Spark II

Big Data Management

Knowledge objectives

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

Application Objectives

- Provide the Spark pseudo-code for a simple problem using RDDs

Resilient Distributed Datasets

Resilient Distributed Datasets

- RDD
  - Resilient: Fault-tolerant   do this by lineage graph
  - Distributed: Partitioned and parallel
  - Dataset: ….. a set of data

each RDD is not replica of each transformation
it is read only
we get new RDD for each transformation applied

"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://…")   example to create RDD

M. Zaharia

Types of RDDs in Spark

- Base RDD Superclass
  Object type
  - RDD<T>
- Pair RDDs Have key value pair similar to map reduce
  - RDD<K,V>
    - Particularly important for MapReduce-style operations
- Other specific types
  - Structured Stream
  - VertexRDD
  - EdgeRDD
  - …

RDD

Object
Object
…
Object

Dataframe

T_1	T_2	…	T_n
$1/A_1$	**$2/A_2$**	**…**	**n/A_n**
x	"x"	…	T
y	"y"	…	F
…	…		…
z	None	…	T

Diff
RDD - has only object
Dataframe - they are typed

Characteristics

- Statically typed — define type in compile time
- Parallel data structures
  - Disk
  - Memory

  RDD Data can be in disk or memory
  In either case, execution can happen in parallel
- User controls … — user controls when data is shared between one machine to another and partitioning
  - Data sharing
  - Partitioning (fixed number per RDD) — determines amount of parallelism you can get, more partition more parallelism
    - Repartition (shuffles data through disk) — expensive
      move data through network
    - Coalesce (reduces partitions in the same worker) — cheap, no movement of data
      simple, limited
- Rich set of coarse-grained operators — join, filter, union
  - Simple and efficient programming interface
- Fault tolerant
- Baseline for more abstract applications — if we use only RDF, it is limiting
  but on top of this other things can be built

MapReduce vs Spark

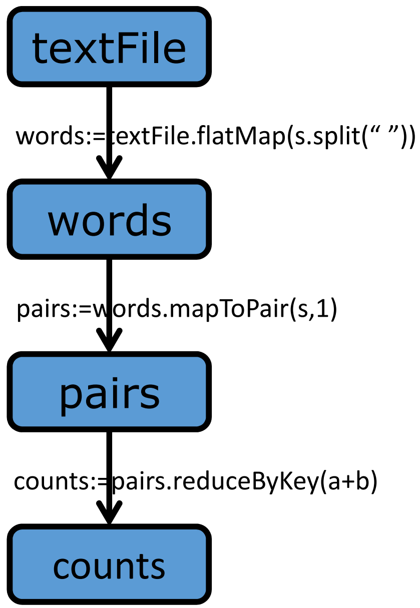
	MapReduce	Spark RDD
Records	Key-Value pairs	Arbitrary
Storage	Results always in disk	Results **can** simply stay in memory
Functions	Only two	Rich palette
Partitioning	Statically decided	Dynamically decided

in compile

in runtime number of partition can be determined using repartitioning and coalescing

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

pass function from one data datatype to another datatype
Result - From RDD of first type we get RDD of second type

1->1 map(f:T→U): RDD[T]→RDD[U]

pass function and RDD of one type and result is boolean
1 input -0 or 1 result
type same, filtered result

1->0-1 filter(f:T→bool): RDD[T]→RDD[T]

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

1->* flatMap(f:T→seq[U]): RDD[T]→RDD[U] for 1 input, sequence of element in output
output can be of different data type

union/intersection/substract(): (RDD[T],RDD[T])→RDD[T] union does not remove repetition
intersection / subtract -remove repetition

☠ cartesian(): (RDD[T1],RDD[T2])→RDD[(T1,T2)]

partitionBy(p:partitioner[T]): RDD[T]→RDD[T] returns same RDD but partitioned internally

sort(c:comparator[T]): RDD[T]→RDD[T]

distinct(T): RDD[T]→RDD[T]

persist(): RDD[T]→RDD[T] lazily evaluated, unless action is called nth is written to disk

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

relevant in Java
not relevant in Scala

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

Equivalent to map reduce - groupByKey + mapValues
Equivalent to combine - reduceByKey

Added transformations on pair RDDs

preserves partitioning based on key

!

mapValues(f:V→W): RDD[(K,V)]→RDD[(K,W)] does not modify key
content or type of value can be changed

reduceByKey(f:(V,V)→V): RDD[(K,V)]→RDD[(K,V)] done pair by pair
similar to combine
can only be used if commutative and associative

groupByKey(): RDD[(K,V)]→RDD[(K,seq(V))] else we can use groupByPair
for every key, there is sequence of values
on top of this, n=mapValues can be used

join(): (RDD[(K,V)],RDD[(K,W)])→RDD[(K,(V,W))] Joins by key for matching key, generates combination of values

cogroup(): (RDD[(K,V)],RDD[(K,W)])→RDD[(K,(seq[V],seq[W])] one pair for each matching key

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

sortByKey(): RDD[T]→RDD[T]

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

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

in java, it is explicit
in scala, it is implicit

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

Actions on base RDDs

Has RDD as parameter
But do not generate RDD

- save(path: String): Writes the RDD to external storage (e.g., HDFS)
- ☠ collect(): RDD[T]→seq[T] sends all data in RDD to driver — danger: we have alot of data in worker so whole data is sent to driver
- take(k): RDD[T]→seq[T] similar to collect but set some limit
- first(): RDD[T]→T take first element
- count(): RDD[T]→Long count number of elements in RDD
- countByValue(): RDD[T]→seq[(T,Long)] value and count for each value
- reduce(f:(T,T)→T): RDD[T]→T for every pair of values, we get one
- foreach(f: → -): RDD[T]→ - is distributed (executes in the workers)

not executed in driver/coordinator

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]

Pass value for key and we get all values associated with that key

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

Example

Analyzing HR data with RDDs

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
  - Time spent at the company (in months)
  - Salary

Sample data

satisfaction_level, ...
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)

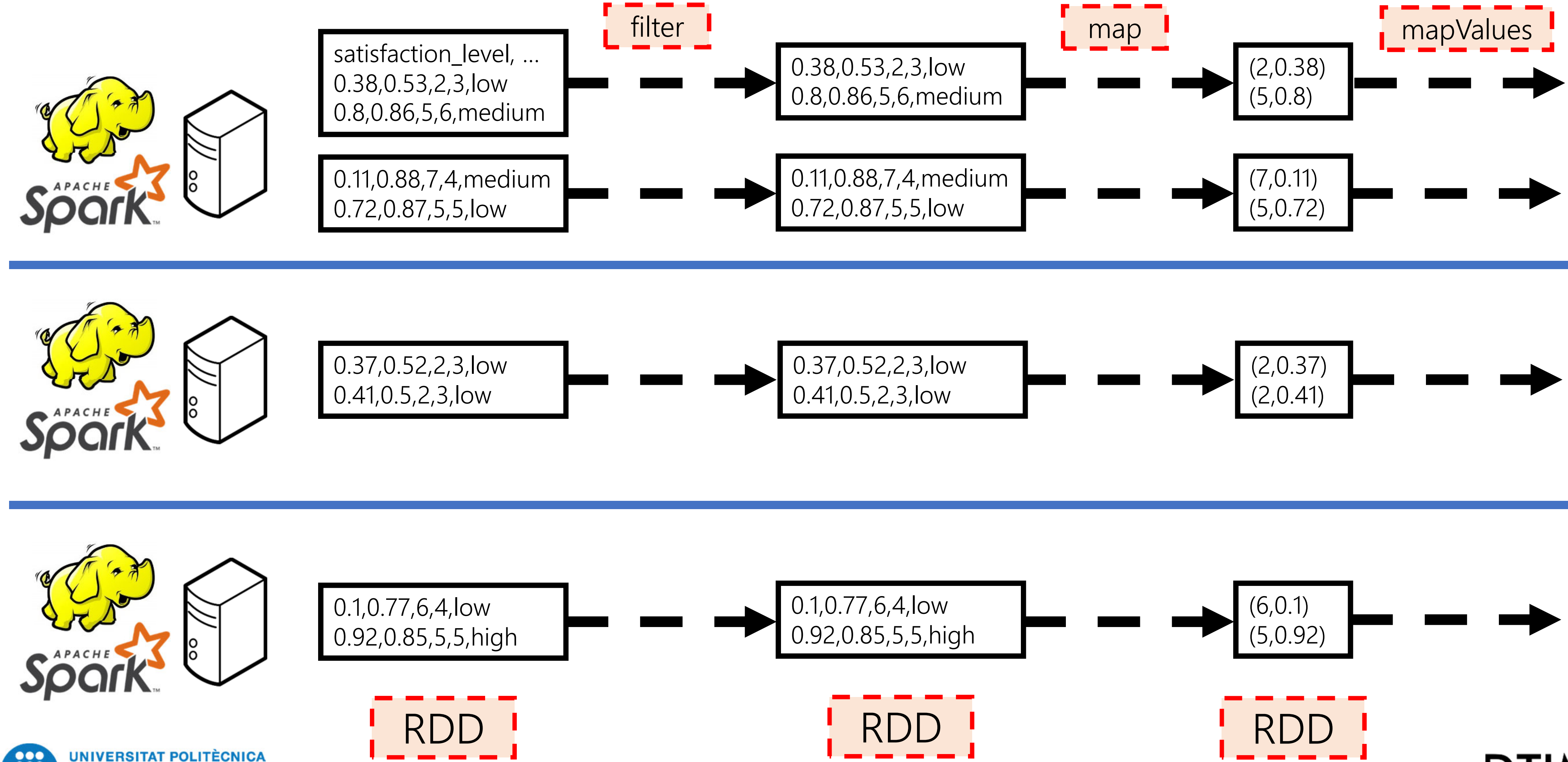
"Average satisfaction level per number of projects, ordered from lowest to highest"

filter -Remove first line
map - take two columns
mapValues - add counter 1
reduceByKey - adding satisfaction and counter
mapValues - find ratio
map - reverse order

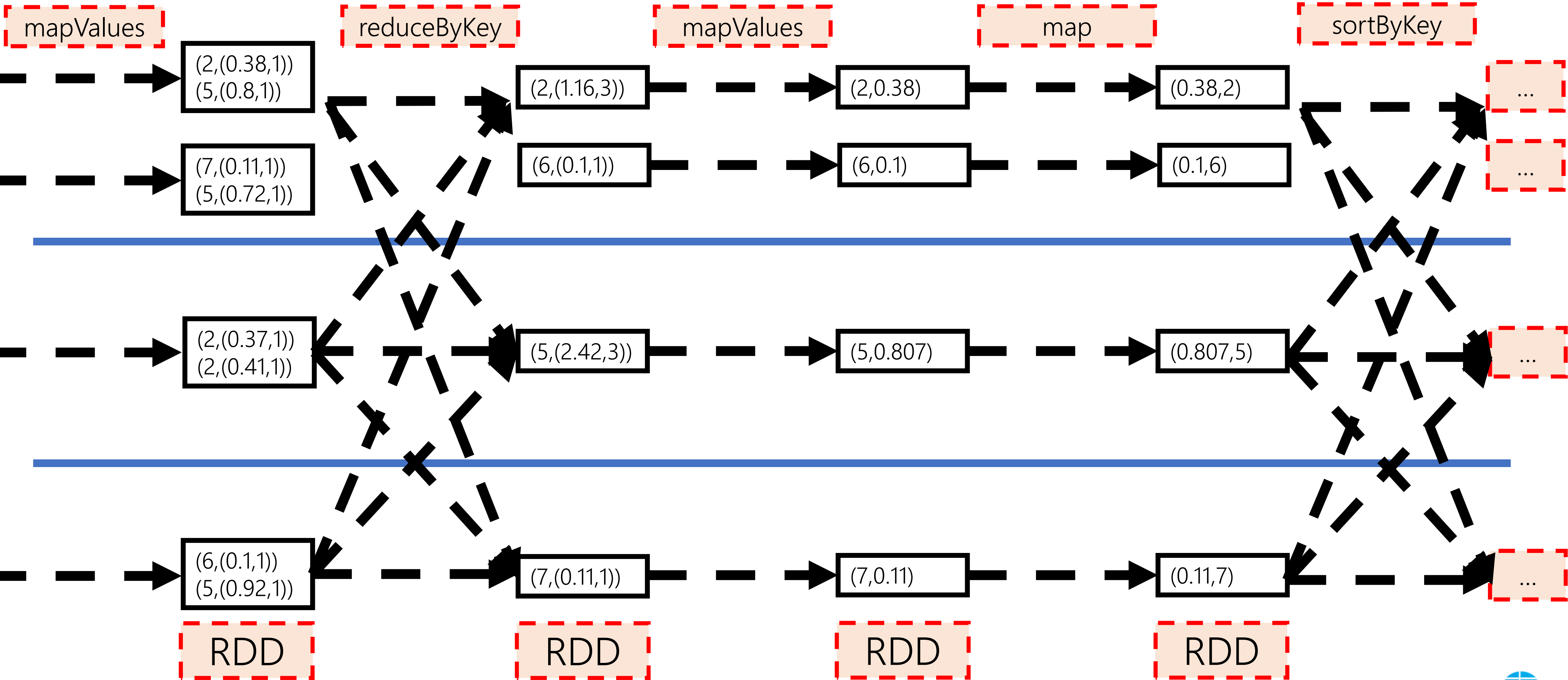
```
sc = pyspark.SparkContext.getOrCreate()

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)
```

collect is action

Runtime execution (I)



Runtime execution (II)

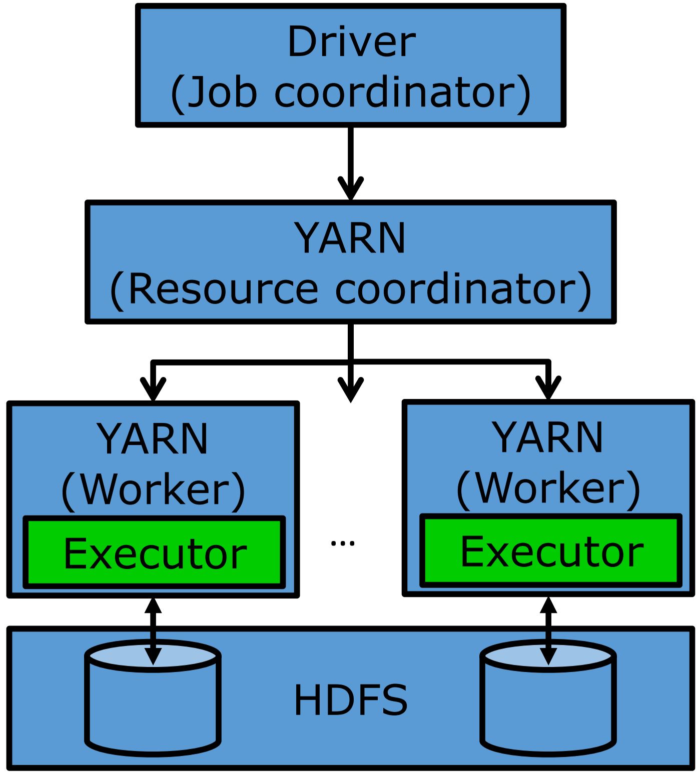


Under the hood

Apache Spark

Runtime architecture

- Driver (Job coordinator)
  - Creates the context
  - Decides on RDDs
  - Converts a program into tasks
  - Schedules tasks
  - Tracks location of cached data
- YARN (Resource coordinator)
  - Resource manager   Task scheduled needs resources
- Executors (Job worker)
  - Run tasks
  - Store data



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

1. In actual, RDD does not contain data
2. It is an abstraction
3. It just contains information that is needed to manage/generate or regenerate data

Parallelism

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

Number of partitions determine parallelism that can be achieved
Less partition - less parallelism
More partition - more parallelism

Partitioning

- Initially based on data locality based on number of blocks and chunks of input file
  - 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

if mapValues is used instead of map, key remains same and remains in same partition

Optimization

1. RDD contains information about where input comes from , who are parent, i.e. **Lineage Graph**
   1. This is translated into physical execution plan
   2. Tries to use cached result
   3. Pipelining as much as possible in memory. Pipeline involving several RDDs happen in same stage so does not go to disk.
      1. This is opposite to map reduce where each read write happens in disk
      2. But in spark, written in disk only when it is necessary

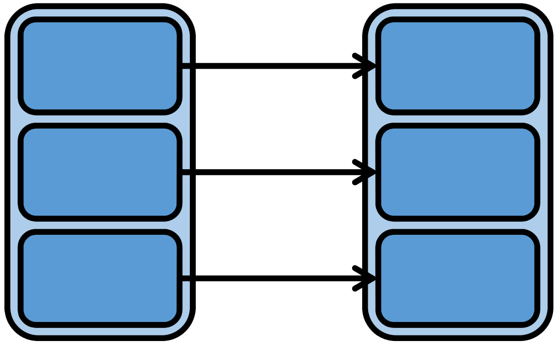
- Lineage graph is translated into a physical execution plan:
  - Truncate the lineage graph to use cached results
  - Pipeline or collapse several RDD into one stage

    What goes inside same stage?
    That does not require movement
    - If no data movement needed

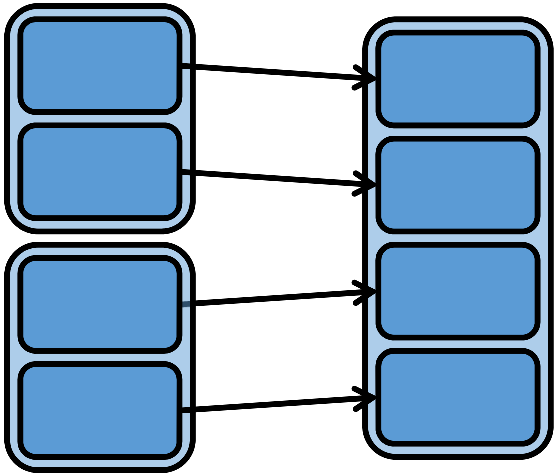
      pipelining in memory
- Decompose one job into several stages
  - Stages are decomposed into tasks per partition
    - Each task has three phases:
      1. Fetch data (from either local or remote disk) in general remote
      2. Execute operations
      3. Write result (for shuffling or returning results to driver)

Narrow Dependencies

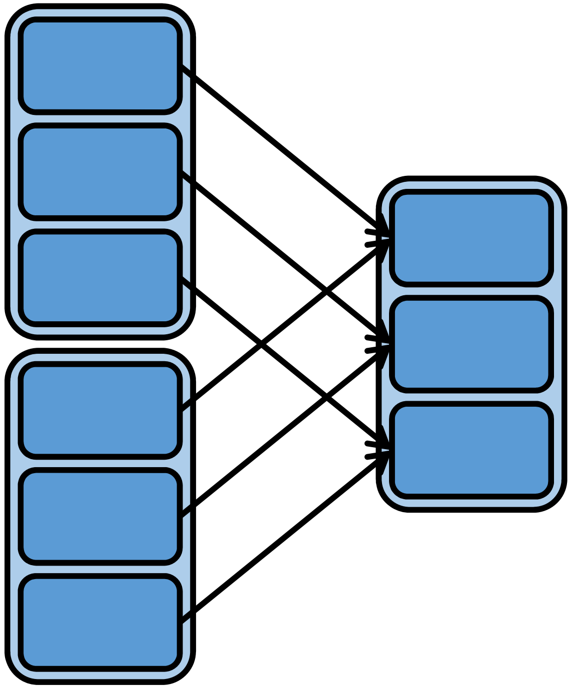
They do not change key
Maps one to one
No shuffling



MapValue/Filter



Union

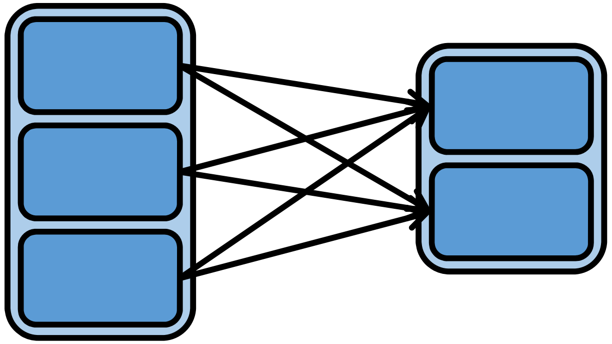


Join with inputs co-partitioned

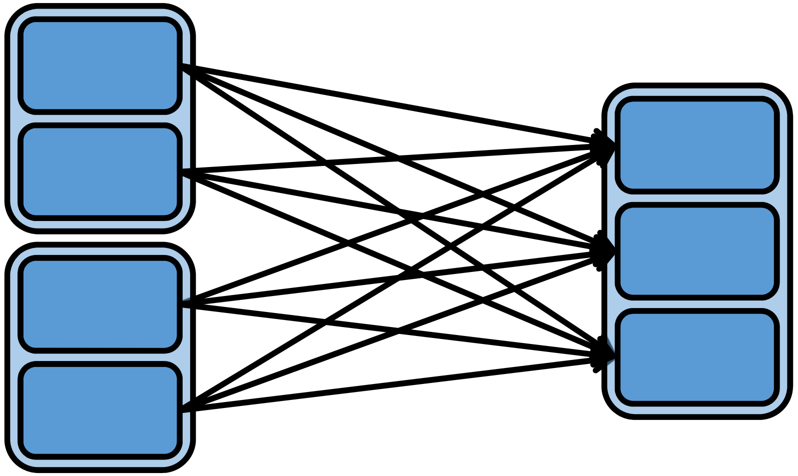
Wide Dependencies

Some pair goes to one machine some goes to other machine
Shuffling happens

determines lineage of stages



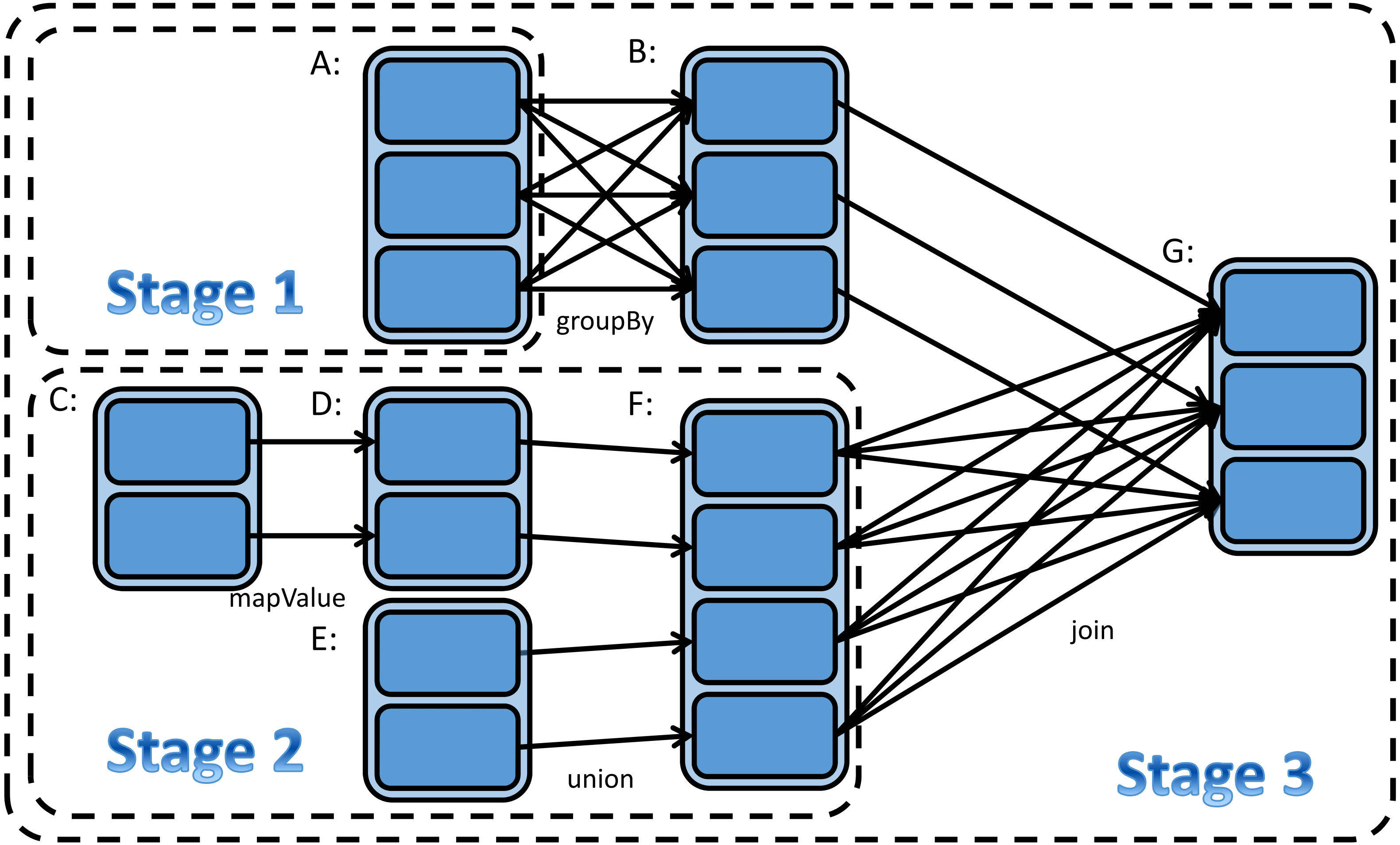
groupByKey



Join with inputs not co-partitioned

Scheduling

narrow dependencies bhako ko euta stage bancha
wide dependecies ko different stage bancha
Number of task = number of partitions



Recovery

- An RDD has enough information to be reconstructed after a failure
  - Lineage graph (logging not needed)
- Data can be cached/persisted (in up to two nodes) if one node fails another can be used
  - 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.

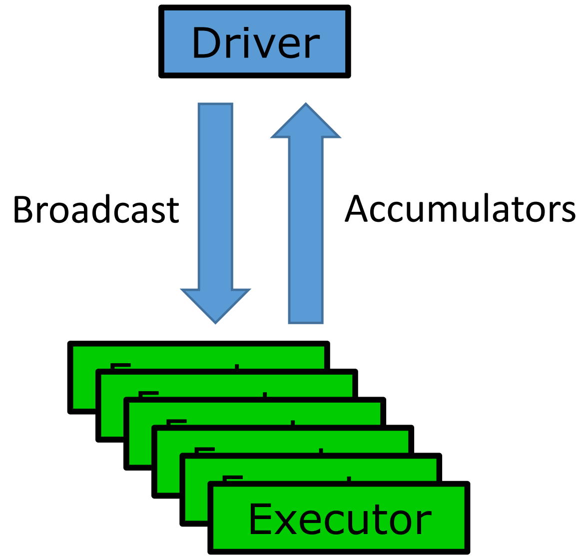
How to cache? has options

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

Shared variables

- Broadcast variables (sparkContext.broadcast())
  - 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 (sparkContext.accumulator())
  - 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

executor cannot read them
because other executors are also writing



Closing

Summary

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

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