Resilient Distributed Datasets

Big Data Management

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

DTIM
www.essi.upc.edu/dtim

Knowledge objectives

1. Define RDD
2. Name the main Spark contributions and characteristics
3. Compare MapReduce and Spark
4. Distinguish between Base RDD and Pair RDD
5. Distinguish between transformations and actions
6. Explain available transformations
7. Explain available actions
8. Name the main Spark runtime components
9. Explain how to manage parallelism in Spark
10. Explain how recoverability works in Spark
11. Distinguish between narrow and wide dependencies
12. Name the two mechanisms to share variables
13. Enumerate some abstraction on top of Spark

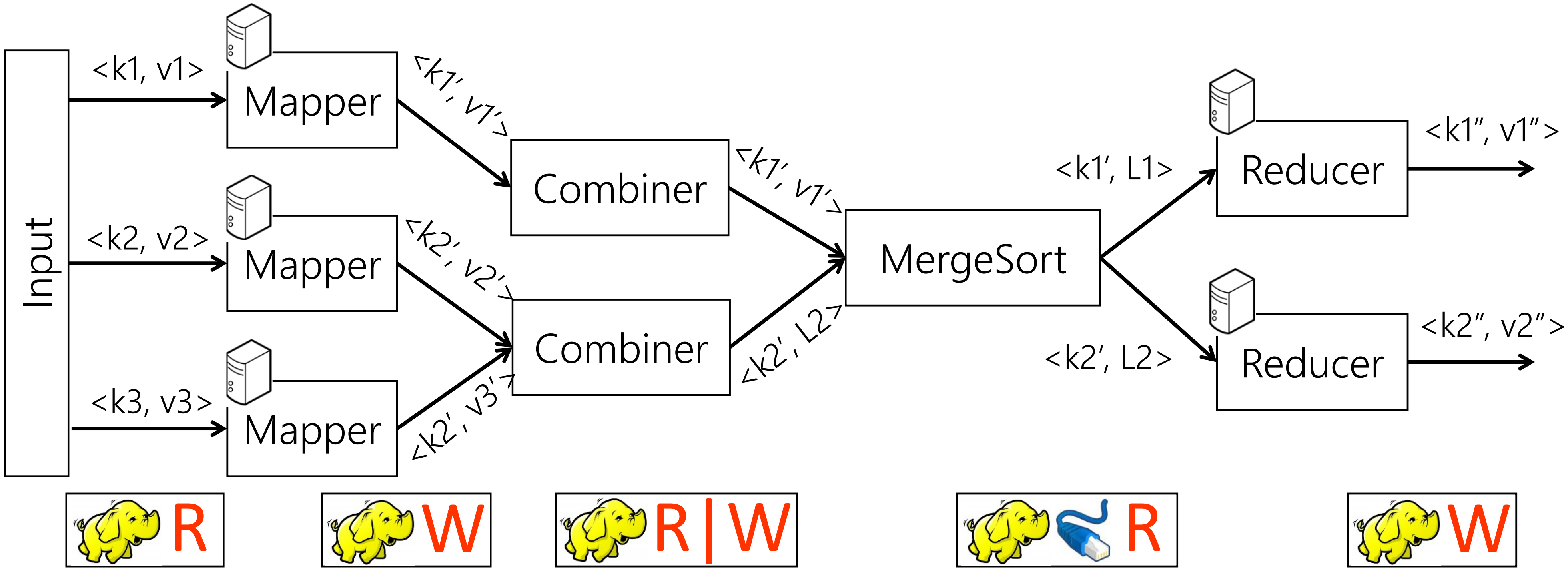
Application Objectives

- Provide the Spark pseudo-code for a simple problem

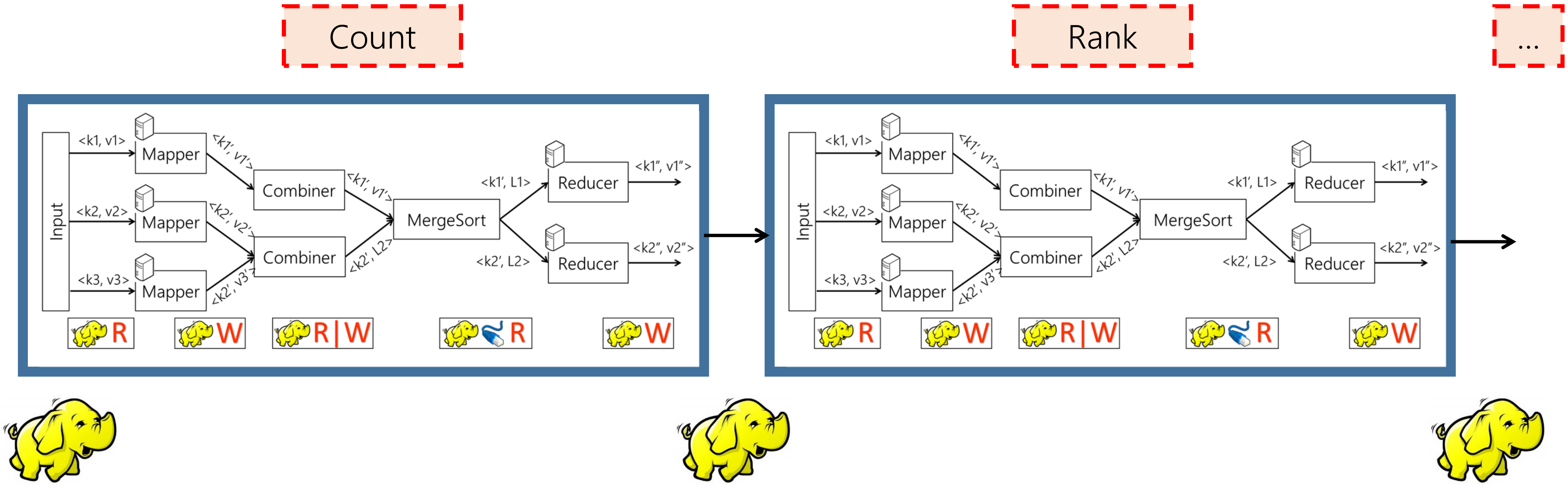
Background

MapReduce limitations

MapReduce intra-job coordination

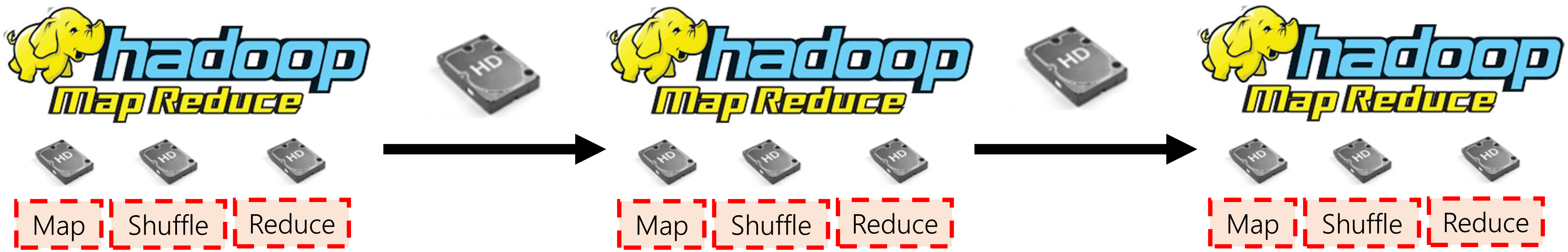


MapReduce inter-job coordination



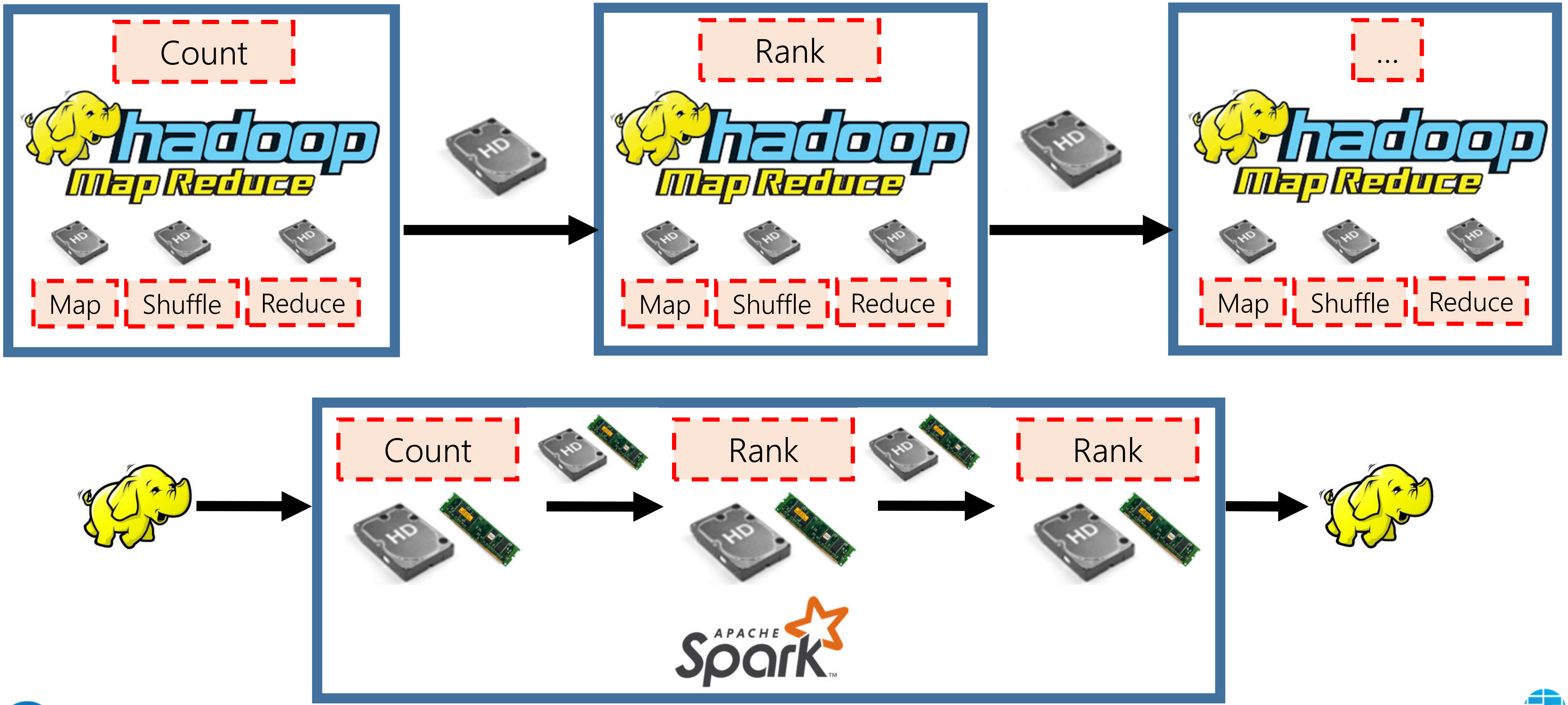
MapReduce limitations

- Coordination between phases using DFS
  - Map, Shuffle, Reduce
- Coordination between jobs using DFS
  - Count, rank, aggregate, …



Apache Spark

Main memory coordination



Resilient Distributed Datasets

- RDD
  - Resilient: Fault-tolerant
  - Distributed: Partitioned
  - Dataset: ….. a set of data

*"Unified **abstraction** for cluster computing, consisting in a **read-only**, partitioned collection of records. Can only be created through deterministic operations on either (1) data in stable storage or (2) other RDDs."*

rdd := spark.textFile("hdfs://…")

M. Zaharia

Types of RDDs in Spark

- Base RDD
  - `RDD<T>`
- Pair RDDs
  - `RDD<K,V>`
  - Particularly important for MapReduce-style operations
- Other specific types
  - `VertexRDD`
  - `EdgeRDD`
  - …

Characteristics

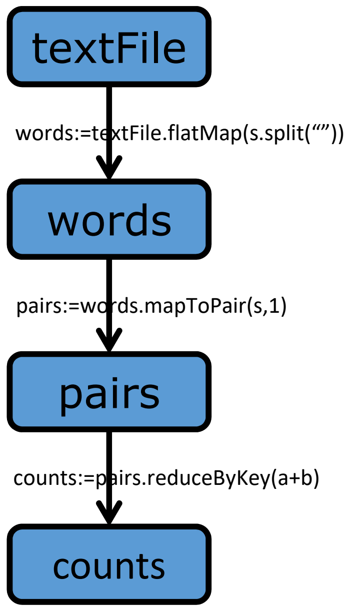
- Statically typed
- Parallel data structures
  - Disk
  - Memory
- User controls …
  - Data sharing
  - Partitioning (fixed number per RDD)
    - Repartition (shuffles data through disk)
    - Coalesce (reduces partitions in the same worker)
- Rich set of coarse-grained operators
  - Simple and efficient programming interface
- Fault tolerant
- Baseline for more abstract applications

Spark vs MapReduce

	MapReduce	Spark
Records	Key-Value pairs	Arbitrary
Storage	Results always in disk	Results can simply stay in memory
Functions	Only two	Rich palette
Partitioning	Statically	Dynamically

Example: Word count (Java)

```
JavaRDD<String> textFile = sc.textFile("hdfs://...");
JavaRDD<String> words = textFile.flatMap(s -> {
    return Arrays.asList(s.split(" "))
});
JavaPairRDD<String, Integer> pairs = words.mapToPair(s -> {
    return new Tuple2<String, Integer>(s, 1);
});
JavaPairRDD<String, Integer> counts = pairs.reduceByKey(a,b -> {
    return a + b;
});
counts.saveAsTextFile("hdfs://...");
```



Transformations and Actions

Apache Spark

Transformations vs. Actions

- Transformations
  - Applied to RDDs and generate new RDDs
  - They are run lazily
    - Only run when required to complete an action
- Actions
  - Trigger the execution of a pipeline of transformations
  - The result is …
    a) … a primitive data type (not an RDD)
    b) … data written to an external storage system

Transformations on base RDDs

map(f:T→U): RDD[T]→RDD[U]

filter(f:T→bool): RDD[T]→RDD[T]

sample(fraction: Float): RDD[T]→RDD[T] (deterministic)

flatMap(f:T→seq[U]): RDD[T]→RDD[U]

union/intersection/substract(): (RDD[T],RDD[T])→RDD[T]

cartesian(): (RDD[K],RDD[V])→RDD[(K,V)]

partitionBy(p:partitioner[T]): RDD[T]→RDD[T]

sort(c:comparator[T]): RDD[T]→RDD[T]

distinct(T): RDD[T]→RDD[T]

persist(): RDD[T]→RDD[T]

mapToPair(f:T→(K,V)): RDD[T]→RDD[(K,V)] (can be implicit)

https://spark.apache.org/docs/latest/api/java/index.html?org/apache/spark/api/java/RDD.html

Added transformations on pair RDDs

mapValues(f:V→W): RDD[(K,V)]→RDD[(K,W)] !

reduceByKey(f:(V,V)→V): RDD[(K,V)]→RDD[(K,V)]

groupByKey(): RDD[(K,V)]→RDD[(K,seq(V))]

join(): (RDD[(K,V)],RDD[(K,W)])→RDD[(K,(V,W))]

cogroup(): (RDD[(K,V)],RDD[(K,W)])→RDD[(K,(seq[V],seq[W])]

partitionBy(p:partitioner[K]): RDD[(K,V)]→RDD[(K,V)]

keys(): RDD[(K,V)] → RDD[K] (can be implicit)

values(): RDD[(K,V)] → RDD[V] (can be implicit)

https://spark.apache.org/docs/latest/api/java/index.html?org/apache/spark/api/java/JavaPairRDD.html

Actions on base RDDs

save(path: String): Writes the RDD to external storage (e.g., HDFS)

collect(): RDD[T]→seq[T] ☠

take(k): RDD[T]→seq[T]

first(): RDD[T]→T

count(): RDD[T]→Long

countByValue(): RDD[T]→seq[(T,Long)]

reduce(f:(T,T)→T): RDD[T]→T

foreach(f:T->U): RDD[T]→ - (executes in the workers)

https://spark.apache.org/docs/latest/api/java/index.html?org/apache/spark/api/java/RDD.html

Added actions on pair RDDs

countByKey(): RDD[(K,V)]→seq[(K,Long)]

lookup(k: K): RDD[(K,V)]→seq[V]

https://spark.apache.org/docs/latest/api/java/index.html?org/apache/spark/api/java/JavaPairRDD.html

Example

Analyzing HR data with Spark

Average satisfaction level

- Does the number of projects an employee works on affect their satisfaction level?
- CSV Dataset (HR_comma_sep.csv)
  - Satisfaction Level
  - Last evaluation
  - Number of projects
  - Salary
  - Time spent at the company (in months)

Sample data

0.38,0.53,2,3,low
0.8,0.86,5,6,medium
0.11,0.88,7,4,medium
0.72,0.87,5,5,low
0.37,0.52,2,3,low
0.41,0.5,2,3,low
0.1,0.77,6,4,low
0.92,0.85,5,5,high
...

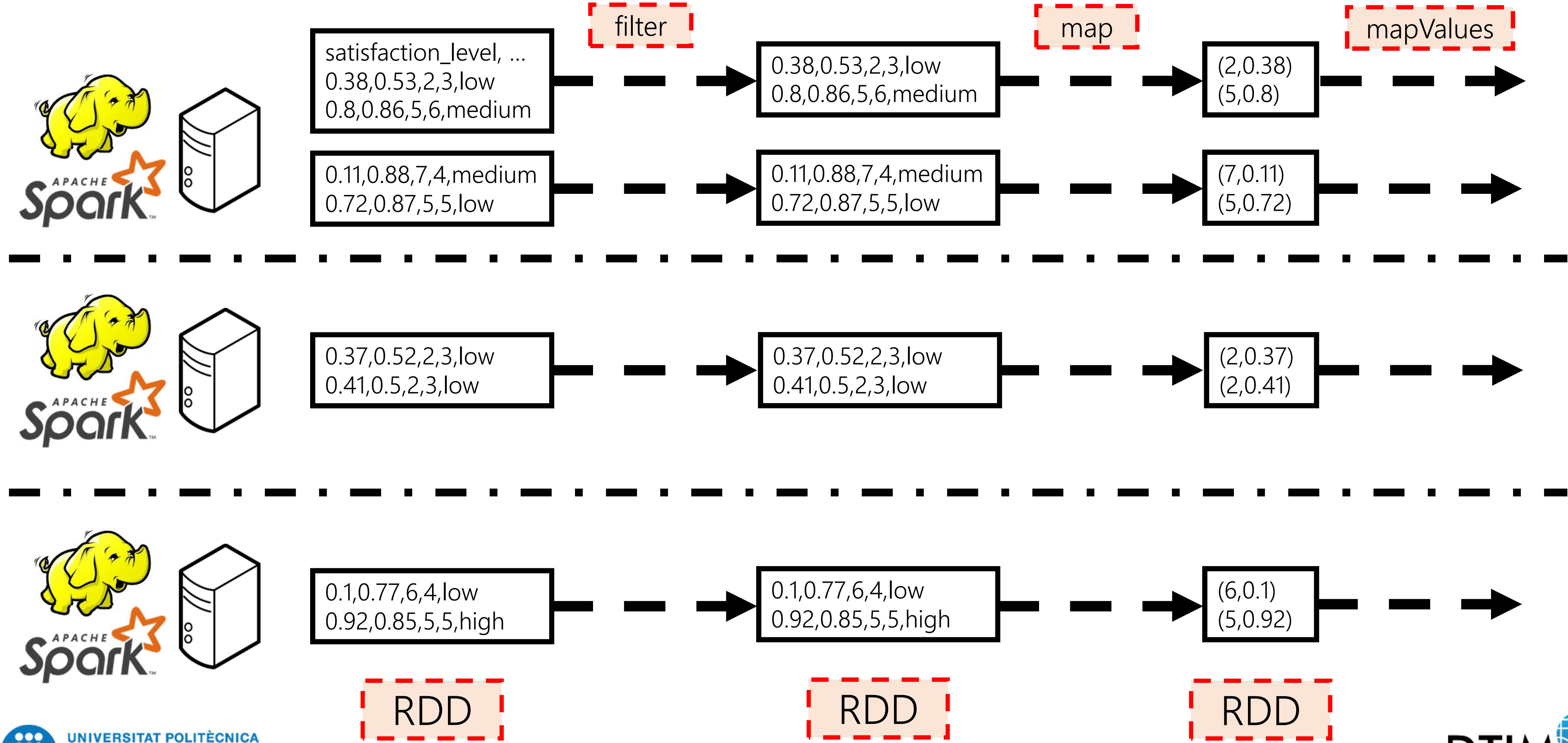
https://www.kaggle.com/liujiaqi/hr-comma-sepcsv

Implementation (Python)

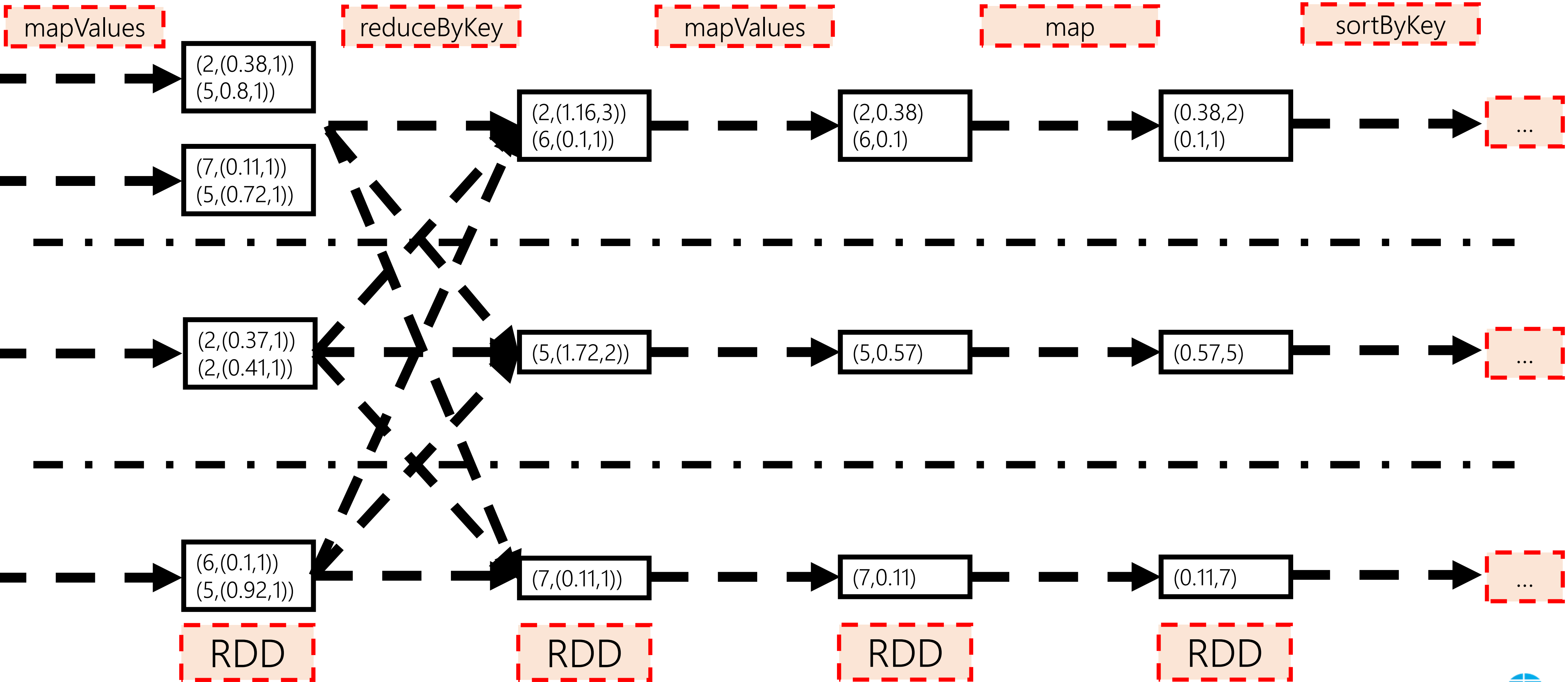
Average satisfaction level per number of projects, ordered from lowest to highest.

```
sc = pyspark.SparkContext.getOrCreate()

out = sc.textFile("HR_comma_sep.csv") \
    .filter(lambda t: "satisfaction_level" not in t) \
    .map(lambda t: (int(t.split(",")[2]), float(t.split(",")[0]))) \
    .mapValues(lambda t: (t,1)) \
    .reduceByKey(lambda a,b: (a[0]+b[0],a[1]+b[1])) \
    .mapValues(lambda t: t[0]/t[1]) \
    .map(lambda t: (t[1],t[0])) \
    .sortByKey()

for x in out.collect():
    print(x)
```

Runtime execution (I)



Runtime execution (II)

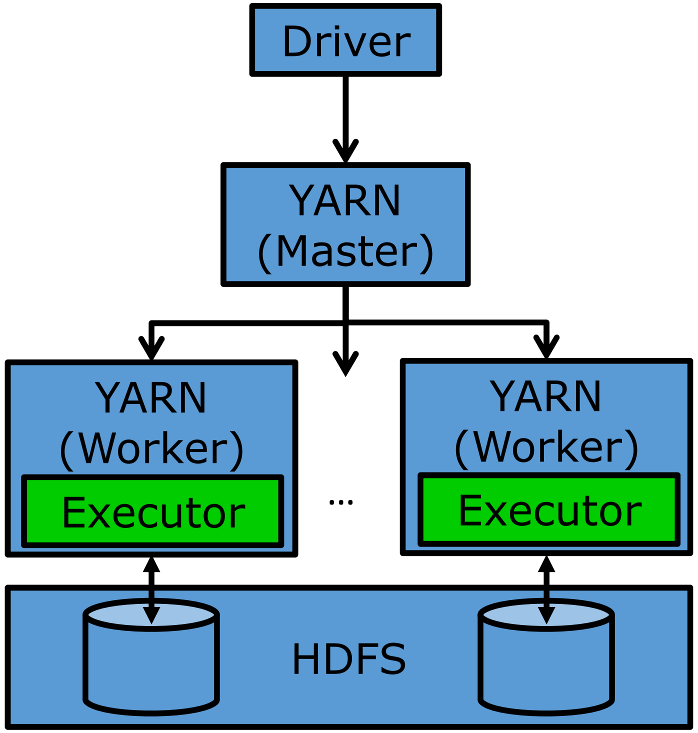


Under the hood

Apache Spark

Runtime architecture

- Driver
  - Creates the context
  - Decides on RDDs
  - Converts a program into tasks
  - Schedules tasks
  - Tracks location of cached data
- YARN (Master)
  - Resource manager
- Executors
  - Run tasks
  - Store data



Parallelism

- Degree is automatically inferred from partitions
- Too few parallelism
  - Wastes resources
  - Hinders work balance
- Too much parallelism
  - May generate significant overheads

RDD Abstraction Representation

- A set of dependencies on parent RDDs
- A function for computing the dataset
- Partitioning schema/metadata
  - Hash
  - Range
- A set of partitions
- Data placement
  - Partitions per node

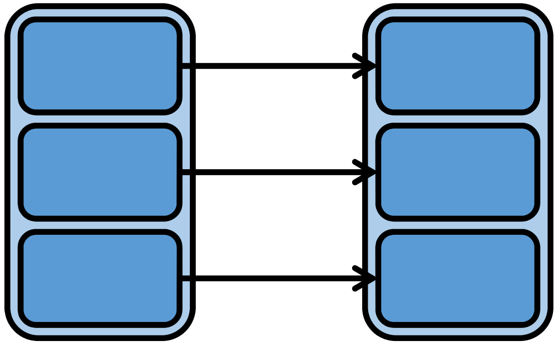
Partitioning

- Initially based on data locality
  - Useful based on keys
    - Hash
      - *partitionBy*
      - *groupByKey*
    - Range
      - *sortByKey*
- Partitions kept in workers' memory
- Different RDDs can use the same key
  - Similar to vertical partitioning
- Transformations lose partitioning information
  - *mapValues* retains partitioning information

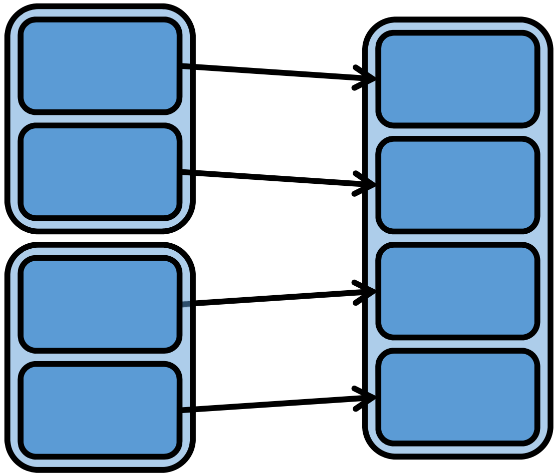
Optimization

- Lineage graph is translated into a physical execution plan:
  - Truncate the lineage graph to use cached results
  - Pipeline or collapse several RDD into one stage
    - If no data movement needed
- Decompose one job into several stages
  - Stages are decomposed into tasks per partition
    - Each task has three phases:
      1. Fetch data (from either local or remote disk)
      2. Execute operations
      3. Write result (for shuffling or returning results to driver)

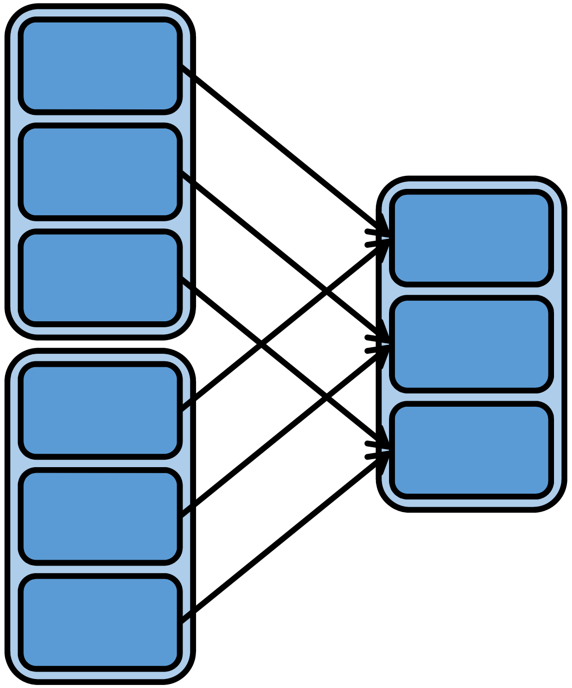
Narrow Dependencies



MapValue/Filter

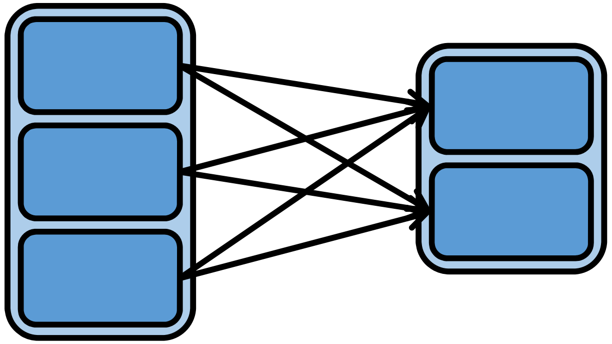


Union

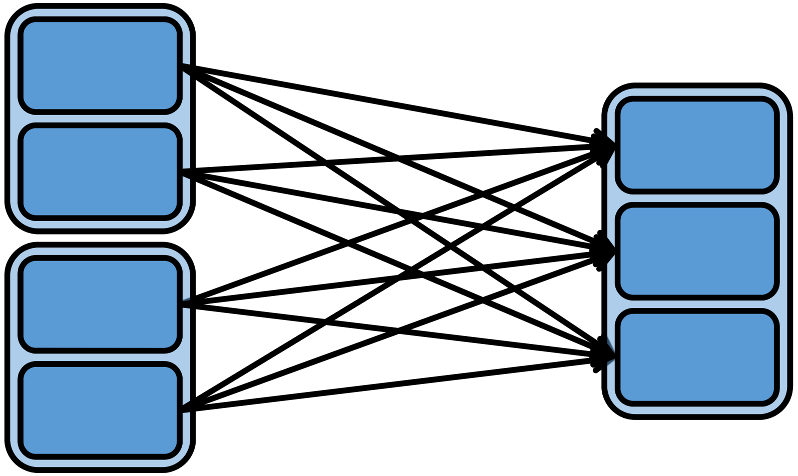


Join with inputs co-partitioned

Wide Dependencies

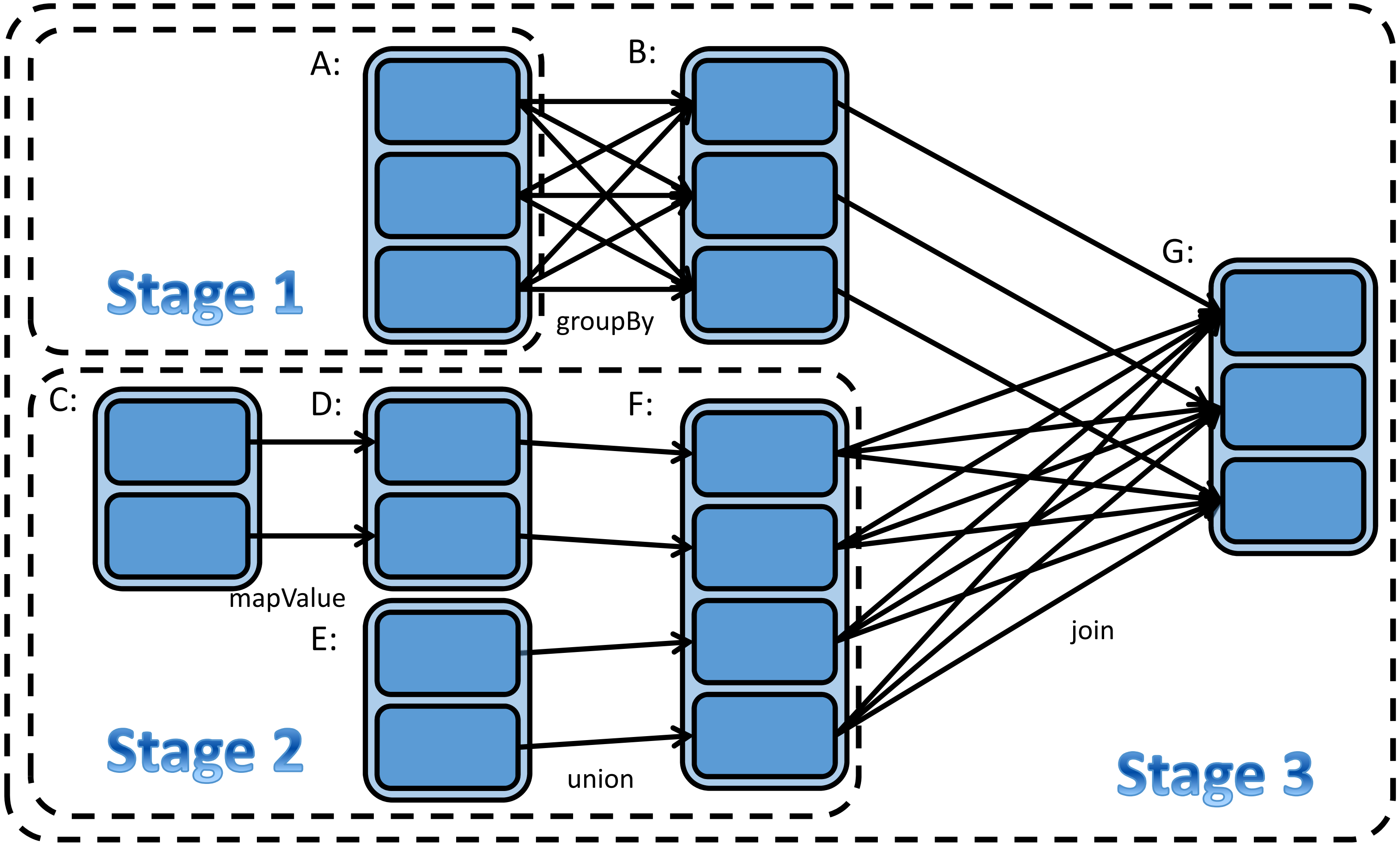


groupByKey



Join with inputs not co-partitioned

Scheduling



Recovery

- An RDD has enough information to be reconstructed after a failure
  - Lineage graph (logging not needed)
- Data can be cached/persisted in two nodes
  - Orthogonal to persistency options
  - Rule of thumb: cache an RDD if it is parent of more than one RDD

Storage Level	Meaning
MEMORY_ONLY	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, some partitions will not be cached and will be recomputed on the fly each time they're needed. This is the default level.
MEMORY_AND_DISK	Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, store the partitions that don't fit on disk, and read them from there when they're needed.
MEMORY_ONLY_SER (Java and Scala)	Store RDD as *serialized* Java objects (one byte array per partition). This is generally more space-efficient than deserialized objects, especially when using a fast serializer, but more CPU-intensive to read.
MEMORY_AND_DISK_SER (Java and Scala)	Similar to MEMORY_ONLY_SER, but spill partitions that don't fit in memory to disk instead of recomputing them on the fly each time they're needed.
DISK_ONLY	Store the RDD partitions only on disk.
MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc.	Same as the levels above, but replicate each partition on two cluster nodes.
OFF_HEAP (experimental)	Similar to MEMORY_ONLY_SER, but store the data in off-heap memory. This requires off-heap memory to be enabled.

https://spark.apache.org/docs/latest/programming-guide.html#rdd-persistence

Shared variables

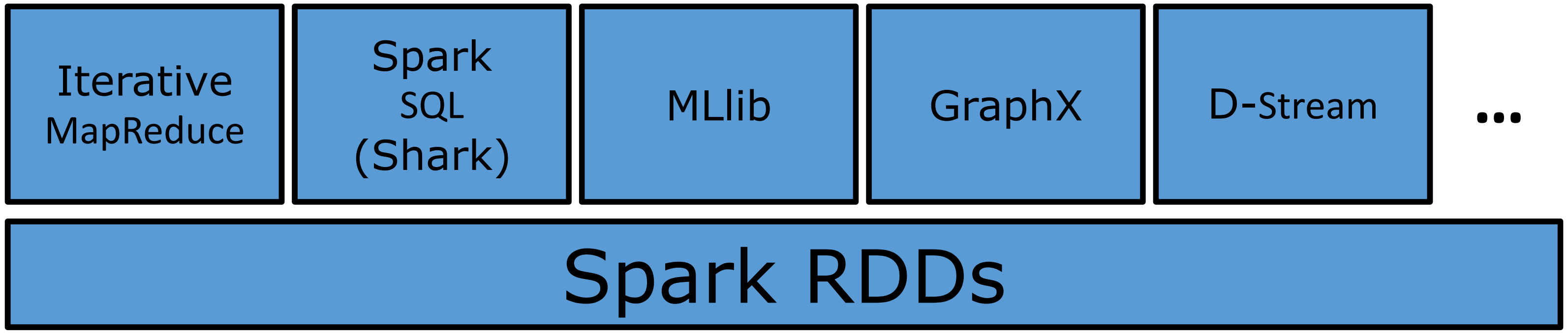
- Broadcast variables
  - Usage
    - Passed as a serializable object to the context
    - Accessed by workers (read-only)
  - Guarantees
    - The value is sent only once to each worker
- Accumulators
  - Usage
    - Initialized by the driver
    - Incremented by workers (write-only)
    - Value accessed by driver
  - Guarantees
    - Consistent inside actions
    - Unpredictable result inside transformations
      - In case of rexecution

Closing

Summary

- Resilient Distributed Datasets
    - Operations
        - Transformations
        - Actions
    - Persisting
    - Architecture
    - Dependencies
    - Scheduling
    - Partitioning
- Abstractions

Abstractions



References

- H. Karau et al. *Learning Spark*. O'Really, 2015
- M. Zaharia. *An Architecture for Fast and General Data Processing on Large Clusters*. ACM Books, 2016
- A. Hogan. *Procesado de Datos Masivos* (Universidad de Chile). http://aidanhogan.com/teaching/cc5212-1-2020