

# Spark and Shark

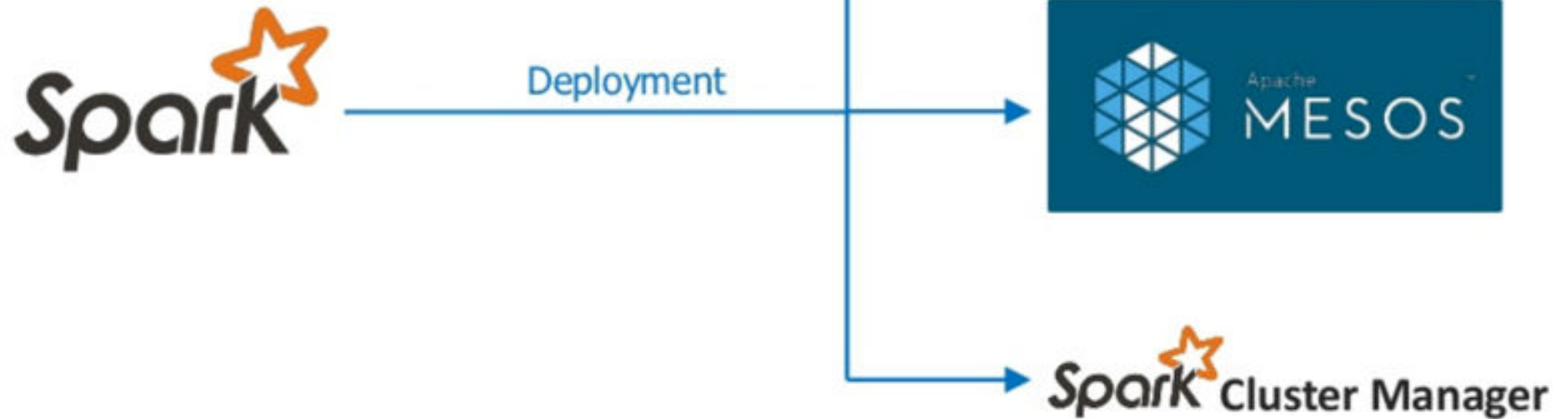
Apache Spark is a general-purpose cluster in-memory computing system

Provides high-level APIs in Java, Scala and Python, and an optimized engine that supports general execution graphs

Provides various high level tools like Spark SQL for structured data processing, Mlib for Machine Learning and more..



The Spark framework can be deployed through Apache Mesos, Apache Hadoop via Yarn, or Spark's own cluster manager.



Spark framework is **polyglot** – Can be programmed in several programming languages (Currently Scala, Java and Python supported).



Polyglot





A fully **Apache Hive** compatible data warehousing system that can run **100x faster** than Hive.

**Spark**  100x faster than




for certain applications.

# What is Spark?

- Not a modified version of Hadoop
- Separate, fast, MapReduce-like engine
  - ▣ In-memory data storage for very fast iterative queries
  - ▣ General execution graphs and powerful optimizations
  - ▣ Up to 40x faster than Hadoop
- Compatible with Hadoop's storage APIs
  - ▣ Can read/write to any Hadoop-supported system, including HDFS, HBase, SequenceFiles, etc

# What is Shark?

- 
- Port of Apache Hive to run on Spark
  - Compatible with existing Hive data, metastores, and queries (HiveQL, UDFs, etc)
  - Similar speedups of up to 40x

# Project History




- Spark project started in 2009, open sourced 2010
- Shark started summer 2011, alpha April 2012
- In use at Berkeley, Princeton, Klout, Foursquare, Conviva, Quantifind, Yahoo! Research & others
- 200+ member meetup, 500+ watchers on GitHub

# Why a New Programming Model?

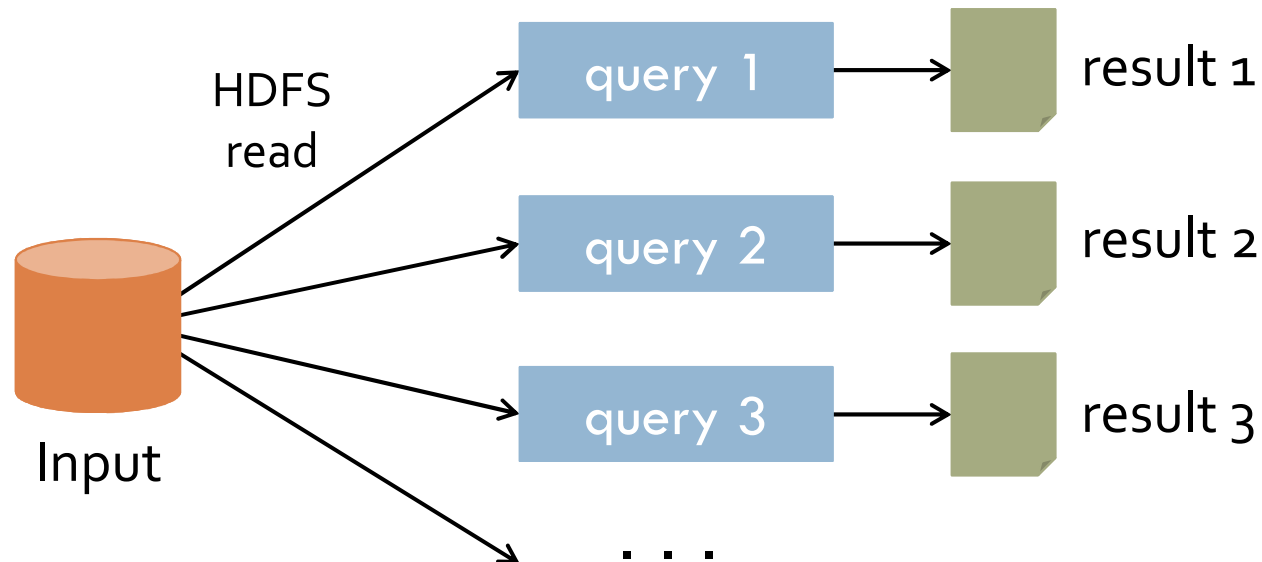
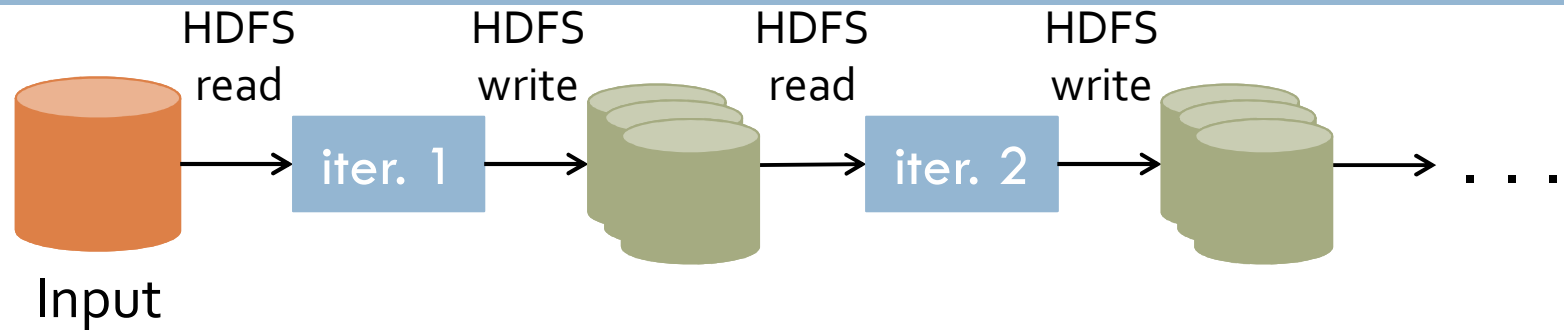
- MapReduce greatly simplified big data analysis
- But as soon as it got popular, users wanted more:
  - ▣ More **complex**, multi-stage applications (e.g. iterative graph algorithms and machine learning)
  - ▣ More **interactive** ad-hoc queries
- Both multi-stage and interactive apps require faster **data sharing** across parallel jobs



# Why Spark?

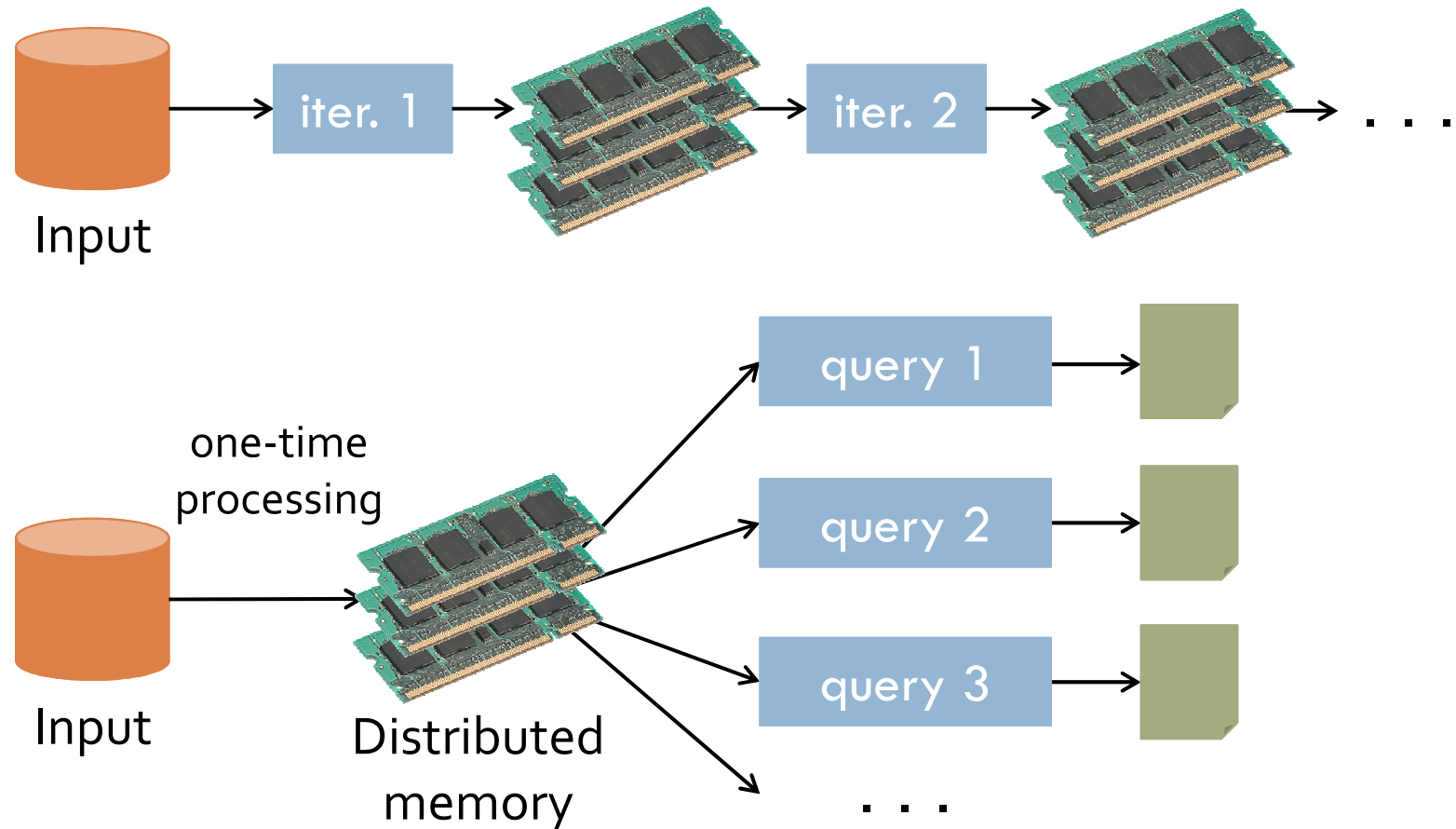
- 
- Provides powerful caching and disk persistence capabilities
  - Interactive Data Analysis
  - Faster Batch
  - Iterative Algorithms
  - Real-Time Stream Processing
  - Faster Decision-Making

# Data Sharing in MapReduce



**Slow** due to replication, serialization, and disk IO

# Data Sharing in Spark



**10-100×** faster than network and disk

# Spark Programming Model



- Key idea: *resilient distributed datasets (RDDs)*
  - ▣ Distributed collections of objects that can be cached in memory across cluster nodes
  - ▣ Manipulated through various parallel operators
  - ▣ Automatically rebuilt on failure
- Interface
  - ▣ Clean language-integrated API in Scala
  - ▣ Can be used *interactively* from Scala console

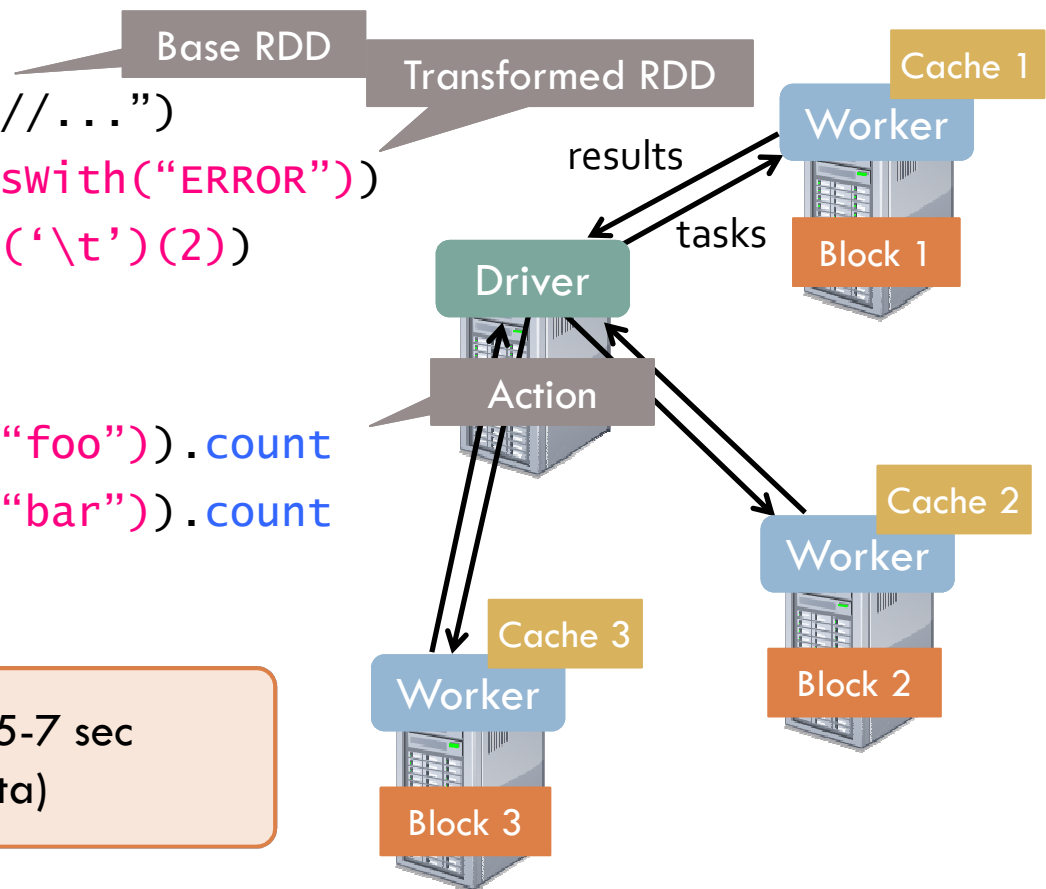
# Example: Log Mining

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)

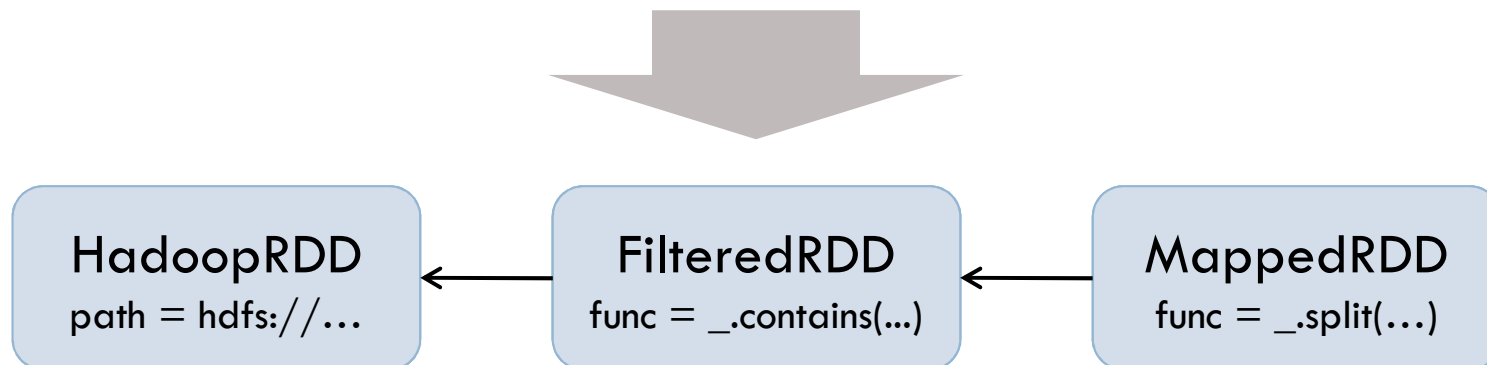


# Fault Tolerance

RDDs track the series of transformations used to build them (their *lineage*) to recompute lost data

E.g:

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



# Example: Logistic Regression

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

Load data in memory once

```
var w = Vector.random(D)
```

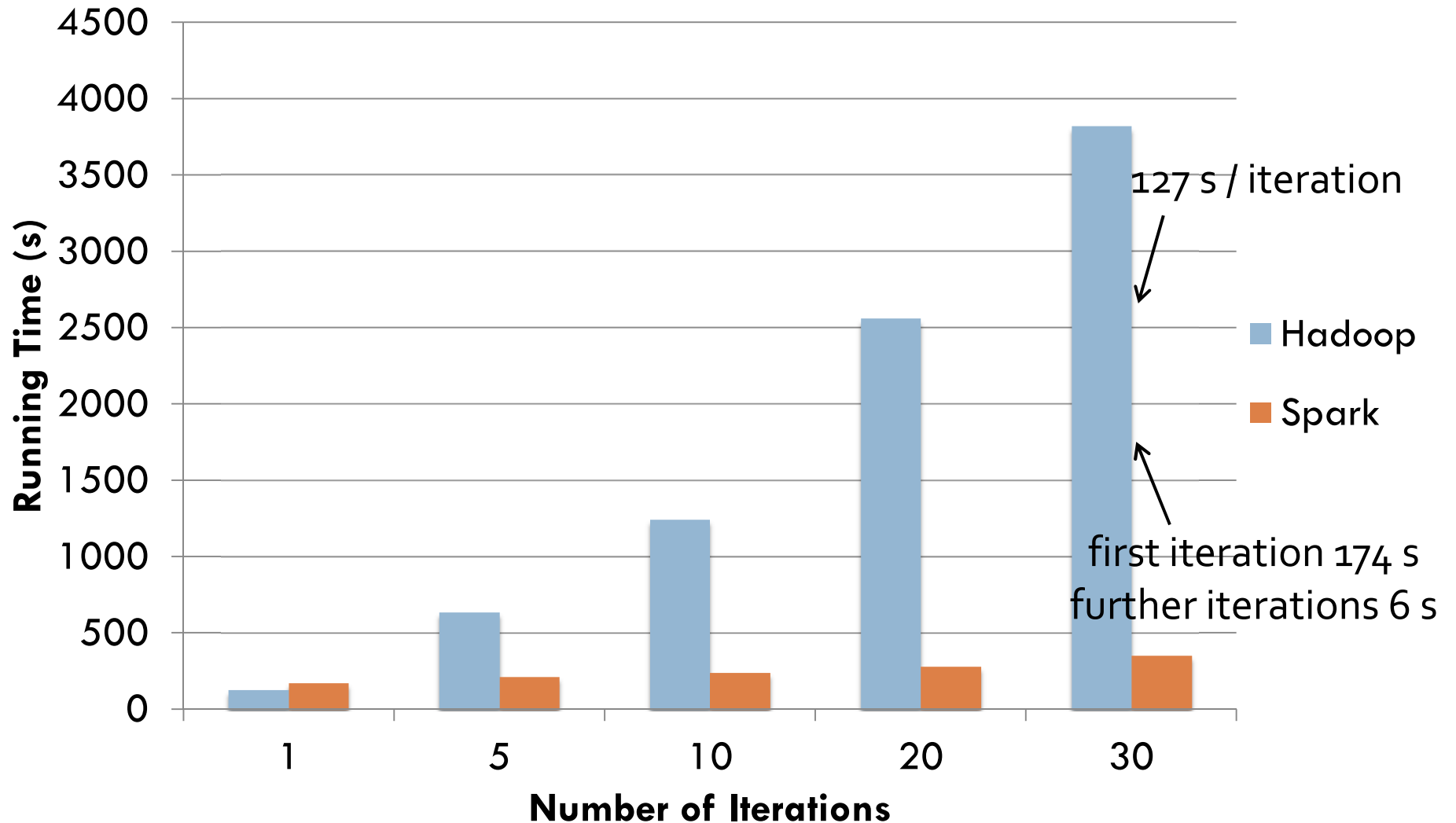
Initial parameter vector

```
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  
}
```

Repeated MapReduce steps  
to do gradient descent

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

# Logistic Regression Performance



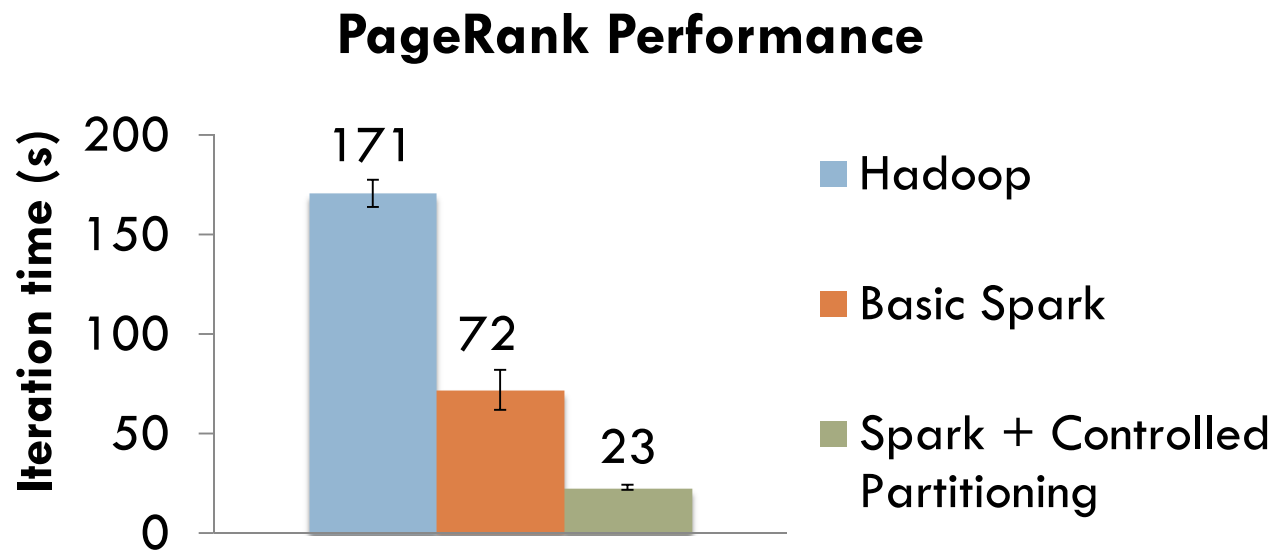


# Supported Operators

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

# Other Engine Features

- General graphs of operators (e.g. map-reduce-reduce)
- Hash-based reduces (faster than Hadoop's sort)
- Controlled data partitioning to lower communication



# Spark Users



CONVIVA®

foursquare

quantifind

KLOUT

YAHOO!  
RESEARCH

University of California  
Berkeley



PRINCETON  
UNIVERSITY

UCSF

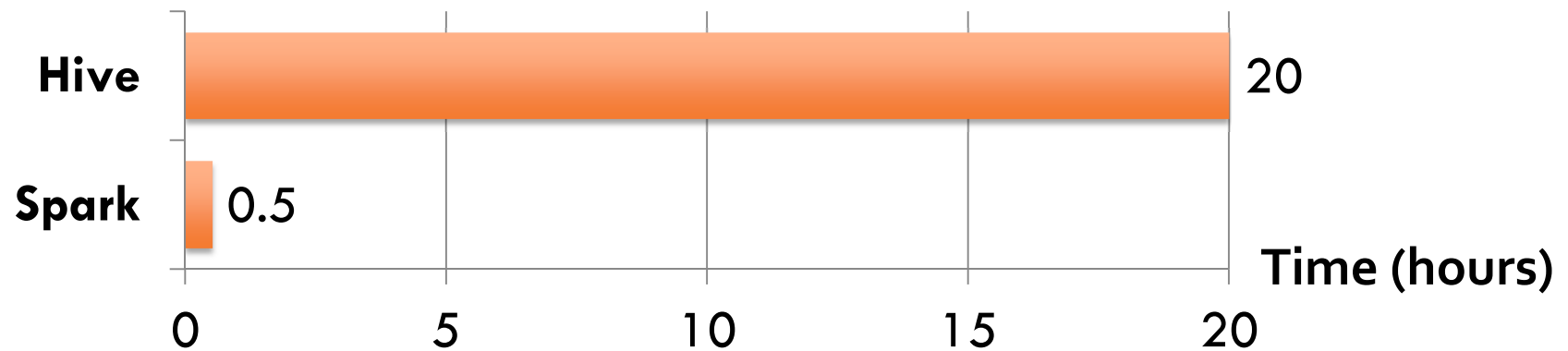
# User Applications



- In-memory analytics & anomaly detection (Conviva)
- Interactive queries on data streams (Quantifind)
- Exploratory log analysis (Foursquare)
- Traffic estimation w/ GPS data (Mobile Millennium)
- Twitter spam classification (Monarch)

...

# Conviva GeoReport



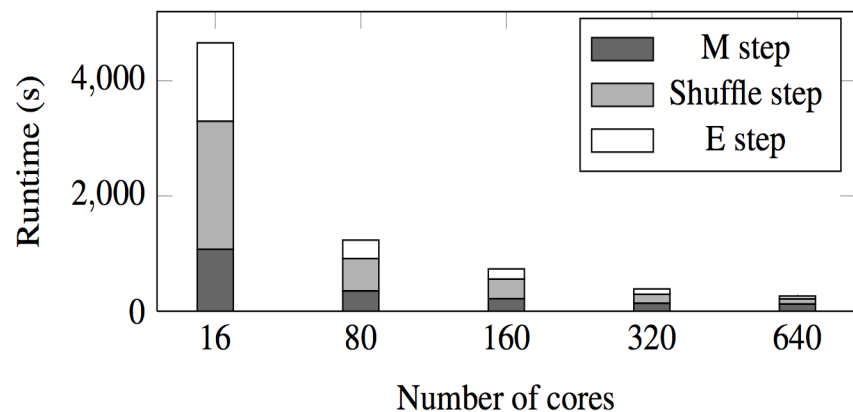
- Group aggregations on many keys w/ same filter
- 40× gain over Hive from avoiding repeated reading, deserialization and filtering

# Mobile Millennium Project

- Estimate city traffic from crowdsourced GPS data



Iterative EM algorithm  
scaling to 160 nodes



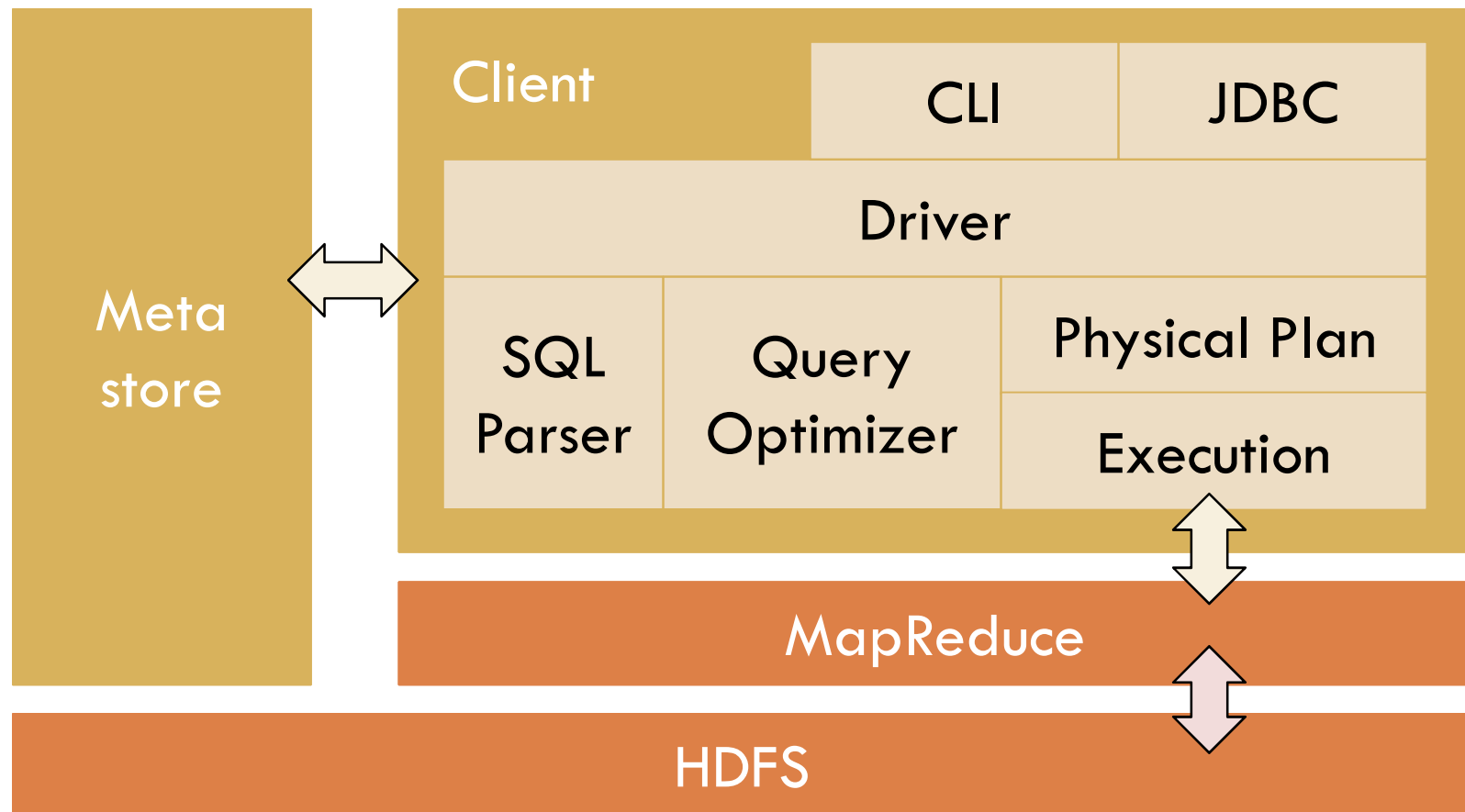
Credit: Tim Hunter, with support of the Mobile Millennium team; P.I. Alex Bayen; [traffic.berkeley.edu](http://traffic.berkeley.edu)

# Shark: Hive on Spark - Motivation



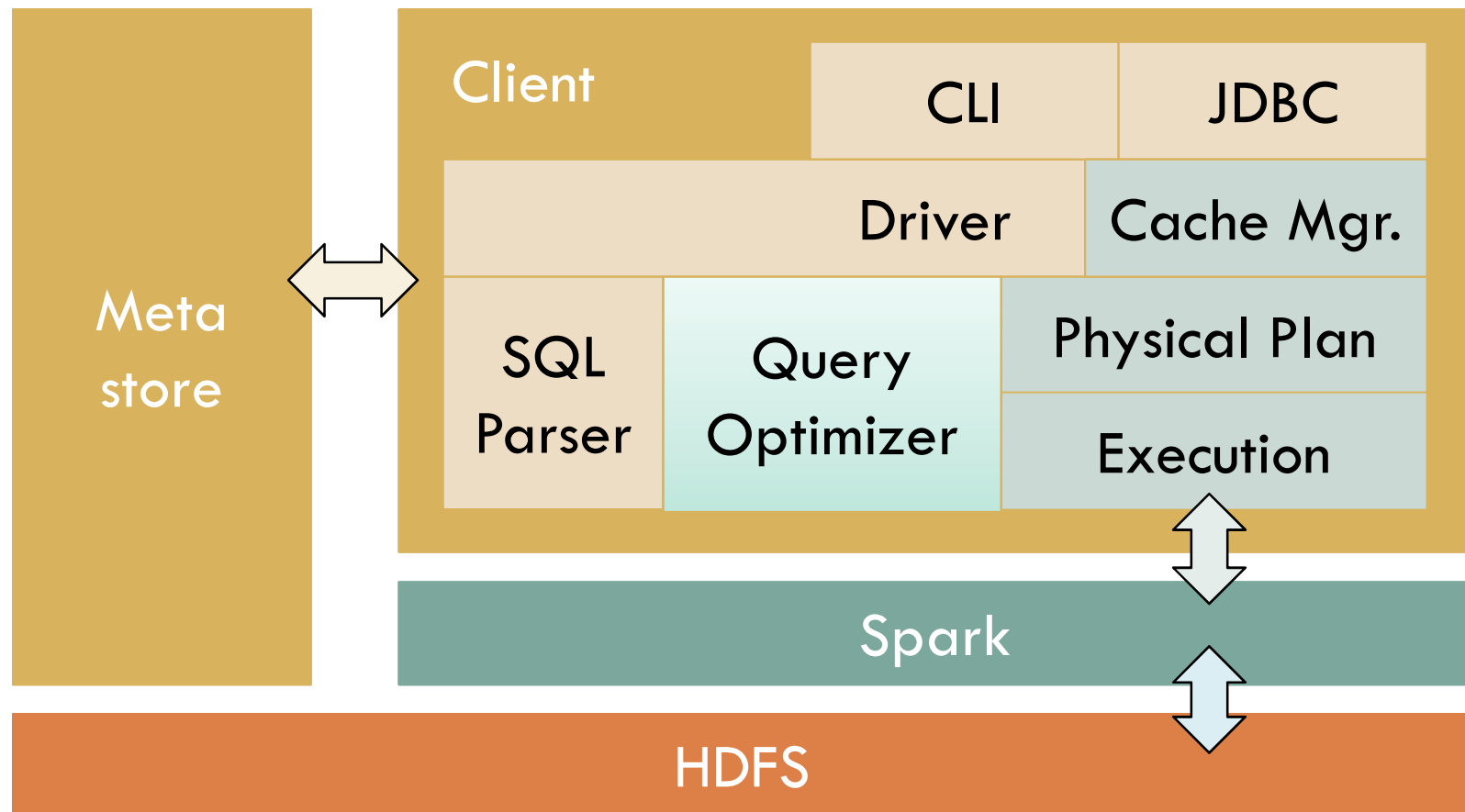
- Hive is great, but Hadoop's execution engine makes even the smallest queries take minutes
- Scala is good for programmers, but many data users only know SQL
- **Can we extend Hive to run on Spark?**

# Hive Architecture





# Shark Architecture



# Efficient In-Memory Storage

- Simply caching Hive records as Java objects is inefficient due to high per-object overhead
- Instead, Shark employs column-oriented storage using **arrays of primitive types**

**Row Storage**

1	john	4.1
2	mike	3.5
3	sally	6.4

**Column Storage**

1	2	3
john	mike	sally
4.1	3.5	6.4

# Efficient In-Memory Storage

- Simply caching Hive records as Java objects is inefficient due to high per-object overhead
- Instead, Shark employs column-oriented storage using **arrays of primitive types**

Row Storage

Column Storage

**Benefit:** similarly compact size to serialized data,  
but  $>5\times$  faster to access



# Using Shark

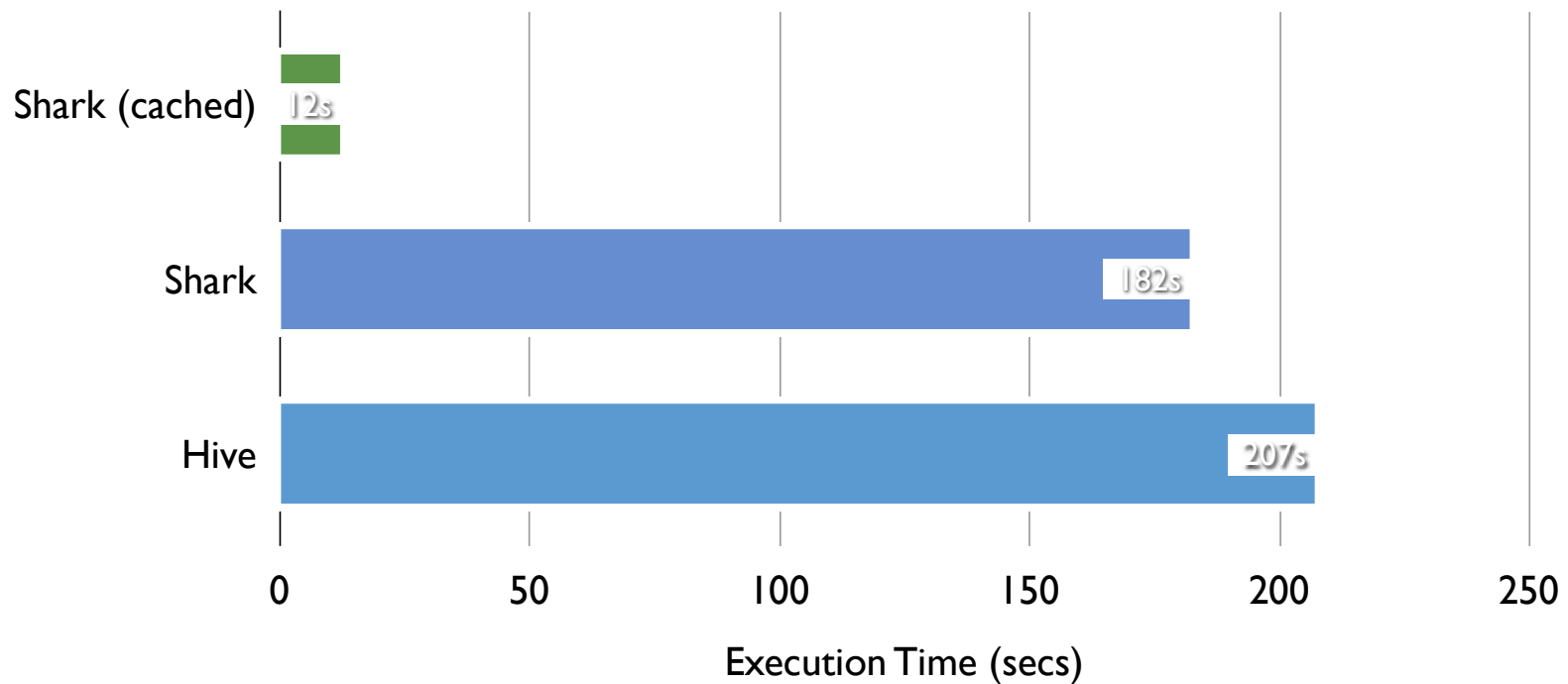


- `CREATE TABLE mydata_cached AS SELECT ...`
- Run standard HiveQL on it, including UDFs
  - ▣ A few esoteric features are not yet supported
- Can also call from Scala to mix with Spark

# Benchmark Query 1

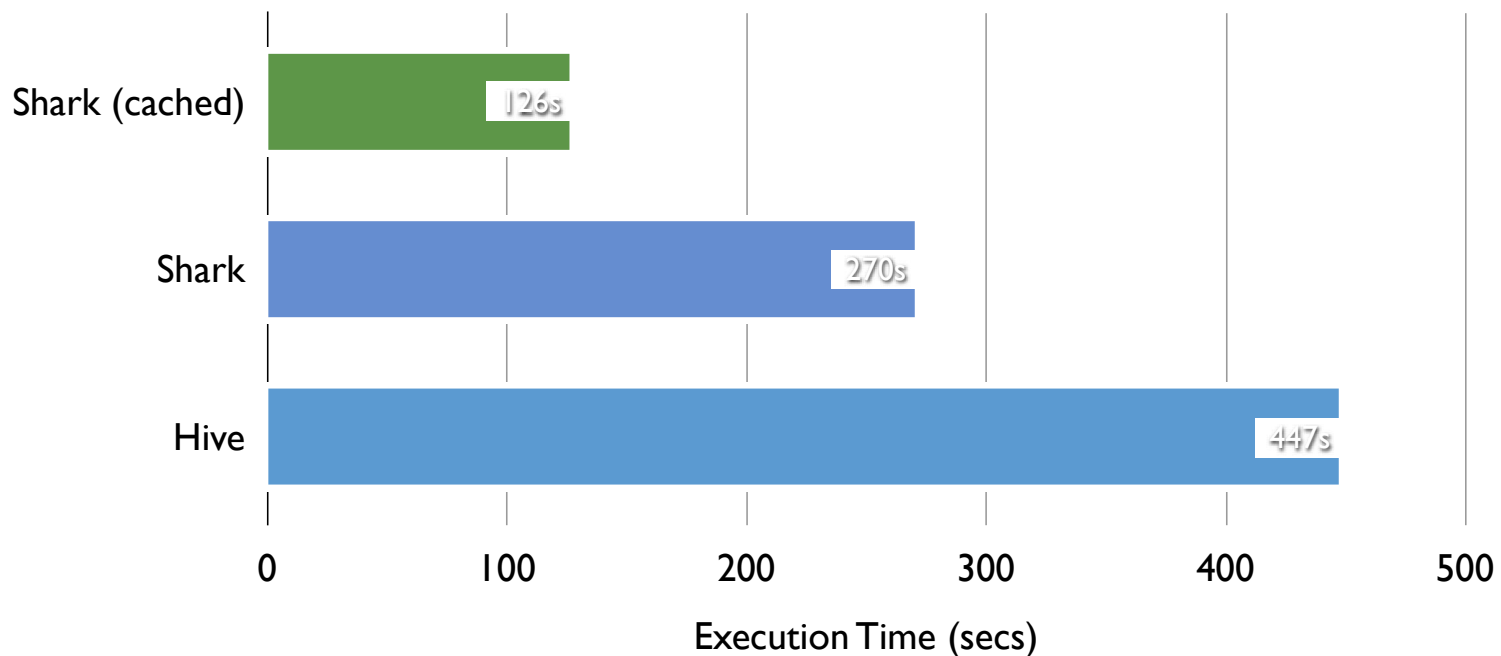


```
SELECT * FROM grep WHERE field LIKE '%XYZ%';
```



# Benchmark Query 2

- SELECT sourceIP, AVG(pageRank), SUM(adRevenue) AS earnings  
FROM rankings AS R, userVisits AS V ON R.pageURL = V.destURL  
WHERE V.visitDate BETWEEN '1999-01-01' AND '2000-01-01'  
GROUP BY V.sourceIP  
ORDER BY earnings DESC  
LIMIT 1;



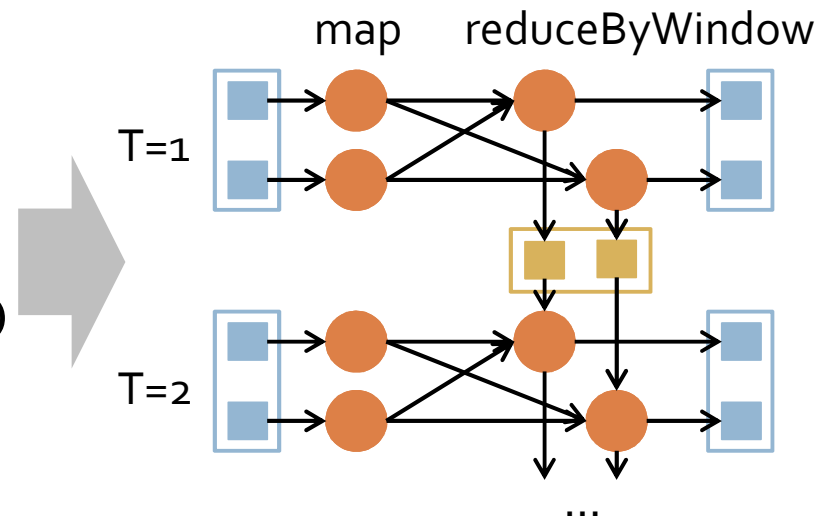
# What's Next?

- Recall that Spark's model was motivated by two emerging uses (interactive and multi-stage apps)
- Another emerging use case that needs fast data sharing is **stream processing**
  - ▣ Track and update state in memory as events arrive
  - ▣ Large-scale reporting, click analysis, spam filtering, etc

# Streaming Spark

- Extends Spark to perform streaming computations
- Runs as a series of small (~1 s) batch jobs, keeping state in memory as fault-tolerant RDDs
- Intermix seamlessly with batch and ad-hoc queries

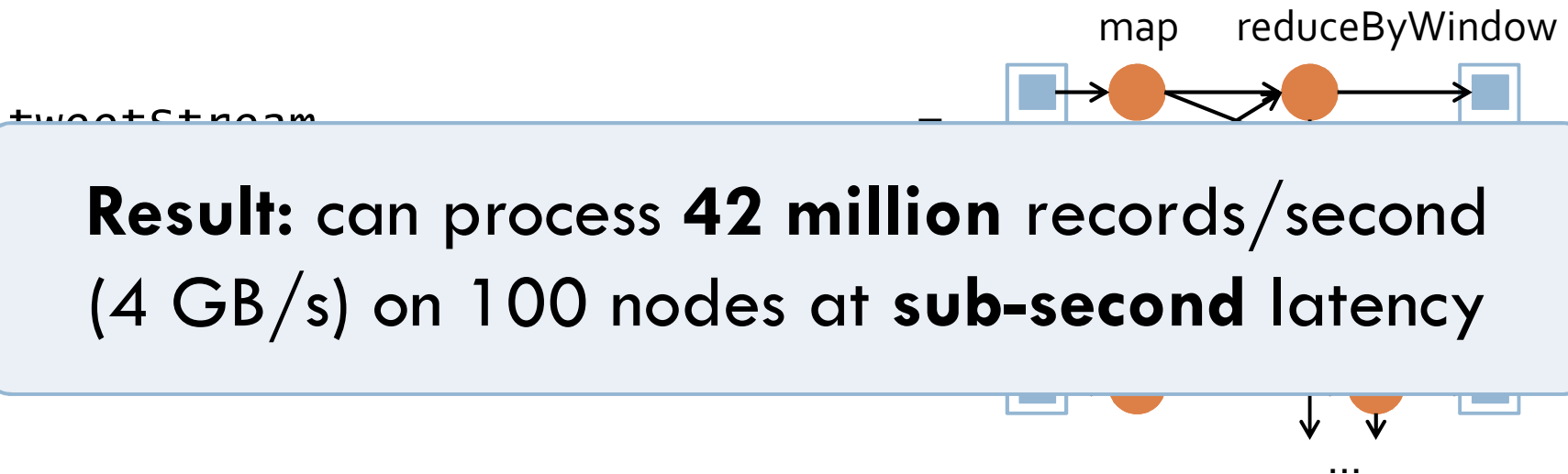
```
tweetStream  
  .flatMap(_.toLowerCase.split)  
  .map(word => (word, 1))  
  .reduceByWindow("5s", _ + _)
```





# Streaming Spark

- ❑ Extends Spark to perform streaming computations
- ❑ Runs as a series of small (~1 s) batch jobs, keeping state in memory as fault-tolerant RDDs
- ❑ Intermix seamlessly with batch and ad-hoc queries

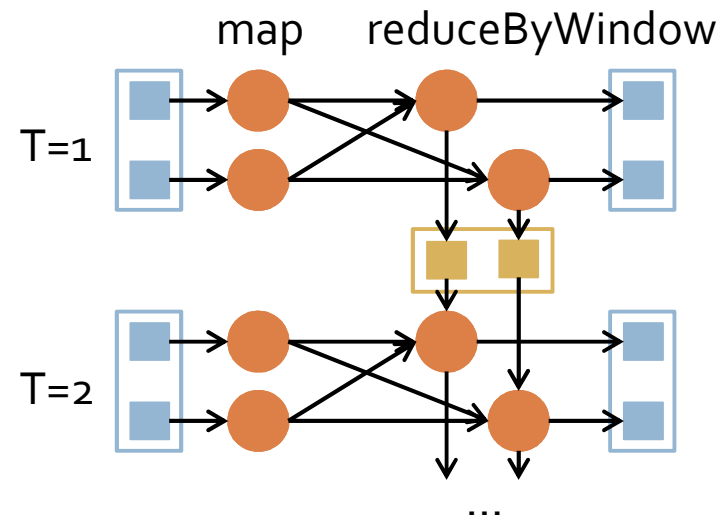


**Result:** can process **42 million** records/second  
(4 GB/s) on 100 nodes at **sub-second** latency

# Streaming Spark

- Extends Spark to perform streaming computations
- Runs as a series of small (~1 s) batch jobs, keeping state in memory as fault-tolerant RDDs
- Intermix seamlessly with batch and ad-hoc queries

```
tweetStream  
  .flatMap(_.toLowerCase.split)  
  .map(word => (word, 1))  
  .reduceByWindow(5, _ + _)
```

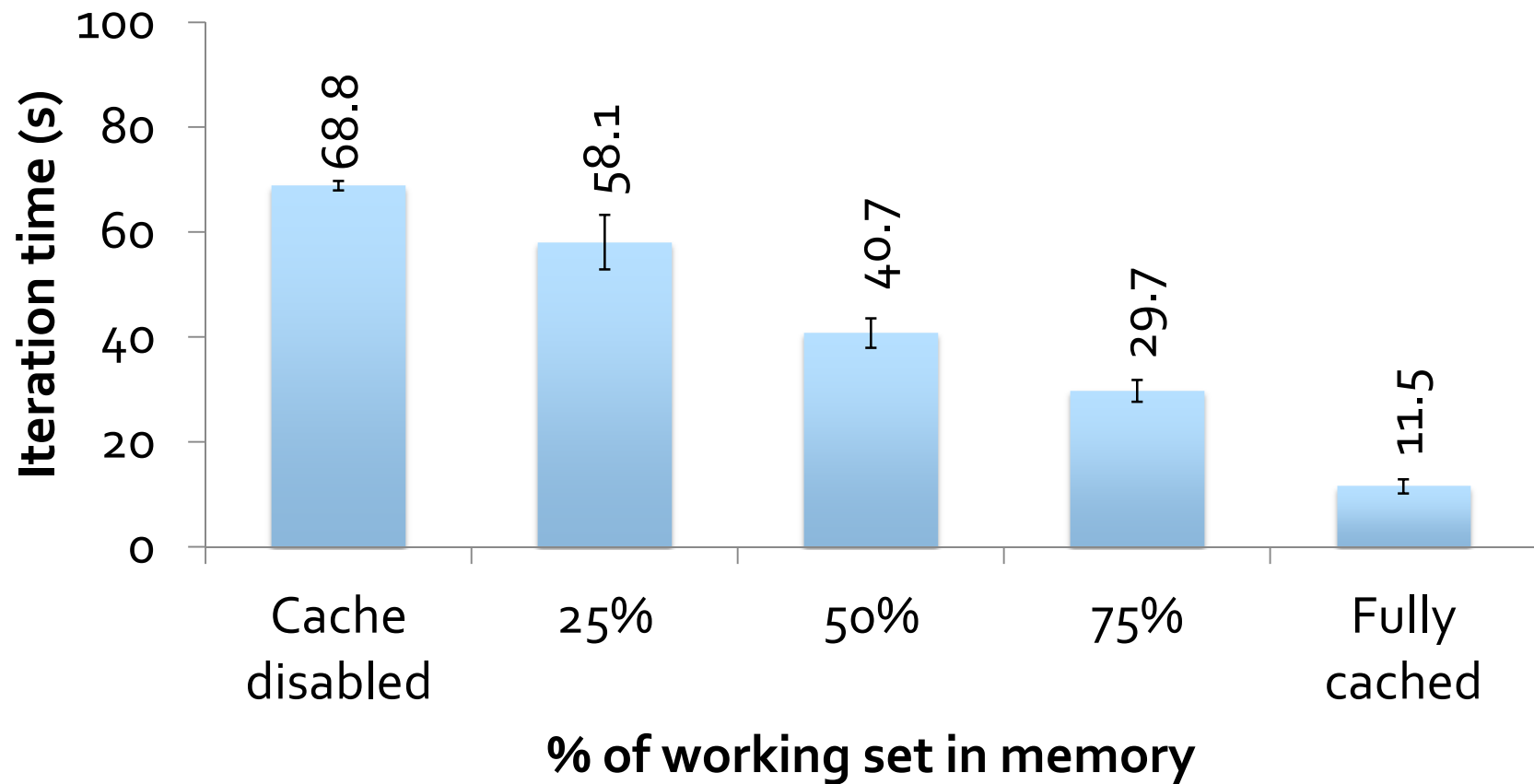


# Conclusion



- Spark and Shark speed up your interactive and complex analytics on Hadoop data
- Download and docs: [www.spark-project.org](http://www.spark-project.org)
  - ▣ Easy to run locally, on EC2, or on Mesos and soon YARN

# Behavior with Not Enough RAM



# Spark Ecosystem

