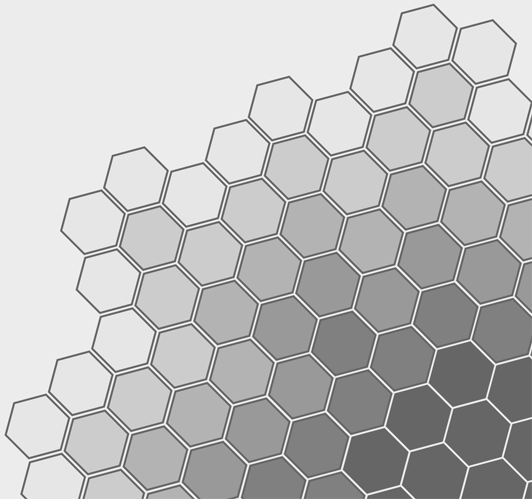
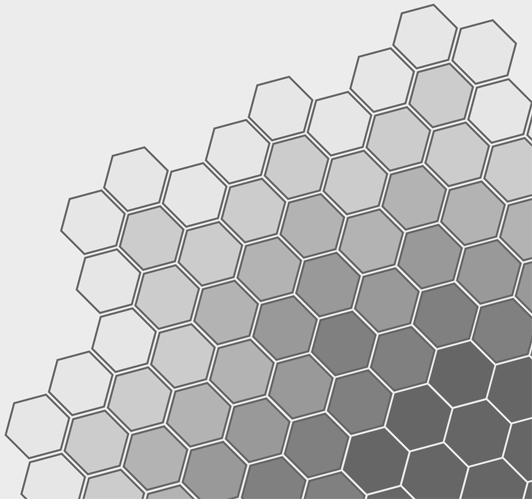
Fast Data Analytics with Spark and Python (PySpark)

District Data Labs



Plan of Study

- Installing Spark
- What is Spark?
- The PySpark interpreter
- Resilient Distributed Datasets
- Writing a Spark Application
- Beyond RDDs
- The Spark libraries
- Running Spark on EC2



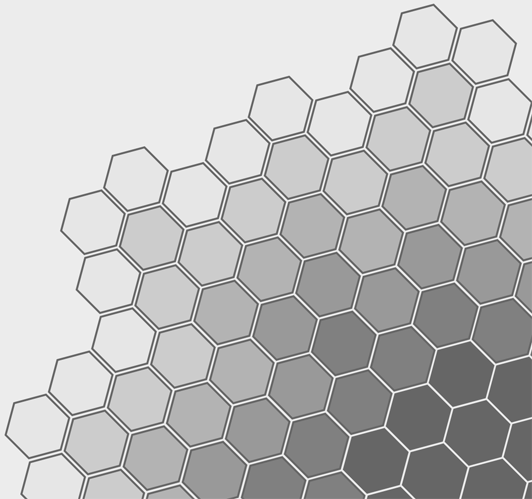
Installing Spark

1. Install Java JDK 7 or 8
2. Set JAVA_HOME environment variable
3. Install Python 2.7
4. Download Spark

Done!

Note: to build you need Maven

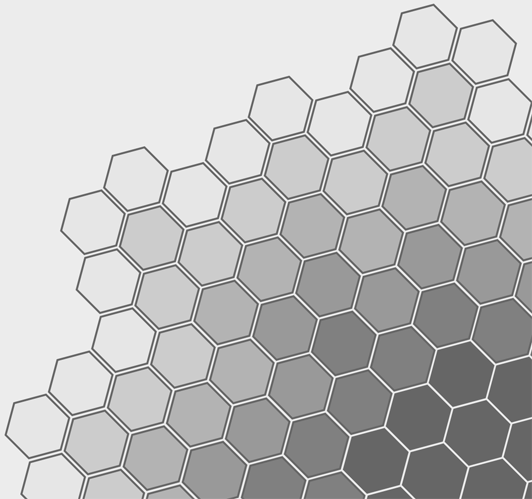
Also you might want Scala 2.11



Managing Services

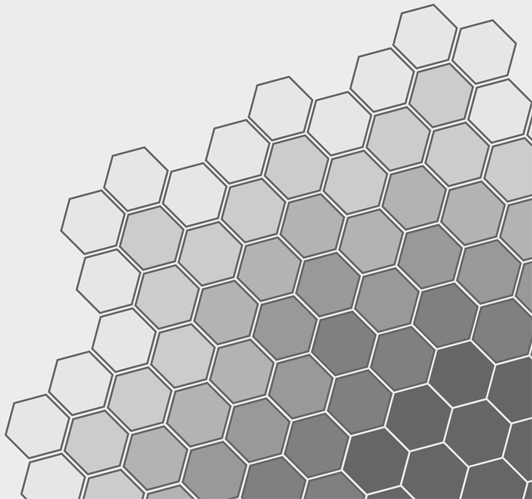
Often you'll be developing and have Hive, Titan, HBase, etc. on your local machine. Keep them in one place as follows:

```
[srv]
  |--- spark-1.2.0
  |--- spark → [srv]/spark-1.2.0
  |--- titan
 ...

export SPARK_HOME=/srv/spark
export PATH=$SPARK_HOME/bin:$PATH
```



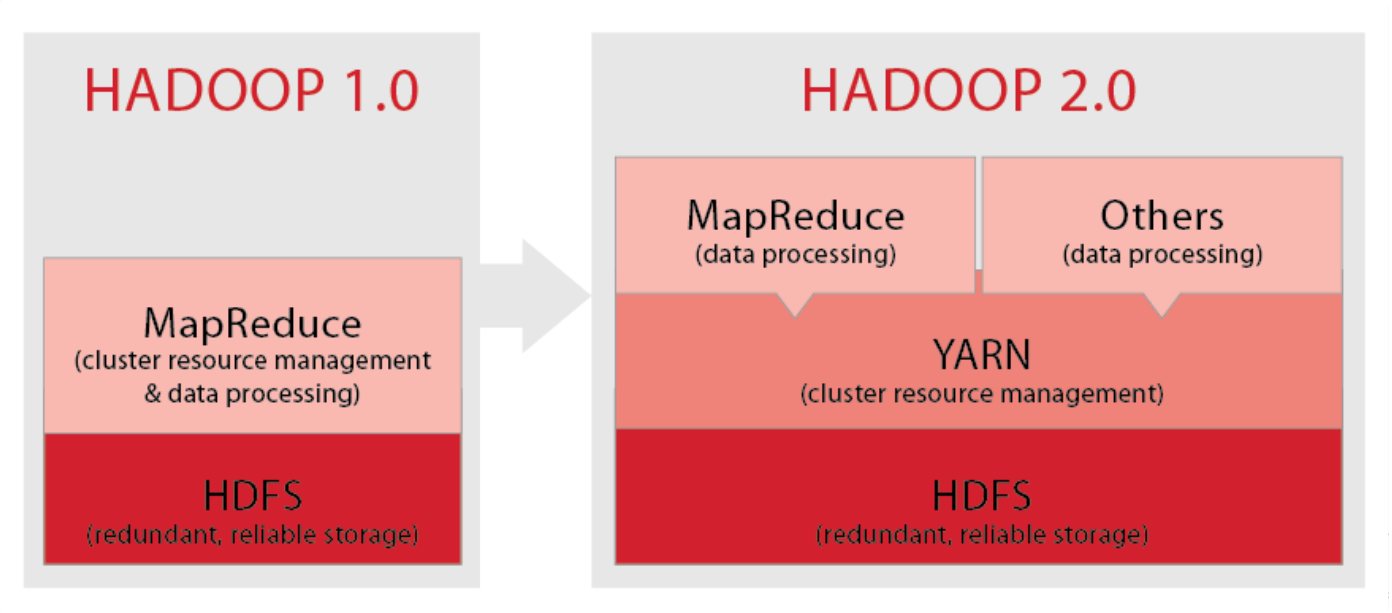
Is that too easy? No daemons to configure no web hosts?

What is Spark?



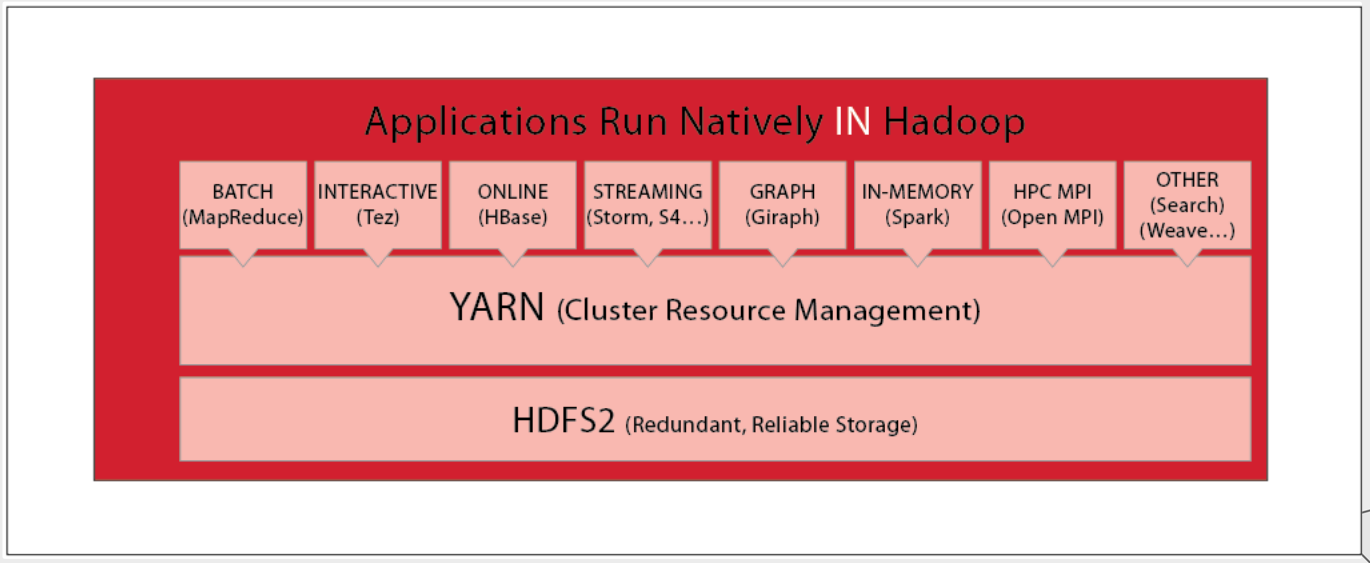
Hadoop 2 and YARN

YARN is the resource management and computation framework that is new as of Hadoop 2, which was released late in 2013.



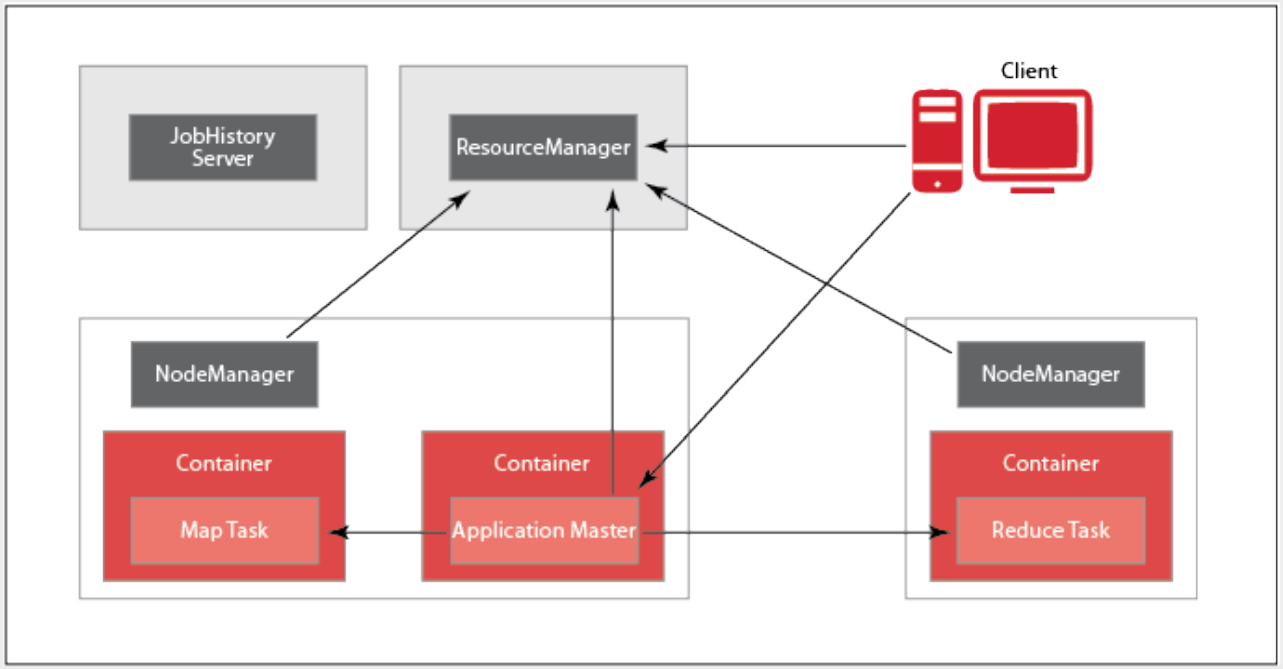
Hadoop 2 and YARN

YARN supports multiple processing models in addition to MapReduce. All share common resource management service.



YARN Daemons

Resource Manager (RM) - serves as the central agent for managing and allocating cluster resources. **Node Manager (NM)** - per node agent that manages and enforces node resources. **Application Master (AM)** - per application manager that manages lifecycle and task scheduling

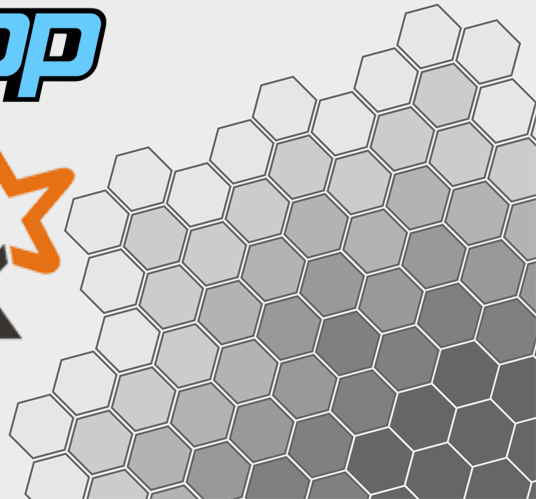


Spark on a Cluster

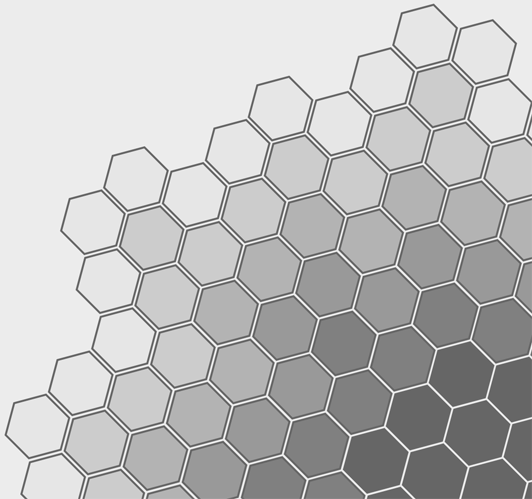
- Amazon EC2 (prepared deployment)
- Standalone Mode (private cluster)
- Apache Mesos
- Hadoop YARN

MESOS

EC2



Spark is a *fast* and *general-purpose* cluster ***computing framework*** (like MapReduce) that has been implemented to run on a resource managed cluster of servers

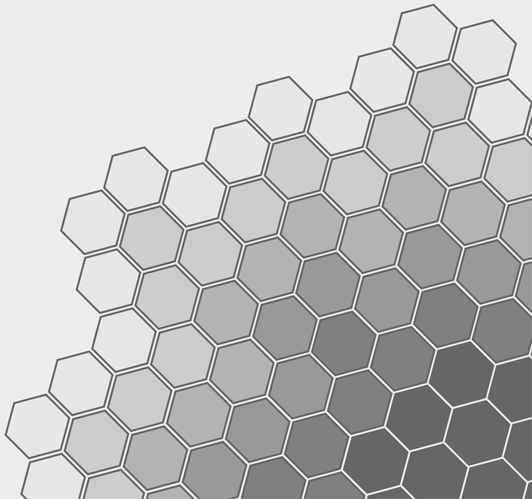


Motivation for Spark

MapReduce has been around as the major framework for distributed computing for 10 years - this is pretty old in technology time! Well known limitations include:

1. Programmability
    a. Requires multiple chained MR steps
    b. *Specialized* systems for applications
2. Performance
    a. Writes to disk between each computational step
    b. Expensive for apps to "reuse" data
        i. Iterative algorithms
        ii. Interactive analysis

Most machine learning algorithms are iterative …



Motivation for Spark

Computation frameworks are becoming *specialized* to solve problems with MapReduce

All of these systems present “data flow” models, which can be represented as a directed acyclical graph.

[The State of Spark and Where We’re Going Next]

Matei Zaharia (Spark Summit 2013, San Francisco)

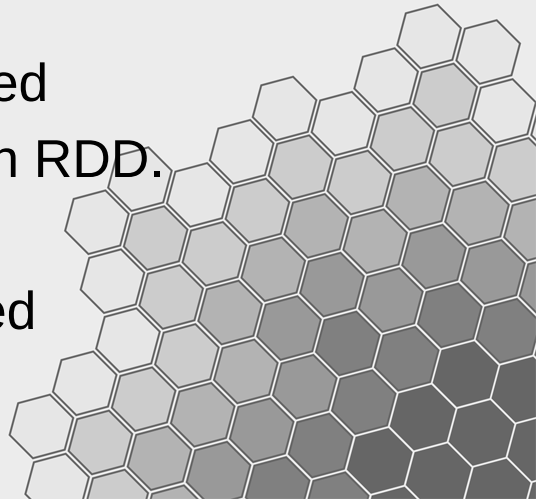
Generalizing Computation

Programming Spark applications takes lessons from other higher order data flow languages learned from Hadoop. Distributed computations are defined in code on a driver machine, then lazily evaluated and executed across the cluster. APIs include:

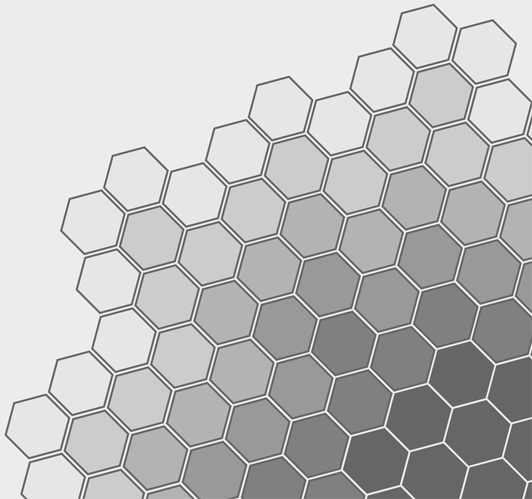
- Java
- Scala
- Python

Under the hood, Spark (written in Scala) is an optimized engine that supports general execution graphs over an RDD.

Note, however - that Spark doesn’t deal with distributed storage, it still relies on HDFS, S3, HBase, etc.



PySpark Practicum
(more show, less tell)



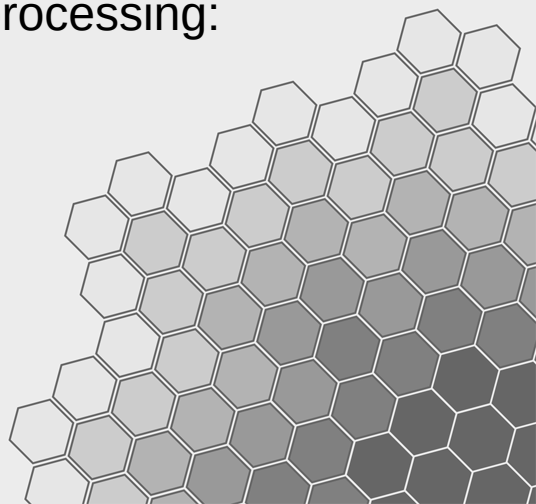
Word Frequency

count how often a word appears in a document or collection of documents (corpus).

Is the “canary” of Big Data/Distributed computing because a distributed computing framework that can run WordCount efficiently in parallel at scale can likely handle much larger and more interesting compute problems - Paco Nathan

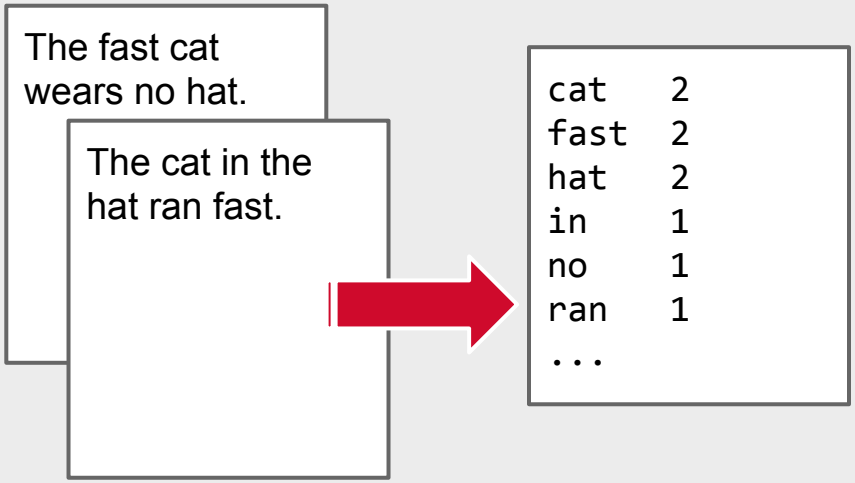
This simple program provides a good test case for parallel processing:

- requires a minimal amount of code
- demonstrates use of both symbolic and numeric values
- isn’t many steps away from search indexing/statistics



Word Frequency

```
def map(key, value):
    for word in value.split():
        emit(word, 1)

def reduce(key, values):
    count = 0
    for val in values:
        count += val
    emit(key, count)

# emit is a function that performs distributed I/O
```

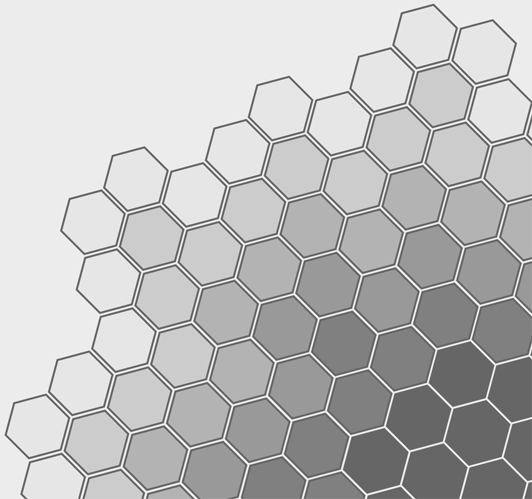


Each document is passed to a mapper, which does the tokenization. The output of the mapper is reduced by key (word) and then counted.

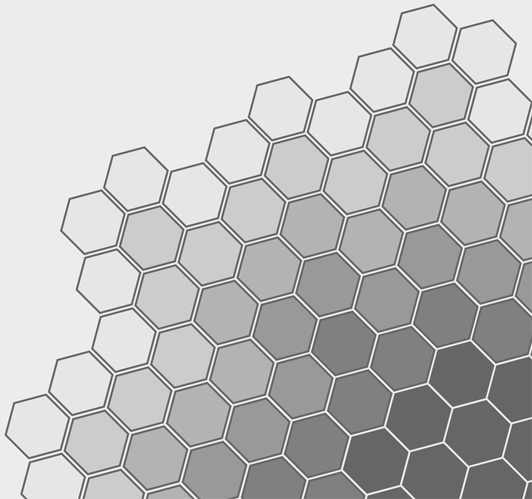
What is the data flow for word count?

Word Frequency

```
from operator import add

def tokenize(text):
    return text.split()

text = sc.textFile("tolstoy.txt")   # Create RDD

# Transform
wc   = text.flatMap(tokenize)
wc   = wc.map(lambda x: (x,1)).reduceByKey(add)

wc.saveAsTextFile("counts")         # Action
```



Resilient Distributed Datasets



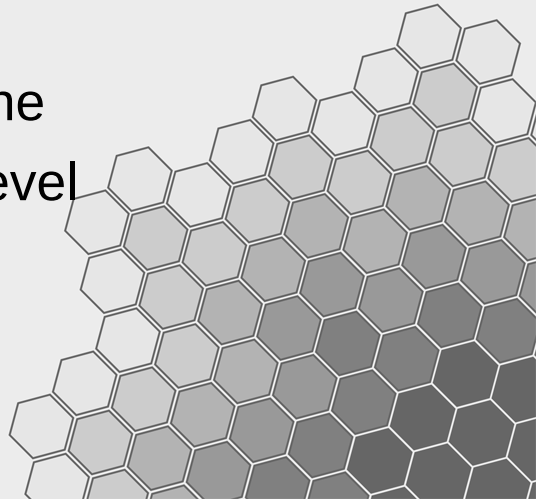
Science (and History)

Like MapReduce + GFS, Spark is based on two important papers authored by Matei Zaharia and the Berkeley AMPLab.

M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "*Spark: cluster computing with working sets*," in Proceedings of the 2nd USENIX conference on Hot topics in cloud computing, 2010, pp. 10–10.

M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "*Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing*," in Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation, 2012, pp. 2–2.

Matei is now the CTO and co-founder of Databricks, the corporate sponsor of Spark (which is an Apache top level open source project).

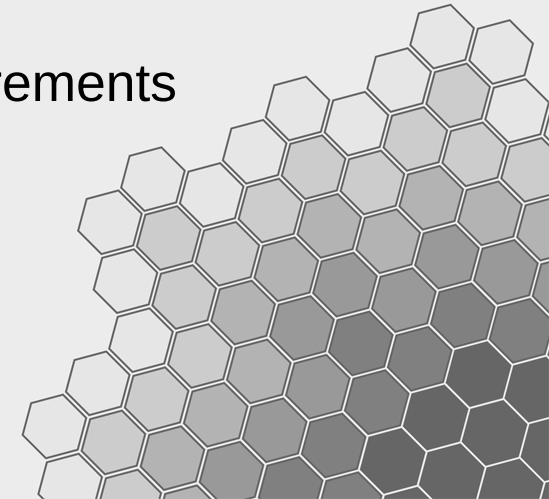


The Key Idea: RDDs

The principle behind Spark's framework is the idea of RDDs - an abstraction that represents a read-only collection of objects that are partitioned across a set of machines. RDDs can be:

1. Rebuilt from lineage (fault tolerance)
2. Accessed via MapReduce-like (functional) parallel operations
3. Cached in memory for immediate reuse
4. Written to distributed storage

These properties of RDDs all meet the Hadoop requirements for a distributed computation framework.



Working with RDDs

Most people focus on the in-memory caching of RDDs, which is great because it allows for:

- batch analyses (like MapReduce)
- interactive analyses (humans exploring Big Data)
- iterative analyses (no expensive Disk I/O)
- real time processing (just “append” to the collection)

However, RDDs also provide a more general interaction with functional constructs at a higher level of abstraction: *not just MapReduce*!

Spark Metrics

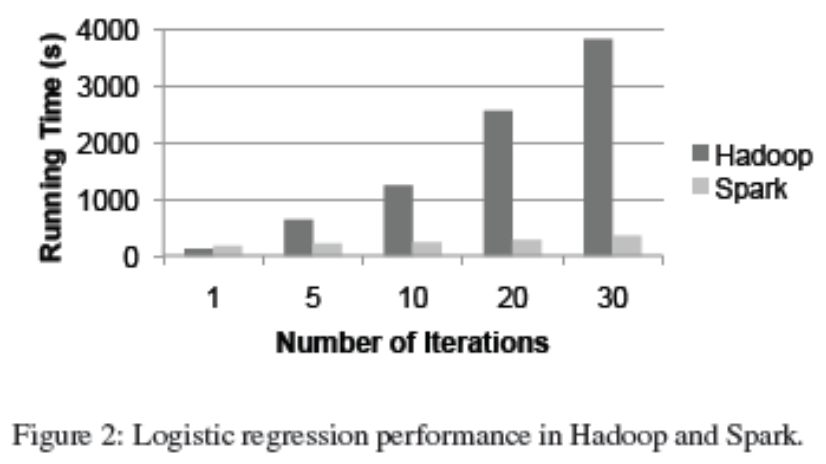
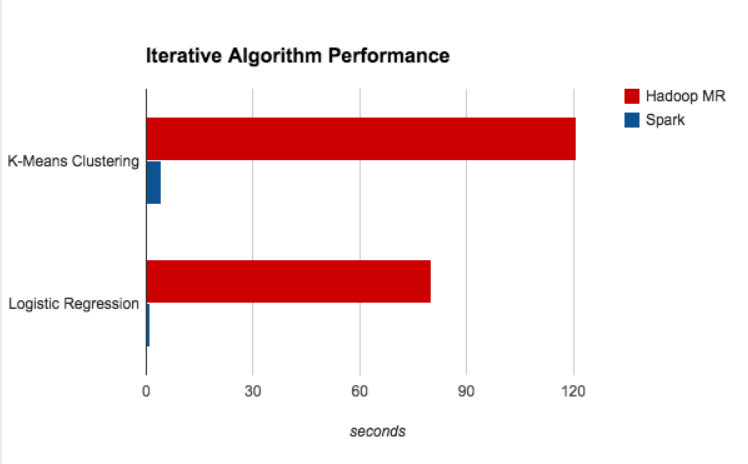
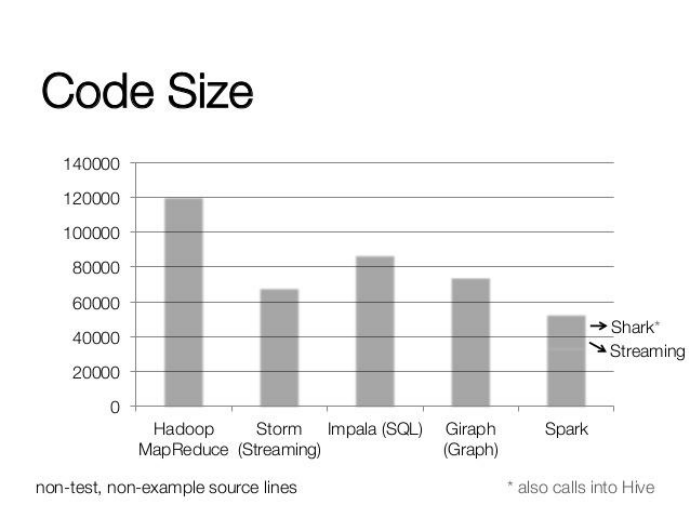


Figure 2: Logistic regression performance in Hadoop and Spark.





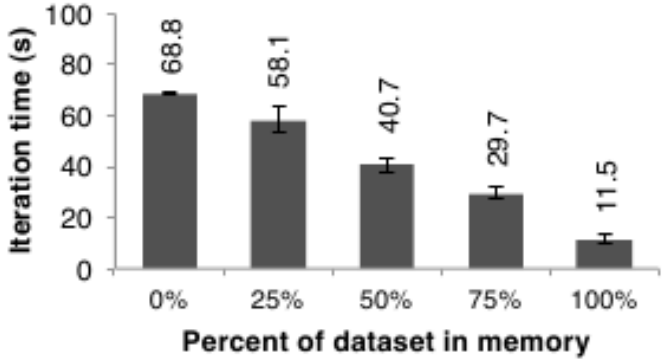


Figure 12: Performance of logistic regression using 100 GB data on 25 machines with varying amounts of data in memory.

Programming Spark

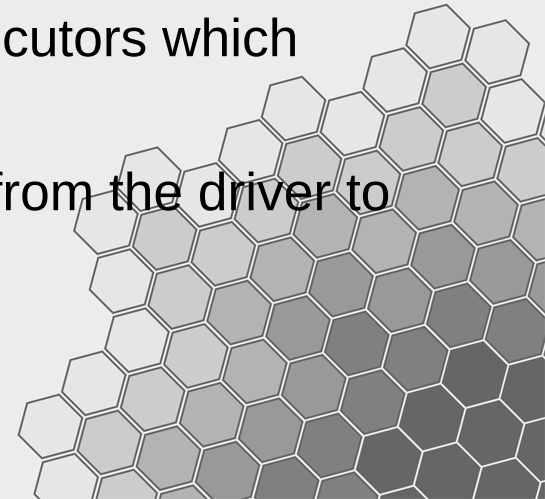
Create a driver program (app.py) that does the following:

1. Define one or more RDDs either through accessing data stored on disk (HDFS, Cassandra, HBase, Local Disk), parallelizing some collection in memory, *transforming* an existing RDD or by *caching* or *saving*.
2. Invoke *operations* on the RDD by passing *closures* (functions) to each element of the RDD. Spark offers over 80 high level operators beyond Map and Reduce.
3. Use the resulting RDDs with *actions* e.g. count, collect, save, etc. Actions kick off the computing on the cluster, not before.

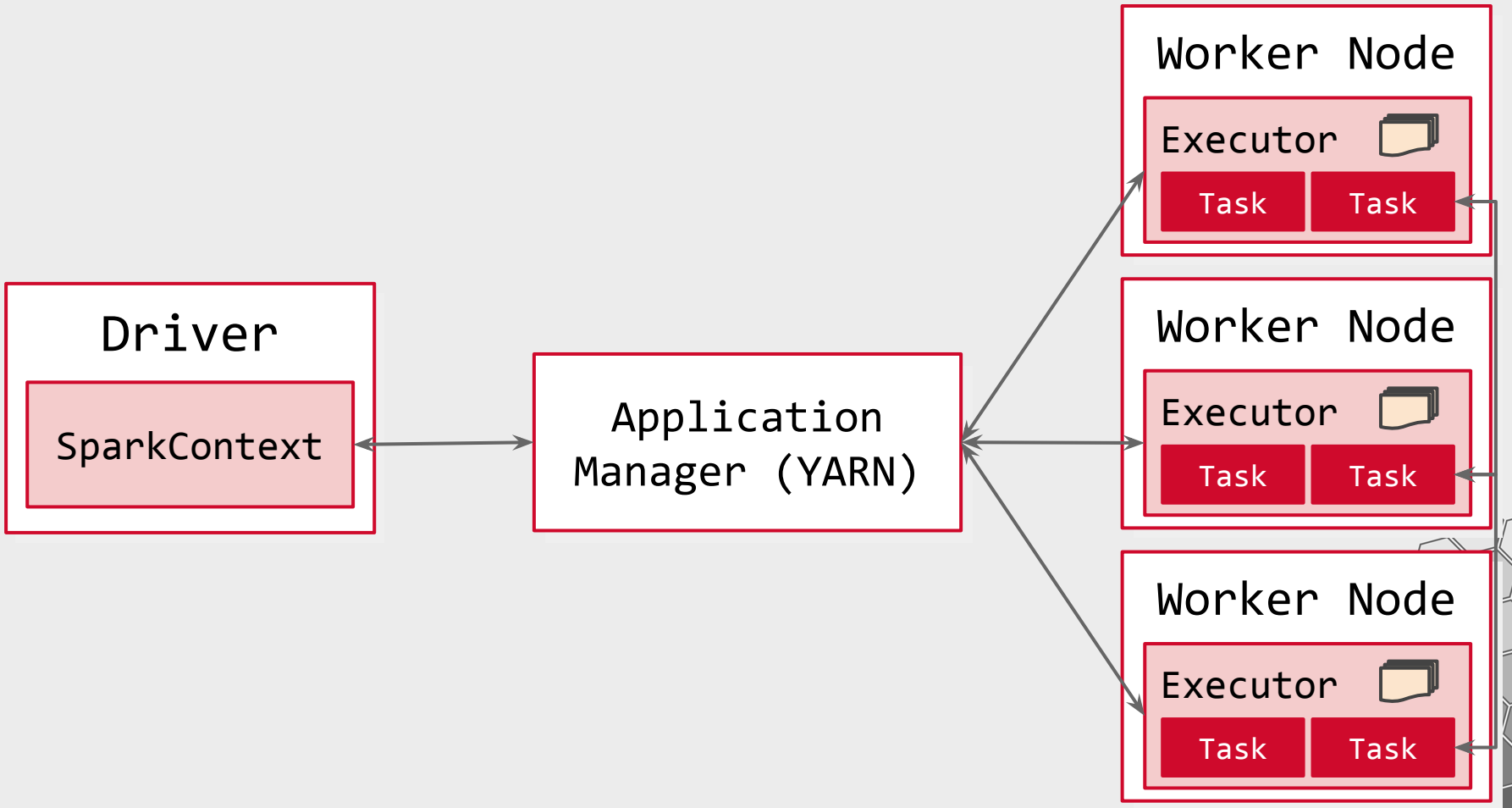
More details on this soon!

Spark Execution

- Spark applications are run as independent *sets* of processes
- Coordination is by a `SparkContext` in a *driver program*.
- The context connects to a cluster manager which allocates computational resources.
- Spark then acquires *executors* on individual nodes on the cluster.
- Executors manage individual worker computations as well as manage the storage and caching of data.
- Application code is sent from the driver to the executors which specifies the context and the *tasks* to be run.
- Communication can occur between workers and from the driver to the worker.

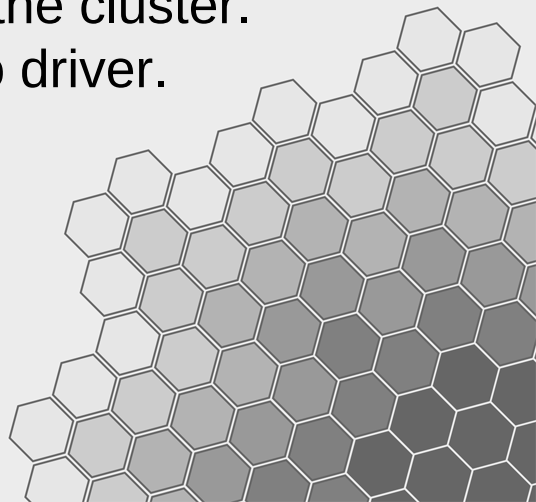


Spark Execution



Key Points regarding Execution

1. Each application gets its own executor for the duration.
2. Tasks run in multiple threads or processes.
3. Data can be shared between executors, but not between different Spark applications without external storage.
4. The Application Manager can be anything - Yarn on Hadoop, Mesos or Spark Standalone. Spark handles most of the resource scheduling.
5. Drivers are key participants in a Spark applications; therefore drivers should be on the same local network with the cluster.
6. Remote cluster access should use RPC access to driver.



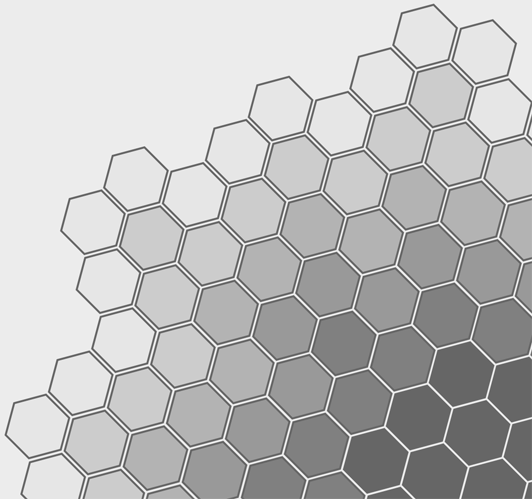
Executing Spark Jobs

Use the `spark-submit` command to send your application to the cluster for execution along with any other Python files and dependencies.

```
# Run on a YARN cluster
export HADOOP_CONF_DIR=XXX
/srv/spark/bin/spark-submit \
  --master yarn-cluster \
  --executor-memory 20G \
  --num-executors 50    \
  --py-files mydeps.egg
  app.py
```

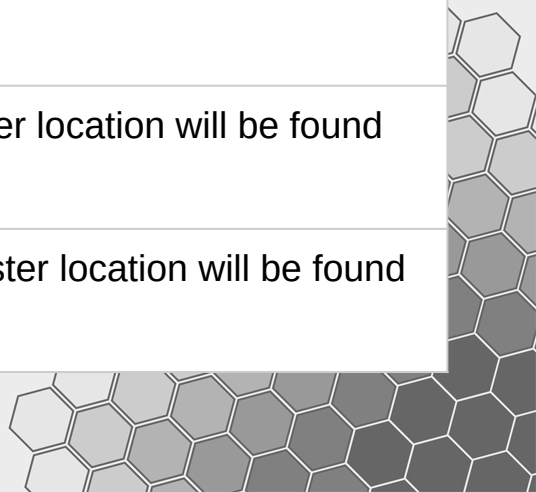
This will cause Spark to allow the driver program to acquire a Context that utilizes the YARN ResourceManager.

You can also specify many of these arguments in your driver program when constructing a `SparkContext`.



The Spark Master URL

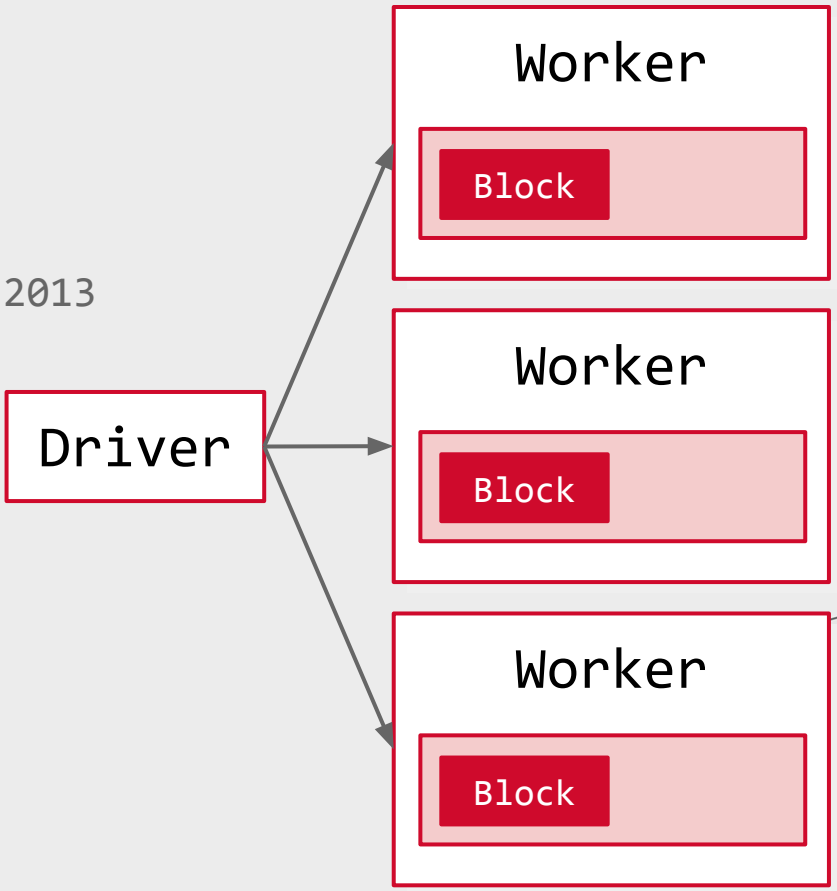
Master URL	Meaning
local	Run Spark locally with one worker thread (i.e. no parallelism at all).
local[K]	Run Spark locally with K worker threads (ideally, set this to the number of cores on your machine).
local[*]	Run Spark locally with as many worker threads as logical cores on your machine.
spark://HOST:PORT	Connect to the given Spark standalone cluster master. The port must be whichever one your master is configured to use, which is 7077 by default.
mesos://HOST:PORT	Connect to the given Mesos cluster. The port must be whichever one your is configured to use, which is 5050 by default. Or, for a Mesos cluster using ZooKeeper, use mesos://zk://....
yarn-client	Connect to a YARN cluster in client mode. The cluster location will be found based on the HADOOP_CONF_DIR variable.
yarn-cluster	Connect to a YARN cluster in cluster mode. The cluster location will be found based on HADOOP_CONF_DIR.



Example Data Flow

```
# Base RDD
orders = sc.textFile("hdfs://...")
orders = orders.map(split).map(parse)
orders = orders.filter(
    lambda order: order.date.year == 2013
)
orders.cache()
```

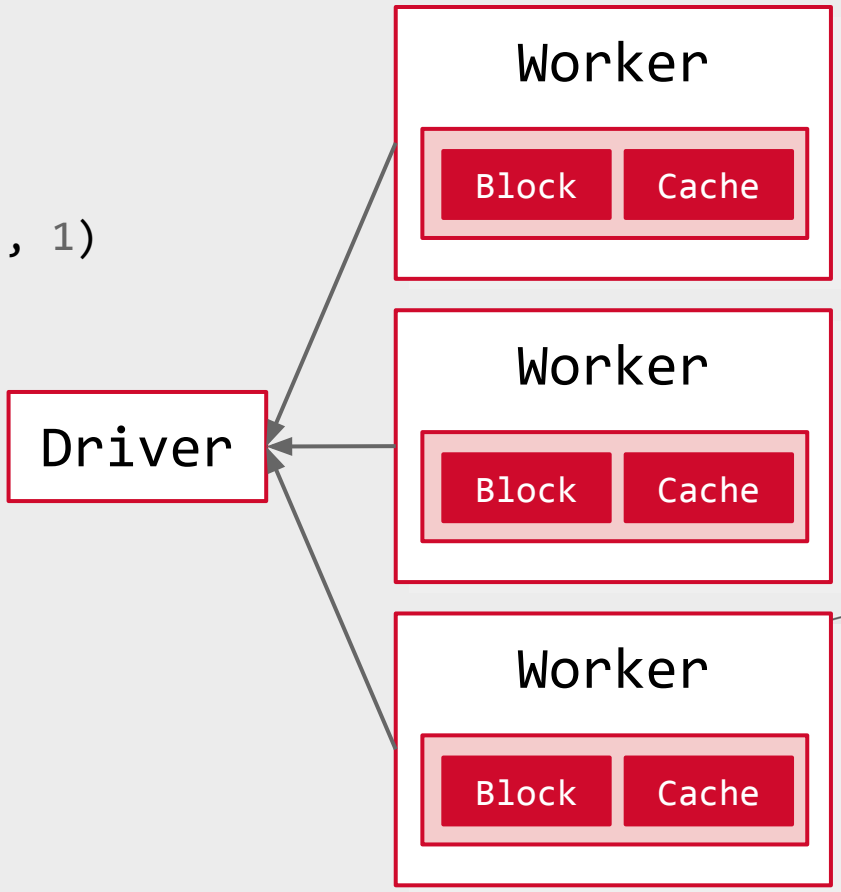
1. Read Block from HDFS
2. Process RDDs
3. Cache the data in memory



Example Data Flow

```
months = orders.map(
    lambda order: ((order.date.year,
                    order.date.month), 1)
    )

months = months.reduceByKey(add)
print months.take(5)
```

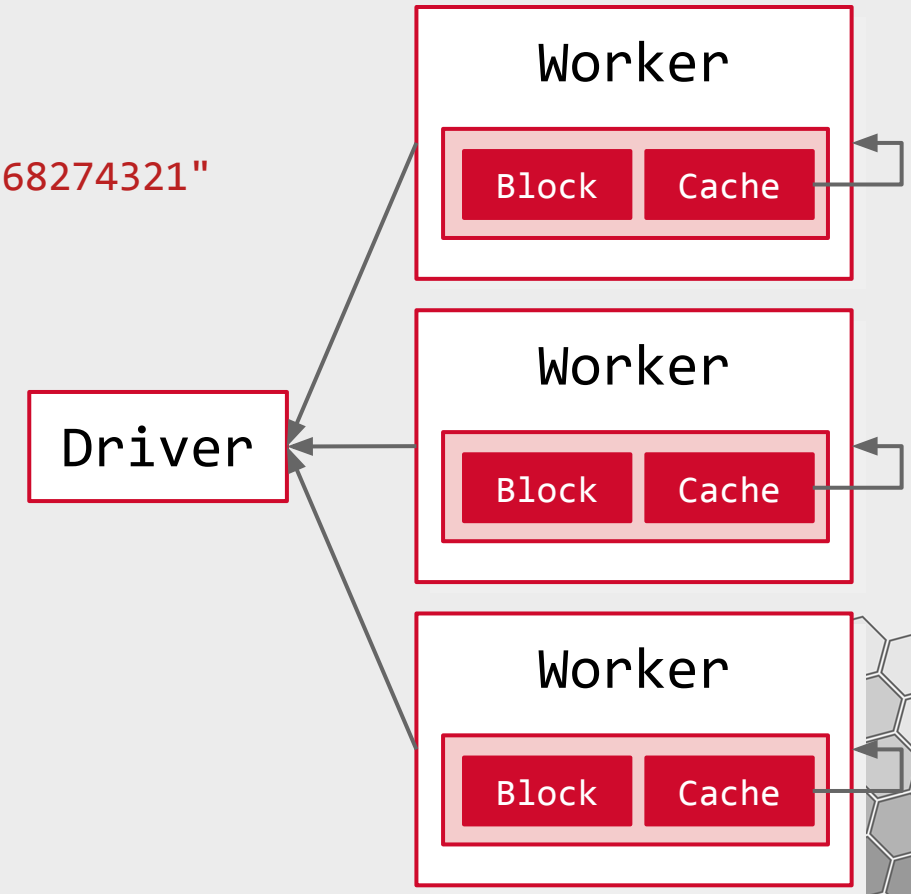
1. Process final RDD
2. On action send result back to driver
3. Driver outputs result (print)



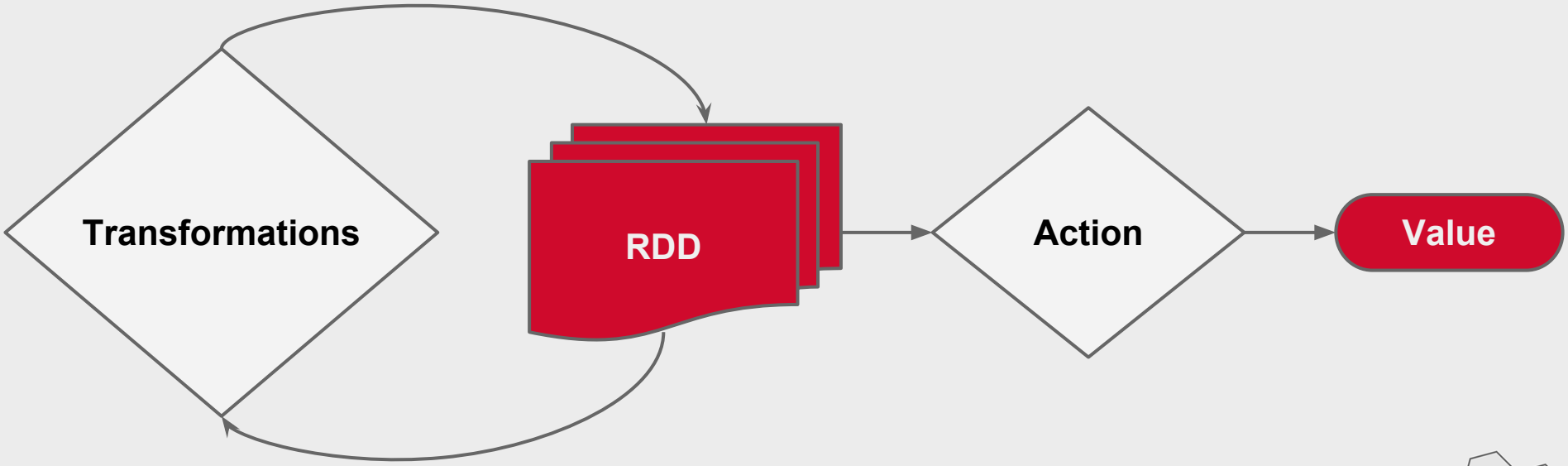
Example Data Flow

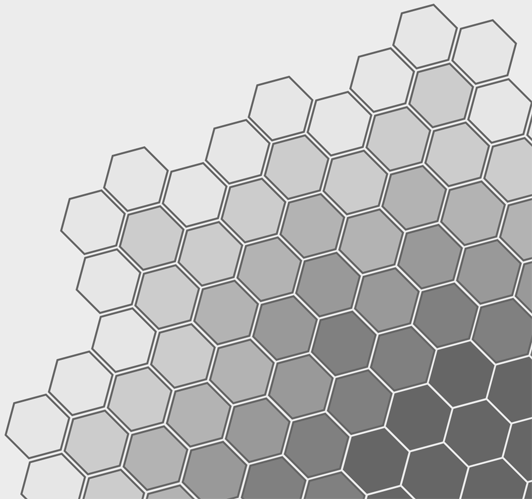
```
products = orders.filter(
    lambda order: order.upc == "098668274321"
)

print products.count()
```

1. Process RDD from cache
2. Send data on action back to driver
3. Driver outputs result (print)



Spark Data Flow

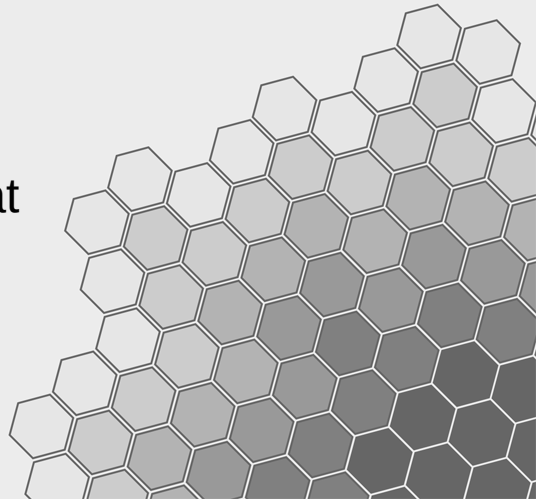




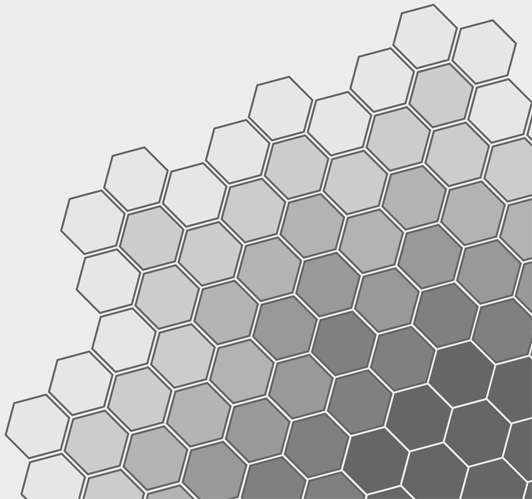
Debugging Data Flow

```
>>> print months.toDebugString()

(9) PythonRDD[9] at RDD at PythonRDD.scala:43
 |  MappedRDD[8] at values at NativeMethodAccessorImpl.java:-2
 |  ShuffledRDD[7] at partitionBy at NativeMethodAccessorImpl.java:-2
 +-(9) PairwiseRDD[6] at RDD at PythonRDD.scala:261
    |  PythonRDD[5] at RDD at PythonRDD.scala:43
    |  PythonRDD[2] at RDD at PythonRDD.scala:43
    |  orders.csv MappedRDD[1] at textFile
    |  orders.csv HadoopRDD[0] at textFile
```

Operator Graphs and Lineage can be shown with the `toDebugString` method, allowing a visual inspection of what is happening under the hood.



Writing Spark Applications



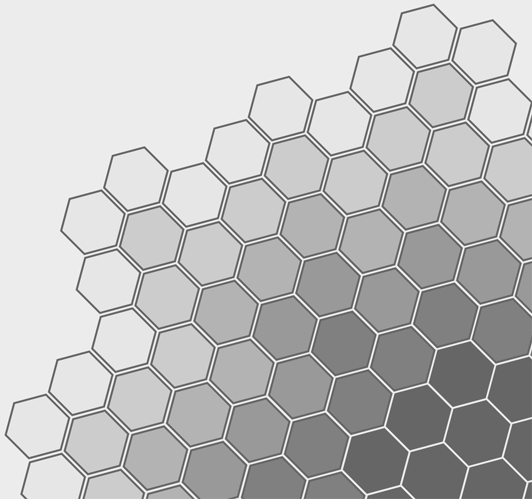
Creating a Spark Application

Writing a Spark application in Java, Scala, or Python is similar to using the interactive console - the API is the same. All you need to do first is to get access to the `SparkContext` that was loaded automatically for you by the interpreter.

```
from pyspark import SparkConf, SparkContext
conf = SparkConf().setAppName("MyApp")
sc = SparkContext(conf=conf)
```

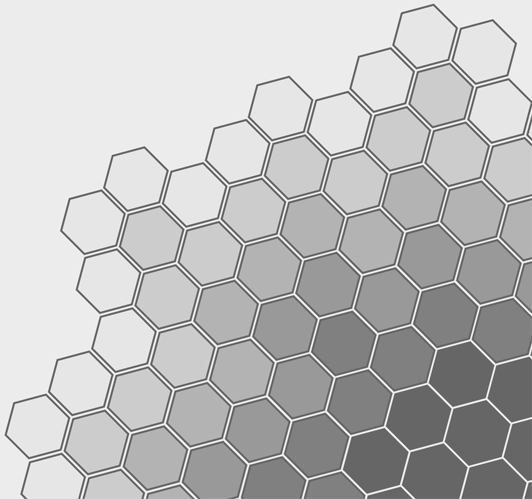
To shut down Spark:

```
sc.stop() or sys.exit(0)
```



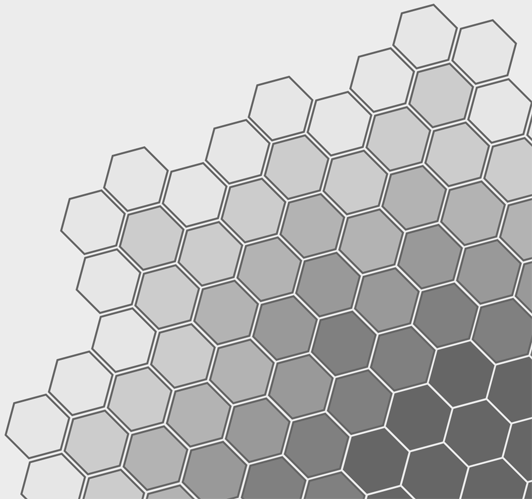
Structure of a Spark Application

- Dependencies (import)
  - third party dependencies can be shipped with app
- Constants and Structures
  - especially `namedtuples` and other constants
- Closures
  - functions that operate on the RDD
- A main method
  - Creates a SparkContext
  - Creates one or more RDDs
  - Applies *transformations* to RDDs
  - Applies *actions* to kick off computation



```
## Spark Application - execute with spark-submit

## Imports
from pyspark import SparkConf, SparkContext

## Module Constants
APP_NAME = "My Spark Application"

## Closure Functions

## Main functionality
def main(sc):
    pass

if __name__ == "__main__":
    # Configure Spark
    conf = SparkConf().setAppName(APP_NAME)
    sc   = SparkContext(conf=conf)

    # Execute Main functionality
    main(sc)
```

A Spark Application Skeleton



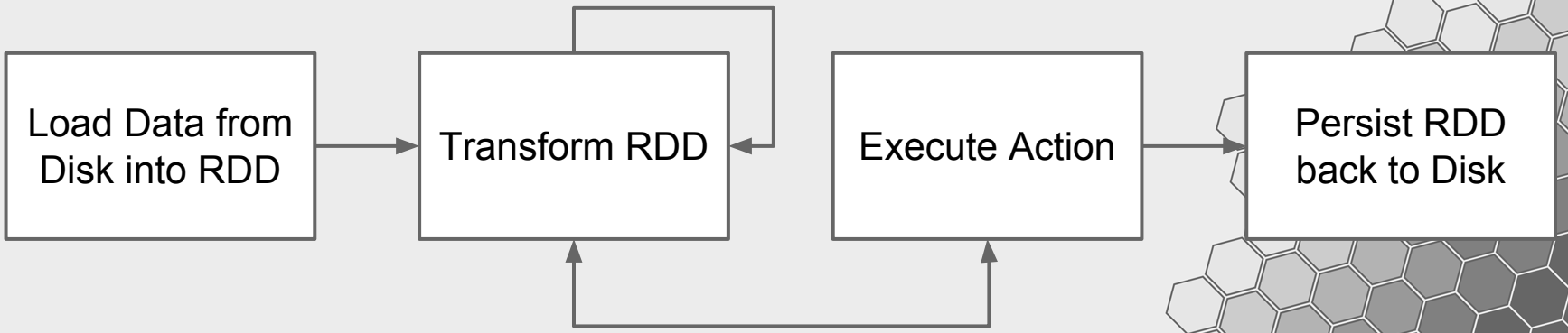
Programming Model

Two types of operations on an RDD:

- *transformations*
- *actions*

Transformations are *lazily* evaluated - they aren't executed when you issue the command.

RDDs are recomputed when an action is executed.



Initializing an RDD

Two types of RDDs:

- *parallelized collections* - take an existing in memory collection (a list or tuple) and run functions upon it in parallel

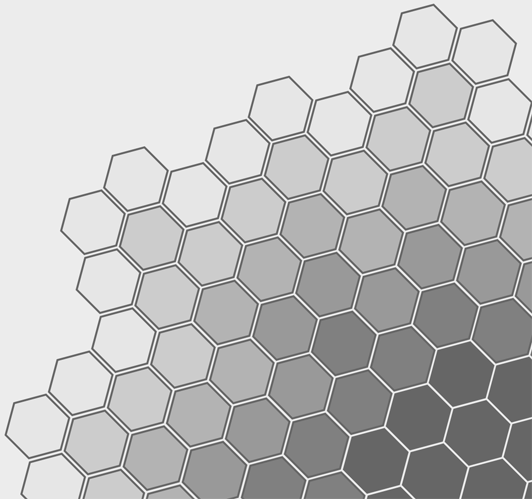
- *Hadoop datasets* - run functions in parallel on any storage system supported by Hadoop (HDFS, S3, HBase, local file system, etc).

Input can be text, `SequenceFiles`, and any other Hadoop `InputFormat` that exists.

Initializing an RDD

```
# Parallelize a list of numbers
distributed_data = sc.parallelize(xrange(100000))

# Load data from a single text file on disk
lines = sc.textFile('tolstoy.txt')

# Load data from all csv files in a directory using glob
files  = sc.wholeTextFiles('dataset/*.csv')

# Load data from S3
data = sc.textFile('s3://databucket/')
```

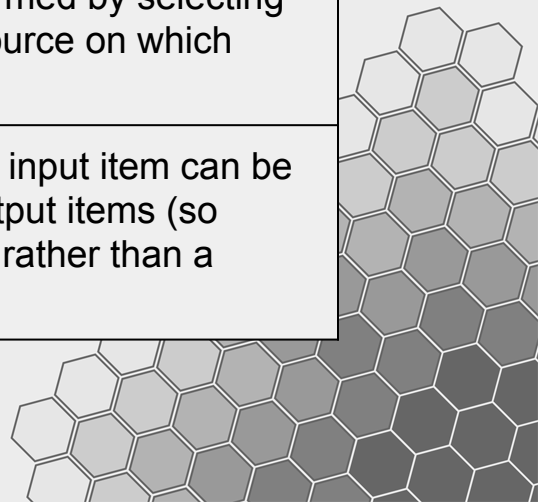
For HBase example, see: [hbase_inputformat.py]



Transformations

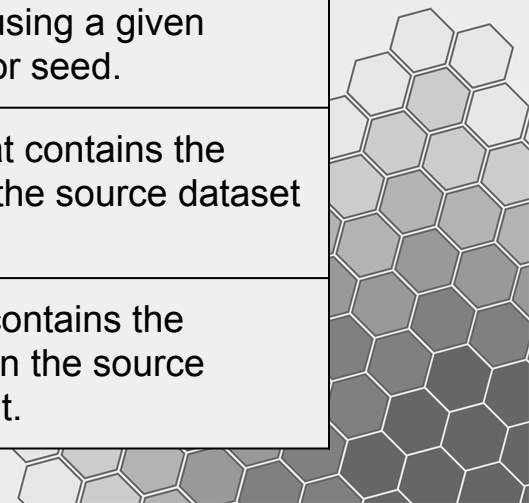
- create a new dataset from an existing one
- evaluated lazily, won’t be executed until action

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 funcreturns true.
`flatMap(func)`	Similar to map, but each input item can be mapped to 0 or more output items (so funcshould return a Seq rather than a single item).



Transformations

Transformation	Description
`mapPartitions(func)`	Similar to map, but runs separately on each partition (block) of the RDD, so func must be of type Iterator<T> => Iterator<U> when running on an RDD of type T.
`mapPartitionsWithIndex(func)`	Similar to mapPartitions, but also provides func with an integer value representing the index of the partition, so func must be of type (Int, Iterator<T>) => Iterator<U> when running on an RDD of type T.
`sample(withReplacement, fraction, seed)`	Sample a fraction fraction of the data, with or without replacement, using a given random number generator seed.
`union(otherDataset)`	Return a new dataset that contains the union of the elements in the source dataset and the argument.
`intersection(otherDataset)`	Return a new RDD that contains the intersection of elements in the source dataset and the argument.

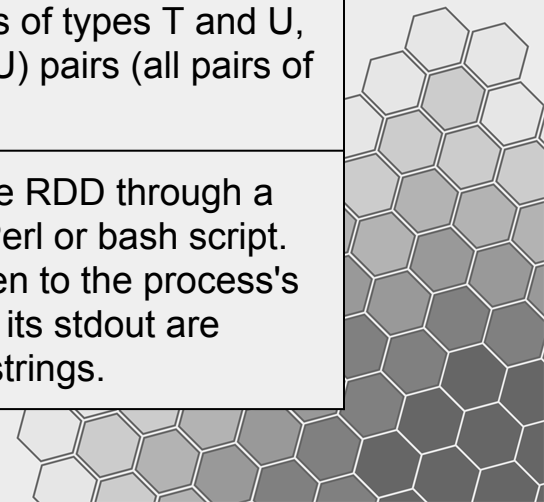


Transformations

Transformation	Description
`distinct([numTasks]))`	Return a new dataset that contains the distinct elements of the source dataset.
`groupByKey([numTasks])`	When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs.
`reduceByKey(func, [numTasks])`	When called on a dataset of (K, V) pairs, returns a 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.
`aggregateByKey(zeroValue)(seqOp, combOp, [numTasks])`	When called on a dataset of (K, V) pairs, returns a dataset of (K, U) pairs where the values for each key are aggregated using the given combine functions and a neutral "zero" value.
`sortByKey([ascending], [numTasks])`	When called on a dataset of (K, V) pairs where K implements Ordered, returns a dataset of (K, V) pairs sorted by keys in ascending or descending order.

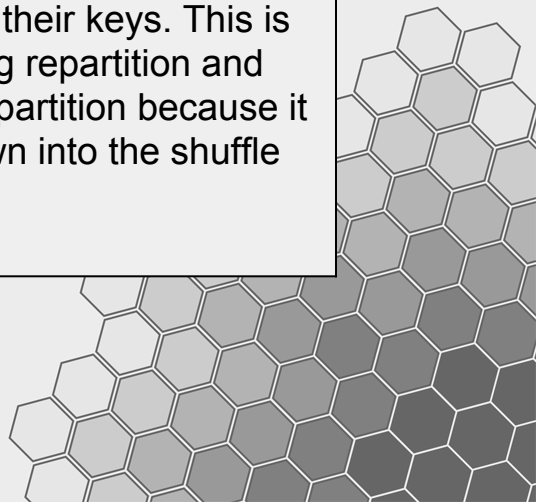
Transformations

Transformation	Description
`join(otherDataset, [numTasks])`	When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key. Outer joins are supported through leftOuterJoin,rightOuterJoin, and fullOuterJoin.
`cogroup(otherDataset, [numTasks])`	When called on datasets of type (K, V) and (K, W), returns a dataset of (K, Iterable<V>, Iterable<W>) tuples. This operation is also called groupWith.
`cartesian(otherDataset)`	When called on datasets of types T and U, returns a dataset of (T, U) pairs (all pairs of elements).
`pipe(command, [envVars])`	Pipe each partition of the RDD through a shell command, e.g. a Perl or bash script. RDD elements are written to the process's stdin and lines output to its stdout are returned as an RDD of strings.



Transformations

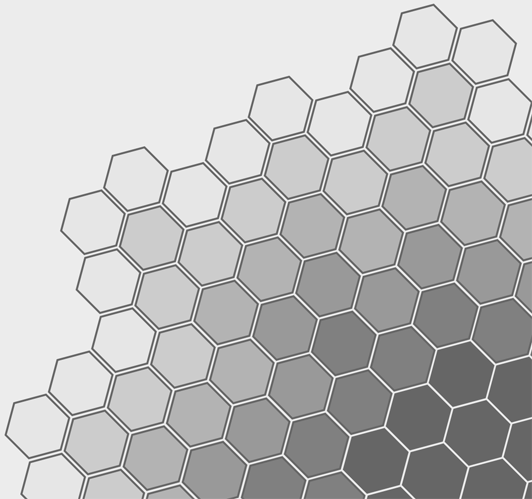
Transformation	Description
`coalesce(numPartitions)`	Decrease the number of partitions in the RDD to numPartitions. Useful for running operations more efficiently after filtering down a large dataset.
`repartition(numPartitions)`	Reshuffle the data in the RDD randomly to create either more or fewer partitions and balance it across them. This always shuffles all data over the network.
`repartitionAndSortWithinPartitions (partitioner)`	Repartition the RDD according to the given partitioner and, within each resulting partition, sort records by their keys. This is more efficient than calling repartition and then sorting within each partition because it can push the sorting down into the shuffle machinery.



Exercise: What is the difference between Map and FlatMap?

```
lines = sc.textFile('fixtures/poem.txt')
lines.map(lambda x: x.split(' ')).collect()
lines.flatMap(lambda x: x.split(' ')).collect()
```

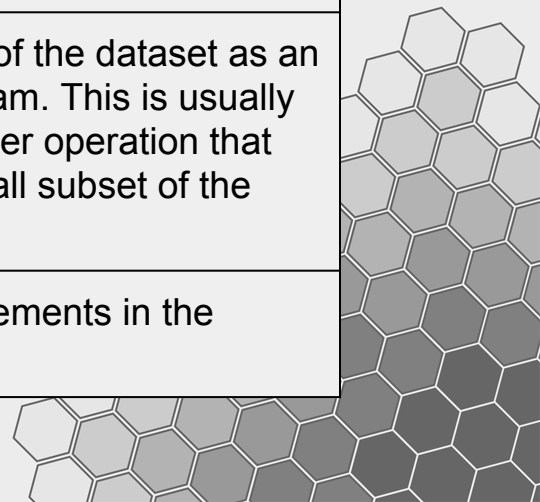
Note the use of closures with the **lambda** keyword



Actions

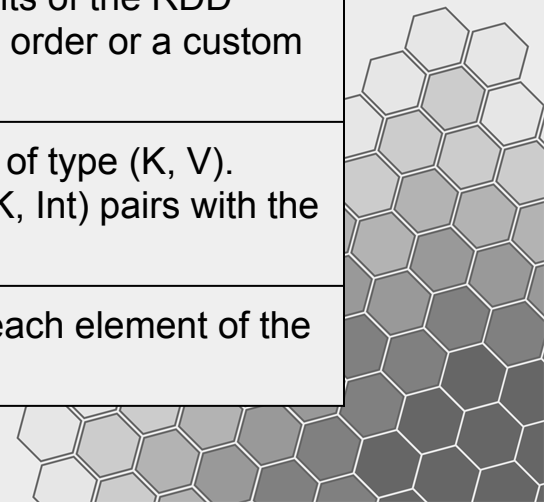
- kick off evaluations and begin computation
- specify the result of an operation or aggregation

Action	Description
`reduce(func)`	Aggregate the elements of the dataset using a function func (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel.
`collect()`	Return all the elements of the dataset as an array at the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data.
`count()`	Return the number of elements in the dataset.



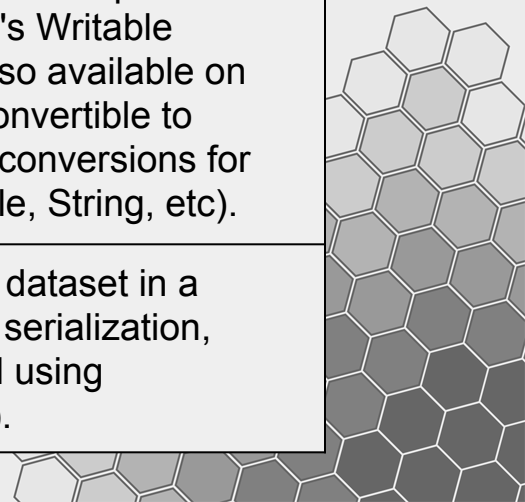
Actions

Action	Description
`first()`	Return the first element of the dataset (similar to take(1)).
`take(n)`	Return an array with the first n elements of the dataset. Note that this is currently not executed in parallel. Instead, the driver program computes all the elements.
`takeSample(withReplacement,num, [seed])`	Return an array with a random sample of num elements of the dataset, with or without replacement.
`takeOrdered(n, [ordering])`	Return the first n elements of the RDD using either their natural order or a custom comparator.
`countByKey()`	Only available on RDDs of type (K, V). Returns a hashmap of (K, Int) pairs with the count of each key.
`foreach(func)`	Run a function func on each element of the dataset.

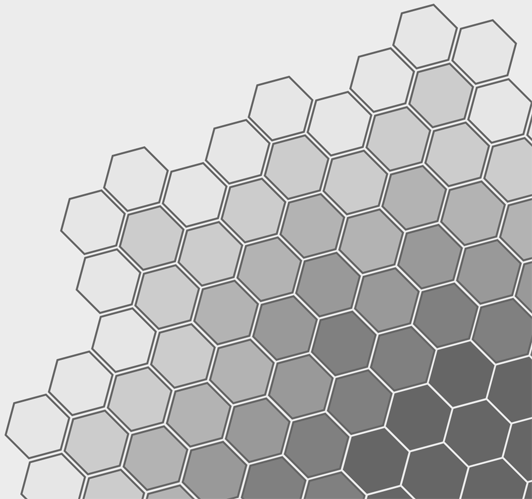


Actions

Action	Description
`saveAsTextFile(path)`	Write the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem, HDFS or any other Hadoop-supported file system. Spark will call toString on each element to convert it to a line of text in the file.
`saveAsSequenceFile(path)` `(Java and Scala)`	Write the elements of the dataset as a Hadoop SequenceFile in a given path in the local filesystem, HDFS or any other Hadoop-supported file system. This is available on RDDs of key-value pairs that either implement Hadoop's Writable interface. In Scala, it is also available on types that are implicitly convertible to Writable (Spark includes conversions for basic types like Int, Double, String, etc).
`saveAsObjectFile(path)` `(Java and Scala)`	Write the elements of the dataset in a simple format using Java serialization, which can then be loaded using SparkContext.objectFile().



Was that comprehensive list really necessary? Remember in MapReduce you only get two operators - map and reduce; so maybe I’m just excited at the 80+ operations in Spark!



Persistence

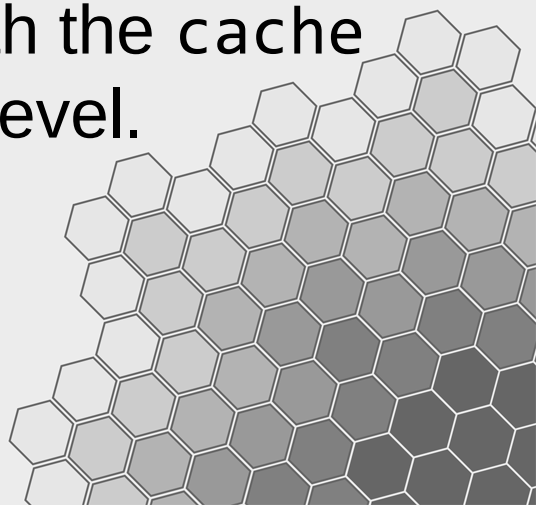
Spark will *persist* or *cache* RDD slices in memory on each node during operations.

Fault tolerant - in case of failure, Spark can rebuild the RDD from the lineage, automatically recreating the slice.

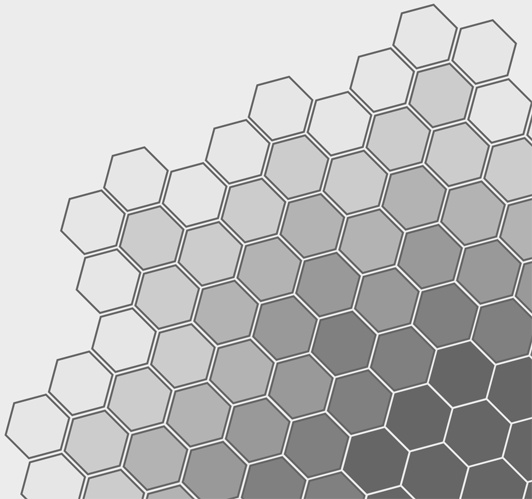
Super fast - will allow multiple operations on the same data set.

You can mark an RDD to be persisted with the `cache` method on an RDD along with a storage level.

Python objects are always pickles.



Beyond RDDs

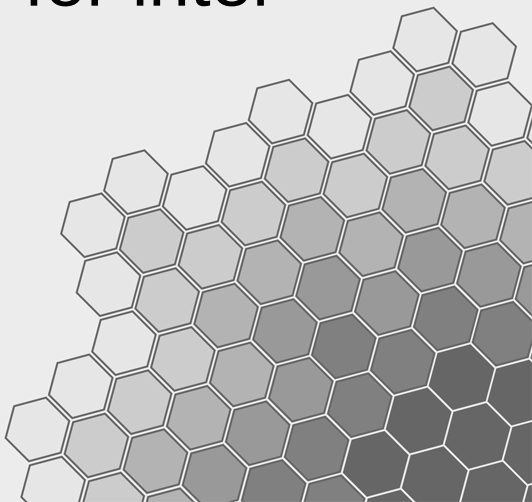


Variables and Memory

Any variable or data within a closure or data structure will be distributed to the worker nodes along with the closure.

Usually these variables are just constants but they cannot be shared across workers. Instead, the following restricted structures are used for inter-process communication:

- *broadcast variables*
- *accumulators*

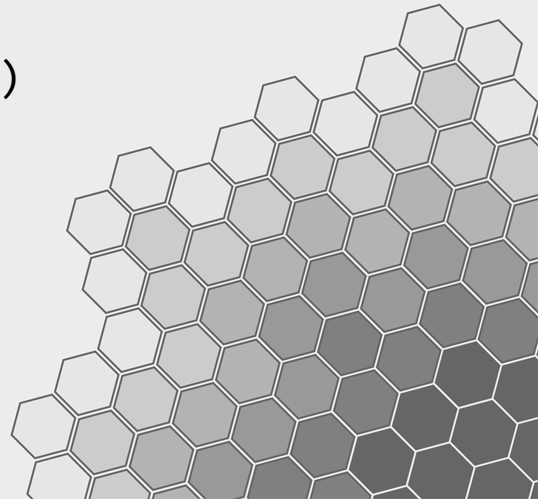


Broadcast Variables

Distribute some large piece of read-only data to all workers only once (e.g. a lookup table or stopwords).

This prevents multiple distribution per task, and efficiently gives nodes a copy of larger data using efficient broadcast algorithms.

```
import nltk

# Initialize the stopwords broadcast variable
stopwords = set(nltk.corpus.stopwords.words('english'))
stopwords = sc.broadcast(stopwords)

# Access the broadcast variable
if word in stopwords.value:
    pass
```

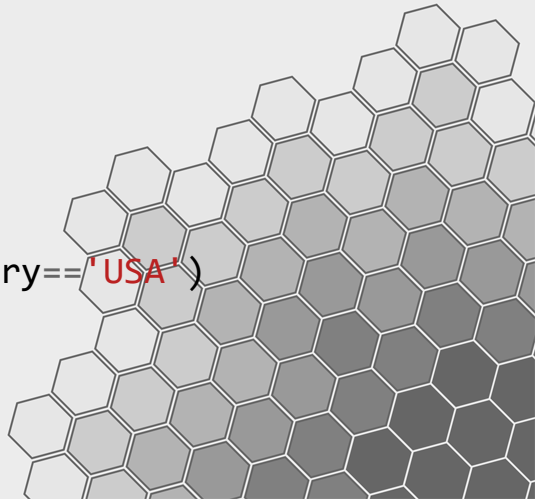


Accumulators

Variables that workers can “add” to using associative operations. These are read-only for the driver, but can be used as counters or summations

- Spark natively supports accumulators of numeric value types and standard mutable collections.
- Accumulators write-only to the workers.

```
gold   = sc.accumulator(0)

def count_medals(events):
    global gold,silver, bronze
    for event in events:
        if event.medal == 'GOLD':
            gold.add(1)

results = sc.textFile('olympics.csv').filter(lambda x: x.country=='USA')
results.foreach(count_medals)

print gold.value
```

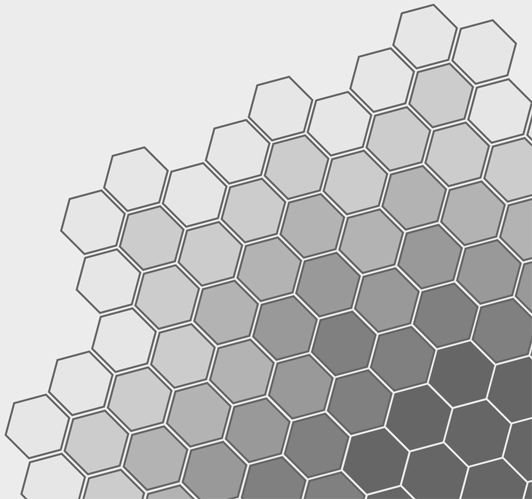


Key Value Pairs

Most Spark operations work on RDDs containing any type of objects, a few special operations are only available on RDDs of key-value pairs (“shuffle” operations, such as grouping or aggregating the elements by a key)

In Python, these operations work on RDDs containing built-in Python tuples such as (1, 2). Simply create such tuples and then call your desired operation.

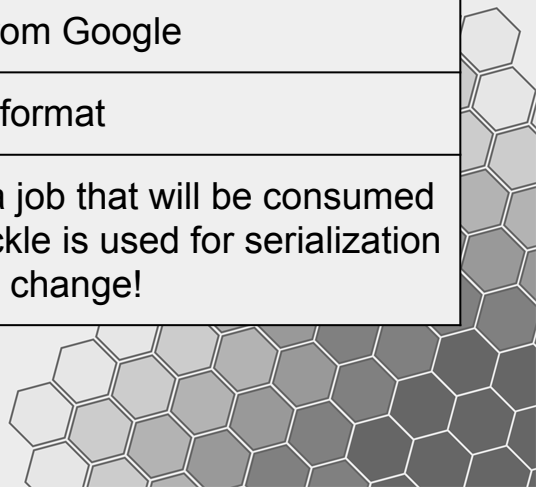
```
lines = sc.textFile("data.txt")
pairs = lines.map(lambda s: (s, 1))
counts = pairs.reduceByKey(lambda a, b: a + b)
```



Input and Output Formats

Selecting the best data file format:

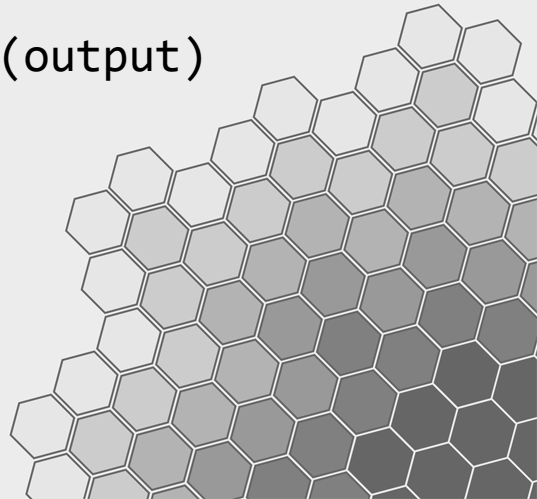
Format	Splitable	Structured	Description
text files	yes	no	records are split by line
JSON	yes	semi	log-json one record per line
CSV	yes	yes	very common format, used with databases
Sequence files	yes	yes	common Hadoop format for key-value data
Protocol buffers	yes	yes	fast, space-efficient format from Google
Avro	yes	yes	compact binary serialization format
Object Files	yes	yes	Useful for saving data from a job that will be consumed by shared code; however pickle is used for serialization so your objects in code can’t change!



Parsing JSON data

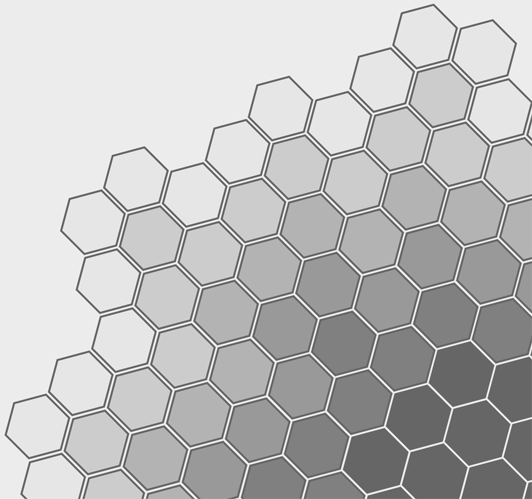
```
import json

# parse multi-line json file
parsed = sc.textFile("data.json").map(lambda x: json.loads(x))

# parse directory of json documents
parsed = sc.wholeTextFiles("data/").map(lambda x: json.loads(x))

# write out json data
data.map(lambda x: json.dumps(x)).saveAsTextFile(output)
```

Be careful - improperly formatted JSON will raise an exception! So too will trying to dump complex types like `datetime`.



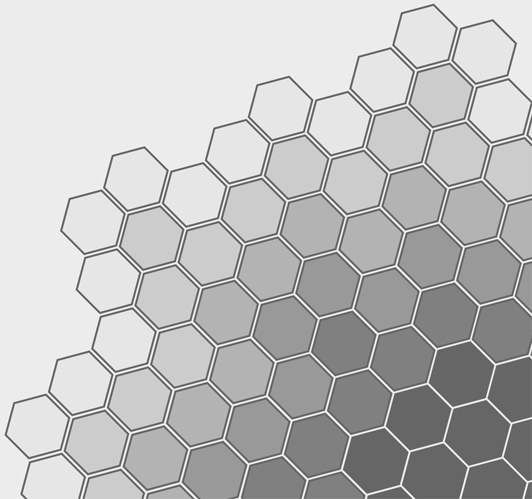
Parsing CSV data

```
import csv
from StringIO import StringIO

# Read from CSV
def load_csv(contents):
    return csv.reader(StringIO(contents[1]))

data = sc.wholetextFiles("data/").flatMap(load_csv)

# Write to CSV
def write_csv(records):
    output = StringIO()
    writer = csv.writer()
    for record in records:
        writer.writerow(record)
    return [output.get_value()]

data.mapPartitions(write_csv).saveAsTextFile("output/")
```



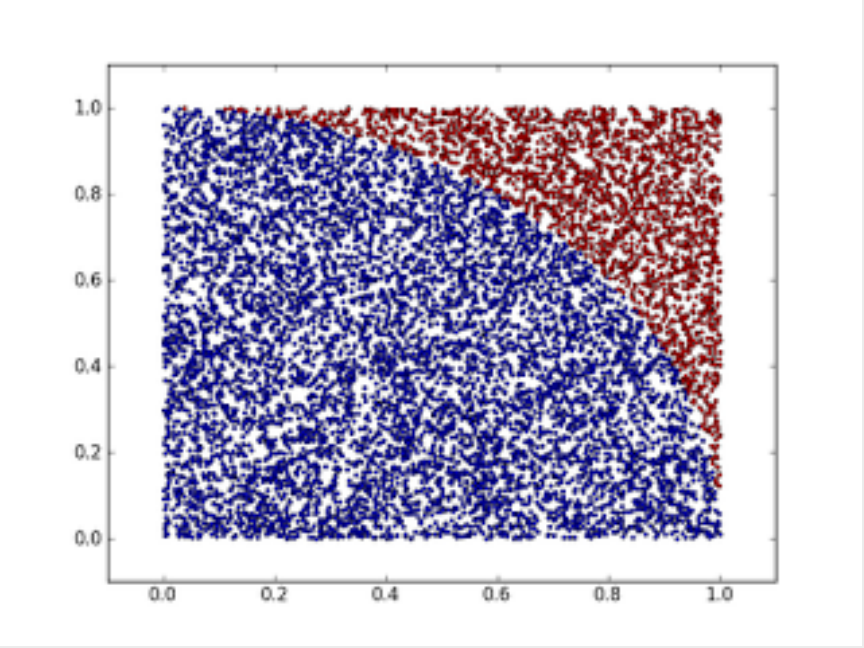
Parsing Structured Objects

```
import csv

from datetime import datetime
from StringIO import StringIO
from collections import namedtuple

DATE_FMT = "%Y-%m-%d %H:%M:%S" # 2013-09-16 12:23:33
Customer = namedtuple('Customer', ('id', 'name', 'registered'))

def parse(row):
    row[0] = int(row[0]) # Parse ID to an integer
    row[4] = datetime.strptime(row[4], DATE_FMT)
    return Customer(*row)

def split(line):
    reader = csv.reader(StringIO(line))
    return reader.next()

customers = sc.textFile("customers.csv").map(split).map(parse)
```

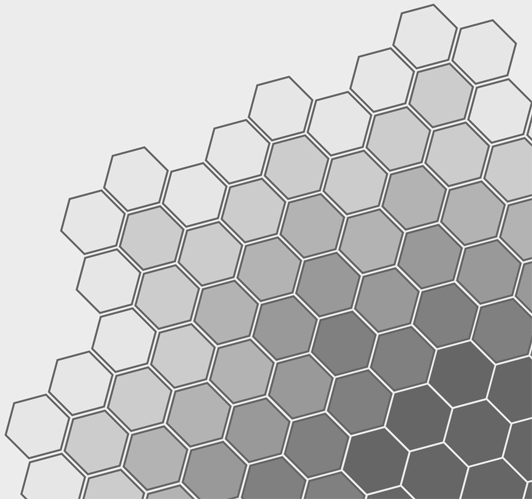


Exercise: Estimate Pi

Databricks has a great example where they use the Monte Carlo method to estimate Pi in a distributed fashion.



```
import sys
import random

from operator import add
from pyspark import SparkConf, SparkContext

def estimate(idx):
    x = random.random() * 2 - 1
    y = random.random() * 2 - 1
    return 1 if (x*x + y*y < 1) else 0

def main(sc, *args):
    slices = int(args[0]) if len(args) > 0 else 2
    N = 100000 * slices

    count  = sc.parallelize(xrange(N), slices).map(estimate)
    count  = count.reduce(add)

    print "Pi is roughly %0.5f" % (4.0 * count / N)
    sc.stop()

if __name__ == '__main__':
    conf = SparkConf().setAppName("Estimate Pi")
    sc   = SparkContext(conf=conf)
    main(sc, *sys.argv[1:])
```

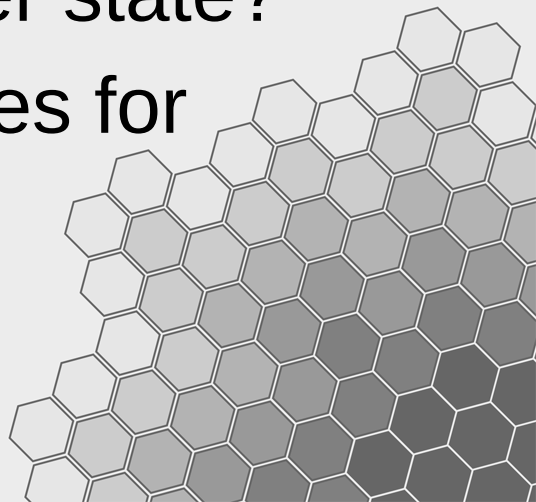


Exercise: Joins

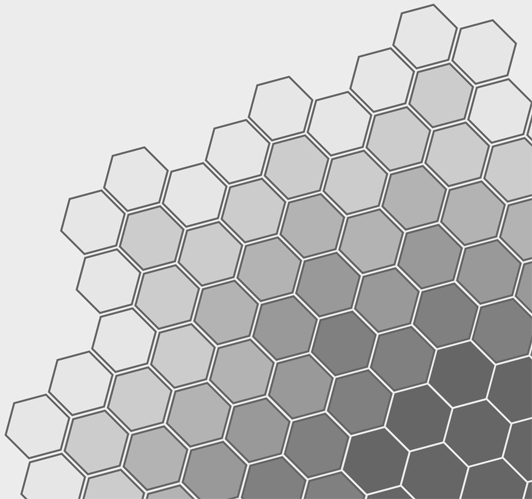
Using the shopping dataset, in particular the customers.csv and the orders.csv - find out what states have ordered the most products from the company.

What is the most popular product per state?

What month sees the most purchases for California? Massachusetts?



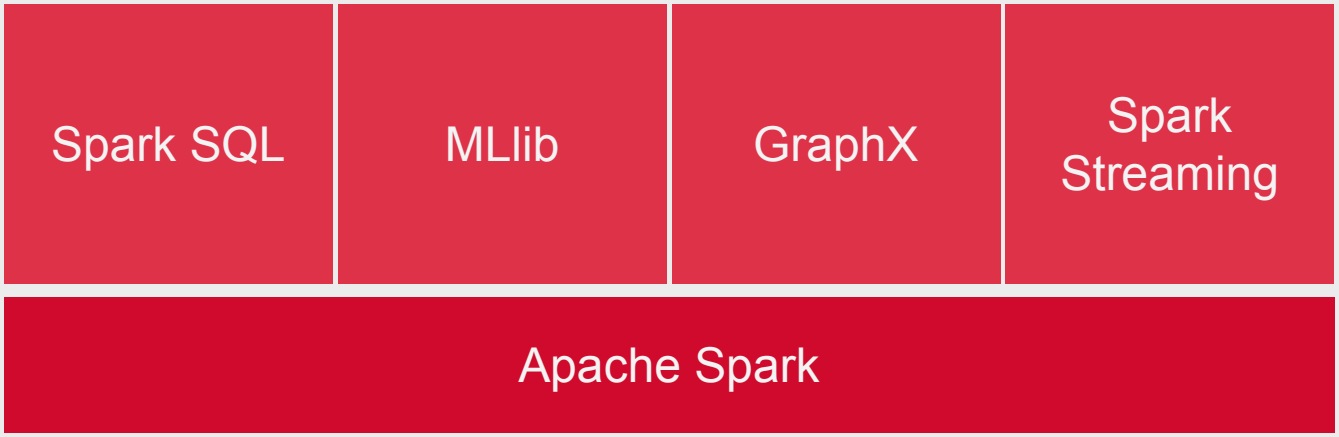
Spark Libraries



Workflows and Tools

The RDD data model and cached memory computing allow Spark to quickly and easily solve similar workflows and use cases that are part of Hadoop.

Spark has a series of high level tools at it's disposal that are added as component libraries, not integrated into the general computing framework:

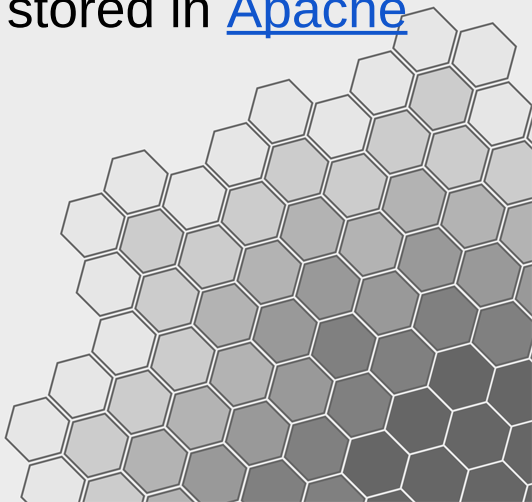


SparkSQL

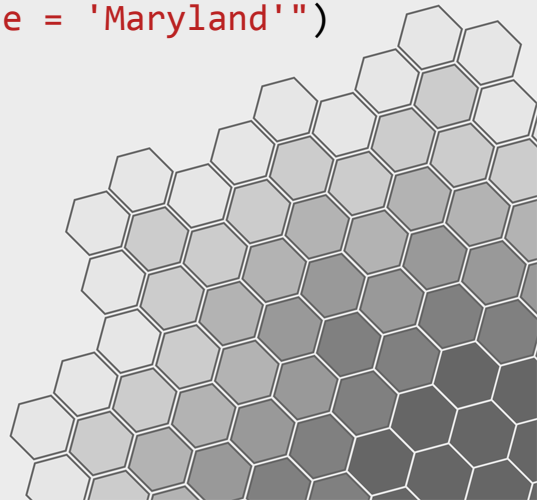
Spark SQL allows relational queries expressed in SQL or HiveQL to be executed using Spark.

- SchemaRDDs are composed of Row objects, along with a schema that describes the data types of each column in the row.
- A SchemaRDD is similar to a table in a traditional relational database and is operated on in a similar fashion.
- SchemaRDDs are created from an existing RDD, a Parquet file, a JSON dataset, or by running HiveQL against data stored in Apache Hive.

Spark SQL is currently an alpha component.



```
import csv

from StringIO import StringIO
from pyspark import SparkConf, SparkContext
from pyspark.sql import SQLContext, Row

def split(line):
    return csv.reader(StringIO(line)).next()

def main(sc, sqlc):
    rows = sc.textFile("fixtures/shopping/customers.csv").map(split)
    customers = rows.map(lambda c: Row(id=int(c[0]), name=c[1], state=c[6]))

    # Infer the schema and register the SchemaRDD
    schema = sqlc.inferSchema(customers).registerTempTable("customers")

    maryland = sqlc.sql("SELECT name FROM customers WHERE state = 'Maryland'")
    print maryland.count()

if __name__ == '__main__':
    conf = SparkConf().setAppName("Query Customers")
    sc   = SparkContext(conf=conf)
    sqlc = SQLContext(sc)
    main(sc, sqlc)
```



Spark Streaming

Spark Streaming is an extension of the core Spark API that enables scalable, high-throughput, fault-tolerant stream processing of live data.

- Data can be ingested from many sources like Kafka, Flume, Twitter, ZeroMQ, Kinesis or TCP sockets
- Can be processed using complex algorithms expressed with high-level functions like map, reduce, join and window.
- Finally, processed data can be pushed out to file systems, databases, and live dashboards; or apply Spark's machine learning and graph processing algorithms on data streams.

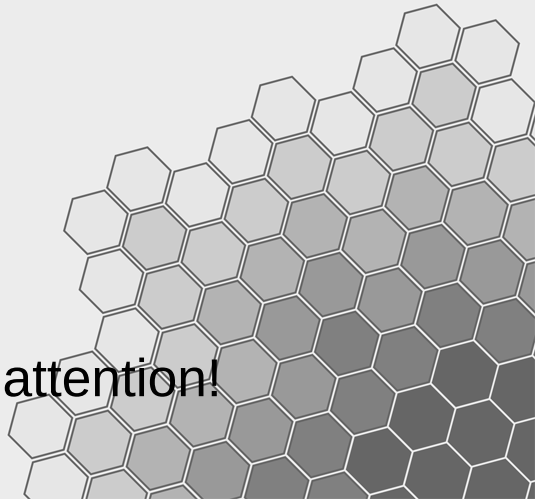
Spark MLLib

Spark’s scalable machine learning library consisting of common learning algorithms and utilities, including classification, regression, clustering, collaborative filtering, dimensionality reduction, as well as underlying optimization primitives.

Highlights include:

- summary statistics and correlation
- hypothesis testing, random data generation
- Linear models of regression (SVMs, logistic and linear regression)
- Naive Bayes and Decision Tree classifiers
- Collaborative Filtering with ALS
- K-Means clustering
- SVD (singular value decomposition) and PCA
- Stochastic gradient descent

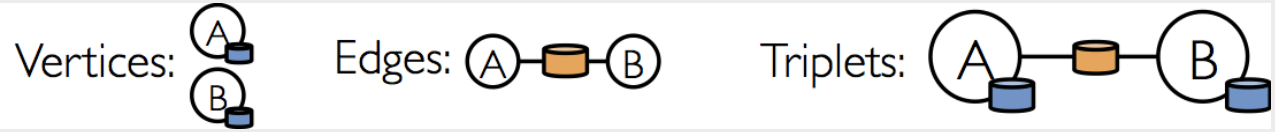
Not fully featured, still experimental - but gets a ton of attention!



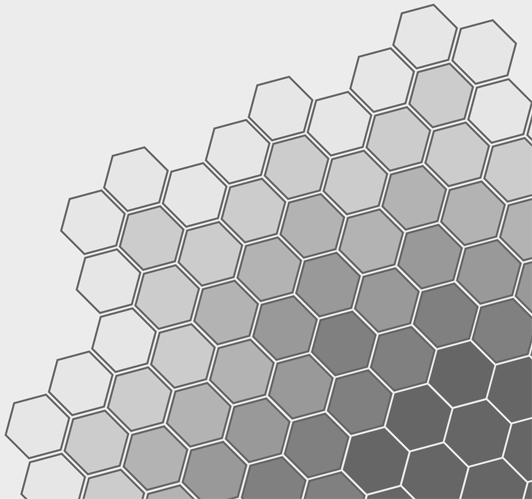
Spark GraphX

GraphX is the (alpha) Spark API for graphs and graph-parallel computation.

- GraphX extends the Spark RDD by introducing the Resilient Distributed Property Graph: a directed multigraph with properties attached to each vertex and edge.
- GraphX exposes a set of fundamental operators (e.g., `subgraph`, `joinVertices`, and `aggregateMessages`) as well as an optimized variant of the Pregel API.
- GraphX includes a growing collection of graph algorithms and builders to simplify graph analytics tasks.



Moving to a Cluster on EC2



Setting up EC2

Spark is for clustered computing - if your data is too large to compute on your local machine - then you're in the right place! An easy way to get Spark running is with EC2.

- a cluster of 5 slaves (and 1 master) used at a rate of approximately 10 hours per week will cost you approximately $45.18 per month.
- go to the AWS Console and obtain a pair of EC2 keys.

Add the following to your bash profile:

```
export AWS_ACCESS_KEY_ID=myaccesskeyid
export AWS_SECRET_ACCESS_KEY=mysecretaccesskey
```

Creating and Destroying Clusters

Create a cluster:

```
$ cd $SPARK_HOME/ec2
$ ./spark-ec2 -k <keypair> -i <key-file> -s <num-slaves> --copy-aws-credentials \
      launch <cluster-name>
```

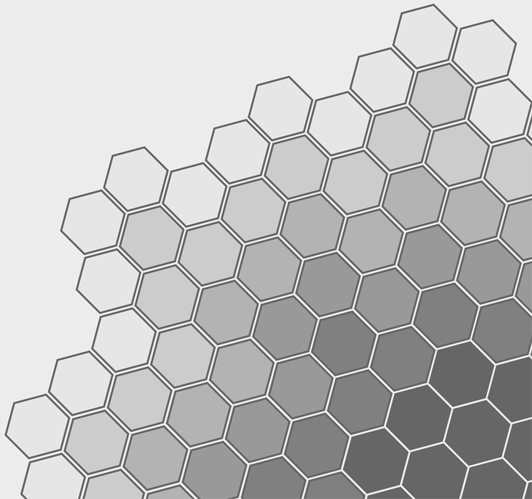
Pause and restart cluster:

```
$ ./spark-ec2 stop <cluster-name>
$ ./spark-ec2 start <cluster-name>
```

You're not billed for a paused cluster.

Destroy a cluster:

```
$ ./spark-ec2 destroy <cluster-name>
```



Synchronizing Data

SSH into your cluster to run jobs:

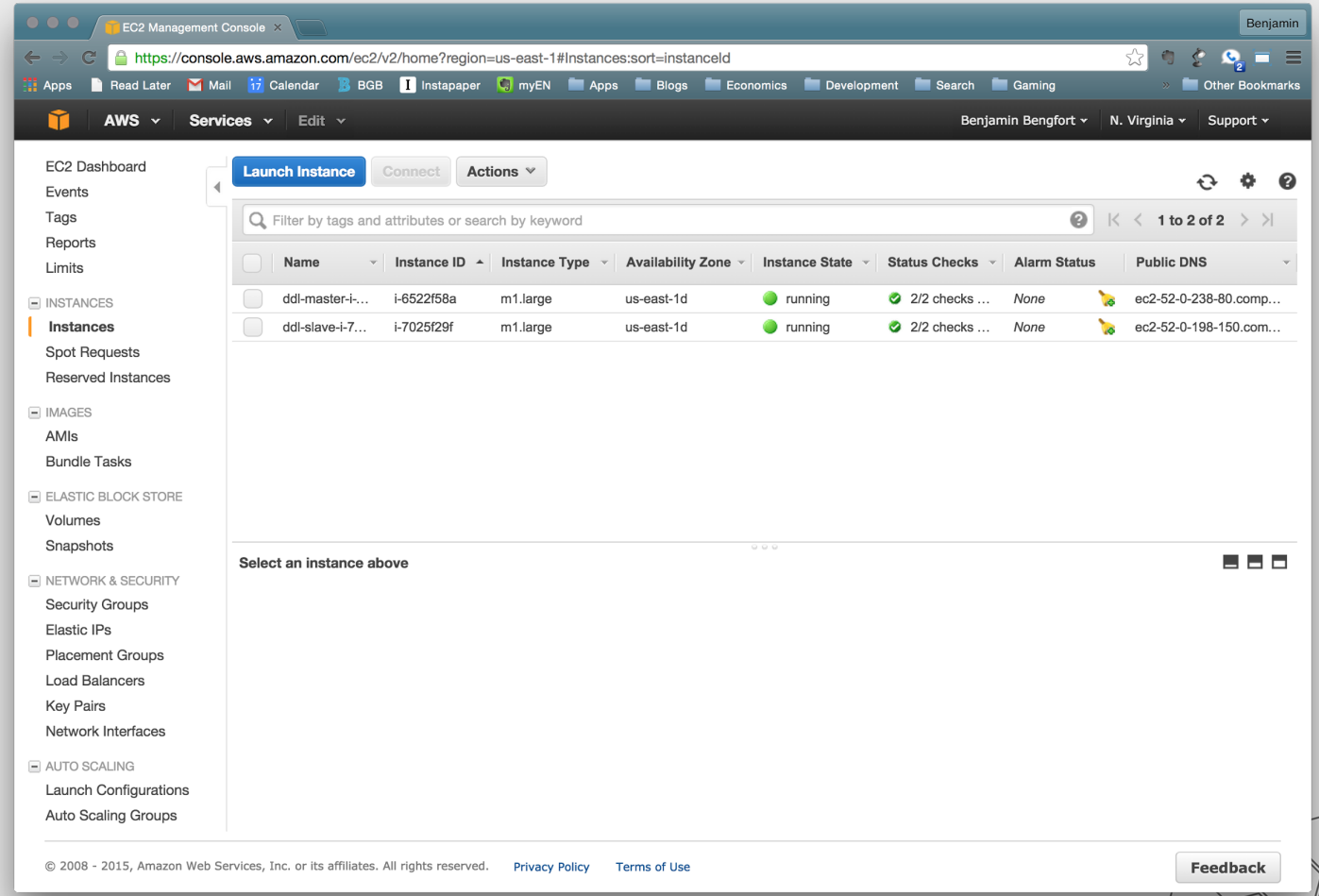
```
$ ./spark-ec2 -k <keypair> -i <key-file> login <cluster-name>
```

rsync data to all the slaves in the cluster:

```
$ ~/spark-ec2/copy-dir
```

But normally you'll just store data in S3 and SCP your driver files to the master node. Note that if you terminate a cluster, all the data on the cluster is lost.

To access data on S3, use the `s3://bucket/path/` URI



Be sure to check your EC2 Console

Don’t get a surprise bill!