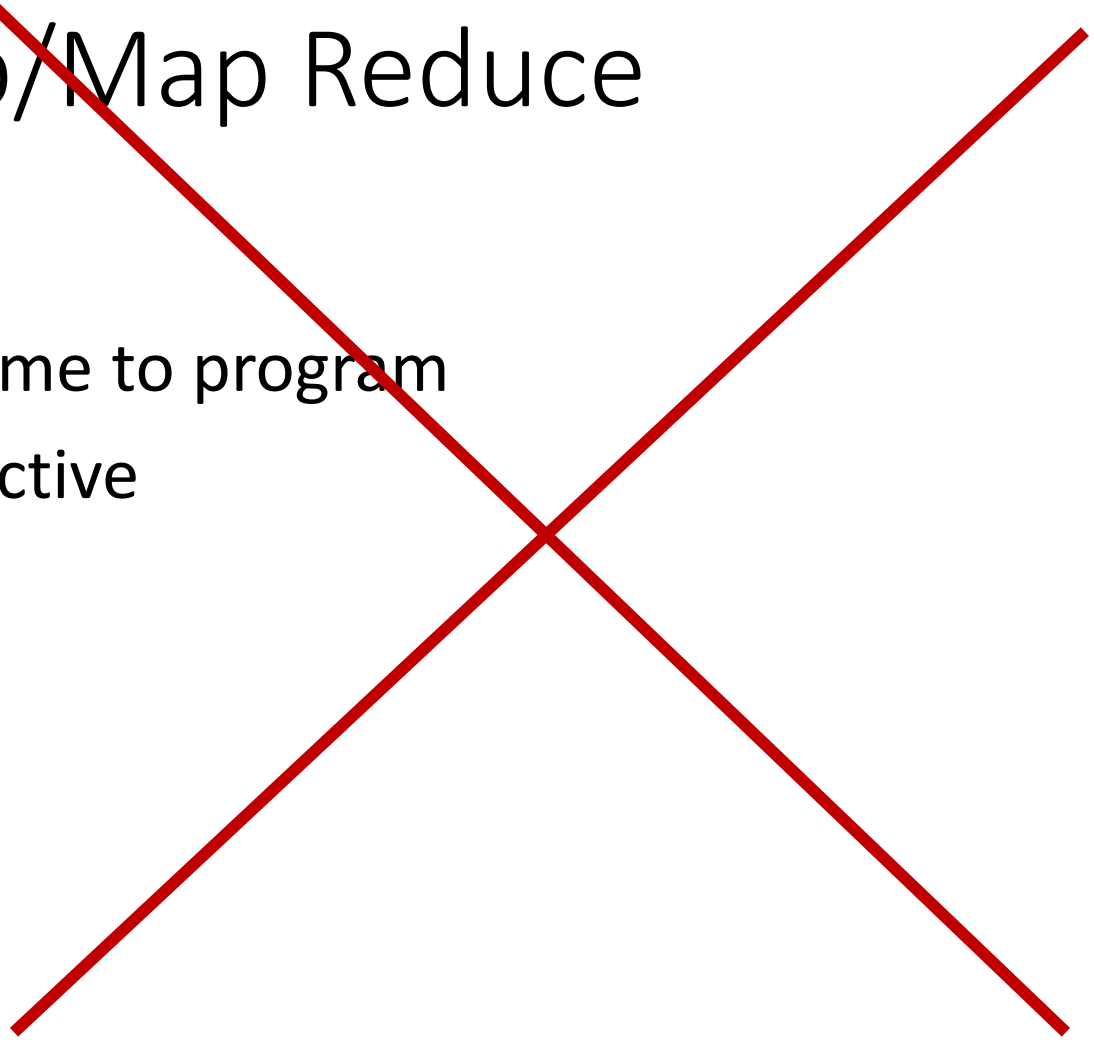
Apache Spark

- Interactive Exploration
- Programmatically
- Workflows

Hadoop/Map Reduce



- Java
- Cumbersome to program
- Not interactive

Apache Spark

https://spark.apache.org/

Apache Spark + Jupyter Notebook

docker pull jupyter/all-spark-notebook
https://jupyter-docker-stacks.readthedocs.io/en/latest/index.html

Apache Spark ‘cluster’

Docker: https://hub.docker.com/r/bitnami/spark
NYU: https://sites.google.com/nyu.edu/nyu-hpc/hpc-systems/cloud-computing/dataproc?authuser=0

Apache Spark + Dask (later)

Spark Research Papers

- *Spark: Cluster Computing with Working Sets*
  Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, Ion Stoica
  USENIX HotCloud (2010)
  people.csail.mit.edu/matei/papers/2010/hotcloud_spark.pdf

- *Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing*
  Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica
  NSDI (2012)
  usenix.org/system/files/conference/nsdi12/nsdi12-final138.pdf

Apache Spark Motivation

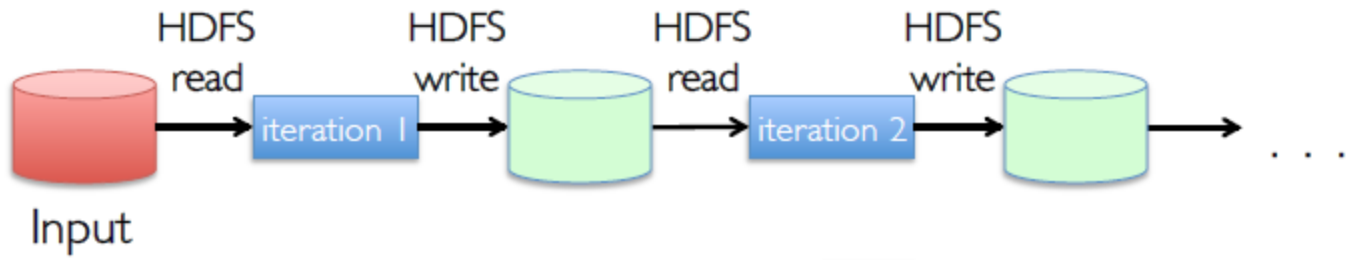
- Using Map Reduce for complex jobs, interactive queries and online processing involves *lots of disk I/O*

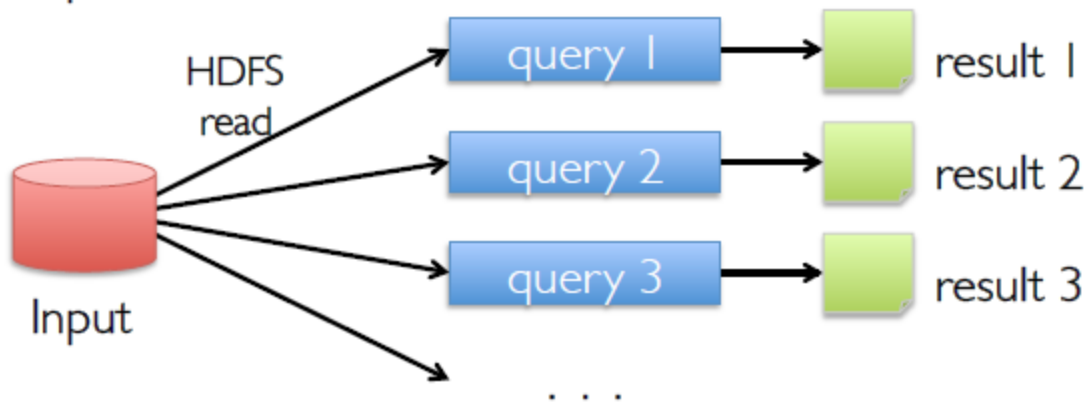
Interactive mining Stream processing

Also, iterative jobs

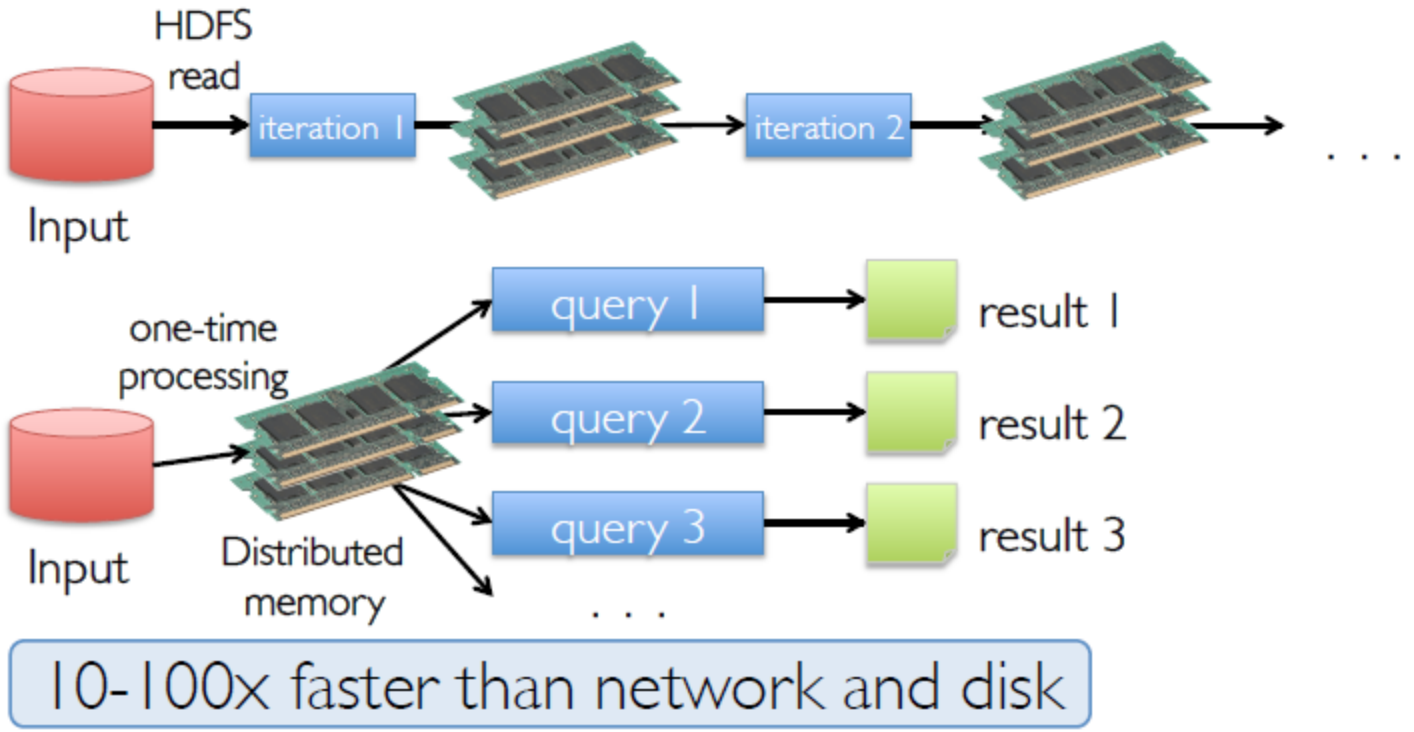
Disk I/O is very slow

Use Memory Instead of Disk



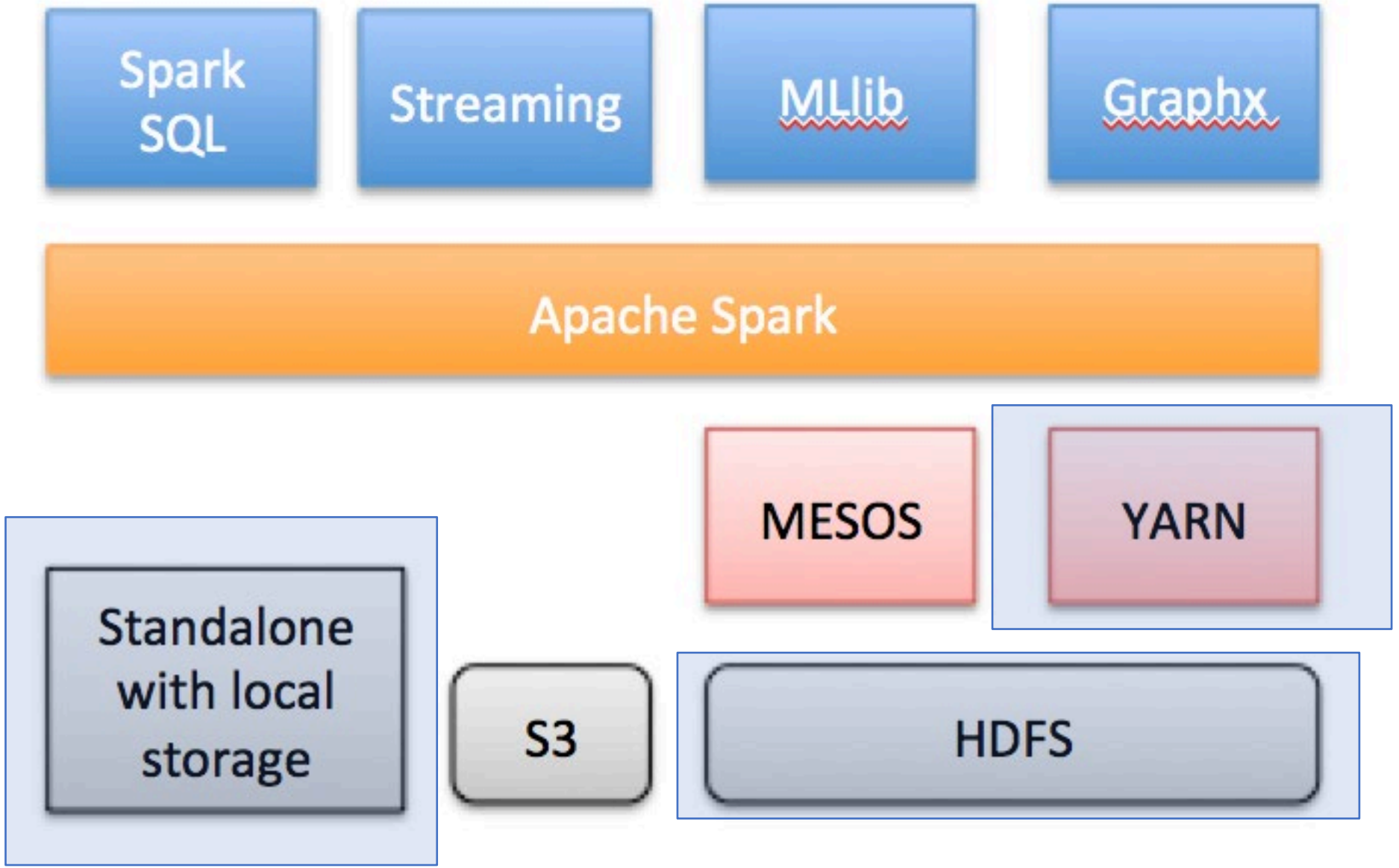


In-Memory Data Sharing

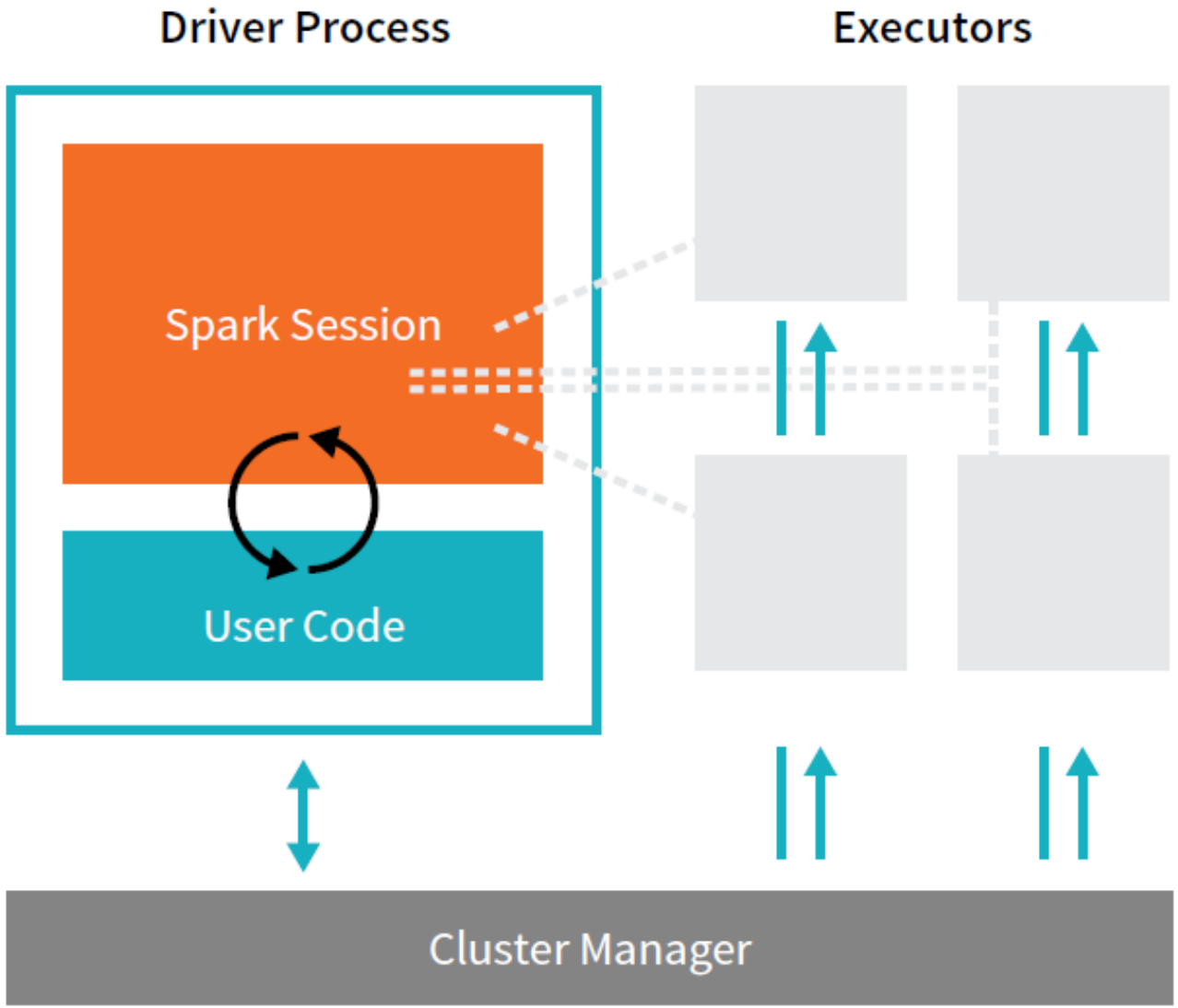


10-100x faster than network and disk

Apache Spark



Spark's Basic Architecture



- Cluster manager maintains an understanding of the resources available.

- The driver process is responsible for executing our driver program’s commands accross the executors

- Executors, for the most part, will always be running Spark code.

- Driver can be "driven" from a number of different languages

Spark's Language APIs

- Scala – the *default*
- Java
- Python - PySpark
- SQL – ANSI SQL 2003
- R - SparkR, sparklyr

Execution Modes:

Batch submit

Local shell interpreters

Notebooks

Spark Essentials: Master

- The `master` parameter for a `SparkContext` determines which type and size of cluster to use

Master Parameter	Description
`local`	run Spark locally with one worker thread (no parallelism)
`local[K]`	run Spark locally with K worker threads (ideally set to number of cores)
`spark://HOST:PORT`	connect to a Spark standalone cluster; PORT depends on config (7077 by default)
`mesos://HOST:PORT`	connect to a Mesos cluster; PORT depends on config (5050 by default)

In the labs, we set the master parameter for you

spark-shell (scala)

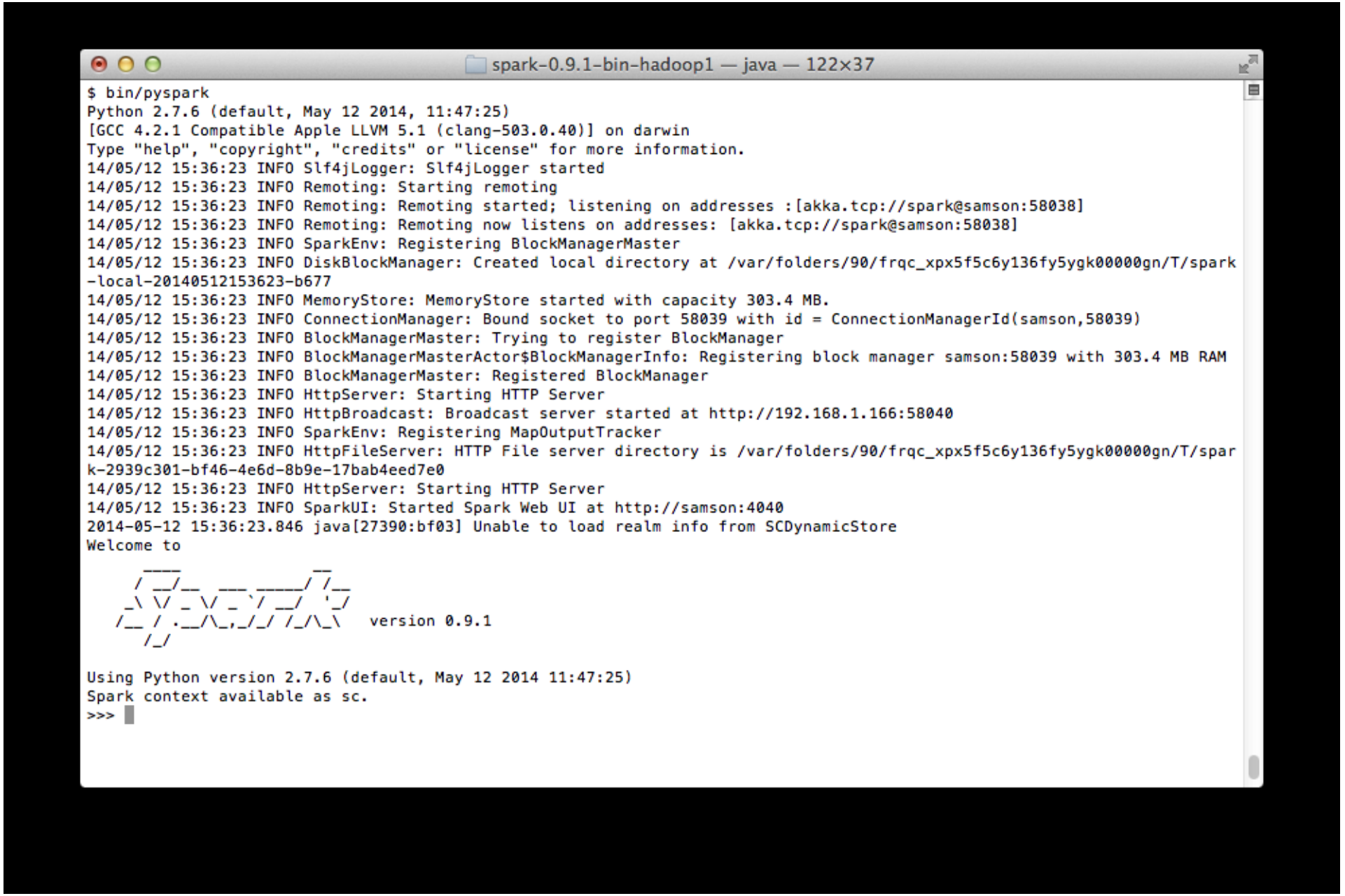
`spark-shell –master local[*]`

Python: **pyspark**

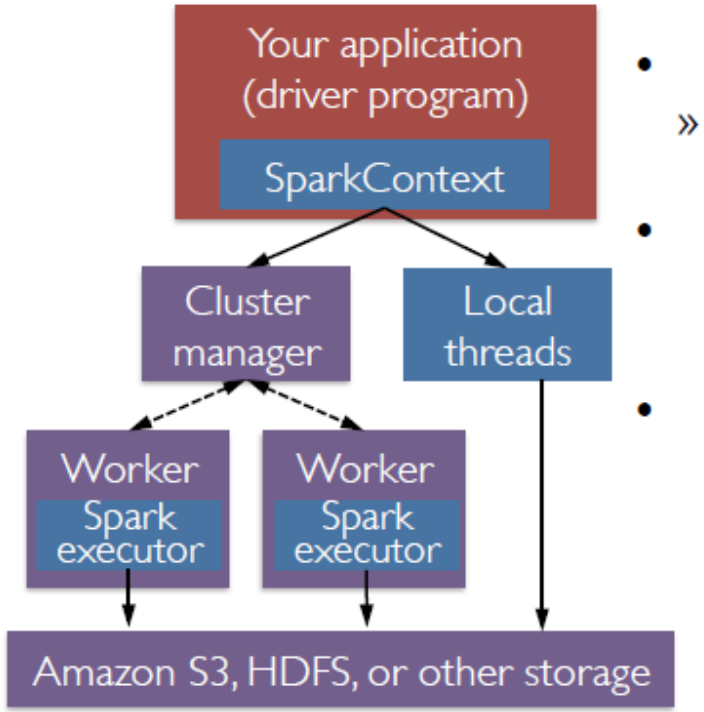
Zeppelin:
http://zeppelin.apache.org/download.html

Jupyter:
https://jupyter.org/
https://github.com/jupyter/docker-stacks
`Docker: jupyter/all-spark-notebook`

`NYU: jupyterhub, Dataproc`

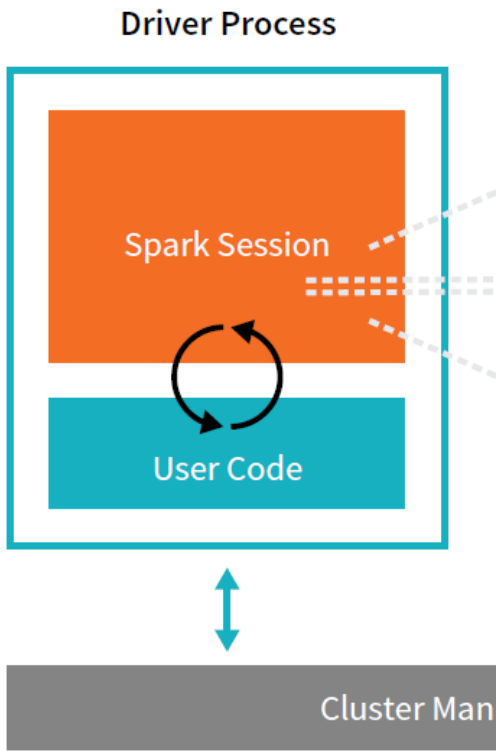

spark-0.9.1-bin-hadoop1 — java — 122×37
$ bin/pyspark
Python 2.7.6 (default, May 12 2014, 11:47:25)
[GCC 4.2.1 Compatible Apple LLVM 5.1 (clang-503.0.40)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
14/05/12 15:36:23 INFO Slf4jLogger: Slf4jLogger started
14/05/12 15:36:23 INFO Remoting: Starting remoting
14/05/12 15:36:23 INFO Remoting: Remoting started; listening on addresses :[akka.tcp://spark@samson:58038]
14/05/12 15:36:23 INFO Remoting: Remoting now listens on addresses: [akka.tcp://spark@samson:58038]
14/05/12 15:36:23 INFO SparkEnv: Registering BlockManagerMaster
14/05/12 15:36:23 INFO DiskBlockManager: Created local directory at /var/folders/90/frqc_xpx5f5c6y136fy5ygk00000gn/T/spark-local-20140512153623-b677
14/05/12 15:36:23 INFO MemoryStore: MemoryStore started with capacity 303.4 MB.
14/05/12 15:36:23 INFO ConnectionManager: Bound socket to port 58039 with id = ConnectionManagerId(samson,58039)
14/05/12 15:36:23 INFO BlockManagerMaster: Trying to register BlockManager
14/05/12 15:36:23 INFO BlockManagerMasterActor$BlockManagerInfo: Registering block manager samson:58039 with 303.4 MB RAM
14/05/12 15:36:23 INFO BlockManagerMaster: Registered BlockManager
14/05/12 15:36:23 INFO HttpServer: Starting HTTP Server
14/05/12 15:36:23 INFO HttpBroadcast: Broadcast server started at http://192.168.1.166:58040
14/05/12 15:36:23 INFO SparkEnv: Registering MapOutputTracker
14/05/12 15:36:23 INFO HttpFileServer: HTTP File server directory is /var/folders/90/frqc_xpx5f5c6y136fy5ygk00000gn/T/spark-2939c301-bf46-4e6d-8b9e-17bab4eed7e0
14/05/12 15:36:23 INFO HttpServer: Starting HTTP Server
14/05/12 15:36:23 INFO SparkUI: Started Spark Web UI at http://samson:4040
2014-05-12 15:36:23.846 java[27390:bf03] Unable to load realm info from SCDynamicStore
Welcome to
version 0.9.1
Using Python version 2.7.6 (default, May 12 2014 11:47:25)
Spark context available as sc.
>>>

Spark Driver and Workers



- A Spark program is two programs:
  - » A driver program and a workers program
- Worker programs run on cluster nodes or in local threads
- RDDs are distributed across workers

The SparkSession



- The SparkSession instance is the way Spark executes user-defined tasks across the cluster.
- one to one correspondance between a SparkSession and a Spark Application.
- In Scala and Python shells the variable is available as **`spark`**

The SparkSession

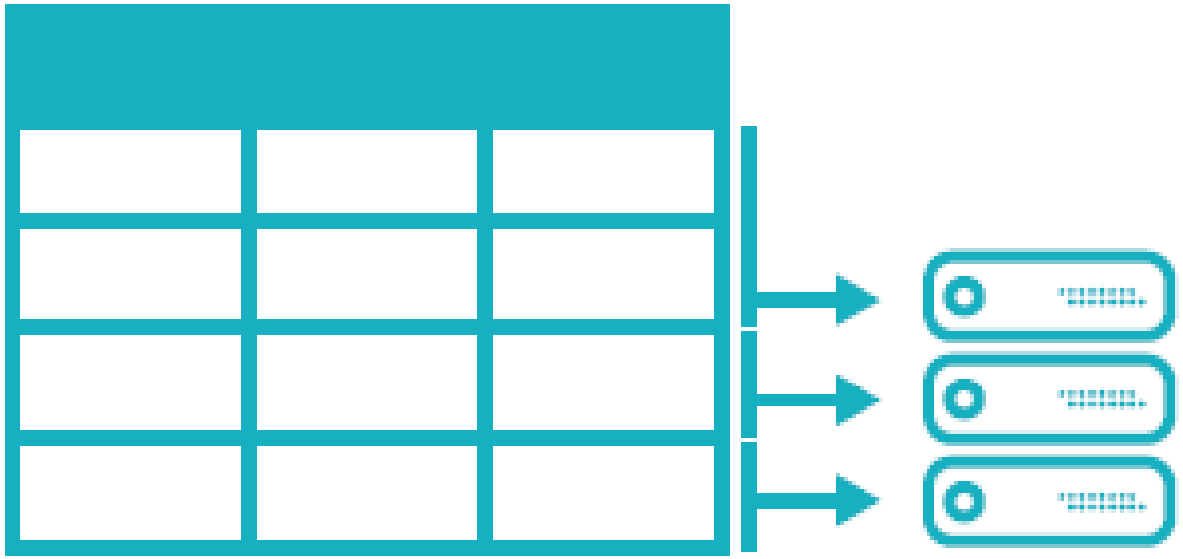
In notebooks, you may have to create the spark context

```
import os
import pyspark
conf = pyspark.SparkConf()

conf.set('spark.ui.proxyBase', '/user/' + os.environ['JUPYTERHUB_USER'] + '/proxy/4041')
conf.set('spark.sql.repl.eagerEval.enabled', True)
conf.set('spark.driver.memory','4g')
sc = pyspark.SparkContext(conf=conf)

spark = pyspark.SQLContext.getOrCreate(sc)
```

DataFrames

Spreadsheet on a single machine

Table or DataFrame partitioned across servers in data center



Source: Databricks

Spark DataFrames

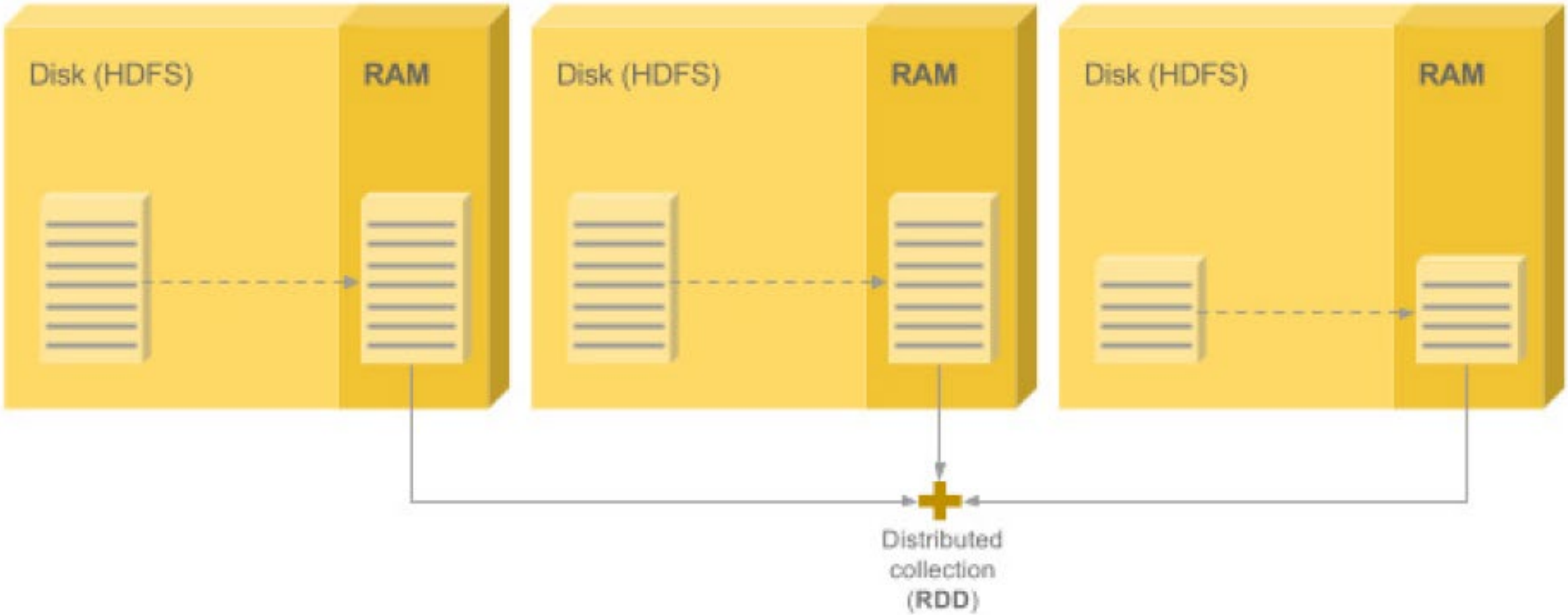
API inspired by R and Python Pandas

- Python, Scala, Java (+ R in dev)
- Pandas integration

Distributed DataFrame

Highly optimized

```
val lines = sc.textFile("hdfs://path/to/the/file")
```



```
%scala
val myRange = spark.range(1000).toDF("number")
```

```
%python
myRange = spark.range(1000).toDF("number")
```

Spark

DataFrames

dept	age	name
Bio	48	H Smith
CS	54	A Turing
Bio	43	B Jones
Chem	61	M Kennedy

Data grouped into named columns

RDD API

```
pdata.map(lambda x: (x.dept, [x.age, 1])) \
  .reduceByKey(lambda x, y: [x[0] + y[0], x[1] + y[1]]) \
  .map(lambda x: [x[0], x[1][0] / x[1][1]]) \
  .collect()
```

DataFrame API

```
data.groupBy("dept").avg("age")
```

DataFrames

dept	age	name
Bio	48	H Smith
CS	54	A Turing
Bio	43	B Jones
Chem	61	M Kennedy

Data grouped into named columns

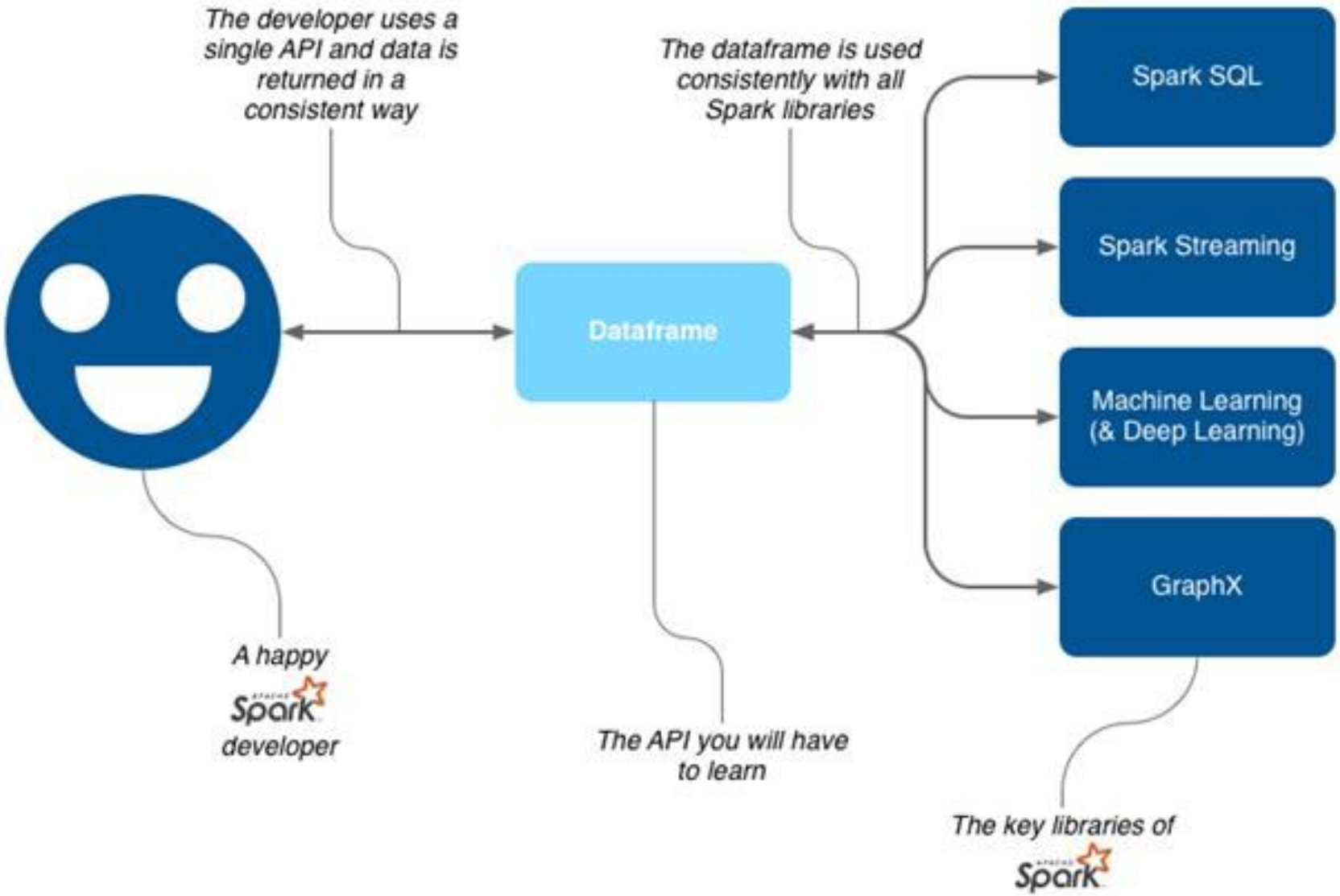
DSL for common tasks

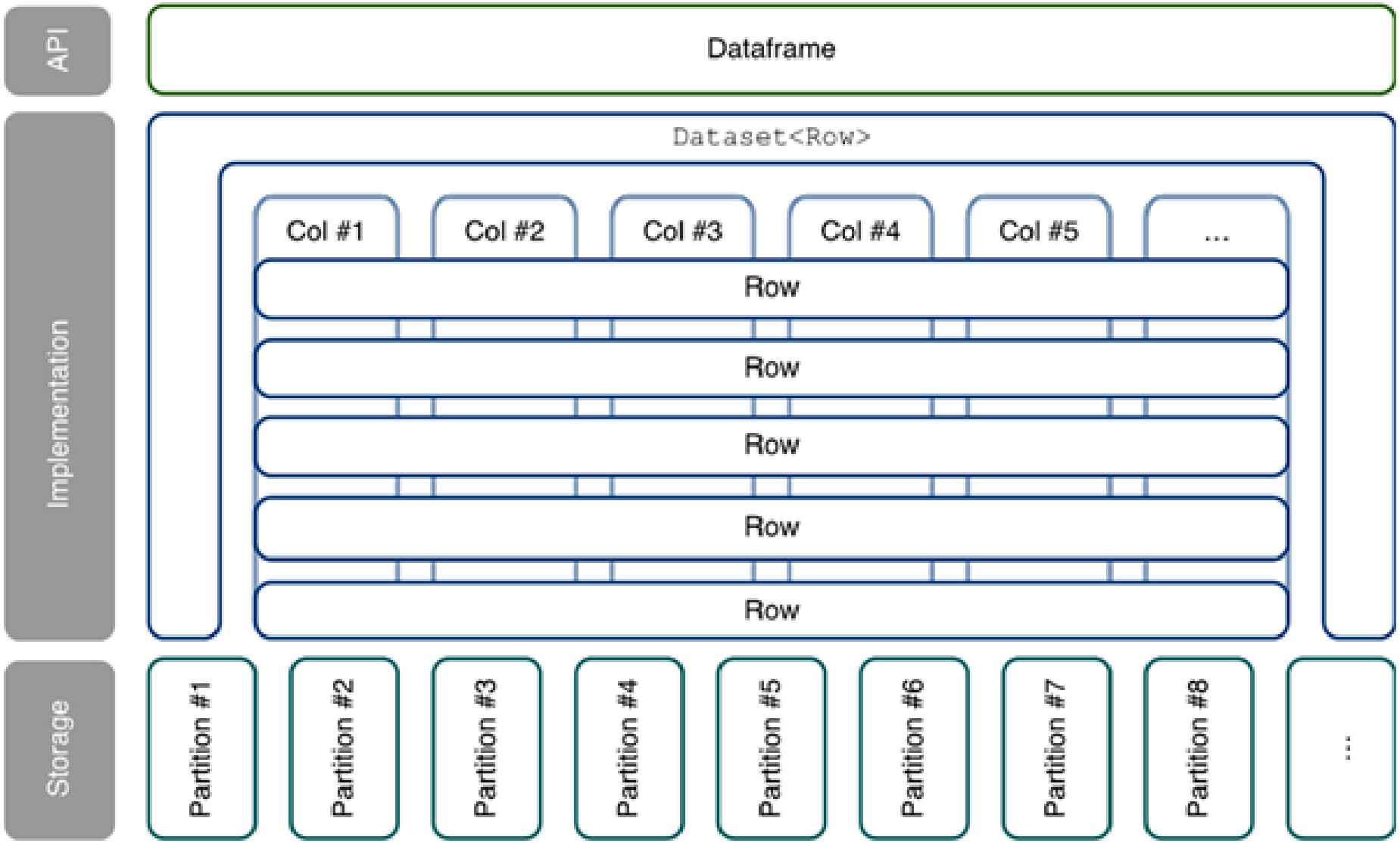
- Project, filter, aggregate, join, …
- Metadata
- UDFs

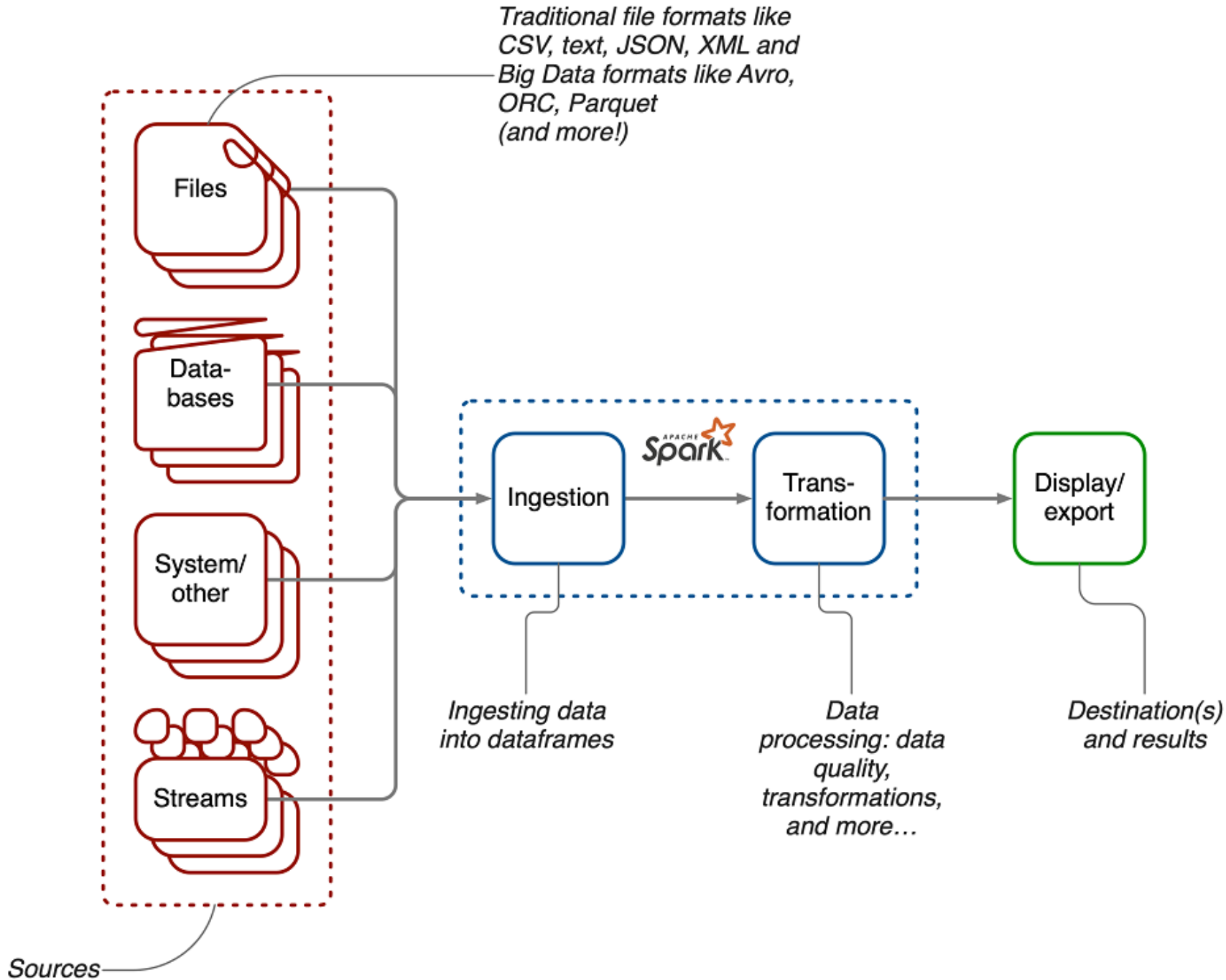
```
data.groupBy("dept").avg("age")
```

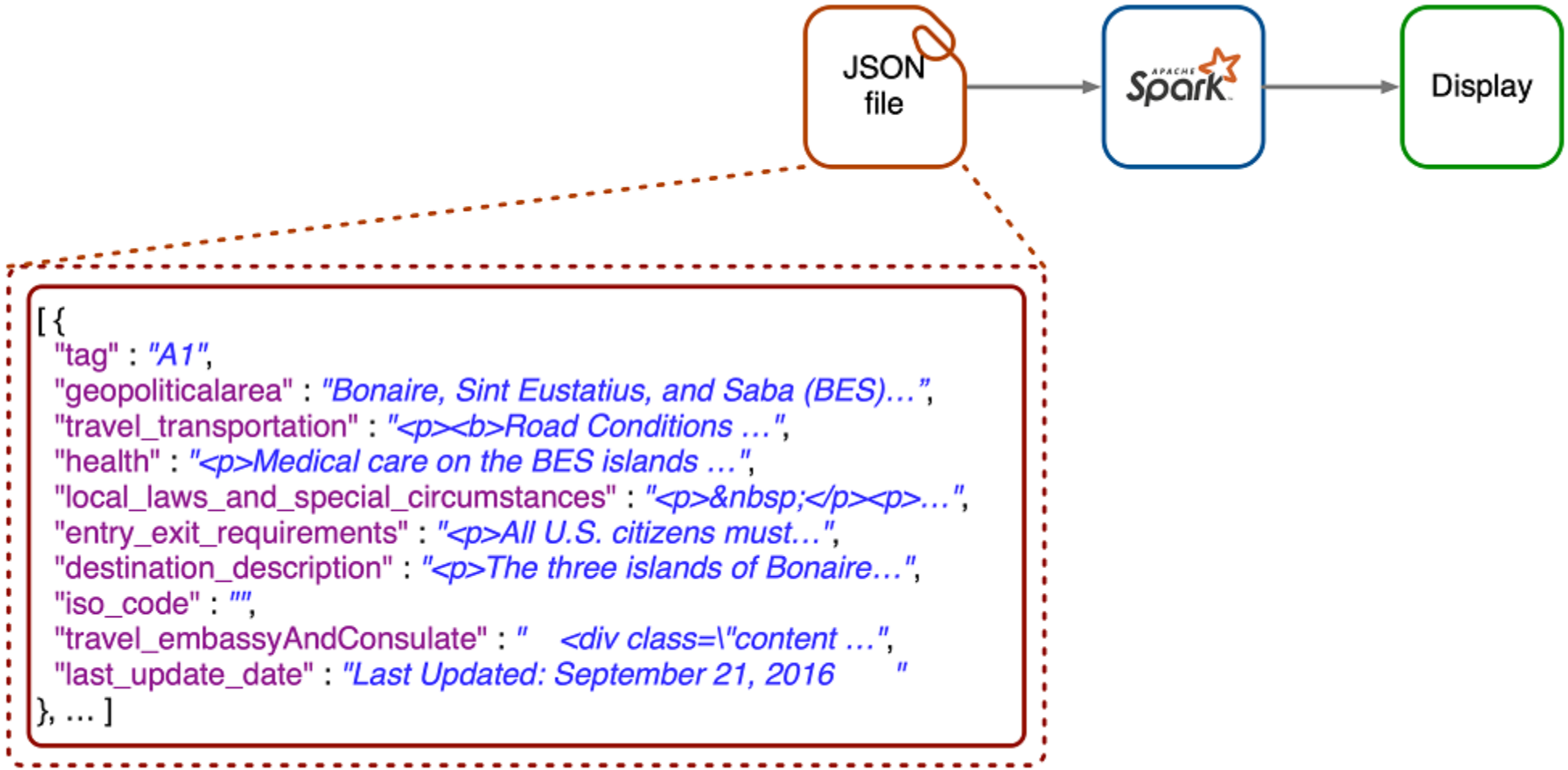
```
scala> val lines = sc.textFile("README.md") // Create an RDD called lines
lines: spark.RDD[String] = MappedRDD[...]

scala> lines.count() // Count the number of items in this RDD
res0: Long = 127

scala> lines.first() // First item in this RDD, i.e. first line of README.md
res1: String = # Apache Spark
```

Dataframe


The developer uses a single API and data is returned in a consistent way
The dataframe is used consistently with all Spark libraries
Spark SQL
Spark Streaming
Machine Learning (& Deep Learning)
GraphX
Dataframe
A happy Spark developer
The API you will have to learn
The key libraries of Spark


API
Dataframe
Implementation
Dataset<Row>
Col #1
Col #2
Col #3
Col #4
Col #5
...
Row
Row
Row
Row
Row
Storage
Partition #1
Partition #2
Partition #3
Partition #4
Partition #5
Partition #6
Partition #7
Partition #8
...


Traditional file formats like CSV, text, JSON, XML and Big Data formats like Avro, ORC, Parquet (and more!)
Files
Data-bases
System/ other
Streams
Sources
Ingestion
Trans-formation
Display/ export
Ingesting data into dataframes
Data processing: data quality, transformations, and more…
Destination(s) and results


JSON
file
Spark
Display
[{
"tag" : "A1",
"geopoliticalarea" : "Bonaire, Sint Eustatius, and Saba (BES)…",
"travel_transportation" : " <b>Road Conditions …",
"health" : " Medical care on the BES islands …",
"local_laws_and_special_circumstances" : "   …",
"entry_exit_requirements" : " All U.S. citizens must…",
"destination_description" : " The three islands of Bonaire…",
"iso_code" : "",
"travel_embassyAndConsulate" : " <div class=\"content …",
"last_update_date" : "Last Updated: September 21, 2016 "
}, …]

Transformations

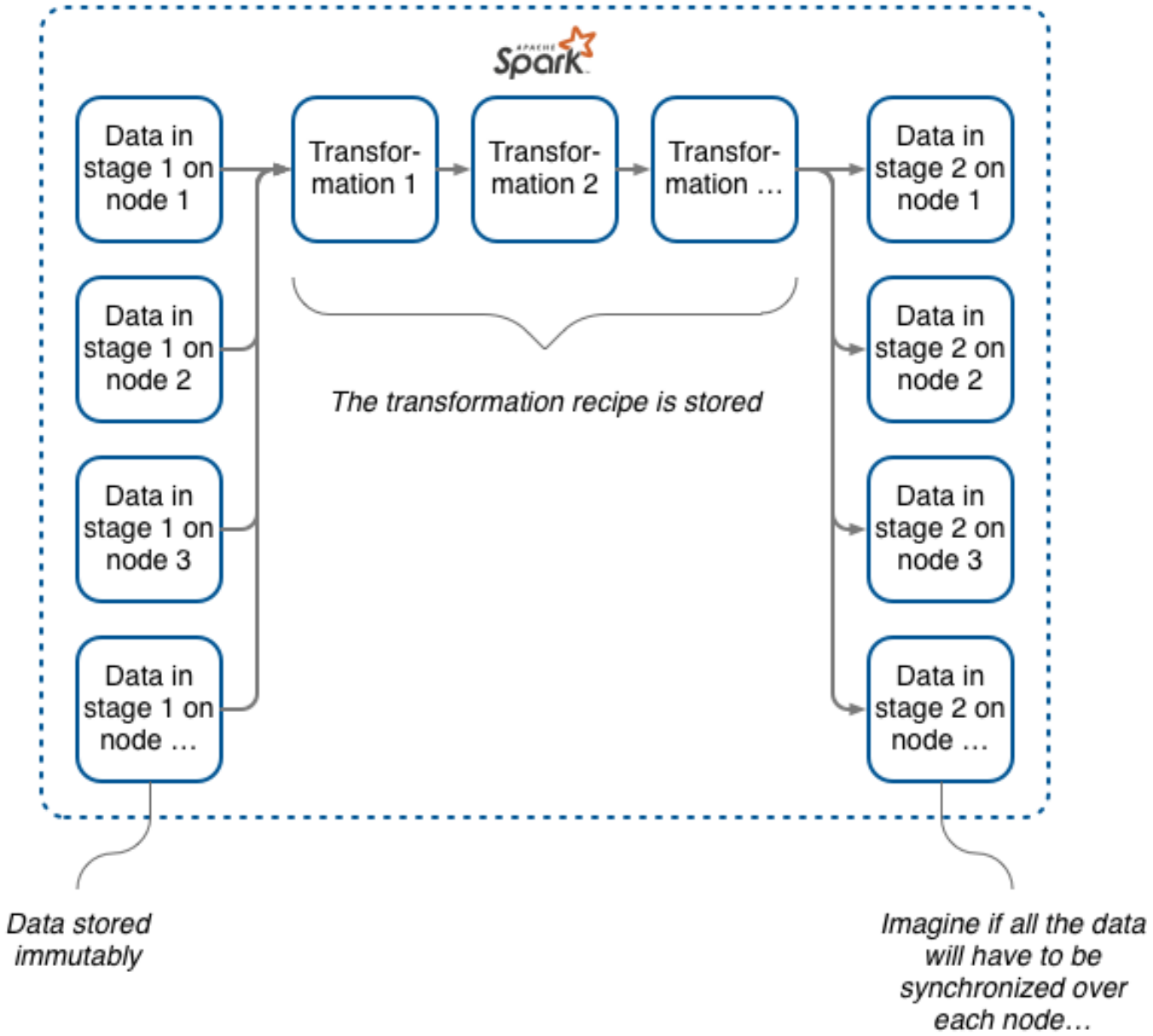
Lazy Evaluation

Actions

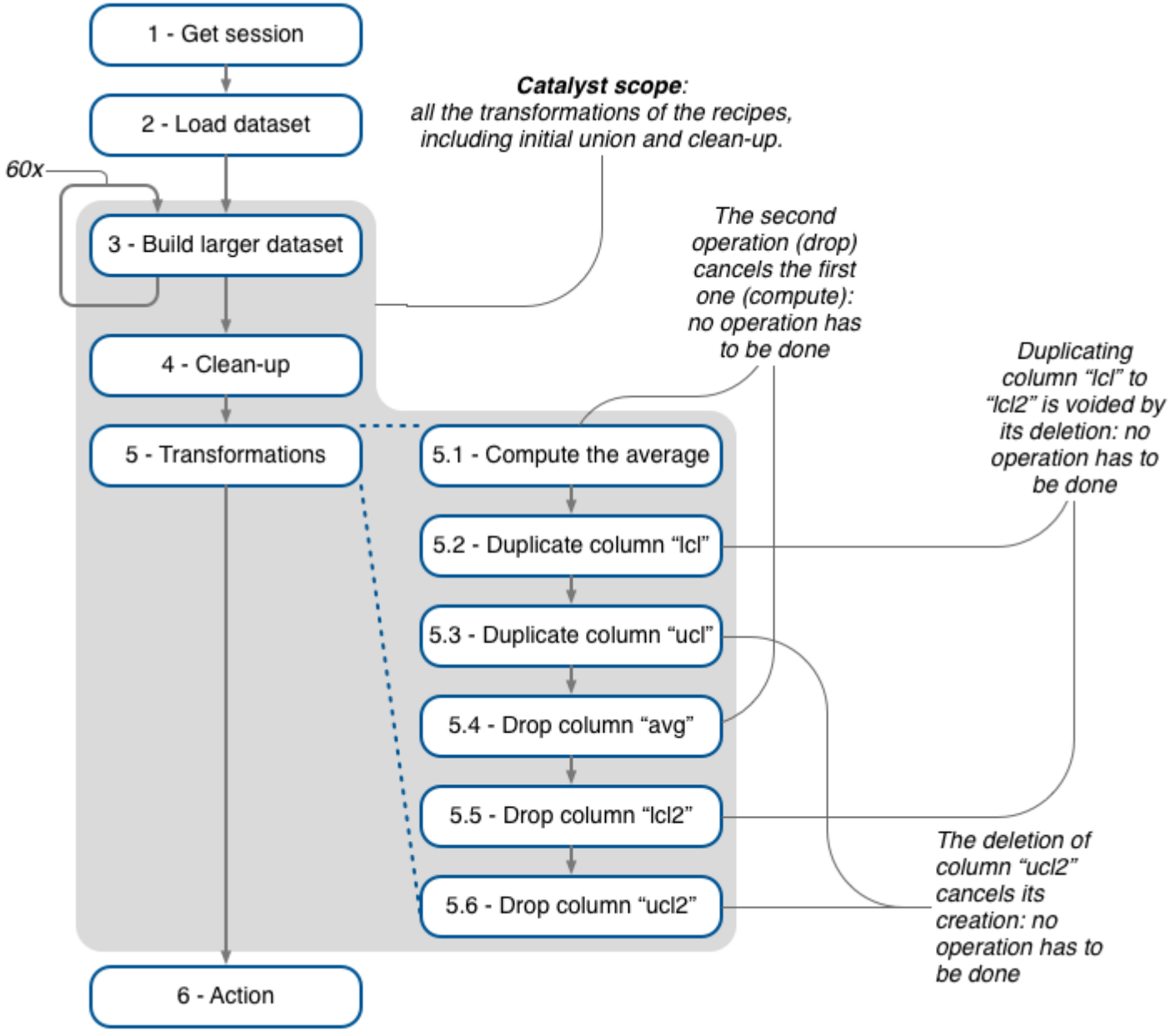
Spark Transformations

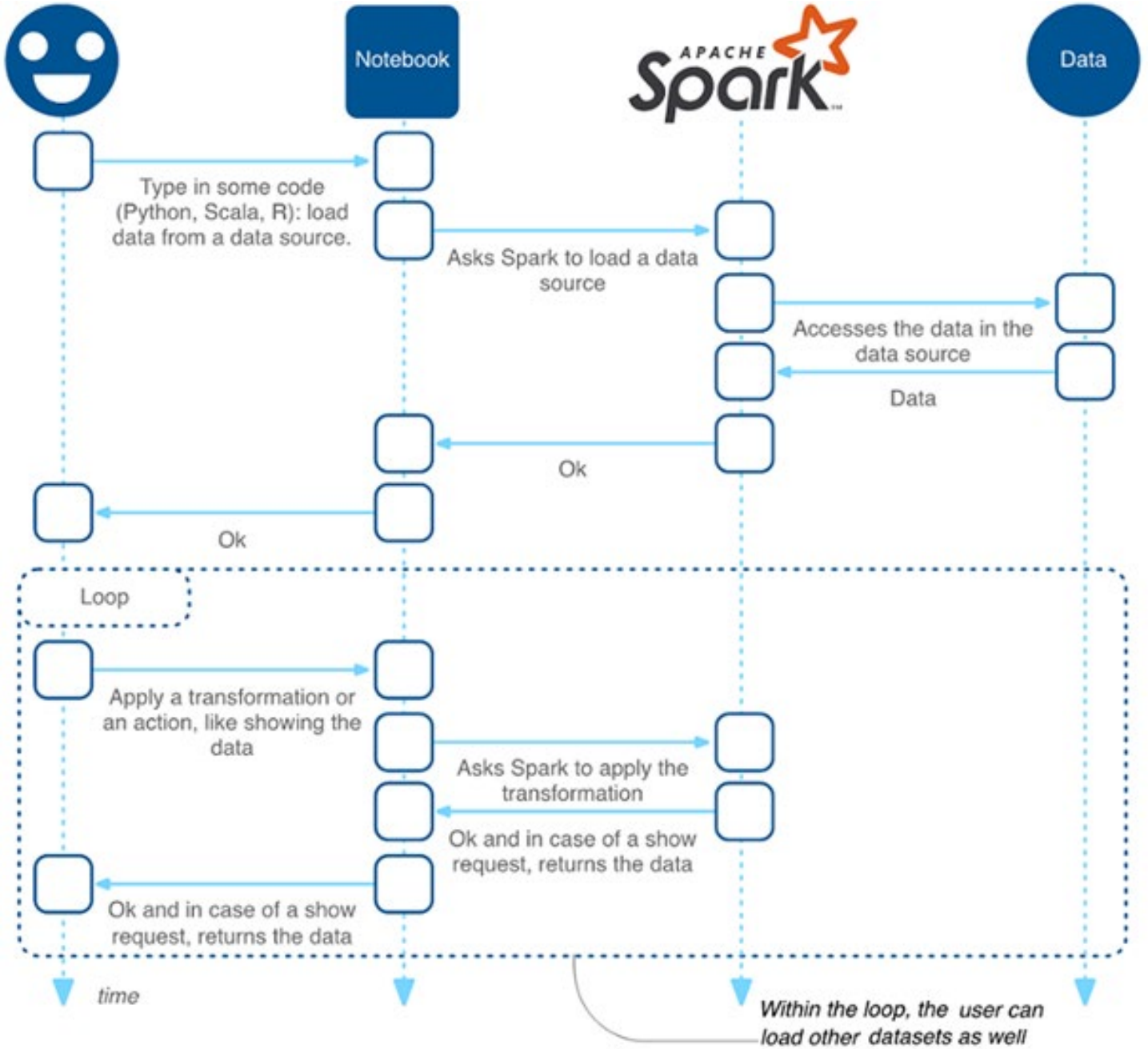
- Create new datasets from an existing one
- Use *lazy evaluation*: results not computed right away – instead Spark remembers set of transformations applied to base dataset
  - » Spark optimizes the required calculations
  - » Spark recovers from failures and slow workers
- Think of this as a recipe for creating result

Lazy Evaluation

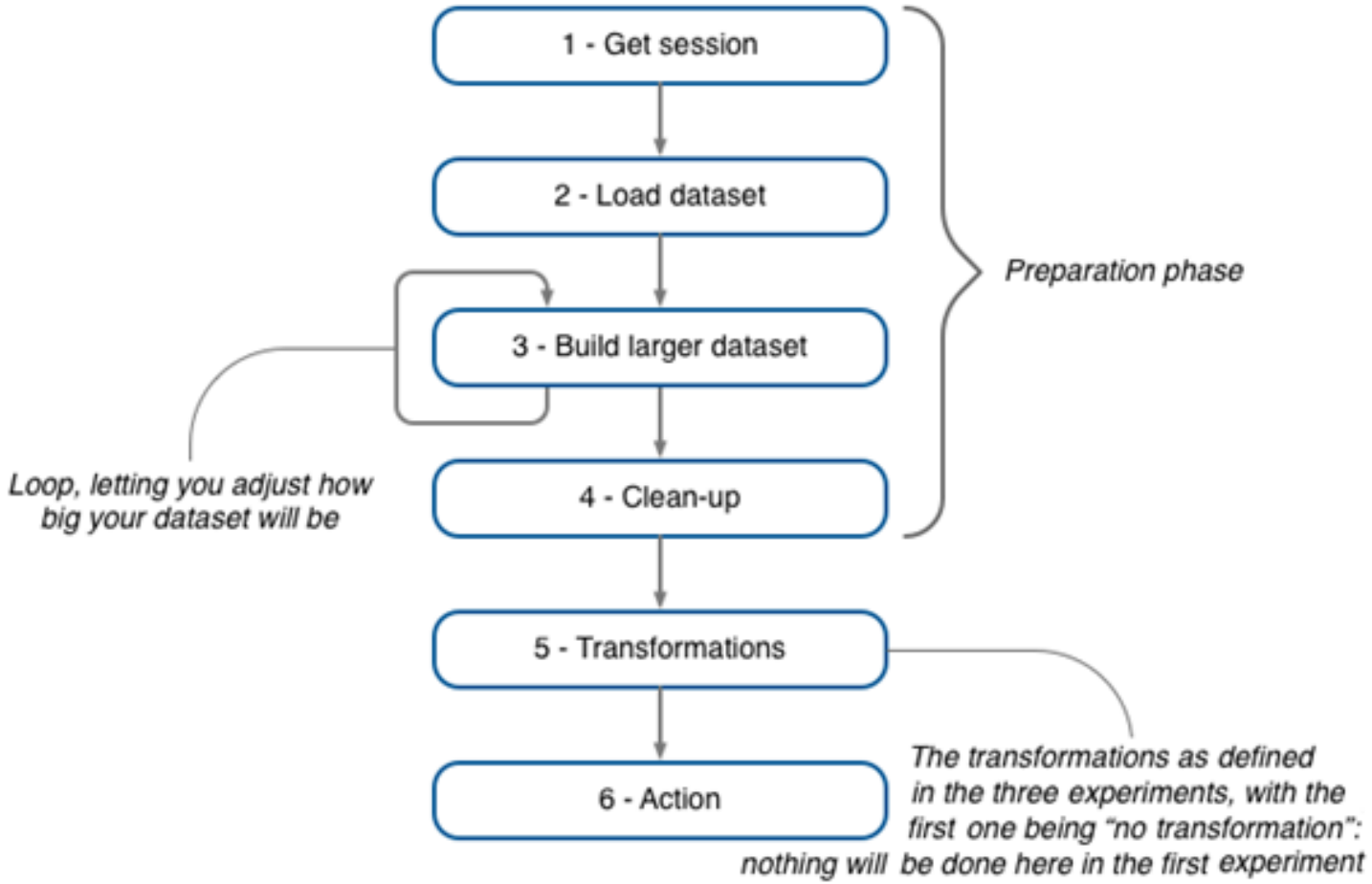


Optimizer (Catalyst)




Notebook
APACHE Spark
Data
Type in some code (Python, Scala, R): load data from a data source.
Asks Spark to load a data source
Accesses the data in the data source
Data
Ok
Ok
Loop
Apply a transformation or an action, like showing the data
Asks Spark to apply the transformation
Ok and in case of a show request, returns the data
Ok and in case of a show request, returns the data
time
Within the loop, the user can load other datasets as well

Lazy Evaluation



Some Transformations

Transformation	Description
`map(`*`func`*`)`	return a new distributed dataset formed by passing each element of the source through a function *func*
`filter(`*`func`*`)`	return a new dataset formed by selecting those elements of the source on which *func* returns true
`distinct([`*`numTasks`*`]))`	return a new dataset that contains the distinct elements of the source dataset
`flatMap(`*`func`*`)`	similar to map, but each input item can be mapped to 0 or more output items (so *func* should return a Seq rather than a single item)

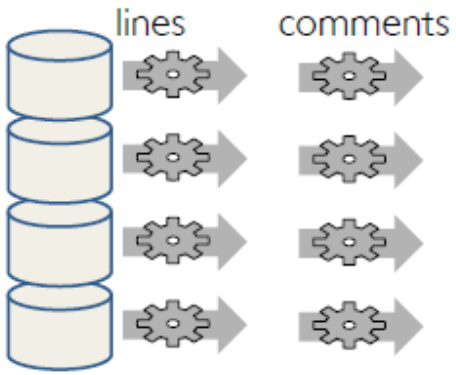
Transformations

```
>>> rdd = sc.parallelize([1, 2, 3])
>>> rdd.Map(lambda x: [x, x+5])
RDD: [1, 2, 3] → [[1, 6], [2, 7], [3, 8]]

>>> rdd.flatMap(lambda x: [x, x+5])
RDD: [1, 2, 3] → [1, 6, 2, 7, 3, 8]
```

Function literals (green) are closures automatically passed to workers

Transforming an RDD

```
lines = sc.textFile("...", 4)

comments = lines.filter(isComment)
```



Lazy evaluation means nothing executes – Spark saves recipe for transforming source

Some Key-Value Transformations

Key-Value Transformation	Description
`reduceByKey(`*`func`*`)`	return a new distributed dataset of (K,V) pairs where the values for each key are aggregated using the given reduce function *func*, which must be of type (V,V) ➔ V
`sortByKey()`	return a new dataset (K,V) pairs sorted by keys in ascending order
`groupByKey()`	return a new dataset of (K, Iterable<V>) pairs

Key-Value Transformations

```
>>> rdd = sc.parallelize([(1,2), (3,4), (3,6)])
>>> rdd.reduceByKey(lambda a, b: a + b)
RDD: [(1,2), (3,4), (3,6)] → [(1,2), (3,10)]

>>> rdd2 = sc.parallelize([(1,'a'), (2,'c'), (1,'b')])
>>> rdd2.sortByKey()
RDD: [(1,'a'), (2,'c'), (1,'b')] →
                [(1,'a'), (1,'b'), (2,'c')]
```

Key-Value Transformations

```
>>> rdd2 = sc.parallelize([(1,'a'), (2,'c'), (1,'b')])
>>> rdd2.groupByKey()
RDD: [(1,'a'), (1,'b'), (2,'c')] →
                [(1,['a','b']), (2,['c'])]
```

Be careful using **groupByKey()** as it can cause a lot of data movement across the network and create large Iterables at workers

Spark Actions

- Cause Spark to execute recipe to transform source
- Mechanism for getting results out of Spark

Some Actions

Action	Description
`reduce(`*`func`*`)`	aggregate dataset's elements using function *func*. *func* takes two arguments and returns one, and is commutative and associative so that it can be computed correctly in parallel
`take(`*`n`*`)`	return an array with the first *n* elements
`collect()`	return all the elements as an array WARNING: make sure will fit in driver program
`takeOrdered(`*`n, key=func`*`)`	return n elements ordered in ascending order or as specified by the optional key function

PySpark

Review: Python `lambda` Functions

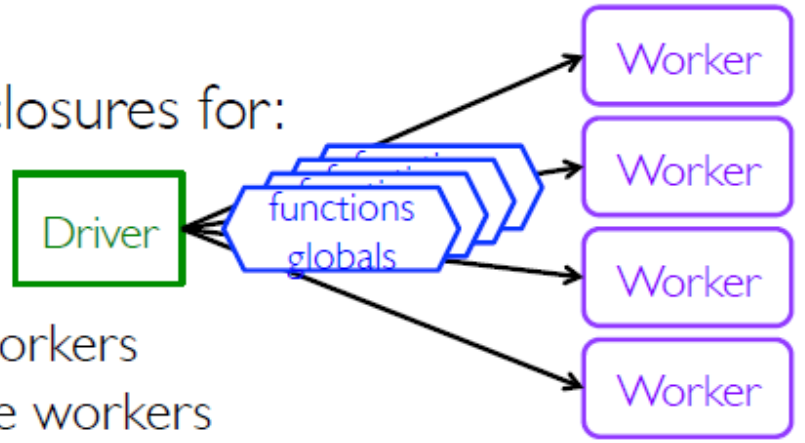
- Small anonymous functions (not bound to a name)
  `lambda a, b: a + b`
  » returns the sum of its two arguments
- Can use lambda functions wherever function objects are required
- Restricted to a single expression

Review: Python `lambda` Functions

- Small anonymous functions (not bound to a name)
  `lambda a, b: a + b`
  » returns the sum of its two arguments
- Can use lambda functions wherever function objects are required
- Restricted to a single expression

pySpark Closures

- Spark automatically creates closures for:
  - » Functions that run on RDDs at workers
  - » Any global variables used by those workers



- One closure per worker
  - » Sent for *every* task
  - » No communication between workers
  - » Changes to global variables at workers are not sent to driver

Consider These Use Cases

- Iterative or single jobs with large global variables
  - » Sending large read-only lookup table to workers
  - » Sending large feature vector in a ML algorithm to workers
- Counting events that occur during job execution
  - » How many input lines were blank?
  - » How many input records were corrupt?

Consider These Use Cases

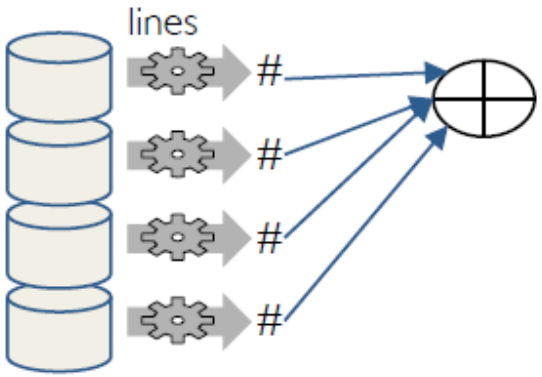
- Iterative or single jobs with large global variables
  - » Sending large read-only lookup table to workers
  - » Sending large feature vector in a ML algorithm to workers
- Counting events that occur during job execution
  - » How many input lines were blank?
  - » How many input records were corrupt?

Problems:

- Closures are (re-)sent with *every* job
- Inefficient to send large data to each worker
- Closures are one way: driver ➔ worker

Spark Programming Model

```
lines = sc.textFile("...", 4)

print lines.count()
```

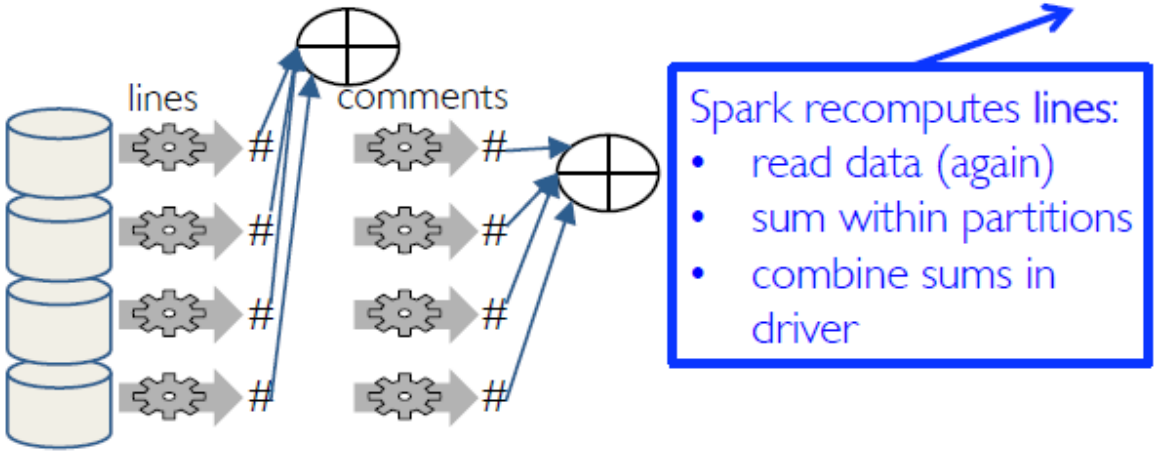


`count()` causes Spark to:

- read data
- sum within partitions
- combine sums in driver

Spark Programming Model

```
lines = sc.textFile("...", 4)
comments = lines.filter(isComment)
print lines.count(), comments.count()
```



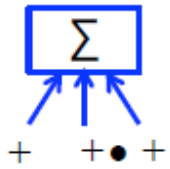
Spark recomputes lines:

- read data (again)
- sum within partitions
- combine sums in driver

pySpark Shared Variables

Broadcast Variables

» Efficiently send large, *read-only* value to all workers
» Saved at workers for use in one or more Spark operations
» Like sending a large, read-only lookup table to all the nodes



Accumulators

» Aggregate values from workers back to driver
» Only driver can access value of accumulator
» For tasks, accumulators are write-only
» Use to count errors seen in RDD across workers

Broadcast Variables Example

Country code lookup for HAM radio call signs

```
# Lookup the locations of the call signs on the
# RDD contactCounts. We load a list of call sign
# prefixes to country code to support this lookup
signPrefixes = loadCallSignTable()

def processSignCount(sign_count, signPrefixes):
    country = lookupCountry(sign_count[0], signPrefixes)
    count = sign_count[1]
    return (country, count)

countryContactCounts = (contactCounts
                        .map(processSignCount)
                        .reduceByKey((lambda x, y: x+ y)))
```

Expensive to send large table
(Re-)sent for every processed file

From: http://shop.oreilly.com/product/0636920028512.do

Broadcast Variables

- Keep *read-only* variable cached on workers
  - » Ship to each worker only once instead of with each task
- Example: efficiently give every worker a large dataset
- Usually distributed using efficient broadcast algorithms

```
At the driver:
>>> broadcastVar = sc.broadcast([1, 2, 3])

At a worker (in code passed via a closure)
>>> broadcastVar.value
[1, 2, 3]
```

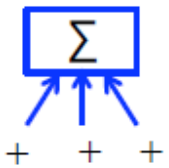
Broadcast Variables Example

Country code lookup for HAM radio call signs

```
# Lookup the locations of the call signs on the
# RDD contactCounts. We load a list of call sign
# prefixes to country code to support this lookup
signPrefixes = sc.broadcast(loadCallSignTable())

def processSignCount(sign_count, signPrefixes):
    country = lookupCountry(sign_count[0], signPrefixes.value)
    count = sign_count[1]
    return (country, count)

countryContactCounts = (contactCounts
                        .map(processSignCount)
                        .reduceByKey((lambda x, y: x+ y)))
```

Efficiently sent once to workers

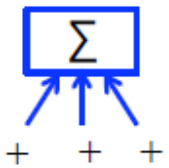
From: http://shop.oreilly.com/product/0636920028512.do



Accumulators Example

- Counting empty lines

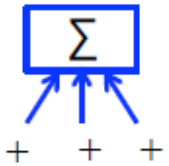
```
file = sc.textFile(inputFile)
# Create Accumulator[Int] initialized to 0
blankLines = sc.accumulator(0)

def extractCallSigns(line):
    global blankLines # Make the global variable accessible
    if (line == ""):
        blankLines += 1
    return line.split(" ")

callSigns = file.flatMap(extractCallSigns)
print "Blank lines: %d" % blankLines.value
```



Accumulators

- Variables that can only be "added" to by associative op
- Used to efficiently implement parallel counters and sums
- Only driver can read an accumulator's value, not tasks

```
>>> accum = sc.accumulator(0)
>>> rdd = sc.parallelize([1, 2, 3, 4])
>>> def f(x):
>>>   global accum
>>>   accum += x

>>> rdd.foreach(f)
>>> accum.value
Value: 10
```



Accumulators

- Tasks at workers cannot access accumulator's values
- Tasks see accumulators as write-only variables
- Accumulators can be used in actions or transformations:
  - » Actions: each task's update to accumulator is *applied only once*
  - » Transformations: *no guarantees* (use only for debugging)
- Types: integers, double, long, float
  - » See lab for example of custom type