



Processing Big Data with Small Programs

**Matei Zaharia, Mosharaf Chowdhury, Tathagata Das,
Ankur Dave, Haoyuan Li, Justin Ma, Murphy McCauley,
Michael Franklin, Scott Shenker, Ion Stoica**

spark-project.org

- **amplab** UC BERKELEY The amplab logo features the word "amplab" in a bold, sans-serif font. The letters "a", "m", "p", "l", and "b" are in orange, while "a", "m", "p", "l", and "b" are in dark blue. To the right of "amplab" is a stylized orange waveform consisting of several peaks and valleys.

Outline

The big data problem

MapReduce

Spark

How it works

Users' experience

The Big Data Problem

Data is growing faster than computation speeds

Growing data sources

- » Web, mobile, scientific, ...

Cheap storage

- » Doubling every 18 months

Stalling CPU speeds

- » Even multicores not enough



Examples

Facebook's daily logs: 60 TB

1000 genomes project: 200 TB

Google web index: 10+ PB

Cost of 1 TB of disk: \$50

Time to read 1 TB from disk: 6 hours (50 MB/s)

The Big Data Problem

Single machine can no longer process or even store all the data!

Only solution is to **distribute** over large clusters

Google Datacenter

How do we program this thing?



Traditional Network Programming

Message-passing between nodes

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 worse: stragglers (node is not failed, but 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

Biggest example: MapReduce

MapReduce

First widely popular programming model for data-intensive apps on clusters

Published by Google in 2004

- » Processes 20 PB of data / day

Popularized by open-source Hadoop project

- » 40,000 nodes at Yahoo!, 70 PB at Facebook

MapReduce Programming Model

Data type: key-value *records*

Map function:

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

Reduce function:

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

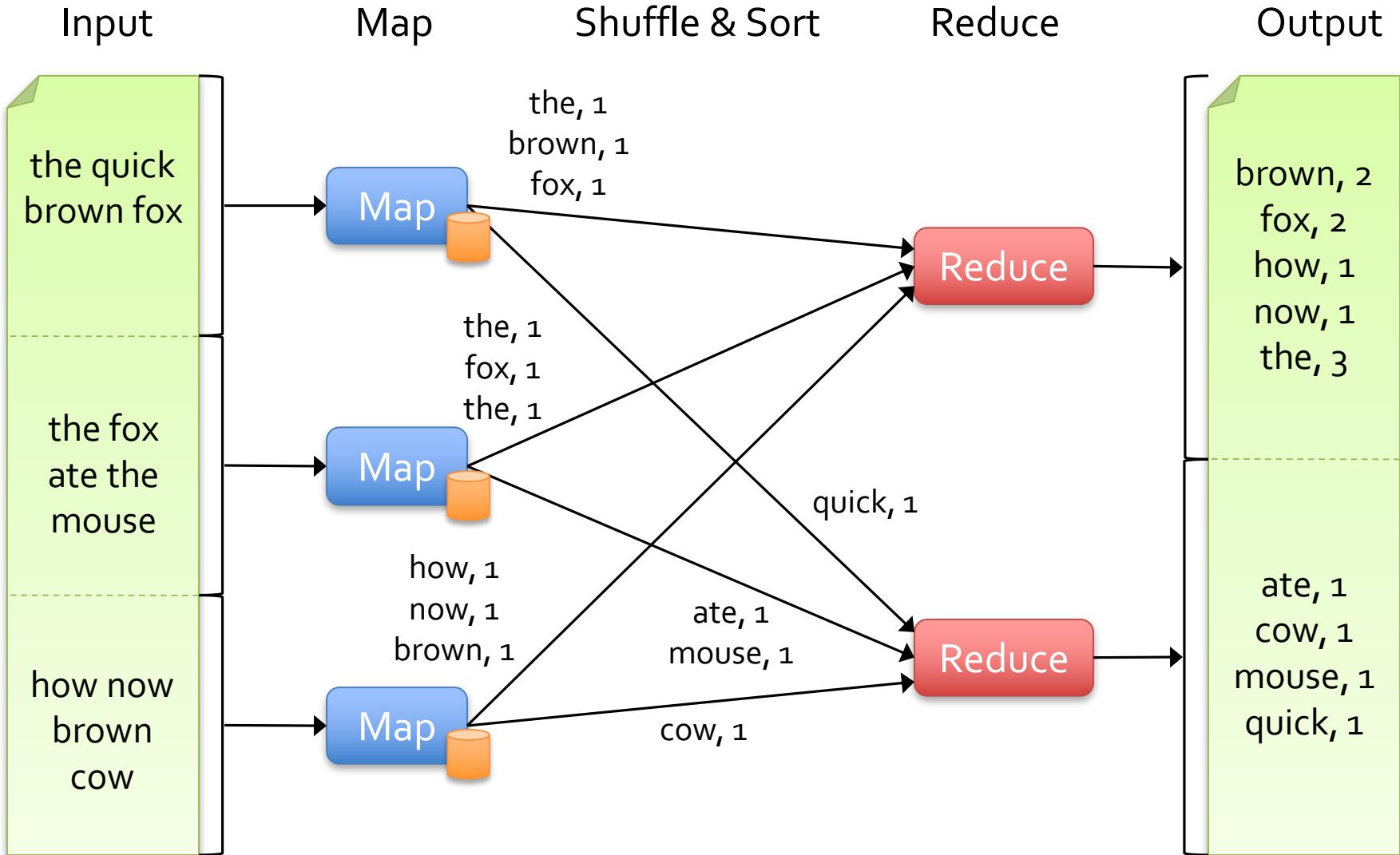
Example: Word Count

```
class Splitwords: public Mapper {  
public:  
    virtual void Map(const MapInput& input) {  
        const string& text = input.value();  
        const int n = text.size();  
        for (int i = 0; i < n; ) {  
            // Skip past leading whitespace  
            while ((i < n) && isspace(text[i]))  
                i++;  
            // Find word end  
            int start = i;  
            while ((i < n) && !isspace(text[i]))  
                i++;  
            if (start < i)  
                Emit(text.substr(start,i-start), "1");  
        }  
    }  
};
```

Example: Word Count

```
class Sum: public Reducer {  
public:  
    virtual void Reduce(ReduceInput* input) {  
        // Iterate over all entries with the  
        // same key and add the values  
        int64 value = 0;  
        while (!input->done()) {  
            value += StringToInt(input->value());  
            input->NextValue();  
        }  
        // Emit sum for input->key()  
        Emit(IntToString(value));  
    }  
};
```

Word Count Execution



MapReduce Execution

Automatically split work into many small *tasks*

Send map tasks to nodes based on data locality

Load-balance dynamically as tasks finish

Fault Recovery

If a task fails, re-run it and re-fetch its input

If a node fails, re-run its map tasks on others

If a task is slow, launch 2nd copy on other node

Summary

By providing a data-parallel model, MapReduce greatly simplified cluster programming:

- » Automatic division of job into tasks
- » Locality-aware scheduling
- » Load balancing
- » Recovery from failures & stragglers

But... the story doesn't end here!

When an Abstraction is Useful...

People want to compose it!

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

Problems with MapReduce

1. Programmability

- » Multi-step jobs create spaghetti code
 - 21 MR steps -> 21 mapper and reducer classes
- » Lots of boilerplate wrapper code per step
- » API doesn't provide type safety
 - Can pass the wrong Mapper class for a given data type

Problems with MapReduce

2. Performance

- » MR only provides acyclic data flow (read from disk -> process -> write to disk)
- » Expensive for applications that need to *reuse* data
 - Iterative algorithms (e.g. PageRank)
 - Interactive data mining (repeated queries on same data)
- » Users often hand-optimize by merging steps together

Spark

Aims to address both problems

Programmability: embedded DSL in Scala

- » Functional transformations on collections
- » Type-safe, automatically optimized
- » 5-10x less code than MR
- » Interactive use from Scala shell

Performance: in-memory computing primitives

- » Can run 10-100x faster than MR

Spark Programmability

Full Google WordCount:

```
#include "mapreduce/mapreduce.h"

// User's map function
class Splitwords: public Mapper {
public:
    virtual void Map(const MapInput& input)
    {
        const string& text = input.value();
        const int n = text.size();
        for (int i = 0; i < n; ) {
            // Skip past leading whitespace
            while (i < n && isspace(text[i]))
                i++;
            // Find word end
            int start = i;
            while (i < n && !isspace(text[i]))
                i++;
            if (start < i)
                Emit(text.substr(
                    start,i-start),"1");
        }
    };
REGISTER_MAPPER(Splitwords);

// User's reduce function
class Sum: public Reducer {
public:
    virtual void Reduce(ReduceInput* input)
    {
        // Iterate over all entries with the
        // same key and add the values
        int64 value = 0;
        while (!input->done()) {
            value += StringToInt(
                input->value());
            input->NextValue();
        }
        // Emit sum for input->key()
        Emit(IntToString(value));
    };
REGISTER_REDUCER(Sum);

int main(int argc, char** argv) {
    ParseCommandLineFlags(argc, argv);
    MapReduceSpecification spec;
    for (int i = 1; i < argc; i++) {
        MapReduceInput* in= spec.add_input();
        in->set_format("text");
        in->set_filepattern(argv[i]);
        in->set_mapper_class("Splitwords");
    }

    // Specify the output files
    MapReduceOutput* out = spec.output();
    out->set_filebase("/gfs/test/freq");
    out->set_num_tasks(100);
    out->set_format("text");
    out->set_reducer_class("Sum");

    // Do partial sums within map
    out->set_combiner_class("Sum");

    // Tuning parameters
    spec.set_machines(2000);
    spec.set_map_megabytes(100);
    spec.set_reduce_megabytes(100);

    // Now run it
    MapReduceResult result;
    if (!MapReduce(spec, &result)) abort();
    return 0;
}
```

Spark Programmability

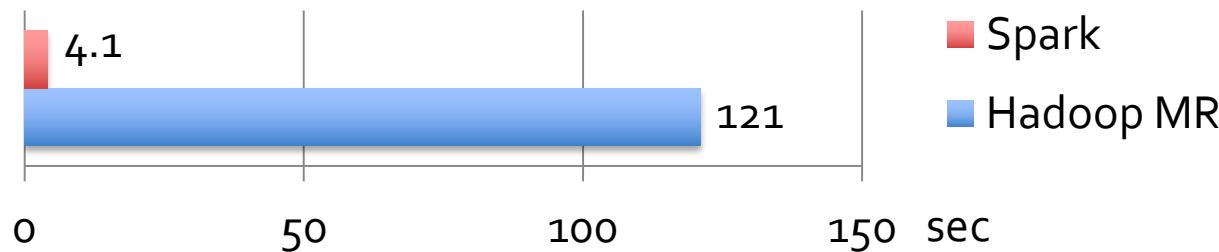
Spark WordCount:

```
val file = spark.textFile("hdfs://...")  
  
val counts = file.flatMap(line => line.split(" ")).  
    .map(word => (word, 1)).  
    .reduceByKey(_ + _)  
  
counts.save("out.txt")
```

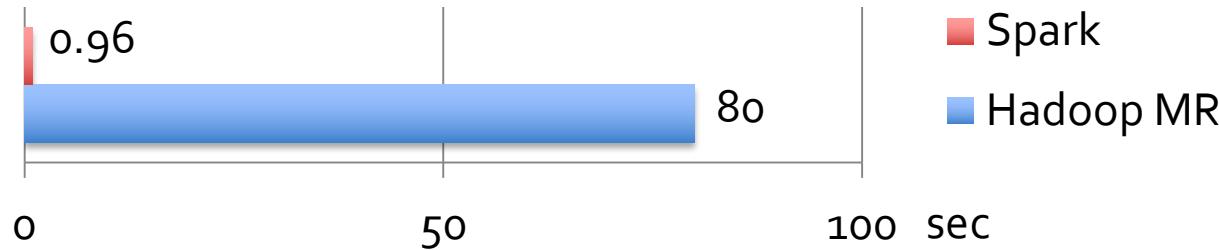
Spark Performance

Iterative machine learning apps:

K-means Clustering



Logistic Regression



Outline

The big data problem

MapReduce

Spark

How it works

Users' experience

Spark In More Detail

Three concepts:

- » Resilient distributed datasets (RDDs)
 - Immutable, partitioned collections of objects
 - May be *cached* in memory for fast reuse
- » Operations on RDDs
 - *Transformations* (define RDDs), *actions* (compute results)
- » Restricted shared variables (broadcast, accumulators)

Goal: make parallel programs look like local ones

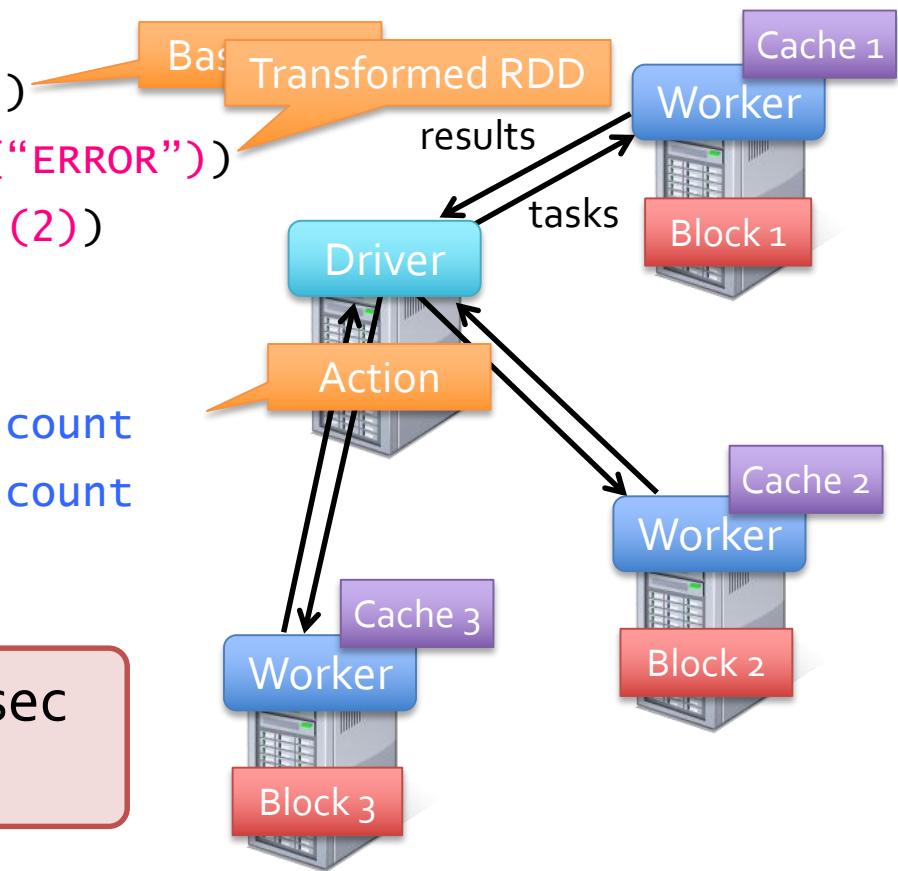
Example: Log Mining

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))  
messages.cache()
```

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

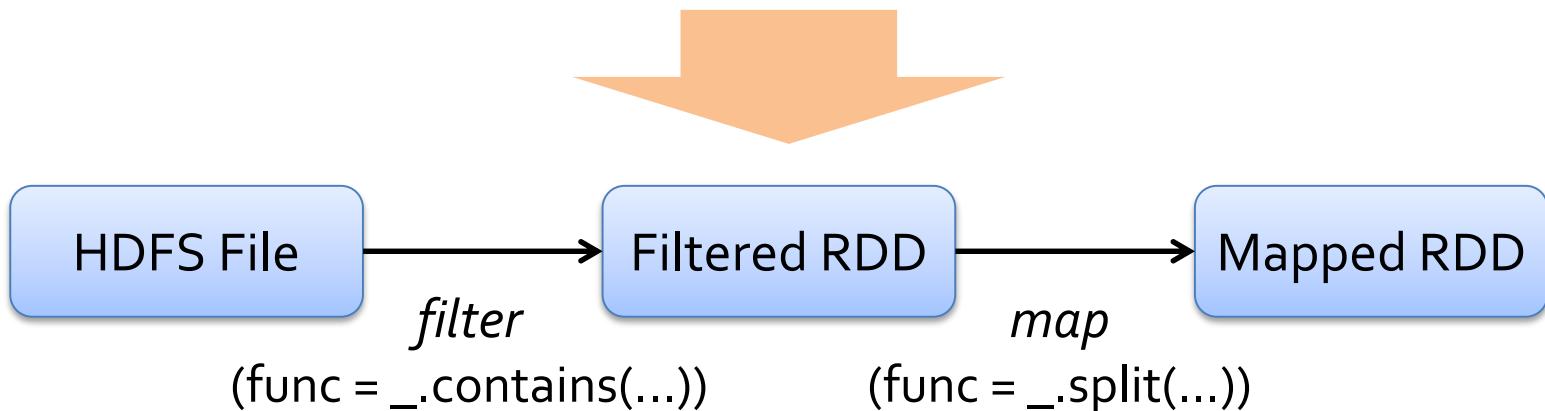
Result: search 1 TB data in 5-7 sec
(vs 170 sec for on-disk data)



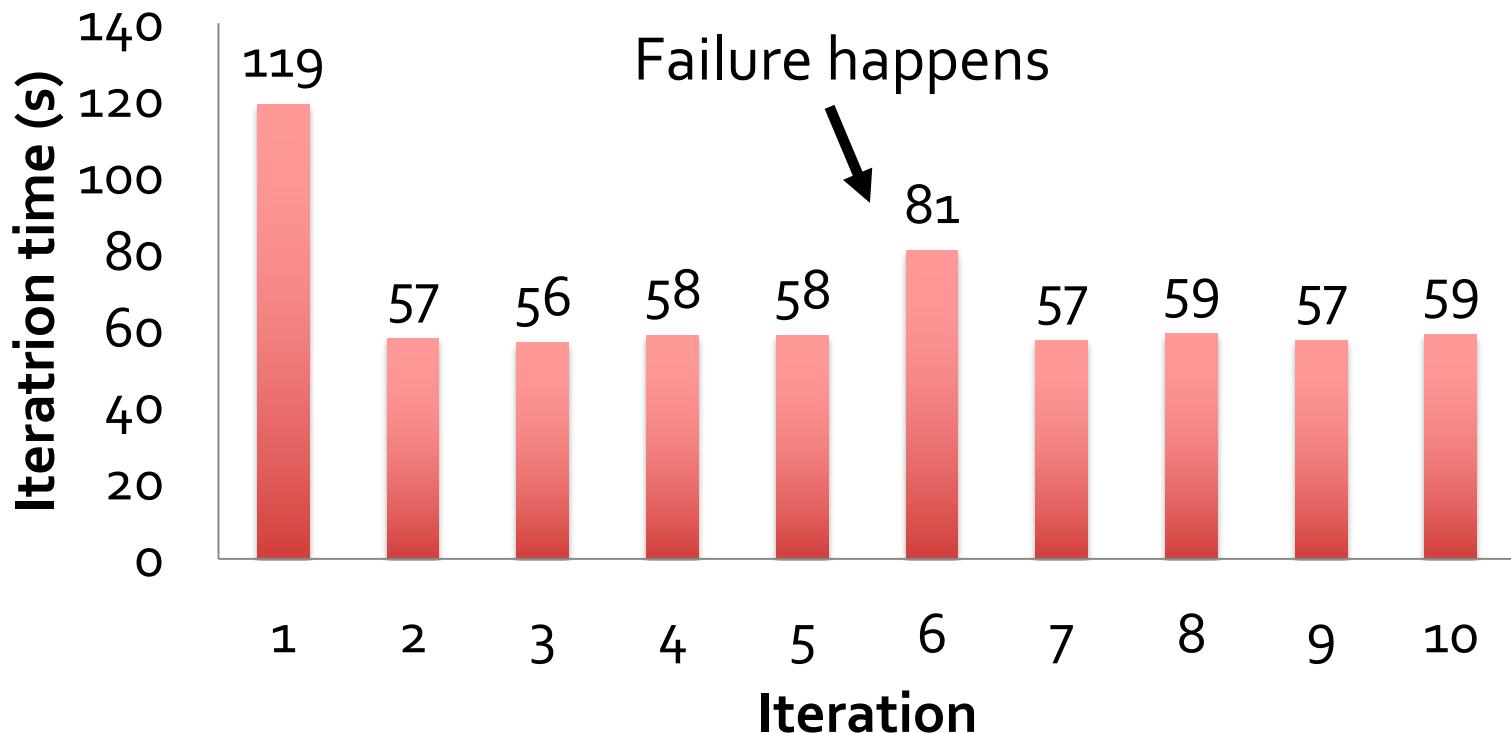
Fault Recovery

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

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

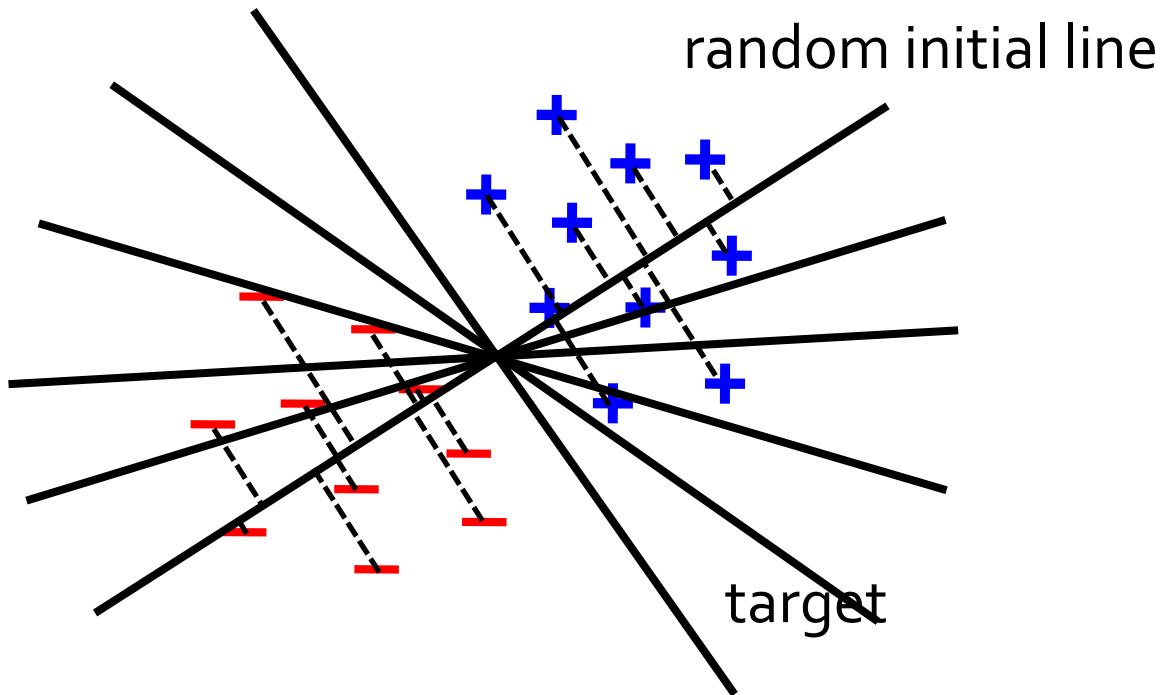


Fault Recovery Results



Example: Logistic Regression

Goal: find best line separating two sets of points

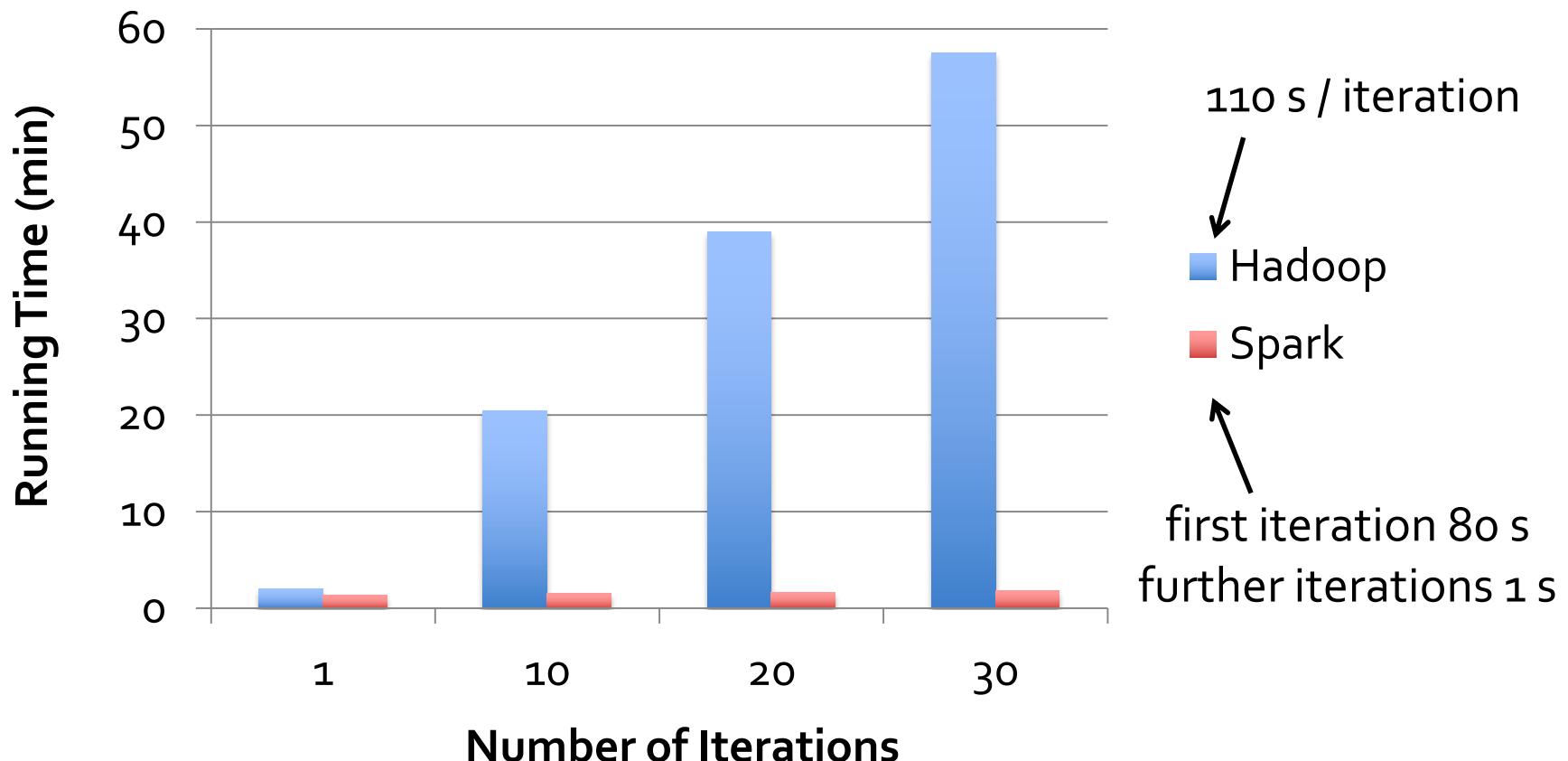


Example: Logistic Regression

```
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

Logistic Regression Performance



Other RDD Operations

Transformations (define a new RDD)	map filter sample groupByKey reduceByKey cogroup	flatMap union join cross mapValues ...
Actions (output a result)	collect reduce take fold	count saveAsTextFile saveAsHadoopFile ...

Demo

Beyond RDDs

So far we've seen that RDD operations can use variables from outside their scope

By default, each task gets a read-only copy of each variable (no sharing)

Good place to enable other sharing patterns!

- » Broadcast variables
- » Accumulators

Example: Collaborative Filtering

Goal: predict users' movie ratings based on past ratings of other movies

$$R = \begin{pmatrix} 1 & ? & ? & 4 & 5 & ? & 3 \\ ? & ? & 3 & 5 & ? & ? & 3 \\ 5 & ? & 5 & ? & ? & ? & 1 \\ 4 & ? & ? & ? & ? & 2 & ? \end{pmatrix}$$

← Movies →

↑
Users
↓

Model and Algorithm

Model R as product of user and movie feature matrices A and B of size $U \times K$ and $M \times K$

$$R = A B^T$$

Alternating Least Squares (ALS)

- » Start with random A & B
- » Optimize user vectors (A) based on movies
- » Optimize movie vectors (B) based on users
- » Repeat until converged

Serial ALS

```
var R = readRatingsMatrix(...)

var A = // array of U random vectors
var B = // array of M random vectors

for (i <- 1 to ITERATIONS) {
    A = (0 until U).map(i => updateUser(i, B, R))
    B = (0 until M).map(i => updateMovie(i, A, R))
}
```



Range objects

Naïve Spark ALS

```
var R = readRatingsMatrix(...)

var A = // array of U random vectors
var B = // array of M random vectors

for (i <- 1 to ITERATIONS) {
    A = spark.parallelize(0 until U, numSlices)
        .map(i => updateUser(i, B, R))
        .collect()
    B = spark.parallelize(0 until M, numSlices)
        .map(i => updateMovie(i, A, R))
        .collect()
}
```

Problem:
R re-sent
to all
nodes in
each
iteration

Efficient Spark ALS

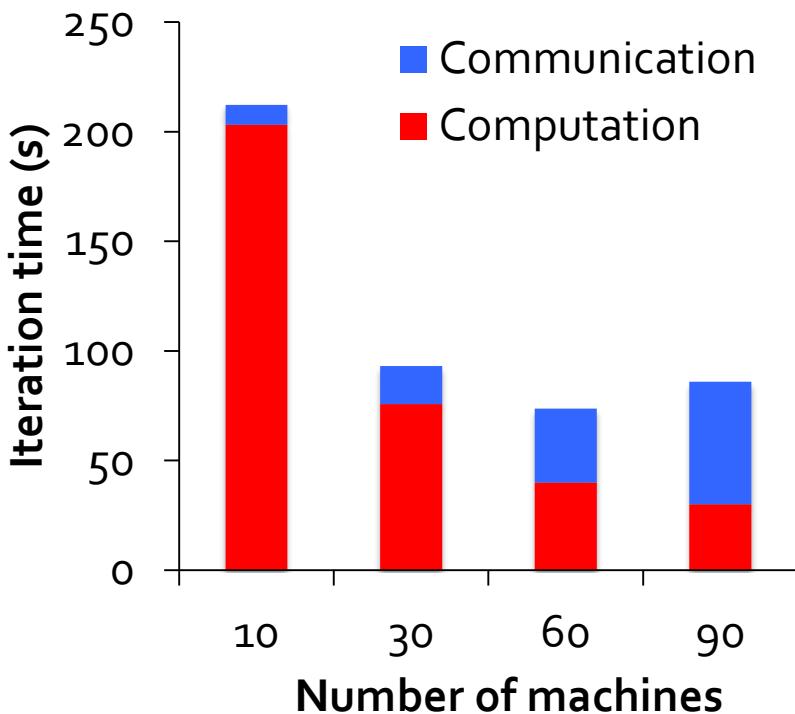
```
var R = spark.broadcast(readRatingsMatrix(...))  
  
var A = // array of U random vectors  
var B = // array of M random vectors  
  
for (i <- 1 to ITERATIONS) {  
    A = spark.parallelize(0 until U, numSlices)  
        .map(i => updateUser(i, B, R.value))  
        .collect()  
    B = spark.parallelize(0 until M, numSlices)  
        .map(i => updateMovie(i, A, R.value))  
        .collect()  
}
```

Solution:
mark R as
broadcast
variable

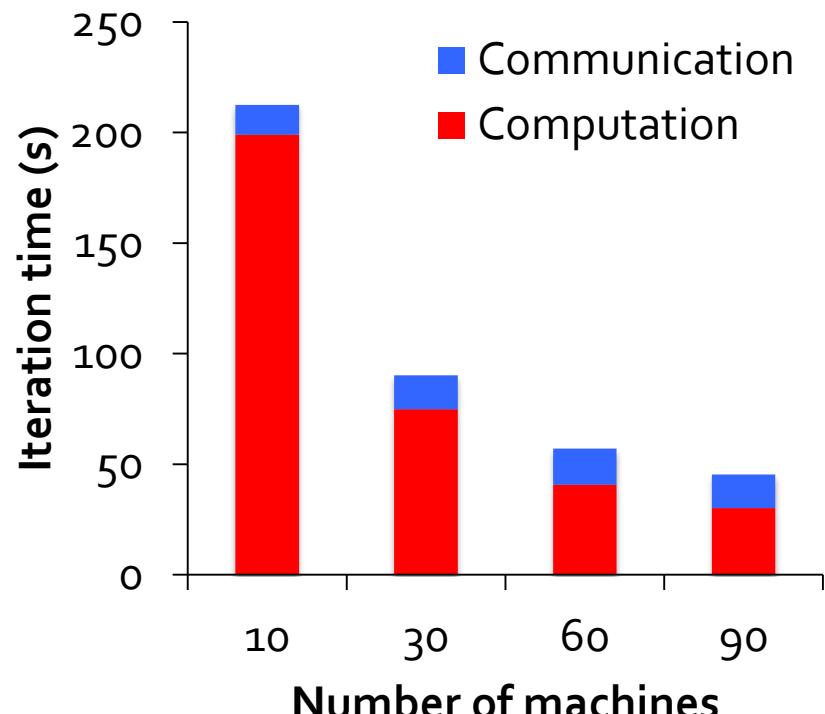
Result: 3x performance improvement

Scaling Up Broadcast

Initial version (HDFS)



Cornet P2P broadcast



Accumulators

Apart from broadcast, another common sharing pattern is aggregation

- » Add up multiple statistics about data
- » Count various events for debugging

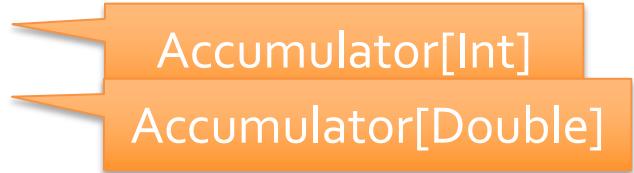
Spark's reduce operation does aggregation, but accumulators are another nice way to express it

Usage

```
val badRecords = sc.accumulator(0)
val badBytes = sc.accumulator(0.0)

records.filter(r => {
    if (isBad(r)) {
        badRecords += 1
        badBytes += r.size
    }
}).save(...)

printf("Total bad records: %d, avg size: %f\n",
    badRecords.value, badBytes.value / badRecords.value)
```



Accumulator[Int]
Accumulator[Double]

Accumulator Rules

Create with `sparkContext.accumulator(initialVal)`

“Add” to the value with `+=` inside tasks

- » Each task’s effect only counted once

Access with `.value`, but only on master

- » Exception if you try it on workers

Retains efficiency and fault tolerance!

Outline

The big data problem

MapReduce

Spark

How it works

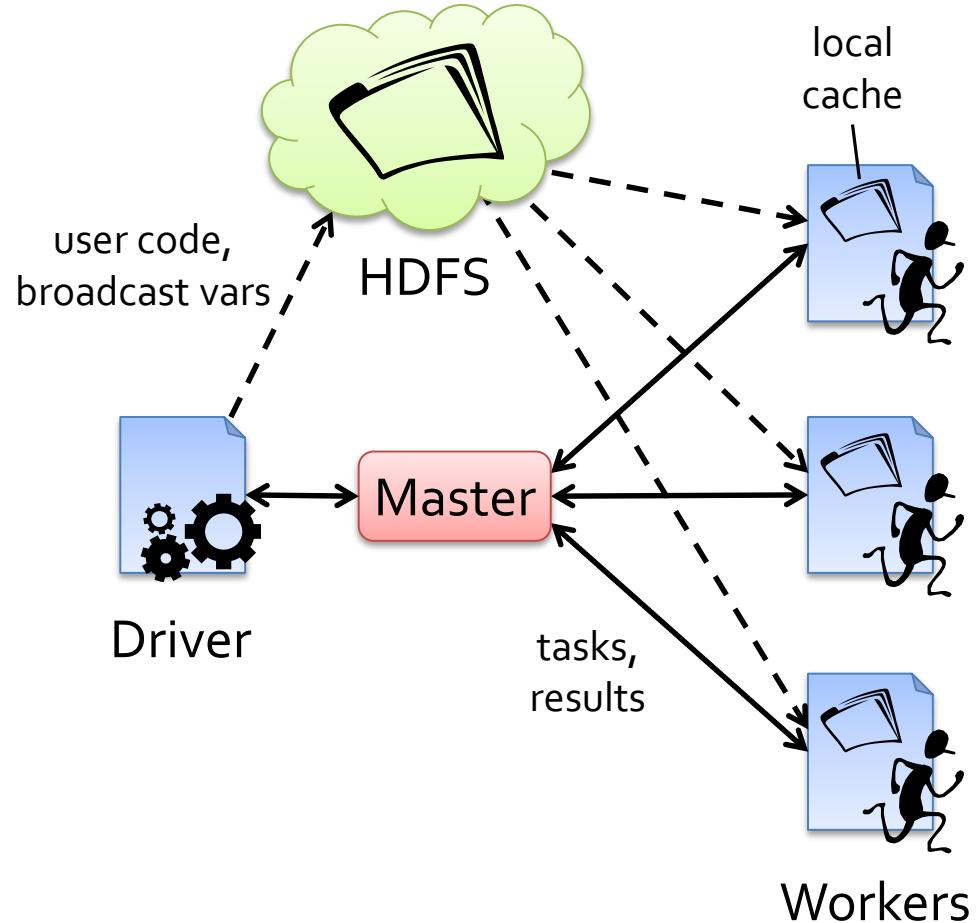
Users' experience

Components

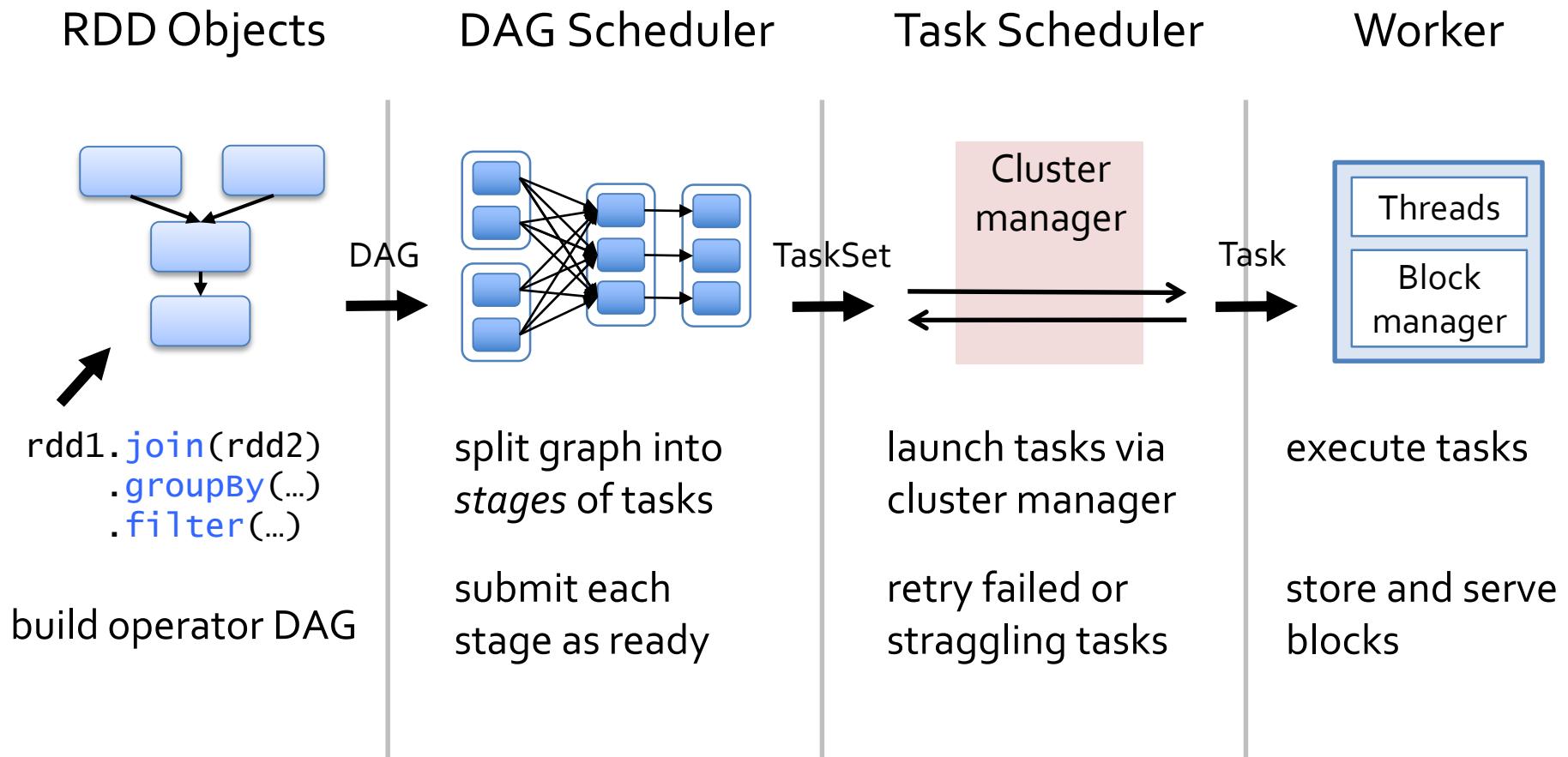
Driver program connects to cluster and schedules tasks

Workers run tasks, report results and variable updates

Data shared through Hadoop file system (HDFS)



Execution Process



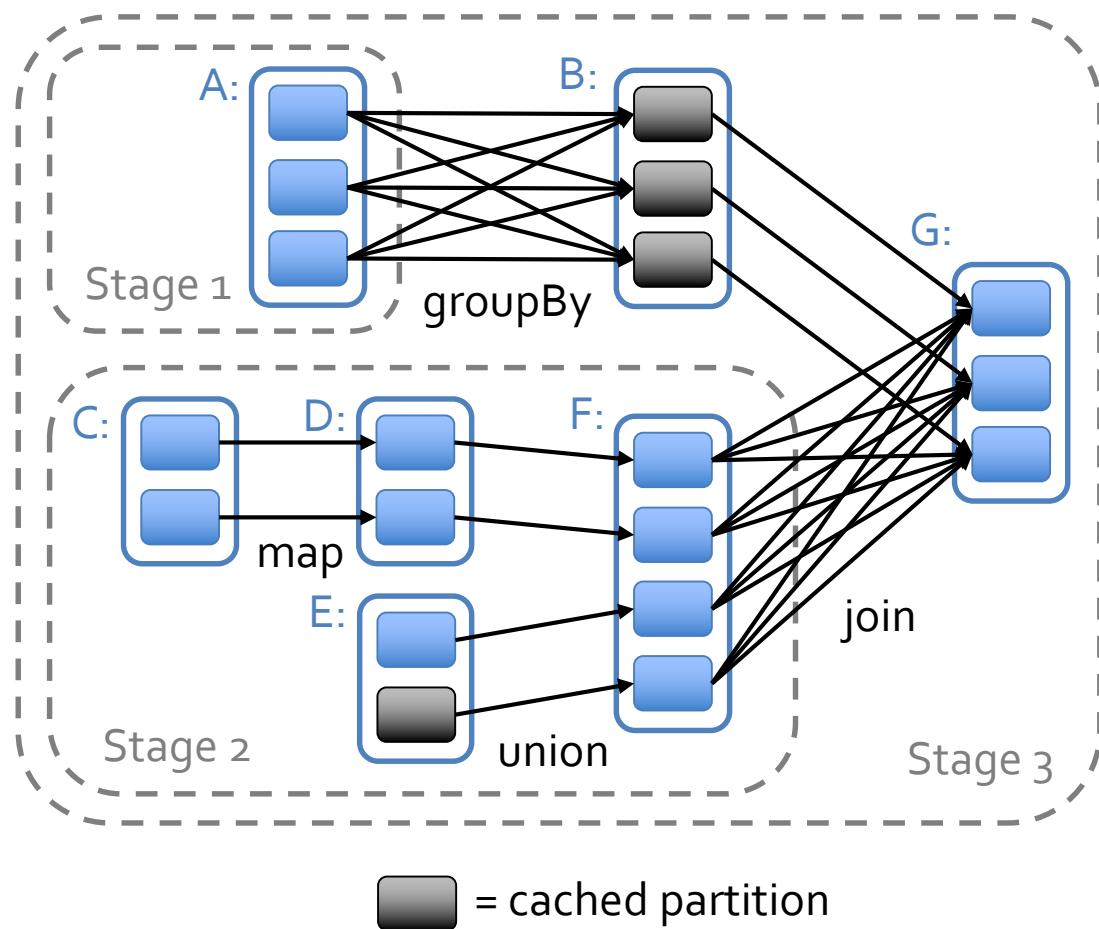
Job Scheduler

Captures RDD dependency graph

Pipelines functions into “stages”

Cache-aware for data reuse & locality

Partitioning-aware to avoid shuffles



Language Integration

No changes to Scala!

Scala closures are Serializable Java objects

- » Contain references to outer variables as fields
- » Serialize on driver, load & call on workers

```
val str = "hi"  
rdd.map(x => str + x)
```

```
class Closure1 {  
    String str;  
    String call(String x) {  
        return str + x;  
    }  
}
```

str: “hi”



Shared Variables

Play tricks with Java serialization!

```
class Broadcast<T> {  
    private transient T value; // will not be serialized  
    private int id = Broadcast.newID();  
  
    Broadcast(T value) {  
        this.value = value;  
        this.id = BroadcastServer.newId();  
        BroadcastServer.register(id, value);  
    }  
  
    // Called by Java when object is deserialized  
    private void readObject(InputStream s) {  
        s.defaultReadObject();  
        value = BroadcastFetcher.getOrFetch(id); // local registry  
    }  
}
```

Shared Variables

Play tricks with Java serialization!

```
class Accumulator<T> {
    private transient T value; // will not be serialized
    private int id = Accumulator.newID();
    ...

    // Called by Java when object is deserialized
    private void readObject(InputStream s) {
        s.defaultReadObject();
        value = zero(); // initialize local value to 0 of type T
        AccumulatorManager.register(this); // worker will send our
                                         // value back to master
    }

    void add(T t) { value.add(t); }
}
```

Interactive Shell

Scala interpreter compiles each line as,
essentially, a separate source file (!)

Modifications to allow use with Spark:

- » Altered code generation to make each line typed capture references to objects it depends on (as fields)
- » Added server to ship generated classes to workers

Outline

The big data problem

MapReduce

Spark

How it works

Users' experience

Some Users

CONVIVA®

foursquare

YAHOO!

quantiFind

KLOUT

UCSF

Berkeley
University of California



PRINCETON
UNIVERSITY

Carnegie
Mellon
University

500 user meetup, 12 companies contributing code

User Applications

Crowdsourced traffic estimation (Mobile Millennium)

Video analytics & anomaly detection (Conviva)

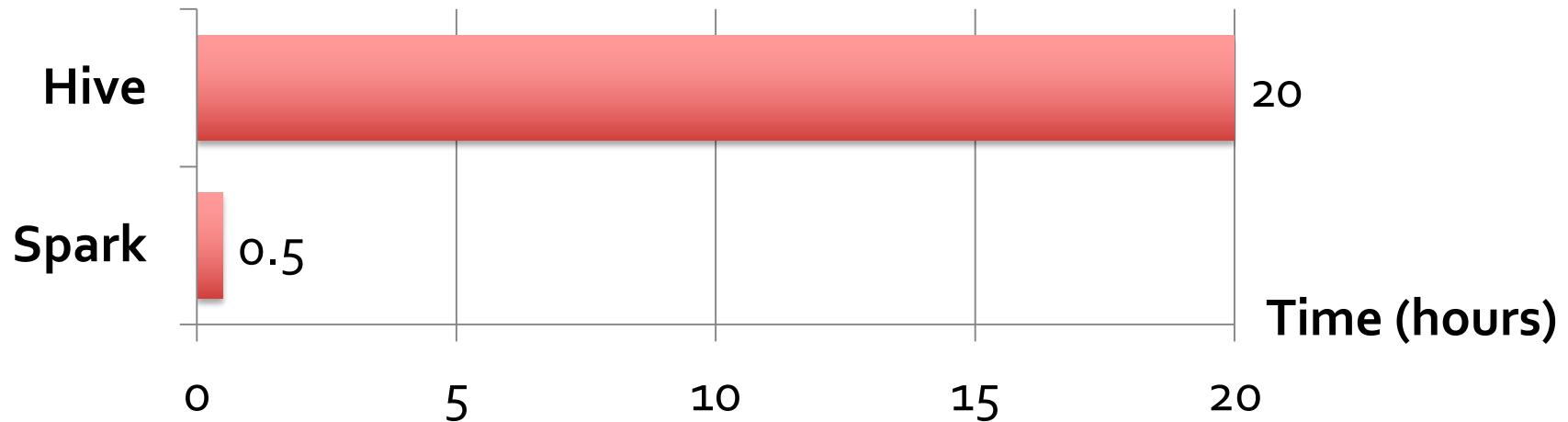
Ad-hoc queries from web app (Quantifind)

Twitter spam classification (Monarch)

DNA sequence analysis (SNAP)

...

Conviva GeoReport



SQL aggregations on many keys w/ same filter

40 × gain over Hive from avoiding repeated I/O,
deserialization and filtering

What do Users Say?

"Matei, we are doing predictive analytics on unstructured data and have been a Cascading/Pig shop [**Hadoop**]. We are a pretty small startup with 12 people and our first client is using our system to do predictions for [...].

Recently someone tried to compute the same predictions in straight Java and it turned out that a single box could outperform Pig running on 4 machines by several orders of magnitude. Then we started trying Spark and it was even better (the Java code was just a prototype and not optimized). Spark took the 60min Pig job running on 4 servers down to 12s running on 1 server.

So needless to say, we are porting our Pig code to Spark (which also shrinks it down to a much smaller size) and will be running out next product version on it."

What do Users Say?

"For simple queries, writing the query in Spark is harder than writing it in Hive [**SQL over Hadoop**]. However, it is much easier to write complicated queries in Spark than in Hive. Most real-world queries tend to be quite complex; hence the benefit of Spark. We can leverage the full power of the Scala programming language, rather than relying on the limited syntax offered by SQL. For example, the Hive expression *IF(os=1, "Windows", IF(os=2, "OSX", IF(os=3, "Linux", "Unknown"))*) can be replaced by a simple *match* clause in Scala. You can also use any Java/Scala library to transform the data."

What do Users Say?

Spark jobs are amazingly easy to test

Writing a test in Spark is as easy as:

```
class SparkTest {  
    @Test  
    def test() {  
        // this is real code...  
        val sc = new SparkContext("local", "MyUnitTest")  
        // and now some psuedo code...  
        val output = runYourCodeThatUsesSpark(sc)  
        assertAgainst(output)  
    }  
}
```

As a technical aside, this “local” mode starts up an in-process Spark instance, backed by a thread-pool, and actually opens up a few ports and temp directories, because it’s a real, live Spark instance. Granted, this is usually more work than you want to be done in an unit test (which ideally would not hit any file or network I/O), but the redeeming quality is that it’s *fast*. Tests run in ~2 seconds.

Interesting Takeaways

Users are as excited about the ease of use as about the performance

- » Even seasoned distributed programmers

Ability to use a “full” programming language (classes, functions, etc) is appreciated over SQL

- » Hard to see in small examples, but matters in big apps

Embedded nature of DSL helps w/ software eng.

- » Call Spark in unit tests, call into existing Scala code
- » Important in any real software eng. setting!

Spark in Java and Python

To further expand Spark's usability, we've now invested substantial effort to add 2 languages

Both support all the Scala features (RDDs, accumulators, broadcast vars)

Spark in Java

```
lines.filter(_.contains("error")).count()
```



```
JavaRDD<String> lines = sc.textFile(...);
```

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

Spark in Python

```
lines = sc.textFile(sys.argv[1])  
  
counts = lines.flatMap(lambda x: x.split(' ')) \  
    .map(lambda x: (x, 1)) \  
    .reduceByKey(lambda x, y: x + y)
```

Usable interactively from Python shell
Coming out this month

Conclusion

Spark makes parallel programs faster to write & run with model based on distributed collections

- » User API resembles working with local collections
- » Caching & lineage-based recovery = fast data sharing

Gets nice syntax while staying soft. eng. friendly

Might be fun to build DSLs on top of!

www.spark-project.org

