Pregel and Spark

Plan for today

Generalizing the computation model

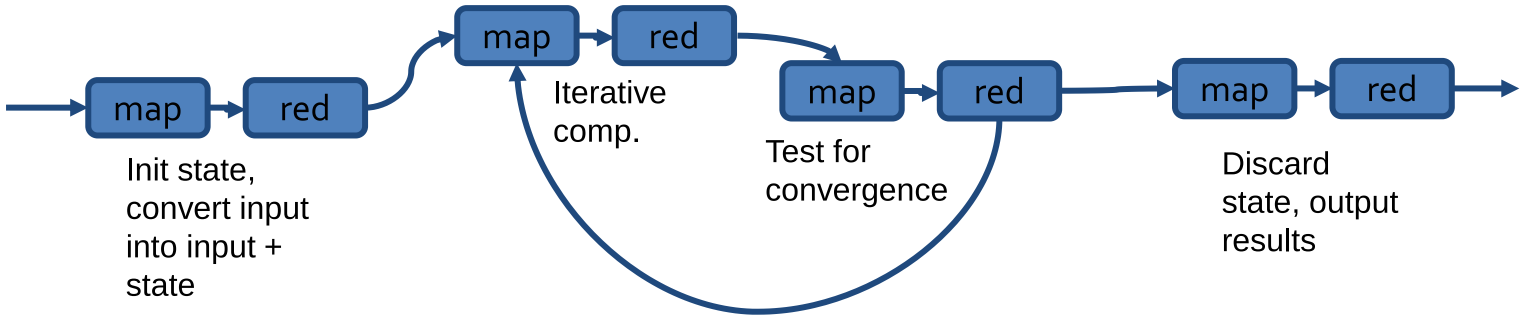
» Bulk synchronous parallelism (BSP)

NEXT

Pregel

Spark

Recall: Iterative computation in MapReduce



MapReduce is functional

- map() and reduce() 'forget' all state between iterations
- Hence, we have no choice but to put the state into the intermediate results
- This is a bit cumbersome

What if we could remember?

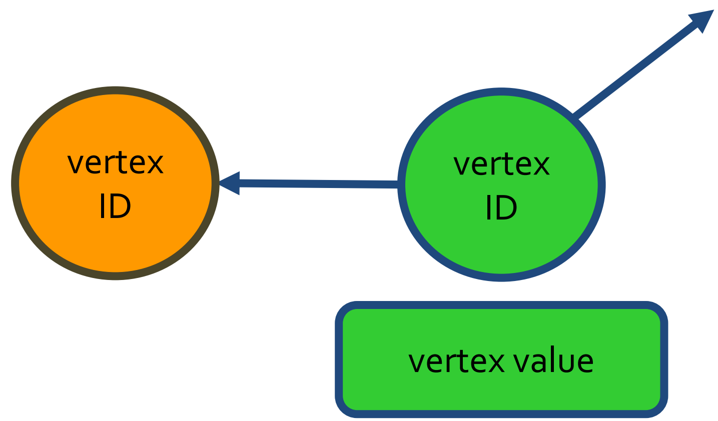
Suppose we were to change things entirely:

» Graph is partitioned across a set of machines
» State is kept entirely in memory
» Computation consists of message passing, i.e., sending updates from one portion to another

Let’s look at two versions of this:

» Pregel (Malewicz et al., SIGMOD'10 – Google's version)
» Spark (Zaharia et al., NSDI'12 – UC Berkeley's version)

Let's think about the MapReduce model



How does MapReduce process graphs?

» "Think like a vertex"

» What do the vertices do?

» What are the edges, really?

How good a fit is MapReduce's keys → values model for this?

» ... and what are the consequences?

The BSP model

This is similar to the bulk-synchronous parallelism (BSP) model

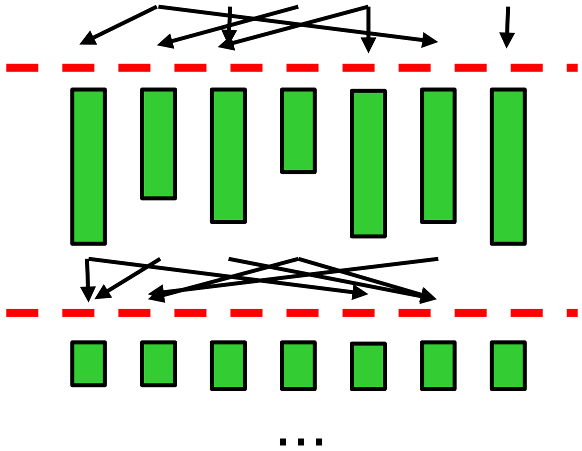
- Developed by Leslie Valiant at Harvard during the 1980s

BSP computations consist of:

- Lots of components that process data
- A network for communication, and a way to synchronize

Three distinct phases:

- Concurrent computation
- Communication
- Barrier synchronization
- Repeat



Valiant, "A bridging model for parallel computation", CACM Vol. 33 No. 8, Aug. 1990

Properties of the BSP model

Can BSP computations have:

» Deadlocks?

» Race conditions?

» If so, when? If not, why note?

How well do BSP computations scale? Why?

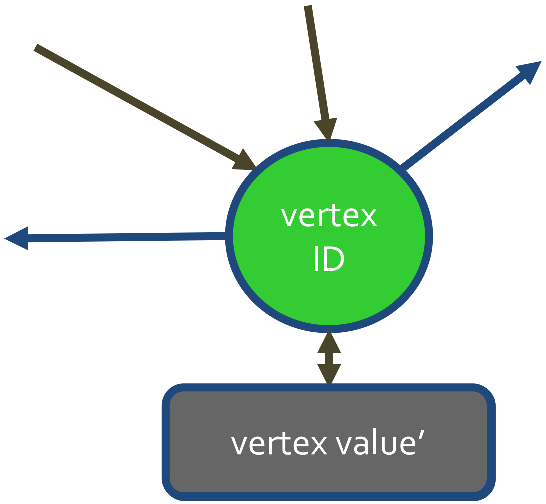
Generalizing the computation model ✔

» Bulk synchronous parallelism (BSP) ✔

Pregel ⇐ NEXT

Spark

The basic Pregel execution model

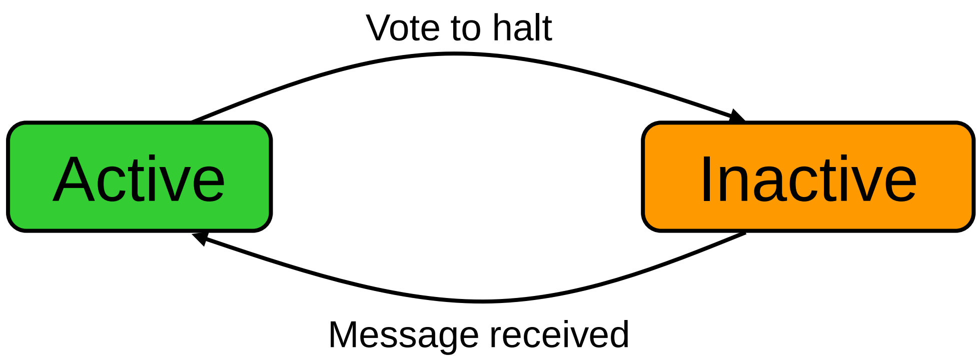


A sequence of supersteps, for each vertex:

- Receive incoming messages
- Compute()
  - Update value / state
- Send outgoing messages
- Optionally change toplogy

Malewicz et al., "Pregel: a system for large-scale graph processing", Proc. ACM SIGMOD 2010

Pregel: Termination test



How do we know when the computation is done?

» Vertexes can be active or inactive

» Each vertex can independently vote to halt, transition to inactive

» Incoming messages reactivate the vertex

» Algorithm terminates when all vertexes are inactive

» Examples of when a vertex might vote to halt?

Pregel: A simple example (max value)

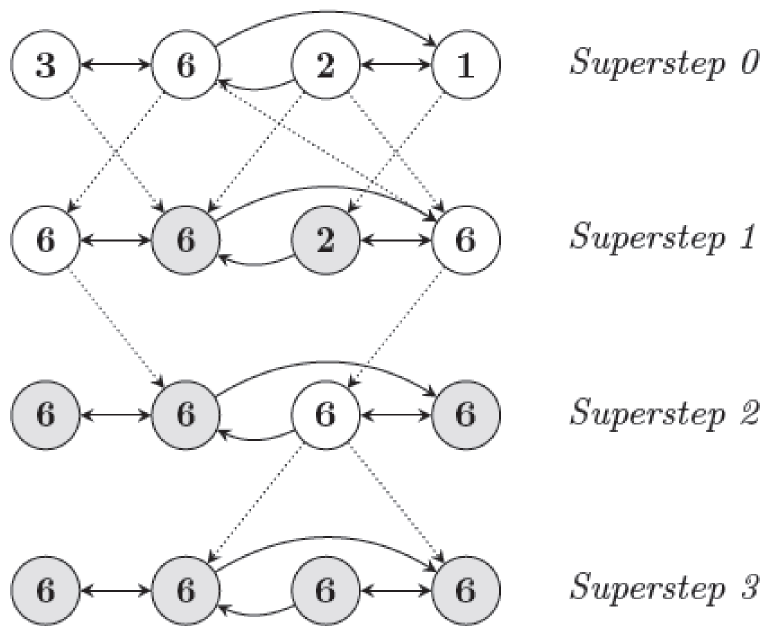


Figure 2: Maximum Value Example. Dotted lines are messages. Shaded vertices have voted to halt.

Pregel: Producing output

Output is the set of values explicitly output by the vertices

- Often a directed graph isomorphic to the input...
- ... but it doesn't have to be (edges can be added or removed)
- Example: Clustering algorithm

What if we need some global statistic instead?

- Example: Number of edges in the graph, average value
- Each vertex can output a value to an aggregator in superstep S
- System combines these values using a form of 'reducer', and result is available to all vertexes in superstep S+1
- Aggregators need to be commutative and associative (why?)

The Pregel API in C++

- A Pregel program is written by sub-classing the Vertex class:

```
template <typename VertexValue,
typename EdgeValue,
typename MessageValue>

class Vertex {
public:
    virtual void Compute(MessageIterator* msgs) = 0;

    const string& vertex_id() const;
    int64 superstep() const;
    const VertexValue& GetValue();
    VertexValue* MutableValue();
    OutEdgeIterator GetOutEdgeIterator();

    void SendMessageTo(const string& dest_vertex,
    const MessageValue& message);

    void VoteToHalt();
};
```

To define the types for vertices, edges and messages

Override the compute function to define the computation at each superstep

To get the value of the current vertex

To modify the value of the vertex

To pass messages to other vertices

Pregel Code for Finding the Max Value

```
Class MaxFindVertex
        : public Vertex<double, void, double> {
  public:
        virtual void Compute(MessageIterator* msgs) {
                int currMax = GetValue();
                SendMessageToAllNeighbors(currMax);
                for ( ; !msgs->Done(); msgs->Next()) {
                        if (msgs->Value() > currMax)
                                currMax = msgs->Value();
                }
                if (currMax > GetValue())
                        *MutableValue() = currMax;
                else VoteToHalt();
        }
};
```

Example: PageRank in Pregel

```
class PageRankVertex : public Vertex<double, void, double> {
public:
  virtual void Compute(MessageIterator* msgs) {
    if (superstep() >= 1) {
      double sum = 0;
      for (; !msgs->Done(); msgs->Next())
        sum += msgs->Value();
      *MutableValue() = 0.15 / NumVertices() + 0.85 * sum;
    }

    if (superstep() < 30) {
      const int64 n = GetOutEdgeIterator().size();
      SendMessageToAllNeighbors(GetValue() / n);
    } else {
      VoteToHalt();
    }
  }
};
```

Pregel: Additional complications

How to coordinate?

» Basic Master/worker design (just like MapReduce)

How to achieve fault tolerance?

» Crucial!! Why?

» Failures detected via heartbeats (just like in MapReduce)

» Uses checkpointing and recovery

» Basic checkpointing vs. confined recovery

How to partition the graph among the workers?

» Very tricky problem!

» Addressed in much more detail in later work

Summary: Pregel

Bulk Syncronous Parallelism – sequence of synchronized supersteps

Consider the nodes to have state (memory) that carries from superstep to superstep

Connections to MapReduce model?

Plan for today

Generalizing the computation model ✓
» Bulk synchronous parallelism (BSP) ✓

Pregel ✓

Spark ⬅ NEXT

Another Abstraction: Spark

Let's think of just having a big block of RAM, partitioned across machines…

» And a series of operators that can be executed in parallel across the different partitions

That's basically Spark's resilient distributed datasets (RDDs)

» Spark programs are written by defining functions to be called over items within collections

» (similar model to LINQ, FlumeJava, Apache Crunch, and several other environments)

Zaharia et al., "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing", Proc. NSDI 2012

Outline

Spark programming model

Implementation

User applications

Programming Model

Resilient distributed datasets (RDDs)

» Immutable, partitioned collections of objects
» Created through parallel *transformations* (map, filter, groupBy, join, …) on data in stable storage
» Can be *cached* for efficient reuse

Actions on RDDs

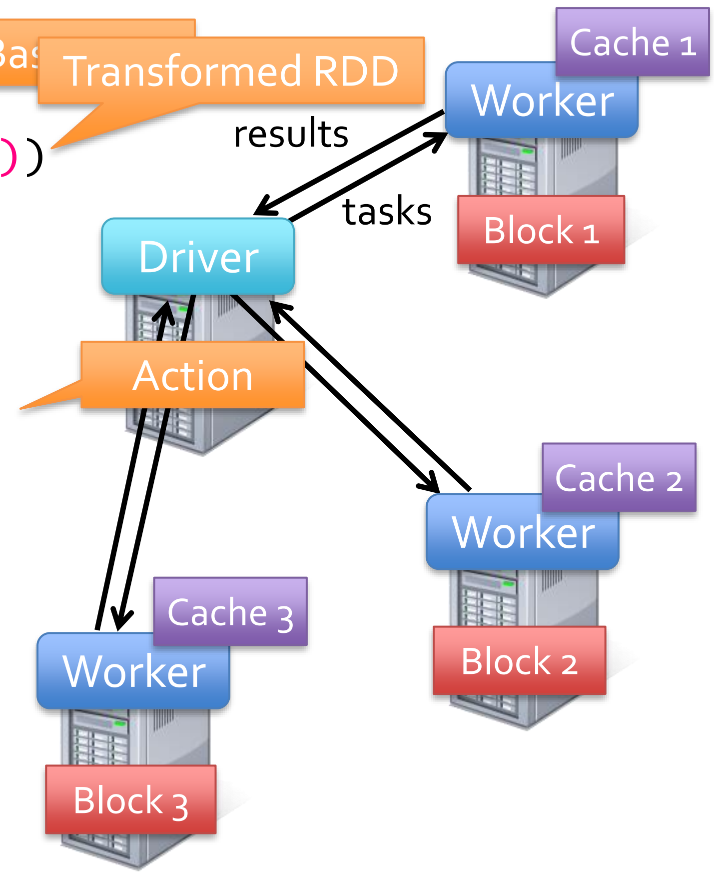
» Count, reduce, collect, save, …

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
. . .
```

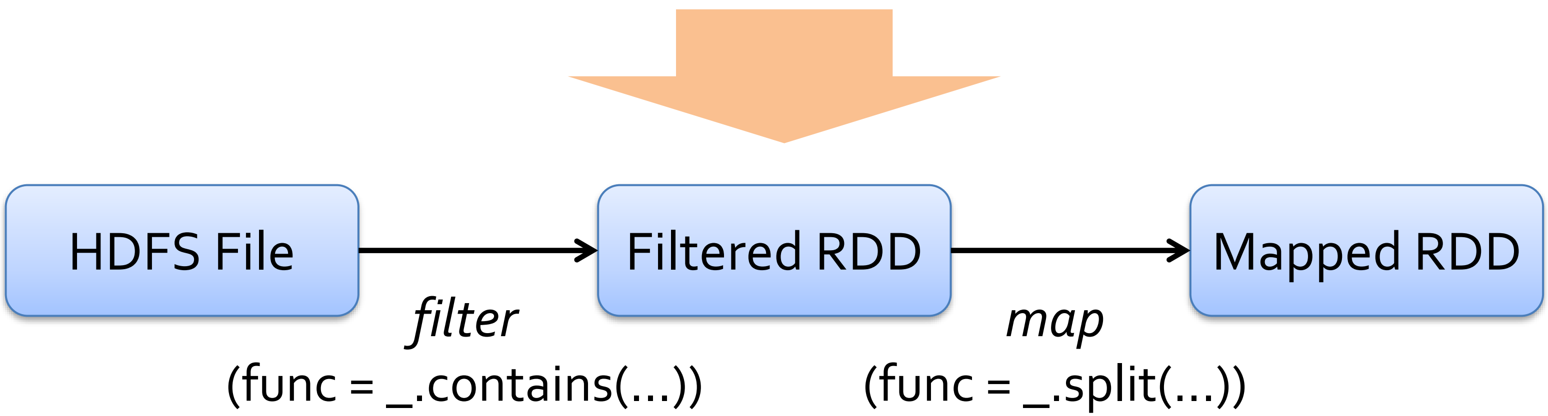


Result: scaled to 1 TB data in 5-7 sec (vs 170 sec for on-disk data)

RDD Fault Tolerance

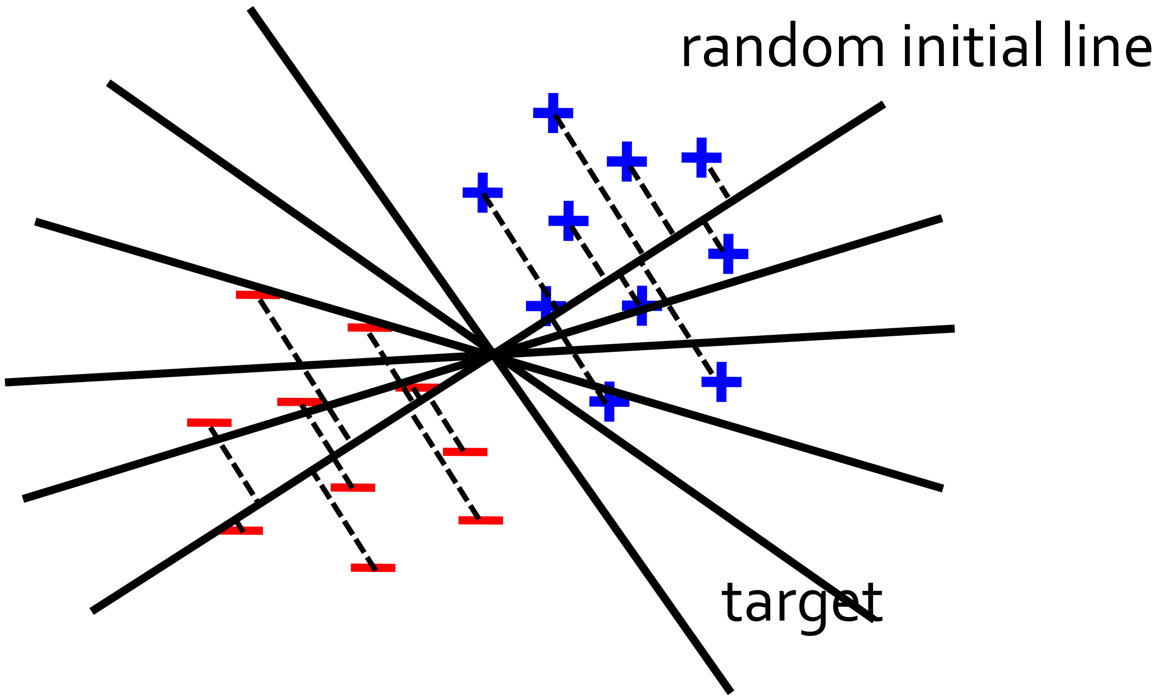
RDDs maintain *lineage* information that can be used to reconstruct lost partitions

Ex:
```
messages = textFile(...).filter(_.startsWith(“ERROR”))
                        .map(_.split(‘\t’)(2))
```



Example: Logistic Regression

Goal: find best line separating two sets of points



Example: Logistic Regression

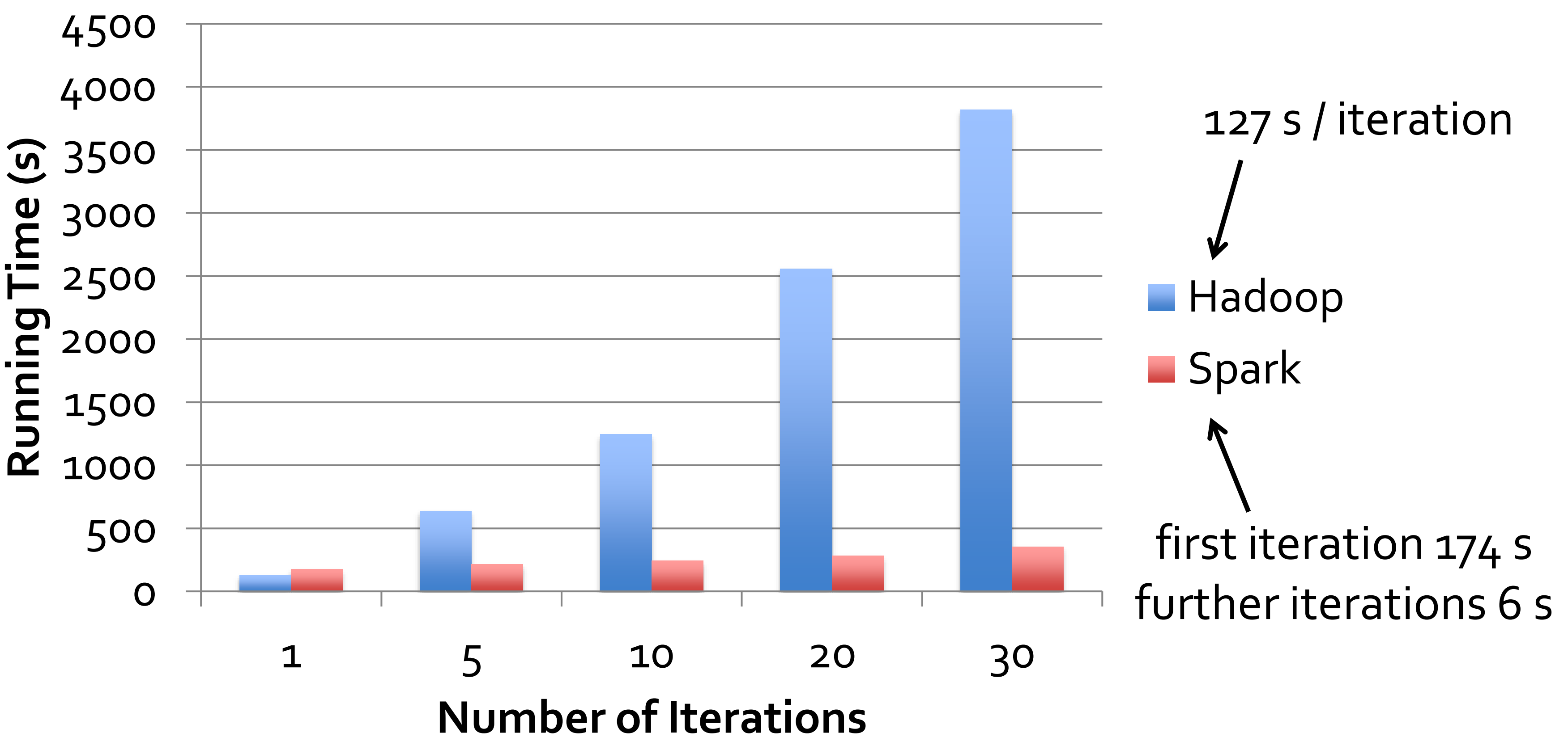
```
val data = spark.textFile(...).map(readPoint).cache()

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)
```

Logistic Regression Performance



Fault-Tolerance Through Lineage

Represent RDD with 5 pieces of information

- A set of partitions
- A set off dependencies on parent partitions
  - Distinguishes between **narrow** (one-to-one)
  - And **wide** dependencies (one-to-many)

- Function to compute dataset based on parent
- Metadata about partitioning scheme and data placement

RDD = Distributed relation + lineage

Query Execution Details

- Lazy evaluation
  - RDDs are not evaluated until an action is called

- In memory caching
  - Spark workers are long-lived processes
  - RDDs can be materialized in memory in workers
  - Base data is not cached in memory

Spark Operations

Transformations (define a new RDD)	map filter sample groupByKey reduceByKey sortByKey	flatMap union join cogroup cross mapValues
Actions (return a result to driver program)	collect reduce count save lookupKey	

Spark Applications

In-memory data mining on Hive data (Conviva)

Predictive analytics (Quantifind)

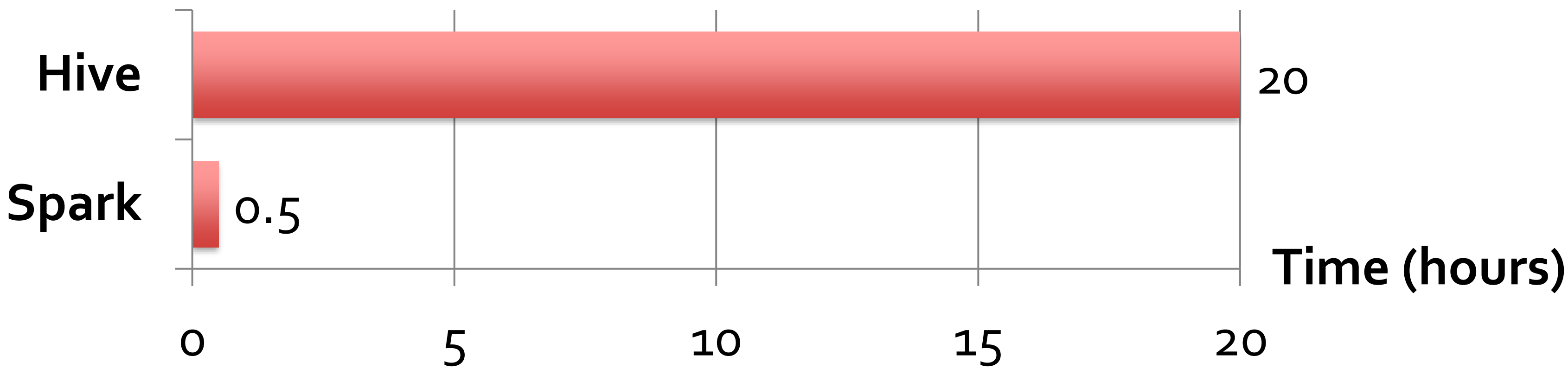
City traffic prediction (Mobile Millennium)

Twitter spam classification (Monarch)

Collaborative filtering via matrix factorization

…

Conviva GeoReport



Aggregations on many keys w/ same WHERE clause

40× gain comes from:

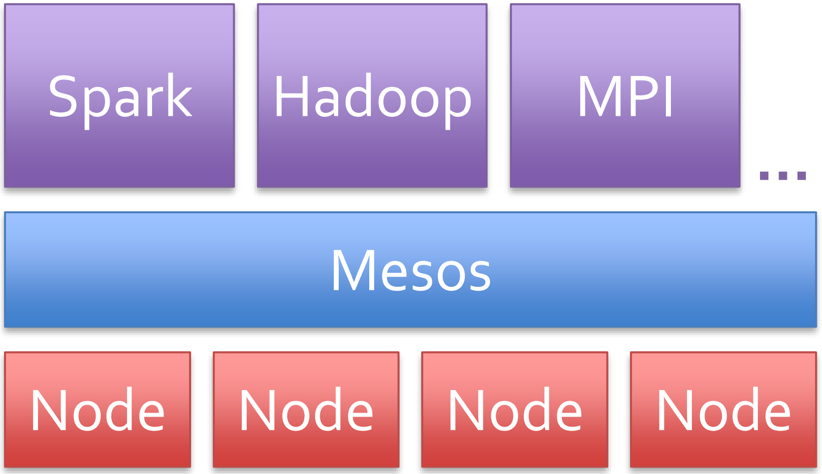
» Not re-reading unused columns or filtered records
» Avoiding repeated decompression
» In-memory storage of deserialized objects

Implementation

Runs on Apache Mesos to share resources with Hadoop & other apps

Can read from any Hadoop input source (e.g. HDFS)

No changes to Scala compiler



Spark Scheduler

Dataflow as DAGs

Pipelines functions within a stage

Cache-aware work reuse & locality

Partitioning-aware to avoid shuffles