ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Latency

Big Data Analysis with Scala and Spark

Heather Miller

Data-Parallel Programming

In the Parallel Programming course, we learned:

- Data parallelism on a single multicore/multi-processor machine.
- Parallel collections as an implementation of this paradigm.

Data-Parallel Programming

In the Parallel Programming course, we learned:

- Data parallelism on a single multicore/multi-processor machine.
- Parallel collections as an implementation of this paradigm.

Today:

- Data parallelism in a *distributed setting*.
- Distributed collections abstraction from Apache Spark as an implementation of this paradigm.

Distribution

Distribution introduces important concerns beyond what we had to worry about when dealing with parallelism in the shared memory case:

- *Partial failure:* crash failures of a subset of the machines involved in a distributed computation.
- *Latency:* certain operations have a much higher latency than other operations due to network communication.

Distribution

Distribution introduces important concerns beyond what we had to worry about when dealing with parallelism in the shared memory case:

- *Partial failure:* crash failures of a subset of the machines involved in a distributed computation.
- *Latency:* certain operations have a much higher latency than other operations due to network communication.

Latency cannot be masked completely; it will be an important aspect that also impacts the *programming model*.

Important Latency Numbers

L1 cache reference	0.5ns	
Branch mispredict	5ns	
L2 cache reference	7ns	
Mutex lock/unlock	25ns	
Main memory reference	100ns	
Compress 1K bytes with Zippy	3,000ns	= 3µs
Send 2K bytes over 1Gbps network	20,000ns	= 20µs
SSD random read	150,000ns	= 150µs
Read 1 MB sequentially from	250,000ns	= 250µs
Roundtrip within same datacenter	500,000ns	= 0.5ms
Read 1MB sequentially from SSD	1,000,000ns	= 1ms
Disk seek	10,000,000ns	= 10ms
Read 1MB sequentially from disk	20,000,000ns	= 20ms
Send packet US → Europe → US	150,000,000ns	= 150ms

Original compilation by Jeff Dean & Peter Norvig, w/ contributions by Joe Hellerstein & Erik Meijer

Important Latency Numbers

L1 cache reference	0.5ns	
Branch mispredict	5ns	
L2 cache reference	7ns	
Mutex lock/unlock	25ns	
Main memory reference	100ns	
Compress 1K bytes with Zippy	3,000ns	= 3µs
Send 2K bytes over 1Gbps network	20,000ns	= 20µs
SSD random read	150,000ns	= 150µs
Read 1 MB sequentially from memory	250,000ns	= 250µs
Roundtrip within same datacenter	500,000ns	= 0.5ms
Read 1MB sequentially from SSD	1,000,000ns	= 1ms
Disk seek	10,000,000ns	= 10ms
Read 1MB sequentially from disk	20,000,000ns	= 20ms
Send packet US → Europe → US	150,000,000ns	= 150ms

100x

Original compilation by Jeff Dean & Peter Norvig, w/ contributions by Joe Hellerstein & Erik Meijer

Important Latency Numbers

L1 cache reference	0.5ns	
Branch mispredict	5ns	
L2 cache reference	7ns	
Mutex lock/unlock	25ns	
Main memory reference	100ns	
Compress 1K bytes with Zippy	3,000ns	= 3µs
Send 2K bytes over 1Gbps network	20,000ns	= 20µs
SSD random read	150,000ns	= 150µs
Read 1 MB sequentially from	250,000ns	= 250µs
Roundtrip within same datacenter	500,000ns	= 0.5ms
Read 1MB sequentially from SSD	1,000,000ns	= 1ms
Disk seek	10,000,000ns	= 10ms
Read 1MB sequentially from disk	20,000,000ns	= 20ms
Send packet US → Europe → US	150,000,000ns	= 150ms

1,000,000x SLOWER

Original compilation by Jeff Dean & Peter Norvig, w/ contributions by Joe Hellerstein & Erik Meijer

Important Latency Numbers

memory

disk

network

L1 cache reference	0.5ns	
Branch mispredict	5ns	
L2 cache reference	7ns	
Mutex lock/unlock	25ns	
Main memory reference	100ns	
Compress 1K bytes with Zippy	3,000ns	= 3µs
Send 2K bytes over 1Gbps network	20,000ns	= 20µs
SSD random read	150,000ns	= 150µs
Read 1 MB sequentially from memory	250,000ns	= 250µs
Roundtrip within same datacenter	500,000ns	= 0.5ms
Read 1MB sequentially from SSD	1,000,000ns	= 1ms
Disk seek	10,000,000ns	= 10ms
Read 1MB sequentially from disk	20,000,000ns	= 20ms
Send packet US → Europe → US	150,000,000ns	= 150ms

memory: fastest
disk: slow
network: slowest

Original compilation by Jeff Dean & Peter Norvig, w/ contributions by Joe Hellerstein & Erik Meijer

Latency Numbers Intuitively

To get a better intuition about the *orders-of-magnitude differences* of these numbers, let's **humanize** these durations.

Method: multiply all these durations by a billion.

Then, we can map each latency number to a *human activity*.

Humanized Latency Numbers

Humanized durations grouped by magnitude:

Minute:

L1 cache reference	0.5 s	One heart beat (0.5 s)
Branch mispredict	5 s	Yawn
L2 cache reference	7 s	Long yawn
Mutex lock/unlock	25 s	Making a coffee

Hour:

Main memory reference	100 s	Brushing your teeth
Compress 1K bytes **with** Zippy	50 min	One episode of a TV show

Humanized Latency Numbers

Day:

Send 2K bytes over 1 Gbps network	5.5 hr	From lunch to end of work day

Week:

SSD random read	1.7 days	A normal weekend
Read 1 MB sequentially from memory	2.9 days	A long weekend
Round trip within same datacenter	5.8 days	A medium vacation
Read 1 MB sequentially from SSD	11.6 days	Waiting for almost 2 weeks for a delivery

More Humanized Latency Numbers

Year:

Disk seek	16.5 weeks	A semester in university
Read 1 MB sequentially from disk	7.8 months	Almost producing a **new** human being
The above 2 together	1 year	

Decade:

Send packet CA->Netherlands->CA	4.8 years	Average time it takes to complete a bachelor's degree

Latency and System Design

Memory	
L1 cache reference	**0.5 s**
Main memory reference	**100 s**
Read 1MB sequentially from memory	**2.9 days**
seconds/days	

Disk	
Disk seek	**16.5 weeks**
Read 1MB sequentially from disk	**7.8 months**
weeks/months	

Network	
Round trip within same datacenter	**5.8 days**
Send packet US→Eur→US	**4.8 years**
weeks/years	

Big Data Processing and Latency?

With some intuition now about how expensive network communication and disk operations can be, one may ask:

How do these latency numbers relate to big data processing?

To answer this question, let's first start with Spark's predecessor, Hadoop.

Hadoop/MapReduce

Hadoop is a widely-used large-scale batch data processing framework. It's an open source implementation of Google's MapReduce.

Hadoop/MapReduce

Hadoop is a widely-used large-scale batch data processing framework. It's an open source implementation of Google's MapReduce.

MapReduce was ground-breaking because it provided:

- a simple API (simple `map` and `reduce` steps)
- $**$ **fault tolerance** $**$

Hadoop/MapReduce

Hadoop is a widely-used large-scale batch data processing framework. It's an open source implementation of Google's MapReduce.

MapReduce was ground-breaking because it provided:

- a simple API (simple `map` and `reduce` steps)
- ** **fault tolerance** **

Fault tolerance is what made it possible for Hadoop/MapReduce to scale to 100s or 1000s of nodes at all.

Hadoop/MapReduce + Fault Tolerance

Why is this important?

For 100s or 1000s of old commodity machines, likelihood of at least one node failing is **very high** midway through a job.

Hadoop/MapReduce + Fault Tolerance

Why is this important?

For 100s or 1000s of old commodity machines, likelihood of at least one node failing is **very high** midway through a job.

Thus, Hadoop/MapReduce's ability to recover from node failure enabled:

- computations on unthinkably large data sets to succeed to completion.

Hadoop/MapReduce + Fault Tolerance

Why is this important?

For 100s or 1000s of old commodity machines, likelihood of at least one node failing is **very high** midway through a job.

Thus, Hadoop/MapReduce's ability to recover from node failure enabled:

- computations on unthinkably large data sets to succeed to completion.

Fault tolerance + simple API =

At Google, MapReduce made it possible for an average Google software engineer to craft a complex pipeline of map/reduce stages on extremely large data sets.

Why Spark?

Why Spark?

Fault-tolerance in Hadoop/MapReduce comes at a cost.

Between each map and reduce step, in order to recover from potential failures, Hadoop/MapReduce shuffles its data and write intermediate data to disk.

Why Spark?

Fault-tolerance in Hadoop/MapReduce comes at a cost.

Between each map and reduce step, in order to recover from potential failures, Hadoop/MapReduce shuffles its data and write intermediate data to disk.

Remember:

Reading/writing to disk: **100~~0~~x slower** than in-memory
Network communication: **1,000,000x slower** than in-memory

Why Spark?

Spark...

- Retains fault-tolerance
- Different strategy for handling latency (latency significantly reduced!)

Why Spark?

Spark...

- Retains fault-tolerance
- Different strategy for handling latency (latency significantly reduced!)

Achieves this using ideas from functional programming!

Why Spark?

Spark...

- Retains fault-tolerance
- Different strategy for handling latency (latency significantly reduced!)

Achieves this using ideas from functional programming!

Idea: Keep all data **immutable and in-memory**. All operations on data are just functional transformations, like regular Scala collections. Fault tolerance is achieved by replaying functional transformations over original dataset.

Why Spark?

Spark...

- Retains fault-tolerance
- Different strategy for handling latency (latency significantly reduced!)

Achieves this using ideas from functional programming!

Idea: Keep all data **immutable and in-memory**. All operations on data are just functional transformations, like regular Scala collections. Fault tolerance is achieved by replaying functional transformations over original dataset.

Result: Spark has been shown to be 100x more performant than Hadoop, while adding even more expressive APIs.

Latency and System Design (Humanized)

Memory

L1 cache reference	**0.5 s**
Main memory reference	**100 s**
Read 1MB sequentially from memory	**2.9 days**

seconds/days

Disk

Disk seek	**16.5 weeks**
Read 1MB sequentially from disk	**7.8 months**

weeks/months

Network

Round trip within same datacenter	**5.8 days**
Send packet US→Eur→US	**4.8 years**

weeks/years

Latency and System Design

Memory		Disk		Network	
L1 cache reference	**0.5 s**	Disk seek	**16.5 weeks**	Round trip within same datacenter	**5.8 days**
Main memory reference	**100 s**	Read 1MB sequentially from disk	**7.8 months**	Send packet US→Eur→US	**4.8 years**
Read 1MB sequentially from memory	**2.9 days**				
seconds/days		**weeks/months**		**weeks/years**	

Hadoop

Latency and System Design

Memory

L1 cache reference	**0.5 s**
Main memory reference	**100 s**
Read 1MB sequentially from memory	**2.9 days**

seconds/days

Disk

Disk seek	**16.5 weeks**
Read 1MB sequentially from disk	**7.8 months**

weeks/months

Network

Round trip within same datacenter	**5.8 days**
Send packet US→Eur→US	**4.8 years**

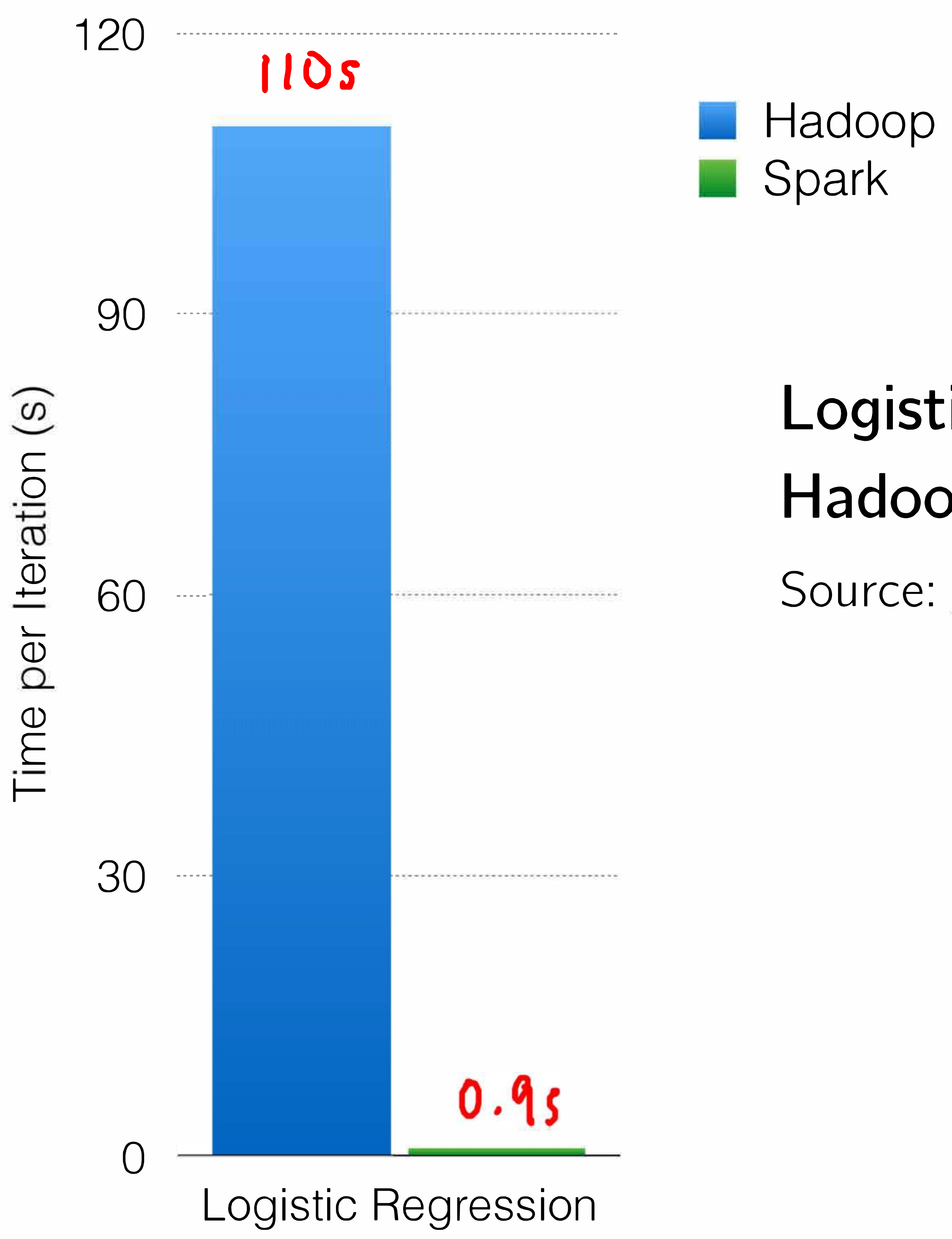
weeks/years

Spark

shift to inmemory

aggressively minimize

Spark versus Hadoop Performance?



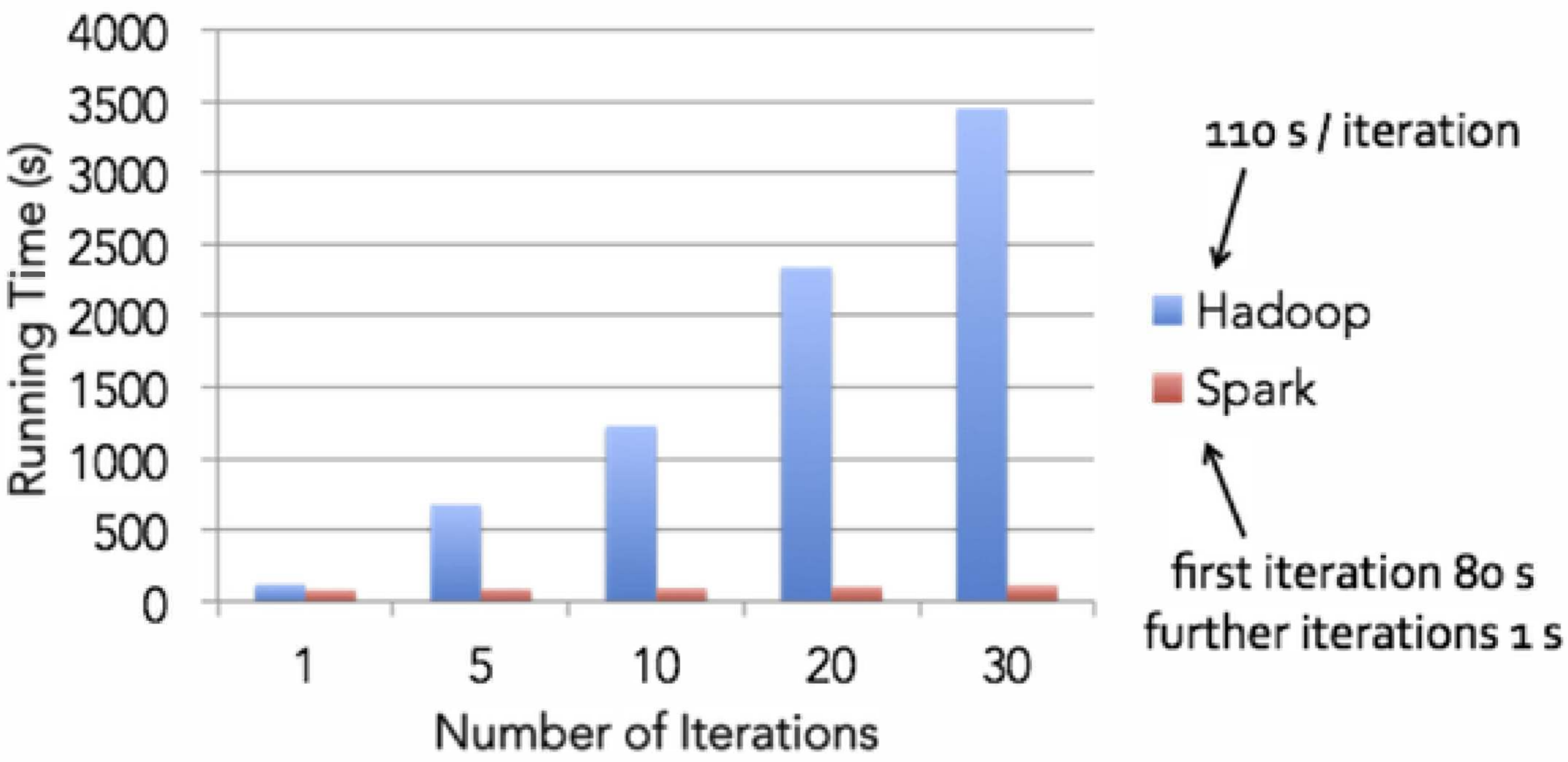
Logistic Regression in Hadoop and Spark

Source: spark.apache.org

Spark versus Hadoop Performance?

Logistic Regression in Hadoop and Spark, more iterations!

Source: https://databricks.com/blog/2014/03/20/apache-spark-a-delight-for-developers.html



Hadoop vs Spark Performance, More Intuitively

Day-to-day, these perforamnce improvements can mean the difference between:

Hadoop/MapReduce

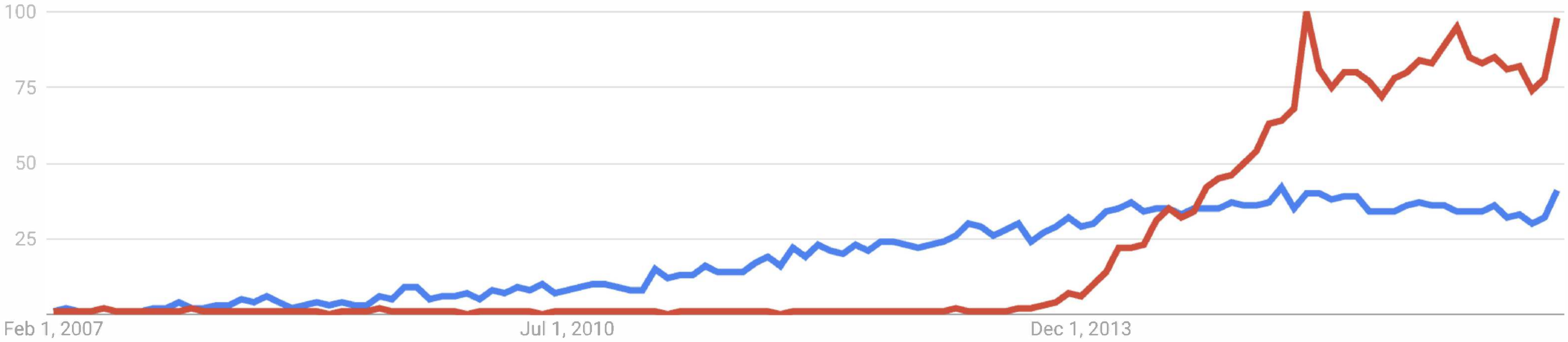
1. start job
2. eat lunch
3. get coffee
4. pick up kids
5. job completes

Spark

1. start job
2. get coffee
3. job completes

Spark versus Hadoop Popularity?

According to Google Trends, Spark has surpassed Hadoop in popularity.



Apache Spark

Apache Hadoop

Google Trends:

Apache Hadoop vs Apache Spark

February 2007 - February 2017