NTNU

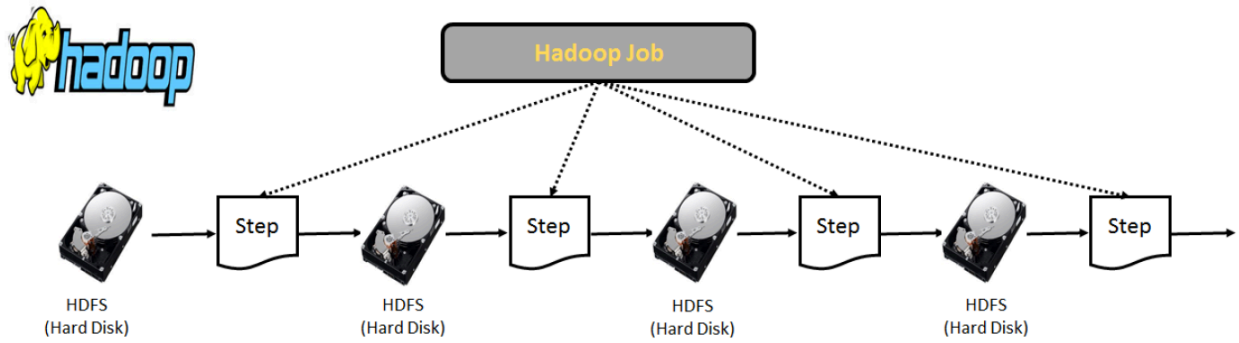
Norwegian University of Science and Technology

Introduction and Concepts
Programming in Spark using RDD API
RDD Representation and Other Spark Features

NTNU

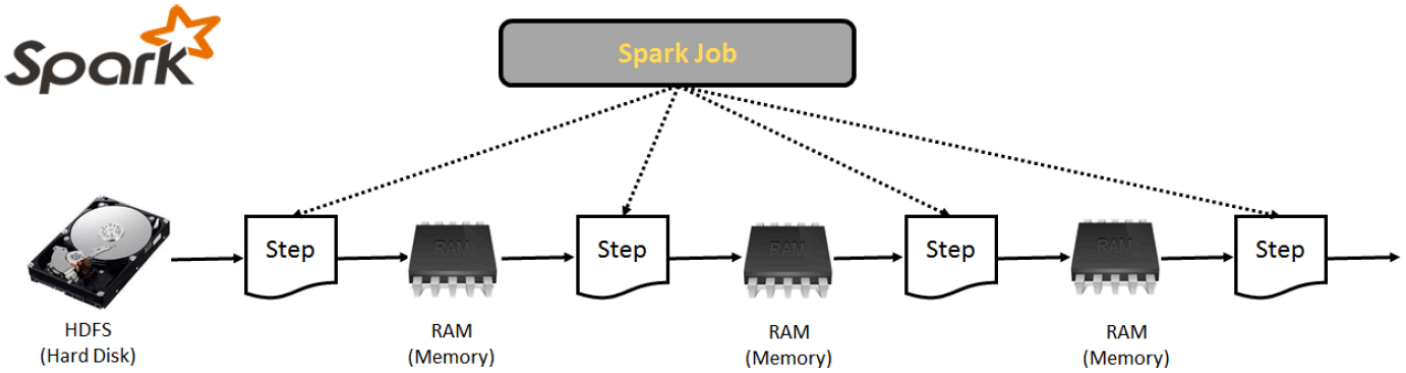
Background and Motivation

- Observation: Hadoop MapReduce enabled Big Data analysis by isolating the issues on data distribution, scheduling and fault tolerance.
- Limitations of MapReduce:
  - Output of each map-reduce stage is materialized on HDFS.
  - Inefficient for interactive queries and iterative algorithms.



Apache Spark...

- offers the ability to run computations in memory, which enables developing applications where data on memory is used by parallel operations,
- provides efficiency for iterative computing jobs and interactive use.



Apache Spark...

- can run in Hadoop clusters and access Hadoop data sources,

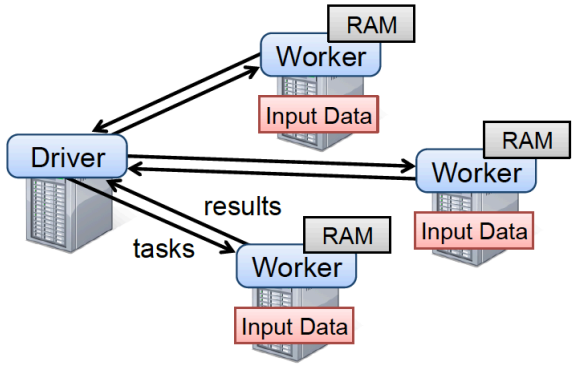
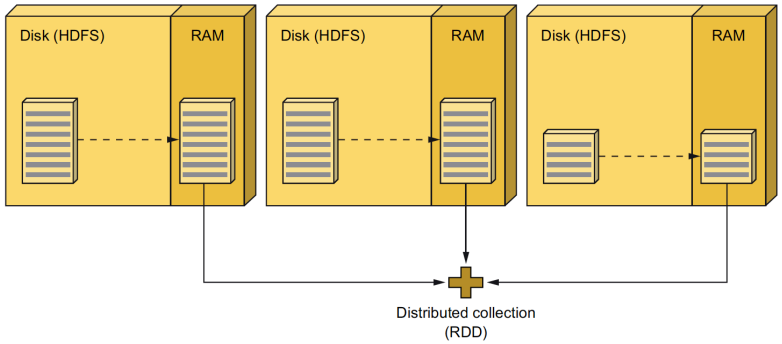


Figure 2: Spark runtime. The user's driver program launches multiple workers, which read data blocks from a distributed file system and can persist computed RDD partitions in memory.

- extends MapReduce model to support more types of computations (filter, join, count, etc...) while retaining scalability and fault tolerance,
- offers APIs in Scala, Python, Java, SQL,
- interactive use in Python and Scala shells.

Programming model: RDD

- RDD: Resilient Distributed Dataset
- RDD is a fault-tolerant, read-only, partitioned collection of records that are distributed on multiple nodes and can be manipulated using a rich set of operators.



- Each RDD is split into multiple partitions, which may be computed on different nodes of the cluster.
- RDDs can contain any type of Python, Java, or Scala objects, including user-defined classes.
- In Spark, all work is expressed as either creating new RDDs, transforming existing RDDs, or performing actions on RDDs to compute a result.

Programming model: RDD

- RDD can be created by loading data from file or parallelizing a collection of objects in driver program.

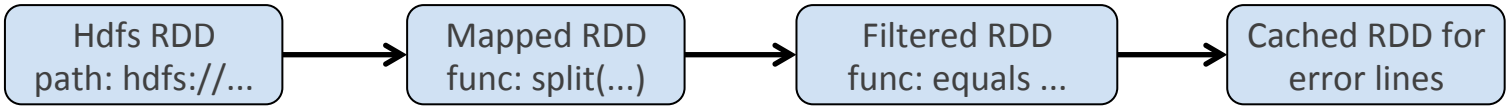
- RDD Transformation: Construct new RDD from previous one using a transformation operation (Example: filter, map, join...)
- RDD Action: Compute a result and store it in a file or return it to the driver program (Example: reduce, count, saveAsTextFile...)

```python
# Create-transform-action example in Python
lines = sc.textFile("hdfs://...")
errorLines = lines.filter(lambda line: "error" in line)
c = errorLines.count()  # returns the number of lines containing "error" in file
```

```scala
// Create-transform-action example in Scala
val lines = sc.textFile("hdfs://...")
val errorLines = lines.filter(_.contains("error"))
val c = errorLines.count()  // returns the number of lines containing "error" in file
```

RDD Characteristics

1. Partitioned collections of objects: Objects can reside in different nodes.
2. Immutable: RDDs can not be changed, but transformed into new RDDs.
3. Resilient: Lost partitions can automatically be reconstructed from previous RDDs (using RDD *lineage graphs* to remember each transformation).



4. Lazy evaluation: RDDs are not computed until they are used in an action.
5. Caching: Users can indicate which RDDs they will reuse. Otherwise, same transformations are applied to obtain the same RDD.

lines is an RDD. Partitions can be distributed on different nodes

Immutable: lines RDD is not changed, but transformed into a new RDD

Resilient: If a partition in errLines is lost, it can be reconstructed by reading the partition from file again.

Lazy: Computation does not start until there is an action (such as, *count*) on the RDD

```
lines = sc.textFile("hdfs://...")
sptLines = lines.map(lambda line: line.split('\t'))
errLines = sptLines.filter(lambda l: l[1] == "error")
errLines.cache()
print errLines.count()
print errLines.filter(lambda l: l[2] == "foo").count()
print errLines.filter(lambda l: l[2] == "bar").count()
```

An RDD can be cached in order to avoid re-computing it in every reference afterwards

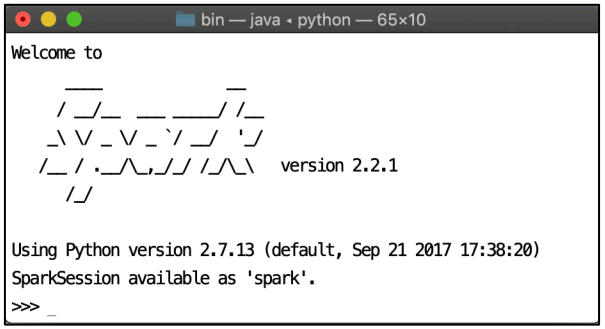
Development Environment

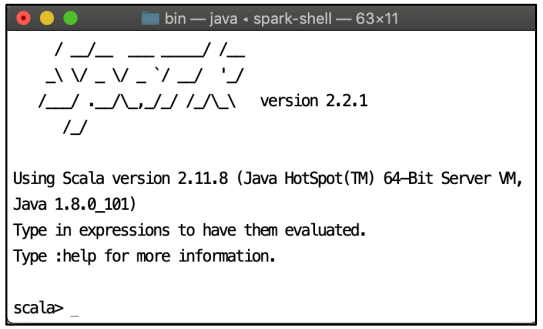
- Basic Installation on PC
  - Java (JDK 1.8)
  - Python (needed for development in Python)
  - Spark (http://spark.apache.org/downloads.html)
    - For Windows, it also needs winutils.exe
  - Set environment variables and paths

```
Mac:spark-2.2.1-bin-hadoop2.7 $ cd bin
Mac:bin $ ls
beeline                 pyspark2.cmd            spark-shell2.cmd
beeline.cmd             run-example             spark-sql
find-spark-home         run-example.cmd         spark-submit
find-spark-home.cmd     spark-class             spark-submit.cmd
load-spark-env.cmd      spark-class.cmd         spark-submit2.cmd
load-spark-env.sh       spark-class2.cmd        sparkR
pyspark                 spark-shell             sparkR.cmd
pyspark.cmd             spark-shell.cmd         sparkR2.cmd
```

Spark Shell for Python

Spark Shell for Scala

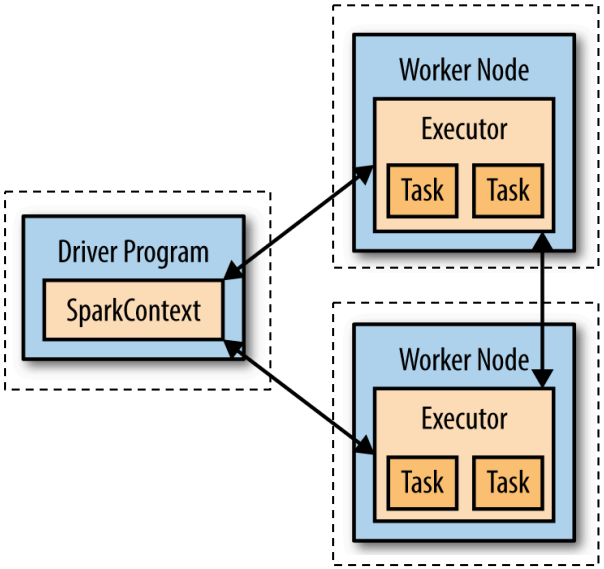




Core Spark Concepts

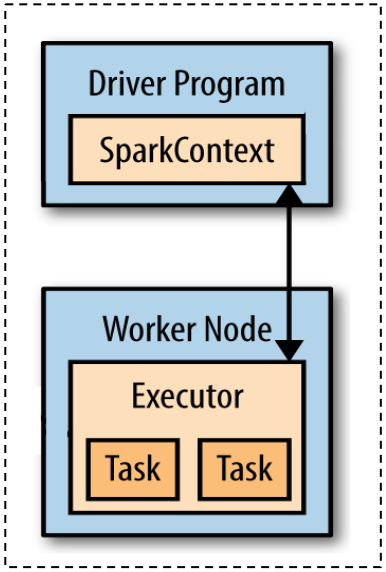
Distributed Mode

(usually configured to run on multiple machines)



Local Mode

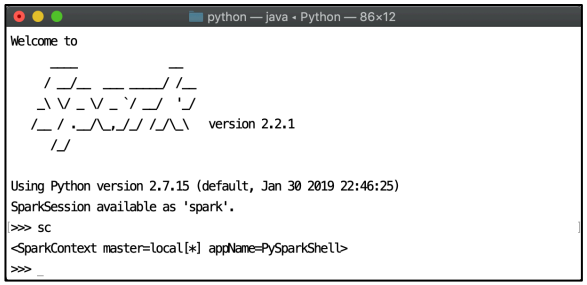
(Single JVM on a single machine, but there may be multiple Worker Threads)

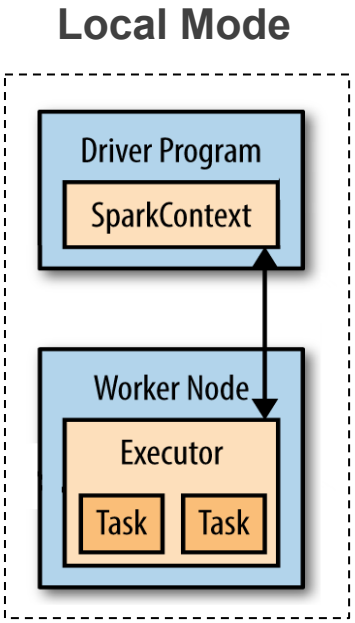


We will use Local Mode for development on a PC

Core Spark Concepts

- Driver program: Launches the parallel operations on Spark
  - Spark Shell is a driver program
- Spark Context: Used by driver programs to access Spark.
  - Shell automatically creates a Spark Context.





- Standalone Application: Application with a main function which acts as a Driver Program (without using Spark Shell).
  - The application must initialize its Spark Context.
  - The application must be built and submitted via *spark-submit* (jar file for Java and Scala, py file for Python).
  - Except for that, the API is the same with Spark Shell.

Spark Shell on your PC is sufficient for the examples in the following slides.

NTNU

Norwegian University of Science and Technology

Programming with RDD API

Basic RDD Operations

Recap

- RDD Creation: RDD can be created by loading data from file or parallelizing a collection of objects.

- RDD Transformation: Construct new RDD from previous one using a transformation operation.

- RDD Action: Compute a result to store in a file or to return to the driver program

All RDD transformations and actions listed in the following slides are in curriculum. Some of the most commonly used operations will be explained in more detail.

Basic Python for Spark

- Examples in the following slides will be mostly in Python. Try in pyspark.

```
>>> linesRDD = sc.textFile('/example/numbers123.txt')
>>> print linesRDD
/example/numbers123.txt MapPartitionsRDD[120] at textFile at NativeMethodAccessorImpl.java:0
>>> print linesRDD.collect()
[u'1', u'2', u'3']
```

Creating an RDD

Assume file has 3 lines with numbers from 1 to 3.

```
>>> numsRDD = linesRDD.map(int)
>>> print numsRDD.collect()
[1, 2, 3]
```

Transforming to a new RDD with Integer elements using int() function in Python.

```
>>> sqrRDD = numsRDD.map(lambda x: x * x)
>>> print sqrRDD.collect()
[1, 4, 9]
```

Defining lambda function with one argument in Python (anonymous function).

```
>>> def squared(s):
...   return s * s
>>> sqrRDD = numsRDD.map(squared)
```

Defining a named function in Python and using it to transform the values in RDD.

```
>>> import math
>>> sqrtRDD = numsRDD.map(math.sqrt)
```

Using existing functions in Python (remember to import the library)

```
>>> numsRDD.map(lambda x: x * x).collect()
```

Possible to write multiple operations at once

Programming Languages for Spark

- Interactive shell is available for Python and Scala

Python | Scala | Java

```python
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://...")
```

Python | Scala | Java

```scala
val textFile = sc.textFile("hdfs://...")
val counts = textFile.flatMap(line => line.split(" "))
                 .map(word => (word, 1))
                 .reduceByKey(_ + _)
counts.saveAsTextFile("hdfs://...")
```

Python | Scala | Java

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

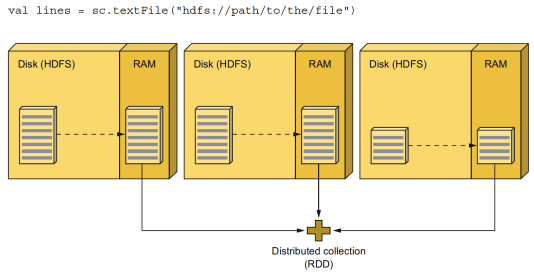
RDD Creation

Two ways to create RDDs:

1. Loading a dataset in an external storage system
   - Each line in the file is handled as an element in RDD
   - Consider availability of file if running in distributed setting

```
lines = sc.textFile("hdfs://...")

lines = sc.textFile("/path/to/file.txt")
```



2. Parallelizing an existing collection of objects in the driver program
   - Once created, the distributed dataset can be operated on in parallel.
   - **Usually for prototyping and testing !!!**

```
lines = sc.parallelize(["tea", "coffee", "caramel macchiato"])

lines = sc.parallelize([1,2,3,4])
```

Basic Transformations

- map: Applies a function on each element in RDD.

```
nums = sc.parallelize([1,2,3,4])
incrNums = nums.map(lambda x: x+1)
# incrNums contains [2, 3, 4, 5]
```

map can be used to create an RDD with different types of elements

```
texts = sc.parallelize(["tea", "coffee"])
textsAndLens = texts.map(lambda x: (x, len(x)))
# textsAndLens contains [('tea', 3), ('coffee', 6)]
lengths = textsAndLens.map(lambda x: x[1])
# lengths contains [3, 6]
```

string → tuple → number

- filter: Selects the elements that satisfy the criteria in a function.

```
nums = sc.parallelize([1,2,3,4])
evenNums = nums.filter(lambda x: x%2 == 0)
# evenNums contains [2, 4]
```

```
lines = sc.textFile("/example-data/samplecoffee.txt")
twoWordItems = lines.filter(lambda line: len(line.split(" ")) == 2)
# twoWordItems contains [u'caramel macchiato', u'hot chocolate']
```

the file contains 4 lines:

```
tea
coffee
caramel macchiato
hot chocolate
```

Basic Transformations

- flatMap: It can produce multiple output elements for each input element. Returns an RDD with elements from iterators.

Often used to extract words in texts

```
lines = sc.textFile("/example-data/samplecoffee.txt")
mapped = lines.map(lambda line: str(line).split(" "))
# mapped contains [['tea'], ['coffee'], ['caramel', 'macchiato'], ['hot', 'chocolate']]
flatmapped = lines.flatMap(lambda line: str(line).split(" "))
# flatmapped contains ['tea', 'coffee', 'caramel', 'macchiato', 'hot', 'chocolate']
```

An element (of type list) for each input element

Multiple output elements for each input element

functions for map vs. flatMap...

```
nums = sc.parallelize([1,2,3])
incr = nums.map(lambda x: x + 1)
incr.collect()
# This prints [2, 3, 4] on Spark shell
```

```
nums = sc.parallelize([1,2,3])
flatincr = nums.flatMap(lambda x: x + 1)
```

```
flatincr.collect()
# This throws an Exception in Spark!
# TypeError: 'int' object is not iterable
```

```
# Remember: LAZY EVALUATION
```

What if the function used in flatMap does not return an iterable?

The return type of input function is not iterable

Basic Transformations

Table 3-2. Basic RDD transformations on an RDD containing {1, 2, 3, 3}

Function name	Purpose	Example	Result
`map()`	Apply a function to each element in the RDD and return an RDD of the result.	`rdd.map(x => x + 1)`	`{2, 3, 4, 4}`
`flatMap()`	Apply a function to each element in the RDD and return an RDD of the contents of the iterators returned. Often used to extract words.	`rdd.flatMap(x => x.to(3))`	`{1, 2, 3, 2, 3, 3, 3}`
`filter()`	Return an RDD consisting of only elements that pass the condition passed to `filter()`.	`rdd.filter(x => x != 1)`	`{2, 3, 3}`
`distinct()`	Remove duplicates.	`rdd.distinct()`	`{1, 2, 3}`
`sample(withReplacement, fraction, [seed])`	Sample an RDD, with or without replacement.	`rdd.sample(false, 0.5)`	Nondeterministic

```
nums = sc.parallelize([2,1,1,2,3,1])
sortedNums = nums.sortBy(lambda x: x, False)
# sortedNums contains [3, 2, 2, 1, 1, 1]
sortedNums = nums.sortBy(lambda x: x, True)
# sortedNums contains [1, 1, 1, 2, 2, 3]
```

sortBy

Sorting elements in RDD in descending/ ascending order

Basic Two-RDD Transformations

Table 3-3. Two-RDD transformations on RDDs containing {1, 2, 3} and {3, 4, 5}

Function name	Purpose	Example	Result
`union()`	Produce an RDD containing elements from both RDDs.	`rdd.union(other)`	`{1, 2, 3, 3, 4, 5}`
`intersection()`	RDD containing only elements found in both RDDs.	`rdd.intersection(other)`	`{3}`
`subtract()`	Remove the contents of one RDD (e.g., remove training data).	`rdd.subtract(other)`	`{1, 2}`
`cartesian()`	Cartesian product with the other RDD.	`rdd.cartesian(other)`	`{(1, 3), (1, 4), … (3,5)}`

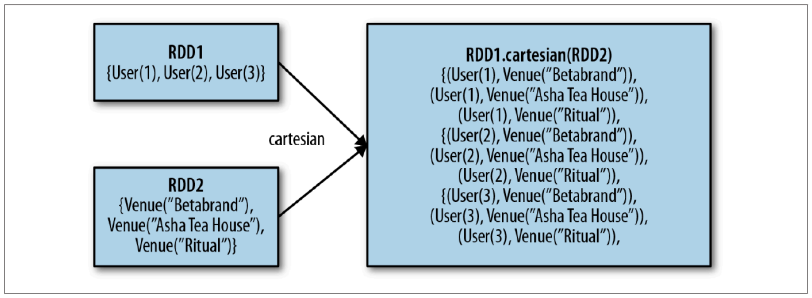


Figure 3-5. Cartesian product between two RDDs

cartesian is an expensive transformation, especially for large RDDs

Basic Actions

Actions DO NOT return RDDs!

- reduce: Aggregates elements in RDD according to a function:
  - The function must take two arguments and return a new element of the same type.
  - The function must be commutative and associative.

```
nums = sc.parallelize([2,1,1,2,3,1])
sumOfNums = nums.reduce(lambda x,y: x + y)
print sumOfNums  # Prints 10
```

```
nums = sc.parallelize([2,1,1,2,3,1])
def getMax(x,y):
    if x > y:
        return x
    else:
        return y
print nums.reduce(getMax) # Prints 3
```

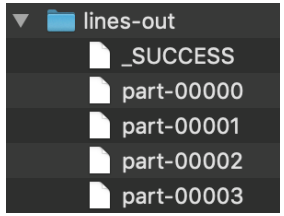
Basic Actions

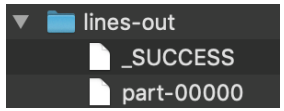
- saveAsTextFile: Writes the elements in RDD as a text file (or set of text files) in a given directory.

```
lines = sc.parallelize(["tea", "coffee"])
lines.saveAsTextFile("/example-data/lines-out")

lines.coalesce(1).saveAsTextFile("/example-data/lines-out")
```

files for partitions!





- collect: Returns the entire RDD's contents to the driver program.
  - Restriction: All data must fit on a single machine. Mostly used for unit tests and sufficiently small data.

```
print nums
# ParallelCollectionRDD[134] at parallelize at PythonRDD.scala:489
print nums.collect()
# Prints [2, 1, 1, 2, 3, 1]
```

Basic RDD Actions

Table 3-4. Basic actions on an RDD containing {1, 2, 3, 3}

Function name	Purpose	Example	Result
`collect()`	Return all elements from the RDD.	`rdd.collect()`	`{1, 2, 3, 3}`
`count()`	Number of elements in the RDD.	`rdd.count()`	`4`
`countByValue()`	Number of times each element occurs in the RDD.	`rdd.countByValue()`	`{(1, 1), (2, 1), (3, 2)}`
`reduce(func)`	Combine the elements of the RDD together in parallel (e.g., `sum`).	`rdd.reduce((x, y) => x + y)`	`9`
`take(num)`	Return `num` elements from the RDD.	`rdd.take(2)`	`{1, 2}`
`top(num)`	Return the top `num` elements the RDD.	`rdd.top(2)`	`{3, 3}`
`takeOrdered(num)(ordering)`	Return `num` elements based on provided ordering.	`rdd.takeOrdered(2)(myOrdering)`	`{3, 3}`
`foreach(func)`	Apply the provided function to each element of the RDD.	`rdd.foreach(func)`	Nothing

NTNU

Norwegian University of Science and Technology

Programming with RDD API

Key-Value Pairs and Design Considerations

Pair RDDs: Key-Value Pairs

- Counting up reviews for each product, grouping together data with the same key, grouping together two different RDDs...
- Creating key-value pairs:
  - Using map transformation to create tuples of the form key-value

```
words = sc.parallelize(["tea", "coffee"])
pairs = words.map(lambda x: (x,1))
print pairs.collect()  # prints [('tea', 1), ('coffee', 1)]
```

  - Special format to-from files

```
pairs.saveAsSequenceFile('/Users/oozdikis/example-data/sequence-out')
pairs = sc.sequenceFile('/Users/oozdikis/example-data/sequence-out')
```

  - Parallelizing a collection of pairs

```
pairs = sc.parallelize([("a",2),("b",3),("a",1)])
print pairs.keys().collect()  # prints ['a', 'b', 'a']
```

Pair RDD Transformations

- reduceByKey and sortByKey
- Word count example...

the file contains 3 lines:

```
Deer Bear River
Car Car River
Deer Car Bear
```

```
lines = sc.textFile("/example-data/sampletextfile.txt")
words = lines.flatMap(lambda line: line.split(" "))
word_tuples = words.map(lambda x: (x,1))
# word_tuples: [(u'Deer', 1), (u'Bear', 1), (u'River', 1),
#   (u'Car', 1), (u'Car', 1), (u'River', 1), (u'Deer', 1), (u'Car', 1), (u'Bear', 1)]
word_counts = word_tuples.reduceByKey(lambda n, m: n + m)
# word_counts: [(u'Deer', 2), (u'Bear', 2), (u'Car', 3), (u'River', 2)]
keysorted_word_counts = word_counts.sortByKey()
# keysorted_word_counts: [(u'Bear', 2), (u'Car', 3), (u'Deer', 2), (u'River', 2)]
```

How could we sort the elements by word counts?

possible to convert key-value tuples using map()

```
counts_words = word_counts.map(lambda (x,y): (y,x))
# counts_words: [(2, u'Deer'), (2, u'Bear'), (3, u'Car'), (2, u'River')]
sorted_counts_words = counts_words.sortByKey(False)
# sorted_counts_words: [(3, u'Car'), (2, u'Deer'), (2, u'Bear'), (2, u'River')]
```

Note that reduceByKey is a transformation, thus it returns an RDD.
(unlike reduce, which is an Action)

Pair RDD Transformations

- More examples...
  - Different data types for keys and values:

```
lines = sc.textFile("/example-data/samplelog.txt")
words = lines.map(lambda line: line.split("\t"))
level_app_tuple = words.map(lambda x: ((str(x[1]),str(x[2])), 1))
level_app_counts = level_app_tuple.reduceByKey(lambda x,y: x + y)
level_app_counts.collect()
# Prints [(('error', 'foo'), 3), (('debug', 'ok'), 5), (('error', 'bar'), 1)]
```

the file contains lines with (id, level, message)

```
1   debug   ok
2   debug   ok
3   error   foo
4   debug   ok
5   error   bar
6   debug   ok
7   debug   ok
8   error   foo
9   error   foo
```

key and value in the pair can also be tuples themselves

  - Maximum value per key:

```
lines = sc.textFile("/example-data/sampletemp.txt")
words = lines.map(lambda line: line.split("\t"))
year_temps = words.map(lambda line: (int(line[0]), int(line[1])))
max_temps_per_year = year_temps.reduceByKey(lambda x,y: x if x > y else y)
# Prints [(1950, 22), (1949, 20)]
```

the file contains lines with (year, temperature)

```
1949    15
1949    20
1950    0
1950    22
1950    -10
```

maximum temperature per year

Pair RDD Transformations

Table 4-1. Transformations on one pair RDD (example: {(1, 2), (3, 4), (3, 6)})

Function name	Purpose	Example	Result
`reduceByKey(func)`	Combine values with the same key.	`rdd.reduceByKey( (x, y) => x + y)`	`{(1, 2), (3, 10)}`
`groupByKey()`	Group values with the same key.	`rdd.groupByKey()`	`{(1, [2]), (3, [4, 6])}`
`keys()`	Return an RDD of just the keys.	`rdd.keys()`	`{1, 3, 3}`
`values()`	Return an RDD of just the values.	`rdd.values()`	`{2, 4, 6}`
`sortByKey()`	Return an RDD sorted by the key.	`rdd.sortByKey()`	`{(1, 2), (3, 4), (3, 6)}`
`mapValues(func)`	Apply a function to each value of a pair RDD without changing the key.	`rdd.mapValues(x => x+1)`	`{(1, 3), (3, 5), (3, 7)}`
`flatMapValues(func)`	Apply a function that returns an iterator to each value of a pair RDD, and for each element returned, produce a key/value entry with the old key. Often used for tokenization.	`rdd.flatMapValues(x => (x to 5)`	`{(1, 2), (1, 3), (1, 4), (1, 5), (3, 4), (3, 5)}`

Two Pair RDD Transformations

Table 4-2. Transformations on two pair RDDs (rdd = {(1, 2), (3, 4), (3, 6)} other = {(3, 9)

Function name	Purpose	Example	Result
`subtractByKey`	Remove elements with a key present in the other RDD.	`rdd.subtractByKey(other)`	`{(1, 2)}`
`join`	Perform an inner join between two RDDs.	`rdd.join(other)`	`{(3, (4, 9)), (3, (6, 9))}`
`rightOuterJoin`	Perform a join between two RDDs where the key must be present in the other RDD.	`rdd.rightOuterJoin(other)`	`{(3,(Some(4),9)), (3,(Some(6),9))}`
`leftOuterJoin`	Perform a join between two RDDs where the key must be present in the first RDD.	`rdd.leftOuterJoin(other)`	`{(1,(2,None)), (3, (4,Some(9))), (3, (6,Some(9)))}`
`cogroup`	Group data from both RDDs sharing the same key.	`rdd.cogroup(other)`	`{(1,([2],[])), (3, ([4, 6],[9]))}`

Pair RDD Actions

Table 4-3. Actions on pair RDDs (example ({(1, 2), (3, 4), (3, 6)}))

Function	Description	Example	Result
`countByKey()`	Count the number of elements for each key.	`rdd.countByKey()`	`{(1, 1), (3, 2)}`
`collectAsMap()`	Collect the result as a map to provide easy lookup.	`rdd.collectAsMap()`	`Map{(1, 2), (3, 4), (3, 6)}`
`lookup(key)`	Return all values associated with the provided key.	`rdd.lookup(3)`	`[4, 6]`

! Scope of Objects in Functions

- Spark computes a task's closure (variables and methods which must be visible for the executor to perform its computations).
- Closure is serialized and sent to each executor.

```
counter = 0
rdd = sc.parallelize(data)

# Wrong: Don't do this!!
def increment_counter(x):
    global counter
    counter += x
rdd.foreach(increment_counter)

print("Counter value: ", counter)
```

- It may work in "Local Mode", but consider "Cluster Mode".
- Distributed execution in cluster mode: No single JVM

Methods should not be used to mutate some global state!

! Serialization of Objects in Functions

- In a transformation, if you pass a function that
  - contains references to fields in an object, or
  - is the member of an object,

  Spark sends **the entire object** to the worker nodes.

Example 3-19. Passing a function with field references (don't do this!)

```
class SearchFunctions(object):
  def __init__(self, query):
      self.query = query
  def isMatch(self, s):
      return self.query in s
  def getMatchesFunctionReference(self, rdd):
      # Problem: references all of "self" in "self.isMatch"
      return rdd.filter(self.isMatch)
  def getMatchesMemberReference(self, rdd):
      # Problem: references all of "self" in "self.query"
      return rdd.filter(lambda x: self.query in x)
```

Example 3-20. Python function passing without field references

```
class WordFunctions(object):
  ...
  def getMatchesNoReference(self, rdd):
      # Safe: extract only the field we need into a local variable
      query = self.query
      return rdd.filter(lambda x: query in x)
```

Careful with the references to large objects in functions!

These objects are serialized to all nodes!

Persistence (Caching)

- Spark recomputes the RDD and all of its dependencies each time we call an action on the RDD.
- To avoid recomputation → Persistence

```
lines = sc.textFile("hdfs://...")
sptLines = lines.map(lambda line: line.split('\t'))
errLines = sptLines.filter(lambda l: l[1] == "error")
errLines.cache()
print errLines.count()
print errLines.filter(lambda l: l[2] == "foo").count()
print errLines.filter(lambda l: l[2] == "bar").count()
```

Table 3-6. Persistence levels from org.apache.spark.storage.StorageLevel and pyspark.StorageLevel; if desired we can replicate the data on two machines by adding _2 to the end of the storage level

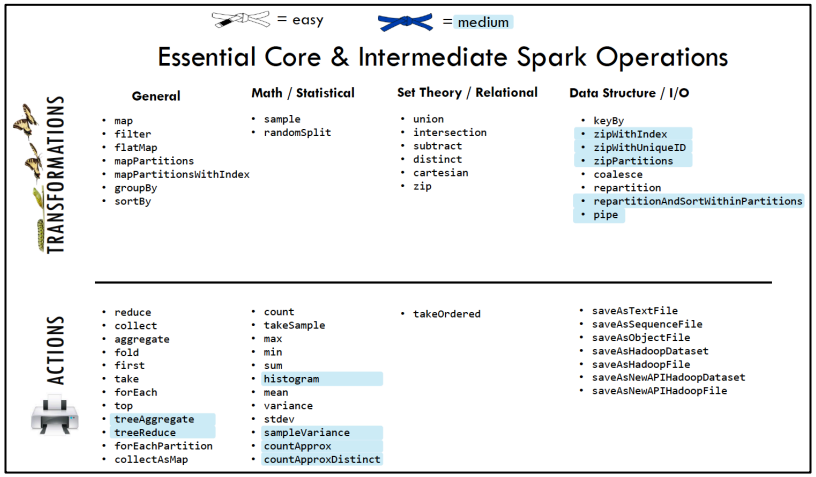
Level	Space used	CPU time	In memory	On disk	Comments
MEMORY_ONLY	High	Low	Y	N	
MEMORY_ONLY_SER	Low	High	Y	N	
MEMORY_AND_DISK	High	Medium	Some	Some	Spills to disk if there is too much data to fit in memory.
MEMORY_AND_DISK_SER	Low	High	Some	Some	Spills to disk if there is too much data to fit in memory. Stores serialized representation in memory.
DISK_ONLY	Low	High	N	Y	

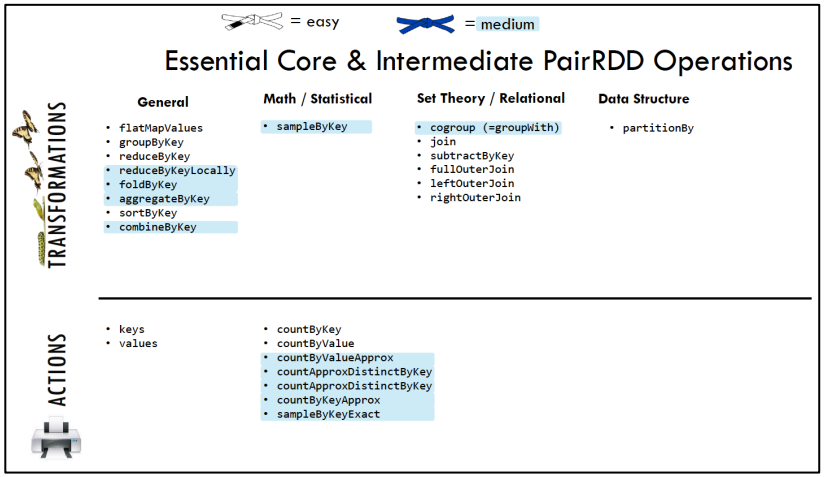
. The `cache()` method is a shorthand for using the default storage level, which is `StorageLevel.MEMORY_ONLY`

For more on RDD API...

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

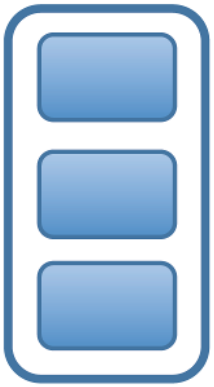
https://training.databricks.com/visualapi.pdf





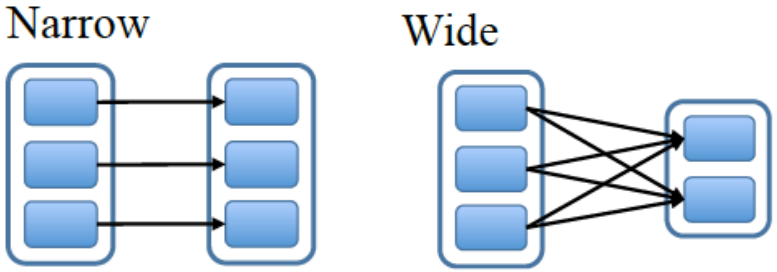
RDD Representation and Other Spark Features

Representing RDDs



Information to represent RDDs:

- Partitions: Atomic pieces of the dataset
  - E.g., one partition for each block of an HDFS file
- Locations of partitions
  - E.g., which machines each block is on for an HDFS file
- Partitioning: Metadata specifying whether the RDD is hash/ range partitioned (can improve the performance)
- Dependencies: Which parent RDDs an RDD is based on (used to reconstruct an RDD if needed)
  - Narrow dependency: Every partition in a parent RDD is used by maximum one partition in child RDD.
  - Wide (shuffle) dependency: Multiple child partitions may depend on a partition in parent RDD
  - Examples →



Narrow vs. Wide Dependencies

- Narrow: Allows for pipelined execution on one cluster node. Provides easier and faster recovery in case of failure.
- Wide: Partitions need to be shuffled across the nodes.

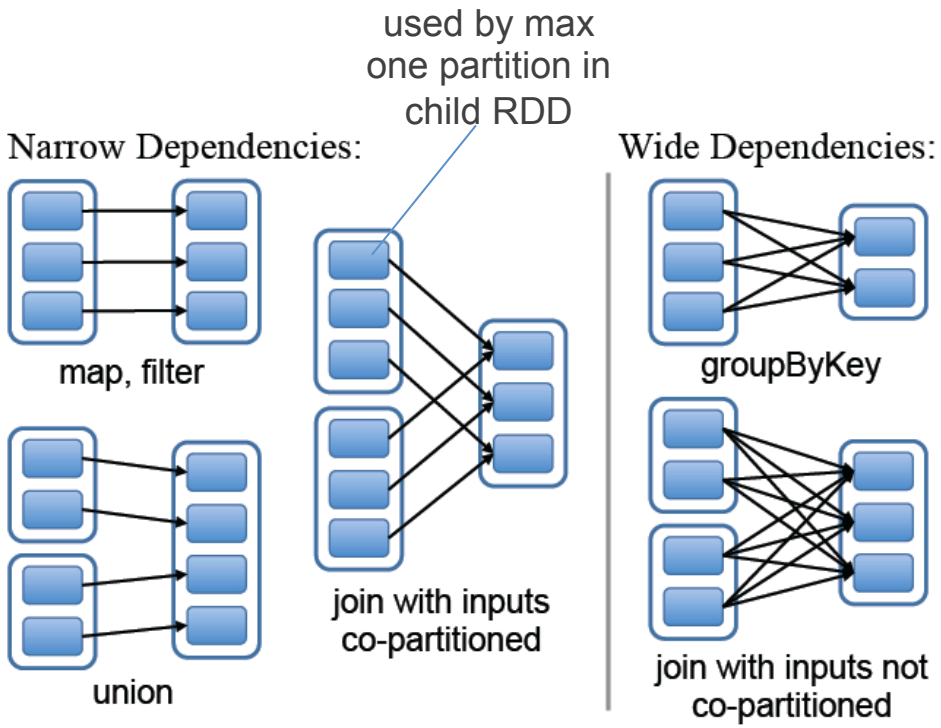


Figure 4: Examples of narrow and wide dependencies. Each box is an RDD, with partitions shown as shaded rectangles.

Example: Partitioning before Join

- Joining two RDDs may lead to:
  - two narrow dependencies if parents are both hash/range partitioned with the same partitioner,
  - two wide dependencies if there is no hash/range partitioning,
  - or a mix if one parent has a partitioner and one does not.

```
val userData = sc.sequenceFile[UserID, UserInfo]("hdfs://...").persist()

val events = sc.sequenceFile[UserID, LinkInfo](logFileName)
val joined = userData.join(events)
```

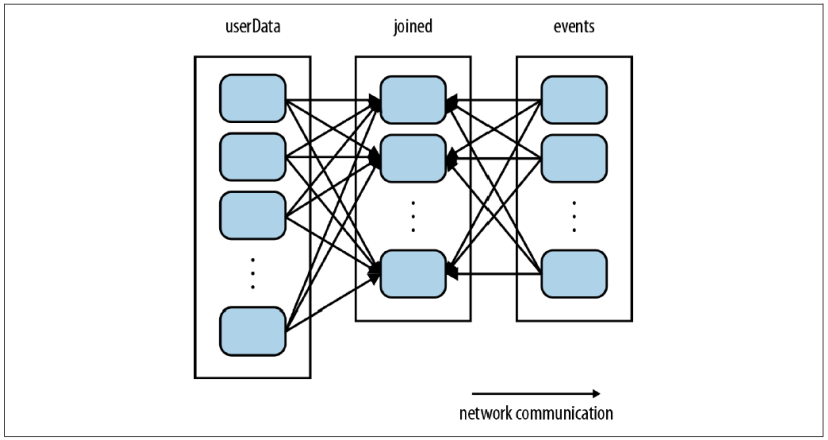


Figure 4-4. Each join of userData and events without using partitionBy()

```
val userData = sc.sequenceFile[UserID, UserInfo]("hdfs://...")
                 .partitionBy(new HashPartitioner(100))
                 .persist()
val events = sc.sequenceFile[UserID, LinkInfo](logFileName)
val joined = userData.join(events)
```

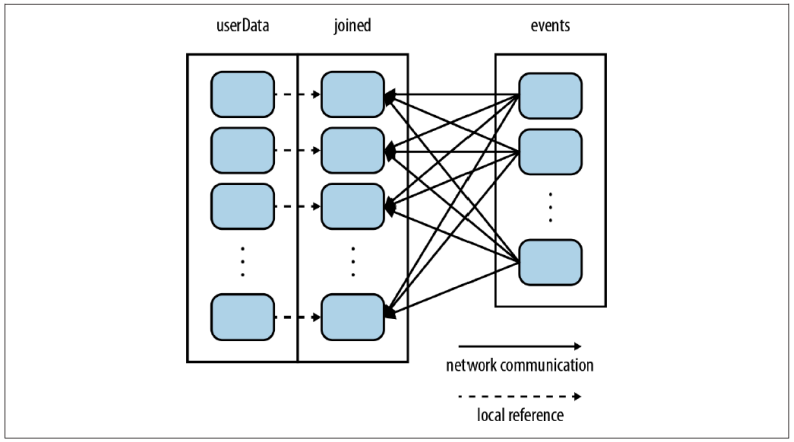
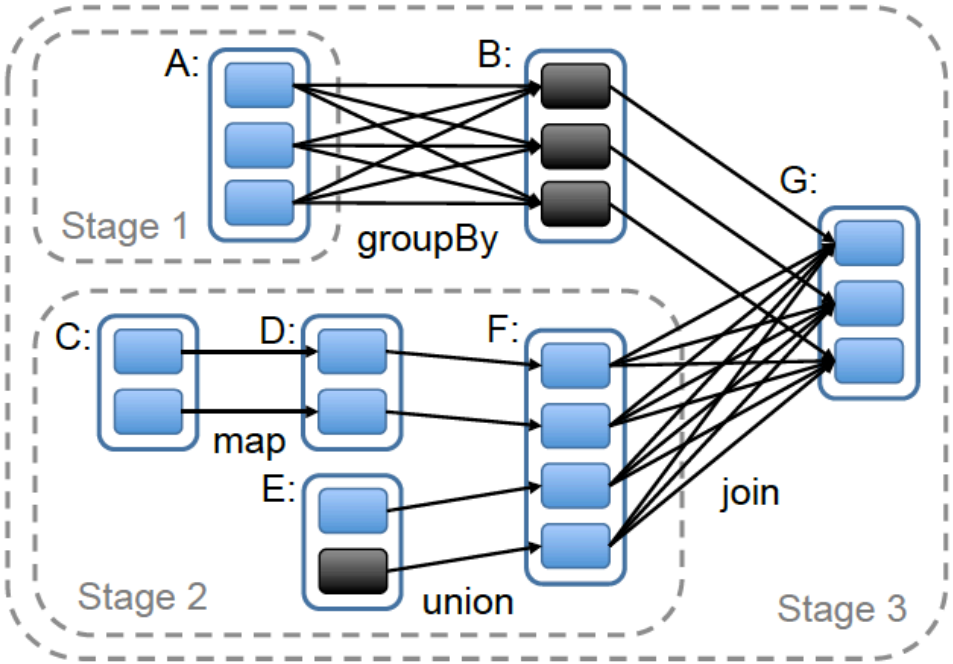


Figure 4-5. Each join of userData and events using partitionBy()

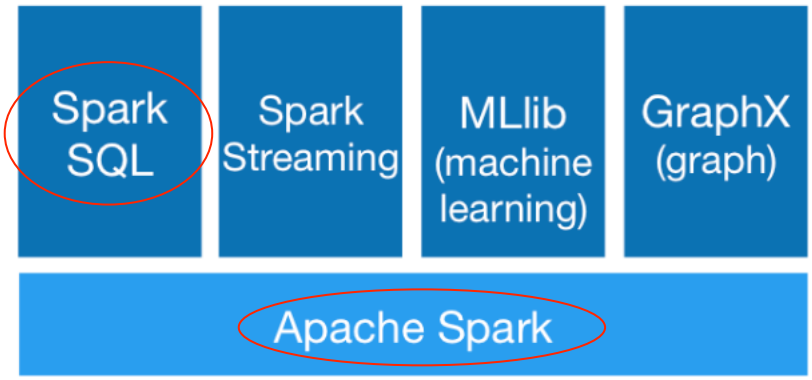
Example Execution in Spark

When you run an action on RDD...

- The scheduler builds a graph of stages (a collection of tasks that run the same code, each on a different subset of the data),
- Each stage contains transformations with narrow dependencies,
- Stage boundaries are shuffle operations and persisted partitions.



Other Spark Components



- Spark SQL and DataFrames:
  - Conceptually equivalent to a table in a relational database system
  - Distributed collection of row-objects and named columns
  - Can be constructed from structured data files, tables in Hive, external databases, or existing RDDs
  - DataFrame API support SQL queries

Spark SQL and DataFrames

- DataFrame (SQL) API example:

```
sqlContext = SQLContext(sc)
temperaturesDF = sqlContext.read.format("com.databricks.spark.csv")
    .option("header", "false").option("inferSchema", "true")
    .option("delimiter", '\t').load("/example-data/sampletemperature.txt")
temperaturesDF.printSchema()


temperaturesDF = temperaturesDF.toDF('year', 'temperature')
temperaturesDF.createOrReplaceTempView('temperatures')
sqlContext.sql('select count(*) from temperatures').show()



sqlContext.sql('select max(temperature) from temperatures').show()



sqlContext.sql('select * from temperatures').show()
```

tab-separated input file
(year, temperature)

```
1949    15
1949    20
1950    0
1950    22
1950    -10
```

```
root
 |-- _c0: integer (nullable = true)
 |-- _c1: integer (nullable = true)
```

```
+--------+
|count(1)|
+--------+
|       5|
+--------+
```

```
+----------------+
|max(temperature)|
+----------------+
|              22|
+----------------+
```

```
+----+-----------+
|year|temperature|
+----+-----------+
|1949|         15|
|1949|         20|
|1950|          0|
|1950|         22|
|1950|        -10|
+----+-----------+
```

For more on Spark...

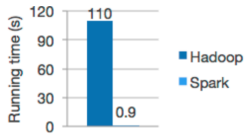
https://spark.apache.org/

Apache Spark™ is a unified analytics engine for large-scale data processing.

Speed

Run workloads 100x faster.

Apache Spark achieves high performance for both batch and streaming data, using a state-of-the-art DAG scheduler, a query optimizer, and a physical execution engine.



Logistic regression in Hadoop and Spark

Ease of Use

Write applications quickly in Java, Scala, Python, R, and SQL.

Spark offers over 80 high-level operators that make it easy to build parallel apps. And you can use it *interactively* from the Scala, Python, R, and SQL shells.

```
df = spark.read.json("logs.json")
df.where("age > 21")
  .select("name.first").show()
```

Spark's Python DataFrame API
Read JSON files with automatic schema inference

Generality

Combine SQL, streaming, and complex analytics.

Spark powers a stack of libraries including SQL and DataFrames, MLlib for machine learning, GraphX, and Spark Streaming. You can combine these libraries seamlessly in the same application.