Spark: Making Big Data Interactive & Real-Time

Matei Zaharia

UC Berkeley / MIT

www.spark-project.org

What is Spark?

Fast and expressive cluster computing system compatible with Apache Hadoop

Improves efficiency through:

- » General execution graphs
- » In-memory storage

→ Up to 10 × faster on disk, 100 × in memory

Improves usability through:

- » Rich APIs in Scala, Java, Python
- » Interactive shell

→ 2-5 × less code

Project History

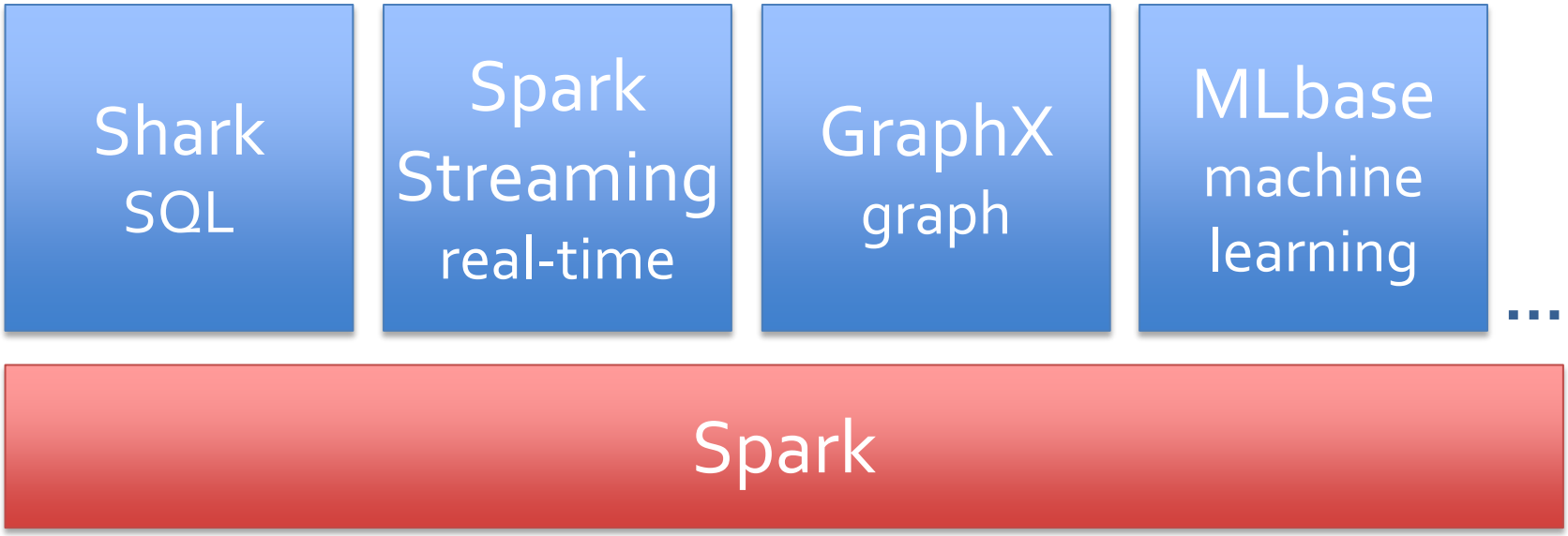
Spark started in 2009, open sourced 2010

In use at Yahoo!, Intel, Adobe, Quantifind, Conviva, Ooyala, Bizo and others

24 companies now contributing code

spark.incubator.apache.org

A Growing Stack



Part of the Berkeley Data Analytics Stack (BDAS) project to build an open source next-gen analytics system

This Talk

Spark introduction & use cases

Shark: SQL on Spark

Spark Streaming

The power of unification

Why a New Programming Model?

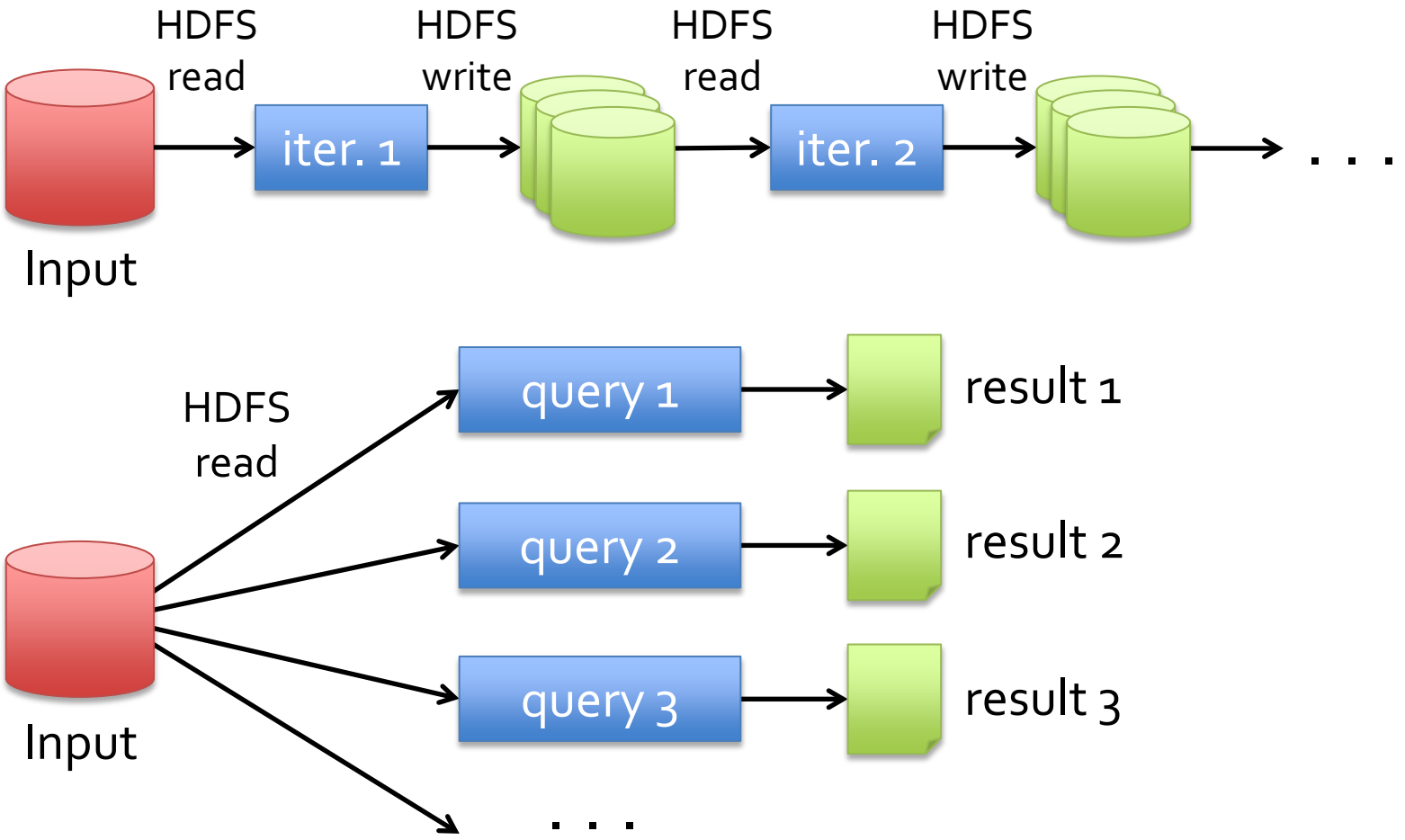
MapReduce greatly simplified big data analysis

But as soon as it got popular, users wanted more:

- More **complex**, multi-pass analytics (e.g. ML, graph)
- More **interactive** ad-hoc queries
- More **real-time** stream processing

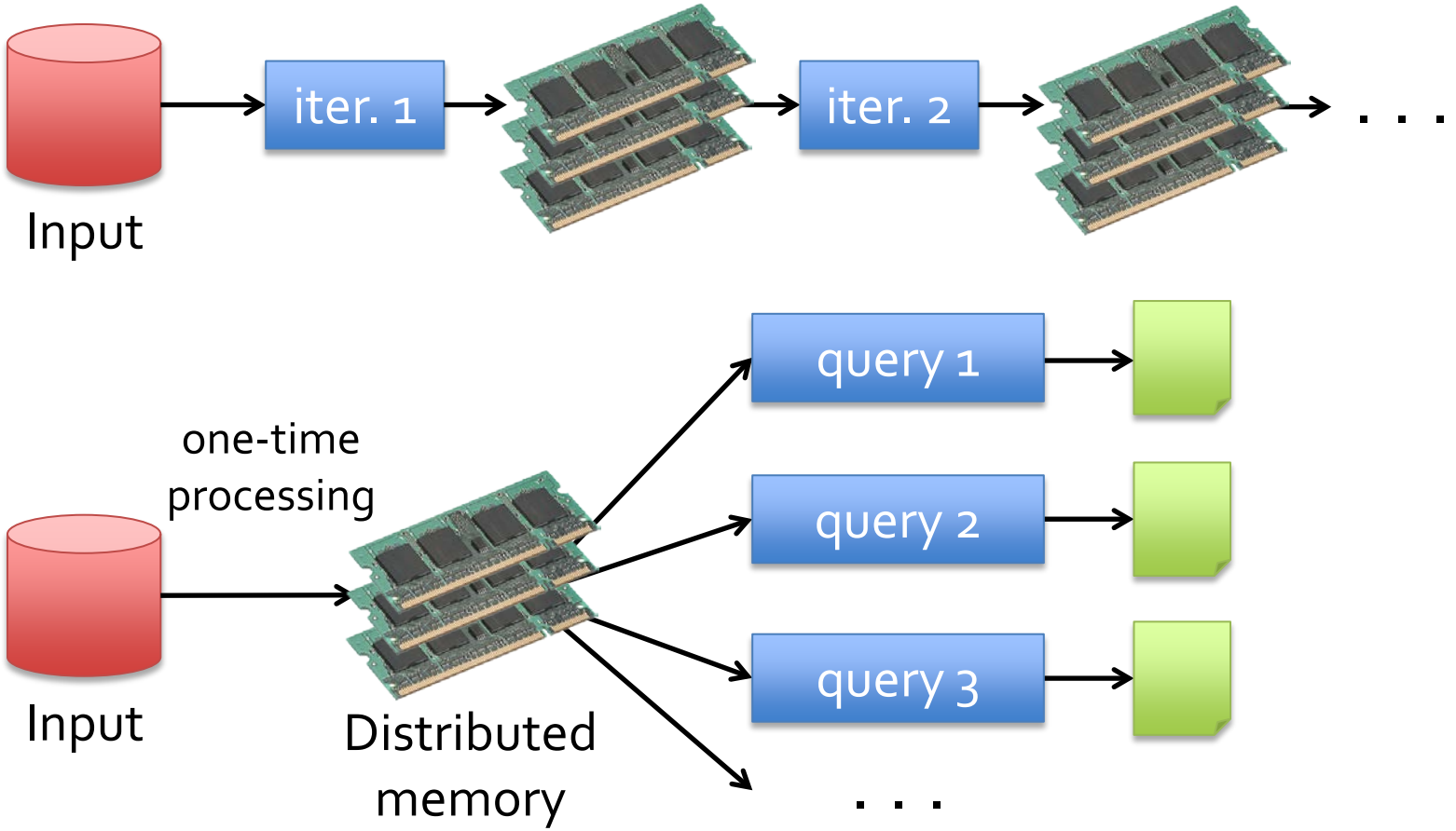
All 3 need faster **data sharing** across parallel jobs

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
- Manipulated through parallel operators
- Automatically recomputed on failure

Programming interface

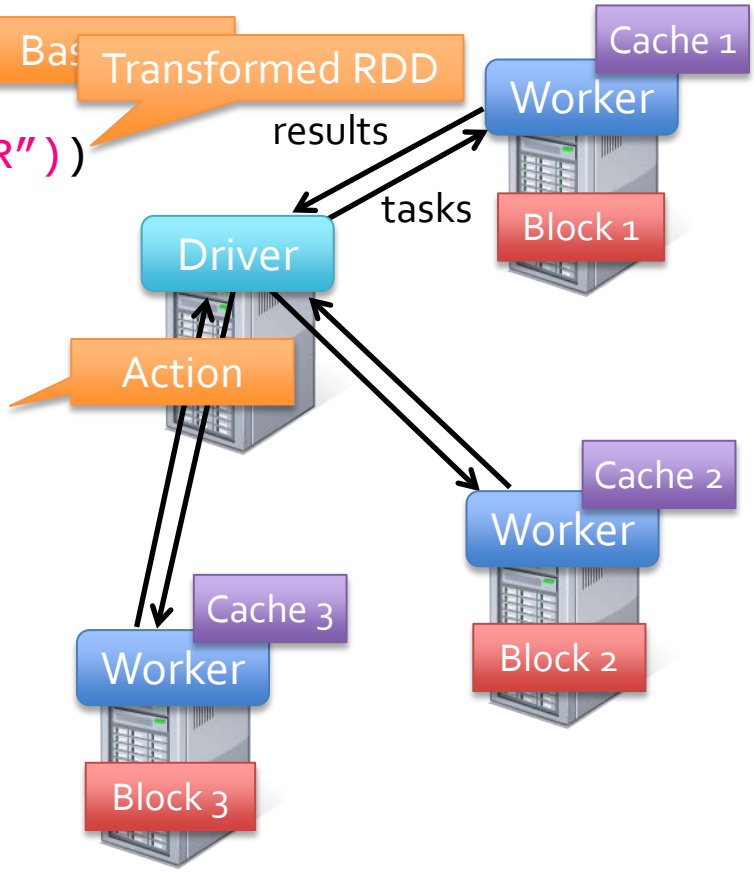
- Functional APIs in Scala, Java, Python
- Interactive use from Scala & Python shells

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
. . .
```

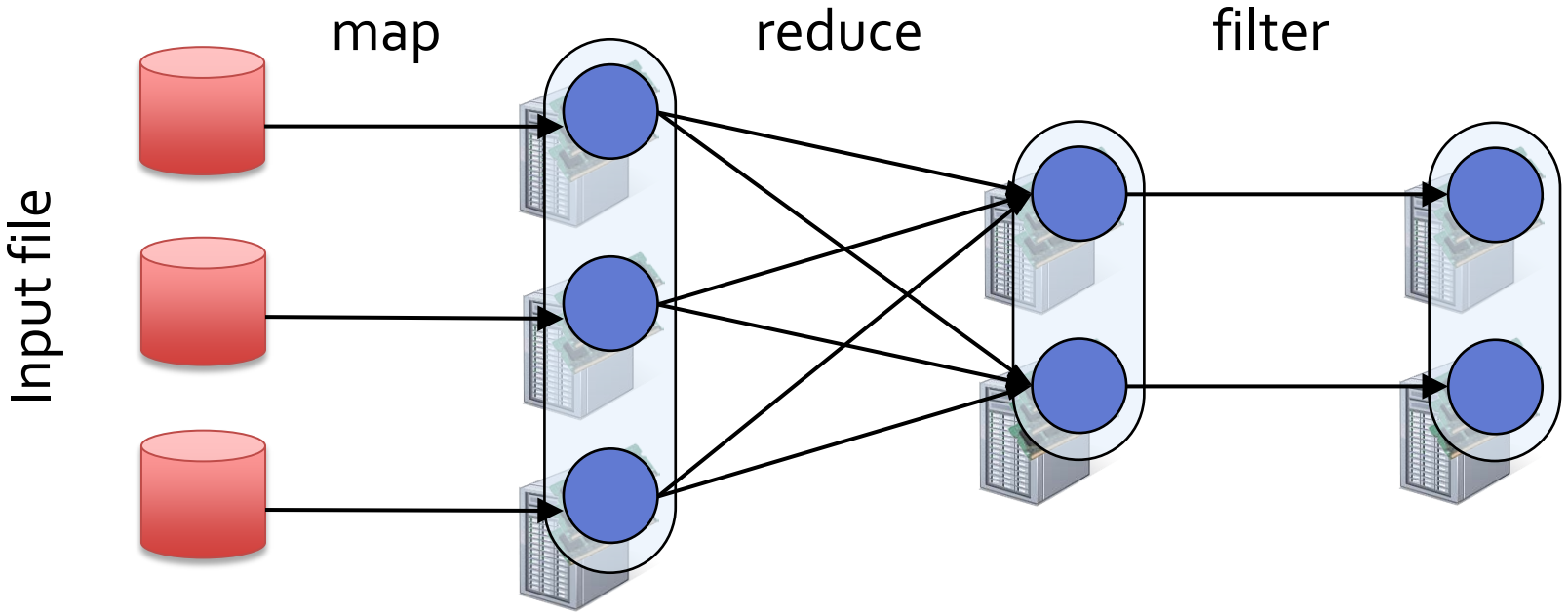


Result: scaled to 1 TB data in 5-7 sec
(vs 170 sec for on-disk data)

Fault Tolerance

RDDs track *lineage* information to rebuild on failure

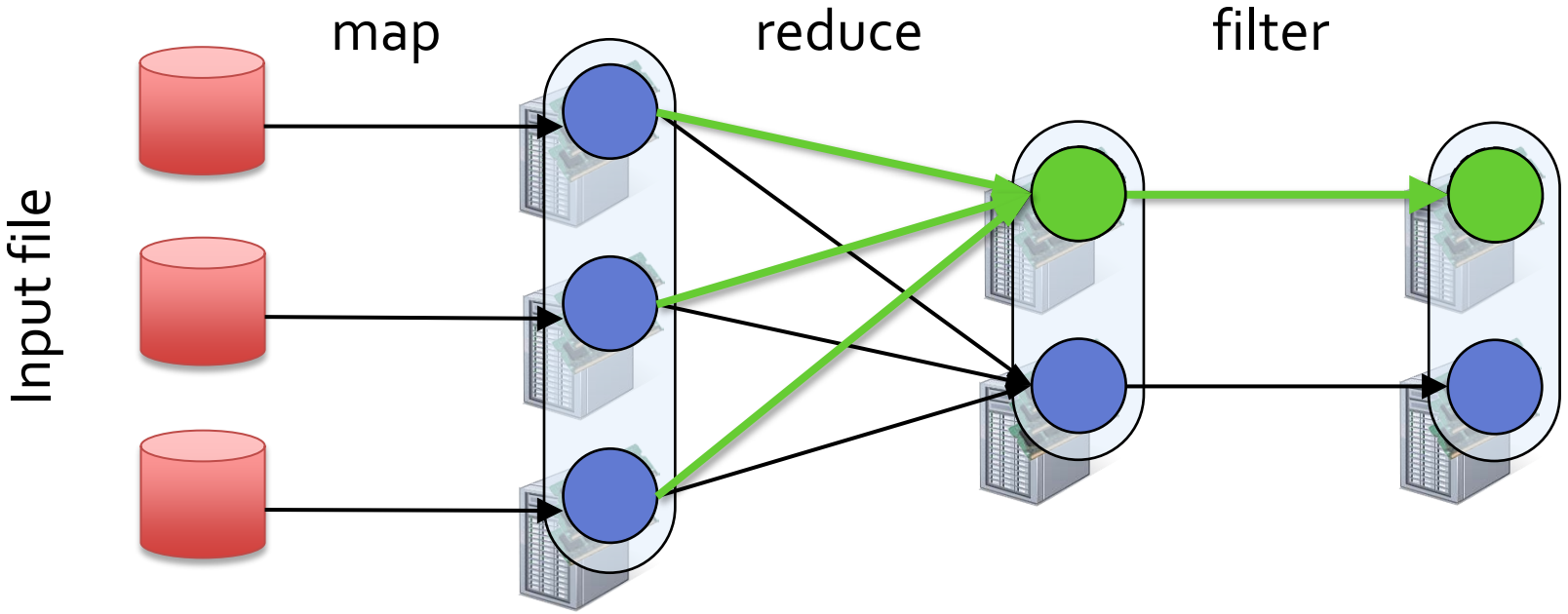
```
file.map(rec => (rec.type, 1))
    .reduceByKey(_ + _)
    .filter((type, count) => count > 10)
```



Fault Tolerance

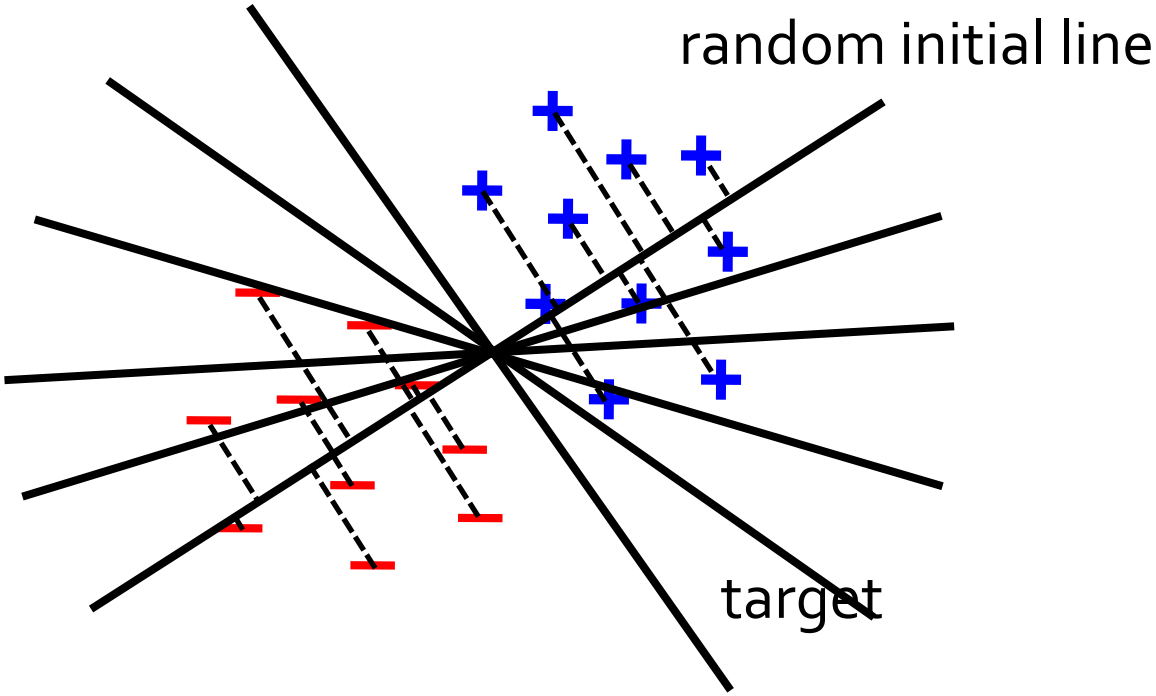
RDDs track *lineage* information to rebuild on failure

```
file.map(rec => (rec.type, 1))
    .reduceByKey(_ + _)
    .filter((type, count) => count > 10)
```



Example: Logistic Regression

Goal: find best line separating two sets of points



Example: Logistic Regression

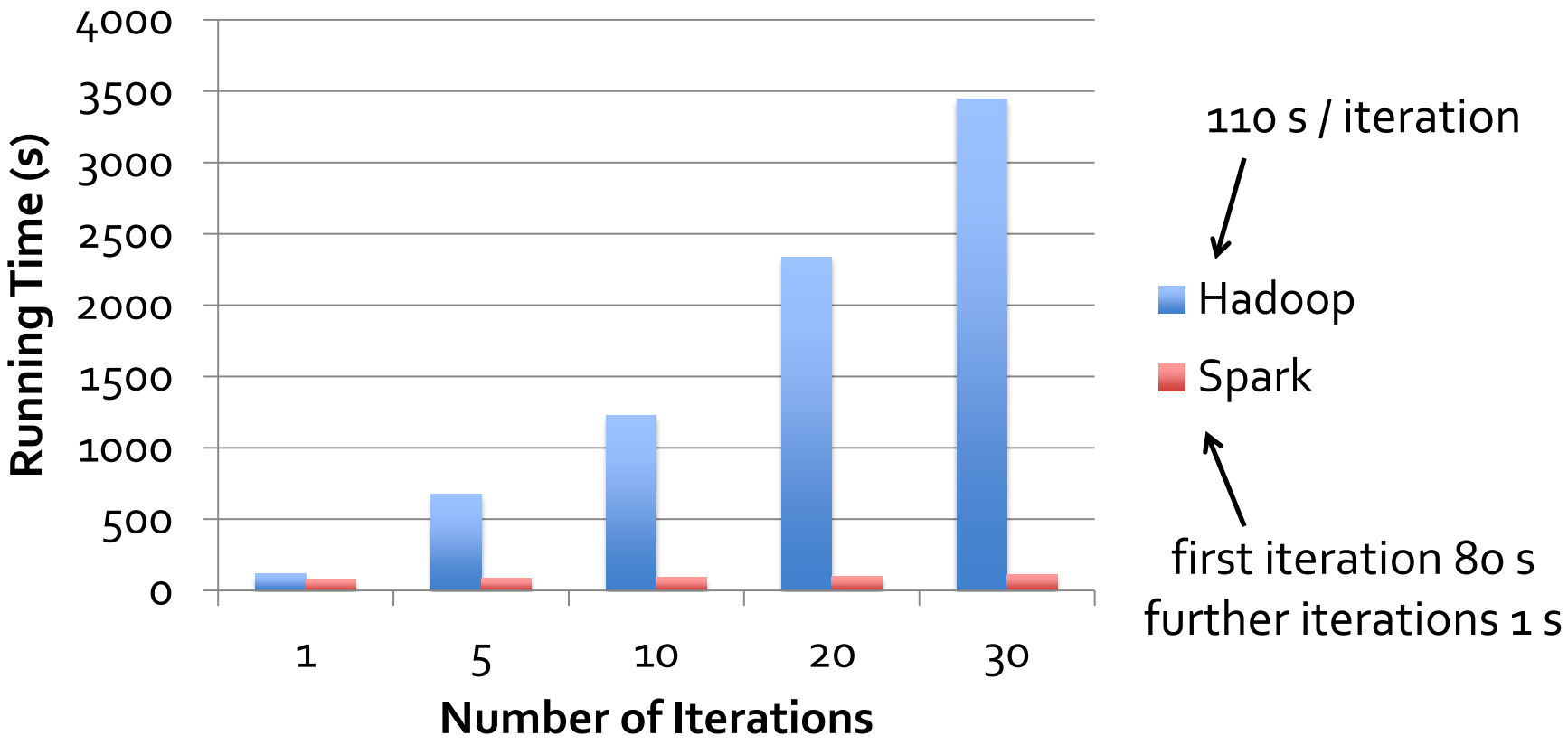
```
val data = spark.textFile(...).map(readPoint).cache()

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



Supported Operators

```
map
filter
groupBy
sort
union
join
leftOuterJoin
rightOuterJoin
```

```
reduce
count
fold
reduceByKey
groupByKey
cogroup
cross
zip
```

```
sample
take
first
partitionBy
mapWith
pipe
save
...
```

Spark in Python and Java

```
// Python:

lines = sc.textFile(...)
lines.filter(lambda x: “ERROR” in x).count()


// Java:

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

User Community

1000+ meetup members

80+ contributors

24 companies contributing

YAHOO! intel Adobe UCSF 阿里巴巴 Alibaba.com

celtra webtrends ClearStory DATA Now You See It AdMobius

CONVIVA bizo TAGGED quantiFind

Generality of RDDs

RDD model was general enough to let us implement several previous models:

- » MapReduce, Dryad
- » Pregel [200 LOC]
- » Iterative MapReduce [200 LOC]
- » GraphLab [600 LOC]
- » SQL (Shark) [6000 LOC]

Allows apps to *intermix* these models

This Talk

Spark introduction & use cases

Shark: SQL over Spark

Spark Streaming

The power of unification

Conventional Wisdom

A Comparison of Approaches to Large-Scale Data Analysis

Andrew Pavlo
Brown University
pavlo@cs.brown.edu

Erik Paulson
University of Wisconsin
epaulson@cs.wisc.edu

Alexander Rasin
Brown University
alexr@cs.brown.edu

Daniel J. Abadi
Yale University
dna@cs.yale.edu

David J. DeWitt
Microsoft Inc.
dewitt@microsoft.com

Samuel Madden
M.I.T. CSAIL
madden@csail.mit.edu

Michael Stonebraker
M.I.T. CSAIL
stonebraker@csail.mit.edu

ABSTRACT

There is currently considerable enthusiasm around the MapReduce (MR) paradigm for large-scale data analysis [17]. Although the basic control flow of this framework has existed in parallel SQL database management systems (DBMS) for over 20 years, some have called MR a dramatically new computing model [8, 17]. In this paper, we describe and compare both paradigms. Furthermore, we evaluate both kinds of systems in terms of performance and de-

model through which users can express relatively sophisticated distributed programs, leading to significant interest in the educational community. For example, IBM and Google have announced plans to make a 1000 processor MapReduce cluster available to teach students distributed programming.

Given this interest in MapReduce, it is natural to ask "Why not use a parallel DBMS instead?" Parallel database systems (which all share a common architectural design) have been commercially

MPP databases 10-100x faster than MapReduce

Why Databases Were Faster

Data representation

- » Schema-aware, column-oriented, etc
- » Compare with MapReduce's "record" model

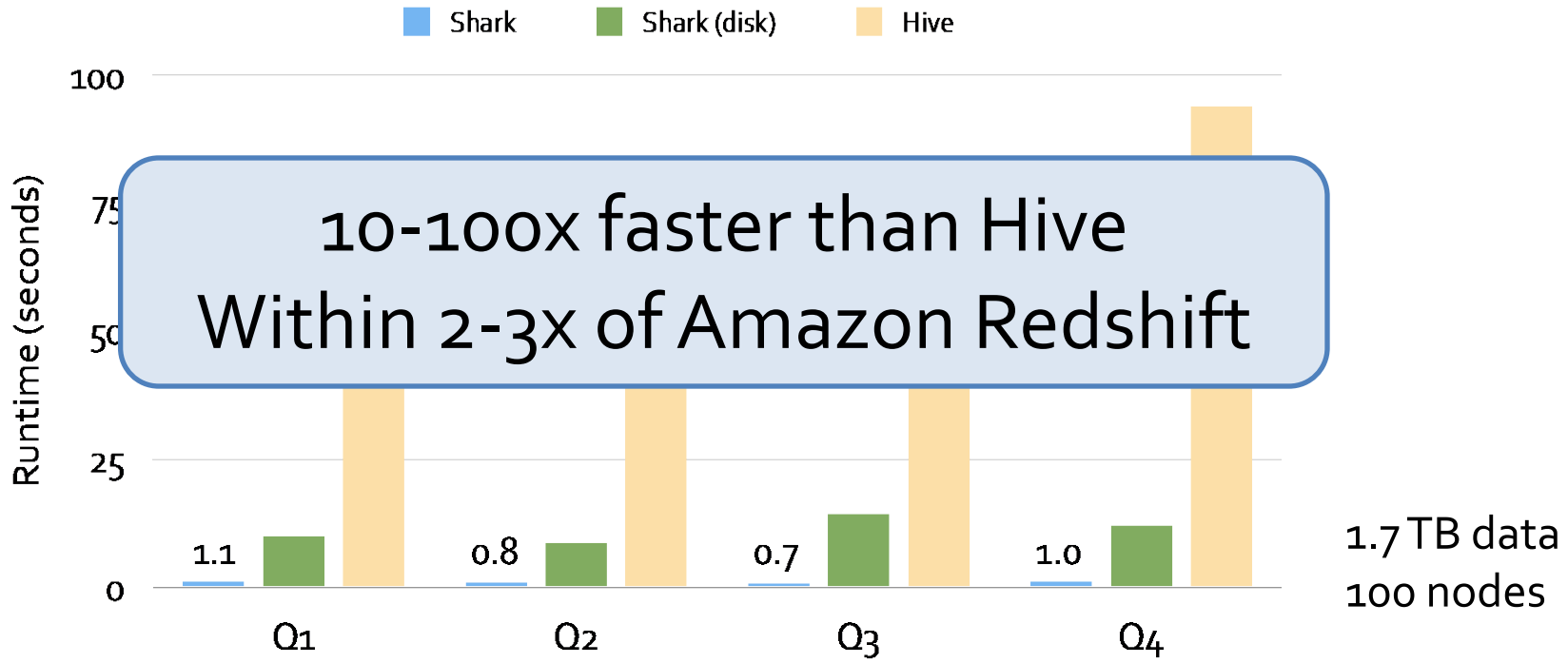
Indexing

Lack of mid-query fault tolerance

- » MR's "pull" model costly compared to "push"

Shark

Provides fast SQL queries on a MapReduce-like engine (Spark), *and* retains full fault tolerance



But What About…

Data representation?

- » Can do it *inside* MapReduce records
- » In Shark, a "record" is a set of rows
- » Implemented columnar storage within these

Indexing?

- » Can still read indices in MR

Mid-query fault tolerance?

- » We have it, but we still perform OK
- » Also enables elasticity, straggler mitigation

Implementation

Port of Apache Hive

» Supports unmodified Hive data and warehouses

Efficient mixing with Spark code (e.g. machine learning functions) in the same system

```
val users = sql2rdd("SELECT * FROM user u
  JOIN comment c ON c.uid=u.uid")

val features = users.mapRows { row =>
  new Vector(extractFeature1(row.getInt("age")),
             extractFeature2(row.getStr("country")),
             ...)}
val trainedVector = logRegress(features.cache())
```

This Talk

Spark introduction & use cases

Shark: SQL over Spark

Spark Streaming

The power of unification

Motivation

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

- Site statistics, intrusion detection, online ML

To scale these apps to 100s of nodes, require:

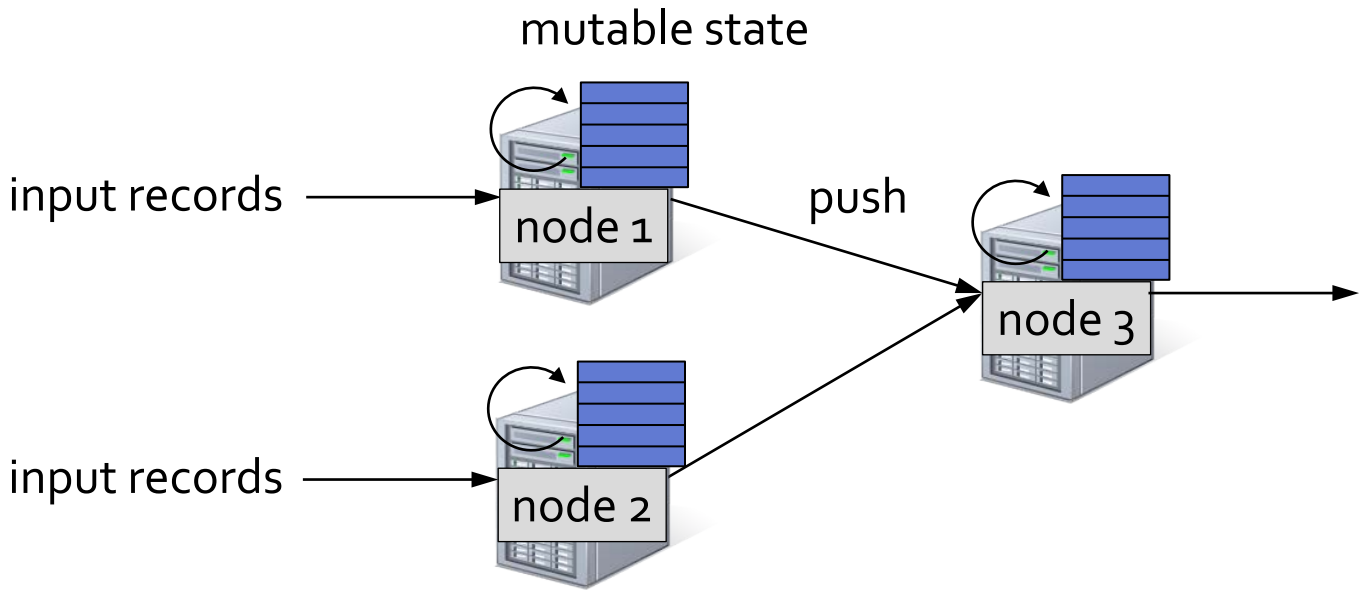
- **Fault-tolerance:** both for crashes and stragglers
- **Efficiency:** low cost beyond base processing

Current steaming systems don't meet both goals

Traditional Streaming Systems

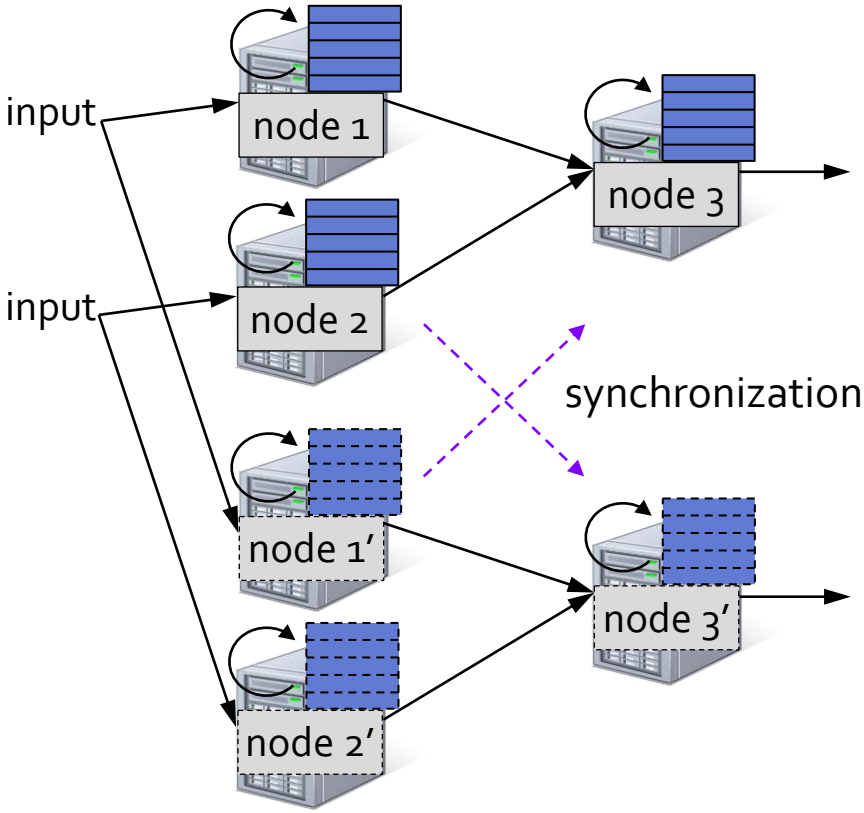
Continuous operator model

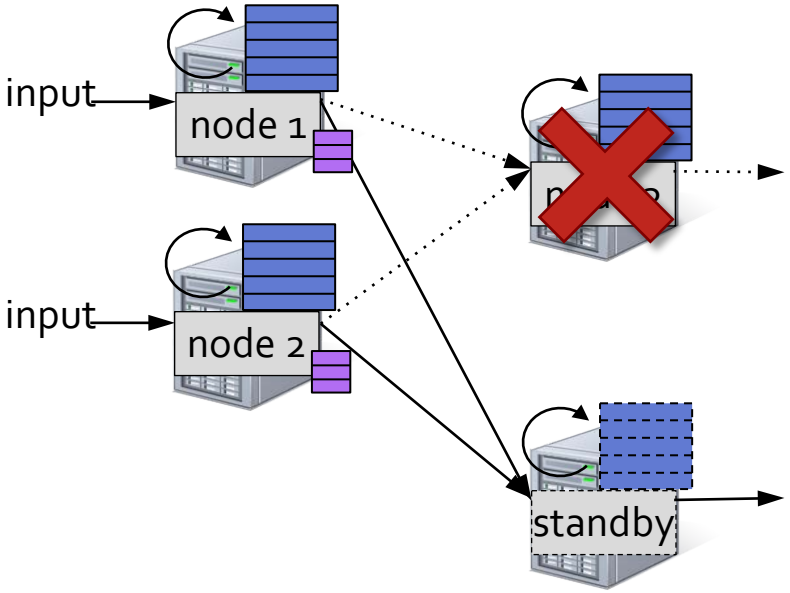
- » Each node has mutable state
- » For each record, update state & send new records



Traditional Streaming Systems

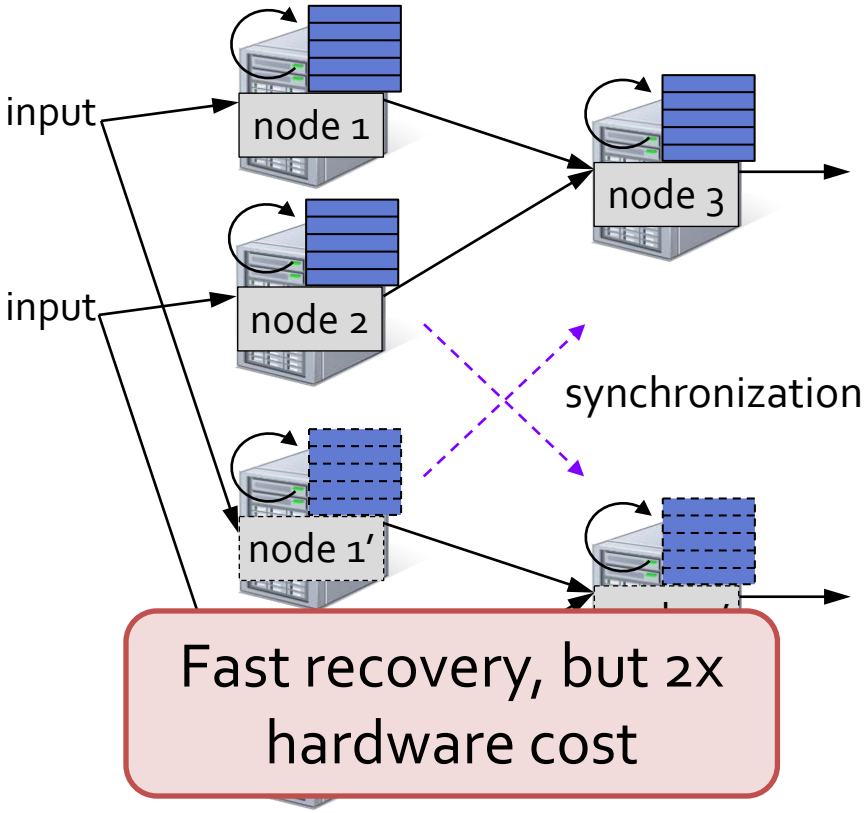
Fault tolerance via *replication* or *upstream backup*:



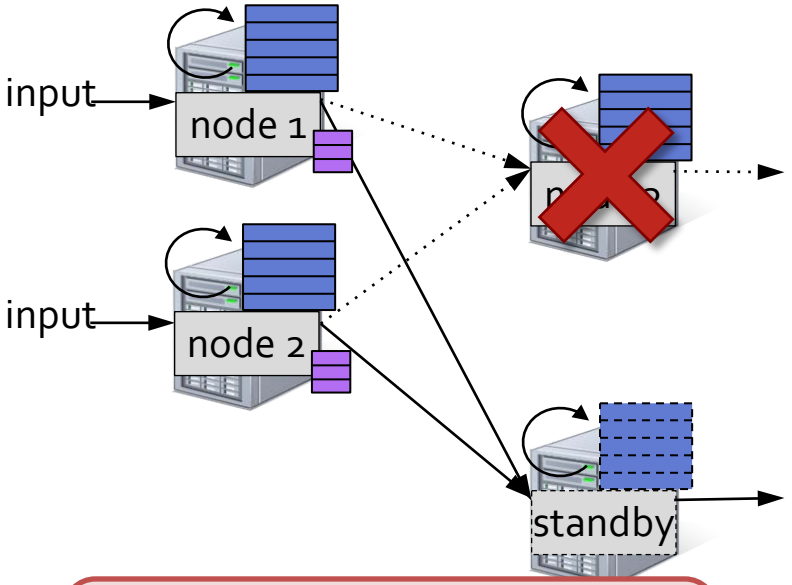


Traditional Streaming Systems

Fault tolerance via *replication* or *upstream backup*:



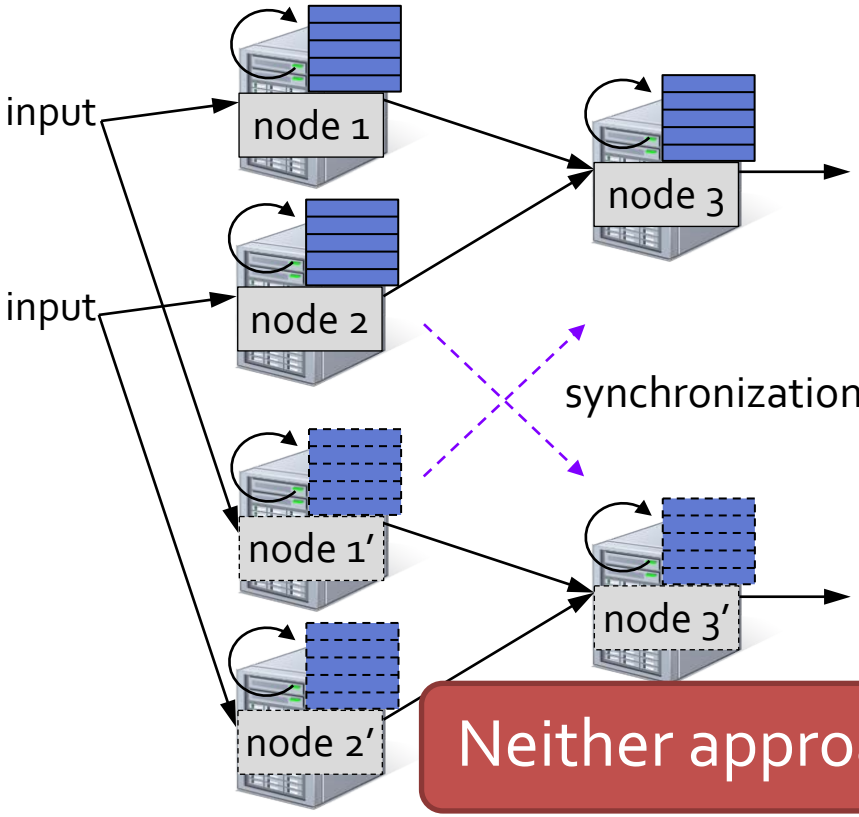
Fast recovery, but 2x hardware cost

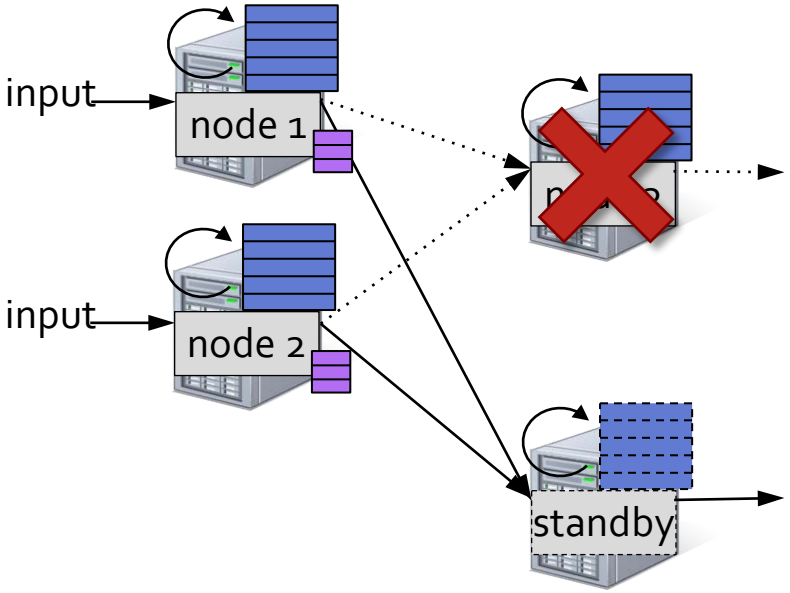


Only need 1 standby, but slow to recover

Traditional Streaming Systems

Fault tolerance via *replication* or *upstream backup*:





Neither approach can handle stragglers

Observation

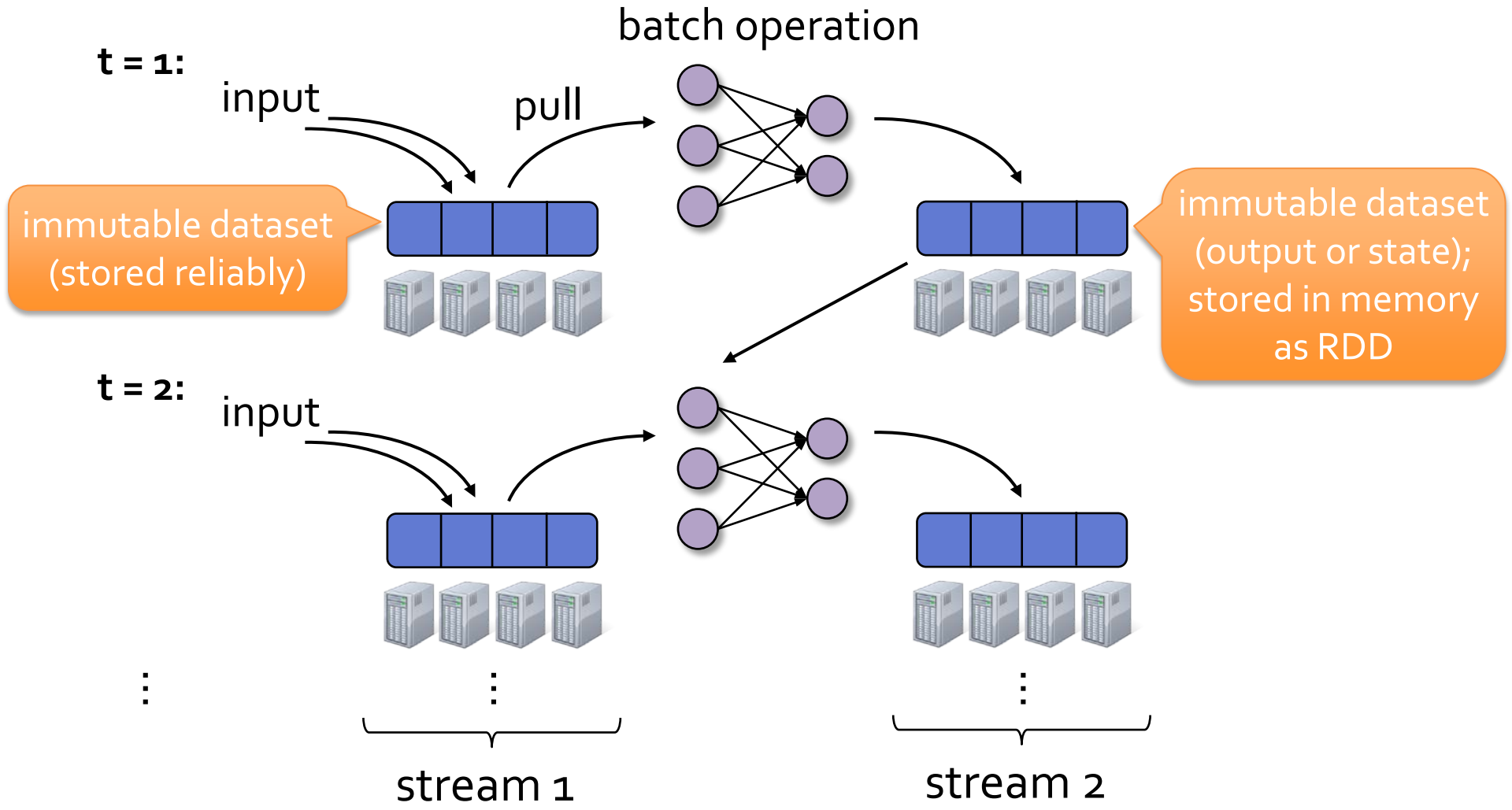
Batch processing models, like MapReduce, provide fault tolerance efficiently

- » Divide job into deterministic tasks
- » Rerun failed/slow tasks in parallel on other nodes

Idea: run streaming computations as a series of small, deterministic batch jobs

- » Same recovery schemes at much smaller timescale
- » To make latency low, store state in RDDs

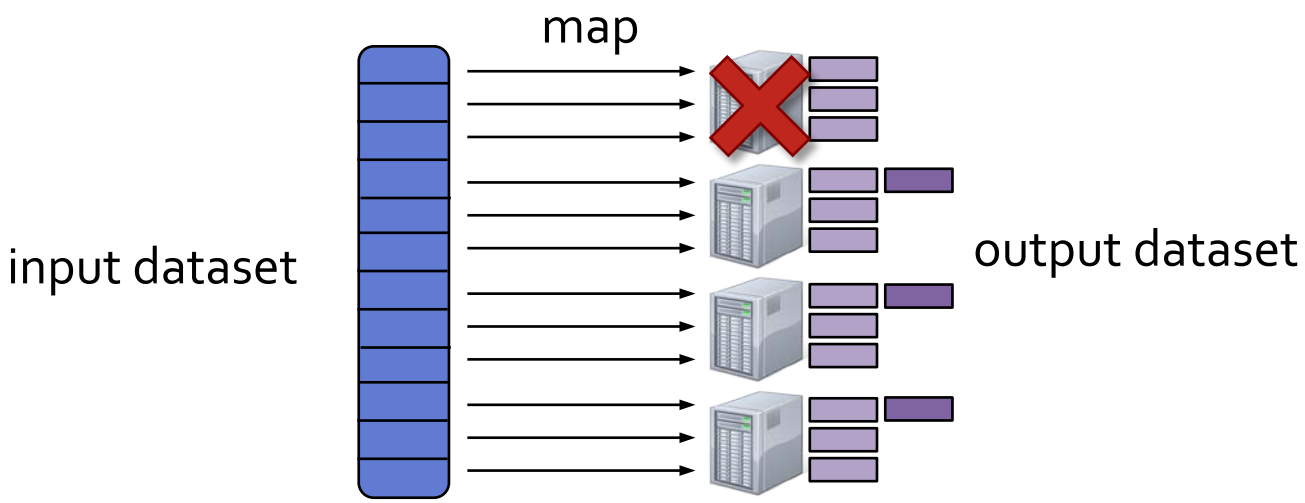
Discretized Stream Processing



Parallel Fault Recovery

Checkpoint state RDDs periodically

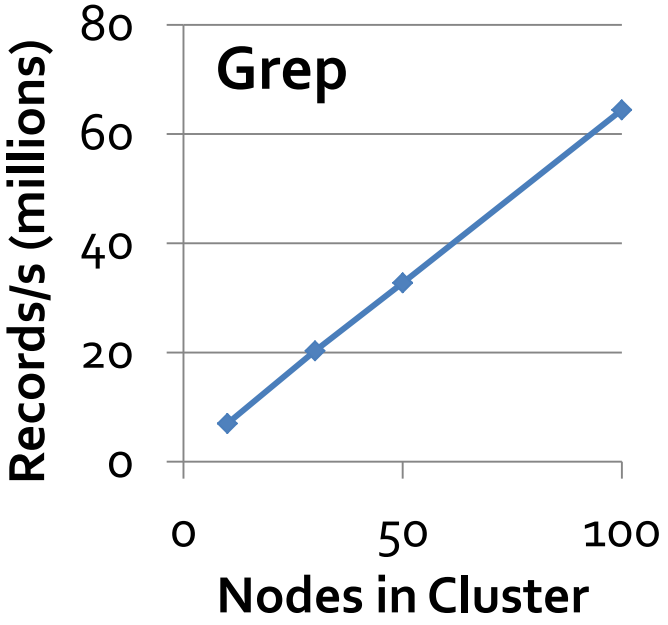
If a node fails or straggles, rebuild lost dataset partitions **in parallel** on other nodes

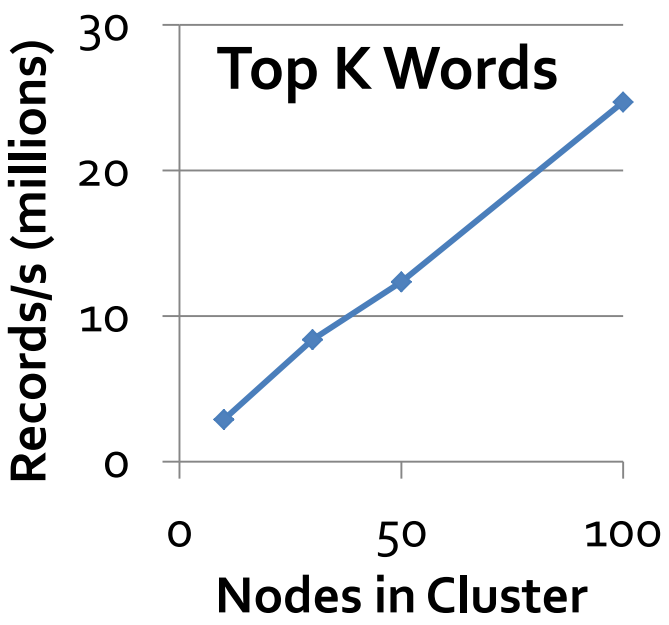


Faster recovery than upstream backup, without the cost of replication

How Fast Can It Go?

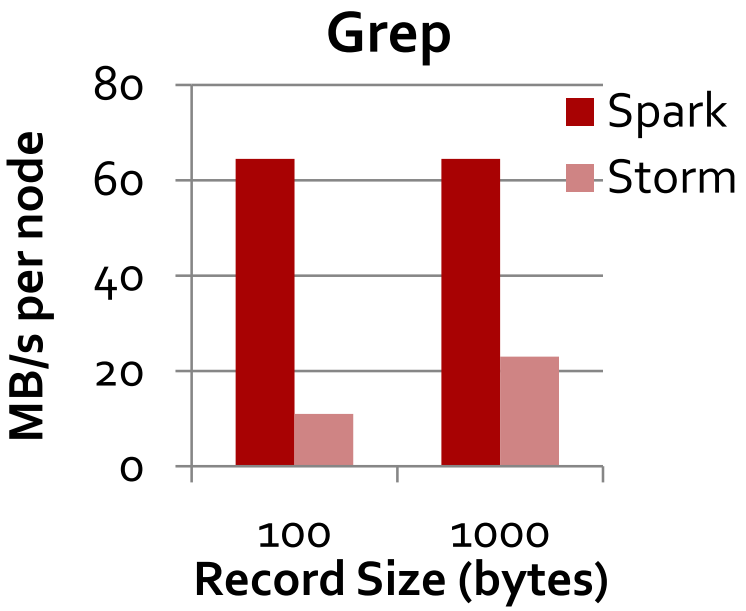
Can process over **60M records/s** (**6 GB/s**) on 100 nodes at **sub-second** latency

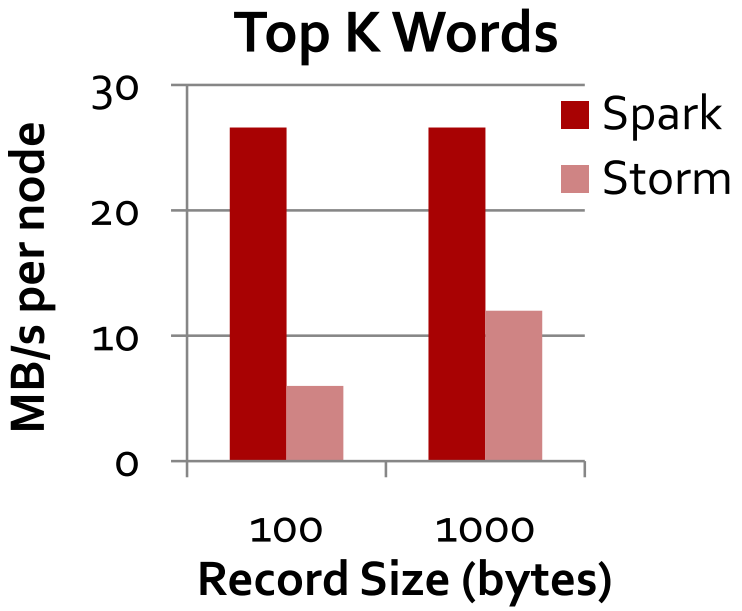




Maximum throughput for latency under 1 sec

Comparison with Storm



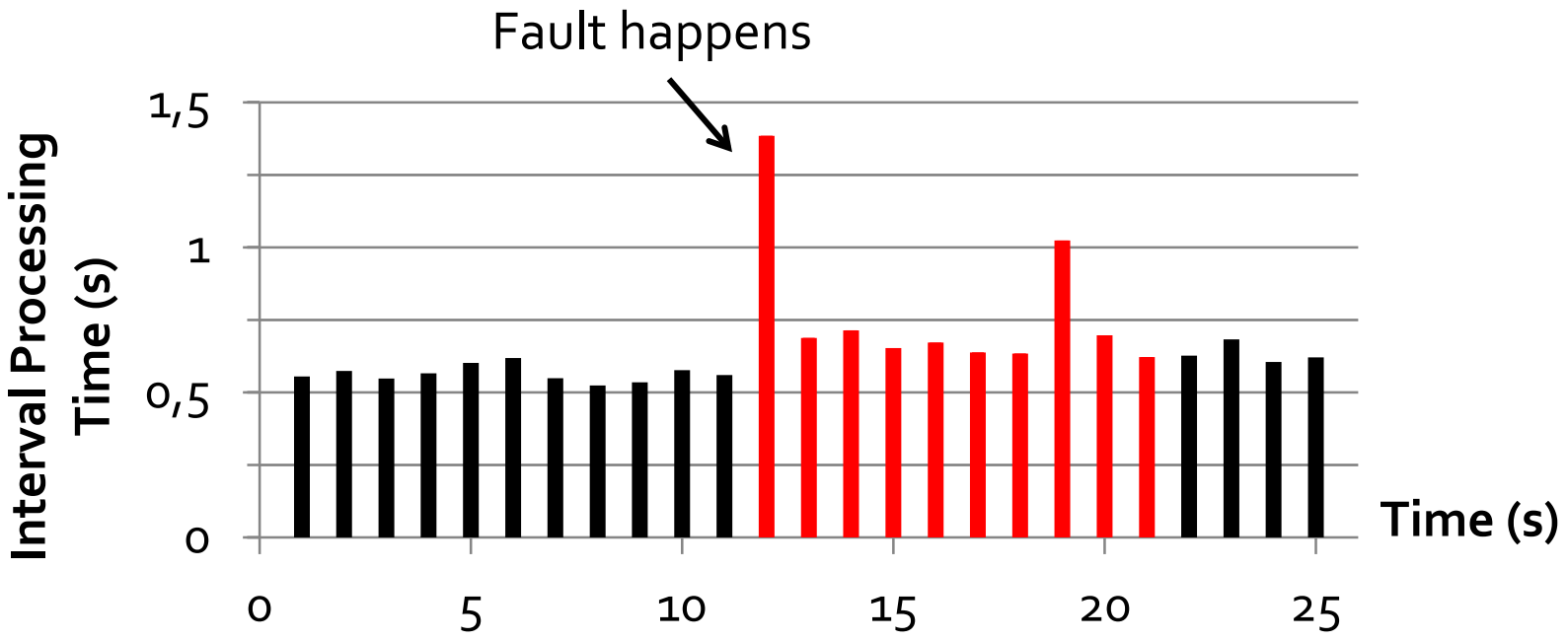


Storm limited to 100K records/s/node

Commercial systems report O(500K) total

How Fast Can It Recover?

Recovers from faults/stragglers within **1 second**

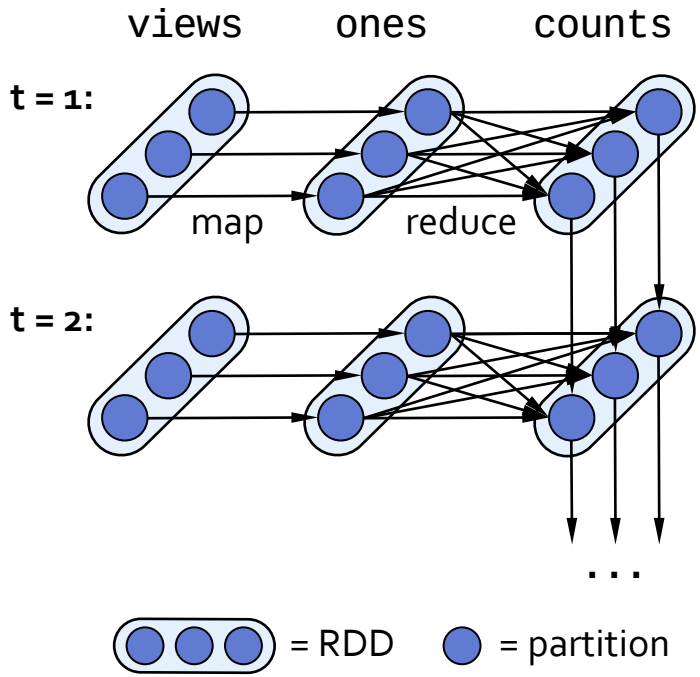


Sliding WordCount on 20 nodes with 10s checkpoint interval

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)
```

This Talk

Spark introduction & use cases

Shark: SQL over Spark

Spark Streaming

The power of unification

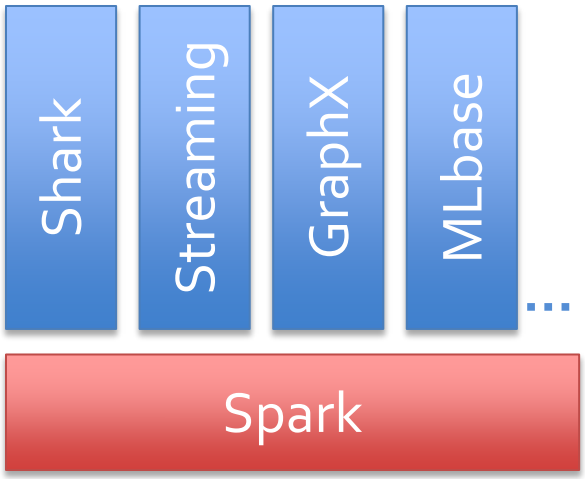
The Power of Unification

One of the things that makes Spark unique is unification of multiple programming models

- » SQL, graphs, streaming, etc on the **same** engine

This had powerful benefits:

- » For the engine
- » For users

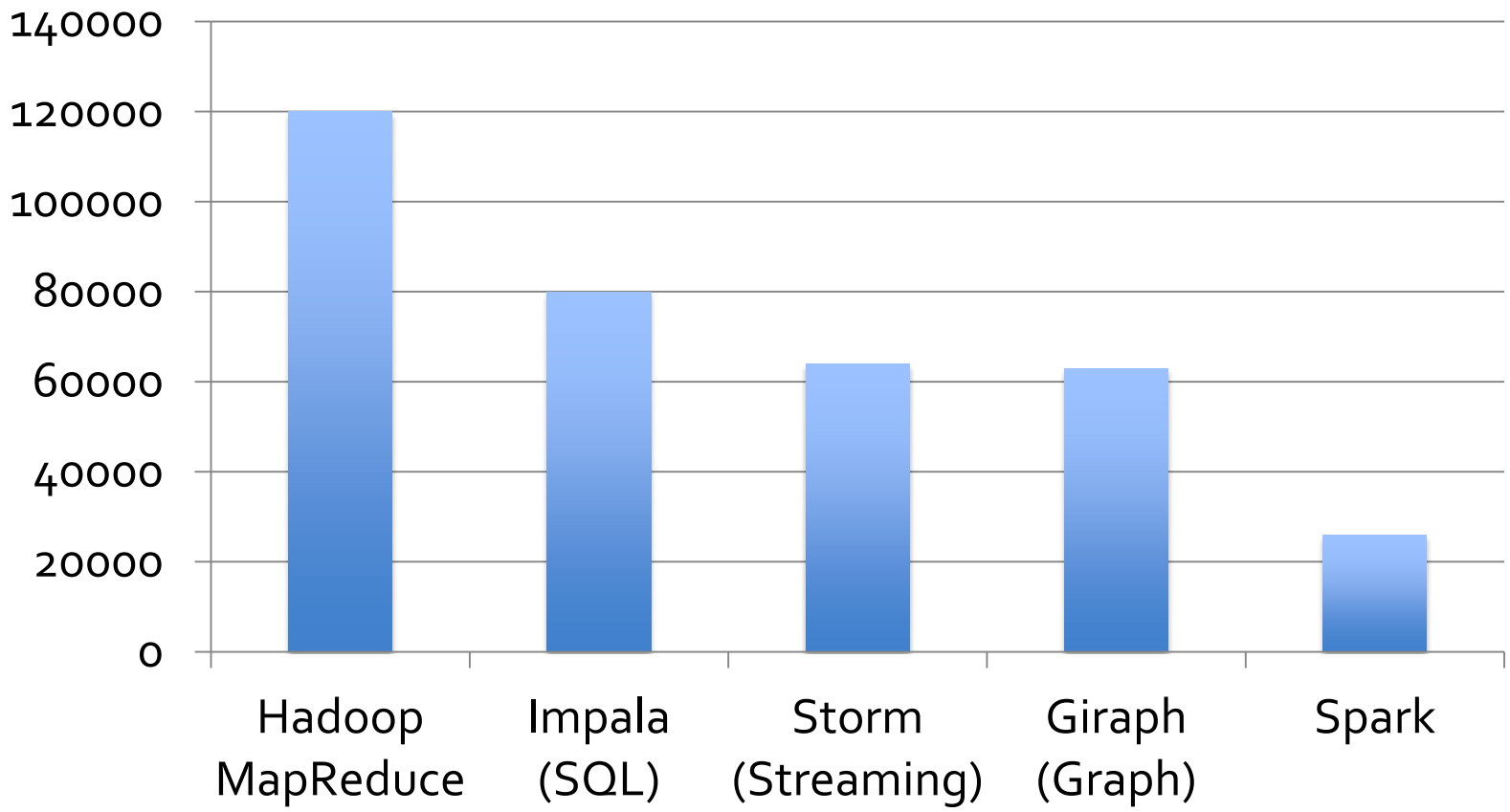


Engine Perspective

Let's compare Spark with leading frameworks in

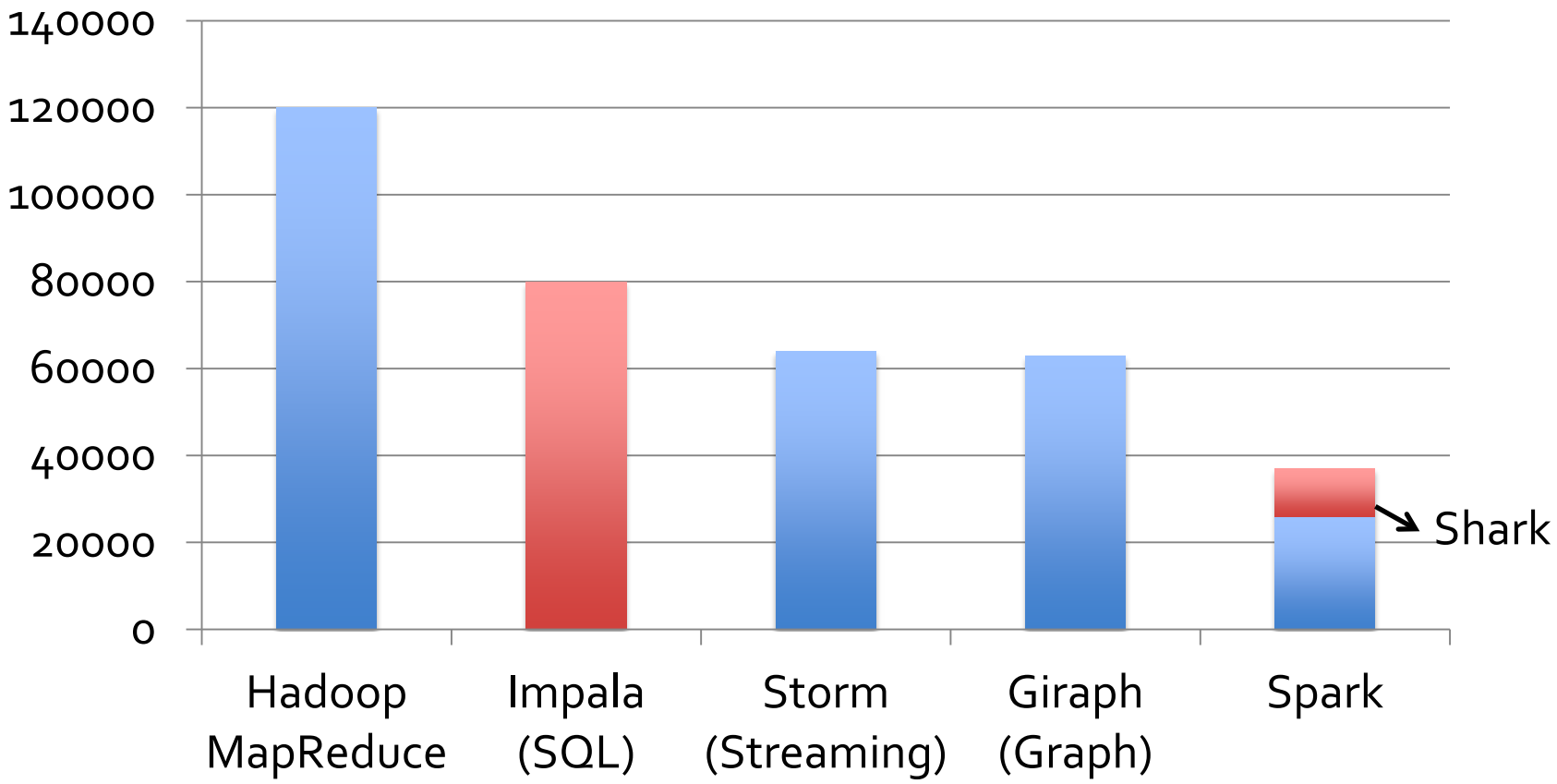
- Code size
- Performance

Code Size



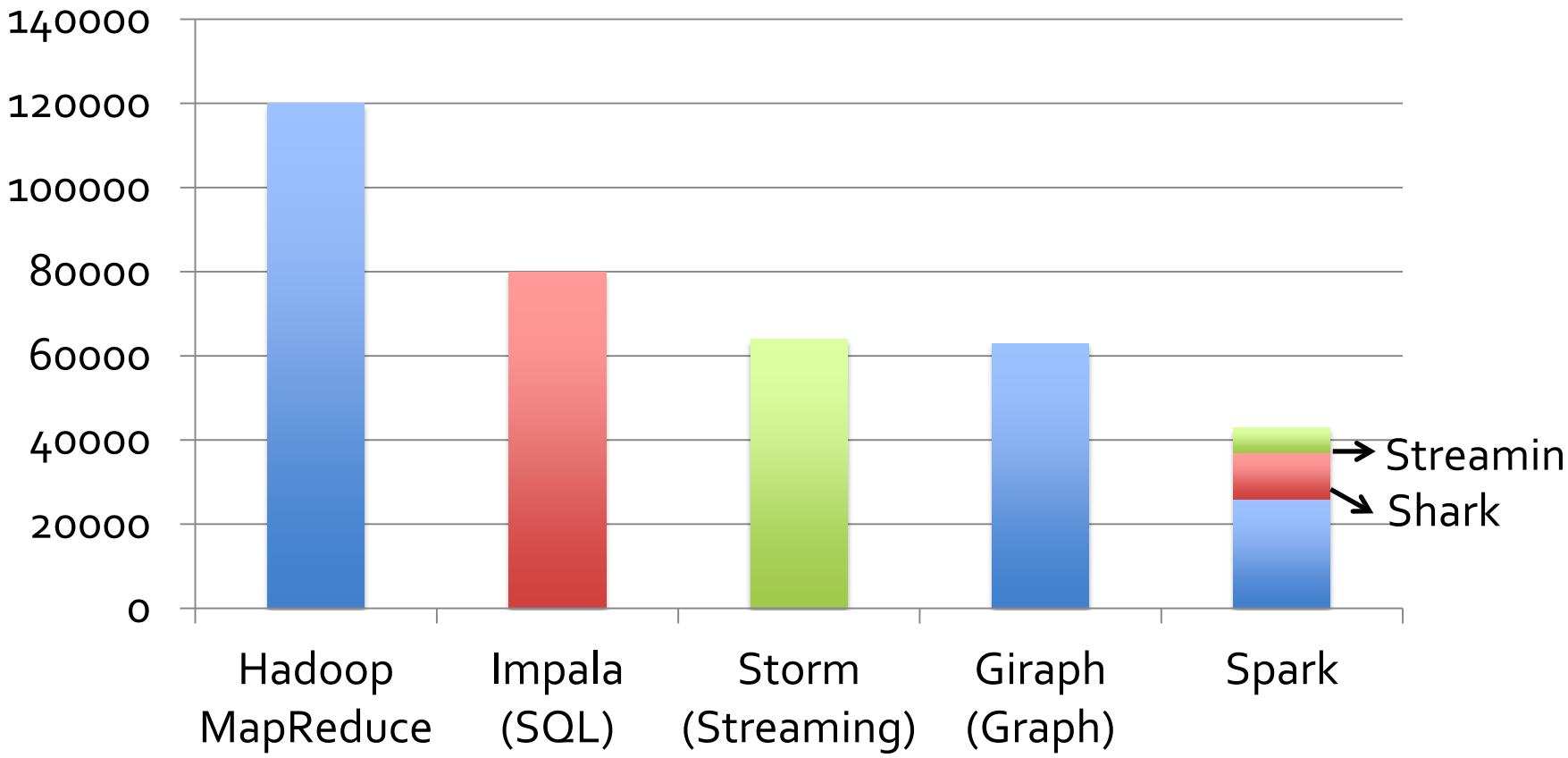
non-test, non-example source lines

Code Size



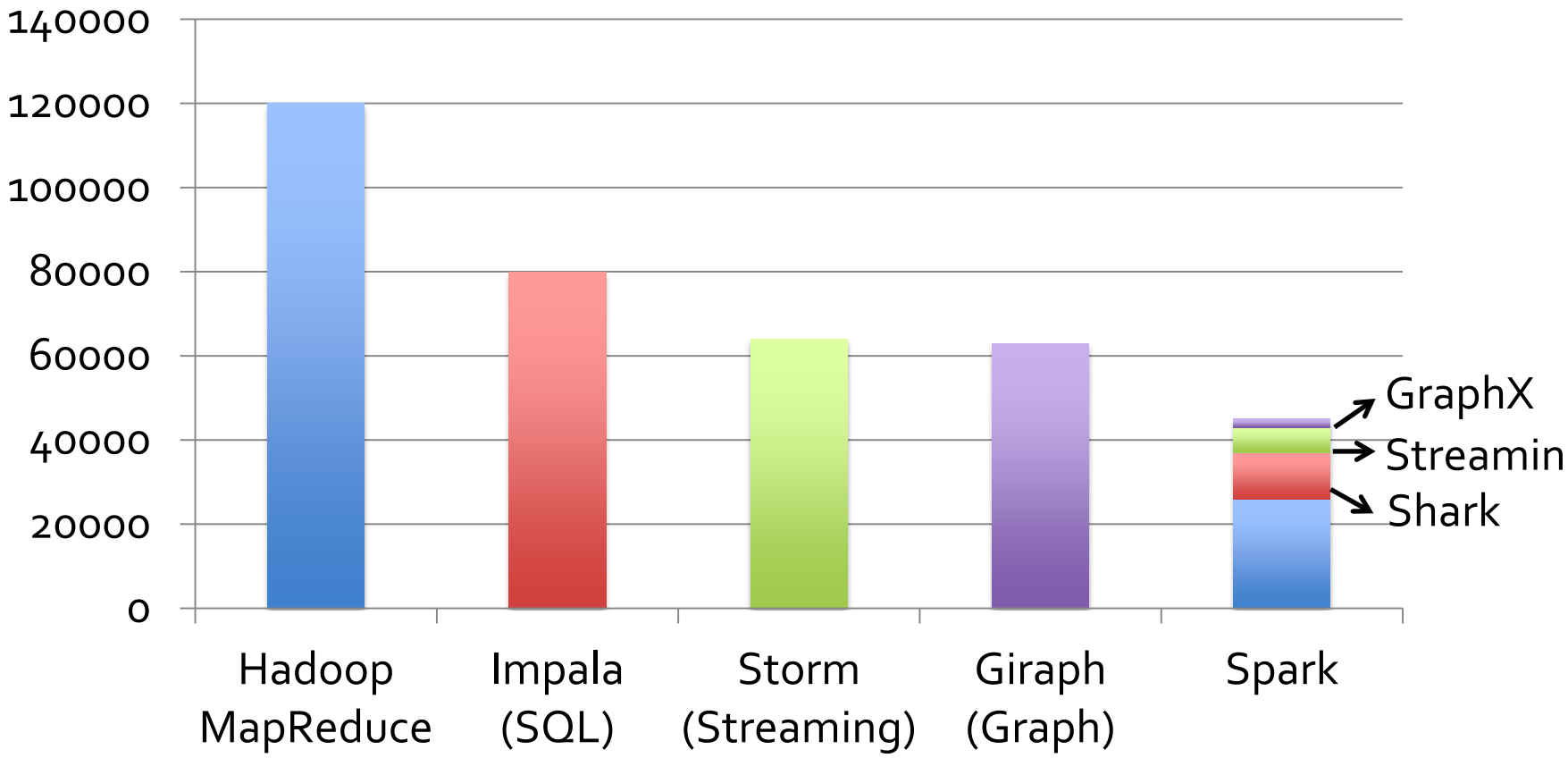
non-test, non-example source lines

Code Size



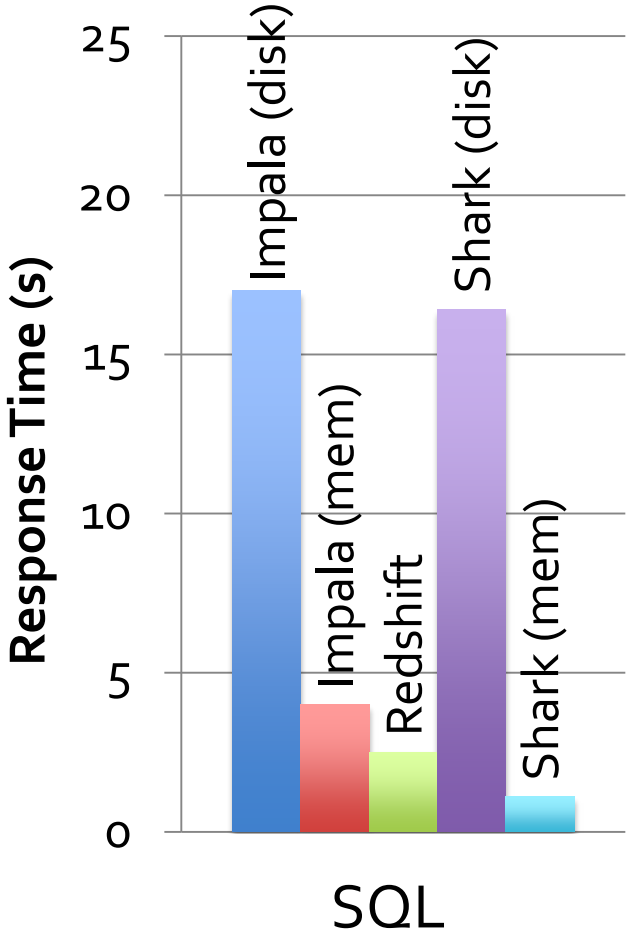
non-test, non-example source lines

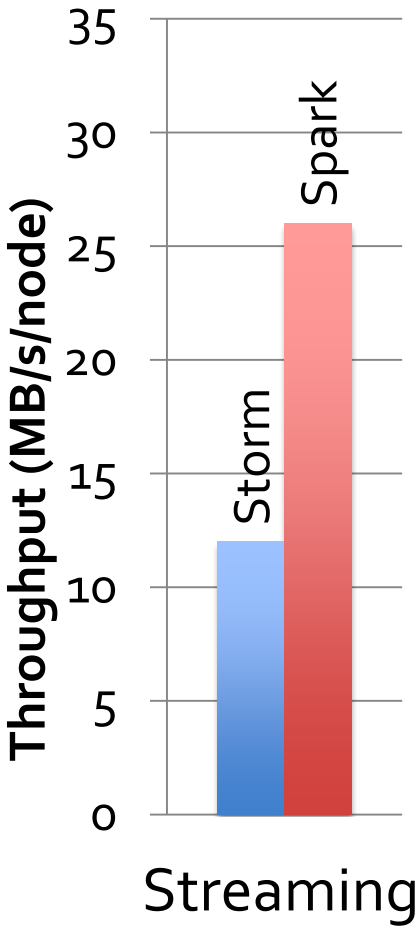
Code Size

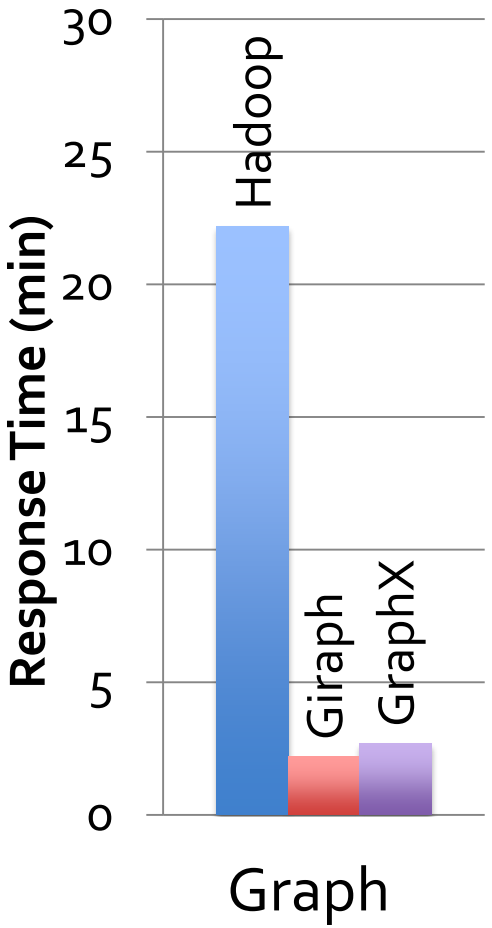


non-test, non-example source lines

Performance







User Perspective

Unified engine means that:

1. Apps can easily **compose** models (e.g. run a SQL query then PageRank on the result)
2. Composition is **fast** (no writing to disk)
3. All models get **Spark shell** for free (e.g. use it to inspect your streams' state, or your graph)

Long-Term Direction

As big data apps grow in complexity, combining processing models is essential

- » E.g. run SQL query then machine learning on result

Data sharing between models is often slower than the computations themselves

- » E.g. HDFS write versus in-memory iteration

Thus, unified engines increasingly important

Getting Started

Visit spark-project.org for videos, tutorials, and hands-on exercises

Easy to run in local mode, private clusters, EC2

Compatible with any Hadoop storage system

Online training camp:
ampcamp.berkeley.edu

Conclusion

Big data analytics is evolving to include:

- More **complex** analytics (e.g. machine learning)
- More **interactive** ad-hoc queries
- More **real-time** stream processing

Spark is a platform that *unifies* these models, enabling sophisticated apps

More info: spark-project.org

Backup Slides

Behavior with Not Enough RAM