Stanford CS347 Guest Lecture

Spark

Reynold Xin @rxin

Guest lecture, May 18, 2015, Stanford

databricks™

Who am I?

Reynold Xin

Spark PMC member

Databricks cofounder & architect

UC Berkeley AMPLab PhD (on leave)

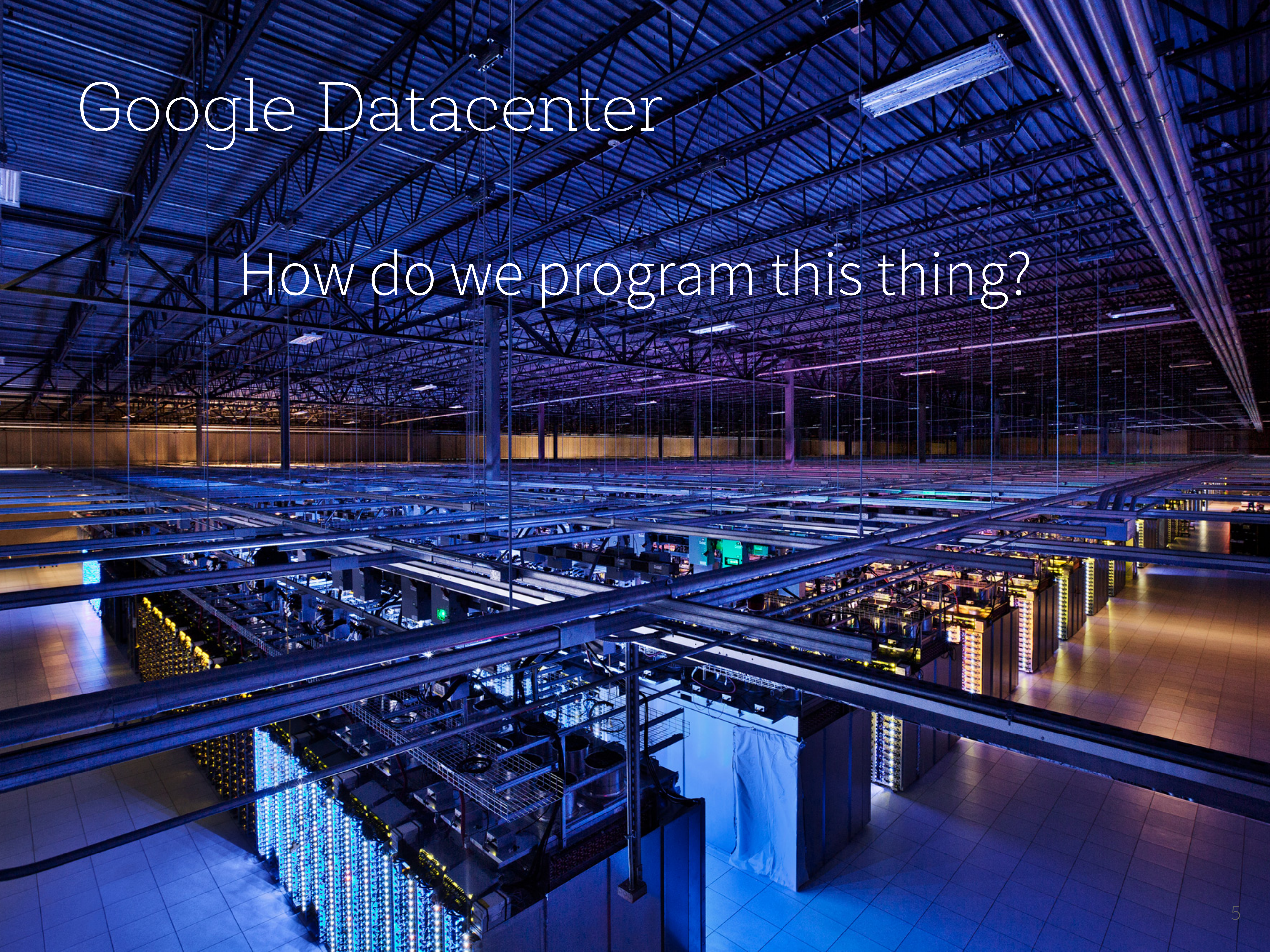
Agenda

1. MapReduce Review
2. Introduction to Spark and RDDs
3. Generality of RDDs (e.g. streaming, ML)
4. DataFrames
5. Internals (time permitting)

Agenda

1. MapReduce Review
2. Introduction to Spark and RDDs
3. Generality of RDDs (e.g. streaming, ML)
4. DataFrames
5. Internals (time permitting)

Google Datacenter



How do we program this thing?

Traditional Network Programming

Message-passing between nodes (MPI, RPC, etc)

Really hard to do at scale:

- How to split problem across nodes?
  - Important to consider network and data locality
- How to deal with failures?
  - If a typical server fails every 3 years, a 10,000-node cluster sees 10 faults/day!
- Even without failures: stragglers (a node is slow)

Almost nobody does this!

Data-Parallel Models

Restrict the programming interface so that the system can do more automatically

"Here's an operation, run it on all of the data"

- I don't care *where* it runs (you schedule that)
- In fact, feel free to run it *twice* on different nodes

MapReduce Programming Model

Data type: key-value *records*

Map function:

$$(K_{in}, V_{in}) \text{ -> } list(K_{inter}, V_{inter})$$

Reduce function:

$$(K_{inter}, list(V_{inter})) \text{ -> } list(K_{out}, V_{out})$$

databricks

MapReduce Programmability

Most real applications require multiple MR steps

- Google indexing pipeline: 21 steps
- Analytics queries (e.g. count clicks & top K): 2 – 5 steps
- Iterative algorithms (e.g. PageRank): 10's of steps

Multi-step jobs create spaghetti code

- 21 MR steps -> 21 mapper and reducer classes
- Lots of boilerplate code per step

MapReduce: A major step backwards

By David DeWitt on January 17, 2008 4:20 PM | Permalink | Comments (44) | TrackBacks (1)
[Note: Although the system attributes this post to a single author, it was written by David J. DeWitt and Michael Stonebraker]

On January 8, a Database Column reader asked for our views on new distributed database research efforts, and we'll begin here w
to discuss it, since the recent trade press has been filled with news of the revolution of so-called "cloud computing." This paradig
processors working in parallel to solve a computing problem. In effect, this suggests constructing a data center by lining up a larg
much smaller number of high-end servers.

For example, IBM and Google have announced plans to make a 1,000 processor cluster available to a few select universities to te
software tool called MapReduce [1]. Berkeley has gone so far as to plan on teaching their freshman how to program using the Ma

As both educators and researchers, we are amazed at the hype that the MapReduce proponents have spread about how it represen
data-intensive applications. MapReduce may be a good idea for writing certain types of general-purpose computations, but to the

1. A giant step backward in the programming paradigm for large-scale data intensive applications
2. A sub-optimal implementation, in that it uses brute force instead of indexing
3. Not novel at all -- it represents a specific implementation of well known techniques developed nearly 25 years ago
4. Missing most of the features that are routinely included in current DBMS

Problems with MapReduce

MapReduce use cases showed two major limitations:

1. difficulty of programming directly in MR.
2. Performance bottlenecks

In short, MR doesn't compose well for large applications

Therefore, people built high level frameworks and specialized systems.

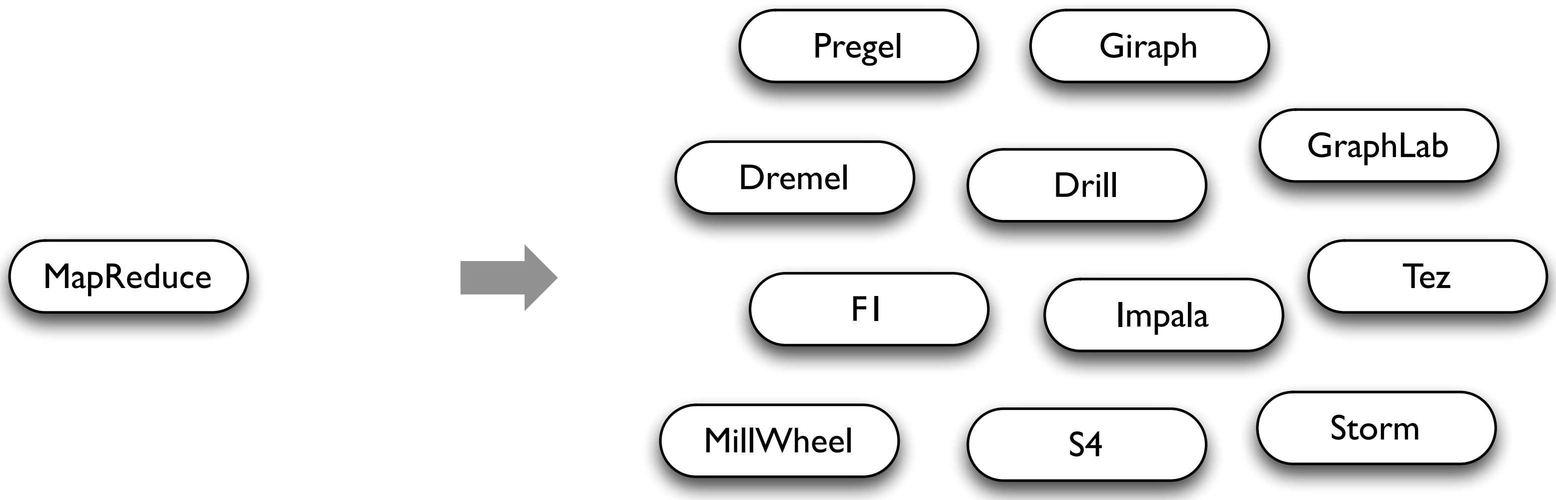
Higher Level Frameworks

```
SELECT count(*) FROM users
```

In reality, 90+% of MR jobs are generated by Hive SQL

```
A = load 'foo';
B = group A all;
C = foreach B generate COUNT(A);
```

Specialized Systems



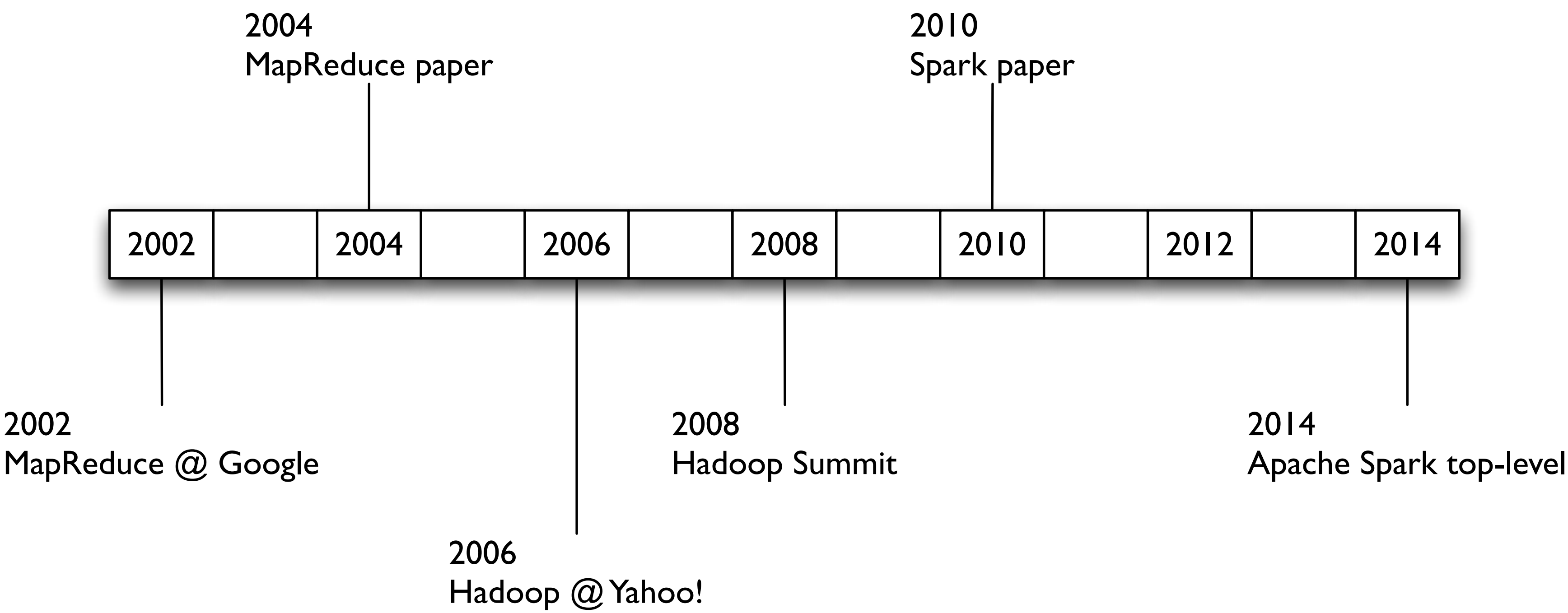
General Batch Processing

Specialized Systems:
iterative, interactive, streaming, graph, etc.

Agenda

1. MapReduce Review
2. Introduction to Spark and RDDs
3. Generality of RDDs (e.g. streaming, ML)
4. DataFrames
5. Internals (time permitting)

Spark: A Brief History



Spark Summary

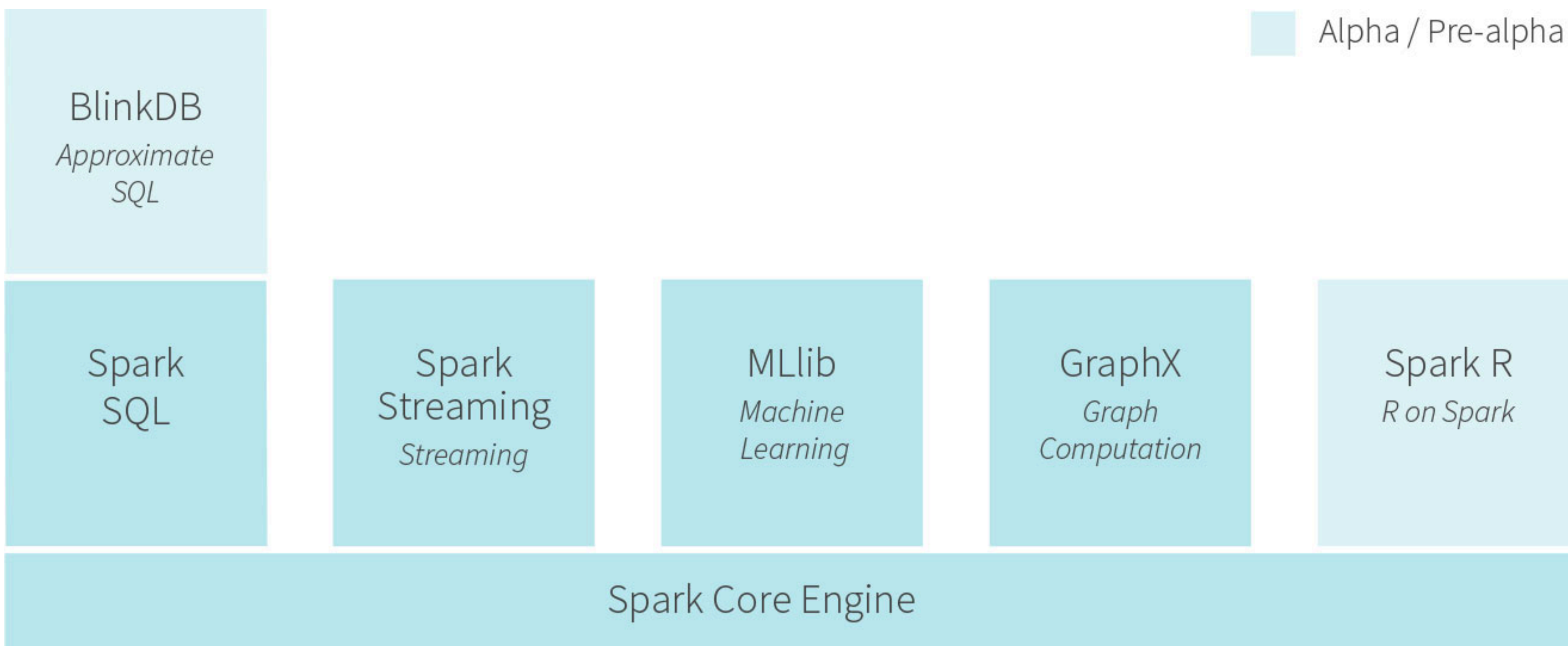
Unlike the various specialized systems, Spark's goal was to generalize MapReduce to support new apps

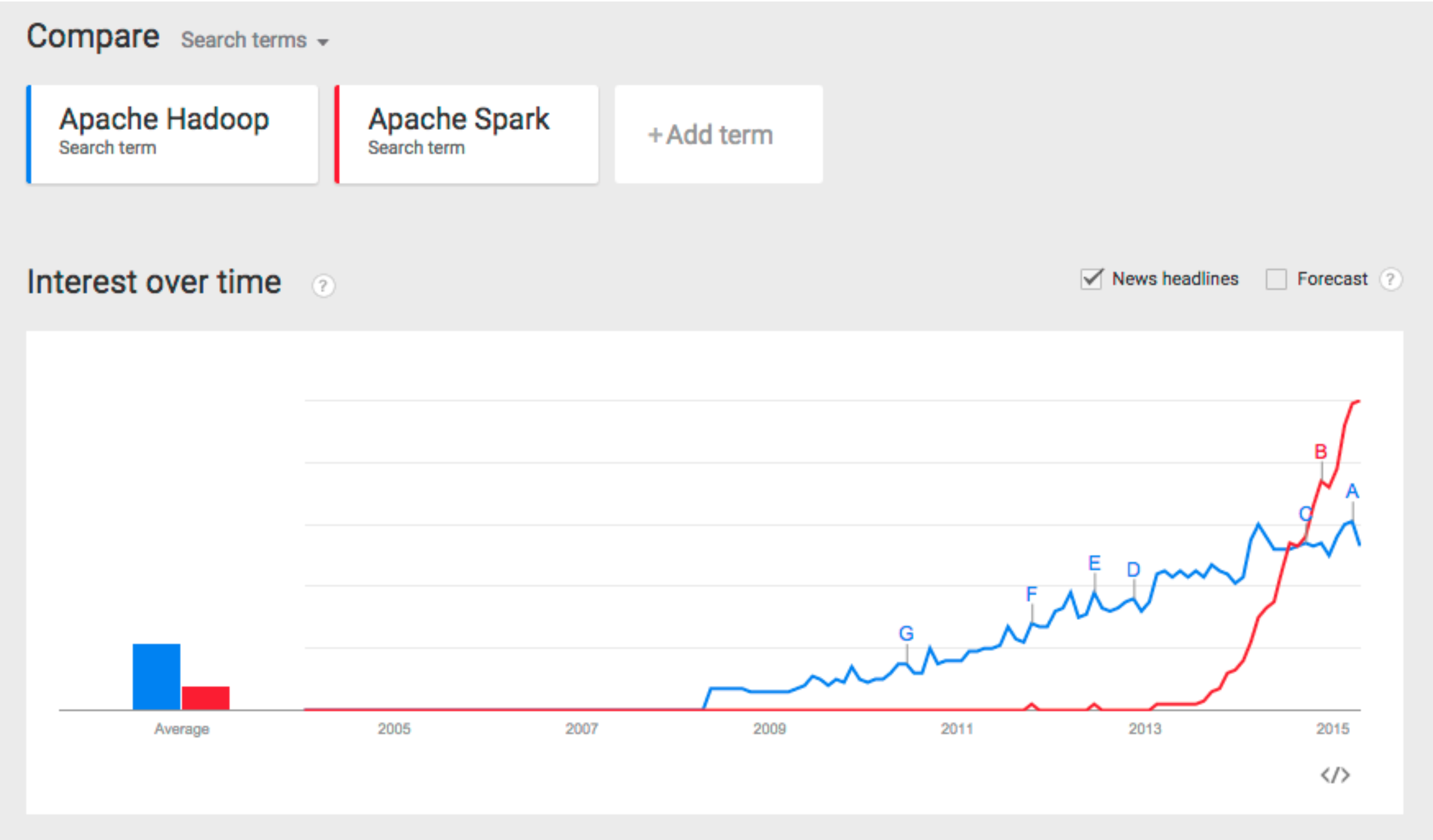
Two small additions are enough:

- fast data sharing
- general DAGs

More efficient engine, and simpler for the end users.

Spark Ecosystem





Note： not a scientific comparison.

Programmability

```java
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

```scala
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

Performance

Time to sort 100TB

2013 Record:
Hadoop

2100 machines

72 minutes

2014 Record:
Spark

207 machines

23 minutes

Also sorted 1PB in 4 hours

Source: Daytona GraySort benchmark, sortbenchmark.org

RDD: Core Abstraction

Write programs in terms of **distributed datasets**

and **operations** on them

Resilient Distributed Datasets

- Collections of objects spread across a cluster, stored in RAM or on Disk
- Built through parallel transformations
- Automatically rebuilt on failure

Operations

- Transformations (e.g. map, filter, groupBy)
- Actions (e.g. count, collect, save)

RDD

Resilient **D**istributed **D**atasets are the primary abstraction in Spark – a fault-tolerant collection of elements that can be operated on in parallel

Two types:

- *parallelized collections* – take an existing single-node collection and parallel it
- *Hadoop datasets: files on HDFS or other compatible storage*

Operations on RDDs

Transformations f(RDD) => RDD

- Lazy (not computed immediately)
- E.g. “map”

Actions:

- Triggers computation
- E.g. “count”, “saveAsTextFile”

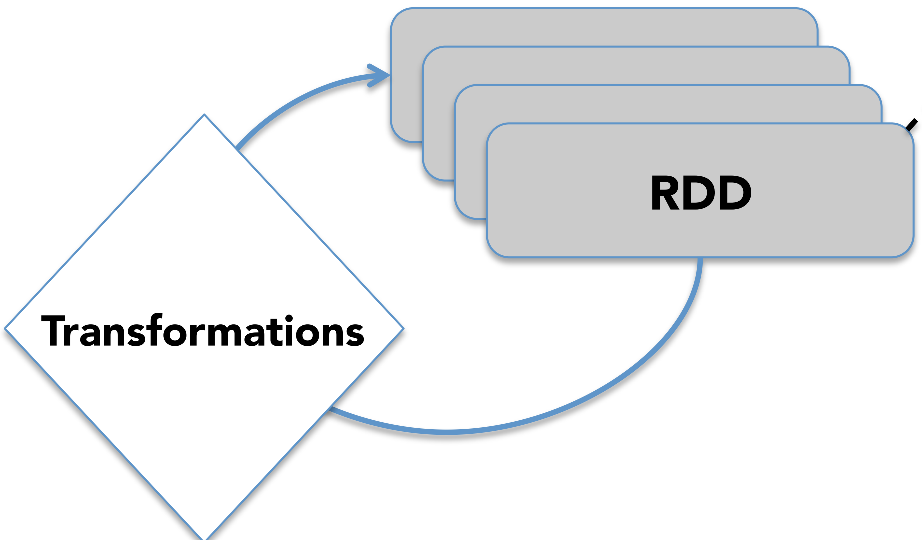
Working With RDDs

```
textFile = sc.textFile("SomeFile.txt")
```

RDD

Working With RDDs

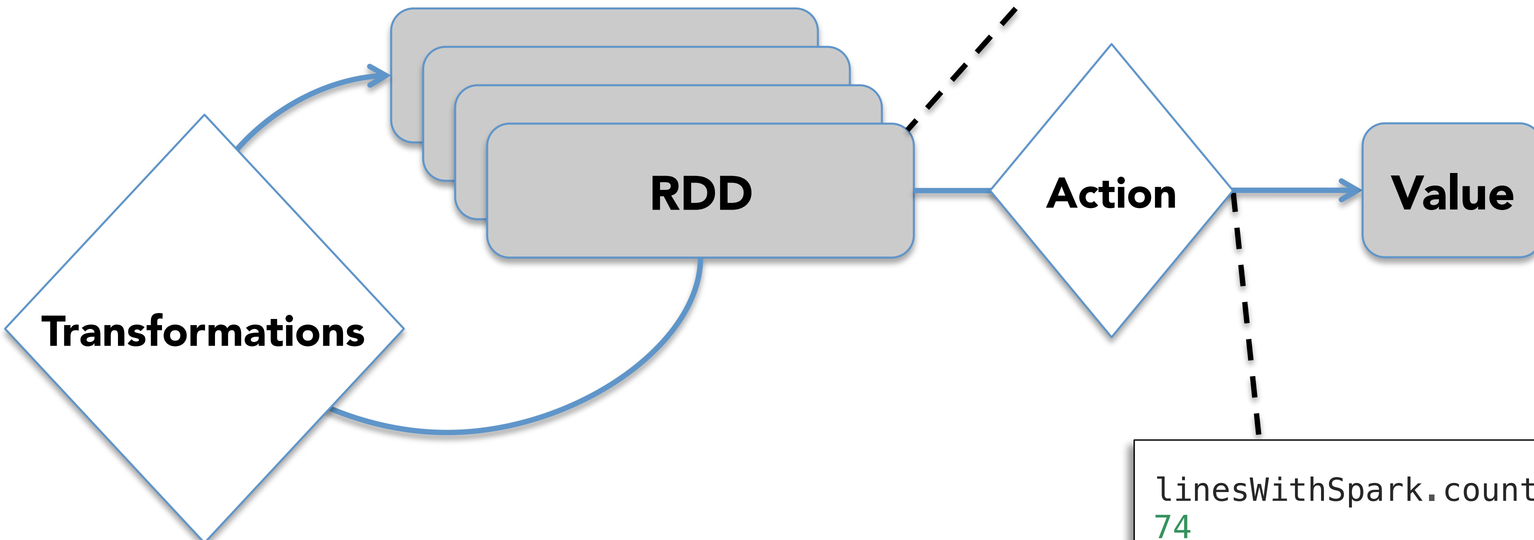
```
textFile = sc.textFile("SomeFile.txt")
```




```
linesWithSpark = textFile.filter(lambda line: "Spark" in line)
```

Working With RDDs

```
textFile = sc.textFile("SomeFile.txt")
```




```
linesWithSpark.count()
74

linesWithSpark.first()
# Apache Spark
```

```
linesWithSpark = textFile.filter(lambda line: "Spark" in line)
```

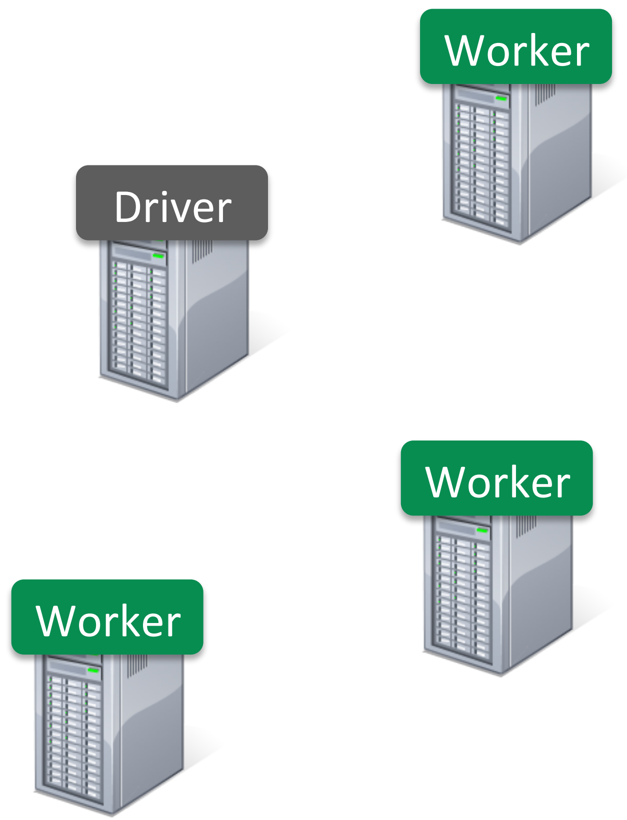
databricks

Example: Log Mining

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

Example: Log Mining

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

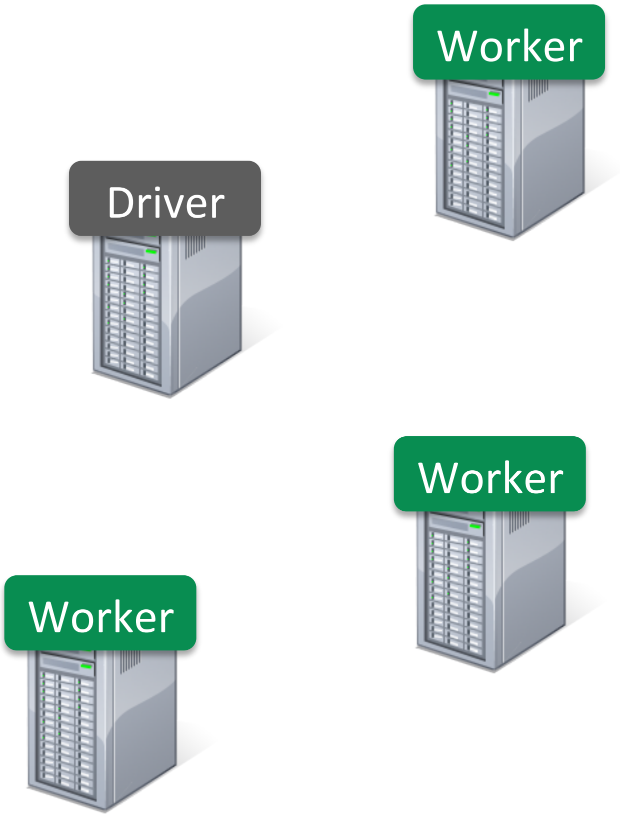


databricks

Example: Log Mining

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

```
lines = spark.textFile("hdfs://...")
```

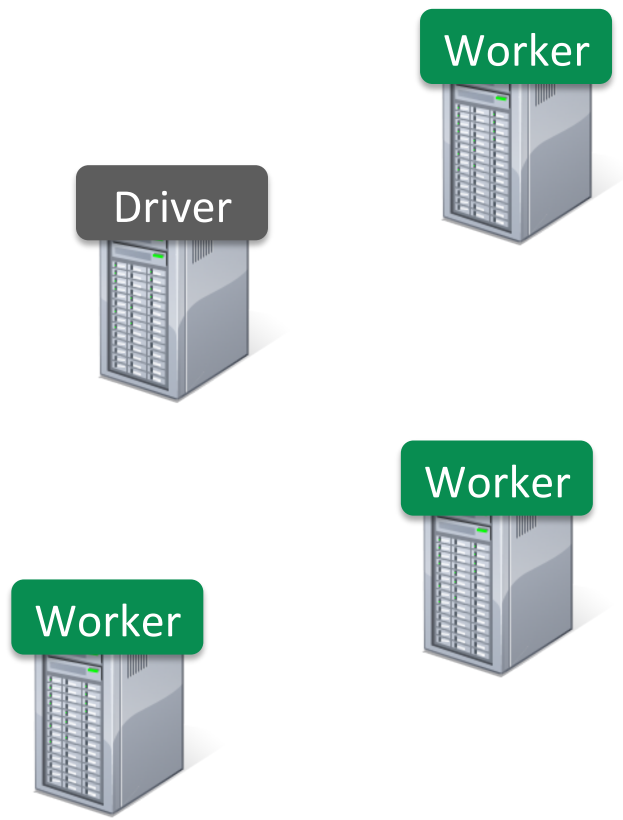


Example: Log Mining

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

Base RDD

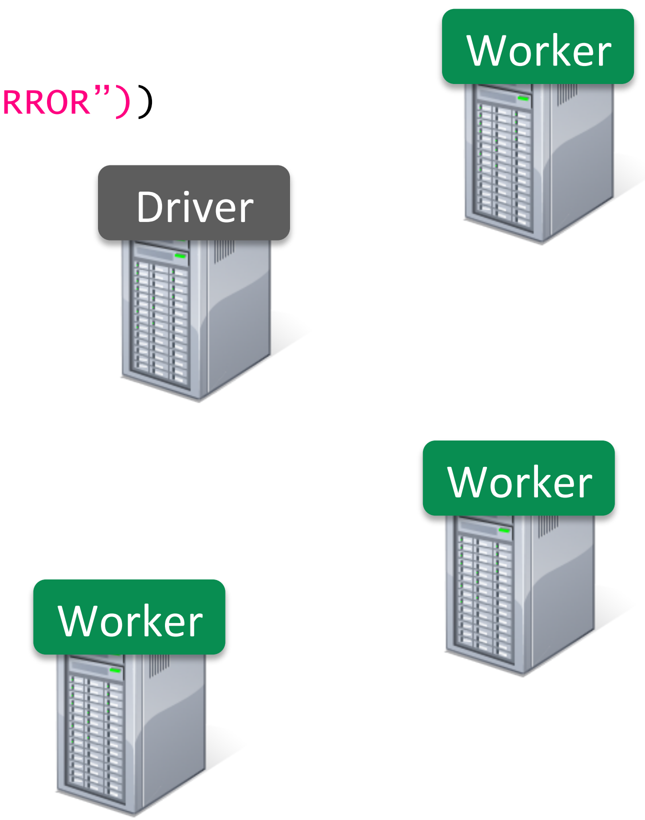
```
lines = spark.textFile("hdfs://...")
```



Example: Log Mining

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

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
```

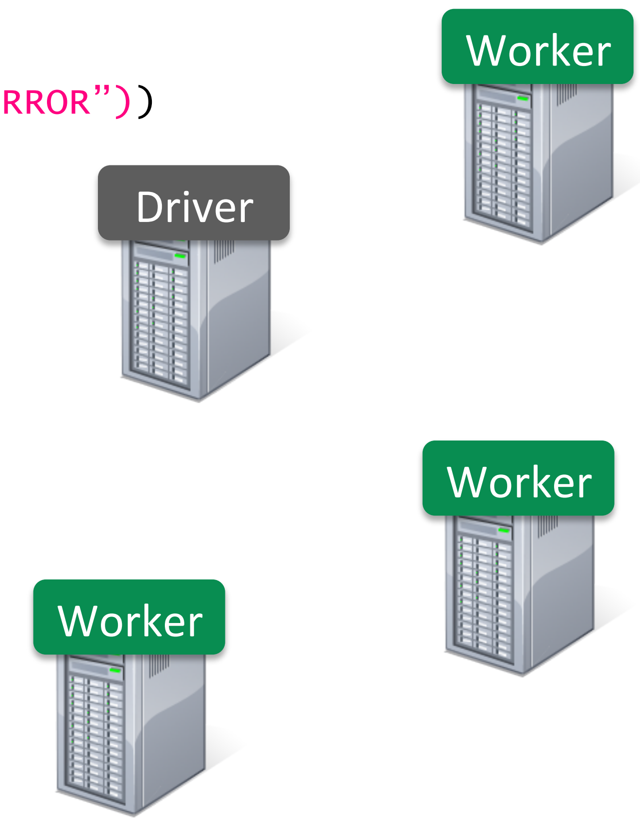


Example: Log Mining

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

Transformed RDD

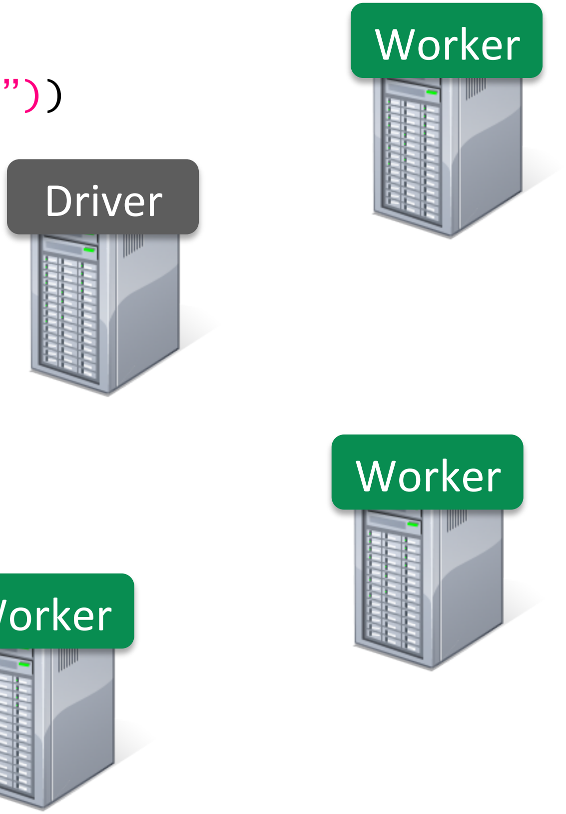
```
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
```



Example: Log Mining

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

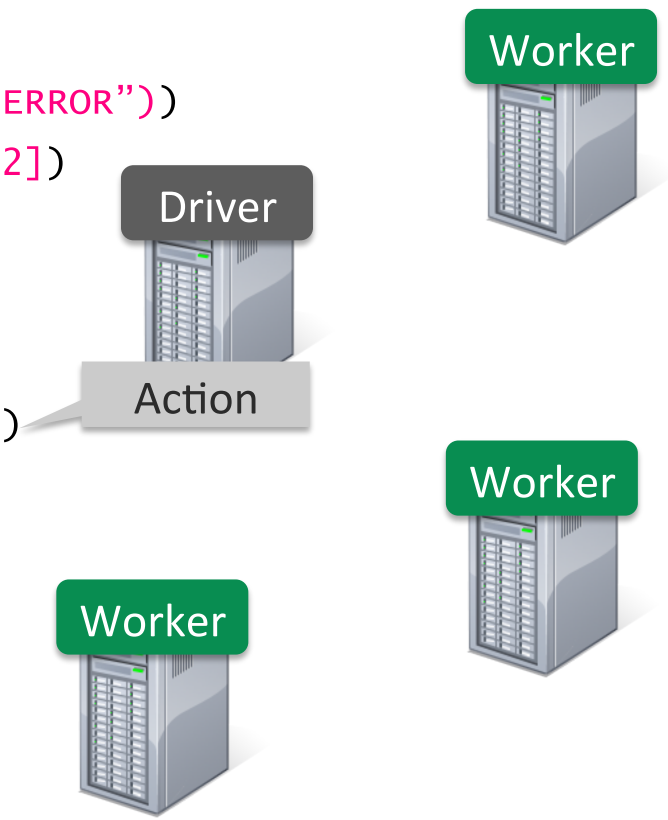
```
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()


messages.filter(lambda s: "mysql" in s).count()
```



Example: Log Mining

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

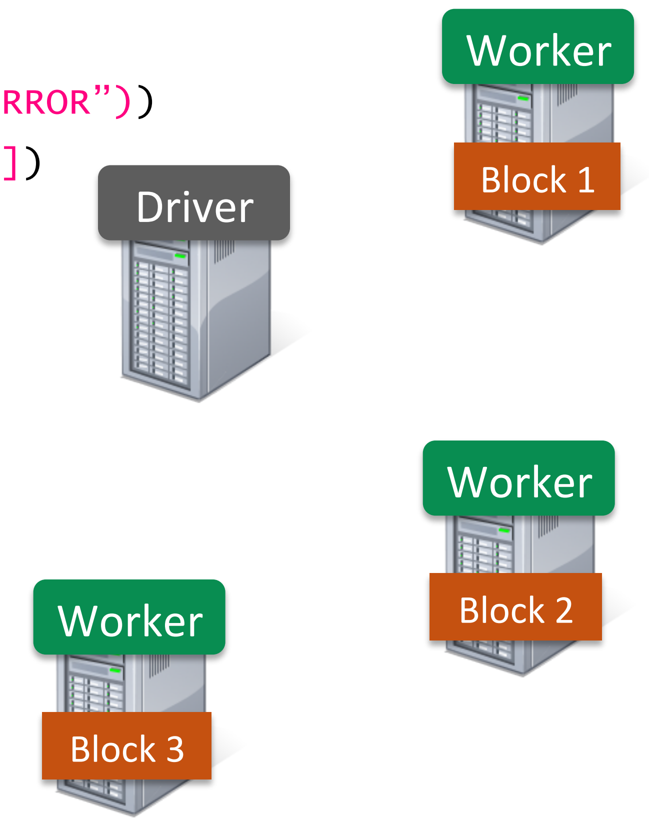
```
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()


messages.filter(lambda s: "mysql" in s).count()
```



Example: Log Mining

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

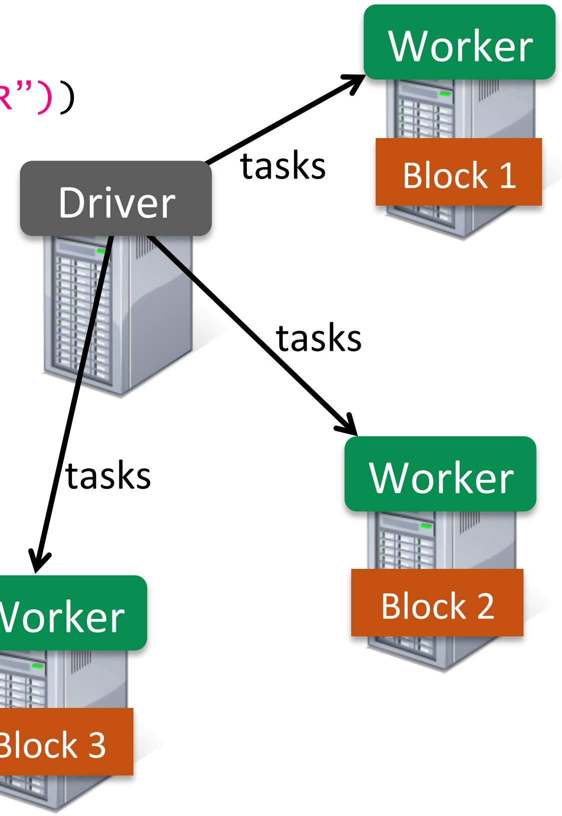
```
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()


messages.filter(lambda s: "mysql" in s).count()
```



Example: Log Mining

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

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()

messages.filter(lambda s: "mysql" in s).count()
```

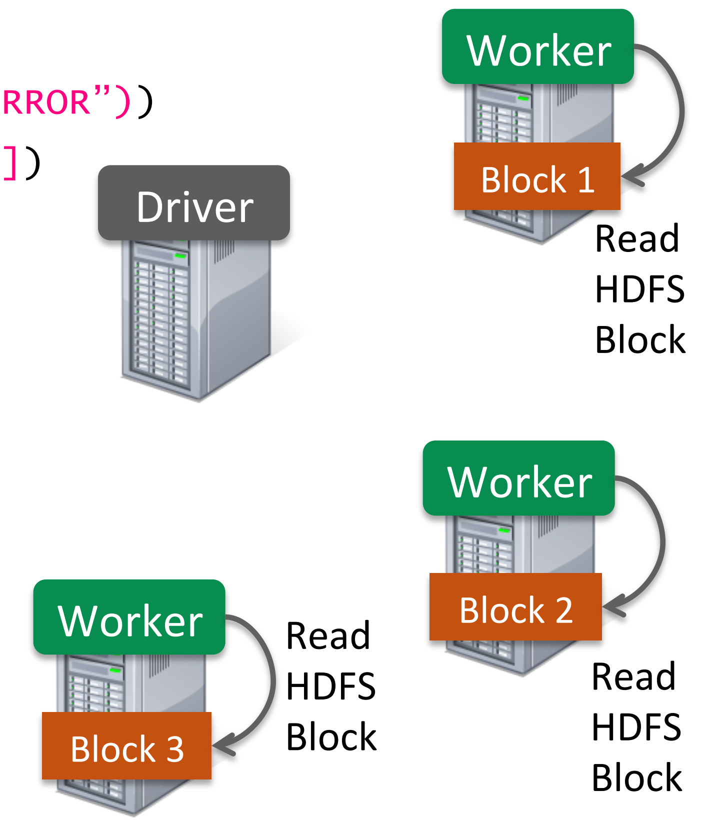


databricks

Example: Log Mining

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

```
lines = spark.textFile(“hdfs://...”)
errors = lines.filter(lambda s: s.startswith(“ERROR”))
messages = errors.map(lambda s: s.split(“\t”)[2])
messages.cache()


messages.filter(lambda s: “mysql” in s).count()
```

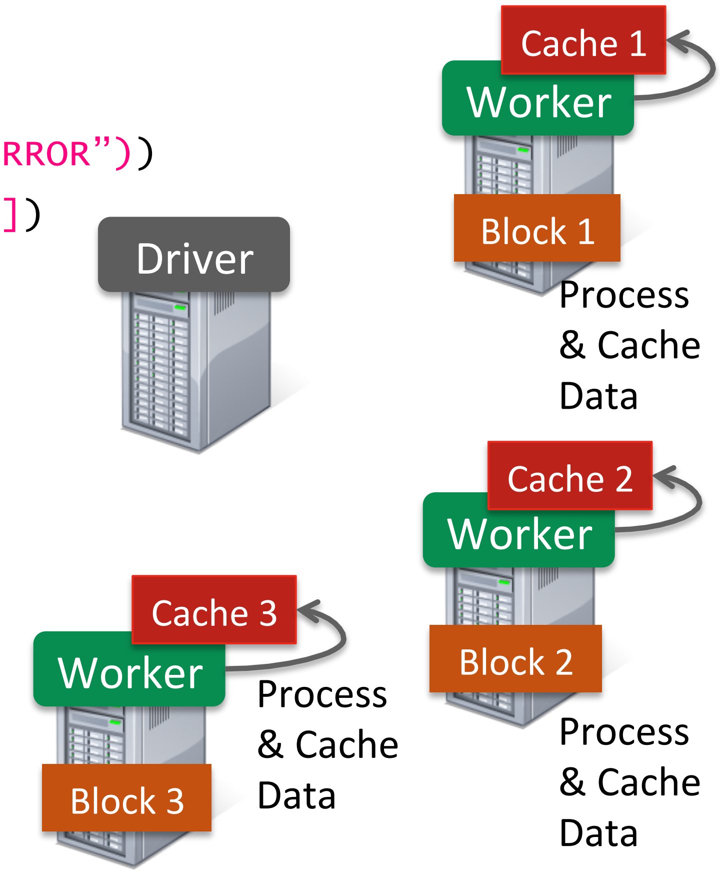


databricks

Example: Log Mining

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

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()


messages.filter(lambda s: "mysql" in s).count()
```

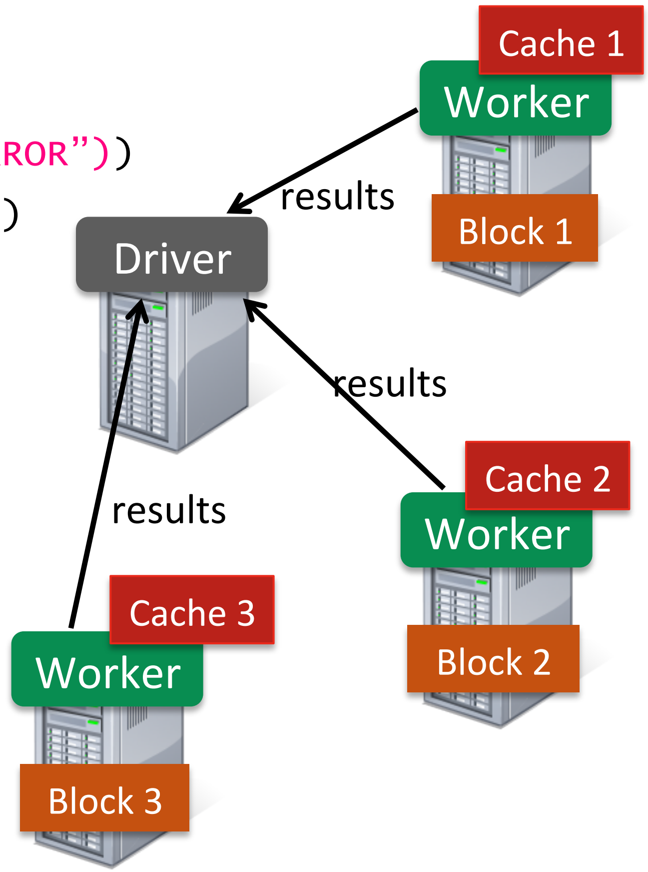


databricks

Example: Log Mining

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

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()


messages.filter(lambda s: "mysql" in s).count()
```

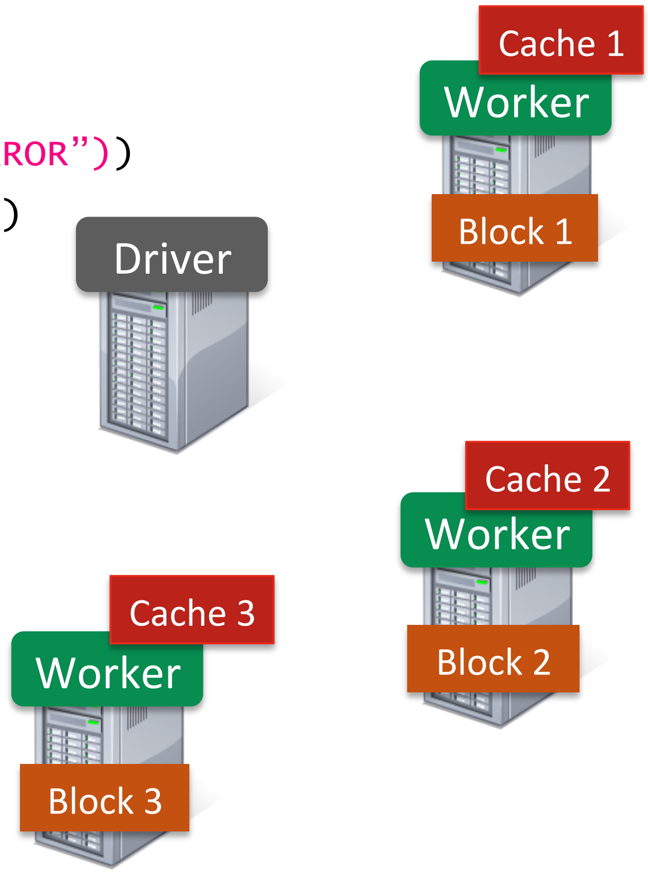


databricks

Example: Log Mining

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

```
lines = spark.textFile(“hdfs://...”)
errors = lines.filter(lambda s: s.startswith(“ERROR”))
messages = errors.map(lambda s: s.split(“\t”)[2])
messages.cache()


messages.filter(lambda s: “mysql” in s).count()
messages.filter(lambda s: “php” in s).count()
```

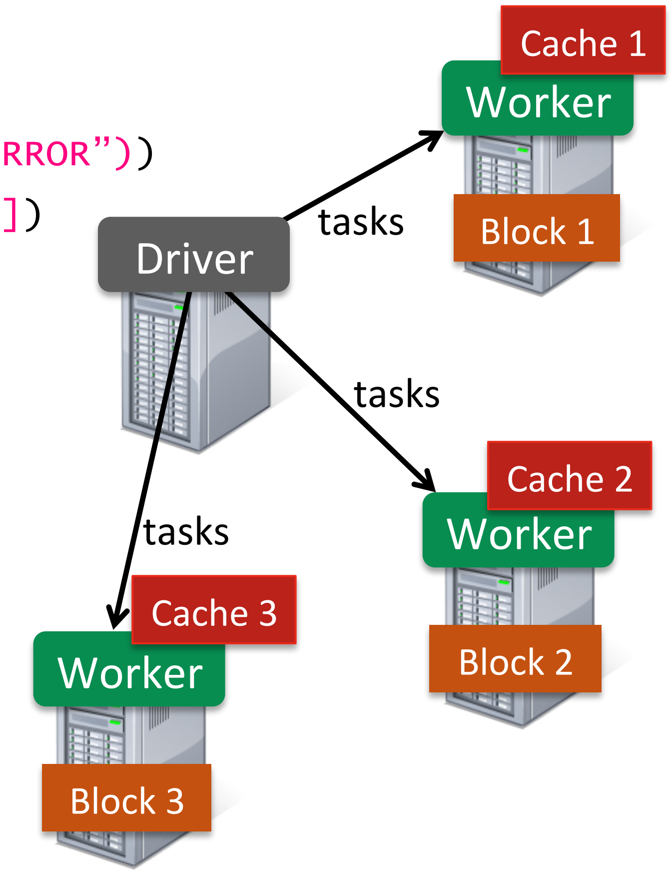


databricks

Example: Log Mining

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

```
lines = spark.textFile(“hdfs://...”)
errors = lines.filter(lambda s: s.startswith(“ERROR”))
messages = errors.map(lambda s: s.split(“\t”)[2])
messages.cache()


messages.filter(lambda s: “mysql” in s).count()
messages.filter(lambda s: “php” in s).count()
```

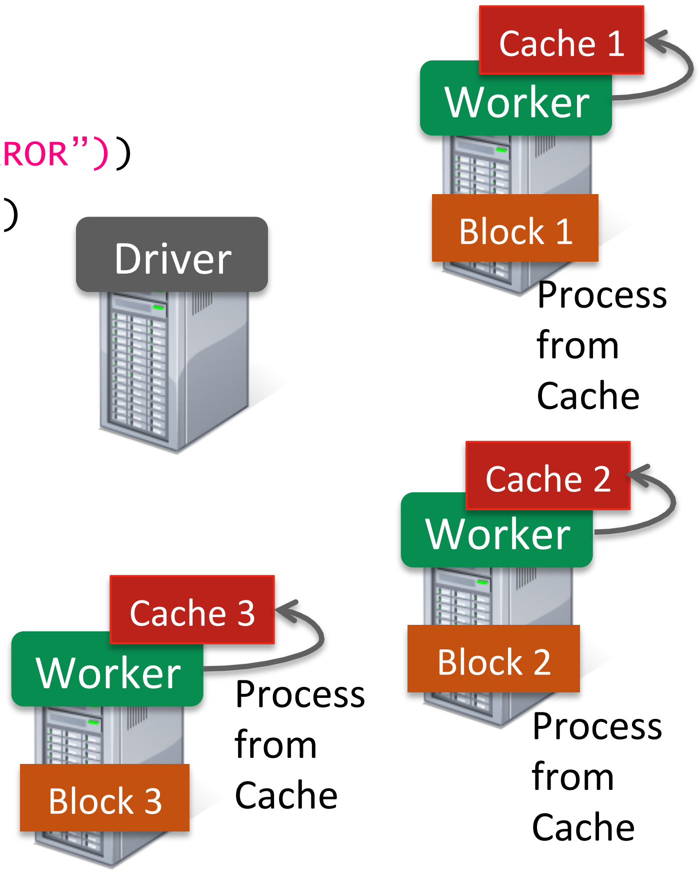


databricks

Example: Log Mining

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

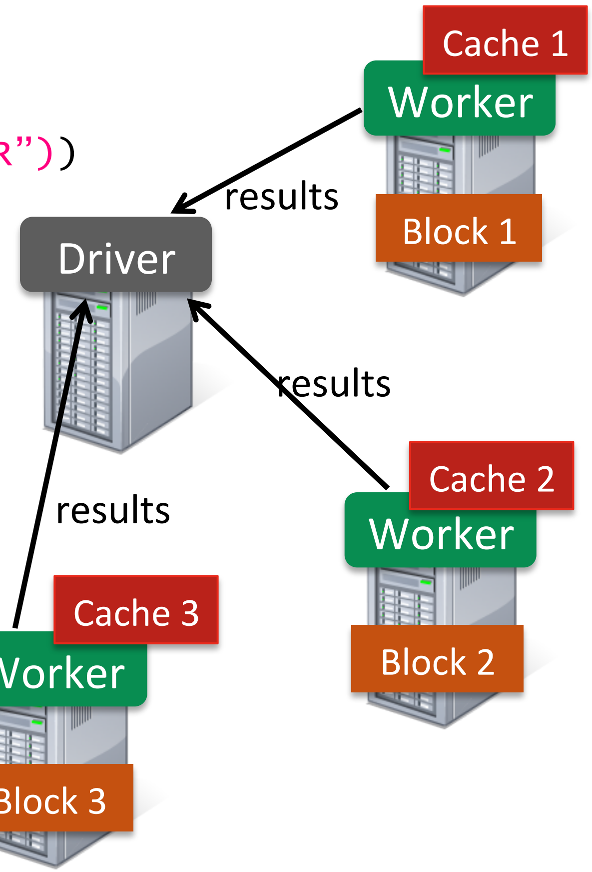
```
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()


messages.filter(lambda s: "mysql" in s).count()
messages.filter(lambda s: "php" in s).count()
```



Example: Log Mining

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

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()


messages.filter(lambda s: "mysql" in s).count()
messages.filter(lambda s: "php" in s).count()
```

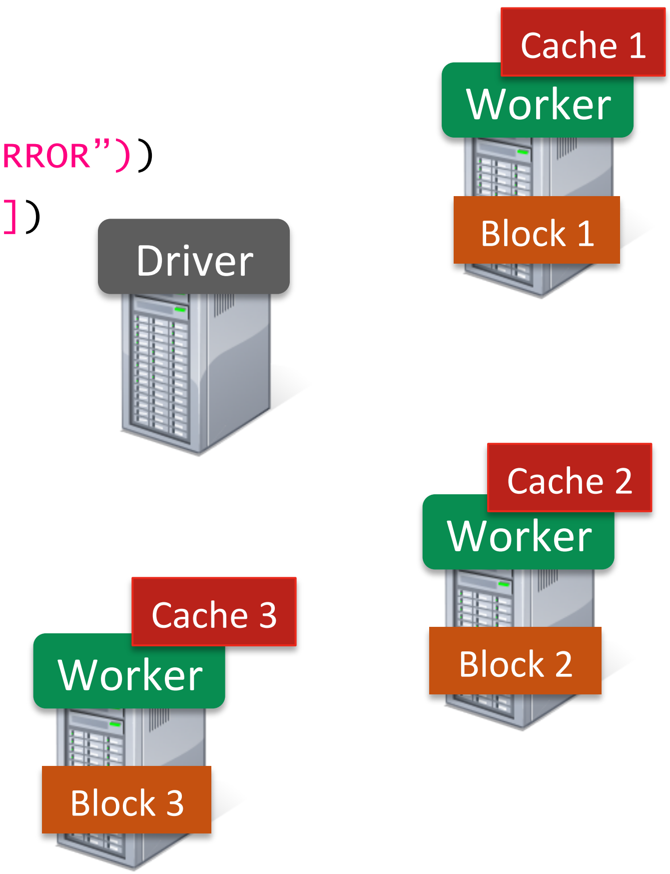


databricks

Example: Log Mining

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

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(lambda s: s.startswith("ERROR"))
messages = errors.map(lambda s: s.split("\t")[2])
messages.cache()


messages.filter(lambda s: "mysql" in s).count()
messages.filter(lambda s: "php" in s).count()
```



Cache your data ➔ Faster Results

Full-text search of Wikipedia

- 60GB on 20 EC2 machines
- 0.5 sec from mem vs. 20s for on-disk

Language Support

Python

```
lines = sc.textFile(...)
lines.filter(lambda s: "ERROR" in s).count()
```

Scala

```
val lines = sc.textFile(...)
lines.filter(x => x.contains("ERROR")).count()
```

Java

```
JavaRDD<String> lines = sc.textFile(...);
lines.filter(new Function<String, Boolean>() {
  Boolean call(String s) {
    return s.contains("error");
  }
}).count();
```

Standalone Programs

Python, Scala, & Java

Interactive Shells

Python & Scala

Performance

Java & Scala are faster due to static typing

…but Python is often fine

Expressive API

`map` `reduce`

Expressive API

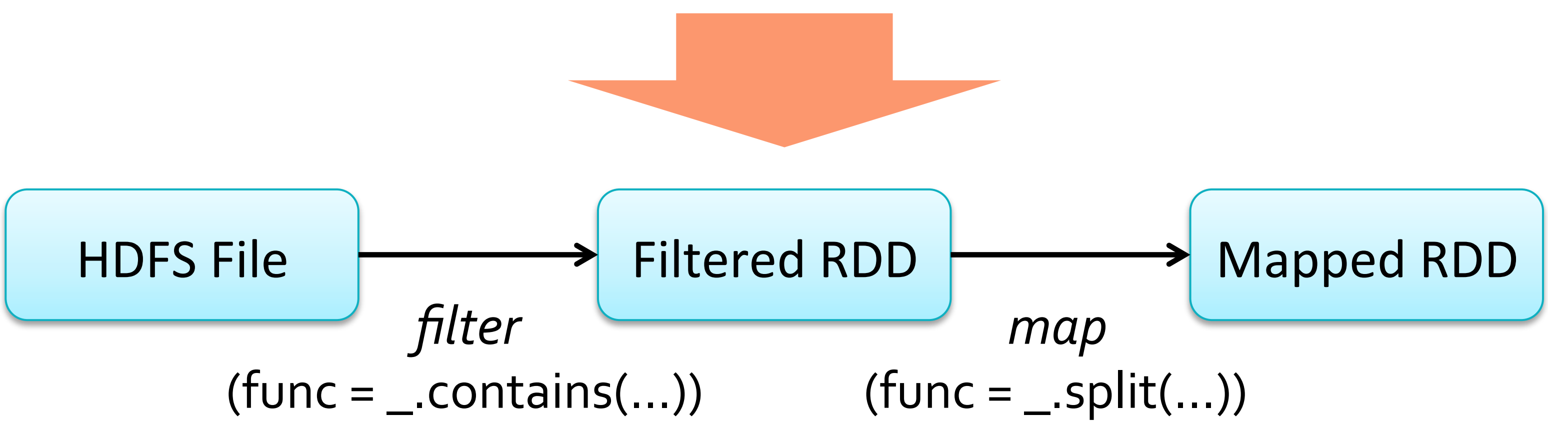
map	reduce	sample
filter	count	take
groupBy	fold	first
sort	reduceByKey	partitionBy
union	groupByKey	mapWith
join	cogroup	pipe
leftOuterJoin	cross	save ...
rightOuterJoin	zip	

Fault Recovery

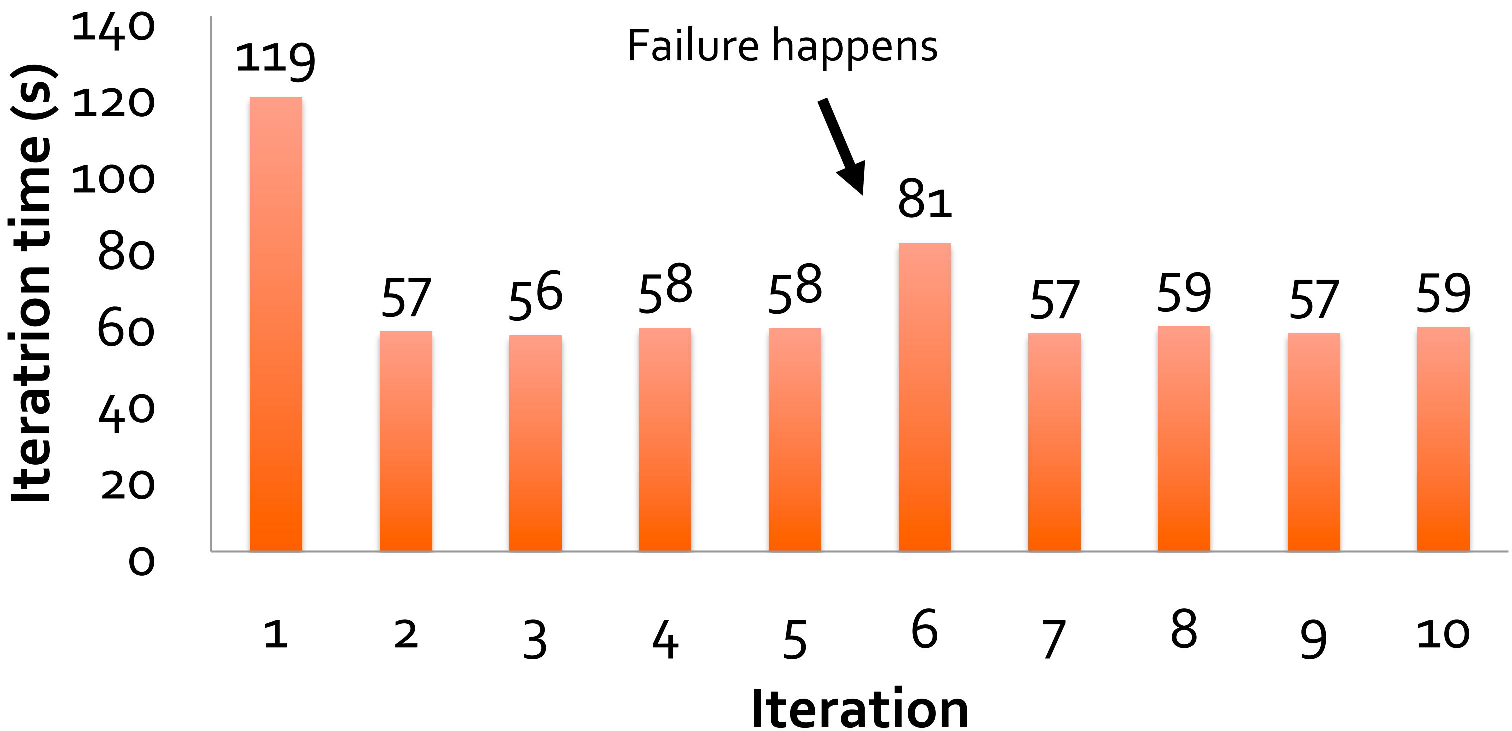
RDDs track *lineage* information that can be used to efficiently reconstruct lost partitions

Ex:

```
messages = textFile(...).filter(_.startsWith("ERROR"))
                        .map(_.split('\t')(2))
```

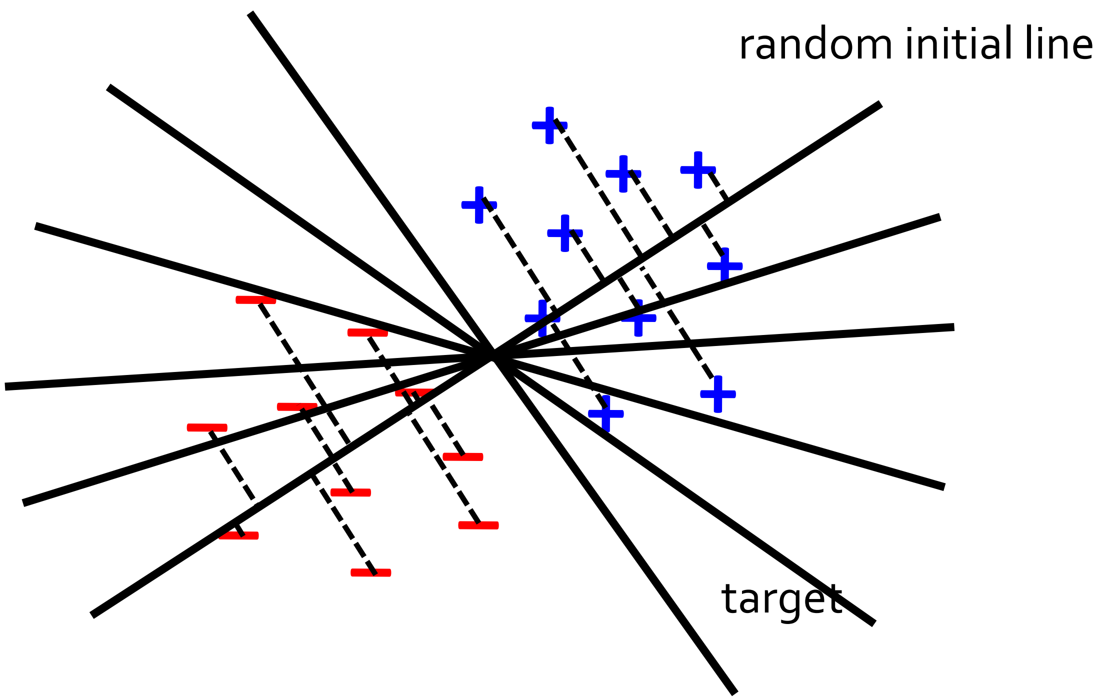


Fault Recovery Results



Example: Logistic Regression

Goal: find best line separating two sets of points



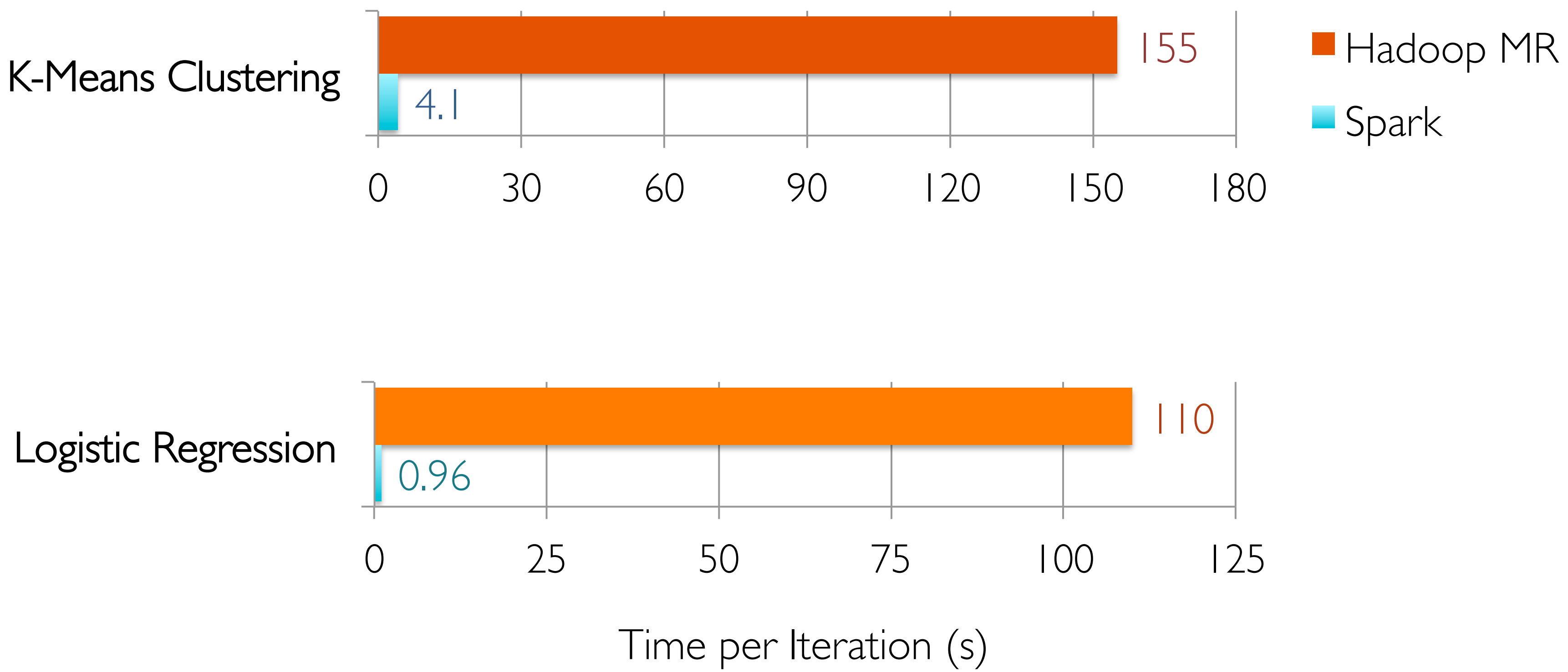
databricks

Example: Logistic Regression

```
val data = spark.textFile(...).map(readPoint).cache()

var w = Vector.random(D)

for (i <- 1 to ITERATIONS) {
  val gradient = data.map(p =>
    (1 / (1 + exp(-p.y*(w dot p.x))) - 1) * p.y * p.x
  ).reduce(_ + _)
  w -= gradient
}

println("Final w: " + w)
```

w automatically shipped to cluster

LR/K-Means Performance

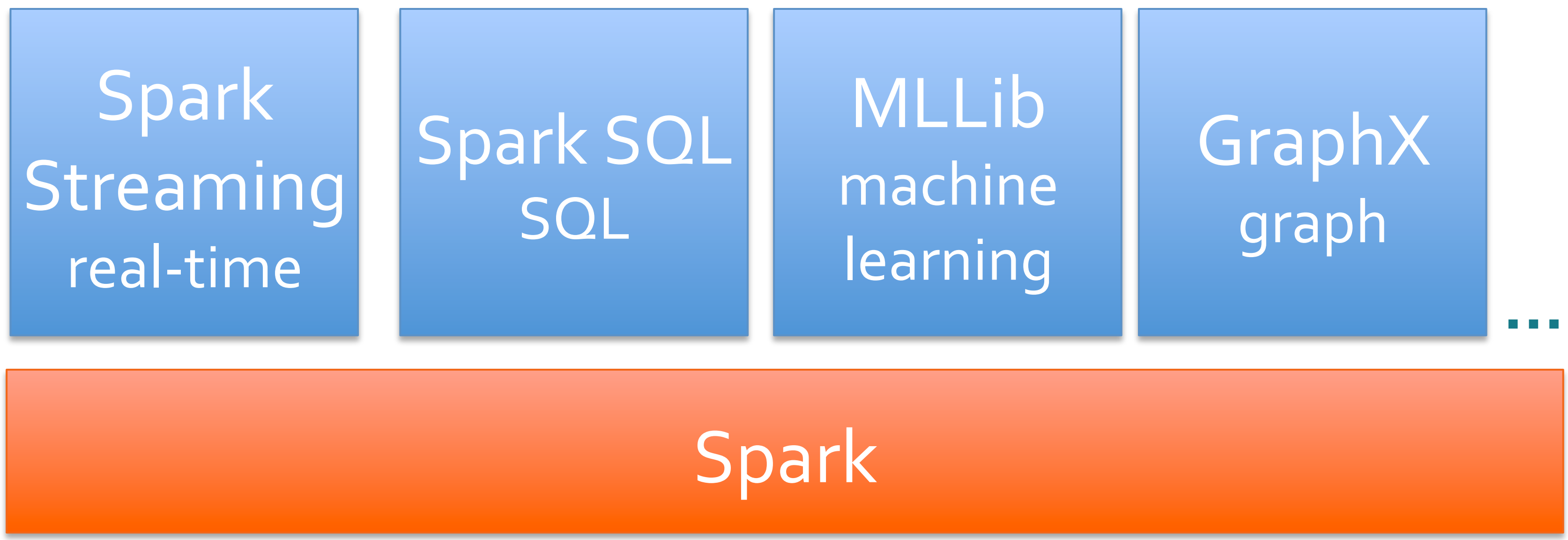


10B points

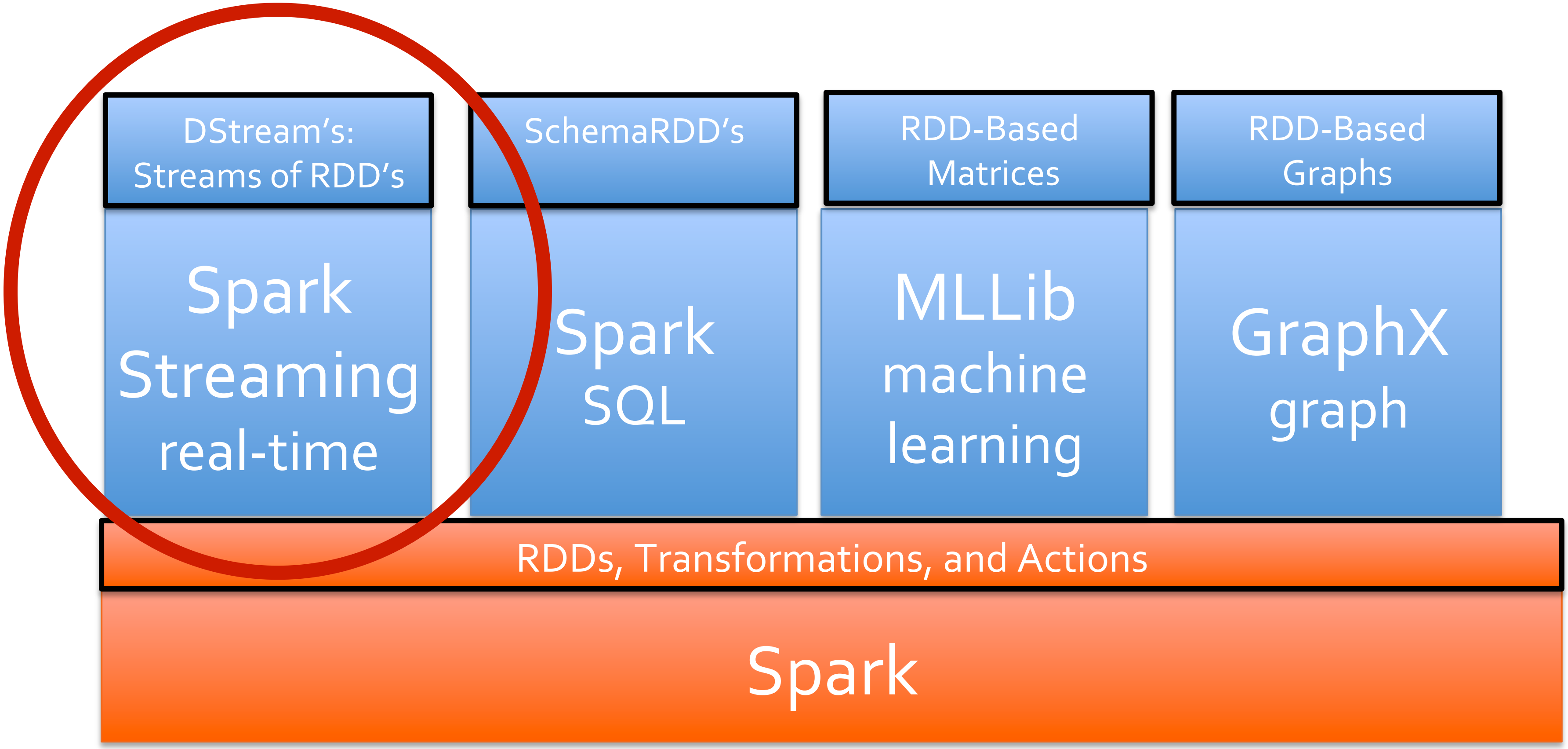
Agenda

1. MapReduce Review
2. Introduction to Spark and RDDs
3. Generality of RDDs (e.g. streaming, ML)
4. DataFrames
5. Internals (time permitting)

Generality of RDDs



Generality of RDDs



databricks

Spark Streaming: Motivation

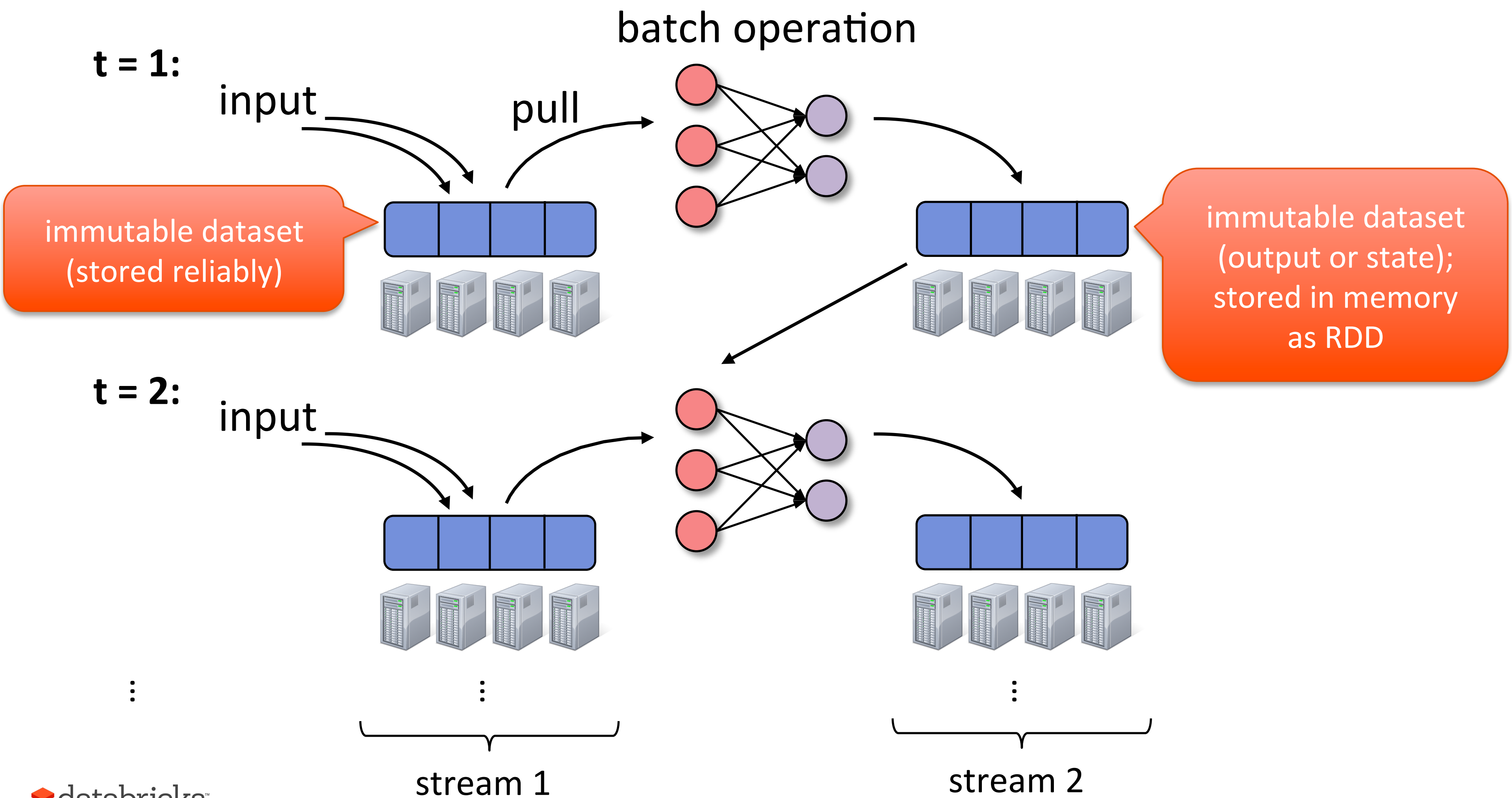
Many important apps must process large data streams at second-scale latencies

- Site statistics, intrusion detection, online ML

To build and scale these apps users want:

- **Integration:** with offline analytical stack
- **Fault-tolerance:** both for crashes and stragglers
- **Efficiency:** low cost beyond base processing

Discretized Stream Processing

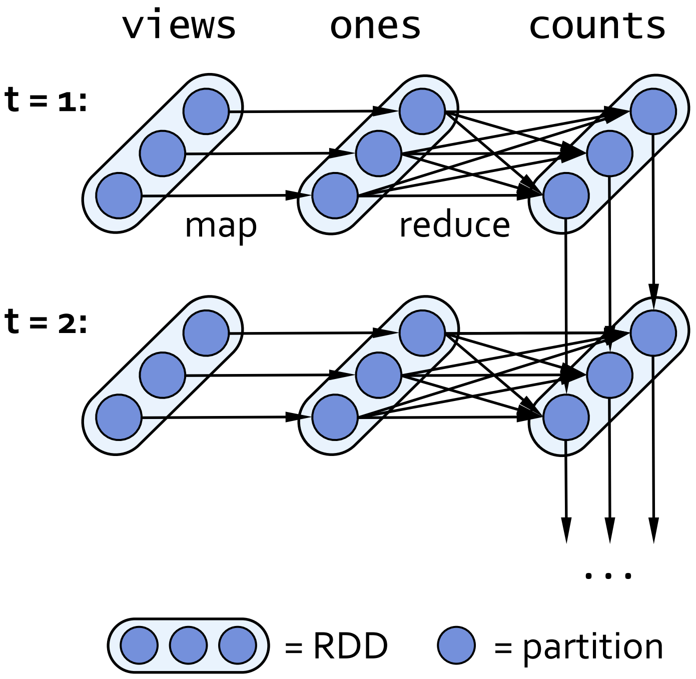


databricks

Programming Interface

Simple functional API

```
views = readStream("http:...", "1s")
ones = views.map(ev => (ev.url, 1))
counts = ones.runningReduce(_ + _)
```



Interoperates with RDDs

```
// Join stream with static RDD
counts.join(historicCounts).map(...)

// Ad-hoc queries on stream state
counts.slice("21:00","21:05").topK(10)
```

Inherited “for free” from Spark

RDD data model and API

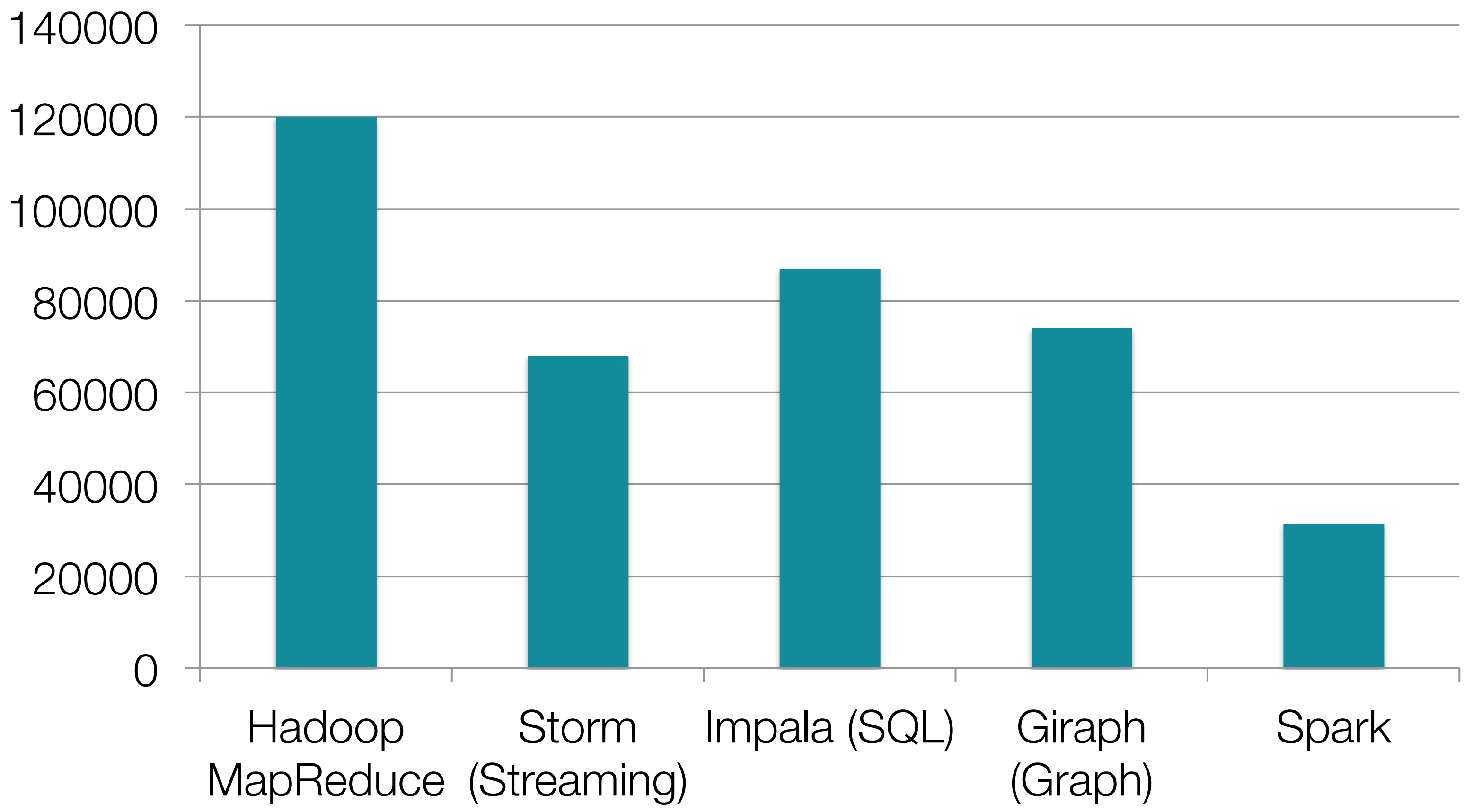
Data partitioning and shuffles

Task scheduling

Monitoring/instrumentation

Scheduling and resource allocation

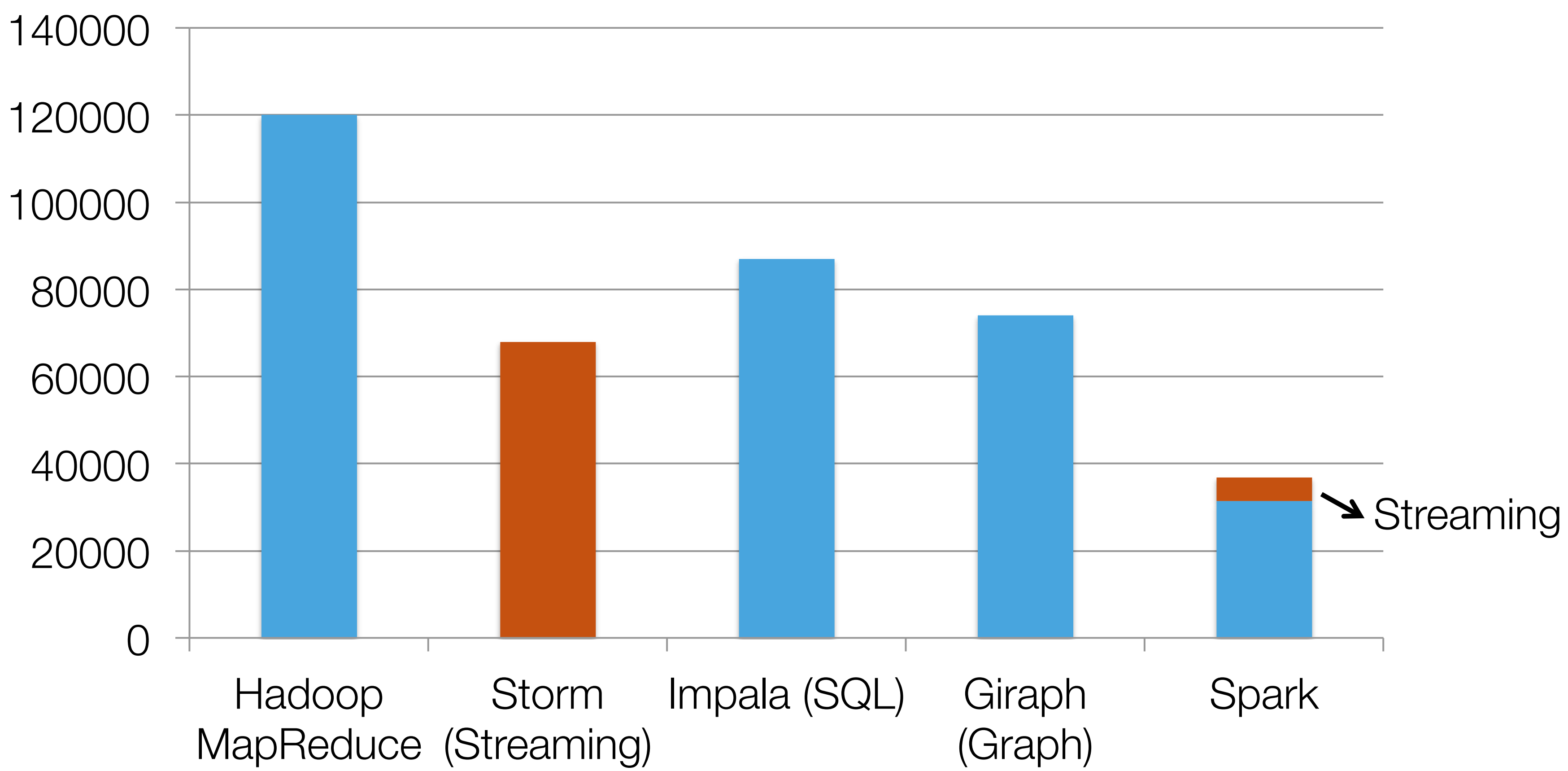
Powerful Stack – Agile Development



non-test, non-example source lines

databricks

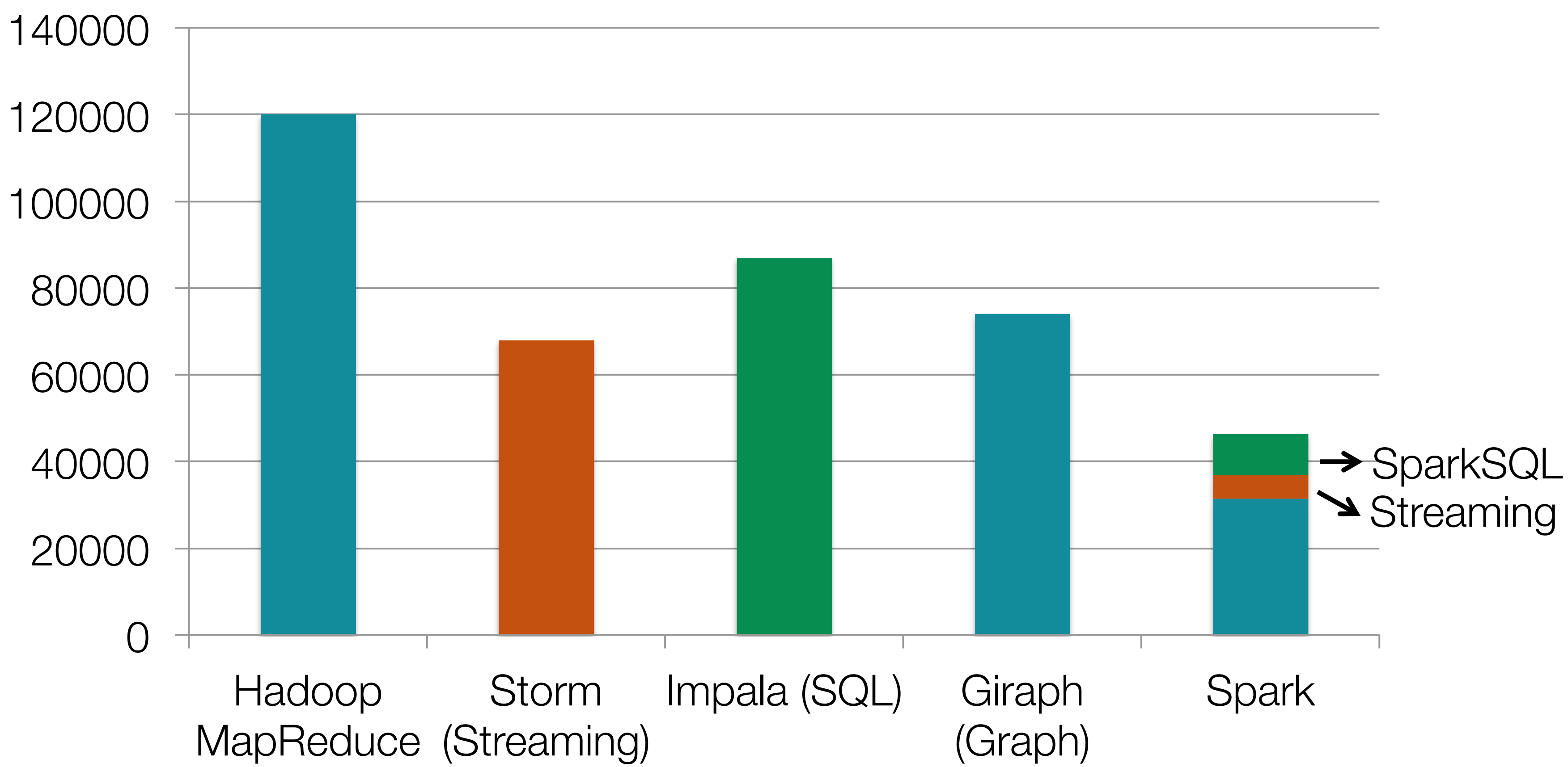
Powerful Stack – Agile Development



non-test, non-example source lines

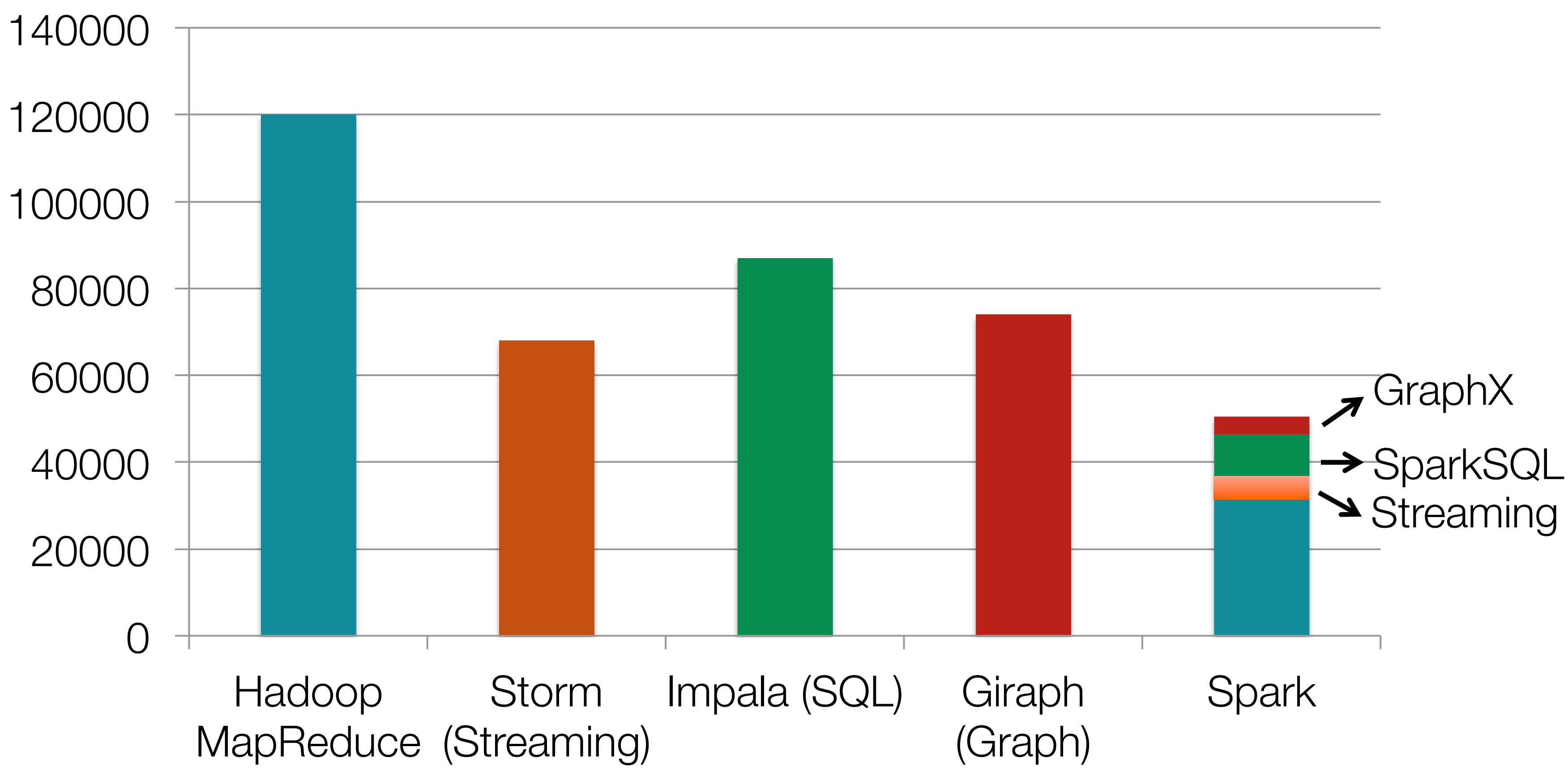
databricks

Powerful Stack – Agile Development



non-test, non-example source lines

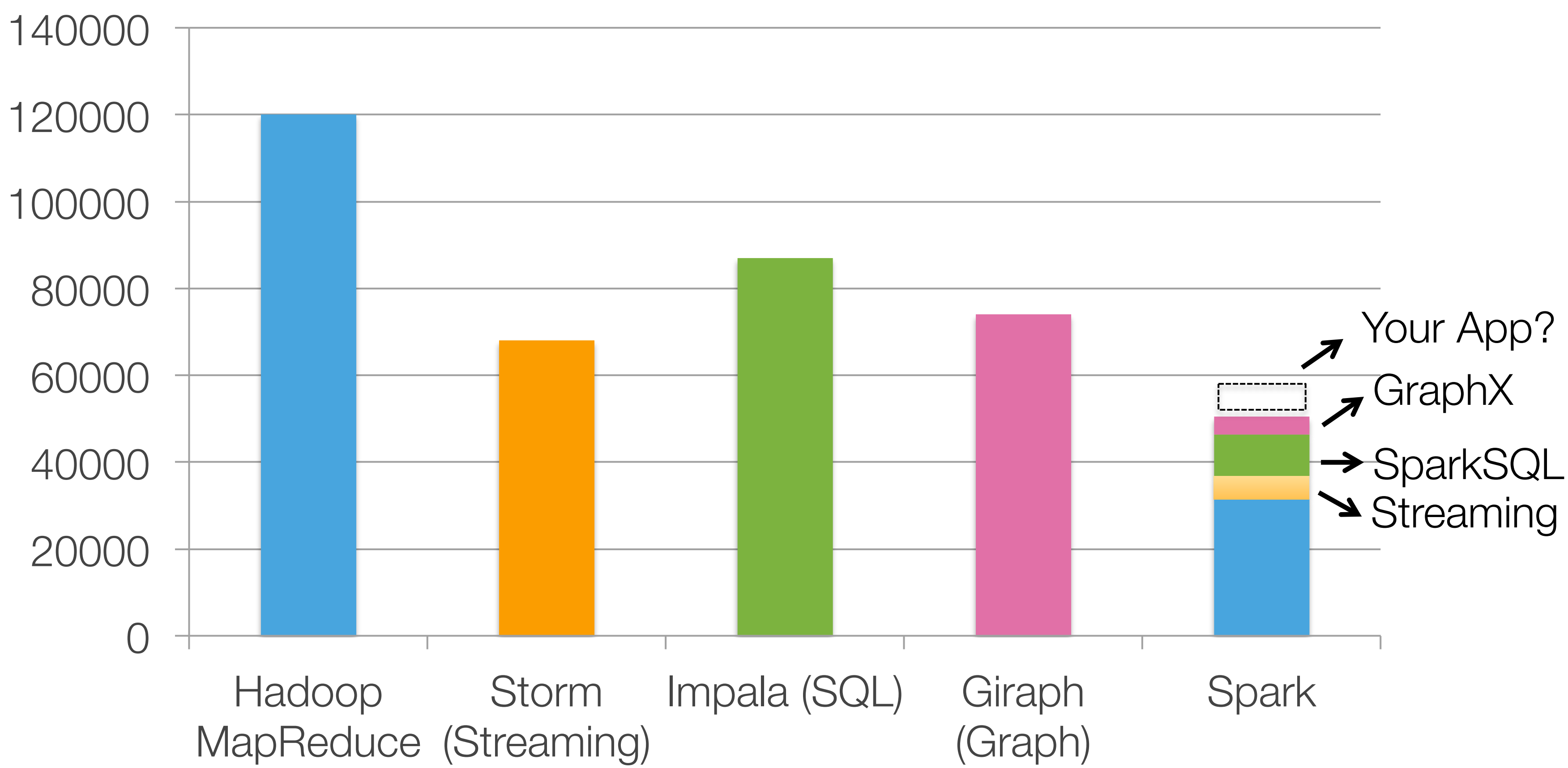
Powerful Stack – Agile Development



non-test, non-example source lines

databricks

Powerful Stack – Agile Development



non-test, non-example source lines

databricks

Benefits for Users

High performance data sharing

- Data sharing is the bottleneck in many environments
- RDD's provide in-place sharing through memory

Applications can compose models

- Run a SQL query and then PageRank the results
- ETL your data and then run graph/ML on it

Benefit from investment in shared functionality

- E.g. re-usable components (shell) and performance optimizations

Agenda

1. MapReduce Review
2. Introduction to Spark and RDDs
3. Generality of RDDs (e.g. streaming, ML)
4. DataFrames
5. Internals (time permitting)

From MapReduce to Spark

```java
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);
  }
}
```

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

Beyond Hadoop Users

Spark early adopters

Users

Understands
MapReduce
& functional APIs

Data Engineers
Data Scientists
Statisticians
R users
PyData …

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

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

DataFrames in Spark

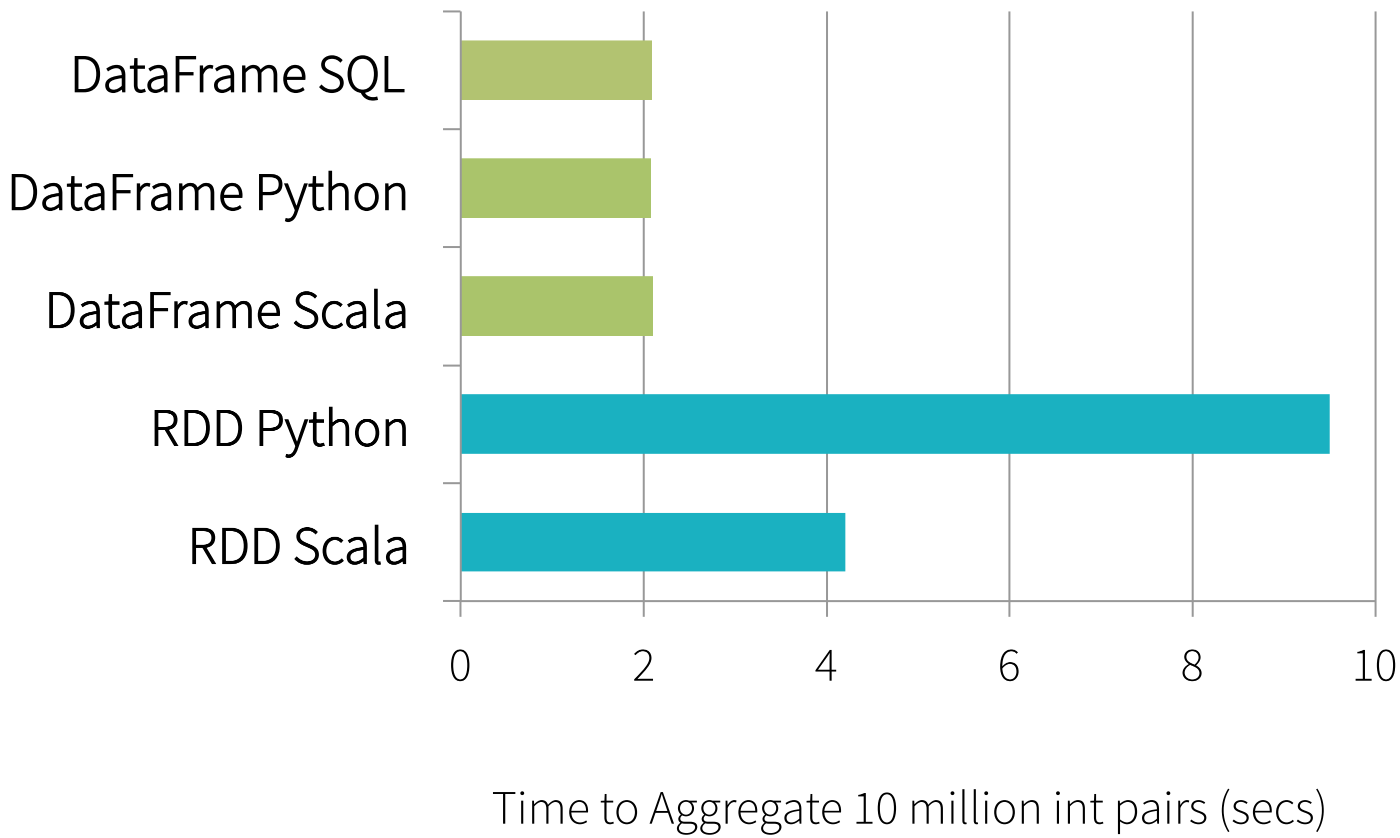
Distributed collection of data grouped into named columns (i.e. RDD with schema)

DSL designed for common tasks

- Metadata
- Sampling
- Project, filter, aggregation, join, …
- UDFs

Available in Python, Scala, Java, and R (via SparkR)

Not Just Less Code: Faster Implementations



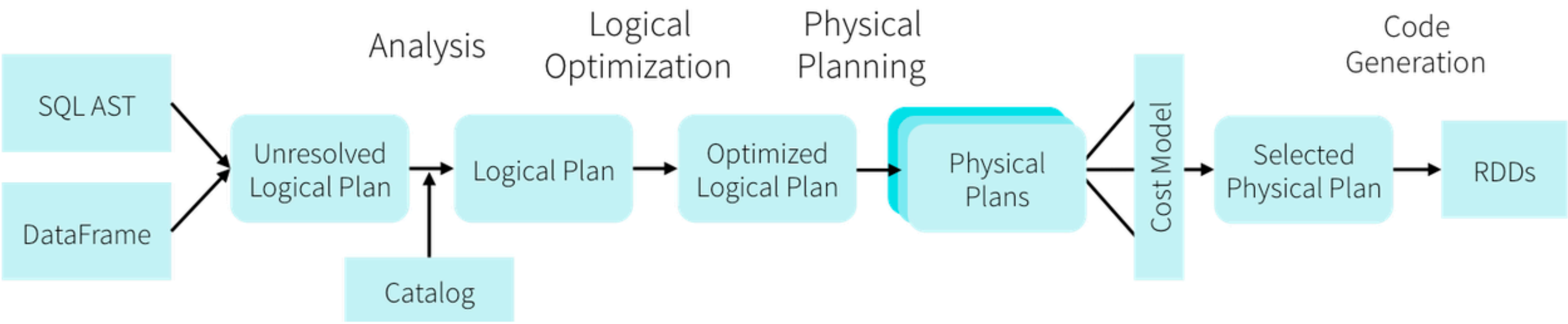
databricks

DataFrame Internals

Represented internally as a “logical plan”

Execution is lazy, allowing it to be optimized by a query optimizer

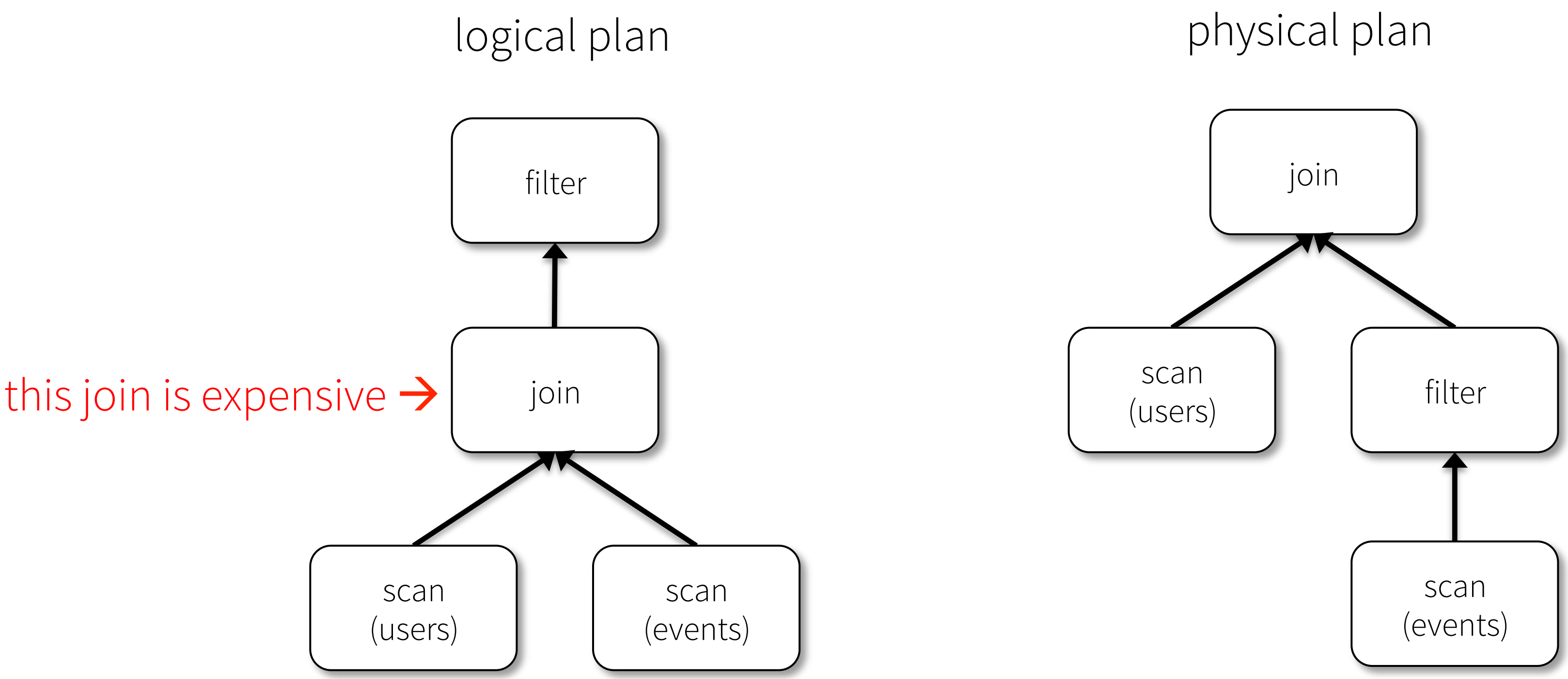
Plan Optimization & Execution



DataFrames and SQL share the same optimization/execution pipeline

Maximize code reuse & share optimization efforts

```
joined = users.join(events, users.id == events.uid)
filtered = joined.filter(events.date >= "2015-01-01")
```



Data Sources supported by DataFrames

built-in

- Parquet
- JDBC
- { JSON }
- PostgreSQL
- HIVE
- MySQL
- HDFS
- amazon web services S3
- H2

external

- AVRO
- CSV
- dBase
- APACHE HBASE
- elasticsearch.
- cassandra
- amazon web services Amazon Redshift

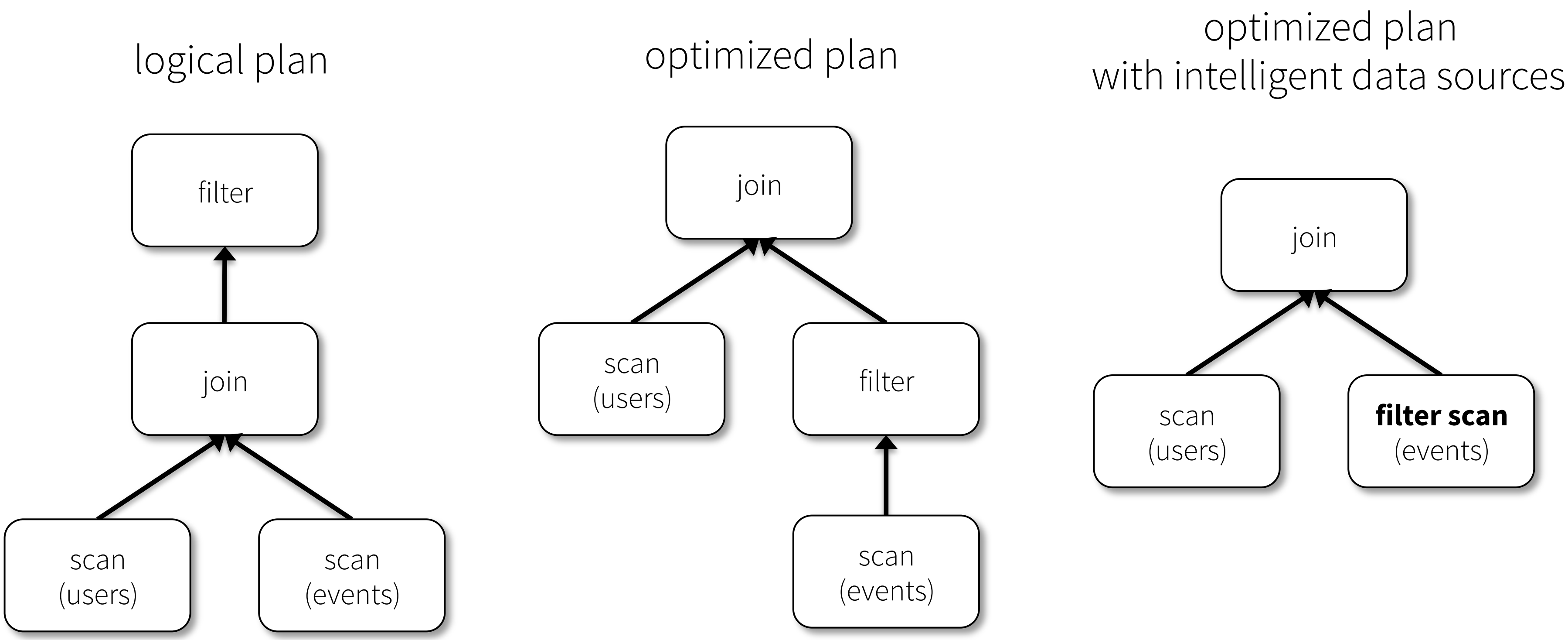
and more …

More Than Naïve Scans

Data Sources API can automatically prune columns and push filters to the source

- Parquet: skip irrelevant columns and blocks of data; turn string comparison into integer comparisons for dictionary encoded data
- JDBC: Rewrite queries to push predicates down

```
joined = users.join(events, users.id == events.uid)
filtered = joined.filter(events.date > ”2015-01-01”)
```



Our Experience So Far

SQL is wildly popular and important

- 100% of Databricks customers use some SQL

Schema is very useful

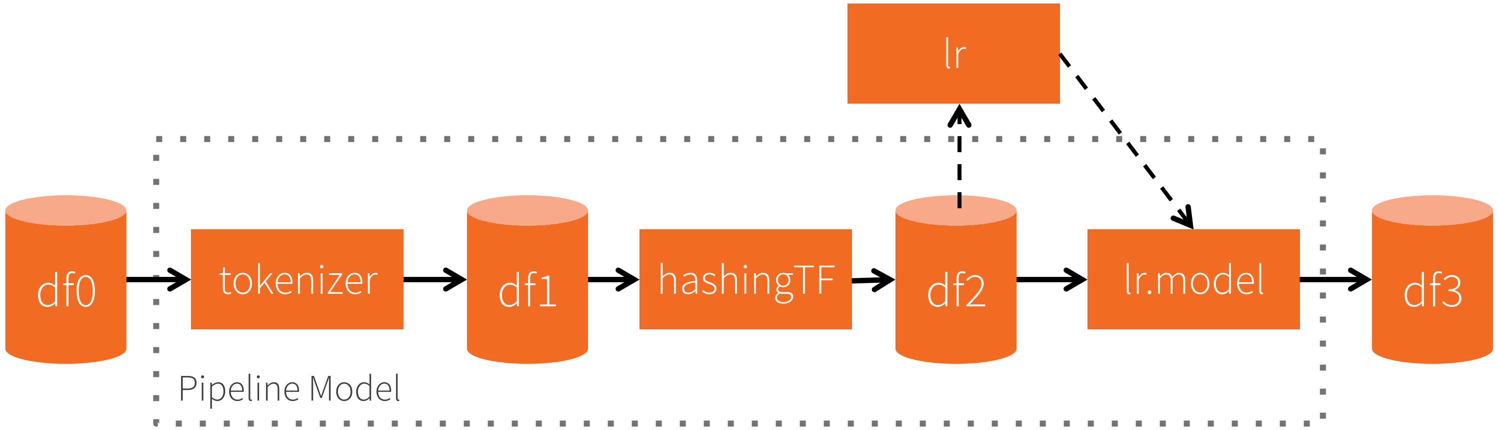
- Most data pipelines, even the ones that start with unstructured data, end up having some implicit structure
- Key-value too limited
- That said, semi-/un-structured support is paramount

Separation of logical vs physical plan

- Important for performance optimizations (e.g. join selection)

Machine Learning Pipelines

```
tokenizer = Tokenizer(inputCol="text", outputCol="words”)
hashingTF = HashingTF(inputCol="words", outputCol="features”)
lr = LogisticRegression(maxIter=10, regParam=0.01)
pipeline = Pipeline(stages=[tokenizer, hashingTF, lr])

df = sqlCtx.load("/path/to/data")
model = pipeline.fit(df)
```



R Interface (SparkR)

Spark 1.4 (June)

Exposes DataFrames, and ML library in R

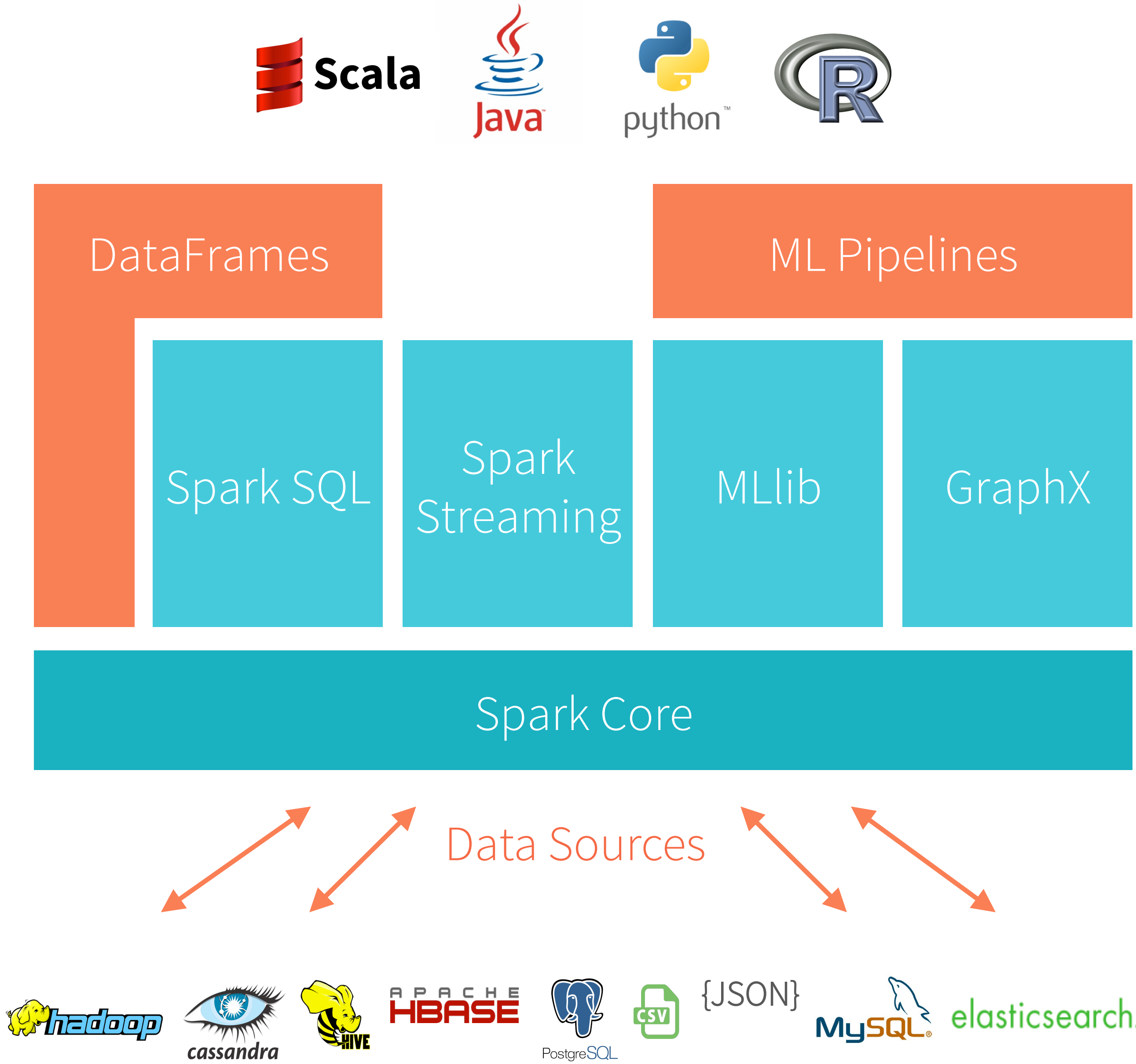
```
df = jsonFile("tweets.json")

summarize(
  group_by(
    df[df$user == "matei",],
    "date"),
  sum("retweets"))
```

Data Science at Scale

Higher level interfaces in Scala, Java, Python, R

Drastically easier to program Big Data

- With APIs similar to single-node tools



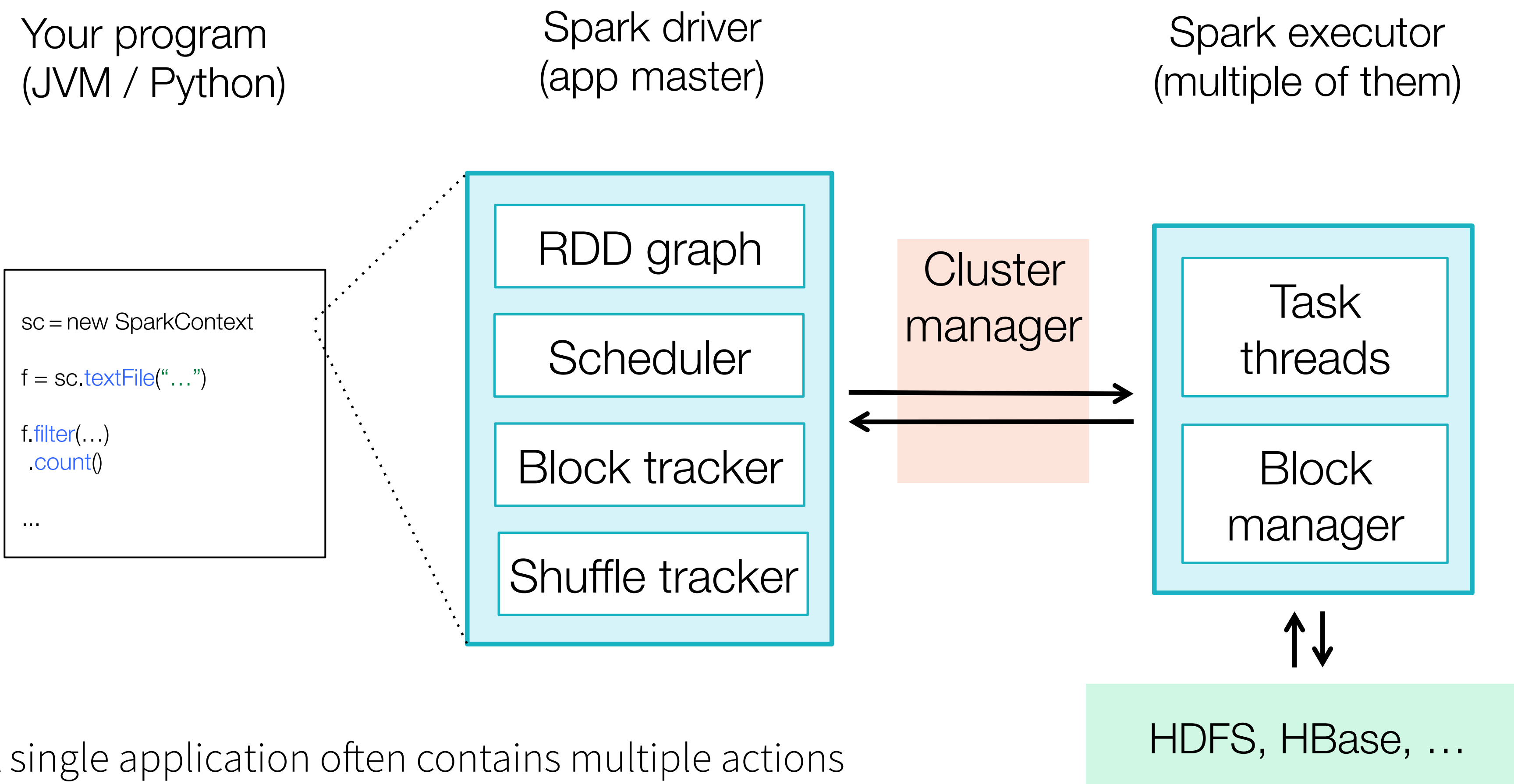
Scala
Java
python
R
DataFrames
ML Pipelines
Spark SQL
Spark Streaming
MLlib
GraphX
Spark Core
Data Sources
hadoop
cassandra
HIVE
APACHE HBASE
PostgreSQL
CSV
{JSON}
MySQL
elasticsearch.

Goal: unified engine across data sources, workloads and environments

Agenda

1. MapReduce Review
2. Introduction to Spark and RDDs
3. Generality of RDDs (e.g. streaming, ML)
4. DataFrames
5. Internals (time permitting)

Spark Application



A single application often contains multiple actions

RDD is an interface

1. Set of *partitions* (“splits” in Hadoop)
2. List of *dependencies* on parent RDDs
3. Function to *compute* a partition (as an Iterator) given its parent(s)

“lineage”

4. (Optional) *partitioner* (hash, range)
5. (Optional) *preferred location*(s) for each partition

optimized execution

databricks™

Example: HadoopRDD

partitions = one per HDFS block

dependencies = none

compute*(part)* = read corresponding block

preferredLocations*(part)* = HDFS block location

partitioner = none

Example: Filtered RDD

partitions = same as parent RDD

dependencies = “one-to-one” on parent

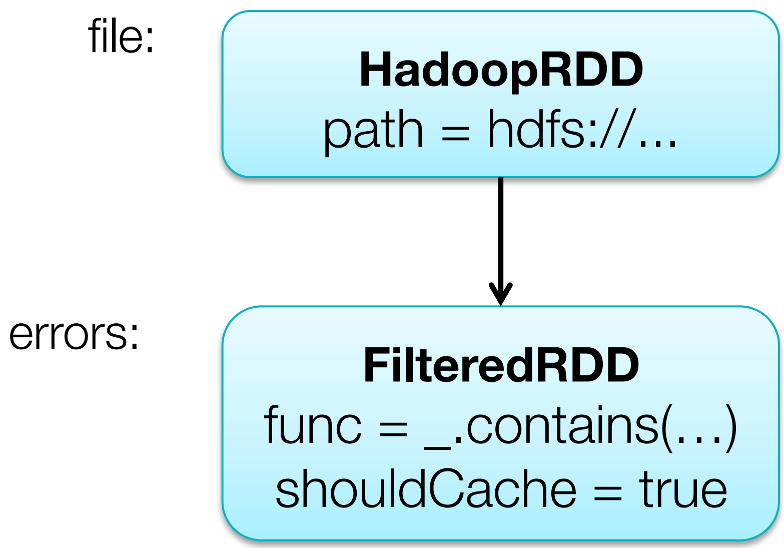
compute*(part)* = compute parent and filter it

preferredLocations*(part)* = none (ask parent)

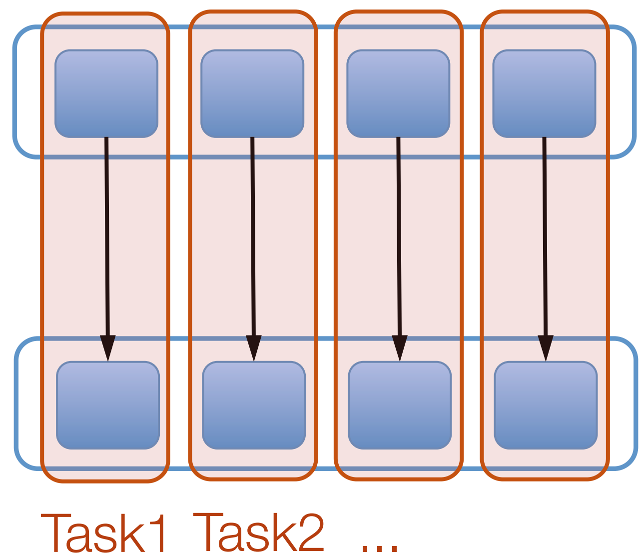
partitioner = none

RDD Graph (DAG of tasks)

Dataset-level view:



Partition-level view:



Example: JoinedRDD

partitions = one per reduce task

dependencies = "shuffle" on each parent

compute*(partition)* = read and join shuffled data

preferredLocations*(part)* = none

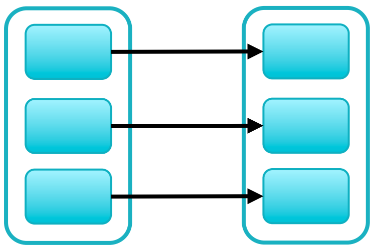
partitioner = HashPartitioner(numTasks)

Spark will now know this data is hashed!

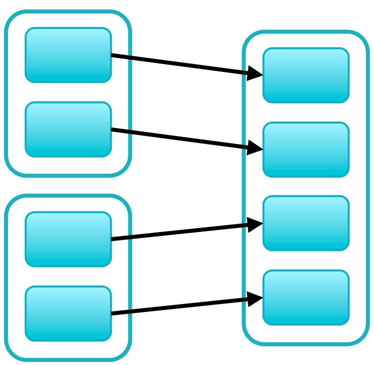
databricks

Dependency Types

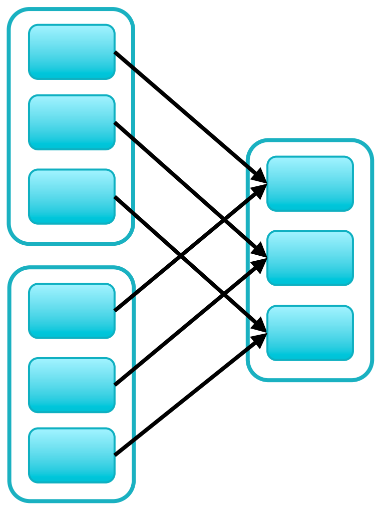
"Narrow" (pipeline-able)



map, filter

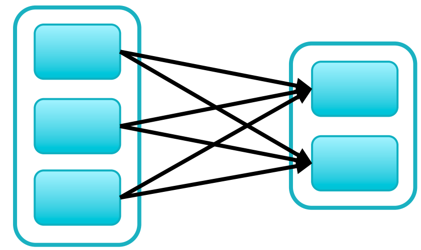


union

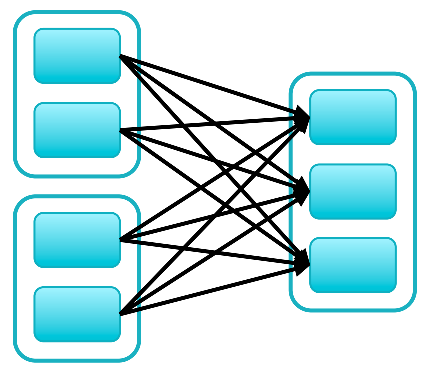


join with inputs co-partitioned

"Wide" (shuffle)

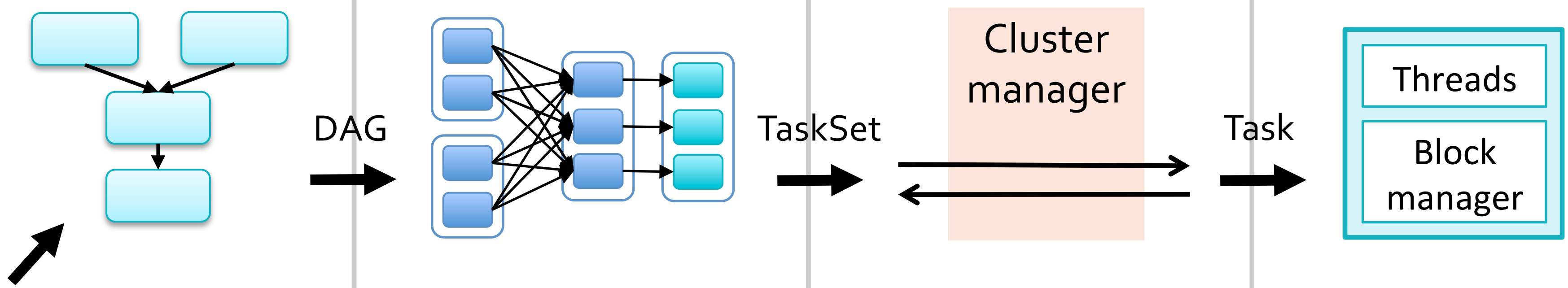


groupByKey on non-partitioned data



join with inputs not co-partitioned

Execution Process



RDD Objects

```
rdd1.join(rdd2)
    .groupBy(…)
    .filter(…)
```

build operator DAG

DAG Scheduler

split graph into *stages* of tasks

submit each stage as ready

Task Scheduler

launch tasks via cluster manager

retry failed or straggling tasks

Worker

execute tasks

store and serve blocks

databricks

DAG Scheduler

Input: RDD and partitions to compute

Output: output from actions on those partitions

Roles:

- Build stages of tasks
- Submit them to lower level scheduler (e.g. YARN, Mesos, Standalone) as ready
- Lower level scheduler will schedule data based on locality
- Resubmit failed stages if outputs are lost

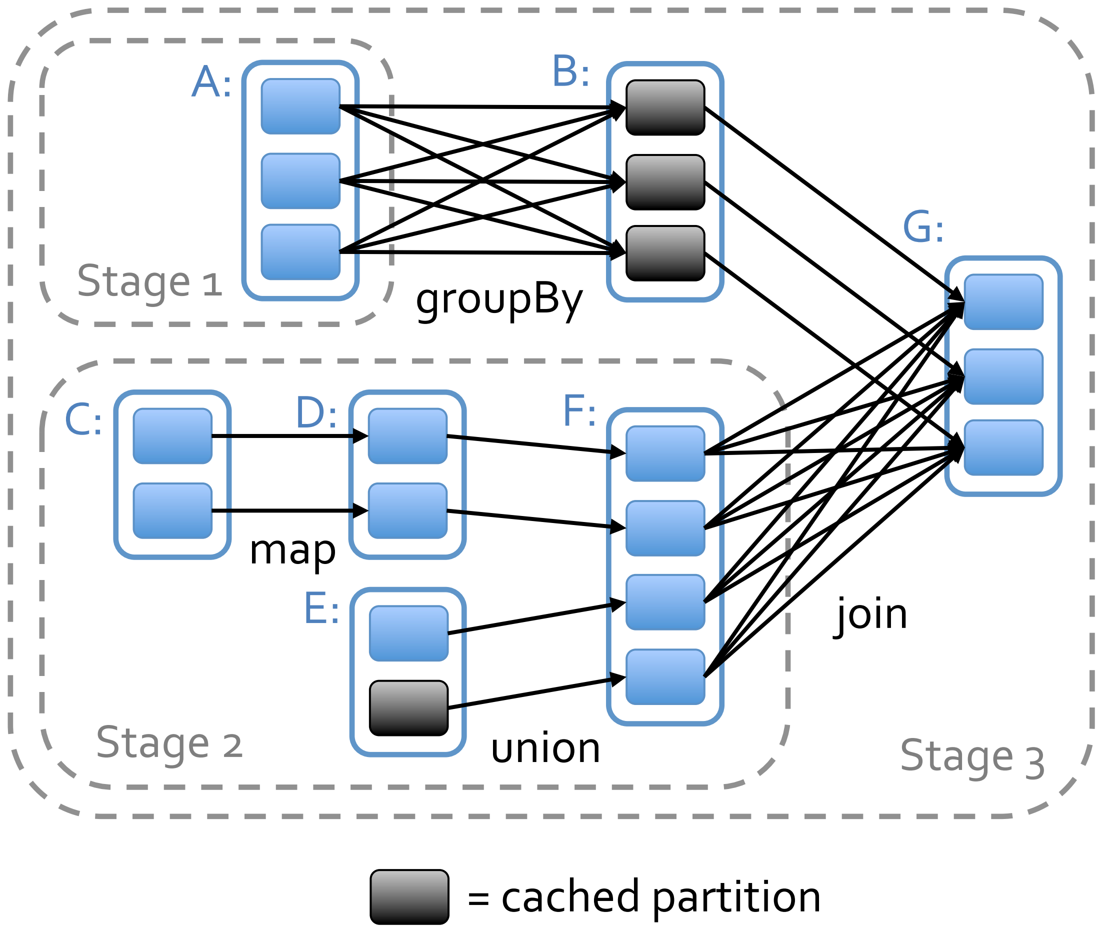
Job Scheduler

Captures RDD dependency graph

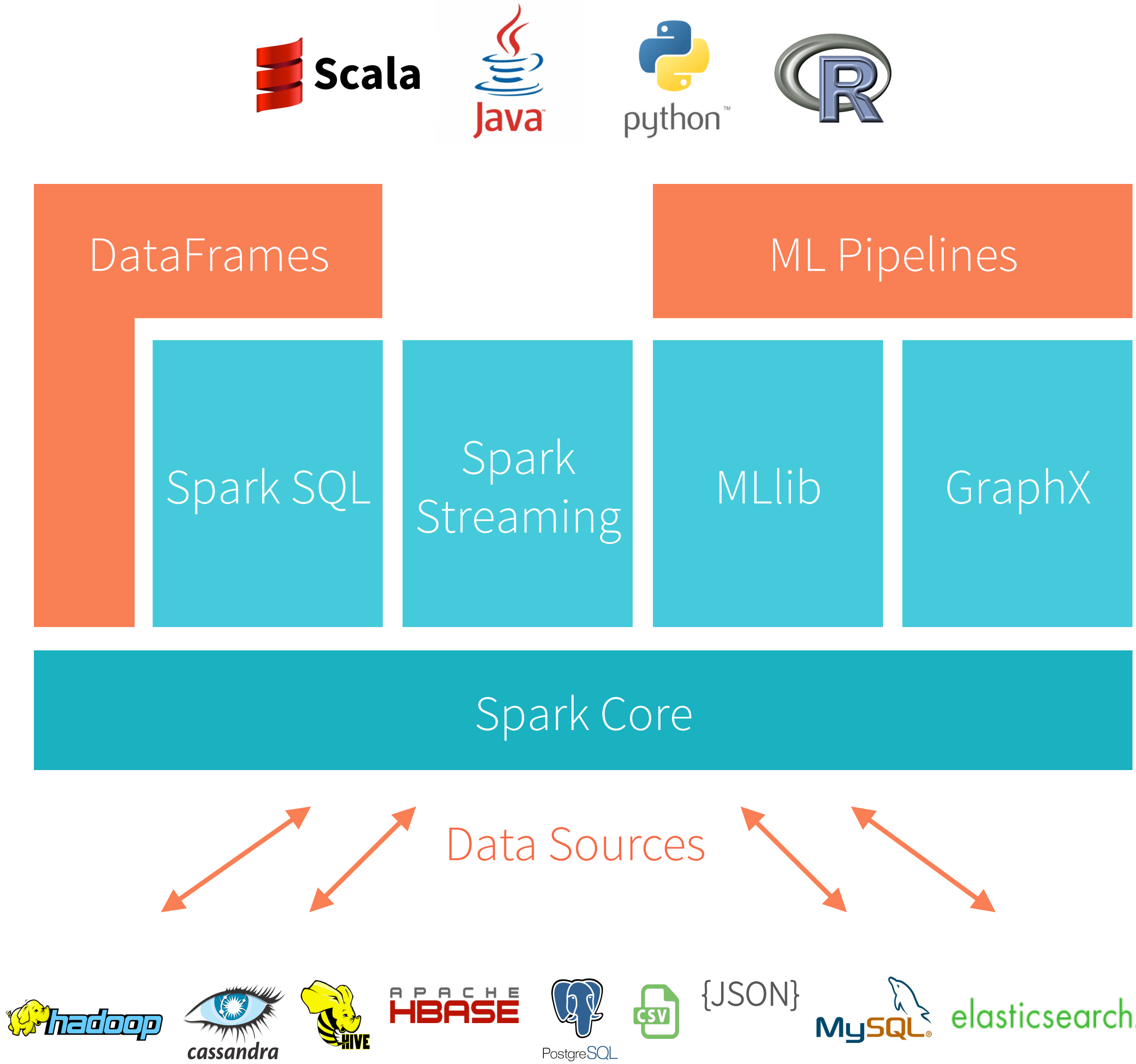
Pipelines functions into “stages”

Cache-aware for data reuse & locality

Partitioning-aware to avoid shuffles



databricks


Scala
Java
python
DataFrames
ML Pipelines
Spark SQL
Spark Streaming
MLlib
GraphX
Spark Core
Data Sources
hadoop
cassandra
HIVE
APACHE HBASE
PostgreSQL
CSV
{JSON}
MySQL
elasticsearch.

Goal: unified engine across data sources, workloads and environments


Spark summit 2015
DATA SCIENCE AND ENGINEERING AT SCALE
SAN FRANCISCO • JUNE 15 – 17
REGISTER NOW

Thank you. Questions?

@rxin

databricks™