Parallel Computing - Spark

® Ka-Chun Wong, City University of Hong Kong
® Harold Liu: “Berkeley Data Analytics Stack”
® Brian Clapper: “Intro to Spark Development”

Hadoop: Batch Processing ?

Growth of Big Datasets

- Internet/Online Data
  - Clicks
  - Searches
  - Server requests
  - Web logs
  - Cell phone logs
  - Mobile GPS locations
  - User generated content
  - Entertainment (YouTube, Netflix, Spotify, …)
- Healthcare and Scientific Computations
  - Genomics, medical images, healthcare data, billing data
- Graph data
  - Telecommunications network
  - Social networks (Facebook, Twitter, LinkedIn, …)
  - Computer networks
- Internet of Things
  - RFID
  - Sensors
- Financial data

First Trend – Big Data

- The Large Hadron Collider produces about 30 petabytes of data per year
- Facebook's data is growing at 8 petabytes per month
- The New York stock exchange generates about 4 terabytes of data per day
- YouTube had around hundreds of petabytes.
- Internet Archive stores around 19 petabytes of data

Memory Capacity Conversion Chart

Term (Abbreviation)	Approximate Size
Byte (B)	8 bits
Kilobyte (KB)	1024 bytes / 10^3 bytes
Megabyte (MB)	1024 KB / 10^6 bytes
Gigabyte (GB)	1024 MB / 10^9 bytes
Terabyte (TB)	1024 GB / 10^{12} bytes
Petabyte (PB)	1024 TB / 10^{15} bytes
Exabyte (EB)	1024 PB / 10^{18} bytes
Zettabyte (ZB)	1024 EB / 10^{21} bytes
Yottabyte (YB)	1024 ZB / 10^{24} bytes

https://medium.com/@ayushinegi5nov/how-big-data-leads-to-big-problems-41ead756d009

Second Trend - Cloud and Distributed Computing

- The second trend is pervasiveness of cloud-based storage and computational resources for processing big datasets
- Cloud characteristics
  - Provide a scalable standard environment
  - On-demand computing
  - Pay as you need
  - Dynamically scalable
  - Cheaper

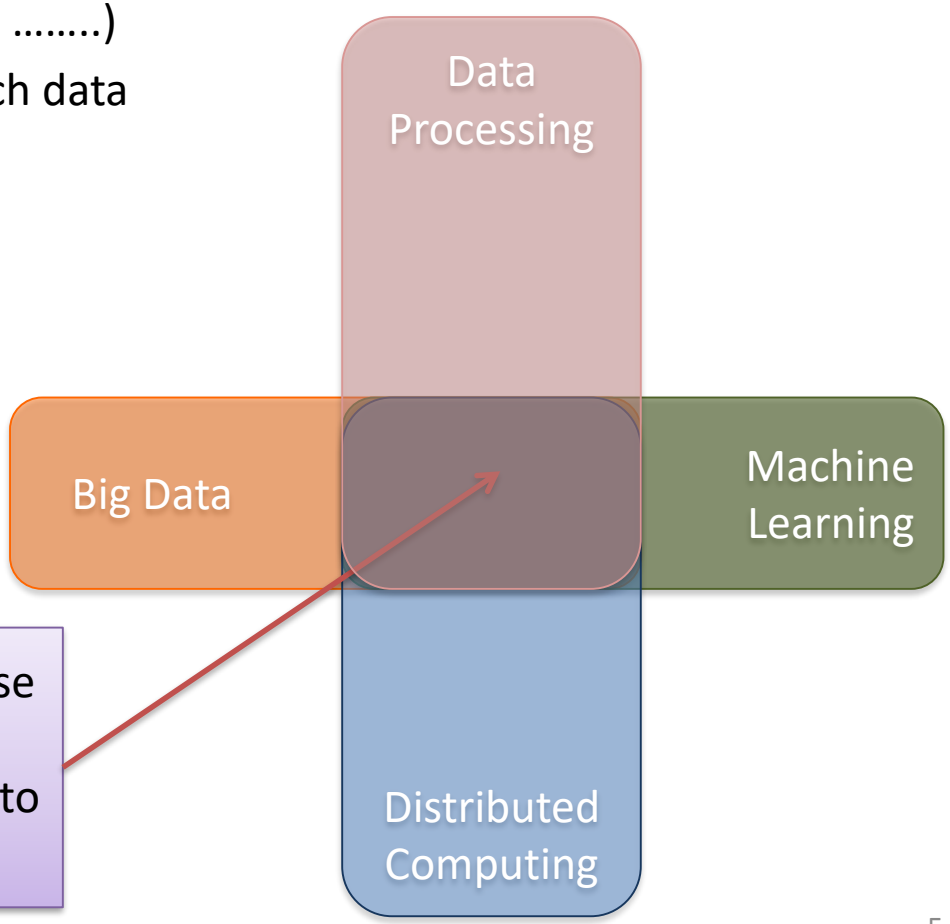
Third Trend

Data Processing

Fourth Trend

Machine Learning

- Third Trend - Data Processing
  - Traditional ETL (Extract, Transform, Load)
  - Data Stores (e.g. HBase, HDFS, ……..)
  - Streaming, multimedia, & batch data
- Fourth Trend - Machine Learning
  - Classification
  - Regression
  - Clustering
  - Collaborative filtering

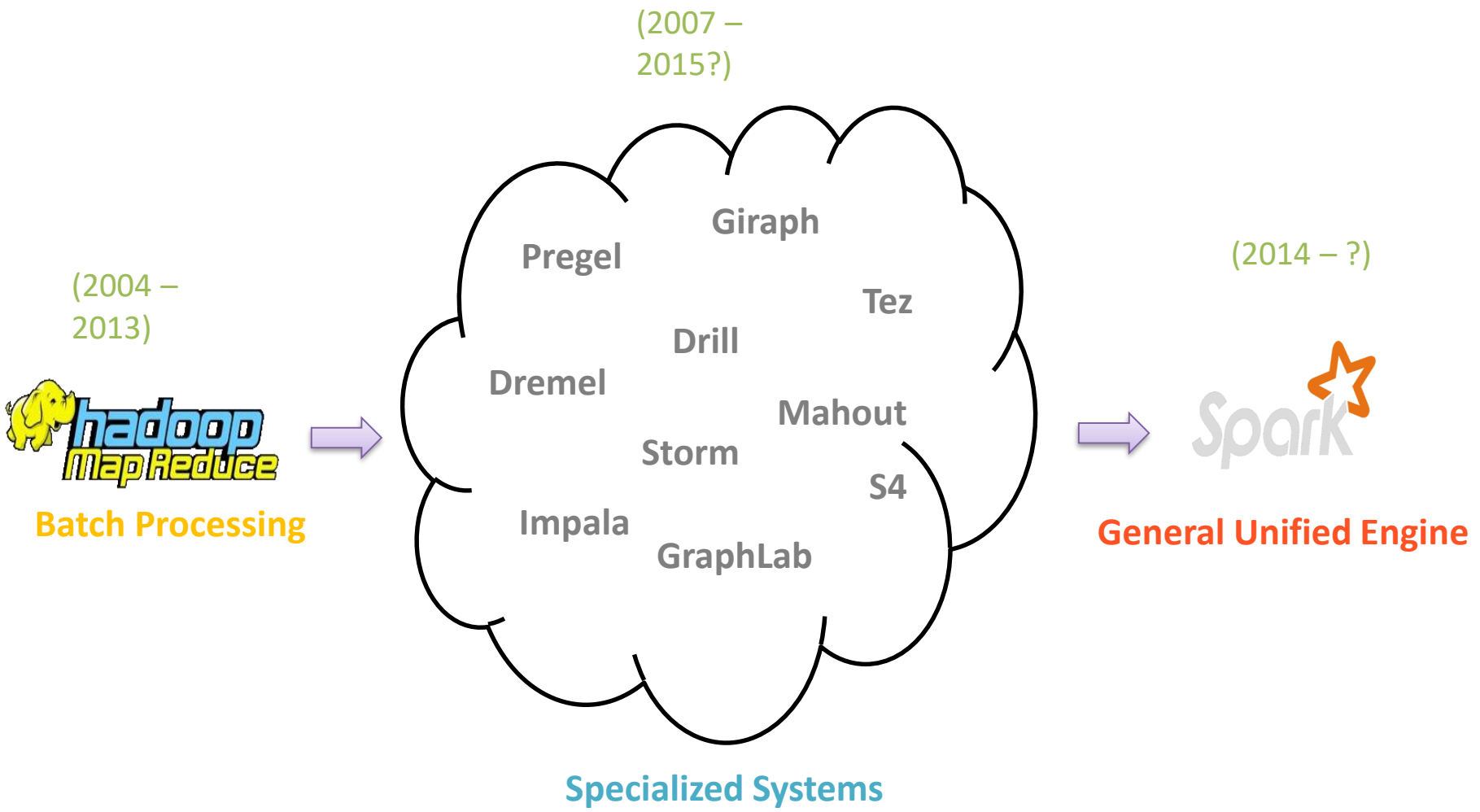


Working at the Intersection of these four trends is very exciting and challenging and require new ways to store and process **Big Data**

® Afzal Godil: "Apache Hadoop and Spark: Introduction and Use Cases for Data Analysis"

Why not Apache Hadoop ?

1. Forced into “Map” and “Reduce” operations.
   - How about “Join” and “Filter” operations?
2. Read from secondary storage (e.g. harddrives)
   - Iterative algorithms?
3. Mainly in Java
   - How about others such as Python or Scala?



(iterative, interactive, ML, streaming, graph, SQL, etc)

® Brian Clapper: “Intro to Spark Development”

100TB Daytona Sort Competition 2014

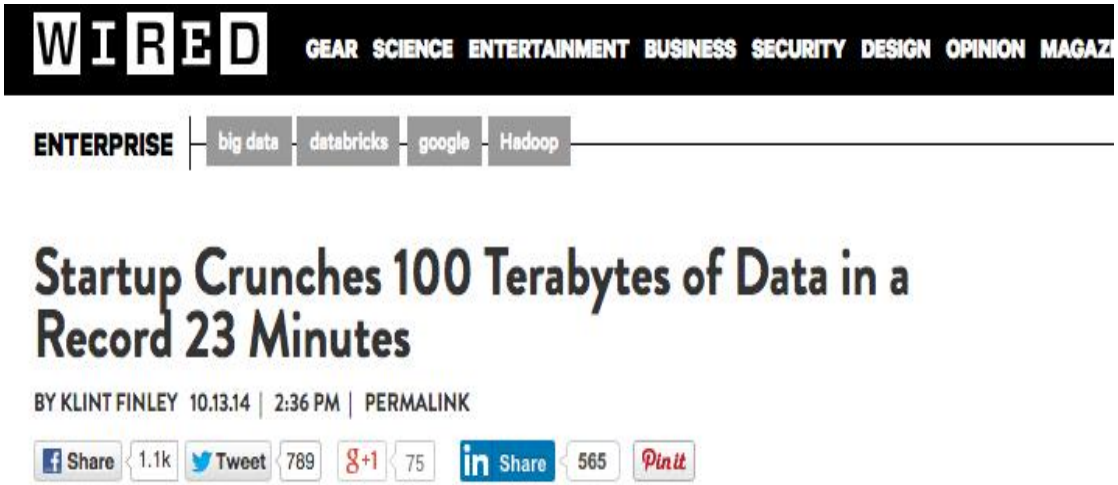
	Hadoop MR Record	Spark Record	Spark 1 PB
Data Size	102.5 TB	100 TB	1000 TB
Elapsed Time	72 mins	23 mins	234 mins
# Nodes	2100	206	190
# Cores	50400 physical	6592 virtualized	6080 virtualized
Cluster disk throughput	3150 GB/s (est.)	618 GB/s	570 GB/s
Sort Benchmark Daytona Rules	Yes	Yes	No
Network	dedicated data center, 10Gbps	virtualized (EC2) 10Gbps network	virtualized (EC2) 10Gbps network
Sort rate	**1.42 TB/min**	**4.27 TB/min**	**4.27 TB/min**
Sort rate/node	**0.67 GB/min**	**20.7 GB/min**	**22.5 GB/min**

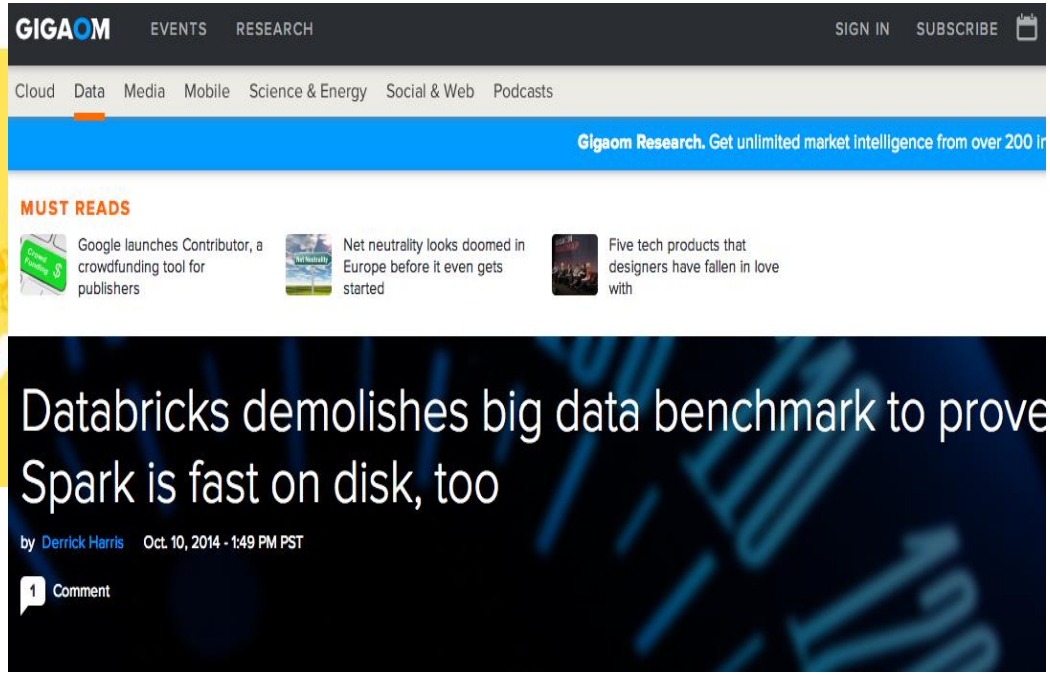
Spark sorted the same data 3X faster using 10X fewer machines than Hadoop

All the sorting took place on disk (HDFS) without using Spark's in-memory cache!

More info:
http://sortbenchmark.org
http://databricks.com/blog/2014/11/05/spark-officially-sets-a-new-record-in-large-scale-sorting.html

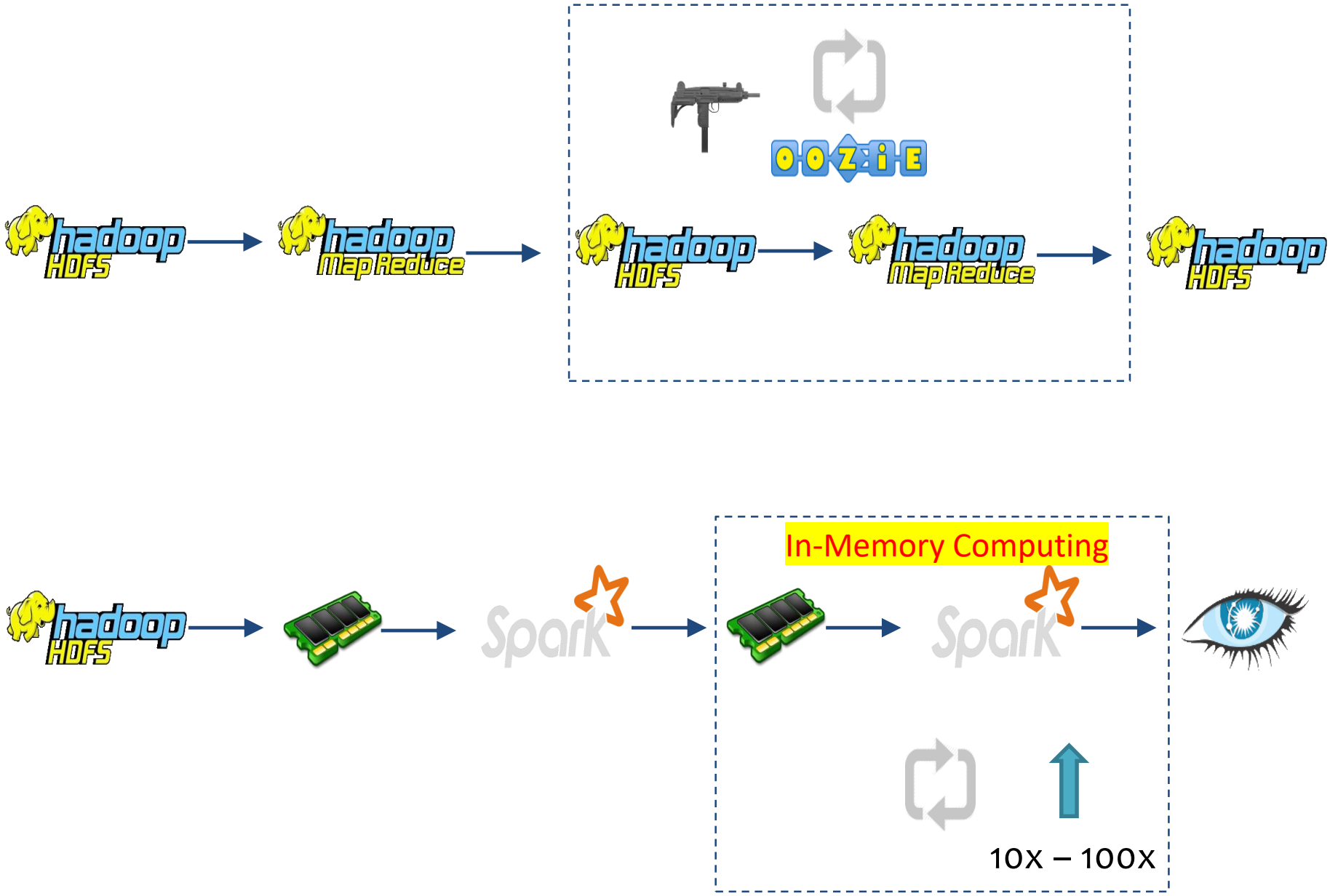
Work by Databricks engineers: Reynold Xin, Parviz Deyhim, Xiangrui Meng, Ali Ghodsi, Matei Zaharia


WIRED
GEAR SCIENCE ENTERTAINMENT BUSINESS SECURITY DESIGN OPINION MAGAZ
ENTERPRISE
big data
databricks
google
Hadoop
Startup Crunches 100 Terabytes of Data in a Record 23 Minutes
BY KLINT FINLEY 10.13.14 | 2:36 PM | PERMALINK
Share 1.1k
Tweet 789
+1 75
Share 565
Pin it


GIGAOM
EVENTS
RESEARCH
SIGN IN
SUBSCRIBE
Cloud
Data
Media
Mobile
Science & Energy
Social & Web
Podcasts
Gigaom Research. Get unlimited market intelligence from over 200 ir
MUST READS
Google launches Contributor, a crowdfunding tool for publishers
Net neutrality looks doomed in Europe before it even gets started
Five tech products that designers have fallen in love with
Databricks demolishes big data benchmark to prove Spark is fast on disk, too
by Derrick Harris
Oct. 10, 2014 - 1:49 PM PST
1 Comment

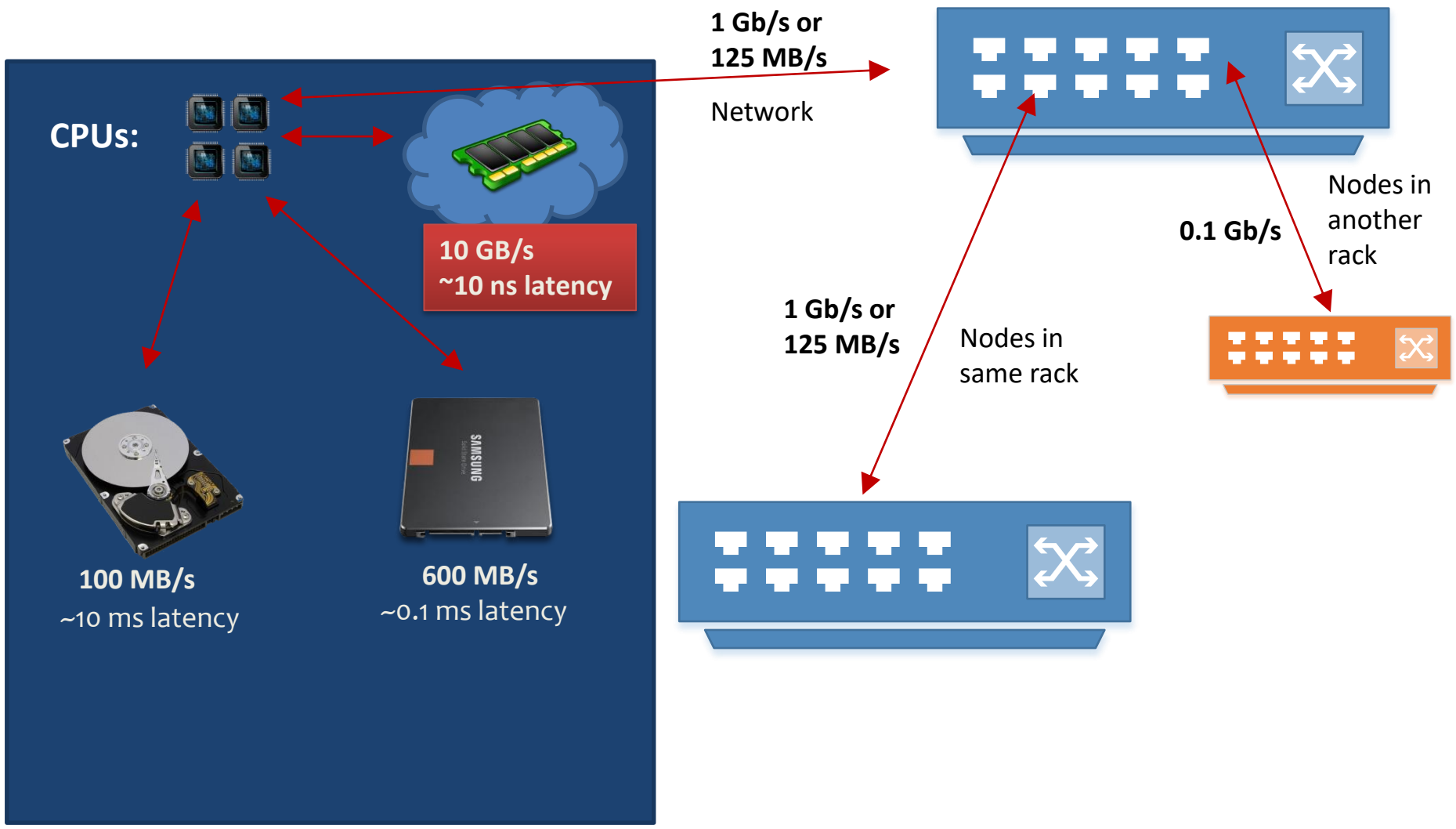
Apache Spark

- Built on top of Hadoop with expanded functions and enhanced performance
- Re-using Hadoop Ecosystem
- In-Memory Computing
  - 100x or 10x Speedup
- Native in Scala
- Supports R, Python, and JAVA



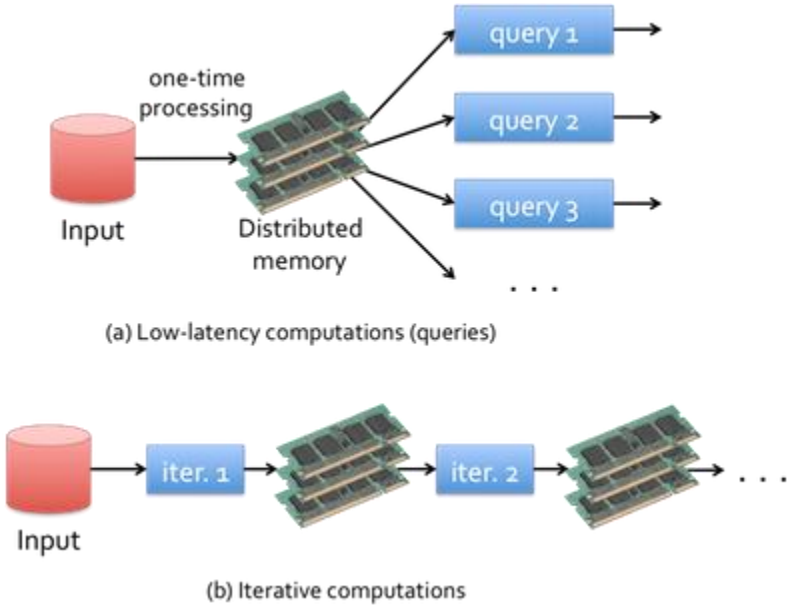
® Brian Clapper: “Intro to Spark Development”

In-Memory Computing



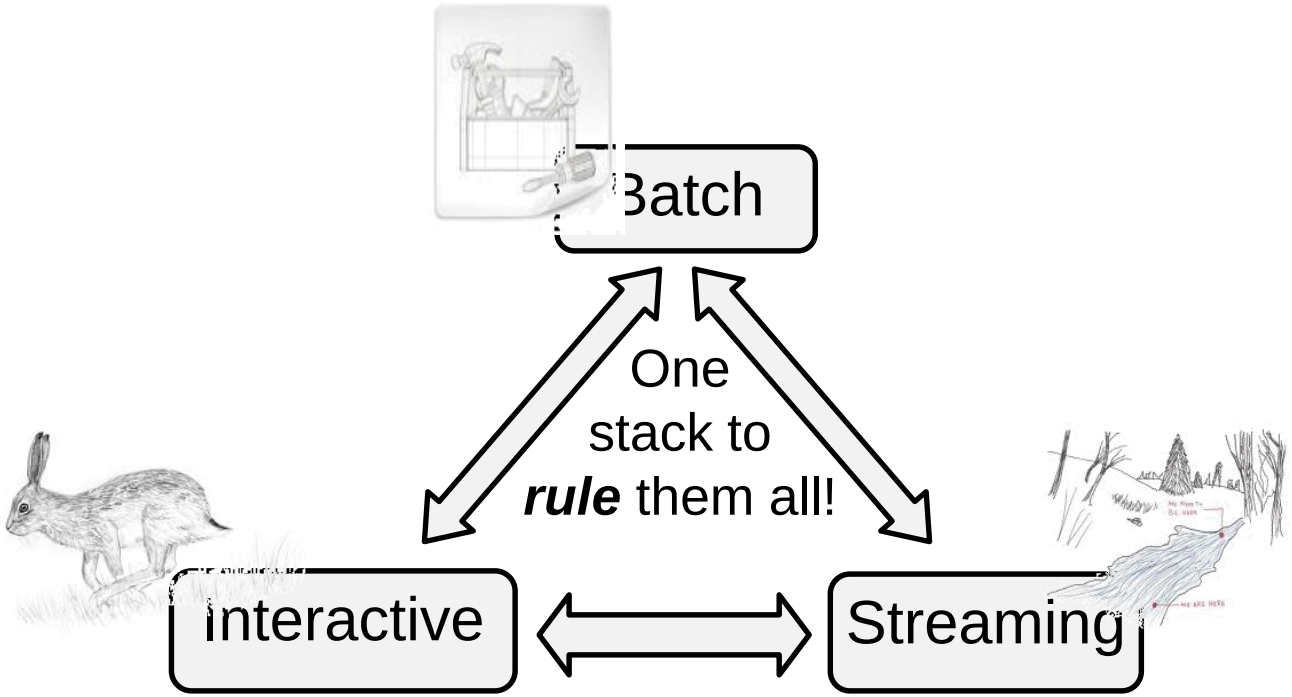
In-Memory Computing

- Low-latency computations by caching the working dataset in memory and then performing computations at memory speeds, and
- Efficient iterative algorithm by having subsequent iterations share data through memory, or repeatedly accessing the same dataset



https://www.databricks.com/blog/2013/11/21/putting-spark-to-use.html

Goals



- ***Easy*** to combine ***batch***, ***streaming***, and ***interactive***e computations
- ***Easy*** to develop ***iterative*** algorithms
- ***Compatible*** with existing open source ecosystem (Hadoop/HDFS)

vs

YARN

Mesos

Tachyon

SQL

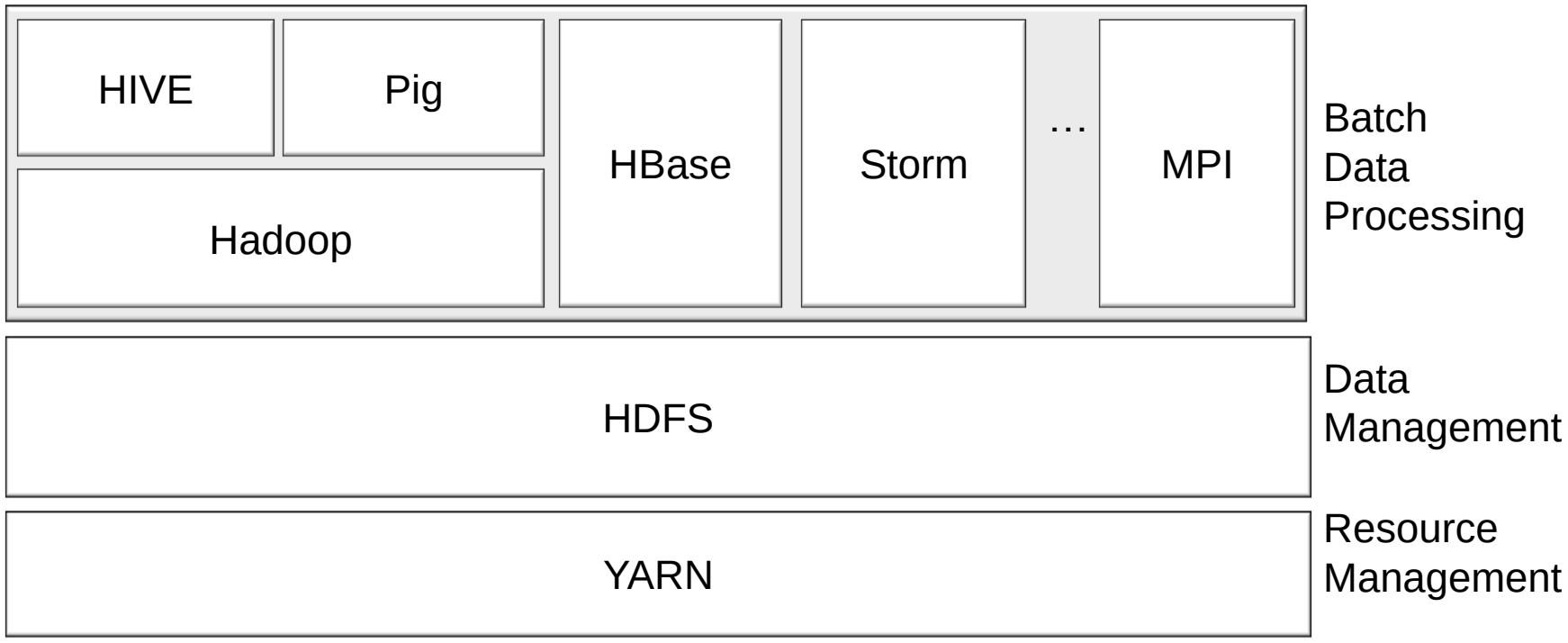
MLlib

Streaming

® Brian Clapper: “Intro to Spark Development”

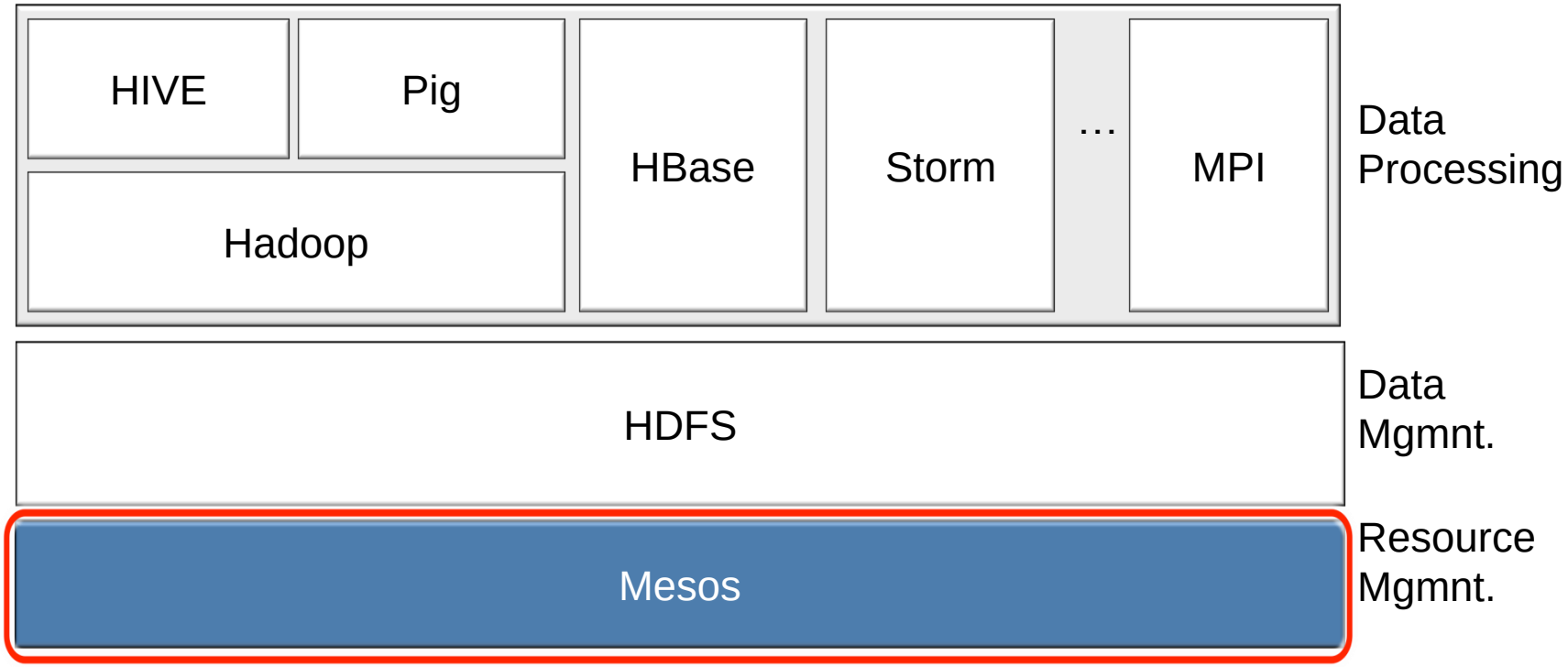
Berkeley Data Analytics Stack (BDAS)

- Existing Hadoop stack components….



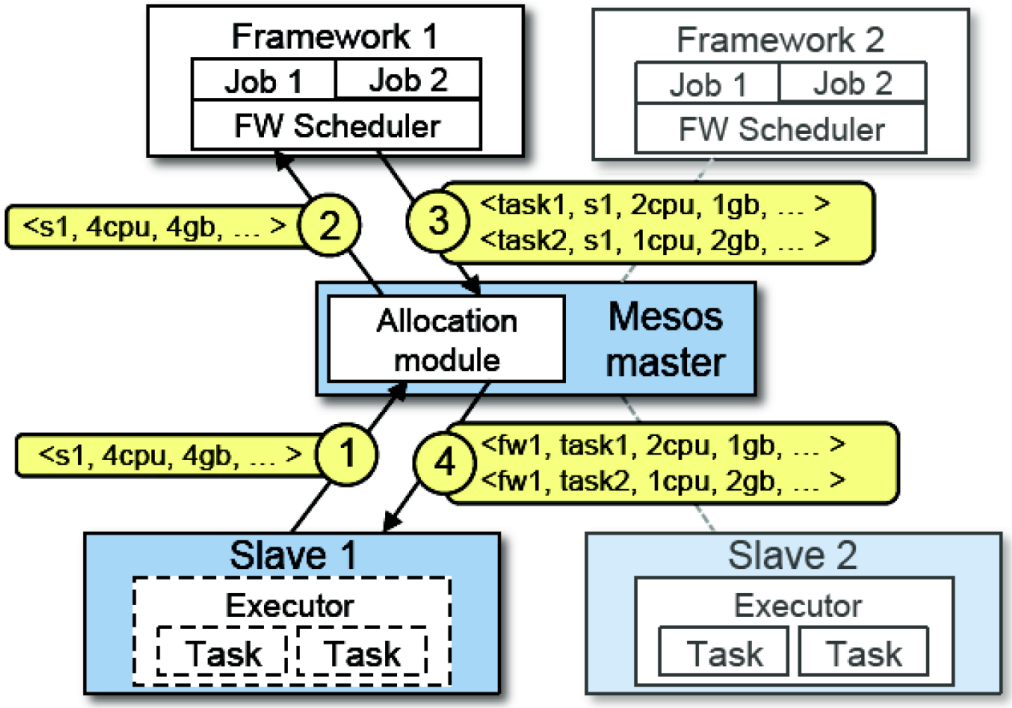
Mesos

- Management platform that allows multiple big data processing frameworks to share a cluster
- Compatible with existing open analytics stack
- Deployed in production at twitter on 3,500+ servers



Mesos – Case Example

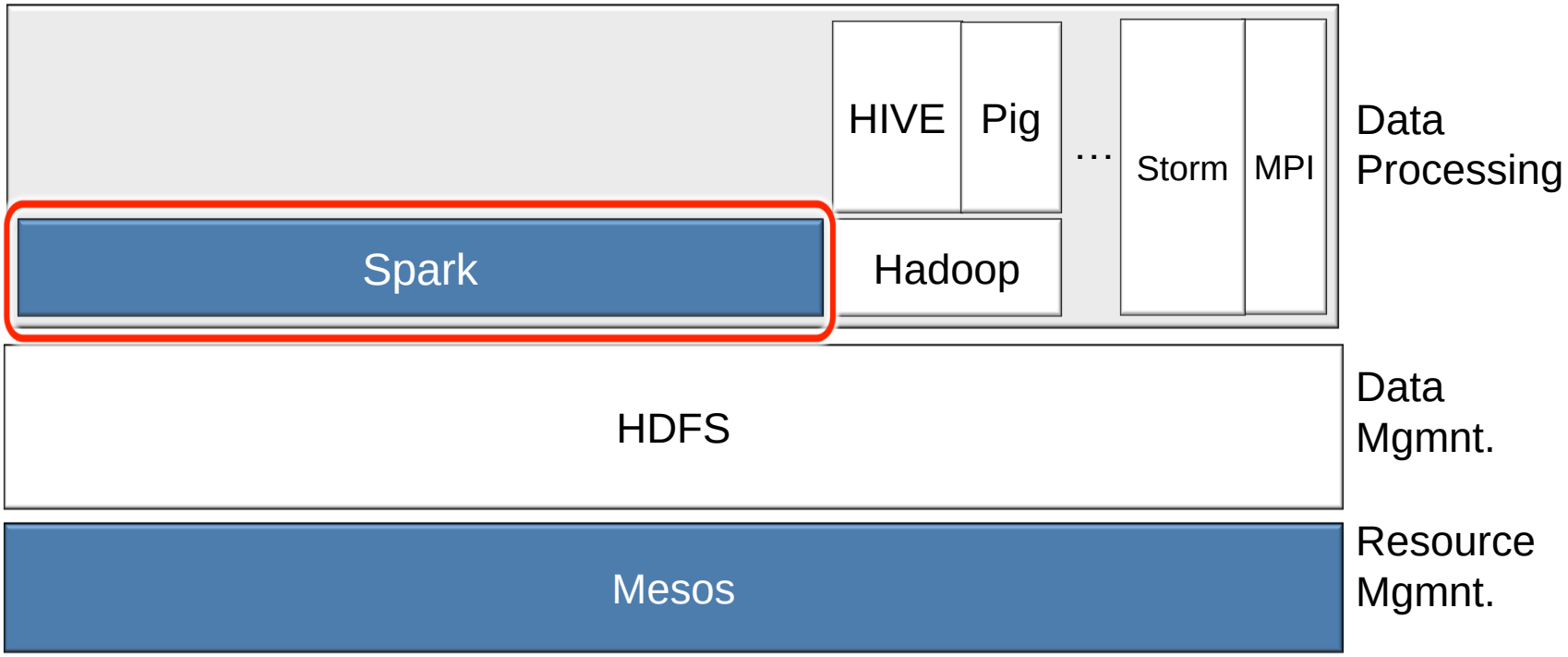
1. Slaves continuously send status updates about resources to the Master.
2. Master sends resource offers to frameworks.
3. Frameworks select which offers to accept and which tasks to run.
4. Unit of allocation is a resource offer (vector of available resources on a node).



https://id2221kth.github.io/

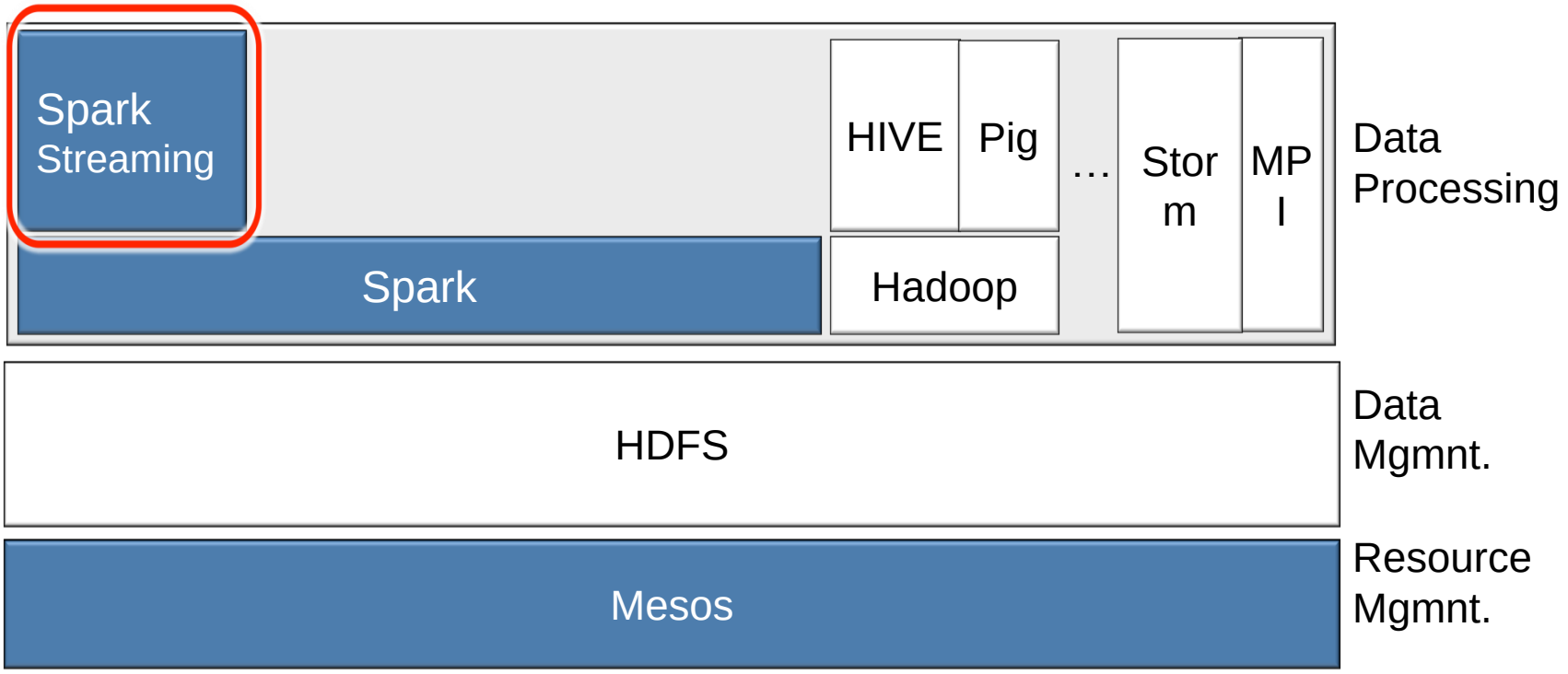
Spark

- In-memory framework for **interactive** and **iterative** computations
  - Resilient Distributed Dataset **(RDD)**:
  - fault-tolerance, robust, and in-memory storage abstraction
- Scala interface, Java and Python APIs



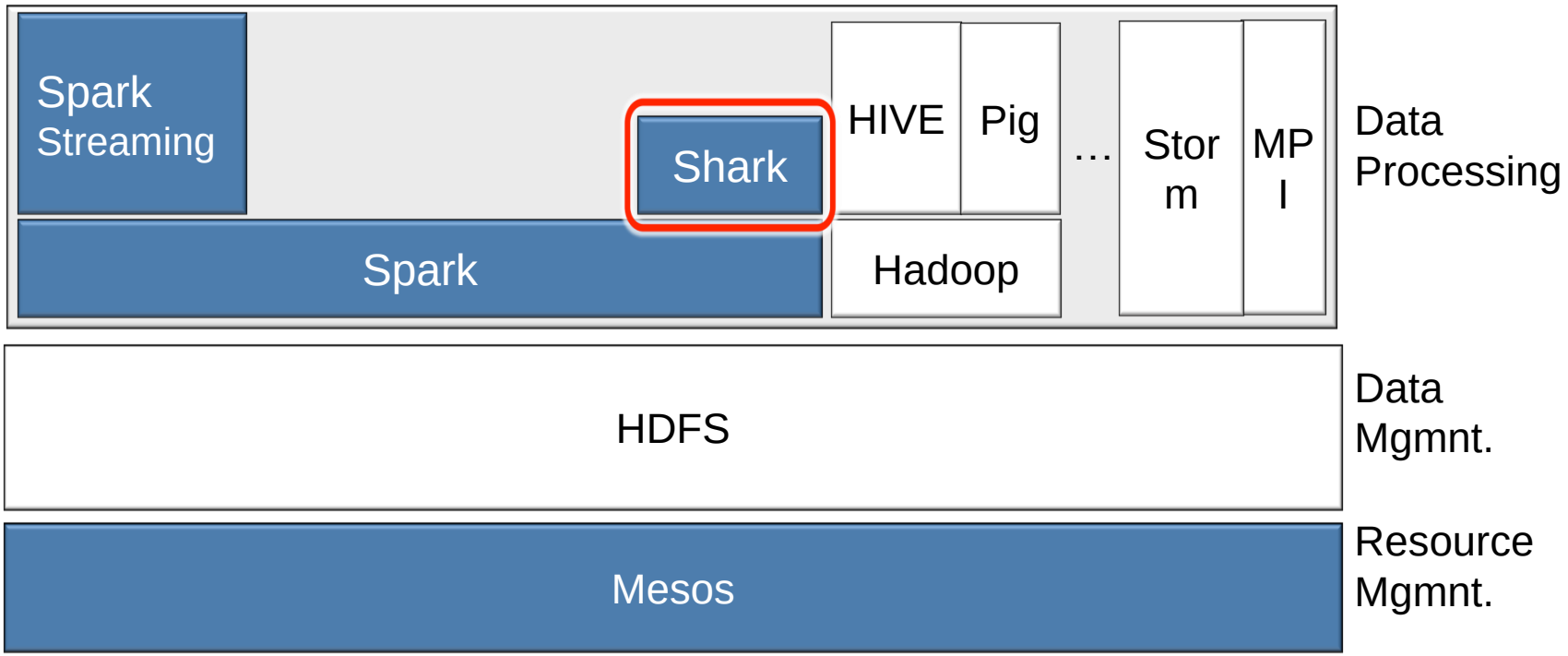
Spark Streaming

- Large scale data streaming computation. e.g.
  - User clickstreams from different IP addresses
  - Video streaming analytics
  - Car calling, pickup, and drop-off events



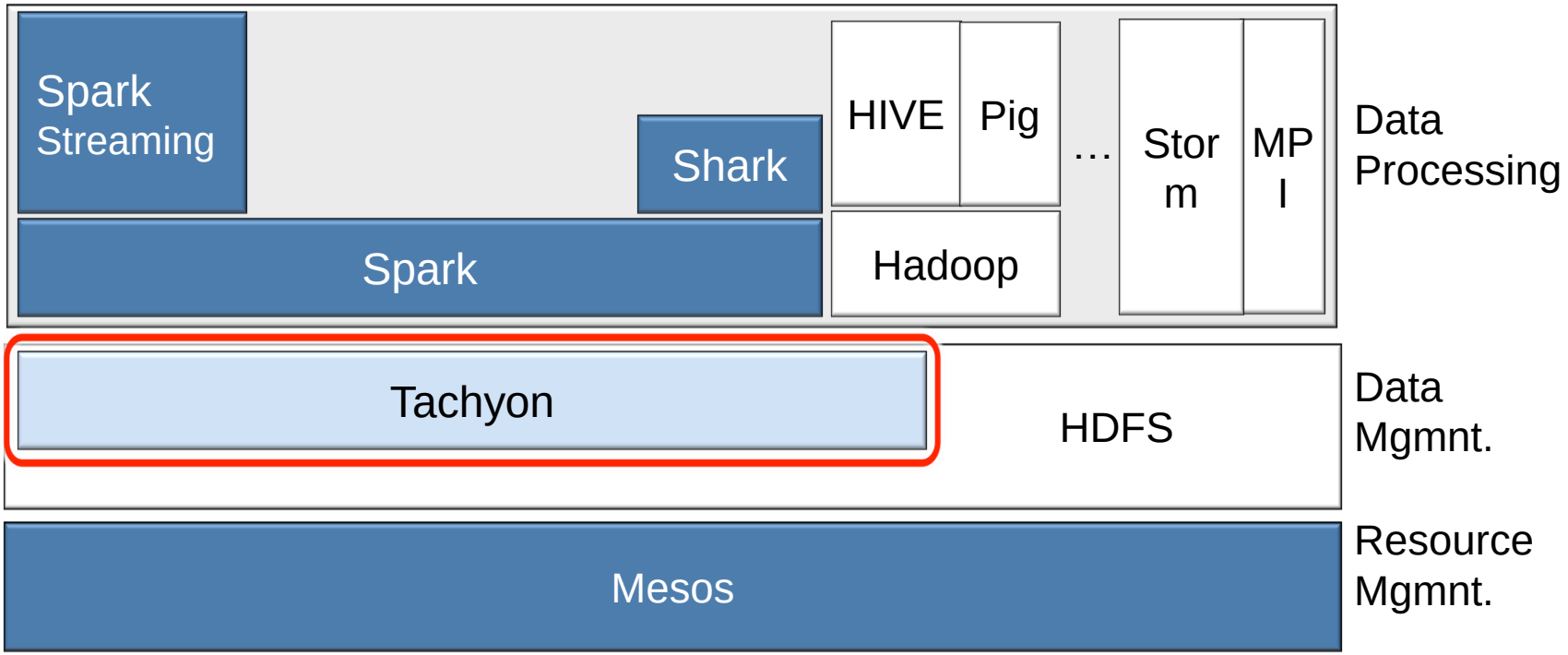
Shark → Spark SQL

- HIVE over Spark: SQL-like interface (supports Hive 0.9)
  - up to 100x faster for in-memory data, and 5-10x for disk
- In tests on hundreds of node clusters at YAHOO!



Tachyon

- Distributed and fault-tolerant in-memory storage
- Interface compatible to HDFS
- Support for Spark and Hadoop



BlinkDB

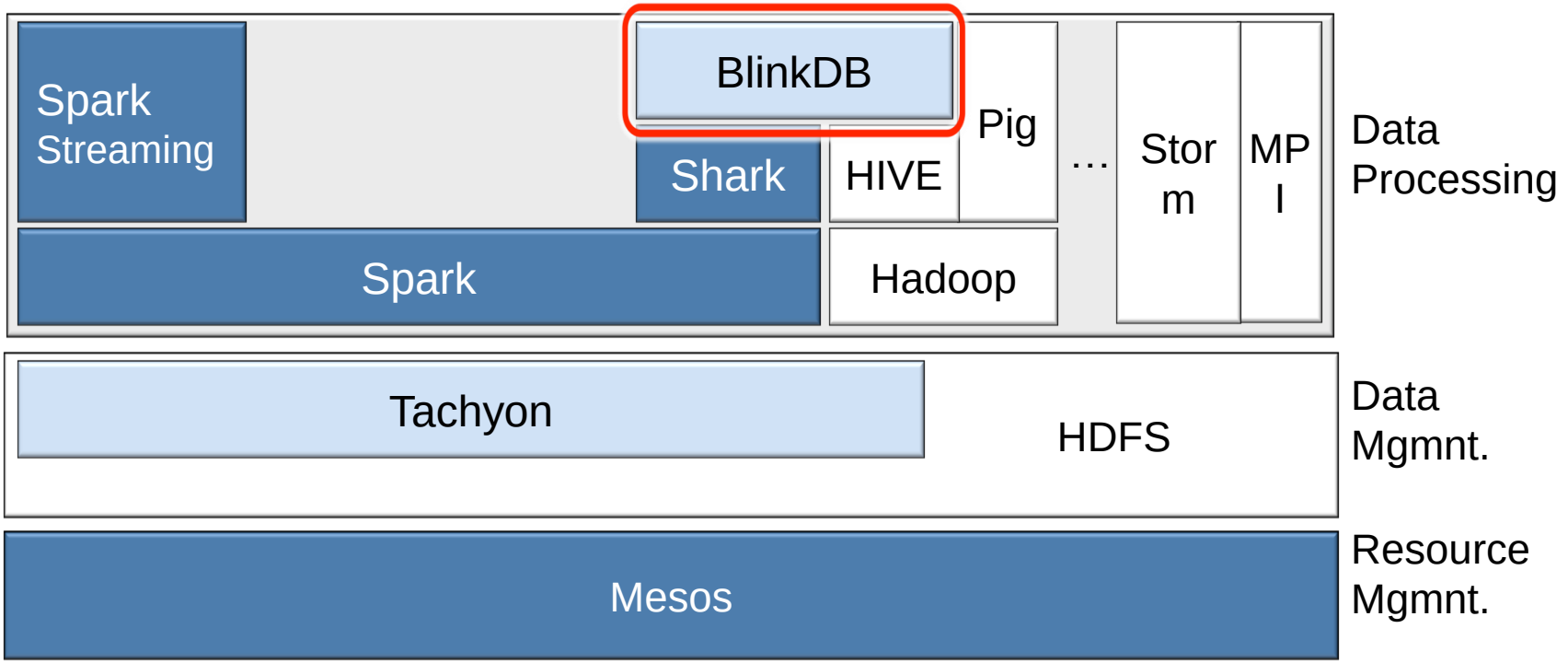
- Large scale approximate query engine
- Allow users to specify **error** or **time** bounds
- Preliminary prototype tested at Facebook

```
SELECT avg(sessionTime)
FROM Table
WHERE city='San Francisco'
WITHIN 2 SECONDS
```

Queries with Time Bounds

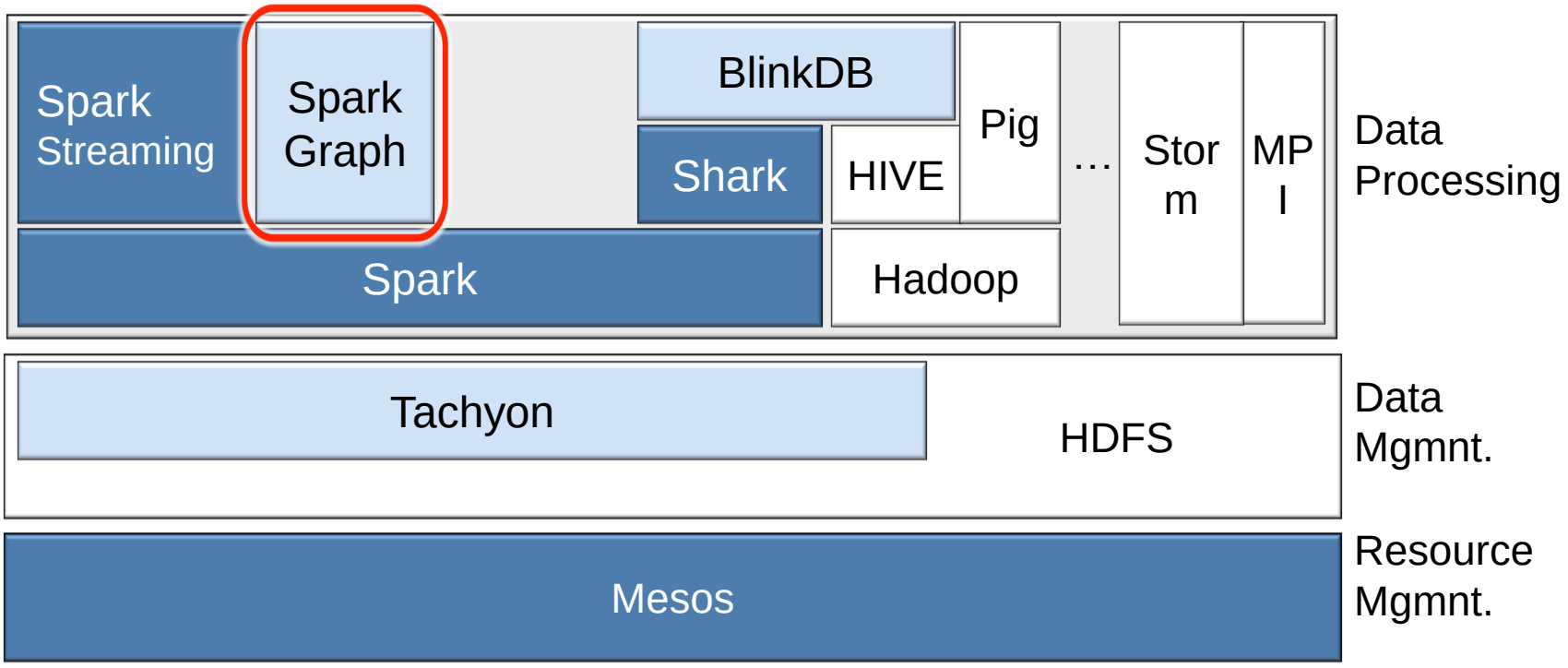
```
SELECT avg(sessionTime)
FROM Table
WHERE city='San Francisco'
ERROR 0.1 CONFIDENCE 95.0%
```

Queries with Error Bounds



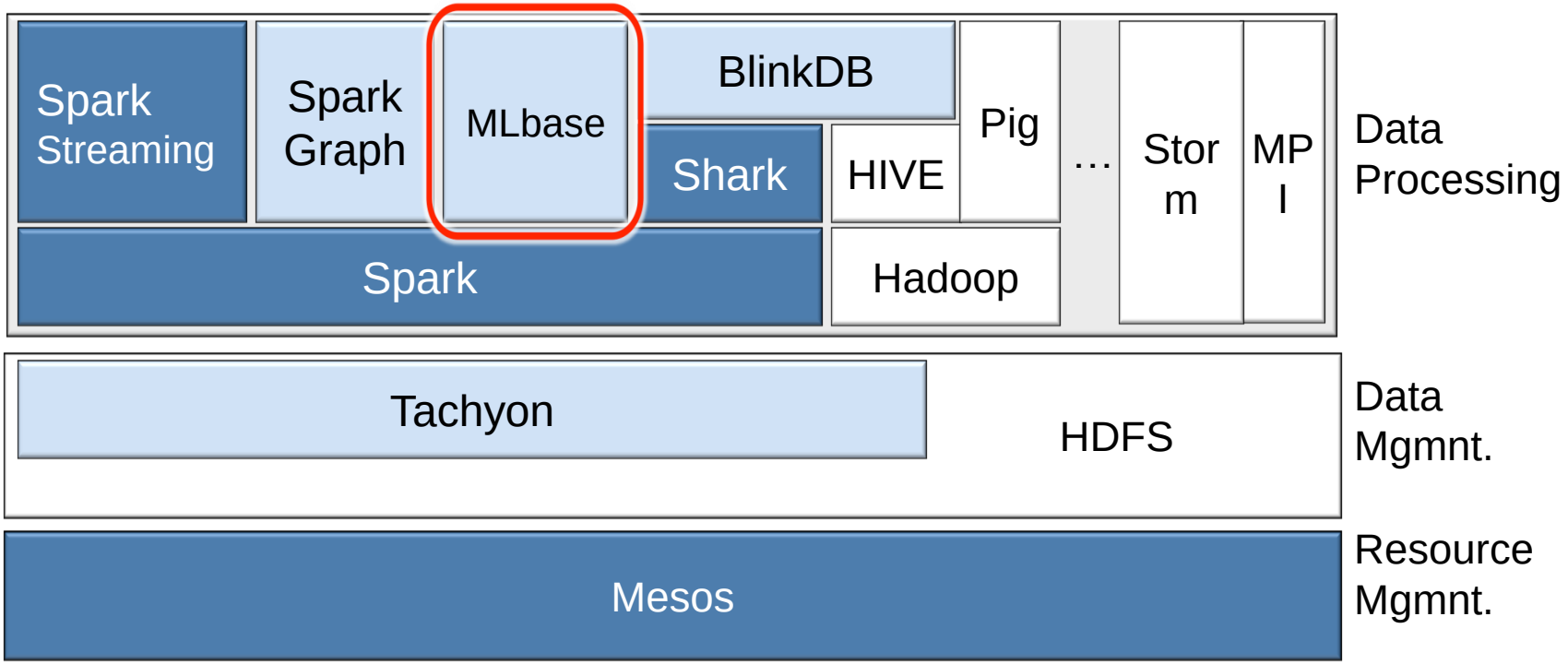
SparkGraph

- **GraphLab API** and **Toolkits** on top of Spark
- Fault tolerance by leveraging Spark



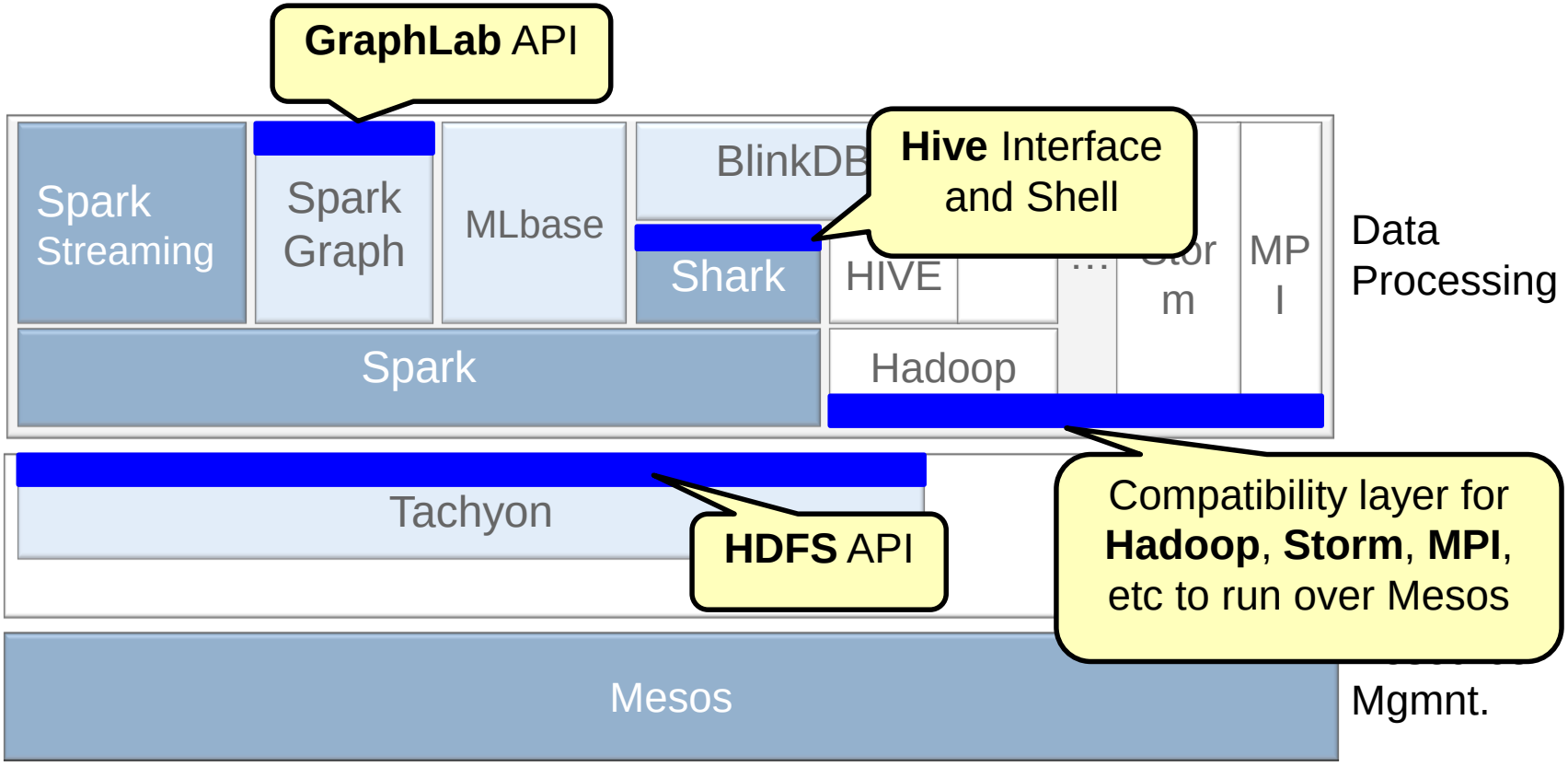
MLbase

- Declarative approach to ML
- Develop scalable ML algorithms
- Make ML accessible to non-experts



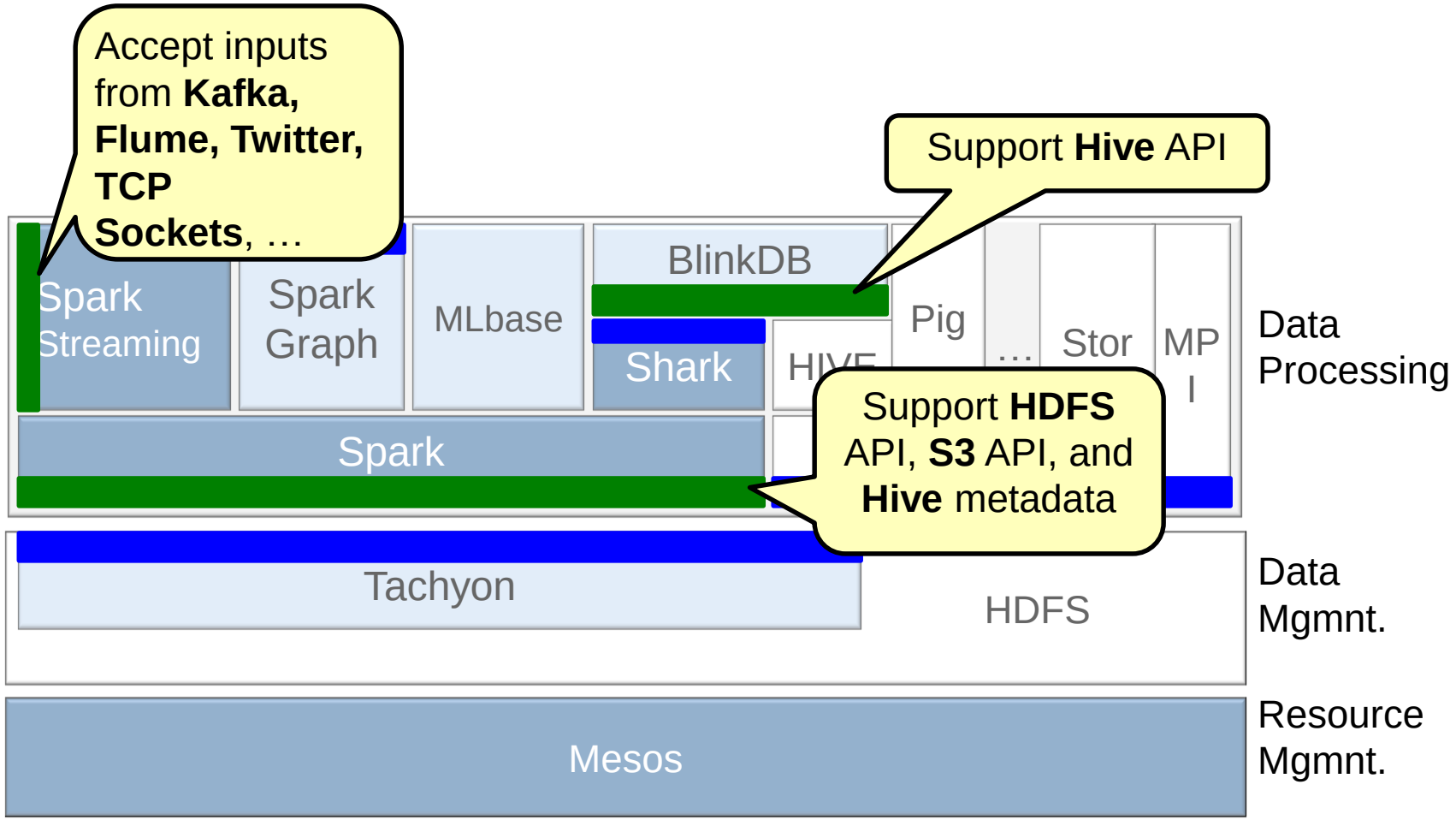
Compatible with Open Source Ecosystem

- ***Support*** existing interfaces whenever possible



Compatible with Open Source Ecosystem

- ***Use*** existing interfaces whenever possible



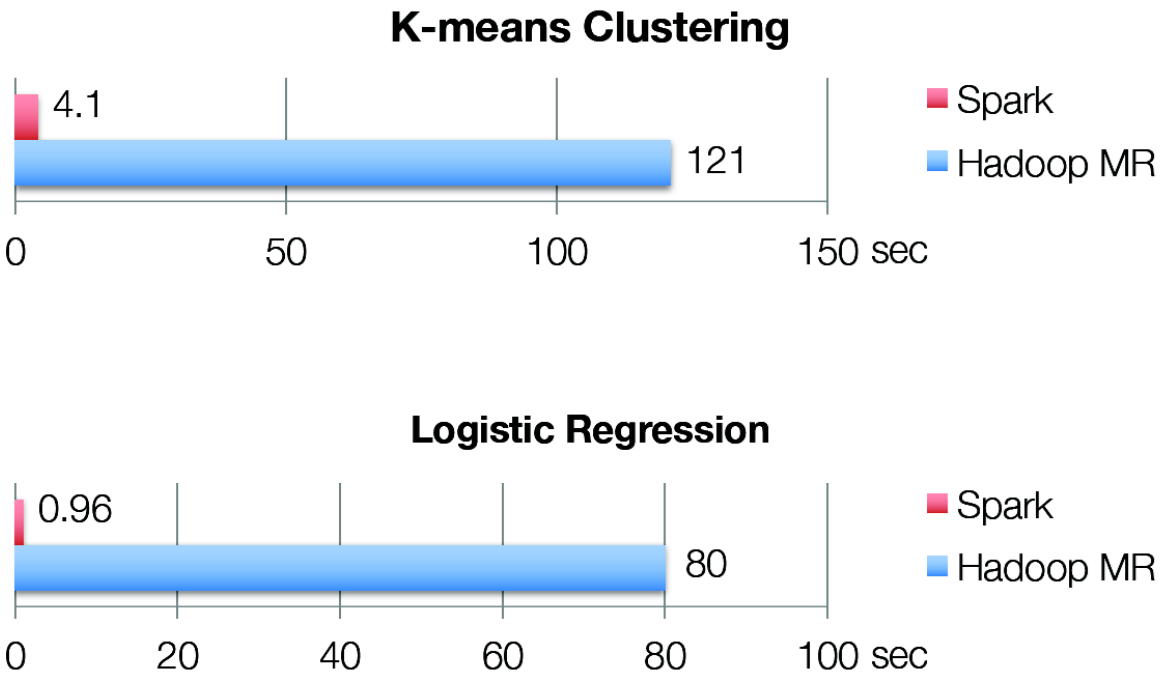
Apache Spark

- Support ***interactive*** and ***streaming*** computations
  - In-memory, fault-tolerant storage abstraction, low-latency scheduling,...
- ***Easy*** to combine ***batch***, ***streaming***, and ***interactive*** computations
  - Spark execution engine supports all comp. models
- ***Easy*** to develop ***iterative*** algorithms
  - Scala interface, APIs for Java, Python, Hive QL, …
  - New frameworks targeted to graph based and ML algorithms
- ***Compatible*** with existing open source ecosystem
- ***Open source*** (Apache/BSD) and fully committed to release ***high quality*** software
  - Three-person software engineering team led by Matt Massie (creator of Ganglia, 5th Cloudera engineer)

Performance: Spark vs MapReduce (1)

- Iterative algorithms
  - Spark is faster ← a simplified data flow
  - Avoids materializing data on HDFS after each iteration
- Example: k-means algorithm, 1 iteration
  - HDFS Read
  - Map(Assign sample to closest centroid)
  - GroupBy(Centroid_ID)
  - NETWORK Shuffle
  - Reduce(Compute new centroids)
  - HDFS Write

Performance: Spark vs MapReduce (2)



Spark in the Real World (I)

- **Uber** –gathers terabytes of event data from its mobile users every day.
  - Uses Kafka, Spark Streaming, and HDFS, to build a continuous ETL pipeline
  - Converts raw unstructured event data into structured data as it is collected
  - Uses it further for customer analytics and optimization of operations

- **Pinterest** – uses a Spark ETL pipeline
  - Leverages Spark Streaming to gain immediate insight into how users all over the world are engaging with Pins—in real time.
  - Can make relevant recommendations as people navigate the site
  - Recommends related Pins
  - Determine which products to buy, or destinations to visit

Spark in the Real World (II)

Here are other Real-World Use Cases:

- **Conviva** – analyzing 4 million video feeds per month
  - Uses Spark to reduce customer churn by optimizing video streams and managing live video traffic
  - Maintains a consistently smooth and high quality viewing experience.

- **Capital One** – using Spark to understand customers in a rapid way.
  - Develops next generation of financial products and services
  - Finds the attributes and patterns of increased probability for fraud

- **Netflix** – leveraging Spark for insights of user viewing habits
  - Recommends movies to them
  - Adopts user data for content creation

Spark: when not to use

- Even though Spark is versatile, that does not mean Spark's in-memory capabilities are the best fit for all use cases:
  - For many simple use cases, Apache MapReduce and Hive might be a more appropriate choice than Spark
  - Spark was not designed as a multi-user environment
  - Spark users are required to know that the memory resources they have is sufficient for a dataset
  - Adding more users adds complications, since the users will have to coordinate memory usage to run code

Lecture Break

Spark

In-Memory Cluster Computing for Iterative and Interactive Applications

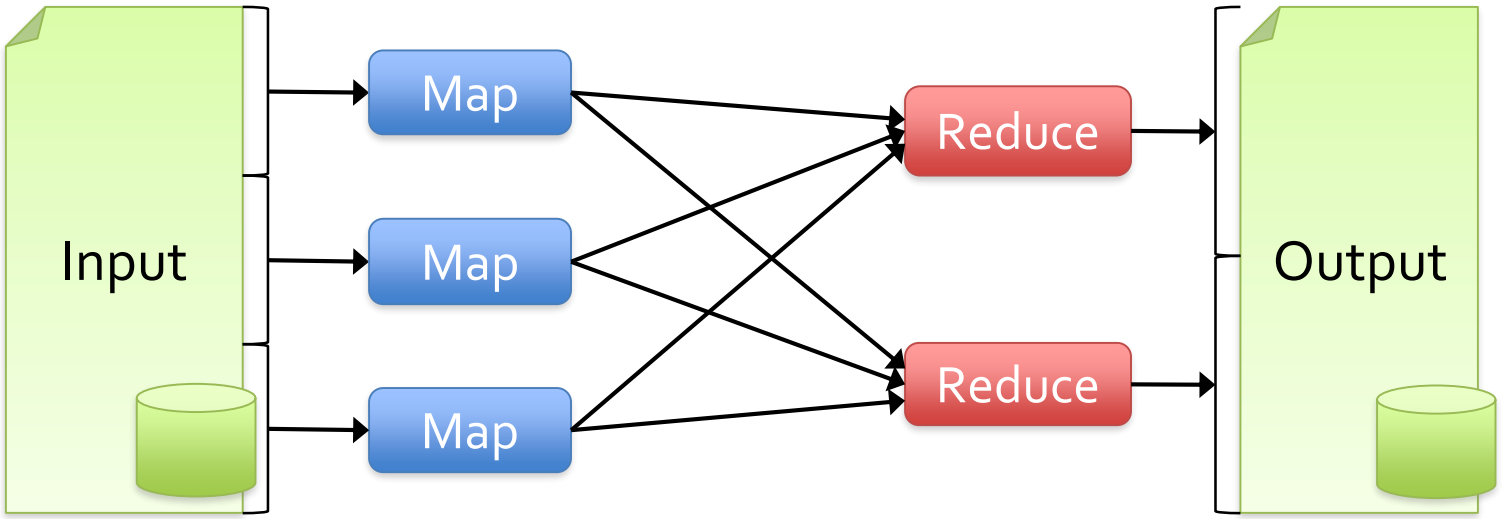
Background

- Commodity clusters have become an important computing platform for a variety of applications
  - **In industry:** search, machine translation, ad targeting, …
  - **In research:** bioinformatics, NLP, climate simulation, …
- High-level distributed cluster programming models such as MapReduce power many of these apps
- *Theme of this work: provide similarly powerful abstractions for a broad class of applications*

Motivation

Current popular programming models for clusters transform data from stable storage to stable storage

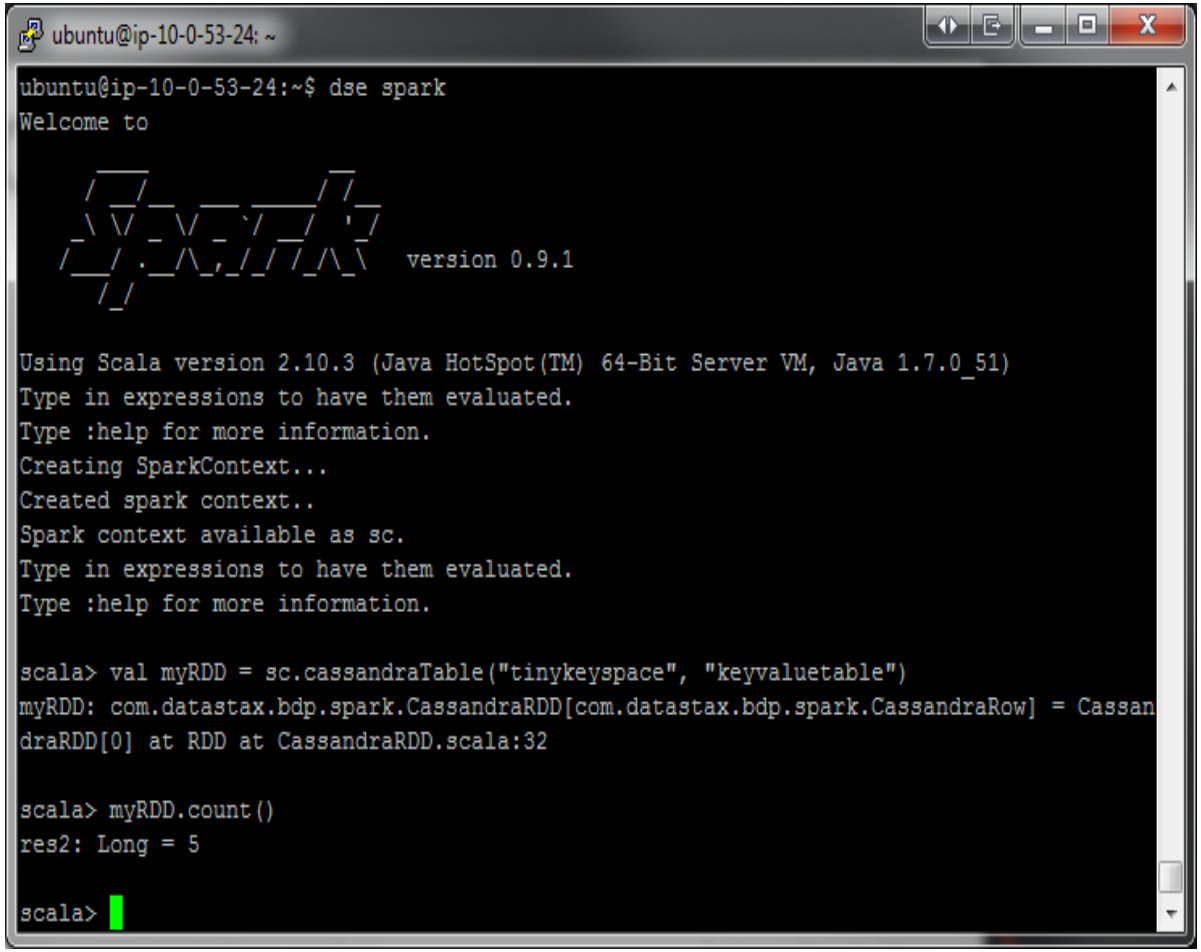
e.g., MapReduce:



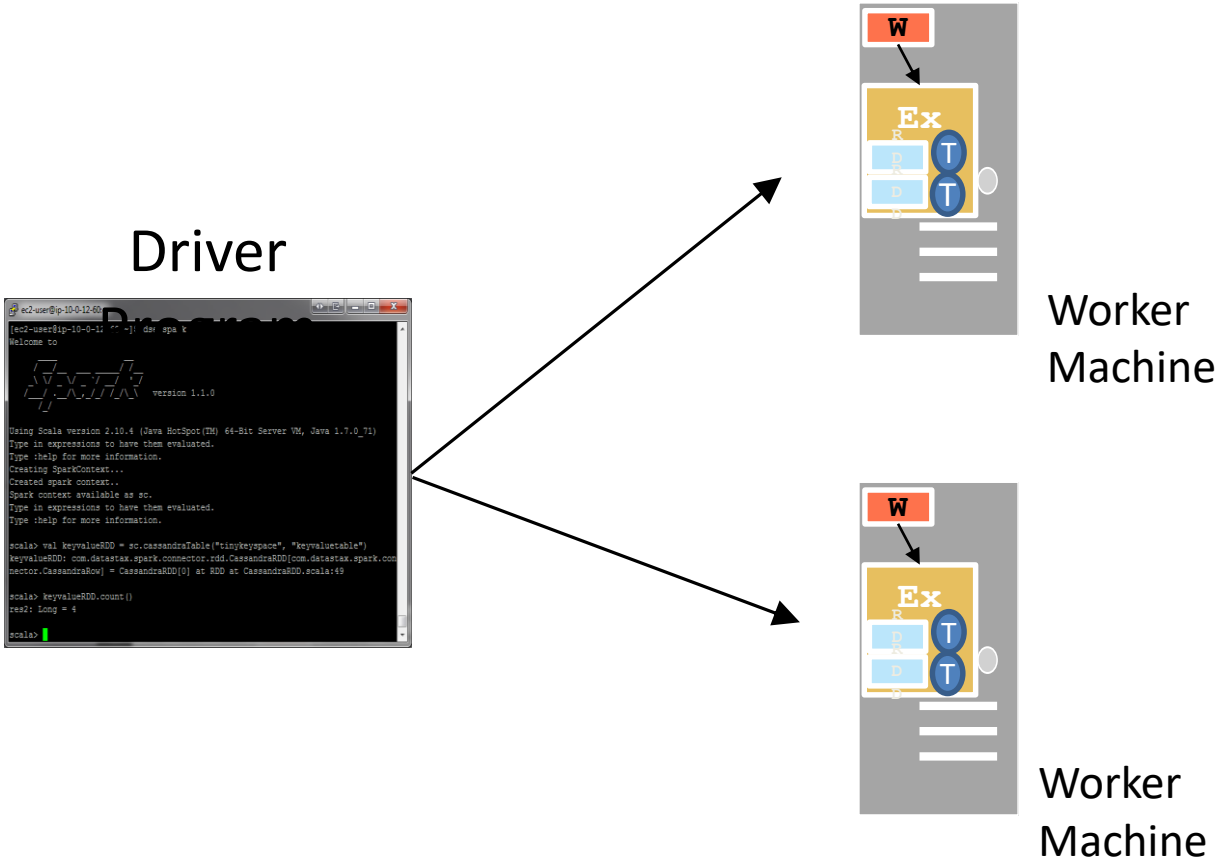
Motivation

- Acyclic Data Flow (e.g. Directed Acyclic Graph (DAG)) is a powerful abstraction, but is not efficient for applications that repeatedly reuse a *working set* of data:
  - **Iterative** algorithms (many in machine learning)
  - **Interactive** data mining tools (R, Excel, Python)
- Spark makes working set a first-class concept to efficiently support these apps

INTERACTIVE SHELL

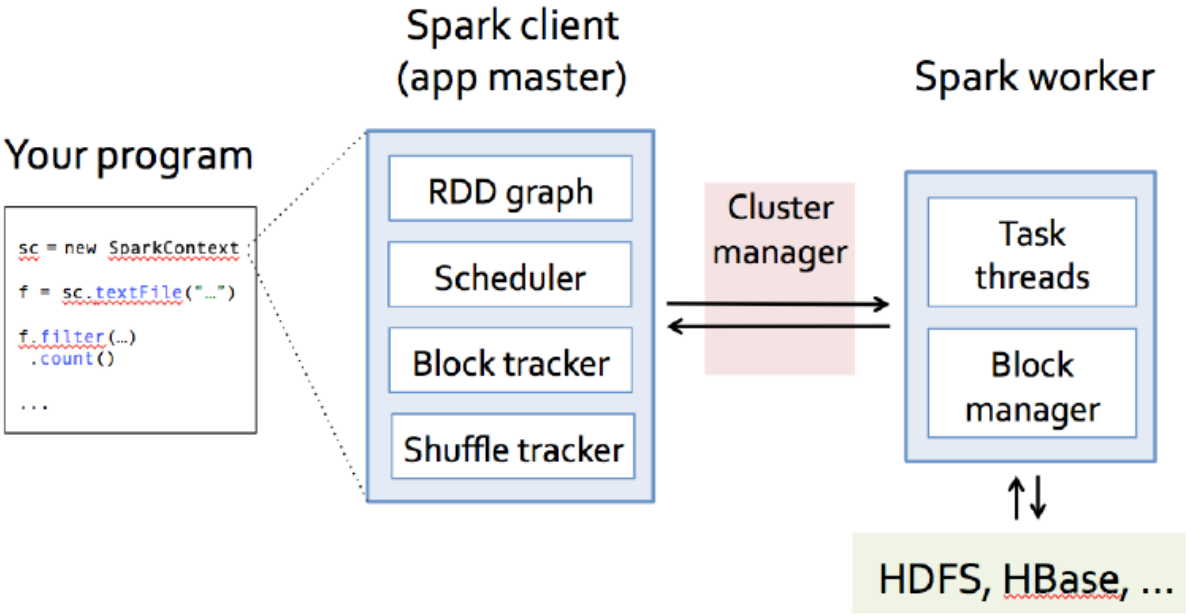


(Scala & Python only)


Driver
Program
W
Ex
T
T
Worker
Machine
W
Ex
T
T
Worker
Machine

Spark Context (sc)

- Spark Context (sc) works as a client and represents connection to a Spark cluster



Spark Context (sc)

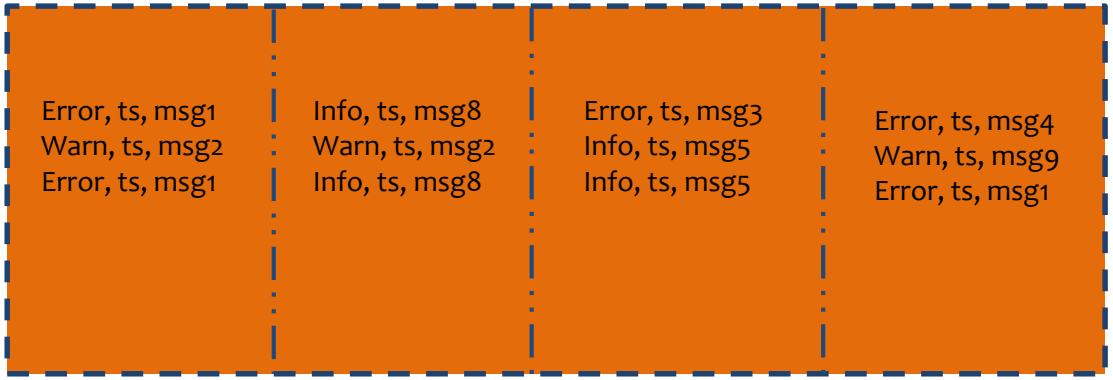
- Provide distributed memory abstractions for clusters to support applications computing with working sets in memory
- Retain the attractive properties of MapReduce:
  - Fault tolerance (for crashes & stragglers)
  - Data locality
  - Scalability

Solution: augment data flow model with "Resilient Distributed Datasets" (RDDs)

Spark Programming Model

- Resilient Distributed Datasets (RDDs)
  - Immutable collections partitioned across cluster that can be rebuilt if a partition is lost
  - Created by transforming data in stable storage using data flow operators (map, filter, group-by, …)
  - Can be *cached* across parallel operations
- Parallel operations on RDDs
  - reduce, collect, count, save, …
- Restricted shared variables
  - Accumulators, broadcast variables

RDD with 4 partitions



logLinesRDD

An RDD can be created 2 ways:

- Parallelize a collection
- Read data from an external source (S3, C*, HDFS, etc)

PARALLELIZE

```
# Parallelize in Python
wordsRDD = sc.parallelize([“fish", “cats“, “dogs”])
```

- Take an existing in-memory collection and pass it to SparkContext (**sc**) parallelize method

- Not generally used outside of prototyping and testing since it requires entire dataset in memory on one machine

```
// Parallelize in Scala
val wordsRDD= sc.parallelize(List("fish", "cats",
"dogs"))
```

```
// Parallelize in Java
JavaRDD<String> wordsRDD = sc.parallelize(Arrays.asList(“fish",
“cats“, “dogs”));
```

Spark Essentials: *RDD*

Spark can create RDDs from any file stored in HDFS or other storage systems supported by Hadoop, e.g., local file system, Amazon S3, Hypertable, HBase, etc.

Spark supports text files, SequenceFiles, and any other Hadoop `InputFormat`, and can also take a directory or a glob (e.g. `/data/201404*`)

READ FROM TEXT FILE

```
# Read a local txt file in Python
linesRDD = sc.textFile("/path/to/README.md")
```

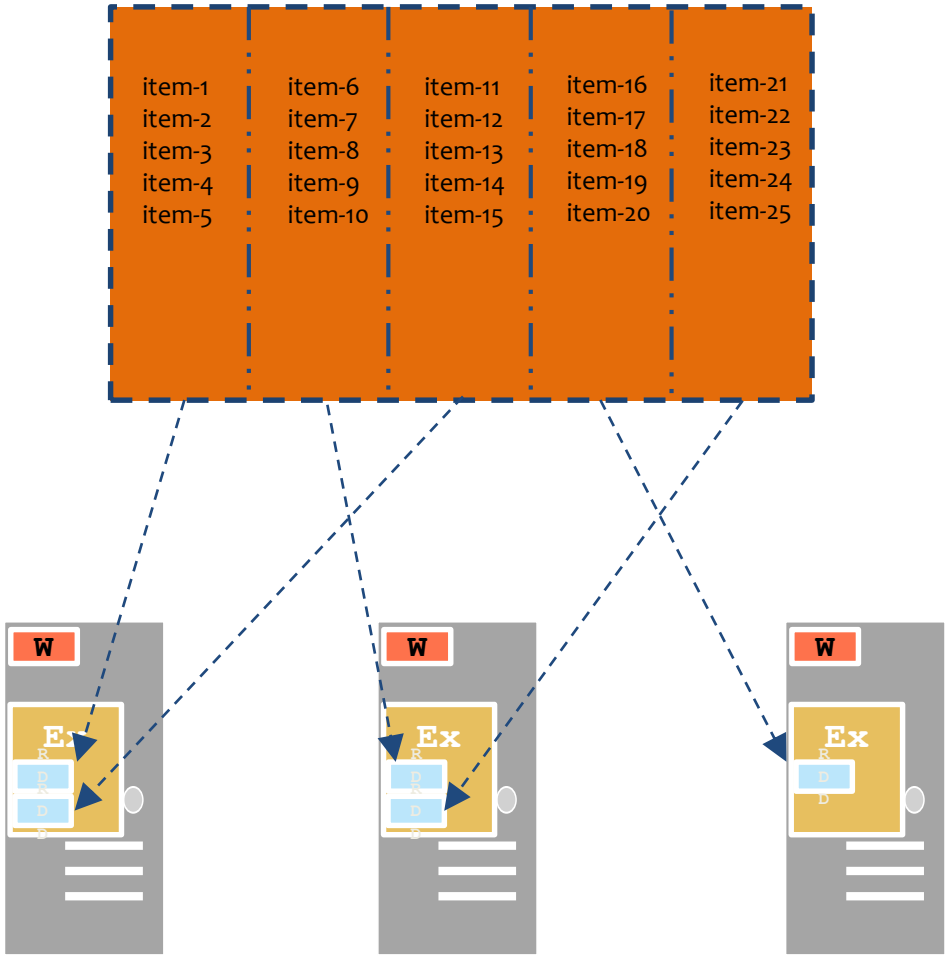
- There are other methods to read data from HDFS, C*, S3, HBase, etc.

```
// Read a local txt file in Scala
val linesRDD = sc.textFile("/path/to/README.md")
```

```
// Read a local txt file in Java
JavaRDD<String> lines = sc.textFile("/path/to/README.md");
```

more partitions = more parallelism

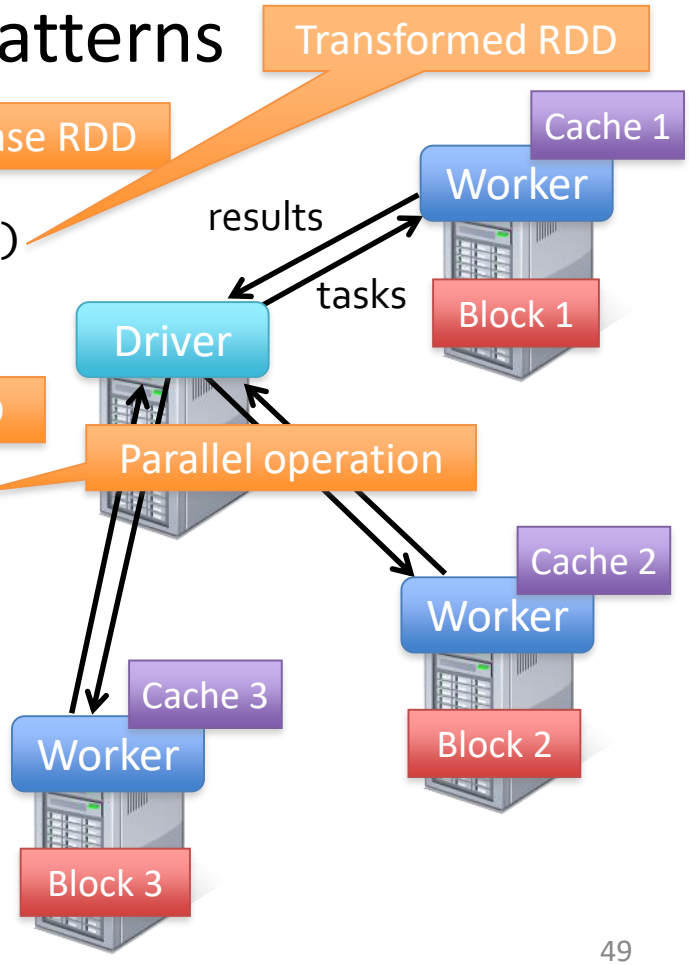
RDD



Example: Log Mining

- Load error messages from a log into memory, then interactively search for various patterns

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
messages = errors.map(_.split('\t')(2))
cachedMsgs = messages.cache()

cachedMsgs.filter(_.contains("foo")).count
cachedMsgs.filter(_.contains("bar")).count
. . .
```



Result: full-text search of Wikipedia in < 1 sec (vs 20 sec for on-disk data)

RDDs in More Detail

- An RDD is an immutable, partitioned, logical collection of data records
  - Need not be materialized, but rather contains information to rebuild a dataset from stable storage

- Partitioning can be based on a key in each record (using hash or range partitioning)

- Built using bulk transformations on other RDDs

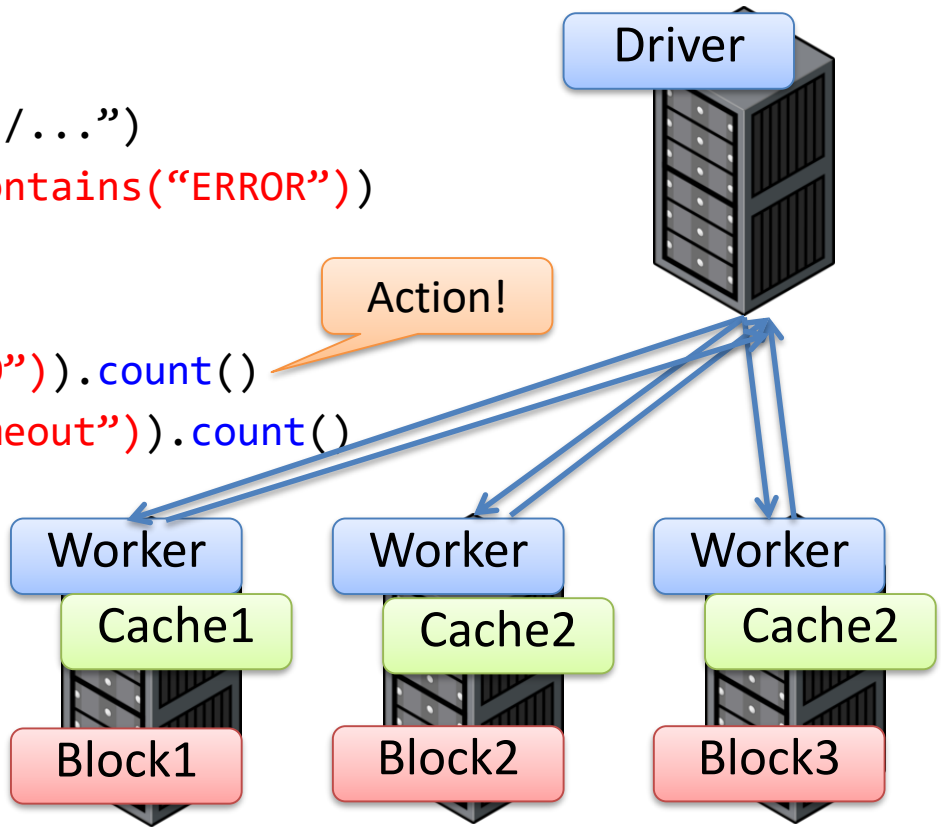
- Can be cached for future reuse

RDDs in More Detail

- RDDs represent data or transformations on data
- RDDs can be created from Hadoop InputFormats (such as HDFS files), “parallelize()” datasets, or by transforming other RDDs (you can stack RDDs)
- Actions can be applied to RDDs; actions force calculations and return values
- Lazy evaluation: Nothing computed until an action requires it
- RDDs are best suited for applications that apply the same operation to all elements of a dataset
  - Less suitable for applications that make asynchronous fine-grained updates to shared state

Lazy Evaluation Example

```
val log = sc.textFile("hdfs://...")
val errors = file.filter(_.contains("ERROR"))
errors.cache()

errors.filter(_.contains("I/O")).count()
errors.filter(_.contains("timeout")).count()
```



Lazy Execution of RDDs (1)

Data in RDDs is not processed until an action is performed

File: purplecow.txt

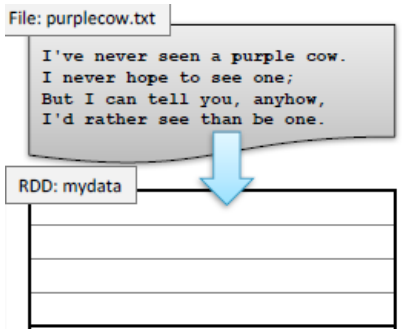
```
I've never seen a purple cow.
I never hope to see one;
But I can tell you, anyhow,
I'd rather see than be one.
```

```
>
```

Lazy Execution of RDDs (2)

Data in RDDs is not processed until an action is performed

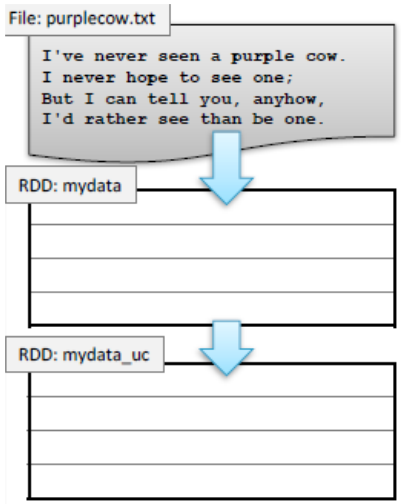
```
> val mydata = sc.textFile("purplecow.txt")
```



Lazy Execution of RDDs (3)

Data in RDDs is not processed until an action is performed

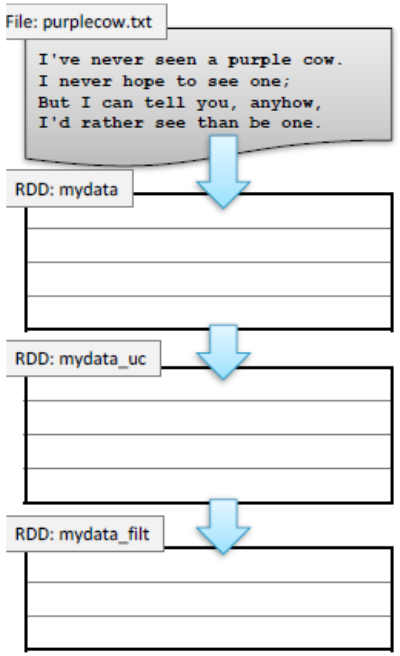
```
> val mydata = sc.textFile("purplecow.txt")
> val mydata_uc = mydata.map(line =>
  line.toUpperCase())
```



Lazy Execution of RDDs (4)

Data in RDDs is not processed until an action is performed

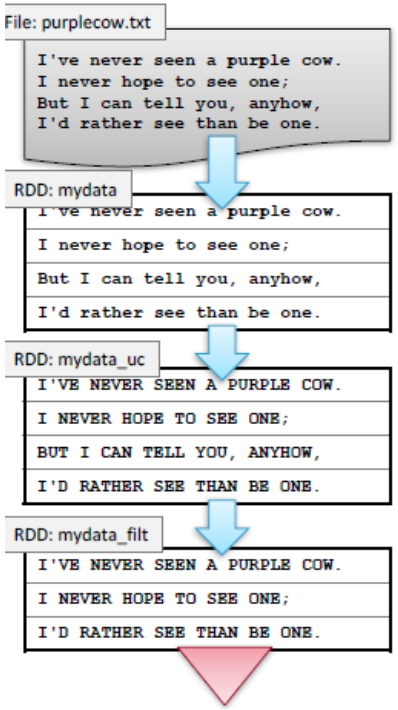
```
> val mydata = sc.textFile("purplecow.txt")
> val mydata_uc = mydata.map(line =>
  line.toUpperCase())
> val mydata_filt = mydata_uc.filter(line
  => line.startsWith("I"))
```



Lazy Execution of RDDs (5)

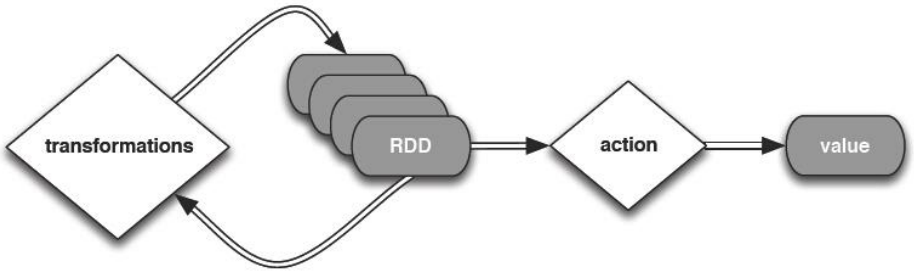
Data in RDDs is not processed until an action is performed

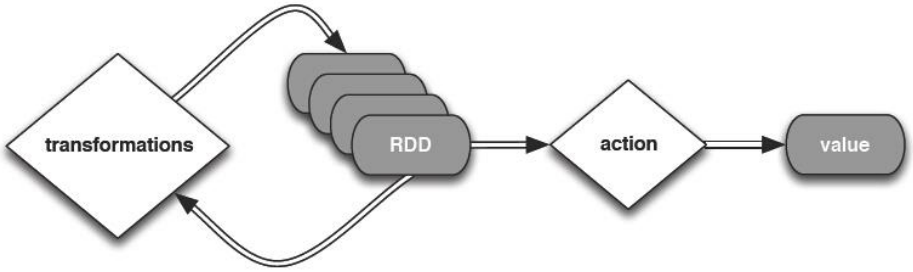
```
> val mydata = sc.textFile("purplecow.txt")
> val mydata_uc = mydata.map(line =>
  line.toUpperCase())
> val mydata_filt = mydata_uc.filter(line
  => line.startsWith("I"))
> mydata_filt.count()
3
```



RDD Operations

Transformations (define a new RDD)	**Actions** (return a result to driver)
map	reduce
filter	collect
sample	count
union	save
groupByKey	lookupKey
reduceByKey	…
join	
cache	





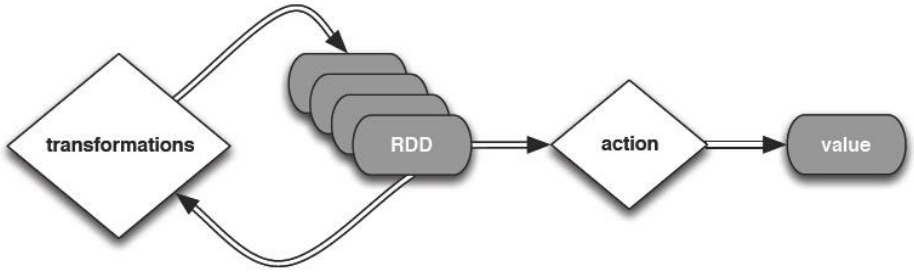
Spark Essentials: *Transformations*

Transformations create a new dataset from an existing one (because RDD is immutable)

All transformations in Spark are *lazy*: they do not compute their results right away – instead they remember the transformations applied to some base dataset

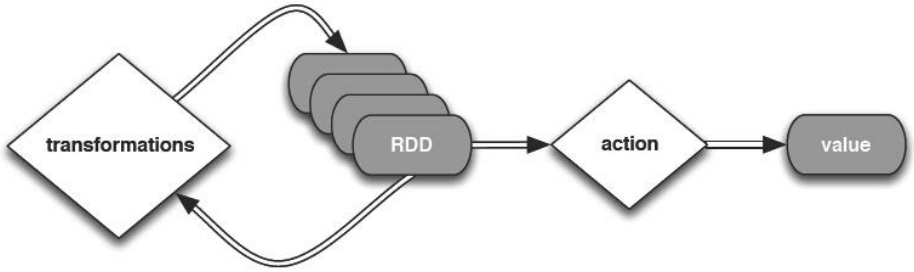
- optimize the required calculations
- recover from lost data partitions

Spark Essentials: *Transformations*



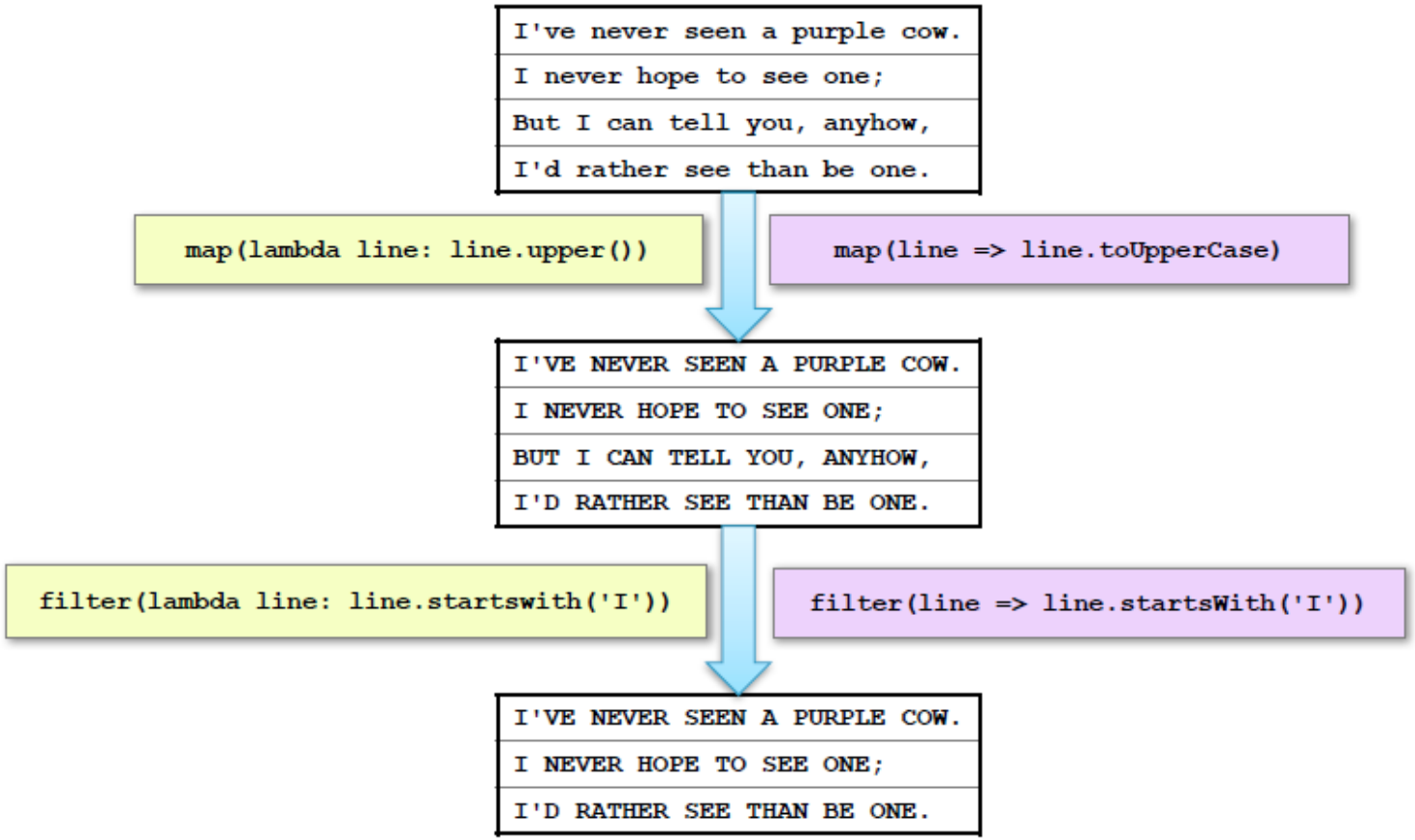
transformation	*description*
map(*func*)	return a new distributed dataset formed by passing each element of the source through a function *func*
filter(*func*)	return a new dataset formed by selecting those elements of the source on which *func* returns true
flatMap(*func*)	similar to map, but each input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

Spark Essentials: *Transformations*

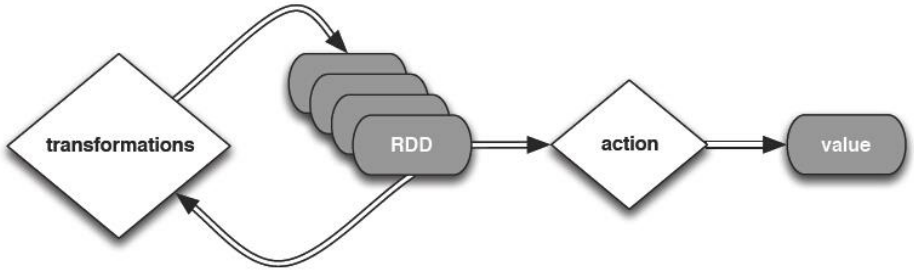


transformation	*description*
groupByKey([*numTasks*]**)**	when called on a dataset of `(K, V)` pairs, returns a dataset of `(K, Seq[V])` pairs
reduceByKey(*func*, [*numTasks*]**)**	when called on a dataset of `(K, V)` pairs, returns a dataset of `(K, V)` pairs where the values for each key are aggregated using the given reduce function
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, as specified in the boolean ascending argument
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
cogroup(*otherDataset*, [*numTasks*]**)**	when called on datasets of type `(K, V)` and `(K, W)`, returns a dataset of `(K, Seq[V], Seq[W])` tuples – 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)

Example: map and filter Transformations

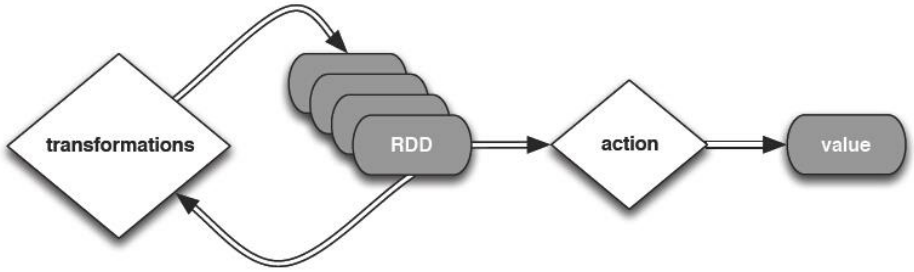


Spark Essentials: *Actions*



action	*description*
reduce(*func***)**	aggregate the elements of the dataset using a function *func* (which takes two arguments and returns one), and should also 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 – 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
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 – currently not executed in parallel, instead the driver program computes all the elements
takeSample(*withReplacement, fraction, seed***)**	return an array with a random sample of *num* elements of the dataset, with or without replacement, using the given random number generator seed

Spark Essentials: *Actions*



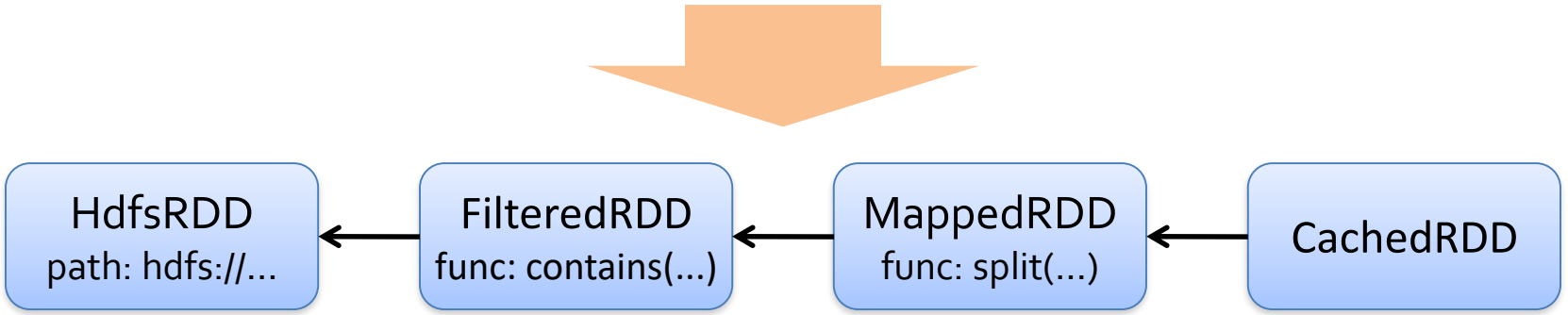
action	*description*
saveAsTextFile(*path*)	write the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem, HDFS or any other Hadoop-supported file system. Spark will call `toString` on each element to convert it to a line of text in the file
saveAsSequenceFile(*path*)	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. Only available on RDDs of key-value pairs that either implement Hadoop's `Writable` interface or are implicitly convertible to `Writable` (Spark includes conversions for basic types like `Int`, `Double`, `String`, etc).
countByKey()	only available on RDDs of type `(K, V)`. Returns a `Map` of `(K, Int)` pairs with the count of each key
foreach(*func*)	run a function *func* on each element of the dataset – usually done for side effects such as updating an accumulator variable or interacting with external storage systems

RDD Fault Tolerance

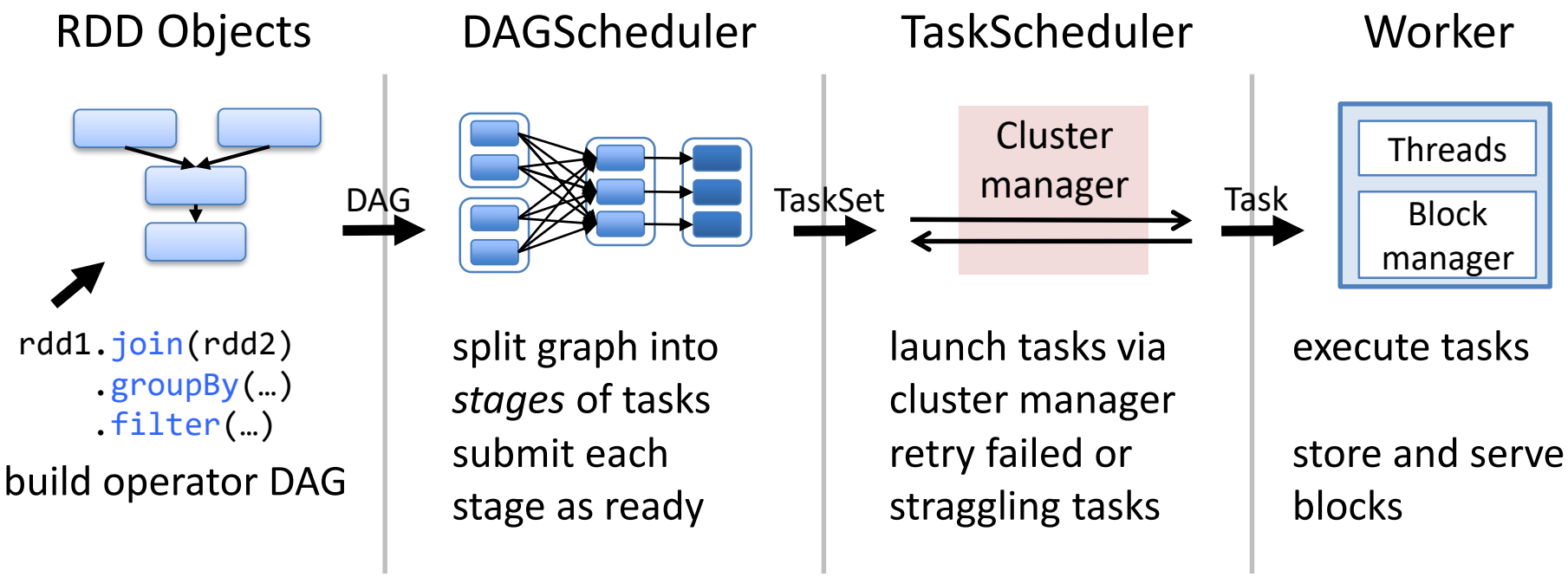
- RDDs maintain *lineage* information that can be used to reconstruct lost partitions

- e.g.:

```
cachedMsgs = textFile(...).filter(_.contains("error"))
                          .map(_.split('\t')(2))
                          .cache()
```



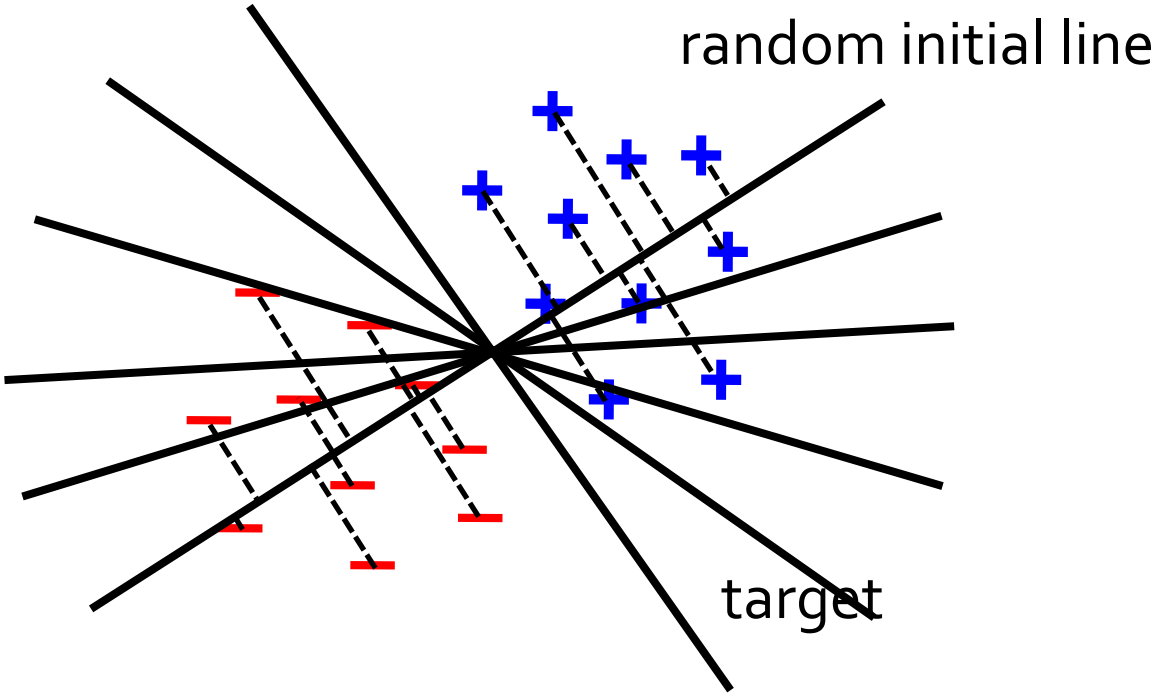
RDD Job scheduling



source: https://cwiki.apache.org/confluence/display/SPARK/Spark+Internals

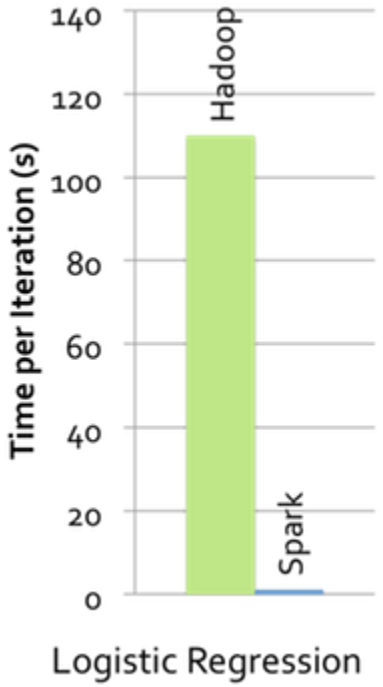
Example 1: Logistic Regression

- Goal: find best line separating two sets of points



Example 1: Logistic Regression

```
……
……
 val data = sc.textFile(……).map(CustomizedReadPoints).cache()
 var theta = Seq.fill(D)(Random.nextInt)
……
 for (i <- 1 to ITERATIONS) {
        val gradient = data.map(p =>
        (1 / (1 + exp(-(dotproduct(theta, p.x)))) - p.y) * p.x).reduce(_ + _)
        theta= theta - alpha*gradient
 }
……
 println("Final theta: " + theta)
……
```



$$\theta_j := \theta_j - \alpha \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$$

where $h_\theta(x) = \frac{1}{1 + e^{-\theta^T x}}$

https://databricks.com/blog/2013/11/21/putting-spark-to-use.html

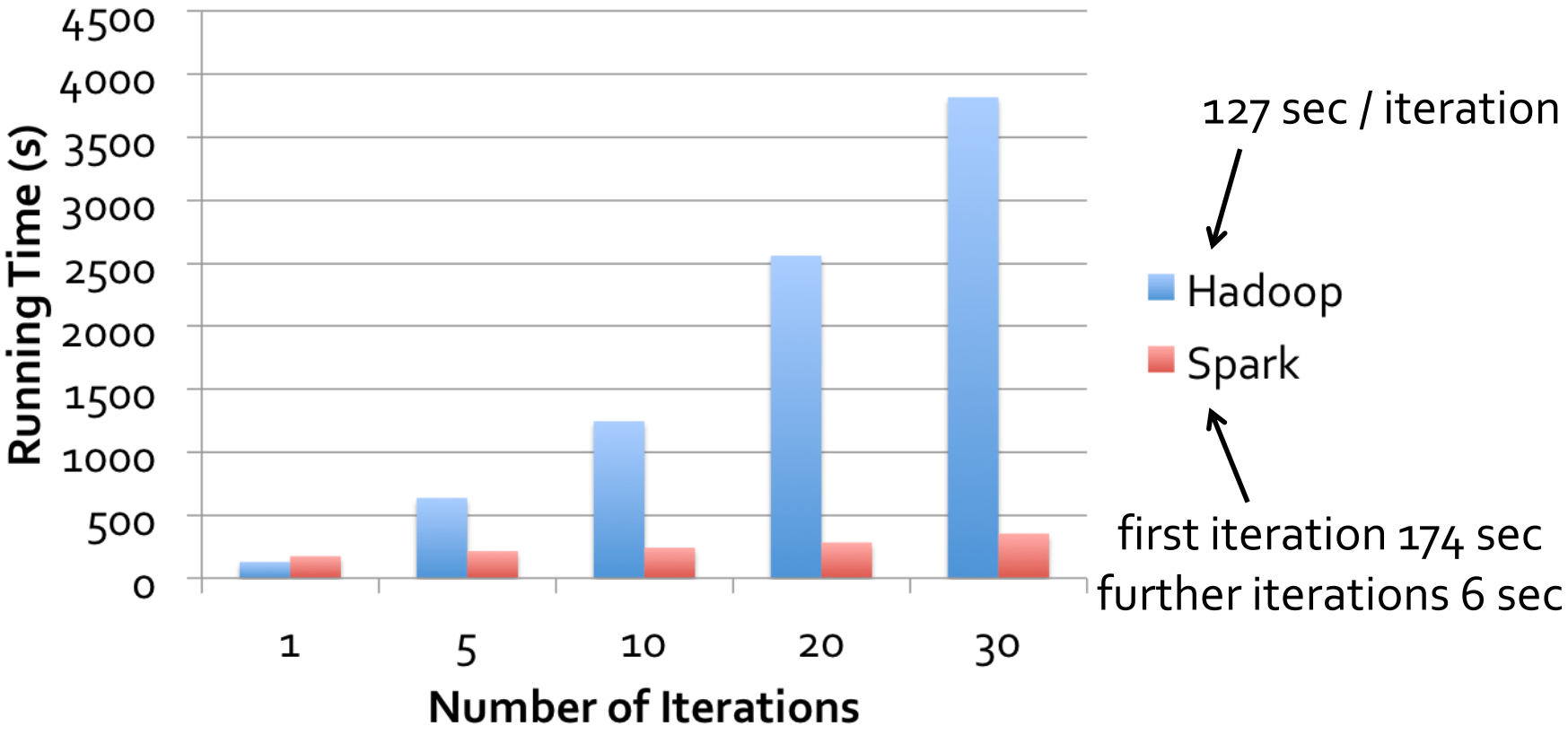
https://stackoverflow.com/questions/45089875/logistic-regression-not-generalizing

Scala Java Python R

More details on parameters can be found in the Scala API documentation.

```scala
import org.apache.spark.ml.classification.LogisticRegression

// Load training data
val training = spark.read.format("libsvm").load("data/mllib/sample_libsvm_data.txt")

val lr = new LogisticRegression()
  .setMaxIter(10)
  .setRegParam(0.3)
  .setElasticNetParam(0.8)

// Fit the model
val lrModel = lr.fit(training)

// Print the coefficients and intercept for logistic regression
println(s"Coefficients: ${lrModel.coefficients} Intercept: ${lrModel.intercept}")

// We can also use the multinomial family for binary classification
val mlr = new LogisticRegression()
  .setMaxIter(10)
  .setRegParam(0.3)
  .setElasticNetParam(0.8)
  .setFamily("multinomial")

val mlrModel = mlr.fit(training)

// Print the coefficients and intercepts for logistic regression with multinomial family
println(s"Multinomial coefficients: ${mlrModel.coefficientMatrix}")
println(s"Multinomial intercepts: ${mlrModel.interceptVector}")
```

Logistic Regression Performance



Example 2: MapReduce

- MapReduce data flow can be expressed using RDD transformations

```
res = data.flatMap(rec => myMapFunc(rec))
          .groupByKey()
          .map((key, vals) => myReduceFunc(key, vals))
```

Or with combiners:

```
res = data.flatMap(rec => myMapFunc(rec))
          .reduceByKey(myCombiner)
          .map((key, val) => myReduceFunc(key, val))
```

Simple Spark Apps: *WordCount*

```
public class WordCount {
  public static class TokenizerMapper
       extends Mapper<Object, Text, Text, IntWritable>{

    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(Object key, Text value, Context context
                    ) throws IOException, InterruptedException {
      StringTokenizer itr = new StringTokenizer(value.toString());
      while (itr.hasMoreTokens()) {
        word.set(itr.nextToken());
        context.write(word, one);
      }
    }
  }

  public static class IntSumReducer
       extends Reducer<Text,IntWritable,Text,IntWritable> {
    private IntWritable result = new IntWritable();

    public void reduce(Text key, Iterable<IntWritable> values,
                       Context context
                       ) throws IOException, InterruptedException {
      int sum = 0;
      for (IntWritable val : values) {
        sum += val.get();
      }
      result.set(sum);
      context.write(key, result);
    }
  }

  public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    String[] otherArgs = new GenericOptionsParser(conf, args).getRemainingArgs();
    if (otherArgs.length < 2) {
      System.err.println("Usage: wordcount <in> [<in>...] <out>");
      System.exit(2);
    }
    Job job = new Job(conf, "word count");
    job.setJarByClass(WordCount.class);
    job.setMapperClass(TokenizerMapper.class);
    job.setCombinerClass(IntSumReducer.class);
    job.setReducerClass(IntSumReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    for (int i = 0; i < otherArgs.length - 1; ++i) {
      FileInputFormat.addInputPath(job, new Path(otherArgs[i]));
    }
    FileOutputFormat.setOutputPath(job,
      new Path(otherArgs[otherArgs.length - 1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```

WordCount in 50+ lines of Java MR

```
val f = sc.textFile(inputPath)
val w = f.flatMap(l => l.split(" ")).map(word => (word, 1)).cache()
w.reduceByKey(_ + _).saveAsText(outputPath)
```

WordCount in 3 lines of Spark

Simple Spark Apps: *WordCount*

Scala:

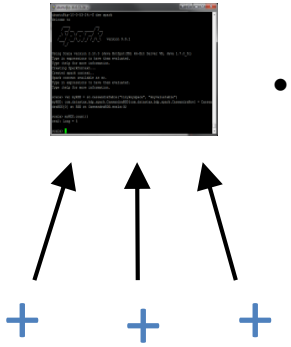
```
val f = sc.textFile("README.md")
val wc = f.flatMap(l => l.split(" ")).map(word => (word, 1)).reduceByKey(_ + _)
wc.saveAsTextFile("wc_out")
```

Python:

```
from operator import add
f = sc.textFile("README.md")
wc = f.flatMap(lambda x: x.split(' ')).map(lambda x: (x, 1)).reduceByKey(add)
wc.saveAsTextFile("wc_out")
```

Spark supports 2 types of shared variables:

- **Broadcast variables** – allows your program to efficiently send a large, read-only value to all the worker nodes for use in one or more Spark operations. Like sending a large, read-only lookup table to all the nodes.



- **Accumulators** – allows you to aggregate values from worker nodes back to the driver program. Can be used to count the # of errors seen in an RDD of lines spread across 100s of nodes. Only the driver can access the value of an accumulator, tasks cannot. For tasks, accumulators are write-only.

BROADCAST VARIABLES

Broadcast variables let programmer keep a read-only variable cached on each machine rather than shipping a copy of it with tasks

For example, to give every node a copy of a large input dataset efficiently

Spark also attempts to distribute broadcast variables using efficient broadcast algorithms to reduce communication cost

Scala:

```
val broadcastVar = sc.broadcast(Array(1, 2, 3))
broadcastVar.value
```

Python:

```
broadcastVar = sc.broadcast(list(range(1, 4)))
broadcastVar.value
```

ACCUMULATORS

Accumulators are variables that can only be “added” through an *associative* operation

Used to implement counters and sums, efficiently in parallel

Spark natively supports accumulators of numeric value types and standard mutable collections, and programmers can extend for new types

Only the driver program can read an accumulator’s value, not the tasks

ACCUMULATORS

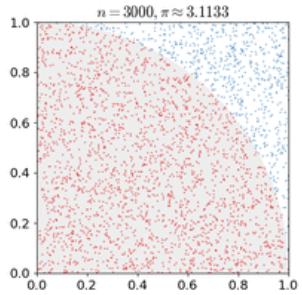
Scala:

```
val accum = sc.accumulator(0)

sc.parallelize(Array(1, 2, 3, 4)).foreach(x =>
accum += x)

accum.value
```

Python:

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

rdd.foreach(f)

accum.value
```

wikipedia.org/wiki/Monte_Carlo_method




```scala
import scala.math.random
import org.apache.spark._

/** Computes an approximation to pi */
object SparkPi {
  def main(args: Array[String]) {
    val conf = new SparkConf().setAppName("Spark Pi")
    val spark = new SparkContext(conf)

    val slices = if (args.length > 0) args(0).toInt else 2
    val n = 100000 * slices

    val count = spark.parallelize(1 to n, slices).map { i =>
      val x = random * 2 - 1
      val y = random * 2 - 1
      if (x*x + y*y < 1) 1 else 0
    }.reduce(_ + _)

    println("Pi is roughly " + 4.0 * count / n)
    spark.stop()
  }
}
```

DATABRICKS

Spark Text Search Example

```scala
val textFile = sc.textFile("hdfs://...")

// Creates a DataFrame having a single column named "line"
val df = textFile.toDF("line")
val errors = df.filter(col("line").like("%ERROR%"))
// Counts all the errors
errors.count()
// Counts errors mentioning MySQL
errors.filter(col("line").like("%MySQL%")).count()
// Fetches the MySQL errors as an array of strings
errors.filter(col("line").like("%MySQL%")).collect()
```

https://spark.apache.org/examples.html

Spark JDBC Example

```scala
// Creates a DataFrame based on a table named "people"
// stored in a MySQL database.
val url =
  "jdbc:mysql://yourIP:yourPort/test?user=yourUsername;password=yourPassword"
val df = sqlContext
  .read
  .format("jdbc")
  .option("url", url)
  .option("dbtable", "people")
  .load()

// Looks the schema of this DataFrame.
df.printSchema()

// Counts people by age
val countsByAge = df.groupBy("age").count()
countsByAge.show()

// Saves countsByAge to S3 in the JSON format.
countsByAge.write.format("json").save("s3a://...")
```

https://spark.apache.org/examples.html

Spark SQL Example

```
val sqlContext = new org.apache.spark.sql.SQLContext(sc)
import sqlContext._

// Define the schema using a case class.
case class Person(name: String, age: Int)

// Create an RDD of Person objects and register it as a table.
val people = sc.textFile("examples/src/main/resources/
people.txt").map(_.split(",")).map(p => Person(p(0), p(1).trim.toInt))

people.registerAsTable("people")

// SQL statements can be run by using the sql methods provided by sqlContext.
val teenagers = sql("SELECT name FROM people WHERE age >= 13 AND age <= 19")

// The results of SQL queries are SchemaRDDs and support all the
// normal RDD operations.
// The columns of a row in the result can be accessed by ordinal.
teenagers.map(t => "Name: " + t(0)).collect().foreach(println)
```

DATABRICKS

Other Spark Applications

- Twitter spam classification
  - –http://www.icir.org/vern/papers/monarch-oak11.pdf
- EM algorithm for traffic prediction
  - –https://cs.stanford.edu/~matei/papers/2011/socc_mobile_millenniu m.pdf
- K-means clustering
- Alternating Least Squares matrix factorization
- In-memory OLAP aggregation on Hive data
- Deep Learning back-propagation training
- Other iterative applications……

MapReduce vs. Spark for Large Scale Data Analysis

- MapReduce and Spark are two very popular open source cluster computing frameworks for big data
- These frameworks hide the complexity of task parallelism and fault-tolerance, by exposing simple programming APIs to users

Tasks	Word Count	K-means	Page-Rank
MapReduce			
Spark			

Conclusion

- By making distributed datasets a first-class primitive, Spark provides a simple and efficient programming model for stateful data analytics on a computer cluster.

- RDD provides:
  - Lineage info for fault recovery and debugging
  - Adjustable in-memory caching
  - Locality-aware parallel operations

- A big data suite of batch and interactive data analysis tools

http://spark.apache.org/docs/latest

Reference

Spark: Cluster Computing with Working Sets

Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, Ion Stoica
University of California, Berkeley

Abstract

MapReduce and its variants have been highly successful in implementing large-scale data-intensive applications on commodity clusters. However, most of these systems are built around an acyclic data flow model that is not suitable for other popular applications. This paper focuses on one such class of applications: those that reuse a working set of data across multiple parallel operations. This includes many iterative machine learning algorithms, as well as interactive data analysis tools. We propose a new framework called Spark that supports these applications while retaining the scalability and fault tolerance of MapReduce. To achieve these goals, Spark introduces an abstraction called resilient distributed datasets (RDDs). An RDD is a read-only collection of objects partitioned across a set of machines that can be rebuilt if a partition is lost. Spark can outperform Hadoop by 10x in iterative machine learning jobs, and can be used to interactively query a 39 GB dataset with sub-second response time.

1 Introduction

A new model of cluster computing has become widely popular, in which data-parallel computations are executed on clusters of unreliable machines by systems that automatically provide locality-aware scheduling, fault tolerance, and load balancing. MapReduce [11] pioneered this model, while systems like Dryad [17] and Map-Reduce-Merge [24] generalized the types of data flows supported. These systems achieve their scalability and fault tolerance by providing a programming model where the user creates acyclic data flow graphs to pass input data through a set of operators. This allows the underlying system to manage scheduling and to react to faults without user intervention.

While this data flow programming model is useful for a large class of applications, there are applications that cannot be expressed efficiently as acyclic data flows. In this paper, we focus on one such class of applications: those that reuse a *working set* of data across multiple parallel operations. This includes two use cases where we have seen Hadoop users report that MapReduce is deficient:

- **Iterative jobs**: Many common machine learning algorithms apply a function repeatedly to the same dataset to optimize a parameter (e.g., through gradient descent). While each iteration can be expressed as a MapReduce/Dryad job, each job must reload the data from disk, incurring a significant performance penalty.
- **Interactive analytics**: Hadoop is often used to run ad-hoc exploratory queries on large datasets, through SQL interfaces such as Pig [21] and Hive [1]. Ideally, a user would be able to load a dataset of interest into memory across a number of machines and query it repeatedly. However, with Hadoop, each query incurs significant latency (tens of seconds) because it runs as a separate MapReduce job and reads data from disk.

This paper presents a new cluster computing framework called Spark, which supports applications with working sets while providing similar scalability and fault tolerance properties to MapReduce.

The main abstraction in Spark is that of a *resilient distributed dataset* (RDD), which represents a read-only collection of objects partitioned across a set of machines that can be rebuilt if a partition is lost. Users can explicitly cache an RDD in memory across machines and reuse it in multiple MapReduce-like *parallel operations*. RDDs achieve fault tolerance through a notion of *lineage*: if a partition of an RDD is lost, the RDD has enough information about how it was derived from other RDDs to be able to rebuild just that partition. Although RDDs are not a general shared memory abstraction, they represent a sweet-spot between expressivity on the one hand and scalability and reliability on the other hand, and we have found them well-suited for a variety of applications.

Spark is implemented in Scala [5], a statically typed high-level programming language for the Java VM, and exposes a functional programming interface similar to DryadLINQ [25]. In addition, Spark can be used interactively from a modified version of the Scala interpreter, which allows the user to define RDDs, functions, variables and classes and use them in parallel operations on a cluster. We believe that Spark is the first system to allow an efficient, general-purpose programming language to be used interactively to process large datasets on a cluster.

Although our implementation of Spark is still a prototype, early experience with the system is encouraging. We show that Spark can outperform Hadoop by 10x in iterative machine learning workloads and can be used interactively to scan a 39 GB dataset with sub-second latency.

This paper is organized as follows. Section 2 describes

1

"The main abstraction in Spark is that of a resilient distributed dataset (RDD), which represents a read-only collection of objects partitioned across a set of machines that can be rebuilt if a partition is lost.

Users can explicitly cache an RDD in memory across machines and reuse it in multiple MapReduce-like parallel operations.

RDDs achieve fault tolerance through a notion of lineage: if a partition of an RDD is lost, the RDD has enough information about how it was derived from other RDDs to be able to rebuild just that partition."

June 2010

http://www.cs.berkeley.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
University of California, Berkeley

Abstract

We present Resilient Distributed Datasets (RDDs), a distributed memory abstraction that lets programmers perform in-memory computations on large clusters in a fault-tolerant manner. RDDs are motivated by two types of applications that current computing frameworks handle inefficiently: iterative algorithms and interactive data mining tools. In both cases, keeping data in memory can improve performance by an order of magnitude. To achieve fault tolerance efficiently, RDDs provide a restricted form of shared memory, based on coarse-grained transformations rather than fine-grained updates to shared state. However, we show that RDDs are expressive enough to capture a wide class of computations, including recent specialized programming models for iterative jobs, such as Pregel, and new applications that these models do not capture. We have implemented RDDs in a system called Spark, which we evaluate through a variety of user applications and benchmarks.

1 Introduction

Cluster computing frameworks like MapReduce [10] and Dryad [19] have been widely adopted for large-scale data analytics. These systems let users write parallel computations using a set of high-level operators, without having to worry about work distribution and fault tolerance.

Although current frameworks provide numerous abstractions for accessing a cluster's computational resources, they lack abstractions for leveraging distributed memory. This makes them inefficient for an important class of emerging applications: those that *reuse* intermediate results across multiple computations. Data reuse is common in many *iterative* machine learning and graph algorithms, including PageRank, K-means clustering, and logistic regression. Another compelling use case is *interactive* data mining, where a user runs multiple ad-hoc queries on the same subset of the data. Unfortunately, in most current frameworks, the only way to reuse data between computations (*e.g.*, between two MapReduce jobs) is to write it to an external stable storage system, *e.g.*, a distributed file system. This incurs substantial overheads due to data replication, disk I/O, and serialization, which can dominate application execution times.

Recognizing this problem, researchers have developed specialized frameworks for some applications that require data reuse. For example, Pregel [22] is a system for iterative graph computations that keeps intermediate data in memory, while HaLoop [7] offers an iterative MapReduce interface. However, these frameworks only support specific computation patterns (*e.g.*, looping a series of MapReduce steps), and perform data sharing implicitly for these patterns. They do not provide abstractions for more general reuse, *e.g.*, to let a user load several datasets into memory and run ad-hoc queries across them.

In this paper, we propose a new abstraction called *resilient distributed datasets (RDDs)* that enables efficient data reuse in a broad range of applications. RDDs are fault-tolerant, parallel data structures that let users explicitly persist intermediate results in memory, control their partitioning to optimize data placement, and manipulate them using a rich set of operators.

The main challenge in designing RDDs is defining a programming interface that can provide fault tolerance *efficiently*. Existing abstractions for in-memory storage on clusters, such as distributed shared memory [24], key-value stores [25], databases, and Piccolo [27], offer an interface based on fine-grained updates to mutable state (*e.g.*, cells in a table). With this interface, the only ways to provide fault tolerance are to replicate the data across machines or to log updates across machines. Both approaches are expensive for data-intensive workloads, as they require copying large amounts of data over the cluster network, whose bandwidth is far lower than that of RAM, and they incur substantial storage overhead.

In contrast to these systems, RDDs provide an interface based on *coarse-grained* transformations (*e.g.*, map, filter and join) that apply the same operation to many data items. This allows them to efficiently provide fault tolerance by logging the transformations used to build a dataset (its *lineage*) rather than the actual data.[1] If a partition of an RDD is lost, the RDD has enough information about how it was derived from other RDDs to recompute

[1]Checkpointing the data in some RDDs may be useful when a lineage chain grows large, however, and we discuss how to do it in §5.4.

"We present Resilient Distributed Datasets (RDDs), a distributed memory abstraction that lets programmers perform in-memory computations on large clusters in a fault-tolerant manner.

RDDs are motivated by two types of applications that current computing frameworks handle inefficiently: iterative algorithms and interactive data mining tools.

In both cases, keeping data in memory can improve performance by an order of magnitude."

"Best Paper Award and Honorable Mention for Community Award"

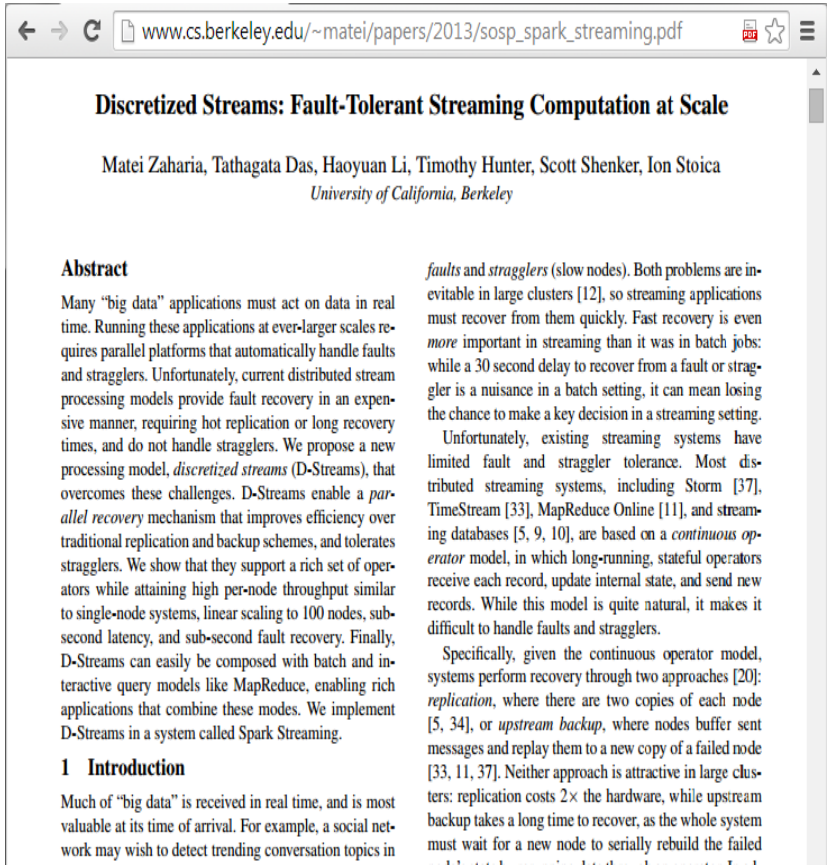
- Cited 400+ times!

April 2012

http://www.cs.berkeley.edu/~matei/papers/2012/nsdi_spark.pdf

STREAMING

Analyze real time streams of data in ½ second intervals



www.cs.berkeley.edu/~matei/papers/2013/sosp_spark_streaming.pdf

Discretized Streams: Fault-Tolerant Streaming Computation at Scale

Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, Ion Stoica
University of California, Berkeley

Abstract

Many "big data" applications must act on data in real time. Running these applications at ever-larger scales requires parallel platforms that automatically handle faults and stragglers. Unfortunately, current distributed stream processing models provide fault recovery in an expensive manner, requiring hot replication or long recovery times, and do not handle stragglers. We propose a new processing model, *discretized streams* (D-Streams), that overcomes these challenges. D-Streams enable a *parallel recovery* mechanism that improves efficiency over traditional replication and backup schemes, and tolerates stragglers. We show that they support a rich set of operators while attaining high per-node throughput similar to single-node systems, linear scaling to 100 nodes, sub-second latency, and sub-second fault recovery. Finally, D-Streams can easily be composed with batch and interactive query models like MapReduce, enabling rich applications that combine these modes. We implement D-Streams in a system called Spark Streaming.

1 Introduction

Much of "big data" is received in real time, and is most valuable at its time of arrival. For example, a social network may wish to detect trending conversation topics in *faults* and *stragglers* (slow nodes). Both problems are inevitable in large clusters [12], so streaming applications must recover from them quickly. Fast recovery is even *more* important in streaming than it was in batch jobs: while a 30 second delay to recover from a fault or straggler is a nuisance in a batch setting, it can mean losing the chance to make a key decision in a streaming setting.

Unfortunately, existing streaming systems have limited fault and straggler tolerance. Most distributed streaming systems, including Storm [37], TimeStream [33], MapReduce Online [11], and streaming databases [5, 9, 10], are based on a *continuous operator* model, in which long-running, stateful operators receive each record, update internal state, and send new records. While this model is quite natural, it makes it difficult to handle faults and stragglers.

Specifically, given the continuous operator model, systems perform recovery through two approaches [20]: *replication*, where there are two copies of each node [5, 34], or *upstream backup*, where nodes buffer sent messages and replay them to a new copy of a failed node [33, 11, 37]. Neither approach is attractive in large clusters: replication costs 2× the hardware, while upstream backup takes a long time to recover, as the whole system must wait for a new node to serially rebuild the failed

```
TwitterUtils.createStream(...)
    .filter(_.getText.contains("Spark"))
    .countByWindow(Seconds(5))
```

- 2 Streaming Paper(s) have been cited 138 times

SQL

Seemlessly mix SQL queries with Spark programs.

Spark SQL: Relational Data Processing in Spark

Michael Armbrust†, Reynold S. Xin†, Cheng Lian†, Yin Huai†, Davies Liu†, Joseph K. Bradley†, Xiangrui Meng†, Tomer Kaftan‡, Michael J. Franklin†‡, Ali Ghodsi†, Matei Zaharia†*

†Databricks Inc. *MIT CSAIL ‡AMPLab, UC Berkeley

ABSTRACT

Spark SQL is a new module in Apache Spark that integrates relational processing with Spark's functional programming API. Built on our experience with Shark, Spark SQL lets Spark programmers leverage the benefits of relational processing (*e.g.,* declarative queries and optimized storage), and lets SQL users call complex analytics libraries in Spark (*e.g.,* machine learning). Compared to previous systems, Spark SQL makes two main additions. First, it offers much tighter integration between relational and procedural processing, through a declarative DataFrame API that integrates with procedural Spark code. Second, it includes a highly extensible optimizer, Catalyst, built using features of the Scala programming language, that makes it easy to add composable rules, control code generation, and define extension points. Using Catalyst, we have built a variety of features (*e.g.,* schema inference for JSON, machine learning types, and query federation to external databases) tailored for the complex needs of modern data analysis. We see Spark SQL as an evolution of both SQL-on-Spark and of Spark itself, offering richer APIs and optimizations while keeping the benefits of the Spark programming model.

Categories and Subject Descriptors

H.2 [**Database Management**]: Systems

Keywords

Databases; Data Warehouse; Machine Learning; Spark; Hadoop

1 Introduction

Big data applications require a mix of processing techniques, data sources and storage formats. The earliest systems designed for these workloads, such as MapReduce, gave users a powerful, but

While the popularity of relational systems shows that users often prefer writing declarative queries, the relational approach is insufficient for many big data applications. First, users want to perform ETL to and from various data sources that might be semi- or unstructured, requiring custom code. Second, users want to perform advanced analytics, such as machine learning and graph processing, that are challenging to express in relational systems. In practice, we have observed that most data pipelines would ideally be expressed with a combination of both relational queries and complex procedural algorithms. Unfortunately, these two classes of systems—relational and procedural—have until now remained largely disjoint, forcing users to choose one paradigm or the other.

This paper describes our effort to combine both models in Spark SQL, a major new component in Apache Spark [39]. Spark SQL builds on our earlier SQL-on-Spark effort, called Shark. Rather than forcing users to pick between a relational or a procedural API, however, Spark SQL lets users seamlessly intermix the two.

Spark SQL bridges the gap between the two models through two contributions. First, Spark SQL provides a *DataFrame API* that can perform relational operations on both external data sources and Spark's built-in distributed collections. This API is similar to the widely used data frame concept in R [32], but evaluates operations lazily so that it can perform relational optimizations. Second, to support the wide range of data sources and algorithms in big data, Spark SQL introduces a novel extensible optimizer called *Catalyst*. Catalyst makes it easy to add data sources, optimization rules, and data types for domains such as machine learning.

The DataFrame API offers rich relational/procedural integration within Spark programs. DataFrames are collections of structured records that can be manipulated using Spark's procedural API, or using new relational APIs that allow richer optimizations. They can

```
sqlCtx = new HiveContext(sc)
results = sqlCtx.sql(
  "SELECT * FROM people")
names = results.map(lambda p:
p.name)
```

GRAPHX

Analyze networks of nodes and edges using graph processing

GraphX: A Resilient Distributed Graph System on Spark

Reynold S. Xin, Joseph E. Gonzalez, Michael J. Franklin, Ion Stoica

AMPLab, EECS, UC Berkeley
{rxin, jegonzal, franklin, istoica}@cs.berkeley.edu

ABSTRACT

From social networks to targeted advertising, big graphs capture the structure in data and are central to recent advances in machine learning and data mining. Unfortunately, directly applying existing data-parallel tools to graph computation tasks can be cumbersome and inefficient. The need for intuitive, scalable tools for graph computation has lead to the development of new *graph-parallel* systems (*e.g.*, Pregel, PowerGraph) which are designed to efficiently execute graph algorithms. Unfortunately, these new graph-parallel systems do not address the challenges of graph construction and transformation which are often just as problematic as the subsequent computation. Furthermore, existing graph-parallel systems provide limited fault-tolerance and support for interactive data mining.

We introduce GraphX, which combines the advantages of both data-parallel and graph-parallel systems by efficiently expressing graph computation within the Spark data-parallel framework. We leverage new ideas in distributed graph representation to efficiently distribute graphs as tabular data-structures. Similarly, we leverage advances in data-flow systems to exploit in-memory computation and fault-tolerance. We provide powerful new operations to simplify graph construction and transformation. Using these primitives we implement the PowerGraph and Pregel abstractions in less than 20 lines of code. Finally, by exploiting the Scala foundation of Spark, we enable users to interactively load, transform, and compute on massive graphs.

1. INTRODUCTION

From social networks to advertising and the web, big graphs can be found in a wide range of important applications. By modeling the and distributed systems. By abstracting away the challenges of large-scale distributed system design, these frameworks simplify the design, implementation, and application of new sophisticated graph algorithms to large-scale real-world graph problems.

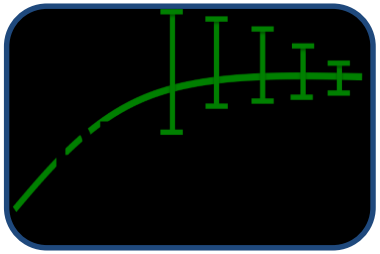
While existing graph-parallel frameworks share many common properties, each presents a slightly different view of graph computation tailored to either the originating domain or a specific family of graph algorithms and applications. Unfortunately, because each framework relies on a separate runtime, it is difficult to compose these abstractions. Furthermore, while these frameworks address the challenges of graph computation, they do not address the challenges of data ETL (preprocessing and construction) or the process of interpreting and applying the results of computation. Finally, few frameworks have built-in support for interactive graph computation.

Alternatively *data-parallel* systems like MapReduce and Spark [12] are designed for scalable data processing and are well suited to the task of graph construction (ETL). By exploiting data-parallelism, these systems are highly scalable and support a range of fault-tolerance strategies. More recent systems like Spark even enable interactive data processing. However, naively expressing graph computation and graph algorithms in these data-parallel abstractions can be challenging and typically leads to complex joins and excessive data movement that does not exploit the graph structure.

To address these challenges we introduce GraphX, a graph computation system which runs in the Spark data-parallel framework. GraphX extends Spark's Resilient Distributed Dataset (RDD) abstraction to introduce the Resilient Distributed Graph (RDG), which associates records with vertices and edges in a graph and provides a collection of expressive computational primitives. Using these

```
graph = Graph(vertices, edges)
messages =
spark.textFile("hdfs://...")
graph2 =
graph.joinVertices(messages) {
  (id, vertex, msg) => ...
}
```

https://amplab.cs.berkeley.edu/wp-content/uploads/2013/05/grades-graphx_with_fonts.pdf



SQL queries with Bounded Errors and Bounded Response Times

BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data

Sameer Agarwal†, Barzan Mozafari°, Aurojit Panda†, Henry Milner†, Samuel Madden°, Ion Stoica*†

†University of California, Berkeley °Massachusetts Institute of Technology *Conviva Inc.
{sameerag, apanda, henrym, istoica}@cs.berkeley.edu, {barzan, madden}@csail.mit.edu

Abstract

In this paper, we present BlinkDB, a massively parallel, approximate query engine for running interactive SQL queries on large volumes of data. BlinkDB allows users to trade-off query accuracy for response time, enabling interactive queries over massive data by running queries on data samples and presenting results annotated with meaningful error bars. To achieve this, BlinkDB uses two key ideas: (1) an adaptive optimization framework that builds and maintains a set of multi-dimensional stratified samples from original data over time, and (2) a dynamic sample selection strategy that selects an appropriately sized sample based on a query's accuracy or response time requirements. We evaluate BlinkDB against the well-known TPC-H benchmarks and a real-world analytic workload derived from Conviva Inc., a company that manages video distribution over the Internet. Our experiments on a 100 node cluster show that BlinkDB can answer queries on up to 17 TBs of data in less than 2 seconds (over 200× faster than Hive), within an error of 2-10%.

1. Introduction

Modern data analytics applications involve computing aggregates over a large number of records to *roll-up* web clicks, cessing of large amounts of data by trading result accuracy for response time and space. These techniques include sampling [10, 14], sketches [12], and on-line aggregation [15]. To illustrate the utility of such techniques, consider the following simple query that computes the average `SessionTime` over all users originating in `New York`:

```
SELECT AVG(SessionTime)
FROM Sessions
WHERE City = 'New York'
```

Suppose the `Sessions` table contains 100 million tuples for `New York`, and cannot fit in memory. In that case, the above query may take a long time to execute, since disk reads are expensive, and such a query would need multiple disk accesses to stream through all the tuples. Suppose we instead executed the same query on a sample containing only 10,000 `New York` tuples, such that the entire sample fits in memory. This would be orders of magnitude faster, while still providing an approximate result within a few percent of the actual value, an accuracy good enough for many practical purposes. Using sampling theory we could even provide confidence bounds on the accuracy of the answer [16].

Previously described approximation techniques make different trade-offs between efficiency and the generality of the

```
SELECT avg(sessionTime)
FROM Table
WHERE city='San Francisco'
WITHIN 2 SECONDS
```

Queries with Time Bounds

```
SELECT avg(sessionTime)
FROM Table
WHERE city='San Francisco'
ERROR 0.1 CONFIDENCE 95.0%
```

Queries with Error Bounds

https://www.cs.berkeley.edu/~sameerag/blinkdb_eurosys13.pdf