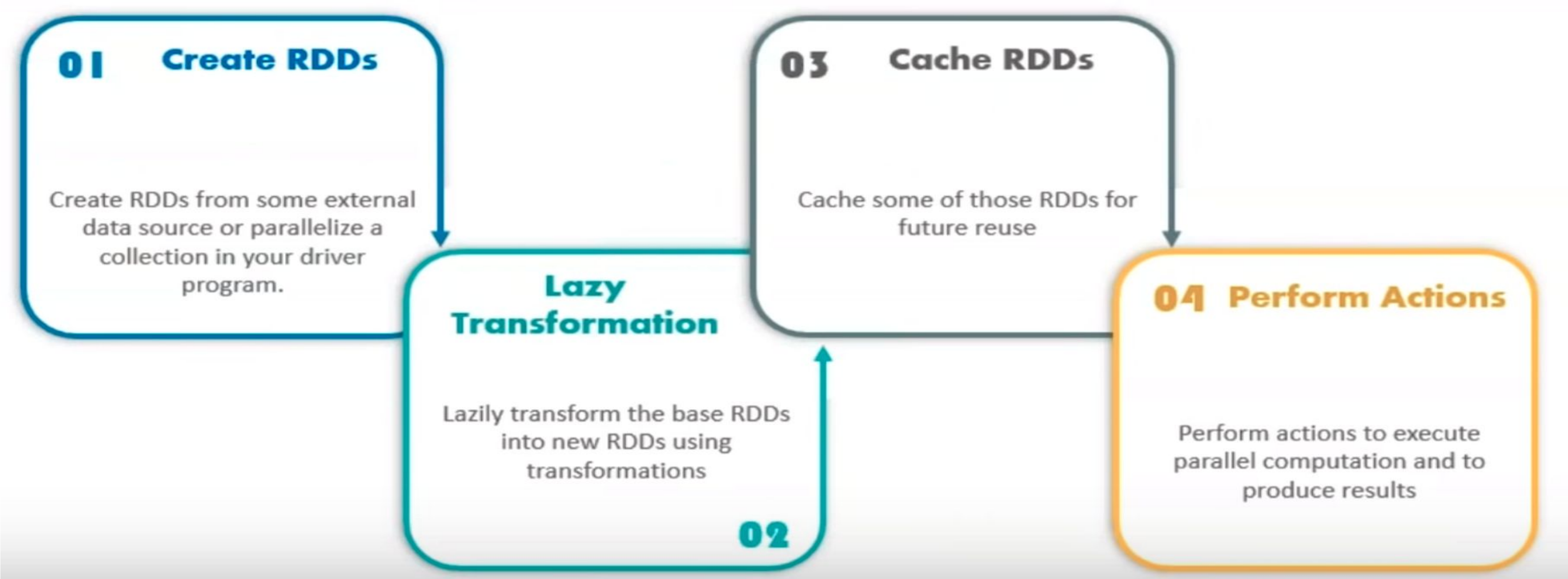
- map
- flatMap
- filter
- distinct
- reduceByKey
- mapPartitions
- sortBy

Actions

- collect
- collectAsMap
- reduce
- countByKey/countByValue
- take
- first

Life Cycle of spark program

To work on this ummutable data, you need to create a new one via Transformations and actions



01 Create RDDs

Create RDDs from some external data source or parallelize a collection in your driver program.

Lazy Transformation

Lazily transform the base RDDs into new RDDs using transformations

02

03 Cache RDDs

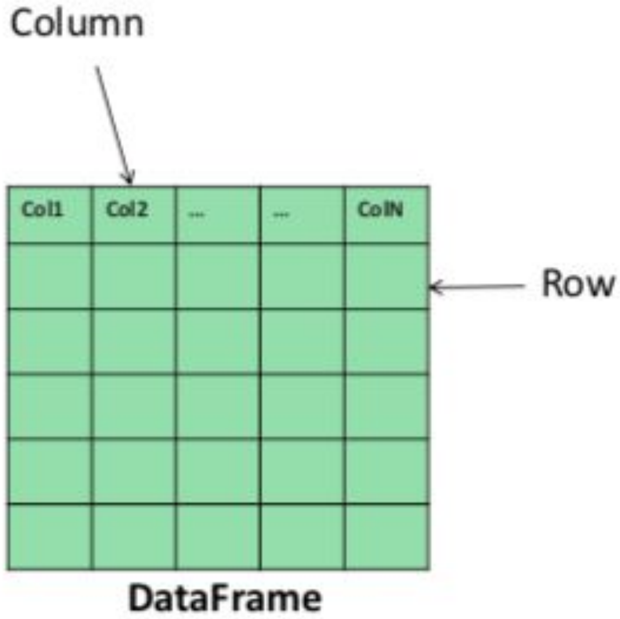
Cache some of those RDDs for future reuse

04 Perform Actions

Perform actions to execute parallel computation and to produce results

Notebooki

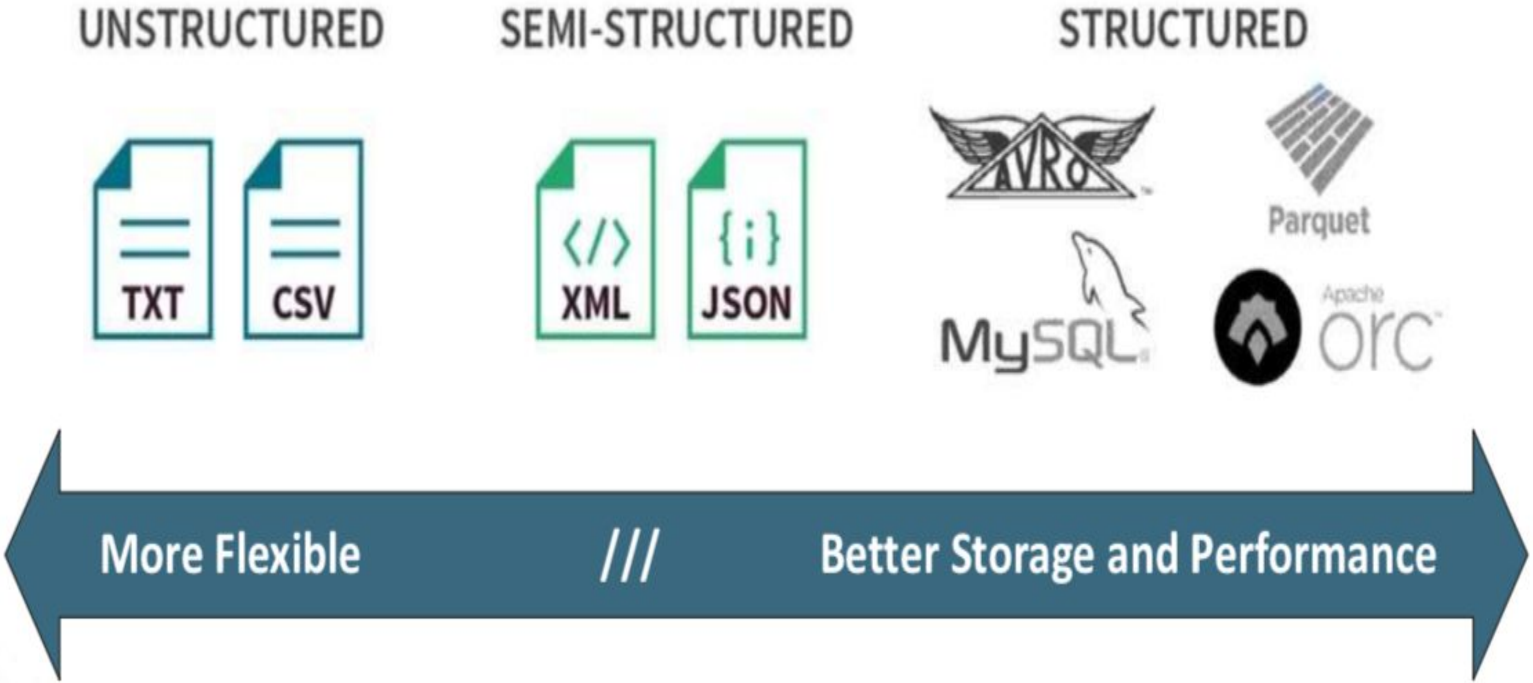
Spark DataFrame



- **Distributed collection** of **data organized** into ***named columns***

- Conceptually equivalent to a table in **relational DB** or a **data frame in Pandas**

- API available in Scala, Java, Python, and R



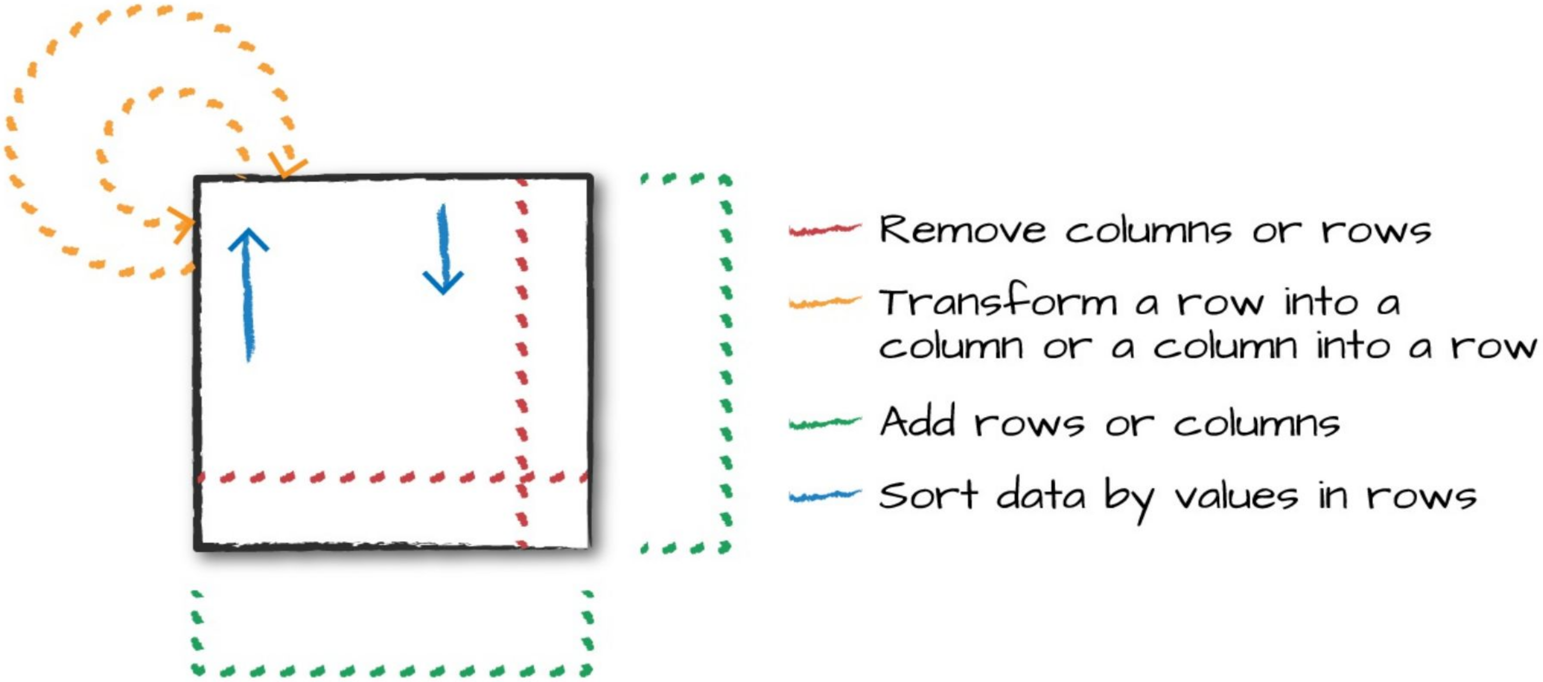
UNSTRUCTURED

SEMI-STRUCTURED

STRUCTURED

More Flexible /// Better Storage and Performance

DataFrame Transformations


Remove columns or rows
Transform a row into a column or a column into a row
Add rows or columns
Sort data by values in rows

Notebooki

Dziękuje za uwagę ;)