In-Memory Computation with Spark

Lecture BigData Analytics

Julian M. Kunkel

julian.kunkel@googlemail.com

University of Hamburg / German Climate Computing Center (DKRZ)

15-01-2016

Outline

Overview to Spark [10, 12]

- In-memory **processing** (and **storage**) engine
  - Load data from HDFS, Cassandra, HBase
  - Resource management via. YARN, Mesos, Spark, Amazon EC2
  - $\Rightarrow$ It can use Hadoop, but also works standalone!
- Platform schedules tasks and monitors them
- Rich APIs
  - APIs for Java, Scala, Python, R
  - Thrift JDBC/ODBC server for SQL
  - High-level domain-specific tools/languages
    - Advanced APIs simplify typical computation tasks
- Interactive shells with tight integration
  - `spark-shell`: Scala (object-oriented functional language running on JVM)
  - `pyspark`: Python
- Execution in either local or cluster mode

Data Model [13]

- Distributed memory model: Resilient Distributed Datasets (RDDs)
  - Named collection of elements distributed in partitions

$X = [1, 2, 3, 4, 5, ..., 1000]$ distributed into 4 partitions

  - Typically a list or a map (key-value) pairs
  - A RDD is immutatable, e.g. cannot be changed
  - High-level APIs provide additional representations
    - e.g. SparkSQL uses DataFrame (aka tables)
- Shared variables offer shared memory access
- Durability of data
  - RDDs live until the SparkContext is terminated
  - To keep them, they need to be persisted (e.g. to HDFS)
- Fault-tolerance is provided by **re-computing** data

Resilient Distributed Datasets (RDDs) [13]

- Creation of a RDD by either
  - Parallelizing an existing collection

```
1 data = [1, 2, 3, 4, 5]
2 rdd = sc.parallelize(data, 5) # create 5 partitions
```

  - Referencing a dataset on distributed storage, HDFS, ...

```
1 rdd = sc.textFile("data.txt")
```

- RDDs can be transformed into derived (newly named) RDDs
  - Computation runs in parallel on the partitions
  - Usually re-computed, but RDD can be kept in memory or stored if large
  - RDDs can be redistributed (called shuffle)
  - Knows its data lineage (how it was computed)
- Fault-tolerant collection of elements (lists, dictionaries)
  - Split into choosable number of partitions and distributed
  - Derived RDDs can be re-computed by using the recorded lineage

Shared Variables [13]

- Broadcast variables for read-only access
  - For readability, do not modify the broadcast variable later

```
broadcastVar = sc.broadcast([1, 2, 3])
print (broadcastVar.value)
# [1, 2, 3]
```

- Accumulators (reduce variables)
  - Are counters that can be incremented
  - Other data types can be supported

```
accum = sc.accumulator(0) # Integer accumulator
accum.add(4)

# Accumulator for adding vectors
class VectorAccumulatorParam(AccumulatorParam):
    def zero(self, initialValue):
        return Vector.zeros(initialValue.size)

    def addInPlace(self, v1, v2):
        v1 += v2
        return v1
# Create an accumulator
vecAccum = sc.accumulator(Vector(...), VectorAccumulatorParam())
```

Computation

- Lazy execution: apply operations when results are needed (by actions)
  - Intermediate RDDs can be re-computed multiple times
  - Users can persist RDDs (in-memory or disk) for later use
- Many operations apply user-defined functions or lambda expressions
- Code and **closure** are serialized on the driver and send to executors
  - Note: When using class instance functions, the object is serialized
- RDD partitions are processed in parallel (data parallelism)
  - Use local data where possible

RDD Operations [13]

- Transformations: create a new RDD locally by applying operations
- Actions: return values to the driver program
- Shuffle operations: re-distribute data across executors

Simple Example

- Example session when using pyspark
- To run with a specific Python version e.g. use

```
PYSPARK_PYTHON=python3 pyspark --master yarn-client
```

Example data-intensive python program

```
# Distribute the data: here we have a list of numbers from 1 to 10 million
# Store the data in an RDD called nums
nums = sc.parallelize( range(1,10000000) )

# Compute a derived RDD by filtering odd values
r1 = nums.filter( lambda x : (x % 2 == 1) )

# Now compute squares for all remaining values and store key/value tuples
result = r1.map( lambda x : (x, x*x*x) )

# Retrieve all distributed values into the driver and print them
# This will actually run the computation
print(result.collect())
# [(1, 1), (3, 27), (5, 125), (7, 343), (9, 729), (11, 1331), ... ]

# Store results in memory
resultCached = result.cache()
```

Compute PI [20]

- Randomly throw NUM_SAMPLES darts on a circle and count hits

Python

```python
def sample(p):
    x, y = random(), random()
    return 1 if x*x + y*y < 1 else 0

count = sc.parallelize(xrange(0, NUM_SAMPLES)).map(sample).reduce(lambda a, b: a + b)
print "Pi is roughly %f" % (4.0 * count / NUM_SAMPLES)
```

Java

```java
int count = spark.parallelize(makeRange(1, NUM_SAMPLES)).filter(
  new Function<Integer, Boolean>() {
  public Boolean call(Integer i) {
    double x = Math.random();
    double y = Math.random();
    return x*x + y*y < 1;
  }
}).count();
System.out.println("Pi is roughly " + 4 * count / NUM_SAMPLES);
```

Execution of Applications [12, 21]

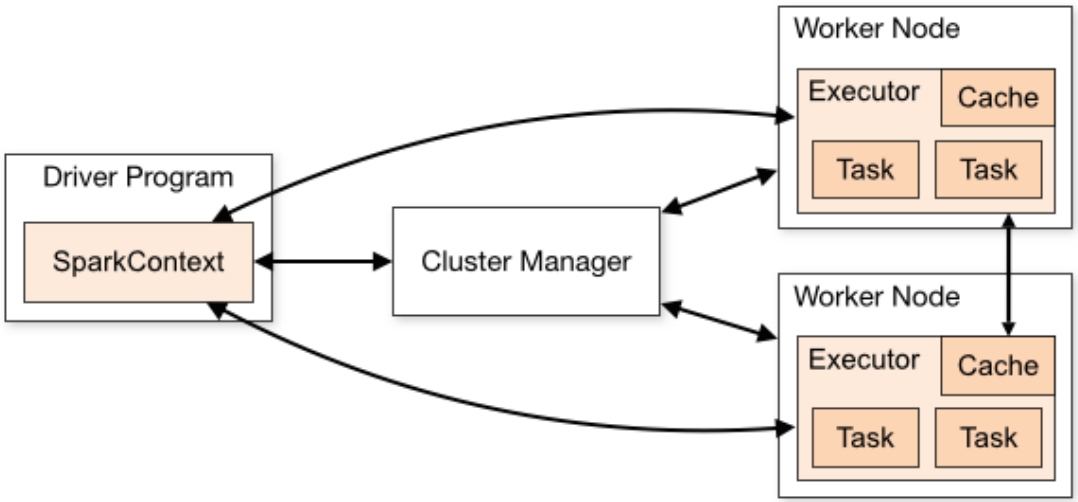


Figure: Source: [12]

- Driver program: process runs `main()`, creates/uses SparkContext
- Task: A unit of work processed by one executor
- Job: A spark action triggering computation starts a job
- Stage: collection of tasks executing the same code; run concurrently
  - Works independently on partitions without data shuffling
- Executor process: provides slots to runs tasks
  - Isolates apps, thus data cannot be shared between apps
- Cluster manager: allocates cluster resources and runs executor

Data Processing [13]

- Driver (main program) controls data flow/computation
- Executor processes are spawned on nodes
  - Store and manage RDDs
  - Perform computation (usually on local partition)
- In local only one executor is created

Execution of code

1. The closure is computed: variables/methods needed for execution
2. The driver serializes the closure together with the task (code)
   - Broadcast vars are useful as they do not require to be packed with the task
3. The driver sends the closure to the executors
4. Tasks on the executor run the closure which manipulates the local data

Transformations Create a New RDD [13]

All RDDs support

- map(func): pass each element through func
- filter(func): include those elements for which func returns true
- flatMap(func): similar to map, but func returns a list of elements
- mapPartitions(func): like map but runs on each partition independently
- sample(withReplacement, fraction, seed): pick a random fraction of data
- union(otherDataset): combine two datasets
- intersection(otherDataset): set that contains only elements in both sets
- distinct([numTasks]): returns unique elements
- cartesian(otherDataset): returns all pairs of elements
- pipe(command, [envVars]): pipe each partition through a shell command

Remember: Transformations return a lazy reference to a new dataset

Transformations Create a New RDD [13]

Key/Value RDDs additionally support

- groupByKey([numTasks]): combines values of identical keys in a list
- reduceByKey(func, [numTasks]): aggregate all values of each key
- aggregateByKey(zeroValue, seqOp, combOp, [numTasks]): aggregates values for keys, uses neutral element
- sortByKey([ascending], [numTasks]): order the dataset
- join(otherDataset, [numTasks]): pairs (K,V) elements with (K,U) and returns (K, (V,U))
- cogroup(otherDataset, [numTasks]): returns (K, iterableV, iterableU)

Actions: Perform I/O or return data to the driver [13]

- reduce(func): aggregates elements $func(x, y) \Rightarrow z$
  - Func should be commutative and associative
- count(): number of RDD elements
- countByKey(): for K/V, returns hashmap with count for each key
- foreach(func): run the function on each element of the dataset
  - Useful to update an accumulator or interact with storage
- collect(): returns the complete dataset to the driver
- first(): first element of the dataset
- take(n): array with the first n elements
- takeSample(withReplacement, num, [seed]): return random array
- takeOrdered(n, [comparator]): first elements according to an order
- saveAsTextFile(path): convert elements to string and write to a file
- saveAsSequenceFile(path): ...
- saveAsObjectFile(path): uses Java serialization

Shuffle [13]

Concepts

- Repartitions the RDD across the executors
- Costly operation (requires all-to-all)
- May be triggered implicitly by operations or can be enforced
- Requires network communication
- The number of partitions can be set

Operations

- repartition(numPartitions): reshuffle the data randomly into partitions
- coalesce(numPartitions): decrease the number of partitions[1]
- repartionAndSortWithinPartitions(partitioner): repartition according to the partitioner, then sort each partition locally.[1]

[1] More efficient than repartition()

Persistence [13]

Concepts

- The data lineage of an RDD is stored
- Actions trigger computation, no intermediate results are kept
- The methods `cache()` and `persist()` enable copy of
  - The first time a RDD is computed, it is then kept for further usage
  - Each executor keeps its local data
  - cache() keeps data in memory (level: MEMORY_ONLY)
  - persist() allows to choose the storage level
- Spark manages the memory cache automatically
  - LRU cache, old RDDs are evicted to secondary storage (or deleted)
  - If an RDD is not in cache, re-computation may be triggered

Storage levels

- MEMORY_ONLY: keep Java objects in memory, or re-compute them
- MEMORY_AND_DISK: keep Java objects in memory or store them on disk
- MEMORY_ONLY_SER: keep serialized Java objects in memory
- DISK_ONLY: store the data only on secondary storage

Parallelism [13]

- Spark runs one task for each partition of the RDD
- Recommendation: create 2-4 partitions for each CPU
- When creating a RDD a default value is set, but can be changed manually

```
1 # Create 10 partitions when the data is distributed
2 sc.parallelize(data, 10)
```

- The number of partitions is inherited from the parent(s) RDD
- Shuffle operations contain the argument numTasks
  - Define the number of partitions for the new RDD
- Some actions/transformations contain numTask
  - Define the number of reducers
  - By default, 8 parallel tasks for groupByKey() and reduceByKey()
- Analyze the data partitioning using glom()
  - It returns a list with RDD elements for each partition

```
1 X.glom().collect()
2 # [[], [ 1, 4 ], [ ], [ 5 ], [ 2 ] ] => here we have 5 partitions for RDD X
3 # Existing values are 1, 2, 4, 5 (not balanced in this example)
```

Typical Mistakes [13]

Use local variables in distributed memory

```
1 counter = 0
2 rdd = sc.parallelize(data)
3
4 # Wrong: since counter is a local variable, it is updated in each JVM
5 # Thus, each executor yields another result
6 rdd.foreach(lambda x: counter += x)
7 print("Counter value: " + counter)
```

Object serialization may be unexpected (and slow)

```
1 class MyClass(object):
2   def func(self, s):
3     return s
4   def doStuff(self, rdd):
5   # Run method in parallel but requires to serialize MyClass with its members
6     return rdd.map(self.func)
```

Writing to STDOUT/ERR on executors

```
1 # This will call println() on each element
2 # However, the executors' stdout is not redirected to the driver
3 rdd.foreach(println)
```

Using YARN with Spark [18, 19]

- Two alternative deployment modes: cluster and client
- Interactive shells/driver requires client mode
- Spark dynamically allocates the number of executors based on the load
  - Set num-executors manually to disable this feature

```
1 PYSPARK_PYTHON=python3 pyspark --master yarn-client --driver-memory 4g
      ↪ --executor-memory 4g  --num-executors 5 --executor-cores 24 --conf
      ↪ spark.ui.port=4711
```

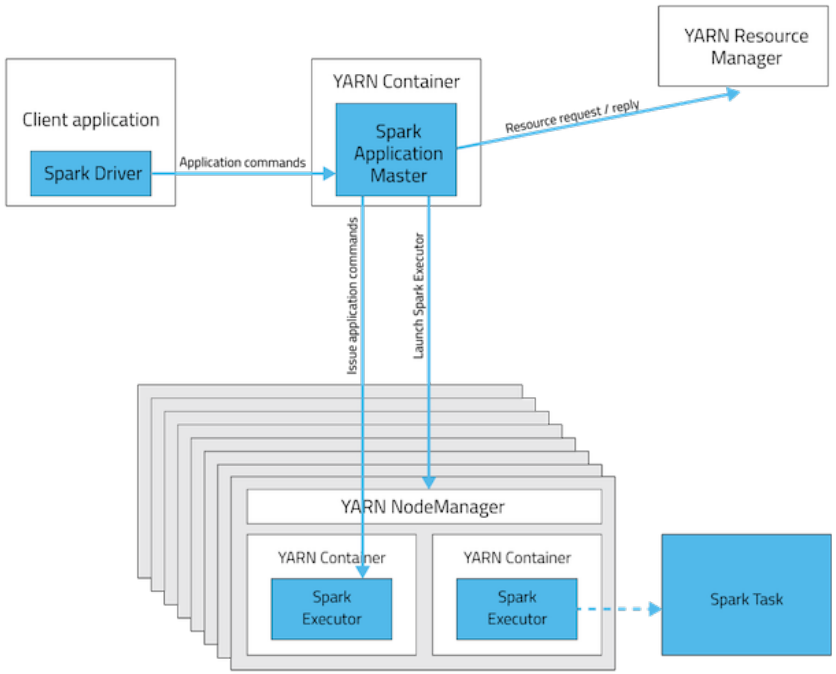


Figure: Source: [18]

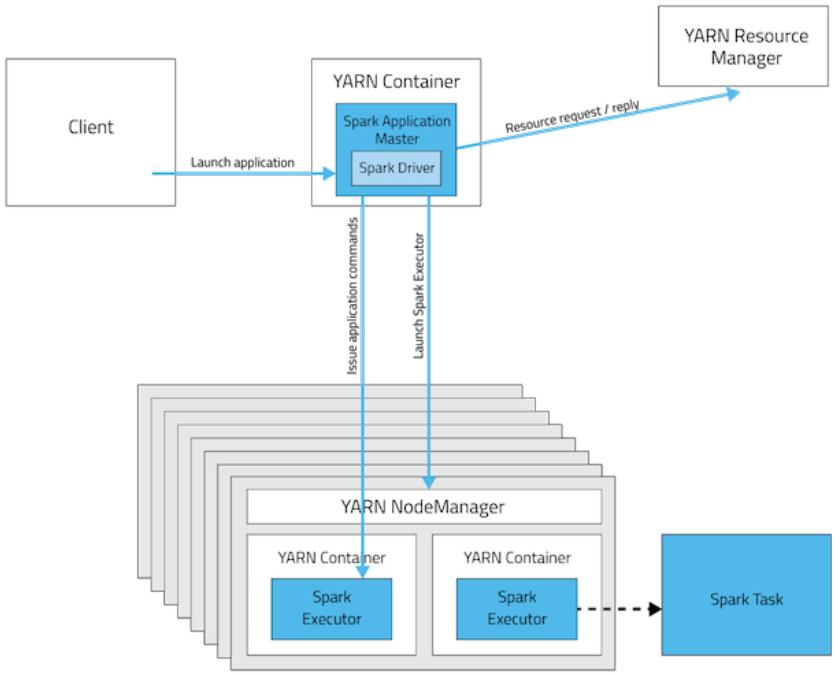


Figure: Source: [18]

Batch Applications

- Submit batch applications via `spark-submit`
- Supports JARs (Scala or Java)
- Supports Python code
- To query results check output (tracking URL)
- Build self-contained Spark applications (see [24])

```
spark-submit --master <master-URL>  --class <MAIN> # for Java/Scala Applications
  --conf <key>=<value>  --py-files x,y,z # Add files to the PYTHONPATH
  --jars <(hdfs|http|file|local)>://<FILE> # provide JARs to the classpath
  <APPLICATION> [APPLICATION ARGUMENTS]
```

Examples for Python and Java

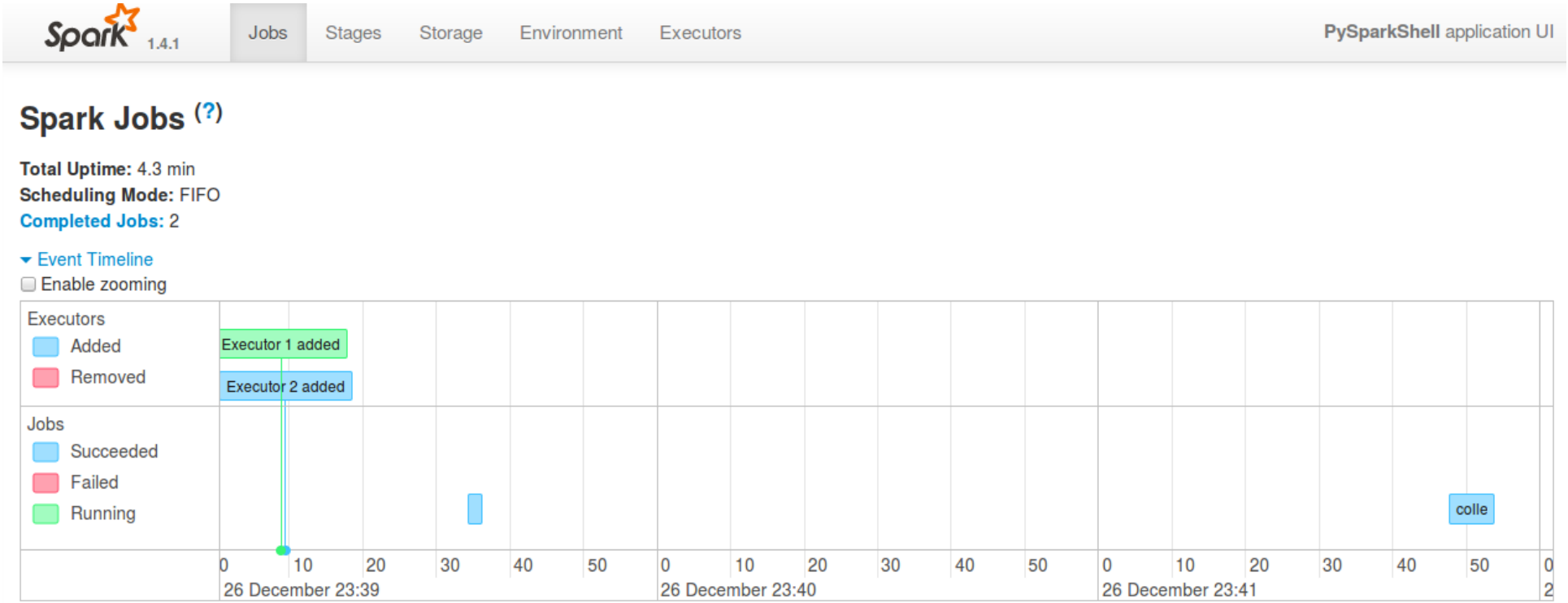
```
SPARK=/usr/hdp/2.3.2.0-2950/spark/
PYSPARK_PYTHON=python3 spark-submit --master yarn-cluster --driver-memory 4g
    ↪ --executor-memory 4g  --num-executors 5 --executor-cores 24
    ↪ $SPARK/examples/src/main/python/pi.py 120
# One minute later, output in YARN log: "Pi is roughly 3.144135"

spark-submit --master yarn-cluster --driver-memory 4g --executor-memory 4g
    ↪ --num-executors 5 --executor-cores 24 --class
    ↪ org.apache.spark.examples.JavaSparkPi $SPARK/lib/spark-examples-*.jar 120
```

Web UI

- Sophisticated analysis of performance issues
- Monitoring features
  - Running/previous jobs
  - Details for job execution
  - Storage usage (cached RDDs)
  - Environment variables
  - Details about executors
- Started automatically when a Spark shell is run
  - On our system available on Port 4040 [2]
- Creates web-pages in YARN UI
  - While running automatically redirects from 4040 to the YARN UI
  - Historic data visit “tracking URL” in YARN UI

[2] Change it by adding `-conf spark.ui.port=P0RT` to e.g. pyspark.

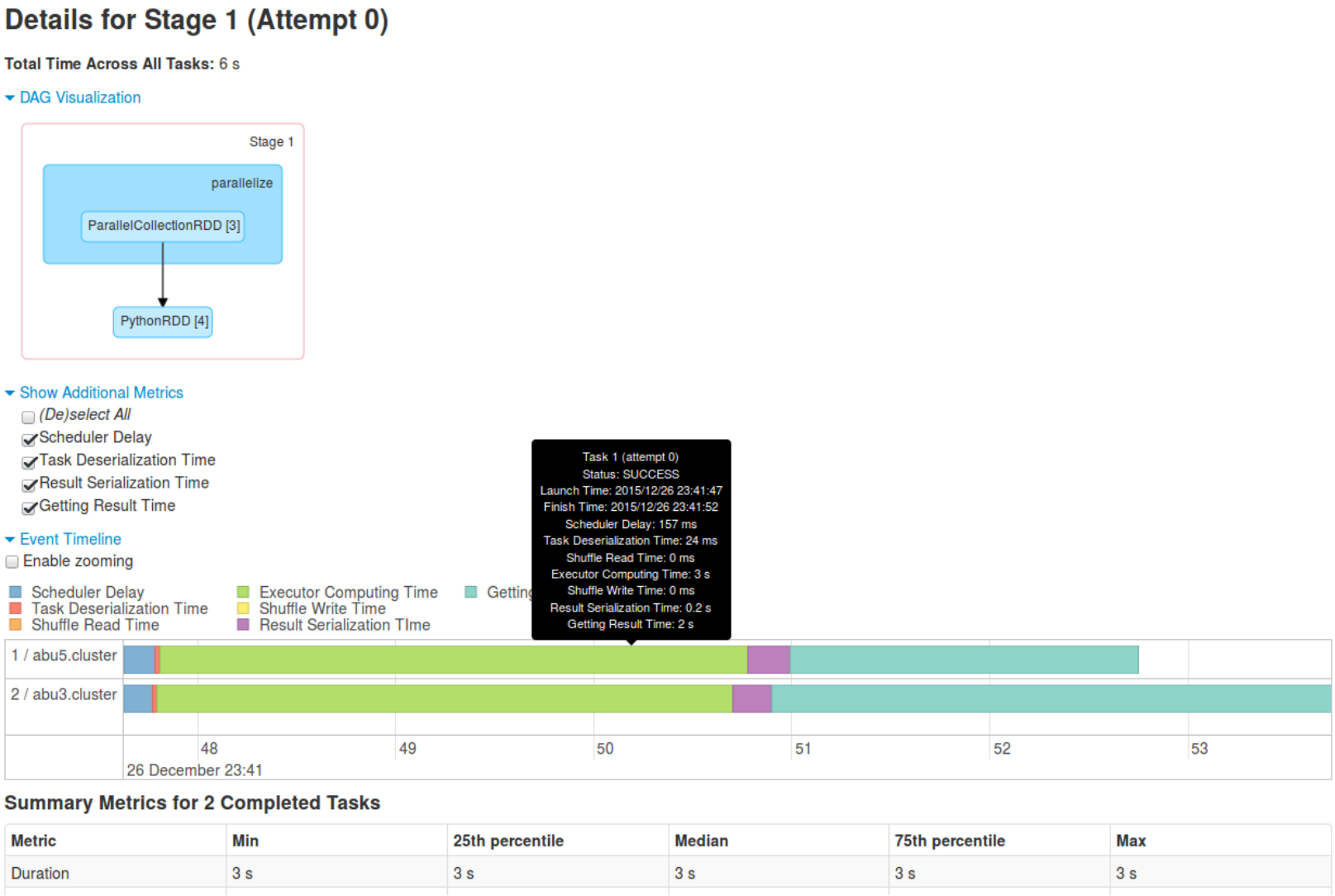
Web UI: Jobs



Completed Jobs (2)

Job Id	Description	Submitted	Duration	Stages: Succeeded/Total	Tasks (for all stages): Succeeded/Total
1	collect at <stdin>:3	2015/12/26 23:41:47	6 s	1/1	2/2
0	collect at <stdin>:3	2015/12/26 23:39:34	1 s	1/1	2/2

Web UI: Stages



Web UI: Stages' Metrics

Summary Metrics for 2 Completed Tasks

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	3 s	3 s	3 s	3 s	3 s
Scheduler Delay	0 ms	0 ms	0 ms	0 ms	0 ms
Task Deserialization Time	22 ms	22 ms	24 ms	24 ms	24 ms
GC Time	71 ms	71 ms	74 ms	74 ms	74 ms
Result Serialization Time	0.2 s	0.2 s	0.2 s	0.2 s	0.2 s
Getting Result Time	2 s	2 s	3 s	3 s	3 s

Aggregated Metrics by Executor

Executor ID	Address	Task Time	Total Tasks	Failed Tasks	Succeeded Tasks
1	abu5.cluster:56484	5 s	1	0	1
2	abu3.cluster:34220	6 s	1	0	1

Tasks

Index	ID	Attempt	Status	Locality Level	Executor ID / Host	Launch Time	Duration	Scheduler Delay	Task Deserialization Time	GC Time	Result Serialization Time	Getting Result Time	Errors
0	2	0	SUCCESS	PROCESS_LOCAL	2 / abu3.cluster	2015/12/26 23:41:47	3 s	0 ms	22 ms	71 ms	0.2 s	3 s	
1	3	0	SUCCESS	PROCESS_LOCAL	1 / abu5.cluster	2015/12/26 23:41:47	3 s	0 ms	24 ms	74 ms	0.2 s	2 s	

Web UI: Storage

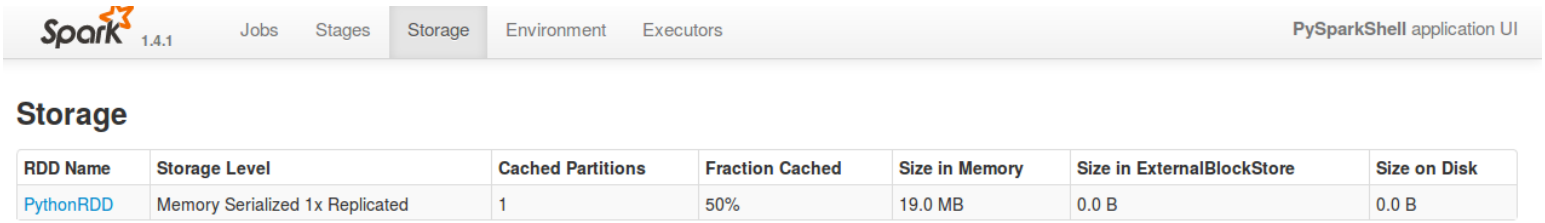


Figure: Overview

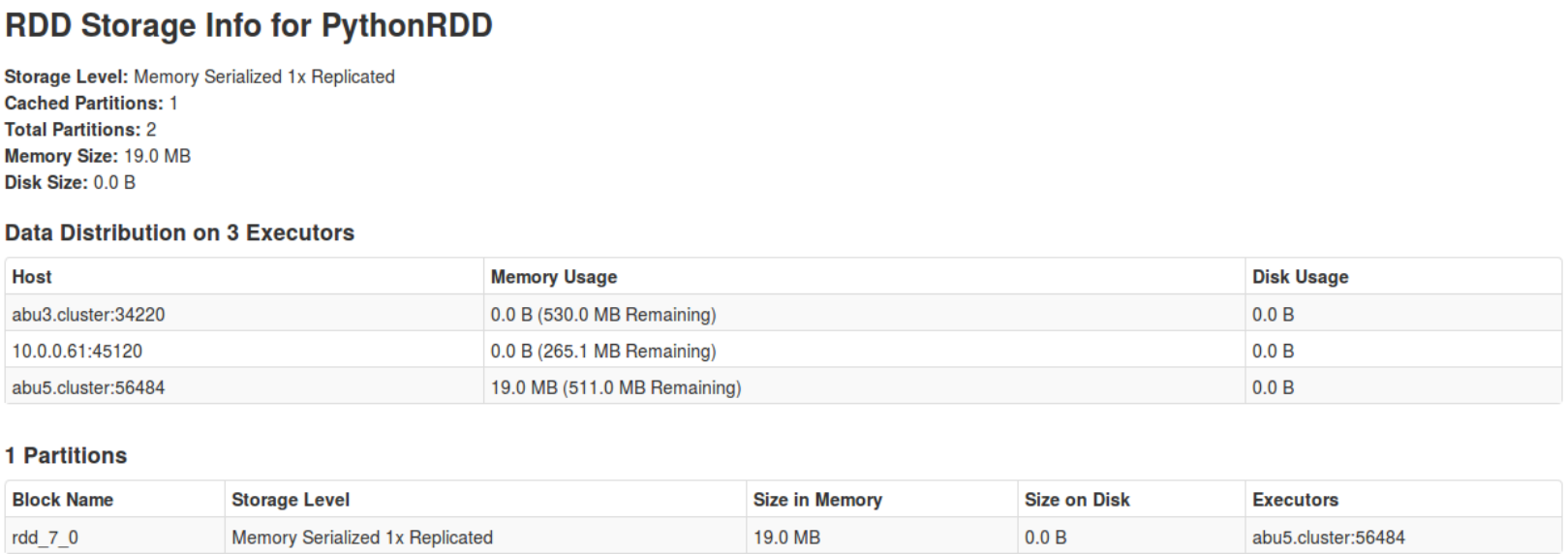
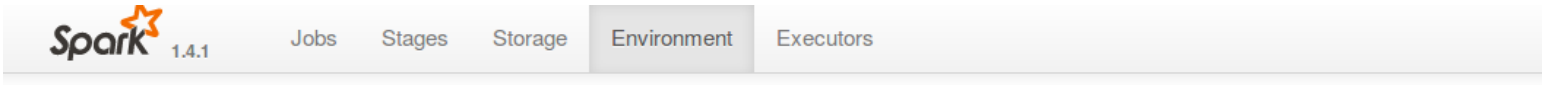


Figure: RDD details

Web UI: Environment Variables



Environment

Runtime Information

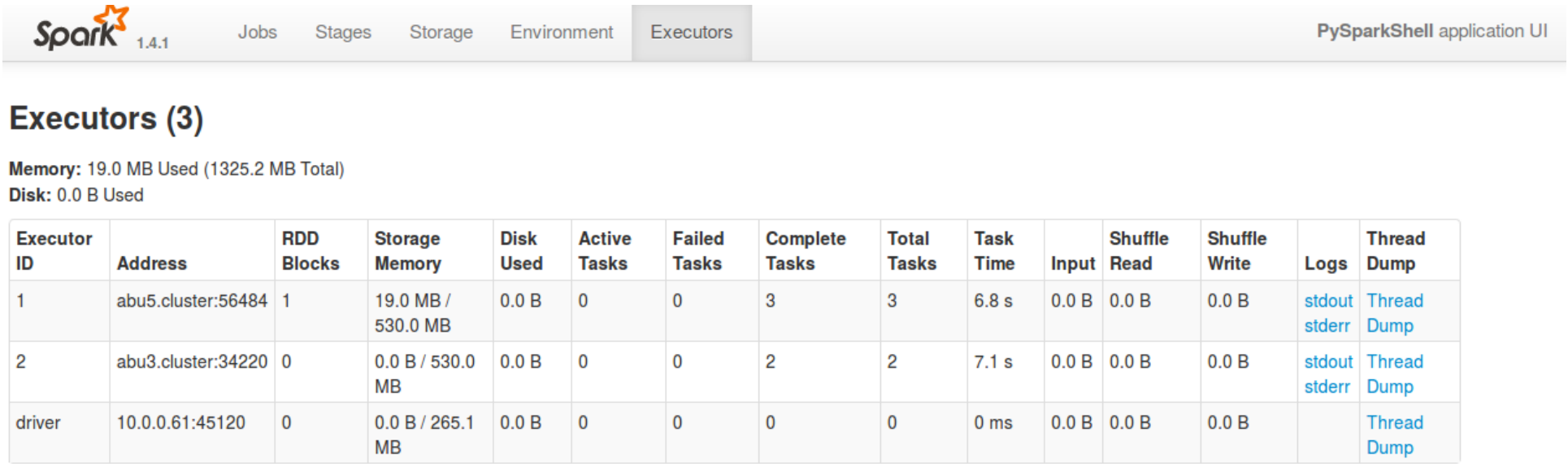
Name	Value
Java Home	/usr/jdk64/jdk1.8.0_40/jre
Java Version	1.8.0_40 (Oracle Corporation)
Scala Version	version 2.10.4

Spark Properties

Name	Value
spark.app.id	local-1448289108772
spark.app.name	Spark shell
spark.driver.extraJavaOptions	-Dhdp.version=2.3.2.0-2950
spark.driver.host	10.0.0.61
spark.driver.port	45900
spark.executor.id	driver
spark.externalBlockStore.folderName	spark-160232fa-3a9f-447e-a1ca-7a4bea06efc1
spark.fileserver.uri	http://10.0.0.61:50693
spark.history.kerberos.keytab	none
spark.history.kerberos.principal	none
spark.history.provider	org.apache.spark.deploy.yarn.history.YarnHistoryProvider
spark.history.ui.port	18080
spark.jars	
spark.master	local[*]
spark.repl.class.uri	http://10.0.0.61:36514
spark.scheduler.mode	FIFO

WARNING we run in local mode not cluster (or yarn)!

Web UI: Executors



Code Examples for our Student/Lecture Data

Preparing the data and some simple operations

```
1  from datetime import datetime
2  # Goal load student data from our CSV, we'll use primitive parsing that cannot handle escaped text
3  # We are just using tuples here without schema. split() returns an array
4  s = sc.textFile("stud.csv").map(lambda line: line.split(",")).filter(lambda line: len(line)>1)
5  l = sc.textFile("lecture.csv").map(lambda line: line.split(";")).filter(lambda line: len(line)>1)
6  print(l.take(10))
7  # [[u'1', u'"Big Data"', u'{(22),(23)}'], [u'2', u'"Hochleistungsrechnen"', u'{(22)}']]
8  l.saveAsTextFile("output.csv") # returns a directory with each partition
9
10 # Now convert lines into tuples, create lectures with a set of attending students
11 l = l.map( lambda t: ((int) (t[0]), t[1], eval(t[2])) ) # eval interprets the text as python code
12 # [(1, u'"Big Data"', set([22, 23])), (2, u'"Hochleistungsrechnen"', set([22]))]
13
14 # Convert students into proper data types
15 s = s.map( lambda t: ((int) (t[0]), t[1], t[2], t[3].upper() == "TRUE", datetime.strptime(t[4], "%Y-%m-%d") ) )
16 # (22, u'"Fritz"', u'"Musterman"', False, datetime.datetime(2000, 1, 1, 0, 0))...
17
18 # Identify how the rows are distributed
19 print(s.map(lambda x: x[0]).glom().collect())
20 # [[22], [23]] => each student is stored in its own partition
21
22 # Stream all tokens as text through cat, each token is input separately
23 m = l.pipe("/bin/cat")
24 # ['(1, u\'"Big Data"\', set([22, 23]))', '(2, u\'"Hochleistungsrechnen"\', set([22]))']
25
26 # Create a key/value RDD
27 # Student ID to data
28 skv = s.map(lambda l: (l[0], (l[1],l[2],l[3],l[4])))
29 # Lecture ID to data
30 lkv = l.map(lambda l: (l[0], (l[1], l[2])))
```

Code Examples for our Student/Lecture Data

- Was the code on the slide before a bit hard to read?
- Better to document tuple format input/output or use pipe diagrams!

Goal: Identify all lectures a student attends (now with comments)

```
1  # s = [(id, firstname, lastname, female, birth), ...]
2  # l = [(id,  name, [attendee student id]), ...]
3  sl = l.flatMap(lambda l: [ (s, l[0]) for s in l[2] ]  ) # can return 0 or more tuples
4  # sl = [ (student id, lecture id) ] = [(22, 1), (23, 1), (22, 2)]
5  # sl is now a key/value RDD.
6
7  # Find all lectures a student attends
8  lsa = sl.groupByKey() # lsa = [ (student id, [lecture id] ) ]
9
10 # print student and attending lectures
11 for (stud, attends) in lsa.collect():
12   print("%d : %s" %(stud, [ str(a) for a in attends ] ))
13 # 22 : ['1', '2']
14 # 23 : ['1']
15
16 # Use a join by the key to identify the students' data
17 j = lsa.join(skv)  # join (student id, [lecture id]) with [(id), (firstname, lastname, female, birth)), ...]
18 for (stud, (attends, studdata)) in j.collect():
19   print("%d: %s %s : %s" %(stud, studdata[0], studdata[1], [ str(a) for a in attends ] ))
20 22: "Fritz" "Musterman" : ['1', '2']
21 23: "Nina" "Musterfrau" : ['1']
```

Code Examples for our Student/Lecture Data

Compute the average age of students

```
1  # Approach: initialize a tuple with (age, 1) and reduce it (age1, count1) + (age2, age2) = (age1+age2, count1+count2)
2  cur = datetime.now()
3  # We again combine multiple operations in one line
4  # The transformations are executed when calling reduce
5  age = s.map( lambda x: ( (cur - x[4]).days, 1) ).reduce( lambda x, y: (x[0]+y[0], x[1]+y[1]) )
6  print(age)
7  # (11478, 2) => total age in days of all people, 2 people
8
9  # Alternative via a shared variable
10 ageSum = sc.accumulator(0)
11 peopleSum = sc.accumulator(0)
12
13 # We define a function to manipulate the shared variable
14 def increment(age):
15   ageSum.add(age)
16   peopleSum.add(1)
17
18 # Determine age, then apply the function to manipulate shared vars
19 s.map( lambda x: (cur - x[4]).days ).foreach( increment )
20 print("(%s, %s): avg %.2f" % (ageSum, peopleSum, ageSum.value/365.0/peopleSum.value))
21 # (11480, 2): avg 15.73
```

Higher-Level Abstractions

- Spark SQL: access relational tables
  - Access JDBC, hive tables or temporary tables
  - Limitations: No UPDATE statements, INSERT only for Parquet files
- GraphX: graph processing [15] (no Python API so far)
- Spark Streaming [16]
  - Discretized streams accept data from sources
    - TCP stream, file, Kafka, Flume, Kinesis, Twitter
  - Some support for executing SQL and MLlib on streaming data
- MLlib: Machine Learning Library
  - Provides efficient algorithms for several fields

SQL Overview [14]

- New data structures: SchemaRDD representing a table with rows
- RDDs can be converted to tables
  - Either create a schema manually or infer the data types
- Tables are file-backed and integrated into Hive
  - Self-describing Parquet files also store the schema
  - Use cacheTable(NAME) to load a table into memory
- Thrift JDBC server (similar to Hives JDBC)
- SQL-DSL: Language-Integrated queries
- Access via HiveContext (earlier SQLContext) class
  - HiveContext provides
    - Better Hive compatible SQL than SQLContext
    - User-defined functions (UDFs)
  - There are some (annoying) restrictions to HiveQL

Creating an In-memory Table from RDD

```
1  # Create a table from an array using the column names value, key
2  # The data types of the columns are automatically inferred
3  r = sqlContext.createDataFrame([('test', 10), ('data', 11)], ["value", "key"])
4
5  # Alternative: create/use an RDD
6  rdd = sc.parallelize(range(1,10)).map(lambda x : (x, str(x)) )
7
8  # Create the table from the RDD using the columnnames given, here "value" and "key"
9  schema = sqlContext.createDataFrame(rdd, ["value", "key"])
10 schema.printSchema()
11
12 # Register table for use with SQL, we use a temporary table, so the table is NOT visible
       ↪ in Hive
13 schema.registerTempTable("nums")
14
15 # Now you can run SQL queries
16 res = sqlContext.sql("SELECT * from nums")
17
18 # res is an DataFrame that uses columns according to the schema
19 print( res.collect() )
20 # [Row(num=1, str='1'), Row(num=2, str='2'), ... ]
21
22 # Save results as a table for Hive
23 from pyspark.sql import DataFrameWriter
24 dw = DataFrameWriter(res)
25 dw.saveAsTable("data")
```

Manage Hive Tables via SQL

```
1  # When using an SQL statement to create the table, the table is visible in HCatalog!
2  p = sqlContext.sql("CREATE TABLE IF NOT EXISTS data (key INT, value STRING)")
3
4  # Bulk loading data by appending it to the table data (if it existed)
5  sqlContext.sql("LOAD DATA LOCAL INPATH 'data.txt' INTO TABLE data")
6
7  # The result of a SQL query is a DataFrame, an RDD of rows
8  rdd = sqlContext.sql("SELECT * from data")
9
10 # Tread rdd as a SchemaRDD, access row members using the column name
11 o = rdd.map(lambda x: x.key) # Access the column by name, here "key"
12 # To print the distributed values they have to be collected.
13 print(o.collect())
14
15 sqlContext.cacheTable("data") # Cache the table in memory
16
17 # Save as Text file/directory into the local file system
18 dw.json("data.json", mode="overwrite")
19 # e.g. {"key":10,"value":"test"}
20
21 sqlContext.sql("DROP TABLE data") # Remove the table
```

Language integrated DSL

- Methods to formulate SQL queries
  - See `help(pyspark.sql.dataframe.DataFrame)` for details
- Applies lazy evaluation

```
1  from pyspark.sql import functions as F
2
3  # Run a select query and visualize the results
4  rdd.select(rdd.key, rdd.value).show()
5  # +---+-------------+
6  # |key|        value|
7  # +---+-------------+
8  # | 10|         test|
9  # | 11|         data|
10 # | 12|        fritz|
11 # +---+-------------+
12
13 # Return the rows where value == 'test'
14 rdd.where(rdd.value == 'test').collect()
15
16 # Aggregate/Reduce values by the key
17 rdd.groupBy().avg().collect() # average(key)=11
18 # Similar call, short for groupBy().agg()
19 rdd.agg({"key": "avg"}).collect()
20 # Identical result for the aggregation with different notation
21 rdd.agg(F.avg(rdd.key)).collect()
```

Code Examples for our Student/Lecture Data

Convert the student RDDs to a Hive table and perform queries

```
1  from pyspark.sql import HiveContext, Row
2
3  sqlContext = HiveContext(sc)
4  # Manually convert lines to a Row (could be done automatically)
5  sdf = s.map(lambda l: Row(matrikel=l[0], firstname=l[1], lastname=l[2], female=l[3], birthday=l[4]))
6  # infer the schema and create a table (SchemaRDD) from the data (inferSchema is deprecated but shows the idea)
7  schemaStudents = sqlContext.inferSchema(sdf)
8  schemaStudents.printSchema()
9  # birthday: timestamp (nullable = true), female: boolean (nullable = true), ...
10 schemaStudents.registerTempTable("student")
11
12 females = sqlContext.sql("SELECT firstname FROM student WHERE female == TRUE")
13 print(females.collect()) # print data
14 # [Row(firstname=u'"Nina"')]
15
16 ldf = l.map(lambda l: Row(id=l[0], name=l[1]))
17 schemaLecture = sqlContext.inferSchema(ldf)
18 schemaLecture.registerTempTable("lectures")
19
20 # Create student-lecture relation
21 slr = l.flatMap(lambda l: [ Row(lid=l[0], matrikel=s) for s in l[2] ] )
22 schemaStudLec = sqlContext.inferSchema(slr)
23 schemaStudLec.registerTempTable("studlec")
24
25 # Print student name and all attended lectures' names, collect_set() bags grouped items together
26 sat = sqlContext.sql("SELECT s.firstname, s.lastname, s.matrikel, collect_set(l.name) as lecs FROM studlec sl JOIN student s
       ↪ ON sl.matrikel=s.matrikel JOIN lectures l ON sl.lid=l.id GROUP BY s.firstname, s.lastname, s.matrikel ")
27 print(sat.collect()) # [Row(firstname=u'"Nina"', lastname=u'"Musterfrau F."', matrikel=23, lecs=[u'"Big Data"']),
       ↪ Row(firstname=u'"Fritz"', lastname=u'"Musterman M."', matrikel=22, lecs=[u'"Big Data"',
       ↪ u'"Hochleistungsrechnen"'])]
```

Code Examples for our Student/Lecture Data

Storing tables as Parquet files

```
# Saved dataFrame as Parquet files keeping schema information.
# Note: DateTime is not supported, yet
schemaLecture.saveAsParquetFile("lecture-parquet")

# Read in the Parquet file created above.  Parquet files are self-describing so the
    ↪ schema is preserved.
# The result of loading a parquet file is also a DataFrame.
lectureFromFile = sqlContext.parquetFile("lecture-parquet")
# Register Parquet file as lFromFile
lectureFromFile.registerTempTable("lFromFile");

# Now it supports bulk insert (we insert again all lectures)
sqlContext.sql("INSERT INTO TABLE lFromFile SELECT * from lectures")
# Not supported INSERT: sqlContext.sql("INSERT INTO lFromFile VALUES(3, 'Neue
    ↪ Vorlesung', {()})")
```

Dealing with JSON Files

Table (SchemaRDD) rows' can be converted to/from JSON

```
1  # store each row as JSON
2  schemaLecture.toJSON().saveAsTextFile("lecture-json")
3  # load JSON
4  ljson = sqlContext.jsonFile("lecture-json")
5  # now register JSON as table
6  ljson.registerTempTable("ljson")
7  # perform SQL queries
8  sqlContext.sql("SELECT * FROM ljson").collect()
9
10 # Create lectures from a JSON snippet with one column as semi-structured JSON
11 lectureNew = sc.parallelize(['{"id":4,"name":"New lecture", "otherInfo":{"url":"http://xy", "mailingList":"xy", "lecturer":
       ↪ ["p1", "p2", "p3"]}}', '{"id":5,"name":"New lecture 2", "otherInfo":{}}'])
12 lNewSchema = sqlContext.jsonRDD(lectureNew)
13 lNewSchema.registerTempTable("lnew")
14
15 # Spark natively understands nested JSON fields and can access them
16 sqlContext.sql("SELECT otherInfo.mailingList FROM lnew").collect()
17 # [Row(mailingList=u'xy'), Row(mailingList=None)]
18 sqlContext.sql("SELECT otherInfo.lecturer[2] FROM lnew").collect()
19 # [Row(_c0=u'p3'), Row(_c0=None)]
```

Spark Streaming [16]

- Alternative to Storm
  - Data model: Continuous stream of RDDs (batches of tuples)
  - Fault tolerance: uses checkpointing of states
- Not all data can be accessed at a given time
  - Only data from one interval or a sliding window
  - States can be kept for key/value RDDs using updateStateByKey()
- Not all transformation and operations available, e.g. foreach, collect
  - Streams can be combined with existing RDDs using transform()
- Workflow: Build the pipeline, then start it
- Can read streams from multiple sources
  - Files, TCP sources, ...
- Note: Number of tasks assigned $>$ than receivers, otherwise it stagnates

Figure: Source: [16]

Processing of Streams

Basic processing concept is the same as for RDDs, example:

```
1 words = lines.flatMap(lambda l: l.split(" "))
```

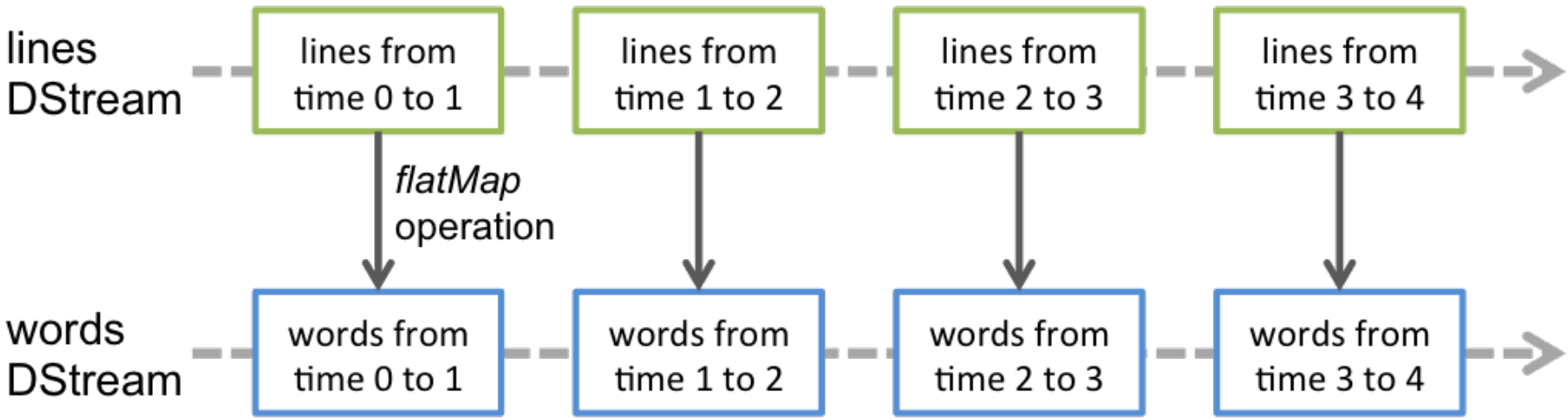


Figure: Source: [16]

Window-Based Operations

```
# Reduce a window of 30 seconds of data every 10 seconds
rdd = words.reduceByKeyAndWindow(lambda x, y: x + y, 30, 10)
```

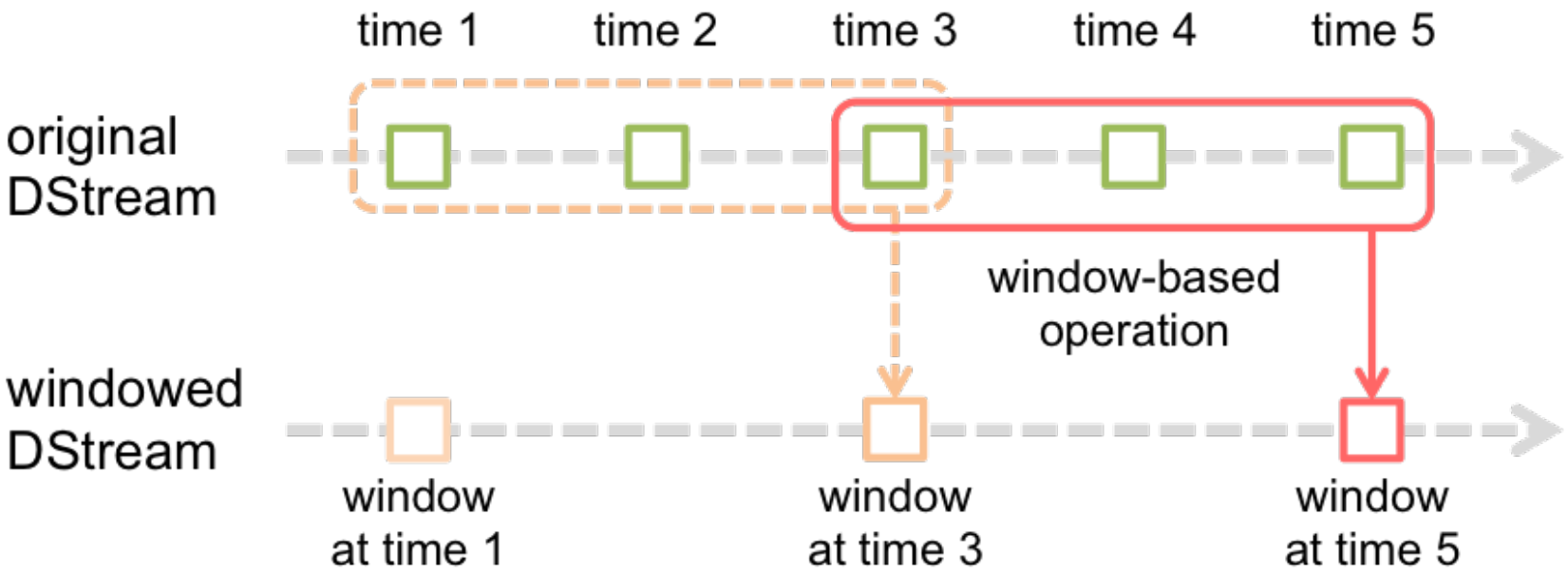


Figure: Source: [16]

Example Streaming Application

```
1  from pyspark.streaming import StreamingContext
2  # Create batches every second
3  ssc = StreamingContext(sc, batchDuration=1)
4  ssc.checkpoint("mySparkCP")
5  # We should use ssc.getOrCreate() to allow restoring the stored
       ↪ checkpoint, see [16]
6
7  # Create a stream from a TCP socket
8  lines = ssc.socketTextStream("localhost", 9999)
9
10 # Alternatively: read newly created files in the directory and process them
11 # Move files into this directory to start computation
12 # lines = scc.textFileStream("myDir")
13
14 # Split lines into tokens and return tuples (word,1)
15 words = lines.flatMap(lambda l: l.split(" ")).map( lambda x: (x,1) )
16
17 # Track the count for each key (word)
18 def updateWC(val, stateVal):
19     if stateVal is None:
20        stateVal = 0
21     return sum(val, stateVal)
22
23 counts = words.updateStateByKey(updateWC) # Requires checkpointing
24
25 # Print the first 10 tokens of each stream RDD
26 counts.pprint(num=10)
27
28 # start computation, after that we cannot change the processing pipeline
29 ssc.start()
30 # Wait until computation finishes
31 ssc.awaitTermination()
32 # Terminate computation
33 ssc.stop()
```

Example output

Started TCP server

`nc -lk4 localhost 9999`

```
Input: das ist ein test
Output:
Time: 2015-12-27 15:09:43
-------------------------
('das', 1)
('test', 1)
('ein', 1)
('ist', 1)

Input: das ist ein haus
Output:
Time: 2015-12-27 15:09:52
-------------------------
('das', 2)
('test', 1)
('ein', 2)
('ist', 2)
('haus', 1)
```

MLlib: Machine Learning Library [22]

- Provides many useful algorithms, some in streaming versions
- Supports many existing data types from other packages
  - Supports Numpy, SciPy, MLlib

Subset of provided algorithms

- Statistics
  - Descriptive statistics, hypothesis testing, random data generation
- Classification and regression
  - Linear models, Decision trees, Naive Bayes
- Clustering
  - k-means
- Frequent pattern mining
  - Association rules
- Higher-level APIs for complex pipelines
  - Feature extraction, transformation and selection
  - Classification and regression trees
  - Multilayer perceptron classifier

Descriptive Statistics [22]

```
1  from pyspark.mllib.stat import Statistics as s
2  import math
3
4  # Create RDD with 4 columns
5  rdd = sc.parallelize( range(1,100) ).map( lambda x : [x, math.sin(x), x*x, x/100] )
6  sum = s.colStats(rdd) # determine column statistics
7  print(sum.mean()) # [  5.00e+01   3.83024876e-03   3.31666667e+03   5.00e-01]
8  print(sum.variance()) # [  8.25e+02   5.10311520e-01   8.788835e+06   8.25e-02]
9
10 x = sc.parallelize( range(1,100) ) # create a simple data set
11 y = x.map( lambda x: x / 10 + 0.5)
12 # Determine Pearson correlation
13 print(s.corr(x, y, method="pearson")) # Correlation 1.0000000000000002
14
15 # Create a random RDD with 100000 elements
16 from pyspark.mllib.random import RandomRDDs
17 u = RandomRDDs.uniformRDD(sc, 1000000)
18
19 # Estimate kernel density
20 from pyspark.mllib.stat import KernelDensity
21 kd = KernelDensity()
22 kd.setSample(u)
23 kd.setBandwidth(1.0)
24 # Estimate density for the given values
25 densities = kd.estimate( [0.2, 0, 4] )
```

Linear Models [23]

```
1  from pyspark.mllib.regression import LabeledPoint, LinearRegressionWithSGD, LinearRegressionModel
2  import random
3
4  # Three features (x, y, label = x + 2*y + small random Value)
5  x = [ random.uniform(1,100) for y in range(1, 10000)]
6  x.sort()
7  y = [ random.uniform(1,100) for y in range(1, 10000)]
8  # LabeledPoint identifies the result variable
9  raw = [ LabeledPoint(i+j+random.gauss(0,4), [i/100, j/200])  for (i,j) in zip(x, y) ]
10 data = sc.parallelize( raw )
11
12 # Build the model using maximum of 10 iterations with stochastic gradient descent
13 model = LinearRegressionWithSGD.train(data, 100)
14
15 print(model.intercept)
16 # 0.0
17 print(model.weights)
18 #[110.908004953,188.96464824] => we except [100, 200]
19
20 # Validate the model with the original training data
21 vp = data.map(lambda p: (p.label, model.predict(p.features)))
22
23 # Error metrics
24 abserror = vp.map(lambda p: abs(p[0] - p[1])).reduce(lambda x, y: x + y) / vp.count()
25 error = vp.map(lambda p: abs(p[0] - p[1]) / p[0]).reduce(lambda x, y: x + y) / vp.count()
26 MSE = vp.map(lambda p: (p[0] - p[1])**2).reduce(lambda x, y: x + y) / vp.count()
27
28 print("Abs error: %.2f" % (abserror)) # 4.41
29 print("Rel. error: %.2f%%" % (error * 100)) # 5.53%
30 print("Mean Squared Error: %.2f" % (MSE))
31
32 # Save / load the model
33 model.save(sc, "myModelPath")
34 model = LinearRegressionModel.load(sc, "myModelPath")
```

Clustering [25]

```
1  # Clustering with k-means is very simple for N-Dimensional data
2  from pyspark.mllib.clustering import KMeans, KMeansModel
3  import random as r
4  from numpy import array
5
6  # Create 3 clusters in 2D at (10,10), (50,30) and (70,70)
7  x = [ [r.gauss(10,4), r.gauss(10,2)] for y in range(1, 100) ]
8  x.extend( [r.gauss(50,5), r.gauss(30,3)] for y in range(1, 900) )
9  x.extend( [r.gauss(70,5), r.gauss(70,8)] for y in range(1, 500) )
10 x = [ array(x) for x in x]
11
12 data = sc.parallelize(x)
13
14 # Apply k-means
15 clusters = KMeans.train(data, 3, maxIterations=10, runs=10, initializationMode="random")
16
17 print(clusters.clusterCenters)
18 # [array([ 70.42953058,  69.88289475]),
19 #  array([ 10.57839294,   9.92010409]),
20 #  array([ 49.72193422,  30.15358142])]
21
22 # Save/load model
23 clusters.save(sc, "myModelPath")
24 sameModel = KMeansModel.load(sc, "myModelPath")
```

Decision Trees [25]

```
1  # Decision trees operate on tables and don't use LabeledPoint ...
2  # They offer the concept of a pipleline to preprocess data in RDD
3
4  from pyspark.mllib.linalg import Vectors
5  from pyspark.sql import Row
6  from pyspark.ml.classification import DecisionTreeClassifier
7  from pyspark.ml.feature import StringIndexer
8  from pyspark.ml import Pipeline
9  from pyspark.ml.evaluation import BinaryClassificationEvaluator
10 import random as r
11
12 # We create a new random dataset but now with some overlap
13 x = [ ("blue", [r.gauss(10,4), r.gauss(10,2)]) for y in range(1, 100) ]
14 x.extend( ("red", [r.gauss(50,5), r.gauss(30,3)]) for y in range(1, 900) )
15 x.extend( ("yellow", [r.gauss(70,15), r.gauss(70,25)]) for y in range(1, 500) ) # Class red and yellow may overlap
16
17 data = sc.parallelize(x).map(lambda x: (x[0], Vectors.dense(x[1])))
18 # The data frame is expected to contain exactly the specified two columns
19 dataset = sqlContext.createDataFrame(data, ["label", "features"])
20
21 # Create a numeric index from string label categories, this is mandatory!
22 labelIndexer = StringIndexer(inputCol="label", outputCol="indexedLabel").fit(dataset)
23
24 # Our decision tree
25 dt = DecisionTreeClassifier(featuresCol='features', labelCol='indexedLabel', predictionCol='prediction', maxDepth=5)
26
27 # Split data into 70% training set and 30% validation set
28 (trainingData, validationData) = dataset.randomSplit([0.7, 0.3])
29
30 # Create a pipeline which processes dataframes and run it to create the model
31 pipeline = Pipeline(stages=[labelIndexer, dt])
32 model = pipeline.fit(trainingData)
```

Decision Trees – Validation [25]

```
# Perform the validation on our validation data
predictions = model.transform(validationData)

# Pick some rows to display.
predictions.select("prediction", "indexedLabel", "features").show(2)
# +----------+------------+--------------------+
# |prediction|indexedLabel|            features|
# +----------+------------+--------------------+
# |       2.0|         2.0|[11.4688967071571...|
# |       2.0|         2.0|[10.8286615821145...|
# +----------+------------+--------------------+

# Compute confusion matrix using inline SQL
predictions.select("prediction", "indexedLabel").groupBy(["prediction",
    ↪ "indexedLabel"]).count().show()
# +----------+------------+-----+
# |prediction|indexedLabel|count|
# +----------+------------+-----+
# |       2.0|         2.0|   69| <= correct
# |       1.0|         1.0|  343| <= correct
# |       0.0|         0.0|  615| <= correct
# |       0.0|         1.0|   12| <= too much overlap, thus wrong
# |       1.0|         0.0|    5| <= too much overlap, thus wrong
# +----------+------------+-----+

# There are also classes for performing automatic validation
```

Summary

- Spark is an in-memory processing and storage engine
  - It is based on the concept of RDDs
  - An RDD is an immutable list of tuples (or a key/value tuple)
  - Computation is programmed by transforming RDDs
- Data is distributed by partitioning an RDD
  - Computation of transformations is done on local partitions
  - Shuffle operations change the mapping and require communication
  - Actions return data to the driver or perform I/O
- Fault-tolerance is provided by re-computing partitions
- Driver program controls the executors and provides code closures
- Lazy evaluation: All computation is deferred until needed by actions
- Higher-level APIs enable SQL, streaming and machine learning
- Interactions with the Hadoop ecosystem
  - Accessing HDFS data
  - Sharing tables with Hive
  - Can use YARN resource management

Bibliography

10 Wikipedia

11 HCatalog InputFormat https://gist.github.com/granturing/7201912

12 http://spark.apache.org/docs/latest/cluster-overview.html

13 http://spark.apache.org/docs/latest/programming-guide.html

14 http://spark.apache.org/docs/latest/api/python/pyspark.sql.html

15 http://spark.apache.org/docs/latest/graphx-programming-guide.html

16 http://spark.apache.org/docs/latest/streaming-programming-guide.html

17 https://databricks.com/blog/2015/04/13/deep-dive-into-spark-sqls-catalyst-optimizer.html

18 http://www.cloudera.com/content/www/en-us/documentation/enterprise/latest/topics/cdh_ig_running_spark_on_yarn.html

19 http://spark.apache.org/docs/latest/running-on-yarn.html

20 http://spark.apache.org/examples.html

21 http://blog.cloudera.com/blog/2015/03/how-to-tune-your-apache-spark-jobs-part-1/

22 http://spark.apache.org/docs/latest/mllib-guide.html

23 http://spark.apache.org/docs/latest/mllib-linear-methods.html

24 http://spark.apache.org/docs/latest/quick-start.html#self-contained-applications

25 http://spark.apache.org/docs/latest/mllib-clustering.html