SPARK Introduction

Sabeur Aridhi

TELECOM Nancy, University of Lorraine

Index

- Overview of Big Data platforms
  - Hadoop and Spark
- Spark architecture
- RDDs
  - Characteristics
  - Operations
  - Persistence and partitioning
- Data types and MLlib
- Programming on Spark
  - Code Examples
- Efficiency
- Other Resources

Index

- Overview of Big Data platforms
  - Hadoop and Spark
- Spark architecture
- RDDs
  - Characteristics
  - Operations
  - Persistence and partitioning
- Data types and MLlib
- Programming on Spark
  - Code Examples
- Efficiency
- Other Resources

Overview of Big Data platforms

Message Passing Interface (MPI)

- Low level framework
- Best effort message delivery policy
- User has to deal himself with:
  - Node failure
  - Load balancing
  - Distributed data storage and management
  - Heterogeneous hardware

Hadoop

- Automatically provides:
  - Fault tolerance
  - Load balancing
  - Distributed data storage and management
  - Heterogeneous hardware support
- Efficient for one-pass problems e.g.
  - Text processing
- Inefficient for iterative problems e.g.
  - Machine learning
  - Convex optimization
  - Graph processing
  - …
- Lack of primitives for data sharing
  - Intermediate results go to HDFS
  - No direct inter-node communication

Hadoop and Spark

Spark

- Cluster management (Fault tolerance, load balance and heterogeneous hw management)
- Rich API beyond Map and Reduce
- Direct node communication though shared variables
- High efficiency for iterative algorithms
- Extremely scalable

Ideal use case for Spark:

Most ML optimization problems are of the type:

$$w \leftarrow w - \alpha \cdot \sum_{i=1}^{n} g(w; \overbrace{x_i, y_i}^{\text{Data}})$$

Weight

Learning rate

Feature vector

Label

Spark scalability example:

- SVD problem
  O(min(m^{2}n,mn^2))
- 68 workers
- 8GB of RAM each

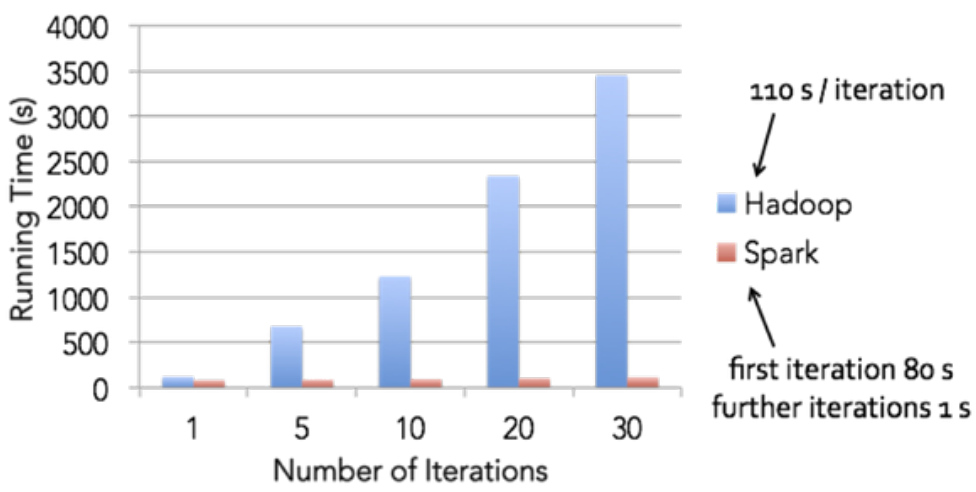
Matrix size	Number of nonzeros	Time per iteration (s)	Total time (s)
23,000,000 x 38,000	51,000,000	0.2	10
63,000,000 x 49,000	440,000,000	1	50
94,000,000 x 4,000	1,600,000,000	0.5	50

Hadoop and Spark

- **Hadoop vs Spark:** Logistic regression competition

Benchmark setup:

- Logistic regression
- 200 workers
- 16 GB of RAM each
- 100 GB of data

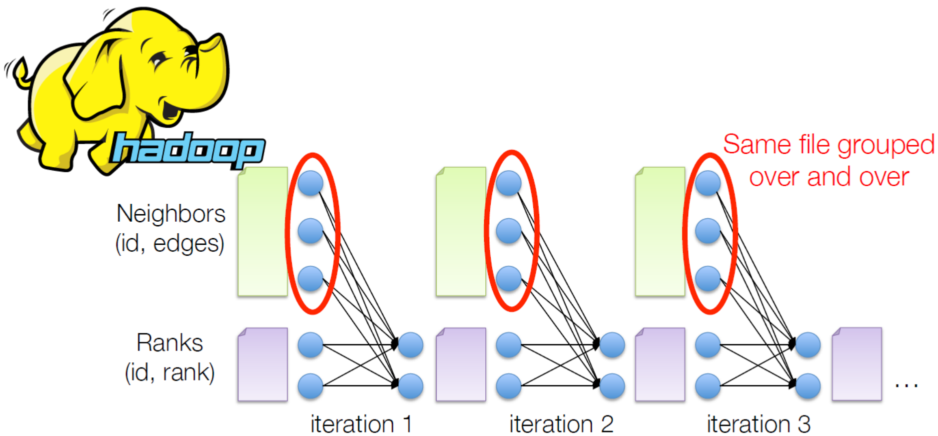


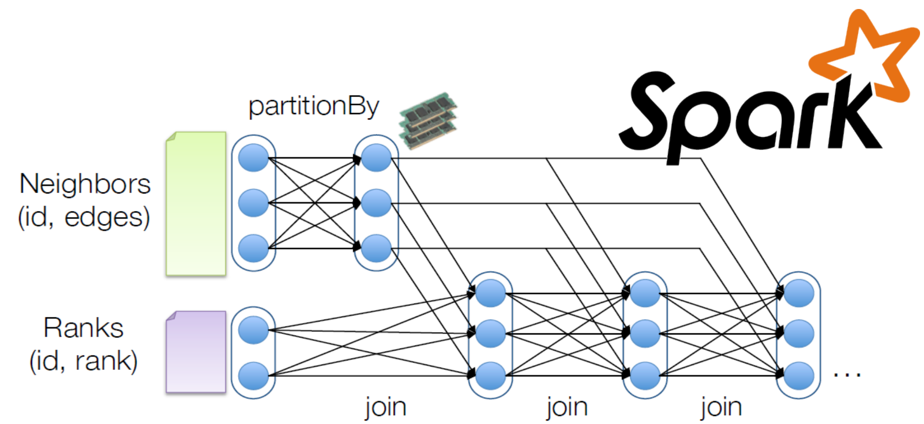
Why that gap?

- Iterative nature of the problem (using the same data in each iteration):
    - Spark iterates over the data stored in the clusters RAM
    - Hadoop reads/writes data to disk in each iteration
- Disk access 5 orders of magnitude slower than RAM!

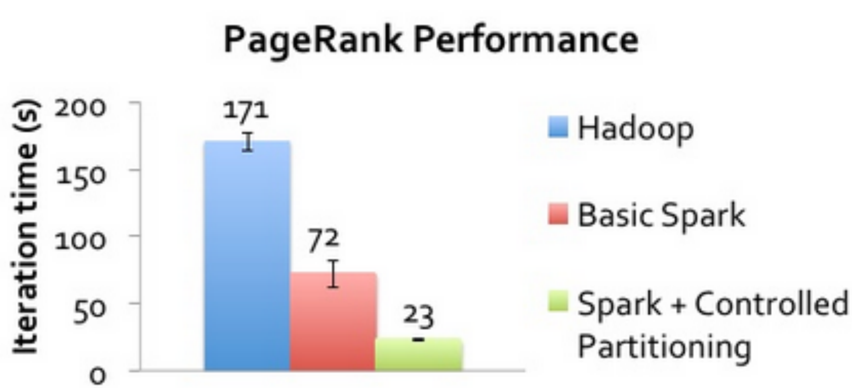
Hadoop and Spark

- **Hadoop vs Spark:** Page Rank competition





- Read/Write on disk in each iteration
- Loads data in RAM and iterates over it



Hadoop and Spark

- **Hadoop vs Spark:** [2014 Gray Sort Benchmark] (Daytona 100TB category)

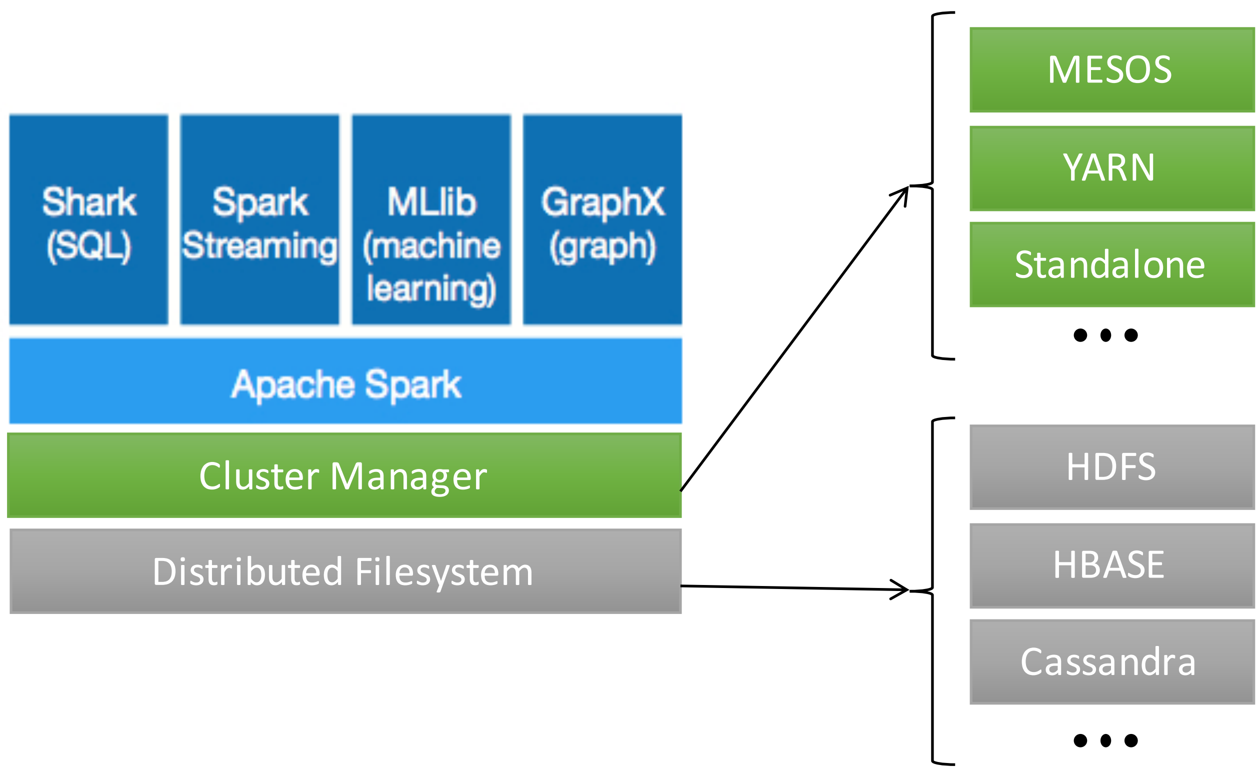
	Hadoop World Record	Spark 100 TB	Spark 1 PB
Data Size	102.5 TB	100 TB	1000 TB
Elapsed Time	72 mins	23 mins	234 mins
# Nodes	2100	206	190
# Cores	50400	6592	6080
# Reducers	10,000	29,000	250,000
Rate	1.42 TB/min	4.27 TB/min	4.27 TB/min
Rate/node	0.67 GB/min	20.7 GB/min	22.5 GB/min
Sort Benchmark Daytona Rules	Yes	Yes	No
Environment	dedicated data center	EC2 (i2.8xlarge)	EC2 (i2.8xlarge)

Index

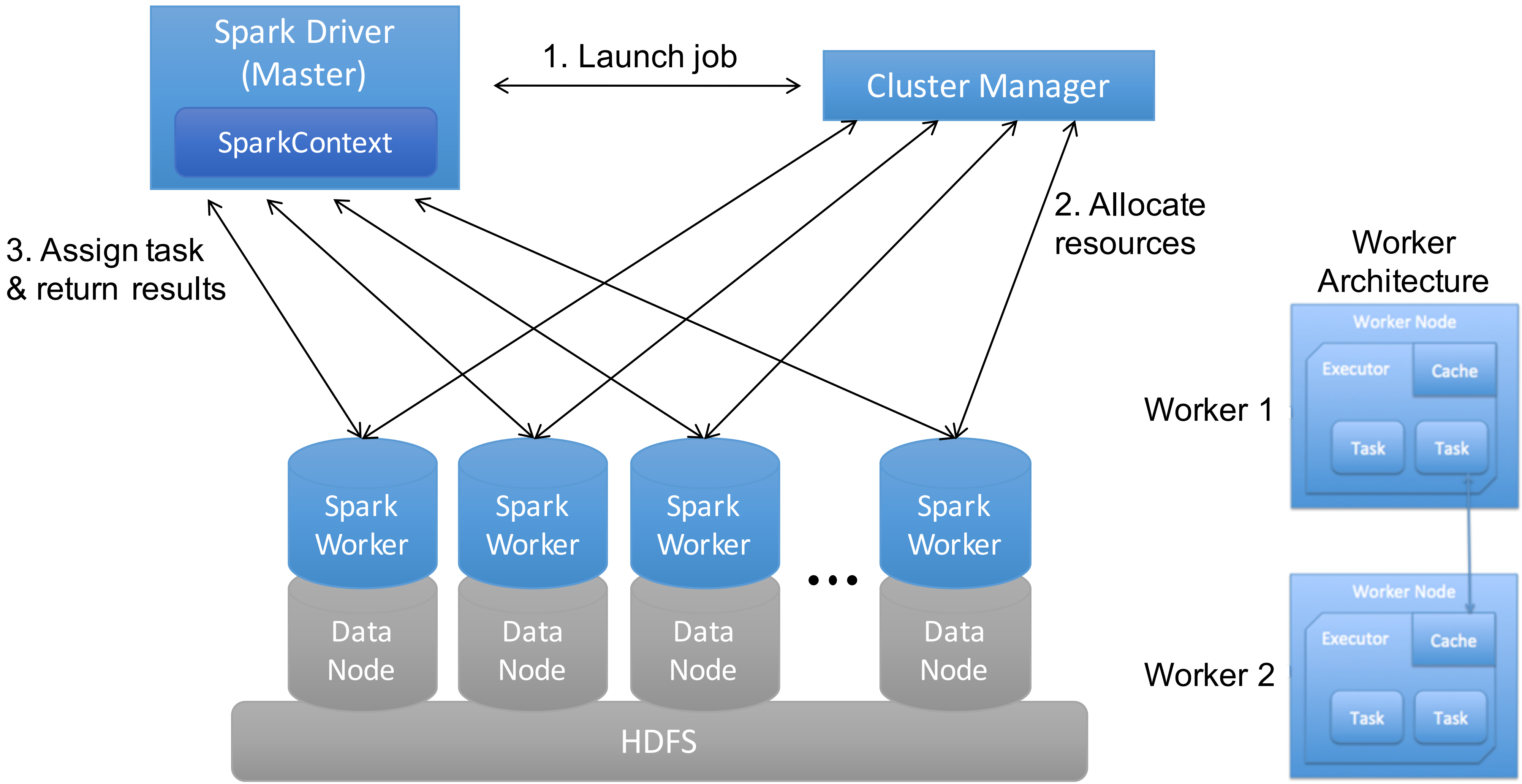
- Overview of Big Data platforms
  - Hadoop and Spark
- Spark architecture
- RDDs
  - Characteristics
  - Operations
  - Persistence and partitioning
- Data types and MLlib
- Programming on Spark
  - Code Examples
- Efficiency
- Other Resources

Spark architecture

v 1.6.1



Spark architecture

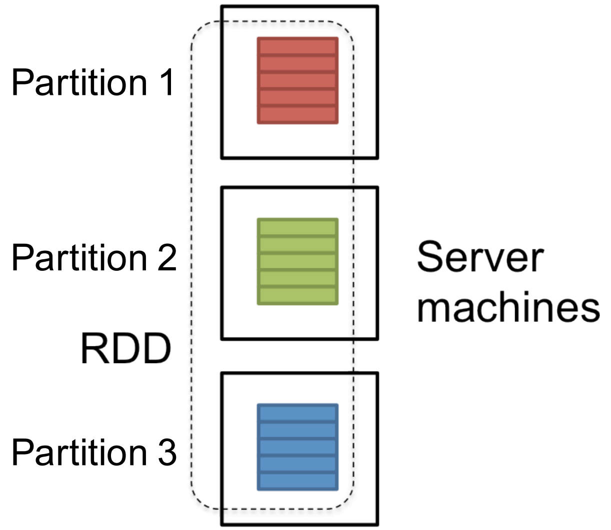


Index

- Overview of Big Data platforms
  - Hadoop and Spark
- Spark architecture
- RDDs
  - Characteristics
  - Operations
  - Persistence and partitioning
- Data types and MLlib
- Programming on Spark
  - Code Examples
- Efficiency
- Other Resources

Resilient Distributed Datasets

- **RDDs are the central concept in Spark!**



Distributed memory abstraction

- Extends programming language with distributed data structure (RDD).
- RDD spread in the cluster with user controlled partitioning & storage.
- As a rule of thumb set number of partitions to number of cores/workers in the cluster.

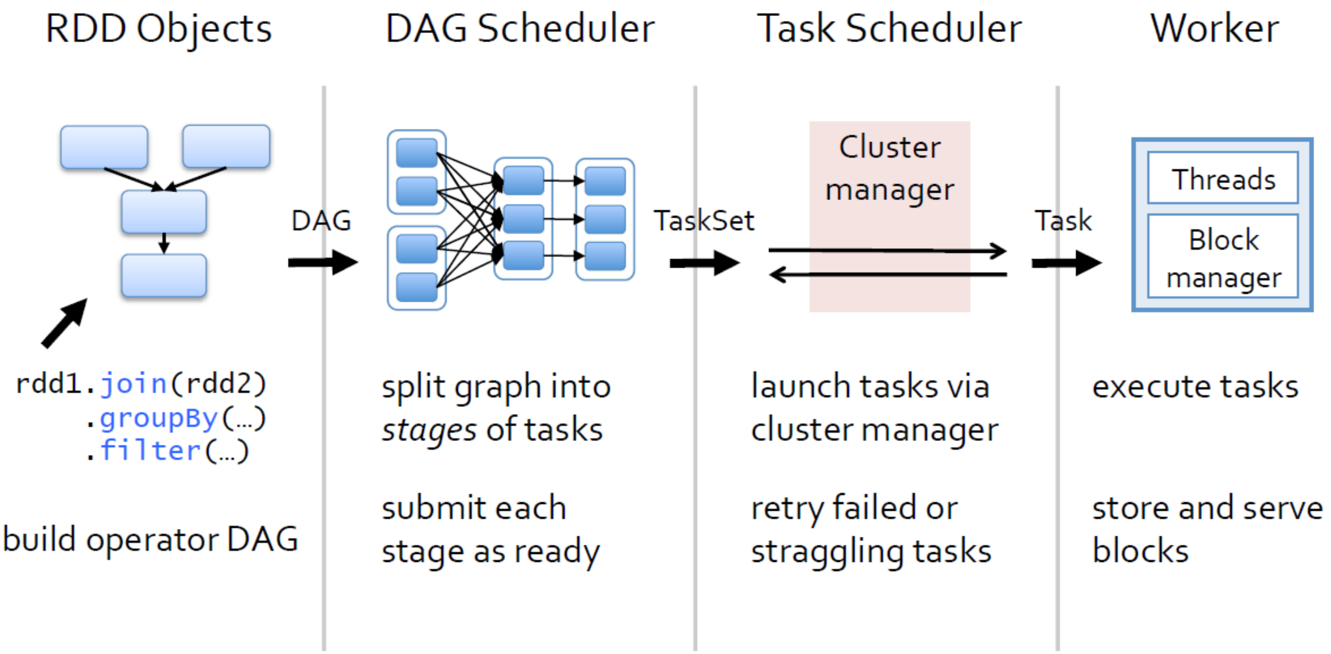
- Partitioning example pseudo code:

```
>>> RDD2 = RDD1.repartition(num_partitions)

>>> RDD3 = RDD2.map(f,preservesPartitioning=True)
```

Resilient Distributed Datasets

Lazy evaluation

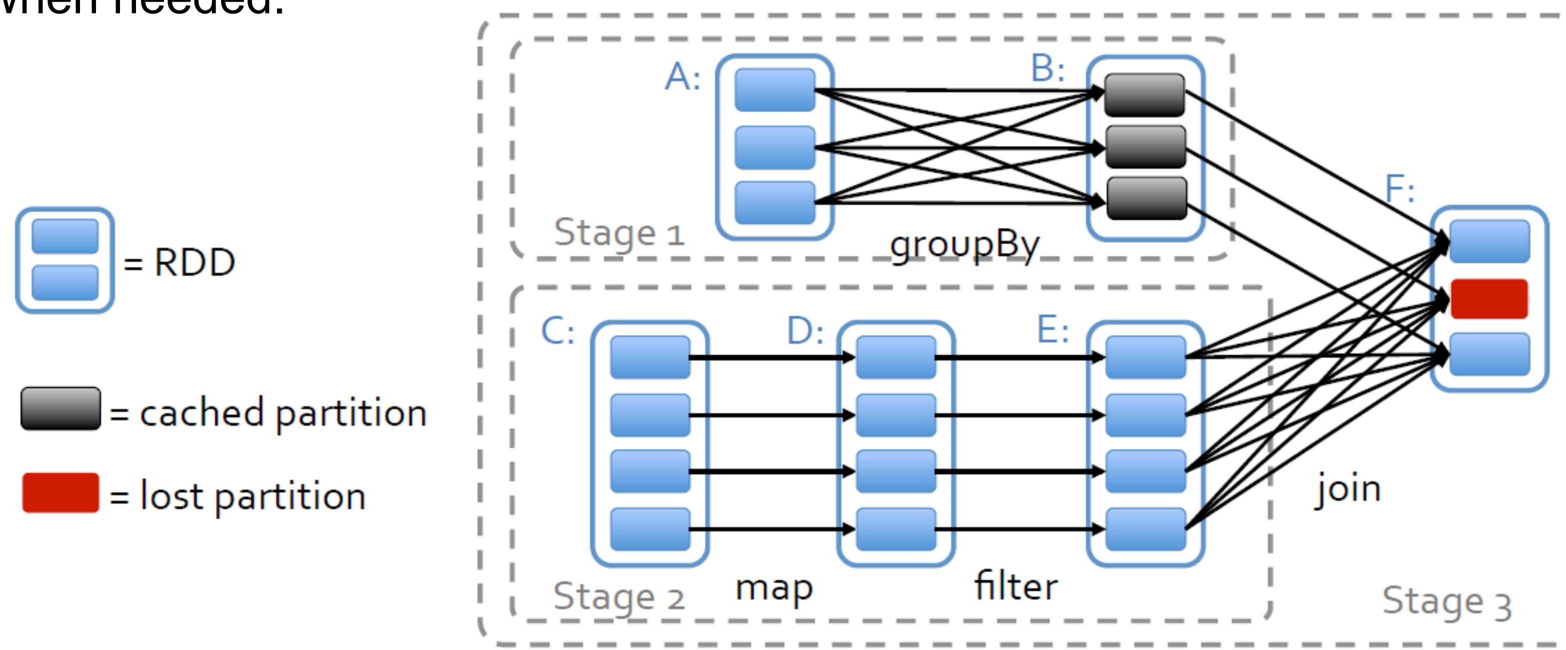
- Changes to the RDD stored as DAG of operations.
- Spark selects optimized order to execute DAG operations.
- Transformation needed to trigger evaluation/computation



Resilient Distributed Datasets

Fault tolerance

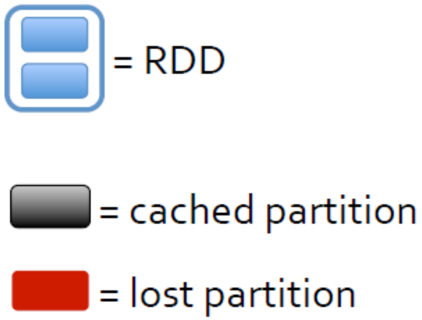
- RDDs automatically rebuild on failure.
- Spark keeps track of lineage of the RDDs so it can compute specific parts or whole RDDs again when needed.

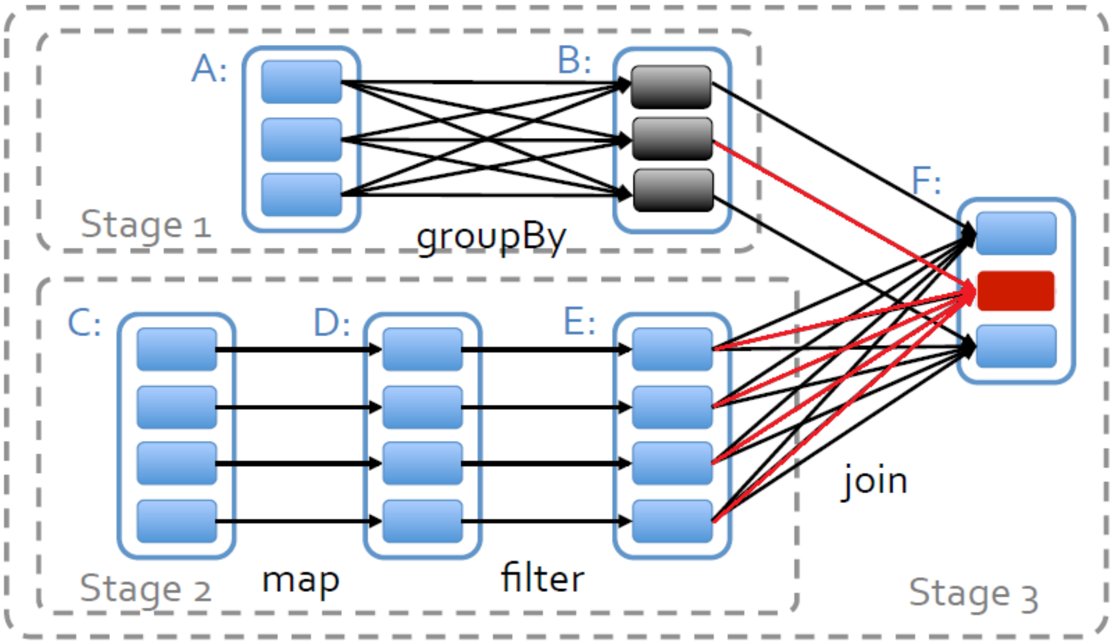


Resilient Distributed Datasets

Fault tolerance

- RDDs automatically rebuild on failure.
- Spark keeps track of lineage of the RDDs so it can compute specific parts or whole RDDs again when needed.

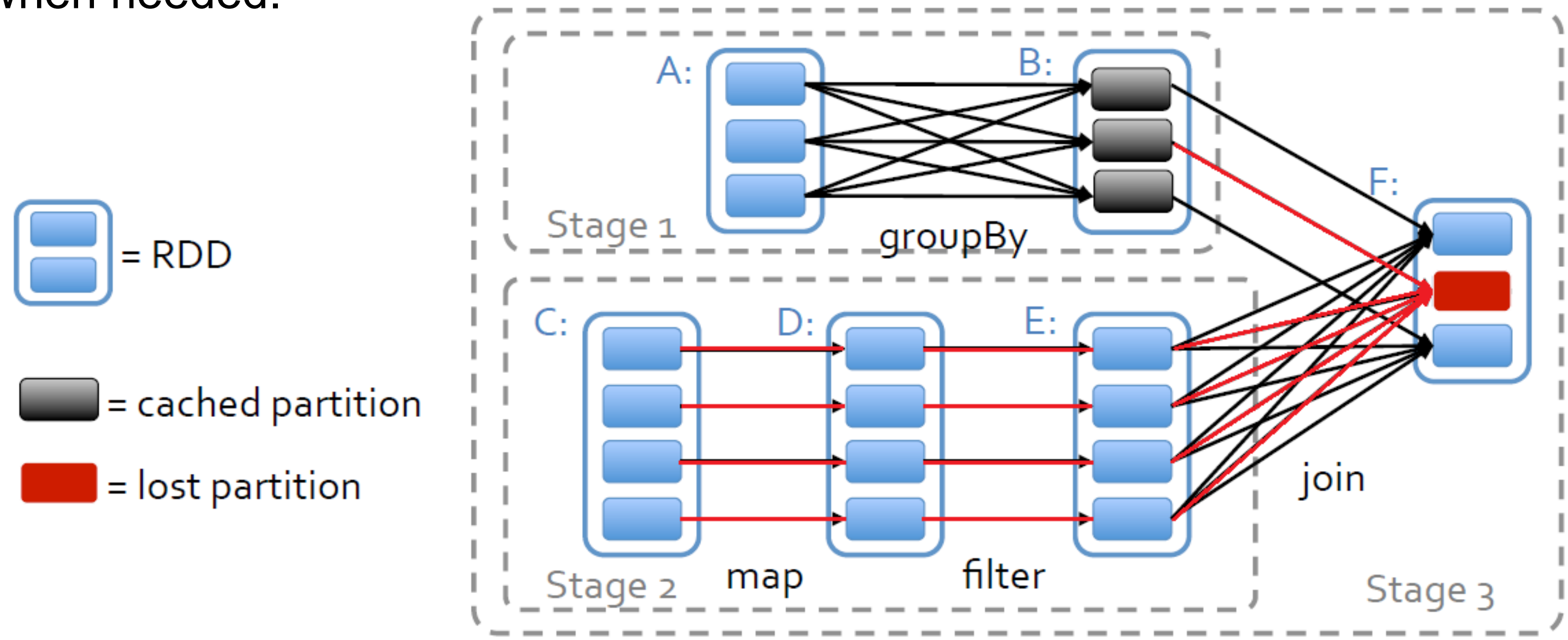




Resilient Distributed Datasets

Fault tolerance

- RDDs automatically rebuild on failure.
- Spark keeps track of lineage of the RDDs so it can compute specific parts or whole RDDs again when needed.



Creating RDDs

- First, in order to create an RDD the "SparkContext" has to be imported in the code. In Python the "SparkContext" is imported just like any other library:

```
>>> import numpy as np

>>> import SparkContext
```

- Now we can create RDDs from different sources:

  - **From local file:**

    ```
    >>> RDD1 = sc.textFile("README.md")
    ```

  - **From file in HDFS:**

    ```
    >>> RDD1 = sc.textFile("hdfs://file.txt")
    ```

  - **Parallelizing some local data:**

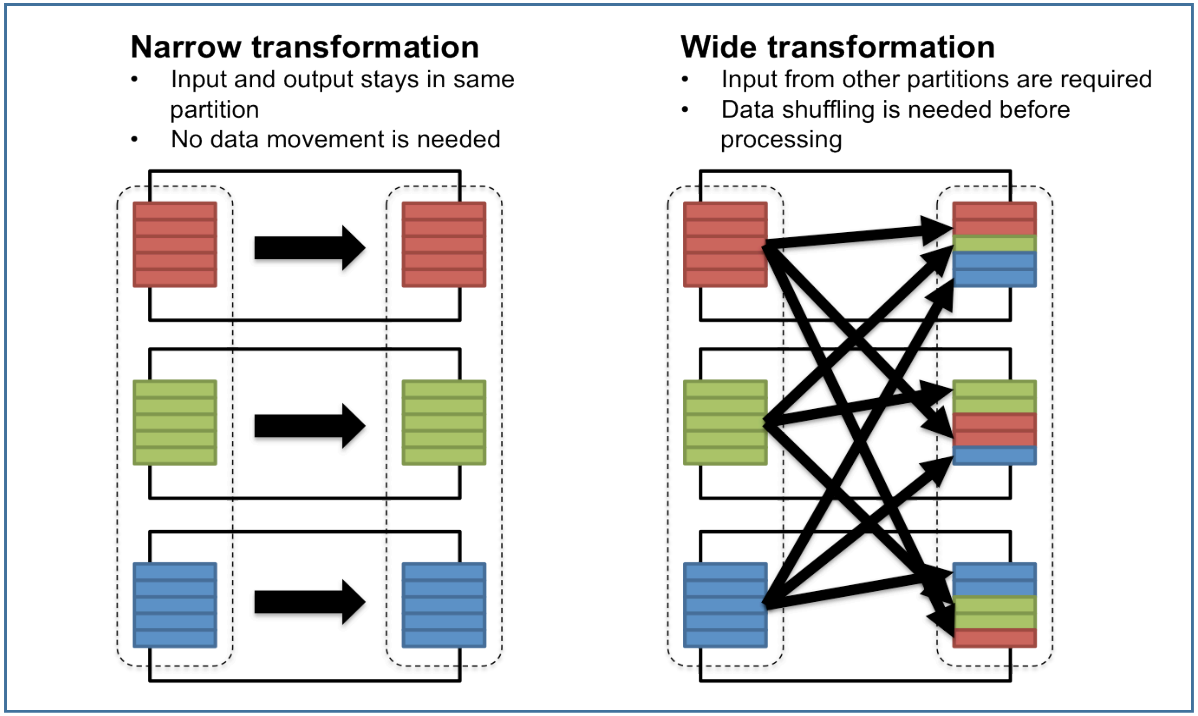
    ```
    >>> list = [1, a, f, b, 5, 6, d, 7, 1, Q, D, 4]

    >>> RDD2 = sc.parallelize(list)
    ```

  - **Creating the RDD in parallel:**

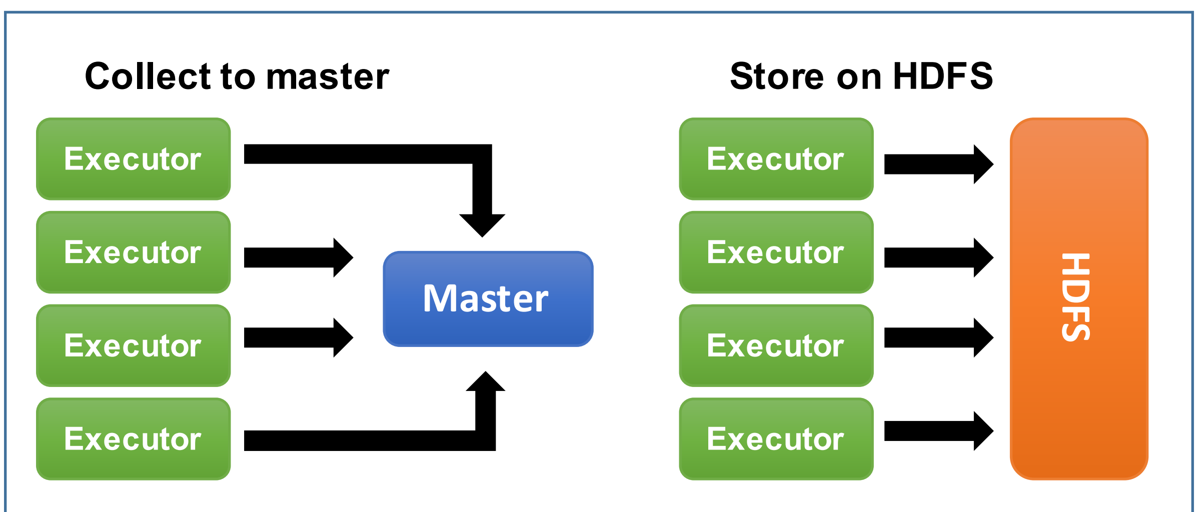
    ```
    >>> RDD2 = sc.parallelize(xrange(100,000))
    ```

RDD Operations

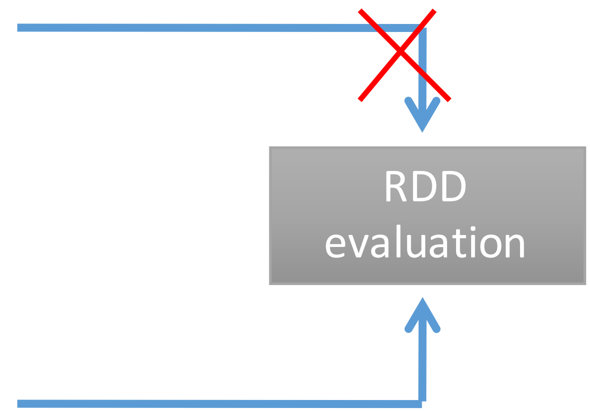
Transformations



Actions



Transformations:
Do not trigger RDD evaluation
Map RDD to another RDD



Actions:
Trigger RDD evaluation
Map RDD to value

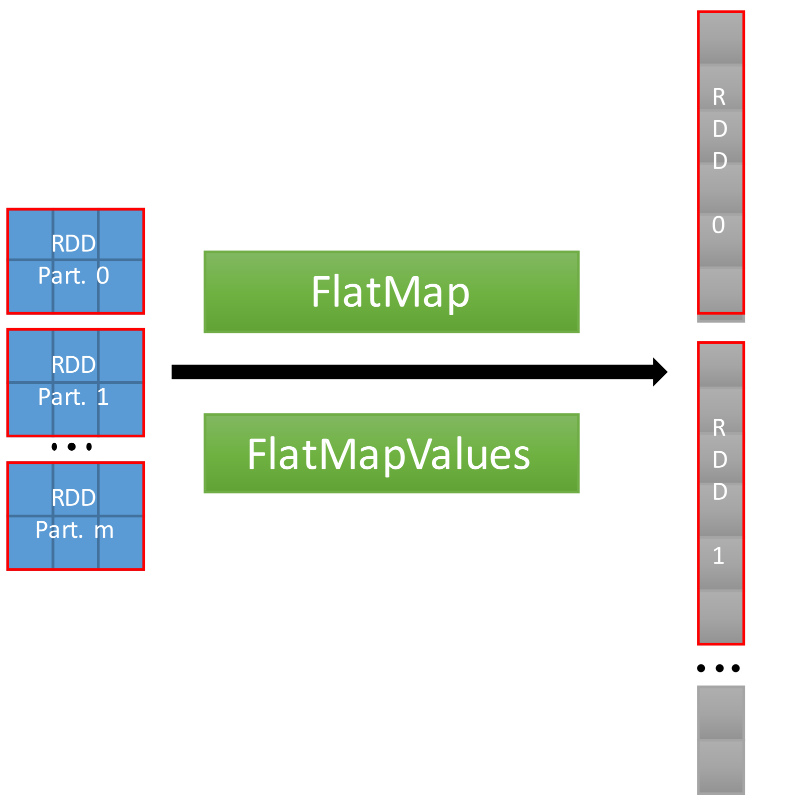
RDD Operations

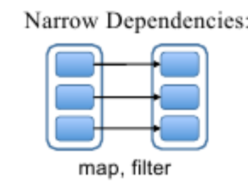
Transformations	
map(func)	sortByKey([ascending], [numTasks])
filter(func)	join(otherDataset, [numTasks])
flatMap(func)	cogroup(otherDataset, [numTasks])
mapPartitions(func)	cartesian(otherDataset)
mapPartitionsWithIndex(func)	pipe(command, [envVars])
sample(withReplacement, fraction, seed)	coalesce(numPartitions)
union(otherDataset)	repartition(numPartitions)
intersection(otherDataset)	repartitionAndSortWithinPartitions(partitioner)
distinct([numTasks]))	
groupByKey([numTasks])	
reduceByKey(func, [numTasks])	
aggregateByKey(zeroValue)(seqOp, combOp, [numTasks])	

RDD Operations

Actions
reduce(func)
collect()
count()
first()
take(n)
takeSample(withReplacement, num, [seed])
takeOrdered(n, [ordering])
saveAsTextFile(path)
saveAsSequenceFile(path) (Java and Scala)
saveAsObjectFile(path) (Java and Scala)
countByKey()
foreach(func)

Transformations





- **Code example:**

```
>>> rdd2 = rdd1.map(splitLines).flatMapValues(lambda x: x)
```

- **Functions:**

```
>>> # Input: string line of <id> <v1> <v2> ... <vM>
>>> # Output: (<id>,[<v1> <v2> ... <vM>])
>>> def splitLines (line):
>>>     l = np.array(line.split(' '),dtype=float)
>>>     return (int(l[0]),l[1:])
```

Transformations

CombineByKey

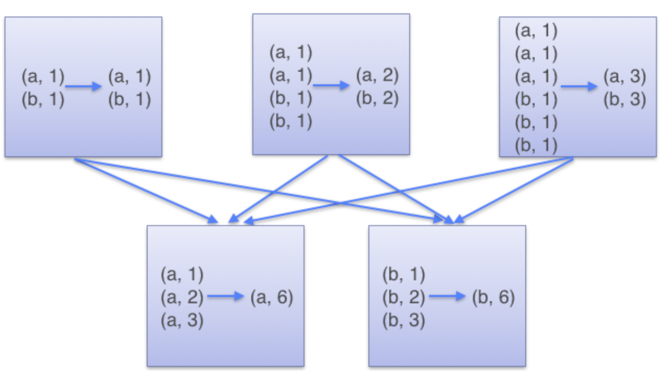
GroupByKey

AggregateByKey

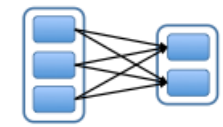
SubtractByKey

SortByKey

ReduceByKey



Wide Dependencies:



- **SubstractByKey:**

```
>>> rdd3 = rdd1.subtractByKey(rdd2,num_partitions)
```

- **ReduceByKey using lambdas:**

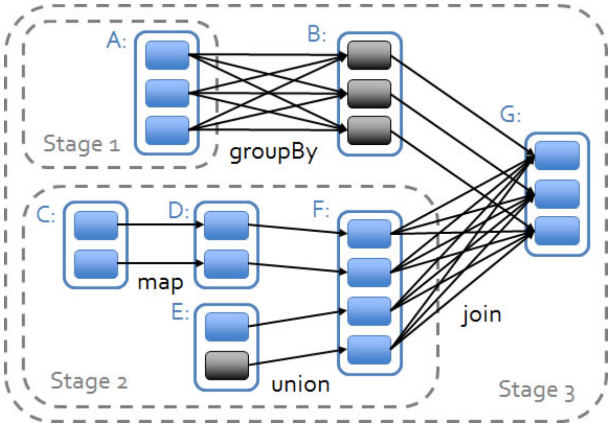
```
>>> rdd5 = rdd4.reduceByKey(lambda (a,b): a+b)
```

- **ReduceByKey using functions:**

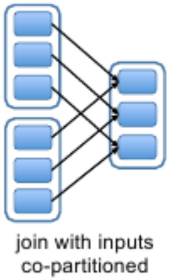
```
>>> rdd7 = rdd6.reduceByKey(redf)

# Input: ([intRow],[floatVal]),([intRow],[floatVal])
# Output: ([intRow],[floatVal])
def redf(tupl1, tupl2):
        t1 = []
        t2 = []
        t1.extend(tupl1[0])
        t1.extend(tupl2[0])
        t2.extend(tupl1[1])
        t2.extend(tupl2[1])
        return (t1,t2)
```

Transformations

- Intersection
- Union
- Join
- Cartesian
- Filter
- Repartition
- RepartitionAndSort



- **Join:**




```
>>> #Several operations leftJoin, rightJoin, fullJoin
>>> rdd1.join(rdd2)
```

- **Union:**

```
>>> rdd3 = sc.union(rdd1,rdd2)
```

Actions

- First
- Take
- Collect
- Count
- SaveAsTextFile
- Reduce

```
>>> rdd = sc.parallelize(xrange(100))
```

- **Count:**

```
>>> n_elems = rdd.count()
```

- **Collect:**

```
>>> data = rdd.collect()
```

- **SaveAsTextFile:**

```
>>> rdd.saveAsTextFile(outputFile)
```

RDDs storage

- RDDs can be stored in:
  - Disk
  - RAM
- Use primitive *RDD.persist(Storage.Level.XX)* with XX any of the options below

Storage Levels	Meaning
MEMORY_ONLY	Default level. If RDD not fit in RAM, some partitions will be cached and recomputed on the fly each time they're needed. Deserialized object.
MEMORY_AND_DISK	If RDD not fit in memory, store the partitions that don't fit on disk, and read them from there when they're needed. Deserialized object.
MEMORY_ONLY_SER	Like Memory_only but serialized objects. More space-efficient but more CPU intensive.
MEMORY_AND_DISK_SER	Like Memory_and_disk but serialized objects.
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.

RDDs communication patterns

- **Communication patterns for RDDs:**
  - **None:** All narrow transformations where data stays in the same node, e.g.:
    - Map
    - Union
    - Store to HDFS
    - …
  - **All to all:** All the wide transformations, executors send data to the rest of executors, e.g.:
    - All the “byKey” transformations
    - Join
    - Cartesian
    - …
  - **One to all:** Master broadcasts data to executors:
    - Broadcast
  - **All to one:** Executors return data to master, e.g.:
    - Reduce
    - Count
    - …

Index

- Overview of Big Data platforms
  - Hadoop and Spark
- Spark architecture
- RDDs
  - Characteristics
  - Operations
  - Persistence and partitioning
- **Data types and MLlib**
- Programming on Spark
  - Code Examples
- Efficiency
- Other Resources

Shared Variables

In addition to RDDs spark defines two types of shared variables:

- **Broadcast variables:** Read-only variable cached in each machine

```
>>> broadcastVar = sc.broadcast([1, 2, 3])
<pyspark.broadcast.Broadcast object at 0x102789f10>

>>> broadcastVar.value
[1, 2, 3]
```

- **Accumulators:** Variables that can only be added to in parallel

```
>>> accum = sc.accumulator(0)
Accumulator<id=0, value=0>

>>> sc.parallelize([1, 2, 3, 4]).foreach(lambda x: accum.add(x))
...
10/09/29 18:41:08 INFO SparkContext: Tasks finished in 0.317106 s

>>> accum.value
10
```

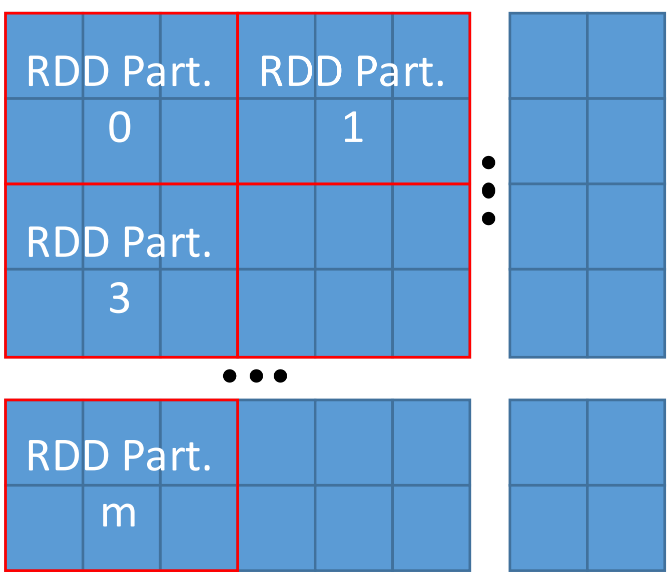
Supported data types

The following RDD data types are supported/provided by Spark:

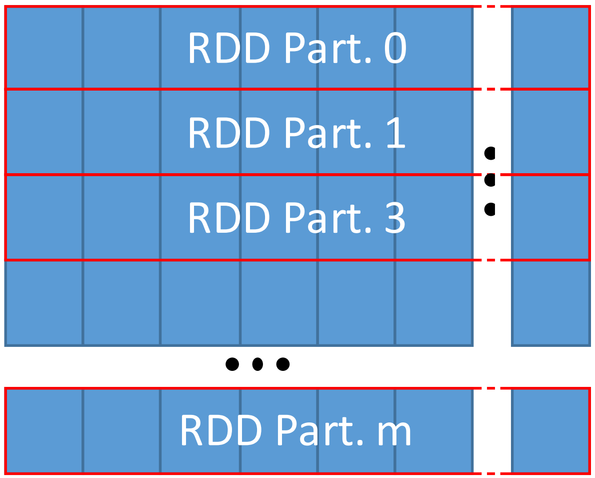
- **Regular vectors and matrixes**

$$A = \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \end{bmatrix} \qquad V = \begin{bmatrix} V_1 & V_2 & V_3 & V_4 \end{bmatrix}$$

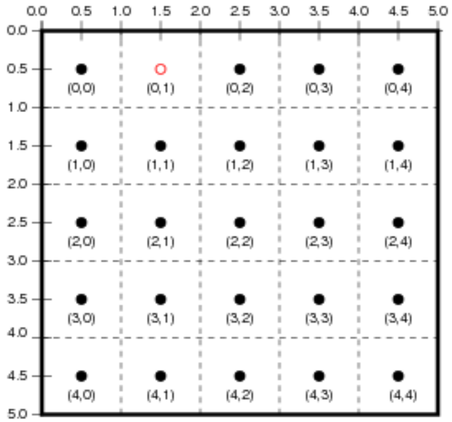
- **BlockMatrix**



- **RowMatrix / IndexedRowMatrix**



- **CoordinateMatrix**



Supported data types

- **RowMatrix:**

```python
from pyspark.mllib.linalg.distributed import RowMatrix
# Create an RDD of vectors.
rows = sc.parallelize([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]])
# Create a RowMatrix from an RDD of vectors.
mat = RowMatrix(rows)
```

- **IndexedRowMatrix:**

```python
from pyspark.mllib.linalg.distributed import IndexedRow, IndexedRowMatrix
# Create an RDD of indexed rows.
#   - This can be done explicitly with the IndexedRow class:
indexedRows = sc.parallelize([IndexedRow(0, [1, 2, 3]),
                              IndexedRow(1, [4, 5, 6]),
                              IndexedRow(2, [7, 8, 9]),
                              IndexedRow(3, [10, 11, 12])])
#   - or by using (long, vector) tuples:
indexedRows = sc.parallelize([(0, [1, 2, 3]), (1, [4, 5, 6]),
                              (2, [7, 8, 9]), (3, [10, 11, 12])])
# Create an IndexedRowMatrix from an RDD of IndexedRows.
mat = IndexedRowMatrix(indexedRows)
```

- **CoordinateMatrix:**

```python
from pyspark.mllib.linalg.distributed import CoordinateMatrix, MatrixEntry
# Create an RDD of coordinate entries.
#   - This can be done explicitly with the MatrixEntry class:
entries = sc.parallelize([MatrixEntry(0, 0, 1.2), MatrixEntry(1, 0, 2.1), MatrixEntry(6, 1, 3.7)])
#   - or using (long, long, float) tuples:
entries = sc.parallelize([(0, 0, 1.2), (1, 0, 2.1), (2, 1, 3.7)])
# Create an CoordinateMatrix from an RDD of MatrixEntries.
mat = CoordinateMatrix(entries)
```

- **BlockMatrix:**

```python
from pyspark.mllib.linalg import Matrices
from pyspark.mllib.linalg.distributed import BlockMatrix
# Create an RDD of sub-matrix blocks.
blocks = sc.parallelize([((0, 0), Matrices.dense(3, 2, [1, 2, 3, 4, 5, 6])),
                         ((1, 0), Matrices.dense(3, 2, [7, 8, 9, 10, 11, 12]))])
# Create a BlockMatrix from an RDD of sub-matrix blocks.
mat = BlockMatrix(blocks, 3, 2)
```

More info:

http://spark.apache.org/docs/latest/mllib-data-types.html

MLlib

MLlib is the Machine Learning library for Spark.
Provides implementations of high-level algorithms such as:

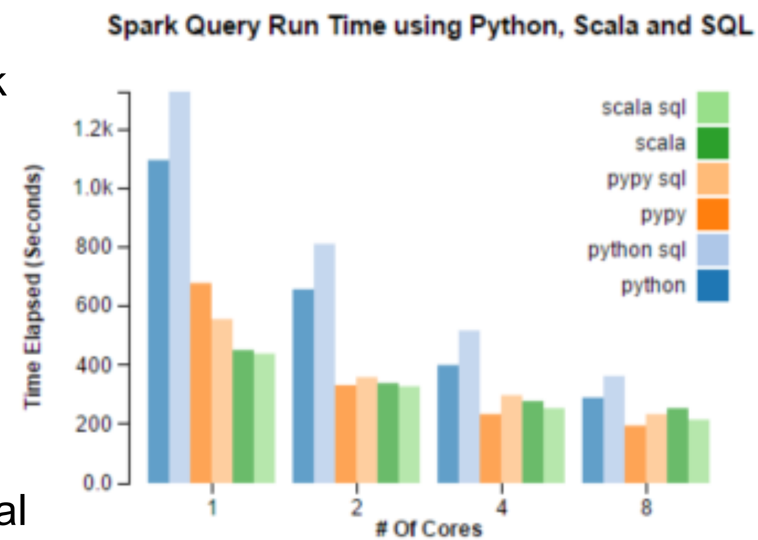
Classification	**Logistic Regression, Linear SVM, Naïve Bayes, Least Squares …**
Regression	**Generalized Linear Models (GLMs), Regression Trees …**
Clustering	**K-means**
Collaborative filter	**Alternating Least Squares (ALS), Non-negative Matrix Factorization (NMF) …**
Decomposition	**SVD, PCA**
Optimization	**Stochastic Gradient Descend, L-BFGS …**

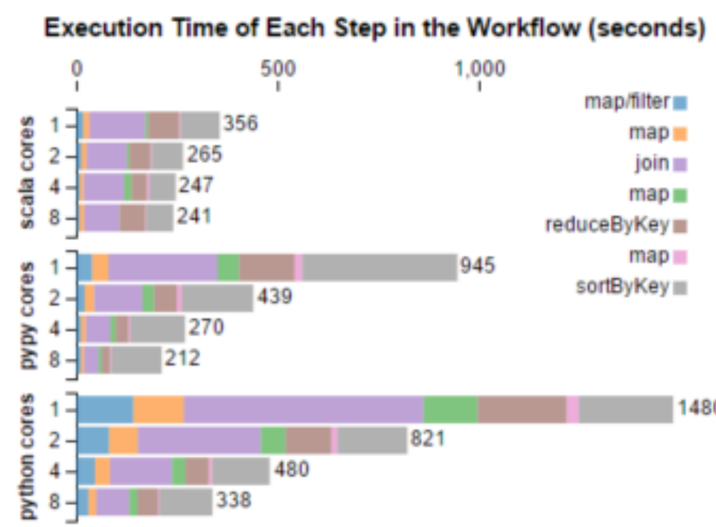
Index

- Overview of Big Data platforms
  - Hadoop and Spark
- Spark architecture
- RDDs
  - Characteristics
  - Operations
  - Persistence and partitioning
- Data types and MLlib
- Programming on Spark
  - Code Examples
- Efficiency
- Other Resources

Programming on Spark

- **Scala:**
  - Spark written in Scala => allows to modify Spark internally
  - All MLlib algorithms and data types
  - Lacks same Data science libs as Python
- **Python:**
  - Using pypy as efficient as Scala
  - Can execute R and other scripts
  - Spark-specific matrix data types still experimental
  - Some MLlib algorithms are not available
- **Java:**
  - Not user friendly coding style
  - No interactive Java shell like Python/Scala
- **R:**
  - R API available since Spark 1.4.0
  - In previous versions run R from Python
  - Not stable!!





Programming on Spark

Scala > Python > Java

Programming on Spark

- **A Spark program workflow:**

  1. Create input RDDs
  2. Transform the RDDs (filter, map, join, union…)
  3. Persist in RAM the RDDs that are used more than once
  4. Launch an action to start the parallel computation

  - **Wordcount example:**

```
text_file = sc.textFile("hdfs://...")
counts = text_file.flatMap(lambda line: line.split(" ")) \
             .map(lambda word: (word, 1)) \
             .reduceByKey(lambda a, b: a + b)
counts.saveAsTextFile("hdfs://...")
```

  - **Other examples: …**

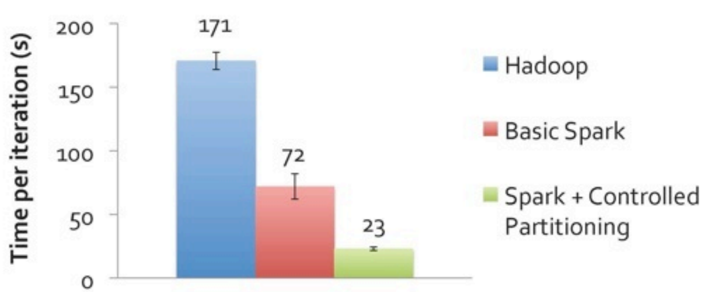
Index

- Overview of Big Data platforms
  - Hadoop and Spark
- Spark architecture
- RDDs
  - Characteristics
  - Operations
  - Persistence and partitioning
- Data types and MLlib
- Programming on Spark
  - Code Examples
- Efficiency
- Other Resources

Efficiency

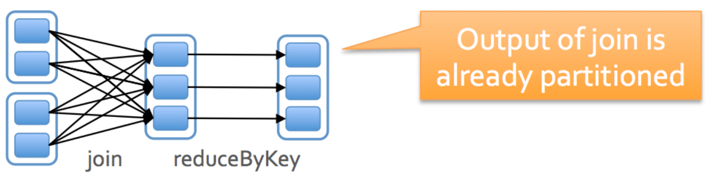
- **Use mostly narrow operations**
  - Keep data in the same node as long as possible
- **Keep RDD shuffled and partitioned**
  - Don't lose track of partitioning (use *mapValues / FlatMapValues* instead of *map / flatMap*)

PageRank Performance

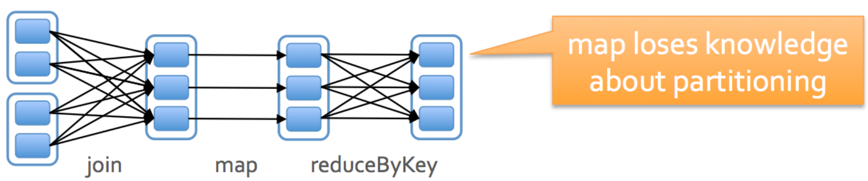


Up to a 70% speedup using smart partitioning !

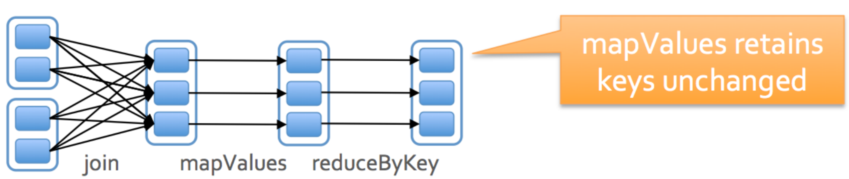
```
pages.join(visits).reduceByKey(...)
```




```
pages.join(visits).map(...).reduceByKey(...)
```

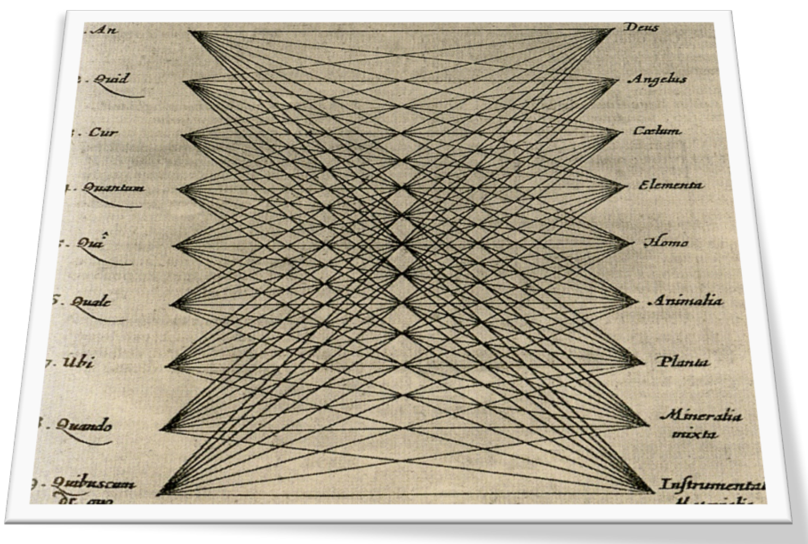


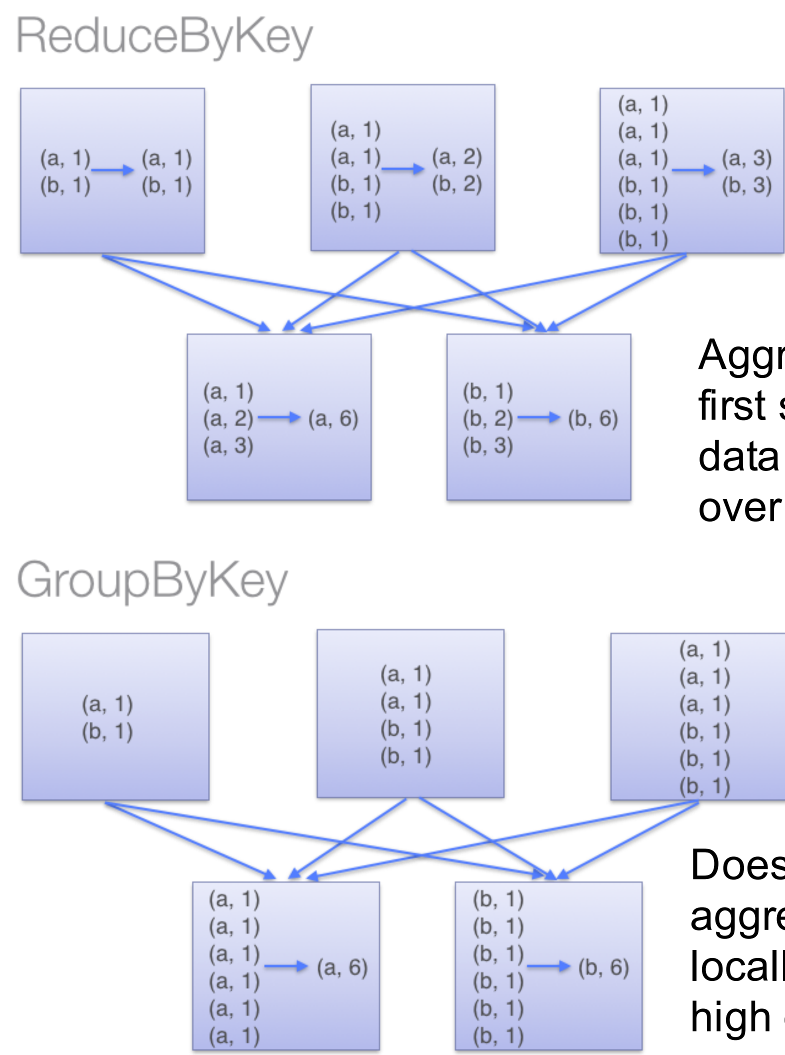

```
pages.join(visits).mapValues(...).reduceByKey(...)
```



Efficiency

- **Try to avoid the following operations:**
  - Cartesian product
  - GroupByKey
  - Collect to driver





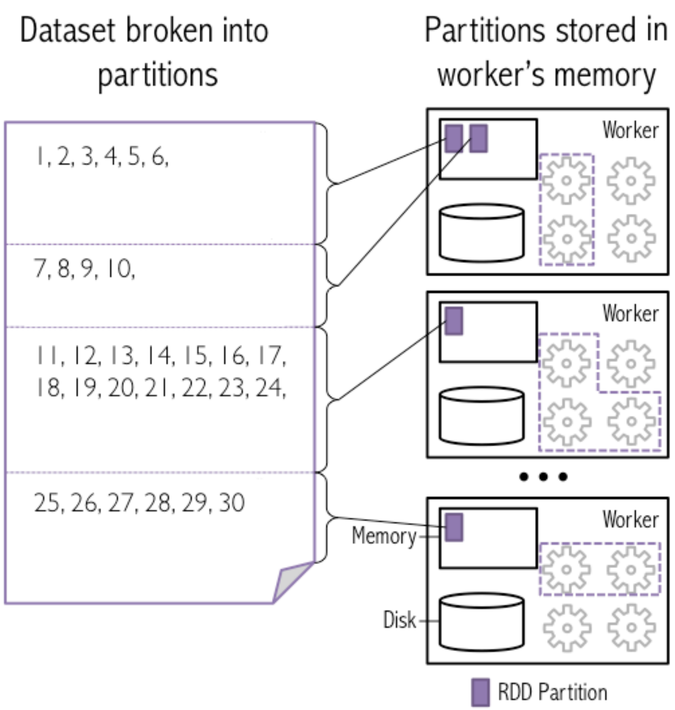
Aggregates first so less data sent over network

Does not aggregate locally! So high data movement

Efficiency

- **Tune:**
  - Number of partitions (equal to the number of cores usually)

- **Also consider tuning:**
  - Spark *conf* variables



Configuration	Description	Default Value
spark.executor.instances (--num-executors)	The number of executors	2
spark.executor.cores (--executor-cores)	Number of CPU cores used by each executor	1
spark.executor.memory (--executor-memory)	Java heap size of each executor	512m
spark.yarn.executor.memoryOverhead	The amount of off-heap memory (in megabytes) to be allocated per executor	executorMemory * 0.07, with minimum of 384

Efficiency

- **Use pypy:**

pypy

Speedup 20% - 3000% !

1. Install pypy*:
   - Download latest pypy binary for you architecture
   - Create softlink to pypy in e.g. /usr/local/bin/:

     $ ln -s /home/dru/Programs/pypy-5.0.1-linux64/bin/pypy pypy
2. Install numpy for pypy:
   - Download numpy for pypy:

     $ git clone https://bitbucket.org/pypy/numpy.git
   - Install:

     $ pypy setup.py install
3. Run the code
   - Regular Spark execution:

     $ ~/Descargas/spark-1.6.0-bin-hadoop2.4/bin/spark-submit spark_sort.py data/testdata100
   - Pypy Spark execution:

     $ PYSPARK_PYTHON=pypy ~/Descargas/spark-1.6.0-bin-hadoop2.4/bin/spark-submit spark_sort.py data/testdata100

Index

Other Resources

- **Spark presentation paper**
https://people.csail.mit.edu/matei/papers/2012/nsdi_spark.pdf

- **Spark Official Webpage:**
https://spark.apache.org/docs/1.3.1/programming-guide.html

- **MLlib presentation paper:**
http://arxiv.org/pdf/1505.06807v1.pdf

- **Databricks Spark guide:**
https://databricks.com/spark/about

- **Databricks Spark Knowledge Base Book:**
https://www.gitbook.com/book/databricks/databricks-spark-knowledge-base/details

- **Reza Zadeh from Stanford University has some realy nice slides on Spark if you want to learn more:**
http://stanford.edu/~rezab/slides/sparksummit2015/

Questions

Questions ?

sabeur.aridhi@telecomnancy.eu