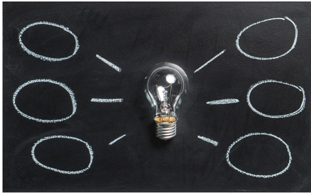
GEORG-AUGUST-UNIVERSITÄT
GÖTTINGEN

Julian Kunkel

In-Memory Computation



2021-12-06

Outline

Learning Objectives

- Define in-memory processing
- Describe the basic data model of Apache Spark and the SQL extension
- Program a simple data flow algorithm using Spark RDDs
- Sketch the architecture of Spark and the roles of its components
- Describe the execution of a simple program on the Spark architecture

In-Memory Computation/Processing/Analytics [26]

- **In-memory processing**: Processing of data stored in memory (database)
  - Optimally: near-interactive response regardless of volume
- Advantage: No slow I/O necessary $\Rightarrow$ fast response times
- Disadvantages
  - Data must fit in the memory of the distributed storage/database
  - Additional persistency (with asynchronous flushing) usually required
  - Fault-tolerance is mandatory
- BI-Solution: SAP Hana
- Big data approaches: Apache Spark, Apache Flink

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
  - ⇒ It can use Hadoop but also works standalone!
- Task scheduling and monitoring
- 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
  - `sparkR`: R (basic support)
- Execution in either local (single node) 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)
  - An RDD is immutatable, e.g., cannot be changed
  - High-level APIs provide additional representations
    - e.g., SparkSQL uses DataFrames (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 (if an error occurs)

Resilient Distributed Datasets (RDDs) [13]

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

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

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

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

- RDDs can be transformed into derived (newly named) RDDs

```
rdd2 = rdd.filter( lambda x : (x % 2 == 0) ) # operation: filter odd tuples
```

  - RDDs can be redistributed (called shuffle)
  - RDD is computed if needed, but RDD can be cached in memory or stored
  - Computation runs in parallel on the partitions
  - RDD 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): transfer to all executors
  - For readability, do not modify the broadcast variable later

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

- Accumulators (reduce variables): 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())
```

Outline

Execution of Applications [12, 21]

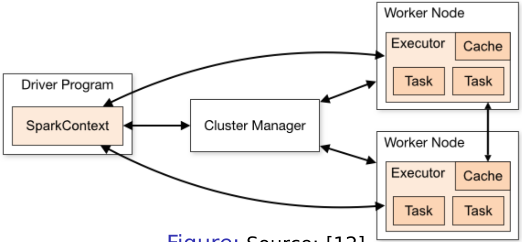


Figure: Source: [12]

- 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 mode, 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

Persistence [13]

Concepts

- The data lineage of an RDD is stored
- **Actions** trigger computation, no intermediate results are kept
- The methods `cache()` and `persist()` enables preserving of results
  - The first time an 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

Parallelism [13]

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

```
# Create 10 partitions when the data is distributed
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

```
  # RDD values are 1, 2, 4, 5
X.glom().collect()
# [[], [ 1, 4 ], [ ], [ 5 ], [ 2 ] ] => here we have 5 partitions for RDD X
```

Outline

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 (and all members) are serialized
- RDD partitions are processed in parallel (data parallelism)
  - Concept: Use local data where possible

RDD Operation Types [13]

- **Transformations**: create a new RDD locally by applying operations
- **Actions**: return values to the driver program (or do I/O)
- **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

```python
# 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) )
# Store results in memory, cached at first invocation of an action
resultCached = result.cache()

# 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), ... ]
```

Compute PI [20]

Approach: Randomly throw NUM_SAMPLES darts on a circle and count hits

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

```
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);
```

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 all partitions through a program

Remember: Transformations return a lazy reference to a new dataset

Transformations Create a New RDD [13]

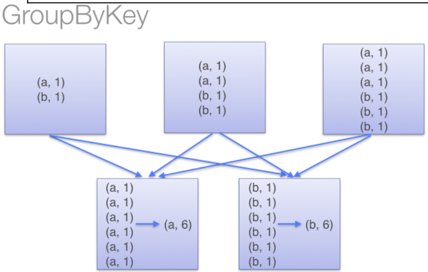
Key/Value RDDs additionally support

- 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 multiple values for keys (e.g., min, max, sum)
  - uses neutral element for initializer
- sortByKey([ascending], [numTasks]): order the dataset
- join(otherDataset, [numTasks]): pairs (K,V) elements with (K,U)
  - returns (K, (V,U))
- cogroup(otherDataset, [numTasks]): returns (K, iterableV, iterableU)

Illustrating processing of KV RDDs [30]

- GroupBy creates a new RDD shuffling all the data
- This can be inefficient

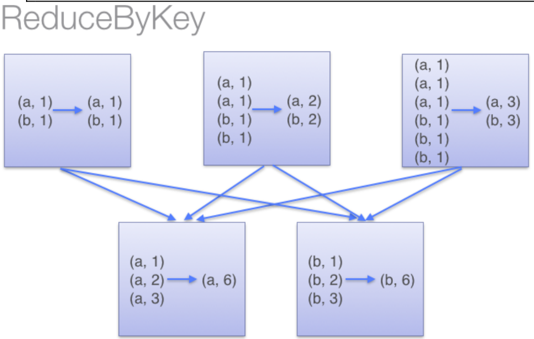
```
wordCountsWithGroup = wordPairsRDD.groupByKey().map(lambda x: (x[0], sum(x[1]))).collect()
def reduce(key, values): # same as above rewritten as Reduction function
  cnt = 0
  for v in values:
    cnt = cnt + v
  return (key, cnt)
wordCountsWithGroup = wordPairsRDD.groupByKey().map(lambda x: reduce(x[0],x[1])).collect()
```



Illustrating processing of KV RDDs [30]

- ReduceBy applies reduction function locally, creates new RDD and then globally
- Reduces network traffic, prefferable solution

```
words = ["one", "two", "two", "three", "three", "three"]
wordPairsRDD = sc.parallelize(words).map(lambda word : (word, 1))
wordCountsWithReduce = wordPairsRDD.reduceByKey(lambda a, b : a+b).collect()
# [('two', 2), ('three', 3), ('one', 1)]
```



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

- reduce(func): aggregates elements using: $func(x,y) \Rightarrow z$
  - 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; optionally: partition function

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()

Typical Mistakes [13]

Use local variables in distributed memory

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

# Wrong: since counter is a local variable, it is updated in each JVM
# Thus, each executor yields another result
rdd.foreach(lambda x: counter += x)
print("Counter value: " + counter)
```

Object serialization may be unexpected (and slow)

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

Writing to STDOUT/ERR on executors

```
# Call println() on each element but the executors' stdout is not redirected to the driver
rdd.foreach(println)
```

Outline

Code Examples for our Student/Lecture Data

Preparing the data and some simple operations

```python
from datetime import datetime
# Goal load student data from our CSV, we'll use primitive parsing that cannot handle escaped text
# We are just using tuples here without schema. split() returns an array
s = sc.textFile("stud.csv").map(lambda line: line.split(",")).filter(lambda line: len(line)>1)
l = sc.textFile("lecture.csv").map(lambda line: line.split(";")).filter(lambda line: len(line)>1)
print(l.take(10))
# [[u'1', u'"Big Data"', u'{(22),(23)}'], [u'2', u'"Hochleistungsrechnen"', u'{(22)}']]
l.saveAsTextFile("output.csv") # returns a directory with each partition

# Now convert lines into tuples, create lectures with a set of attending students
l = l.map( lambda t: ((int) (t[0]), t[1], eval(t[2])) ) # eval interprets the text as python code
# [(1, u'"Big Data"', set([22, 23])), (2, u'"Hochleistungsrechnen"', set([22]))]

# Convert students into proper data types
s = s.map( lambda t: ((int) (t[0]), t[1], t[2], t[3].upper() == "TRUE", datetime.strptime(t[4], "%Y-%m-%d") ) )
# (22, u'"Fritz"', u'"Musterman"', False, datetime.datetime(2000, 1, 1, 0, 0))...

# Identify how the rows are distributed
print(s.map(lambda x: x[0]).glom().collect())
# [[22], [23]] => each student is stored in its own partition

# Stream all tokens as text through cat, each token is input separately
m = l.pipe("/bin/cat")
# ['(1, u\'"Big Data"\', set([22, 23]))', '(2, u\'"Hochleistungsrechnen"\', set([22]))']

# Create a key/value RDD
# Student ID to data
skv = s.map(lambda l: (l[0], (l[1],l[2],l[3],l[4])))
# Lecture ID to data
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)

```
# s = [(id, firstname, lastname, female, birth), ...]
# l = [(id,  name, [attendee student id]), ...]
sl = l.flatMap(lambda l: [ (s, l[0]) for s in l[2] ]  ) # can return 0 or more tuples
# sl = [ (student id, lecture id) ] = [(22, 1), (23, 1), (22, 2)]
# sl is now a key/value RDD.

# Find all lectures a student attends
lsa = sl.groupByKey() # lsa = [ (student id, [lecture id] ) ]

# print student and attending lectures
for (stud, attends) in lsa.collect():
  print("%d : %s" %(stud, [ str(a) for a in attends ] ))
# 22 : ['1', '2']
# 23 : ['1']

# Use a join by the key to identify the students' data
j = lsa.join(skv)  # join (student id, [lecture id]) with [(id), (firstname, lastname, female, birth)), ...]
for (stud, (attends, studdata)) in j.collect():
  print("%d: %s %s : %s" %(stud, studdata[0], studdata[1], [ str(a) for a in attends ] ))
22: "Fritz" "Musterman" : ['1', '2']
23: "Nina" "Musterfrau" : ['1']
```

Code Examples for our Student/Lecture Data

Compute the average age of students

```
# Approach: initialize a tuple with (age, 1) and reduce it (age1, count1) + (age2, age2) = (age1+age2, count1+count2)
cur = datetime.now()
# We again combine multiple operations in one line
# The transformations are executed when calling reduce
age = s.map( lambda x: ( (cur - x[4]).days, 1) ).reduce( lambda x, y: (x[0]+y[0], x[1]+y[1]) )
print(age)
# (11478, 2) => total age in days of all people, 2 people

# Alternative via a shared variable
ageSum = sc.accumulator(0)
peopleSum = sc.accumulator(0)

# We define a function to manipulate the shared variable
def increment(age):
  ageSum.add(age)
  peopleSum.add(1)

# Determine age, then apply the function to manipulate shared vars
s.map( lambda x: (cur - x[4]).days ).foreach( increment )
print("(%s, %s): avg %.2f" % (ageSum, peopleSum, ageSum.value/365.0/peopleSum.value))
# (11480, 2): avg 15.73
```

Code Example for KeyValues

Compute min/max/sum at the same time

- Use aggregateByKey() create new tuple type from original tuple
  - Here, we start with (word, 1) tuples
  - We like to receive as output: (key, (min,max,sum))
- First, we aggregate in each partition the neutral element with each local element
- Then aggregate the new triple (min,max,sum) for all partitions together
- We note: the combiner function in Hadoop MapReduce was similar

```python
# assume wordPairsRDD is our set of tuples [(word, 1)]
wordPairsRDD.aggregateByKey((0,0,0),
  lambda x,y: (min(x[0],y), max(x[1],y), x[2]+y), # aggregate neutral + each element
  lambda x,y: (min(x[0],y[0]), max(x[1],y[1]), x[2]+y[2]) ) # aggregate across partitions
```

Group Work

- Sketch in pseudocode a program that processes the following data

```
u = RDD((user, timestamp, itemID, quantity))
i = RDD((itemID, price))
```

  - Calculate the total number of items sold yesterday
  - Harder task: calculate income made yesterday (and per itemID)
- Time: 10 min
- Organization: breakout groups - please use your mic or chat

Outline

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

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

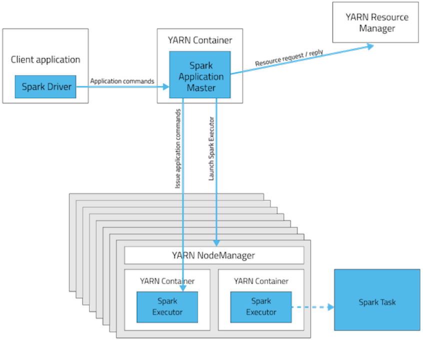


Figure: Client mode, Source: [18]

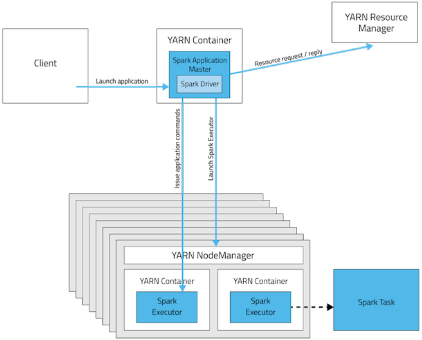


Figure: Cluster mode, Source: [18]

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
- Spark history-server keeps the data of previous jobs

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

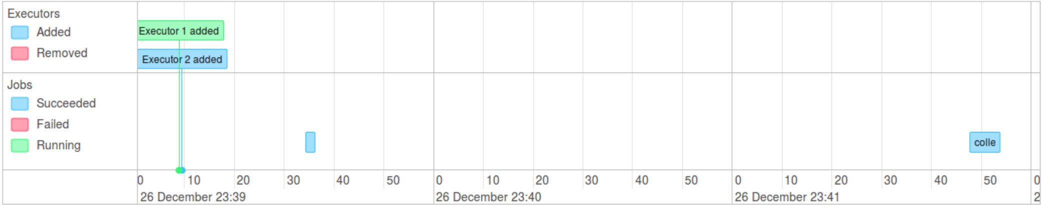
Web UI: Jobs

Spark 1.4.1 | Jobs | Stages | Storage | Environment | Executors | **PySparkShell** application UI

Spark Jobs (?)

Total Uptime: 4.3 min
Scheduling Mode: FIFO
Completed Jobs: 2

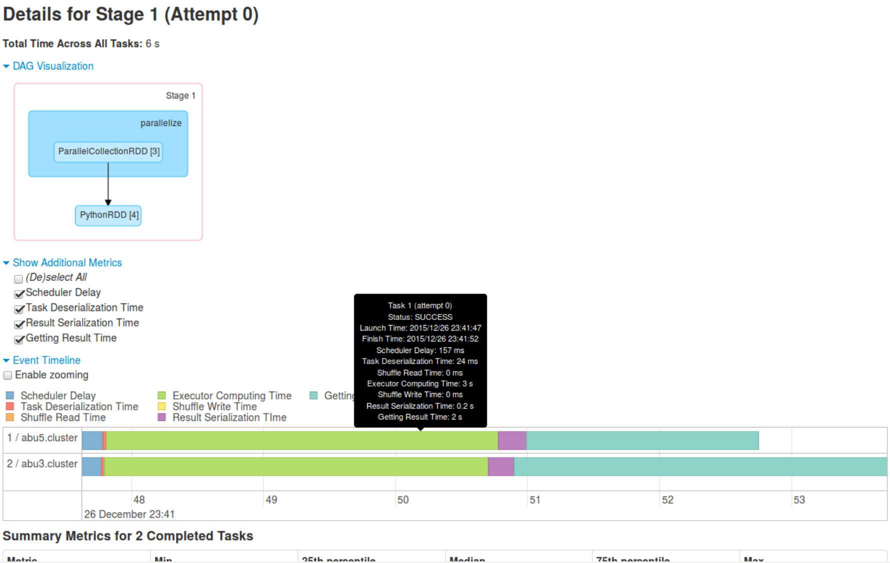
▾ Event Timeline
☐ Enable zooming



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

Spark 1.4.1 | Jobs | Stages | Storage | Environment | Executors | **PySparkShell** application UI

Storage

RDD Name	Storage Level	Cached Partitions	Fraction Cached	Size in Memory	Size in ExternalBlockStore	Size on Disk
PythonRDD	Memory Serialized 1x Replicated	1	50%	19.0 MB	0.0 B	0.0 B

Figure: Overview

RDD Storage Info for PythonRDD

Storage Level: Memory Serialized 1x Replicated
Cached Partitions: 1
Total Partitions: 2
Memory Size: 19.0 MB
Disk Size: 0.0 B

Data Distribution on 3 Executors

Host	Memory Usage	Disk Usage
abu3.cluster:34220	0.0 B (530.0 MB Remaining)	0.0 B
10.0.0.61:45120	0.0 B (265.1 MB Remaining)	0.0 B
abu5.cluster:56484	19.0 MB (511.0 MB Remaining)	0.0 B

1 Partitions

Block Name	Storage Level	Size in Memory	Size on Disk	Executors
rdd_7_0	Memory Serialized 1x Replicated	19.0 MB	0.0 B	abu5.cluster:56484

Figure: RDD details

Web UI: Environment Variables

Spark 1.4.1 Jobs Stages Storage Environment Executors

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

Web UI: Executors

Spark 1.4.1 | Jobs | Stages | Storage | Environment | Executors | **PySparkShell** application UI

Executors (3)

Memory: 19.0 MB Used (1325.2 MB Total)
Disk: 0.0 B Used

Executor ID	Address	RDD Blocks	Storage Memory	Disk Used	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time	Input	Shuffle Read	Shuffle Write	Logs	Thread Dump
1	abu5.cluster:56484	1	19.0 MB / 530.0 MB	0.0 B	0	0	3	3	6.8 s	0.0 B	0.0 B	0.0 B	stdout stderr	Thread Dump
2	abu3.cluster:34220	0	0.0 B / 530.0 MB	0.0 B	0	0	2	2	7.1 s	0.0 B	0.0 B	0.0 B	stdout stderr	Thread Dump
driver	10.0.0.61:45120	0	0.0 B / 265.1 MB	0.0 B	0	0	0	0	0 ms	0.0 B	0.0 B	0.0 B		Thread Dump

Outline

Spark 2.0 Data Structures [28, 29]

RDD

- Provide low-level access / transformations
- No structure on data imposed – just some bag of tuples

DataFrames extending RDDs

- Imposes a schema on tuples but tuples remain untyped
- Like a table / relational database
- Additional higher-level semantics / operators, e.g., aggregation
- Since embedded: these operators extract better performance

Datasets [28] (Spark 2 only)

- Offer strongly-typed and untyped API
- Converts tuples individually into classes with efficient encoders
- Compile time checks for datatypes (not for Python)

Higher-Level Abstractions

- Spark SQL: deal with relational tables
  - Access JDBC, hive tables or temporary tables
  - Limitations: no UPDATE statements, INSERT only for Parquet files
  - Spark SQL engine offers **Catalyst** optimizer for datasets/dataframes
- GraphX: graph processing [15]
- 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 ML tasks

Spark SQL Overview [14]

- New data structures: DataFrame representing a table with rows
  - Spark 1.X name: SchemaRDD
- RDDs can be converted to Dataframes
  - Either create a schema manually or infer the data types
- Tables can be 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 an RDD

```
# Create a table from an array using the column names value, key
# The data types of the columns are automatically inferred
r = sqlContext.createDataFrame([('test', 10), ('data', 11)], ["value", "key"])

# Alternative: create/use an RDD
rdd = sc.parallelize(range(1,10)).map(lambda x : (x, str(x)) )

# Create the table from the RDD using the columnnames given, here "key" / "value"
schema = sqlContext.createDataFrame(rdd, ["key", "value"])
schema.printSchema()

# Register table for use with SQL, we use a temporary table, so the table is NOT visible in Hive
schema.registerTempTable("nums")

# Now you can run SQL queries
res = sqlContext.sql("SELECT * from nums")

# res is an DataFrame that uses columns according to the schema
print( res.collect() ) # [Row(key=1, value='1'), Row(key=2, value='2'), ... ]

# Save results as a table for Hive
from pyspark.sql import DataFrameWriter
dw = DataFrameWriter(res)
dw.saveAsTable("data")
```

Manage Hive Tables via SQL

```
# When using an SQL statement to create the table, the table is visible in HCatalog!
p = sqlContext.sql("CREATE TABLE IF NOT EXISTS data (key INT, value STRING)")

# Bulk loading data by appending it to the table data (if it existed)
sqlContext.sql("LOAD DATA LOCAL INPATH 'data.txt' INTO TABLE data")

# The result of a SQL query is a DataFrame, an RDD of rows
rdd = sqlContext.sql("SELECT * from data")

# Treat RDD as a SchemaRDD, access row members using the column name
o = rdd.map(lambda x: x.key) # Access the column by name, here "key"
# To print the distributed values they have to be collected.
print(o.collect())

sqlContext.cacheTable("data") # Cache the table in memory

# Save as Text file/directory into the local file system
dw.json("data.json", mode="overwrite") # e.g., {"key":10,"value":"test"}

sqlContext.sql("DROP TABLE data") # Remove the table
```

Language Integrated DSL

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

```
from pyspark.sql import functions as F
# Run a select query and visualize the results
rdd.select(rdd.key, rdd.value).show()
# |key|        value|
# +---+-------------+
# | 10|         test|
# | 11|         data|
# | 12|        fritz|

# Return the rows where value == 'test'
rdd.where(rdd.value == 'test').collect()
# Print the lines from rdd where the key is bigger than 10
rdd.filter(rdd['key'] > 10).show() # rdd[X] access the column X
# Aggregate/Reduce values by the key
rdd.groupBy().avg().collect() # average(key)=11
# Similar call, short for groupBy().agg()
rdd.agg({"key": "avg"}).collect()
# Identical result for the aggregation with different notation
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

```
from pyspark.sql import HiveContext, Row
sqlContext = HiveContext(sc)
# Manually convert lines to a Row (could be done automatically)
sdf = s.map(lambda l: Row(matrikel=l[0], firstname=l[1], lastname=l[2], female=l[3], birthday=l[4]))
# infer the schema and create a table (SchemaRDD) from the data (inferSchema is deprecated but shows the idea)
schemaStudents = sqlContext.inferSchema(sdf)
schemaStudents.printSchema()
# birthday: timestamp (nullable = true), female: boolean (nullable = true), ...
schemaStudents.registerTempTable("student")

females = sqlContext.sql("SELECT firstname FROM student WHERE female == TRUE")
print(females.collect()) # print data
# [Row(firstname=u'"Nina"')]

ldf = l.map(lambda l: Row(id=l[0], name=l[1]))
schemaLecture = sqlContext.inferSchema(ldf)
schemaLecture.registerTempTable("lectures")

# Create student-lecture relation
slr = l.flatMap(lambda l: [ Row(lid=l[0], matrikel=s) for s in l[2] ] )
schemaStudLec = sqlContext.inferSchema(slr)
schemaStudLec.registerTempTable("studlec")

# Print student name and all attended lectures' names, collect_set() bags grouped items together
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 ")
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 the Parquet file.  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

```
# store each row as JSON
schemaLecture.toJSON().saveAsTextFile("lecture-json")
# load JSON
ljson = sqlContext.jsonFile("lecture-json")
# now register JSON as table
ljson.registerTempTable("ljson")
# perform SQL queries
sqlContext.sql("SELECT * FROM ljson").collect()

# Create lectures from a JSON snippet with one column as semi-structured JSON
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":{}}'])
lNewSchema = sqlContext.jsonRDD(lectureNew)
lNewSchema.registerTempTable("lnew")

# Spark natively understands nested JSON fields and can access them
sqlContext.sql("SELECT otherInfo.mailingList FROM lnew").collect()
# [Row(mailingList=u'xy'), Row(mailingList=None)]
sqlContext.sql("SELECT otherInfo.lecturer[2] FROM lnew").collect()
# [Row(_c0=u'p3'), Row(_c0=None)]
```

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 also adds new types)

Subset of provided algorithms

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

Descriptive Statistics [22]

```python
from pyspark.mllib.stat import Statistics as s
import math
# Create RDD with 4 columns
rdd = sc.parallelize( range(1,100) ).map( lambda x : [x, math.sin(x), x*x, x/100] )
sum = s.colStats(rdd) # determine column statistics
print(sum.mean()) # [  5.00e+01   3.83024876e-03   3.31666667e+03   5.00e-01]
print(sum.variance()) # [  8.25e+02   5.10311520e-01   8.788835e+06   8.25e-02]

x = sc.parallelize( range(1,100) ) # create a simple data set
y = x.map( lambda x: x / 10 + 0.5)
# Determine Pearson correlation
print(s.corr(x, y, method="pearson")) # Correlation 1.0000000000000002

# Create a random RDD with 100000 elements
from pyspark.mllib.random import RandomRDDs
u = RandomRDDs.uniformRDD(sc, 1000000)
# Estimate kernel density
from pyspark.mllib.stat import KernelDensity
kd = KernelDensity()
kd.setSample(u)
kd.setBandwidth(1.0)
# Estimate density for the given values
densities = kd.estimate( [0.2, 0, 4] )
```

Linear Models [23]

```python
from pyspark.mllib.regression import LabeledPoint, LinearRegressionWithSGD, LinearRegressionModel
import random
# Three features (x, y, label = x + 2*y + small random Value)
x = [ random.uniform(1,100) for y in range(1, 10000)]
x.sort()
y = [ random.uniform(1,100) for y in range(1, 10000)]
# LabeledPoint identifies the result variable
raw = [ LabeledPoint(i+j+random.gauss(0,4), [i/100, j/200])  for (i,j) in zip(x, y) ]
data = sc.parallelize( raw )

# Build the model using maximum of 10 iterations with stochastic gradient descent
model = LinearRegressionWithSGD.train(data, 100)

print(model.intercept)
# 0.0
print(model.weights)
#[110.908004953,188.96464824] => we except [100, 200]

# Validate the model with the original training data
vp = data.map(lambda p: (p.label, model.predict(p.features)))

# Error metrics
abserror = vp.map(lambda p: abs(p[0] - p[1])).reduce(lambda x, y: x + y) / vp.count()
error = vp.map(lambda p: abs(p[0] - p[1]) / p[0]).reduce(lambda x, y: x + y) / vp.count()
MSE = vp.map(lambda p: (p[0] - p[1])**2).reduce(lambda x, y: x + y) / vp.count()
print("Abs error: %.2f" % (abserror)) # 4.41
print("Rel. error: %.2f%%" % (error * 100)) # 5.53%
print("Mean Squared Error: %.2f" % (MSE))

# Save / load the model
model.save(sc, "myModelPath")
model = LinearRegressionModel.load(sc, "myModelPath")
```

Clustering [25]

```
# Clustering with k-means is very simple for N-Dimensional data
from pyspark.mllib.clustering import KMeans, KMeansModel
import random as r
from numpy import array
# Create 3 clusters in 2D at (10,10), (50,30) and (70,70)
x = [ [r.gauss(10,4), r.gauss(10,2)] for y in range(1, 100) ]
x.extend( [r.gauss(50,5), r.gauss(30,3)] for y in range(1, 900) )
x.extend( [r.gauss(70,5), r.gauss(70,8)] for y in range(1, 500) )
x = [ array(x) for x in x]

data = sc.parallelize(x)

# Apply k-means
clusters = KMeans.train(data, 3, maxIterations=10, runs=10, initializationMode="random")

print(clusters.clusterCenters)
# [array([ 70.42953058,  69.88289475]),
#  array([ 10.57839294,   9.92010409]),
#  array([ 49.72193422,  30.15358142])]

# Save/load model
clusters.save(sc, "myModelPath")
sameModel = KMeansModel.load(sc, "myModelPath")
```

Decision Trees [25]

```
# Decision trees operate on tables and don't use LabeledPoint ...
# They offer the concept of a pipleline to preprocess data in RDD
from pyspark.mllib.linalg import Vectors
from pyspark.sql import Row
from pyspark.ml.classification import DecisionTreeClassifier
from pyspark.ml.feature import StringIndexer
from pyspark.ml import Pipeline
from pyspark.ml.evaluation import BinaryClassificationEvaluator
import random as r

# We create a new random dataset but now with some overlap
x = [ ("blue", [r.gauss(10,4), r.gauss(10,2)]) for y in range(1, 100) ]
x.extend( ("red", [r.gauss(50,5), r.gauss(30,3)]) for y in range(1, 900) )
x.extend( ("yellow", [r.gauss(70,15), r.gauss(70,25)]) for y in range(1, 500) ) # Class red and yellow may overlap

data = sc.parallelize(x).map(lambda x: (x[0], Vectors.dense(x[1])))
# The data frame is expected to contain exactly the specified two columns
dataset = sqlContext.createDataFrame(data, ["label", "features"])

# Create a numeric index from string label categories, this is mandatory!
labelIndexer = StringIndexer(inputCol="label", outputCol="indexedLabel").fit(dataset)

# Our decision tree
dt = DecisionTreeClassifier(featuresCol='features', labelCol='indexedLabel', predictionCol='prediction', maxDepth=5)

# Split data into 70% training set and 30% validation set
(trainingData, validationData) = dataset.randomSplit([0.7, 0.3])

# Create a pipeline which processes dataframes and run it to create the model
pipeline = Pipeline(stages=[labelIndexer, dt])
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
```

Integration into R

Integrated R shell: `sparkR`

Features

- Store/retrieve data frames in/from Spark
- In-memory SQL and access to HDFS data and Hive tables
- Provides functions to: (lazily) access/derive data and ML-algorithms
- Enables (lazy) parallelism in R!

```r
# Creating a DataFrame from the iris (plant) data
df = as.DataFrame(data=iris, sqlContext=sqlContext)
# Register it as table to enable SQL queries
registerTempTable(df, "iris")
# Run an SQL query
d = sql(sqlContext, "SELECT Species FROM iris WHERE Sepal_Length >= 1 AND Sepal_Width <= 19")

# Compute the number of instances for each species using a reduction
x = summarize(groupBy(df, df$Species), count = n(df$Species))
head(x) # Returns the three species with 50 instances

# Retrieving a Spark DataFrame and converting it into a regular (R) data frame
s = as.data.frame(d)
summary(s)
```

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 / DataFrame / DataSet
  - 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

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

26 https://en.wikipedia.org/wiki/In-memory_processing

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

28 https://databricks.com/blog/2016/07/14/a-tale-of-three-apache-spark-apis-rdds-dataframes-and-datasets.html

29 https://databricks.com/blog/2016/01/04/introducing-apache-spark-datasets.html

30 https://databricks.gitbooks.io/databricks-spark-knowledge-base/content/best_practices/prefer_reducebykey_over_groupbykey.html