ĐẠI HỌC BÁCH KHOA HÀ NỘI
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

1

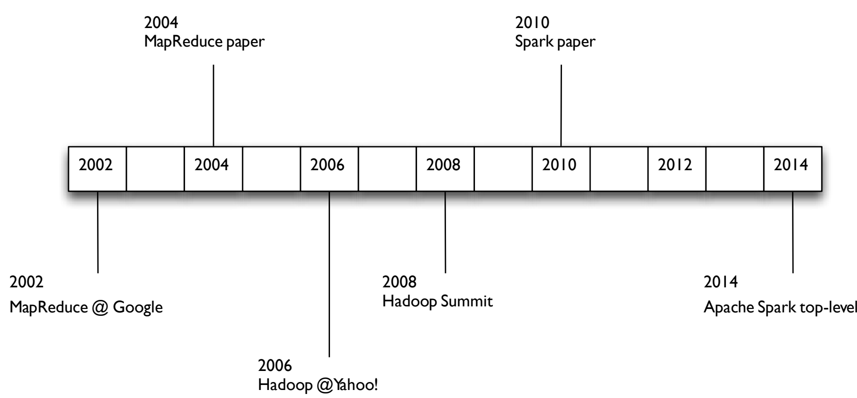
Spark Introduction

2

Agenda

- History of Spark
- Introduction
- Components of Stack
- Resilient Distributed Dataset – RDD

History of Spark



History of Spark

circa 1979 – **Stanford**, **MIT**, **CMU**, etc.
set/list operations in LISP, Prolog, etc., for parallel processing
www-formal.stanford.edu/ jmc/ history / lisp/ lisp.htm

circa 2004 – **Google**
MapReduce: Simplified Data Processing on Large Clusters Jeffrey Dean and Sanjay Ghemawat
research.google.com/ archive/ mapreduce.html

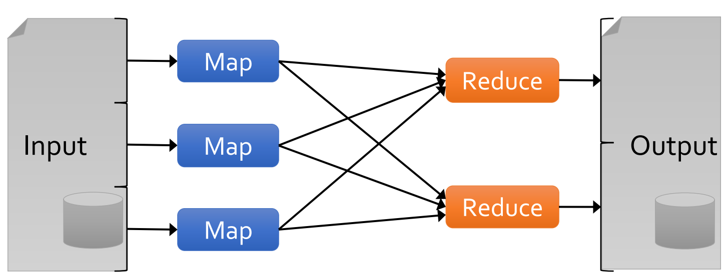
circa 2006 – **Apache**
Hadoop, originating from the Nutch Project Doug Cutting
research.yahoo.com/ files/ cutting.pdf

circa 2008 – **Yahoo**
web scale search indexing Hadoop Submit, HUG, etc.
developer.yahoo.com/hadoop/

circa 2009 – **Amazon AWS**
Elastic MapReduce
Hadoop modified for EC2/S3, plus support for Hive, Pig, Cascading, etc.
aws.amazon.com/ elasticmapreduce/

MapReduce

Most current cluster programming models are based on *acyclic data flow* from stable storage to stable storage



MapReduce

- Acyclic data flow is inefficient for applications that repeatedly reuse a *working set* of data:
  - **Iterative** algorithms (machine learning, graphs)
  - **Interactive** data mining tools (R, Excel, Python)

Data Processing Improvement Goals

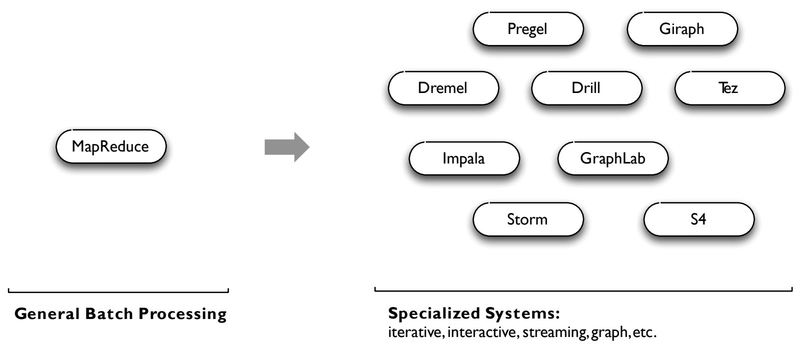
- **Low latency (interactive) queries on historical data:** enable faster decisions
  - E.g., identify why a site is slow and fix it

- **Low latency queries on live data (streaming):** enable decisions on real-time data
  - E.g., detect & block worms in real-time (a worm may infect **1mil** hosts in **1.3sec**)

- **Sophisticated data processing:** enable "better" decisions
  - E.g., anomaly detection, trend analysis

Therefore, people built specialized systems as workarounds…

Specialized Systems



The State of Spark, and Where We're Going Next
Matei Zaharia
Spark Summit (2013)
youtu.be/ nU6vO2EJAb4

Storage vs Processing Wars

NoSQL battles

- Relational vs NoSQL
- HBase vs Cassanrdra
- Redis vs Memcached vs Riak
- MongoDB vs CouchDB vs Couchbase
- Neo4j vs Titan vs Giraph vs OrientDB
- Solr vs Elasticsearch

Compute battles

- MapReduce vs Spark
- Spark Streaming vs Storm
- Hive vs Spark SQL vs Impala
- Mahout vs MLlib vs H20

Storage vs Processing Wars

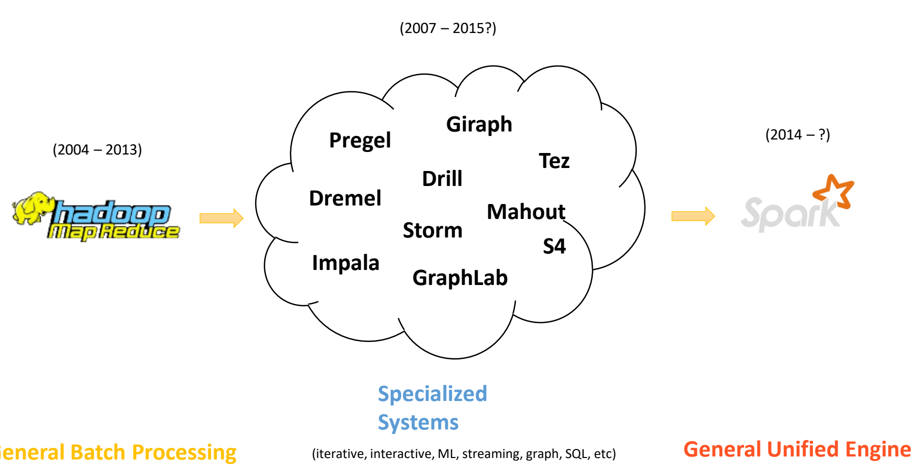
NoSQL battles

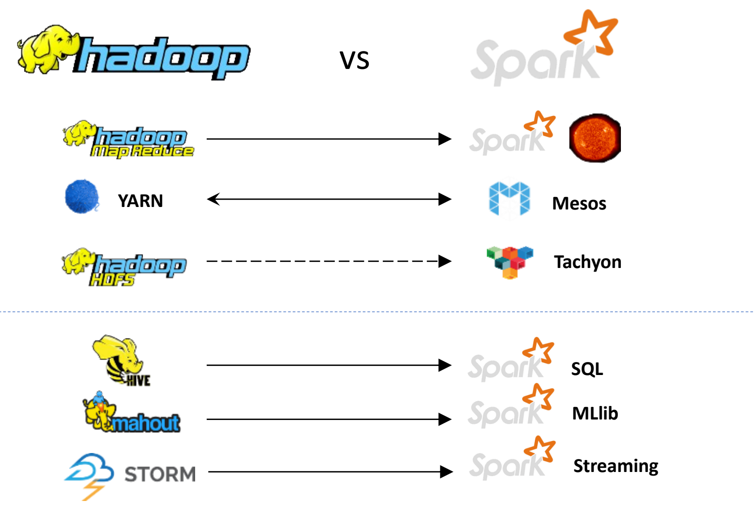
- Relational vs NoSQL
- HBase vs Cassanrdra
- Redis vs Memcached vs Riak
- MongoDB vs CouchDB vs Couchbase
- Neo4j vs Titan vs Giraph vs OrientDB
- Solr vs Elasticsearch

Compute battles

- MapReduce vs Spark
- Spark Streaming vs Storm
- Hive vs Spark SQL vs Impala
- Mahout vs MLlib vs H20

Specialized Systems

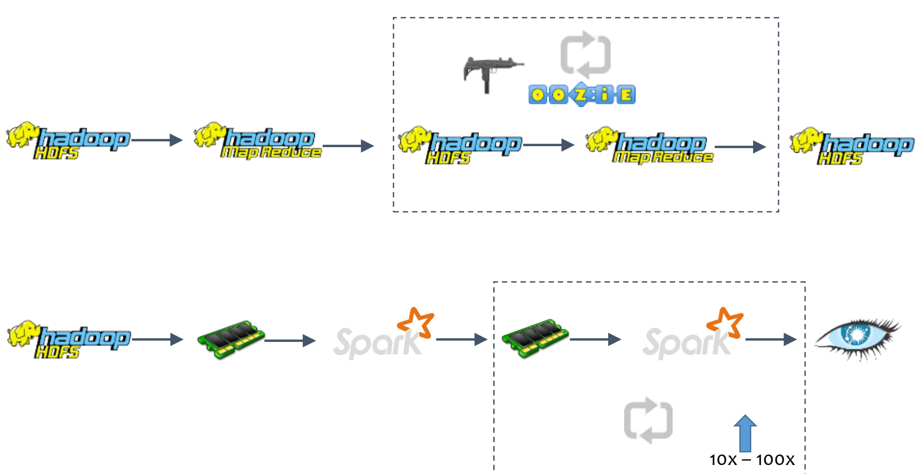




SOICT VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

13

Spark

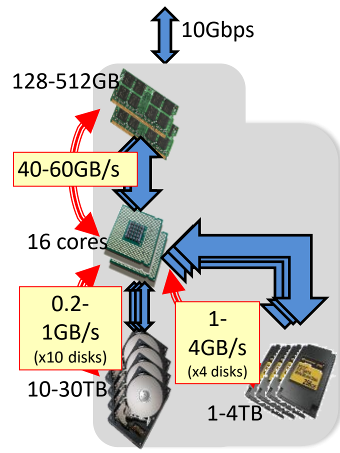


SOICT VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

14

Support Interactive and Streaming Comp.

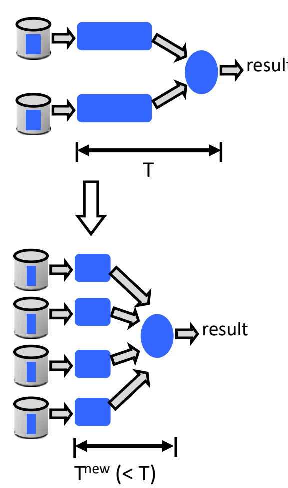
- Aggressive use of ***memory***
- Why?
  1. Memory transfer rates >> disk or SSDs
  2. Many datasets already fit into memory
     - Inputs of over 90% of jobs in Facebook, Yahoo!, and Bing clusters fit into memory
     - e.g., 1TB = 1 billion records @ 1KB each
  3. Memory density (still) grows with Moore's law
     - RAM/SSD hybrid memories at horizon



High end datacenter node

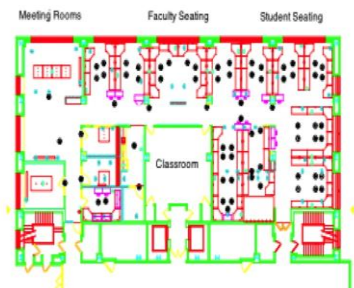
Support Interactive and Streaming Comp.

- Increase ***parallelism***
- Why?
  - Reduce work per node → improve latency
- Techniques:
  - Low latency parallel scheduler that achieve high locality
  - Optimized parallel communication patterns (e.g., shuffle, broadcast)
  - Efficient recovery from failures and straggler mitigation



Berkeley AMPLab

- "Launched" January 2011: 6 Year Plan
- 8 CS Faculty
- ~40 students
- 3 software engineers
- Organized for collaboration:



SoICT VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

17

Berkeley AMPLab

- Funding:
  - XData, CISE Expedition Grant
  - Industrial, founding sponsors
  - 18 other sponsors, including

Goal: Next Generation of Analytics Data Stack for Industry & Research:
- Berkeley Data Analytics Stack (BDAS)
- Release as Open Source

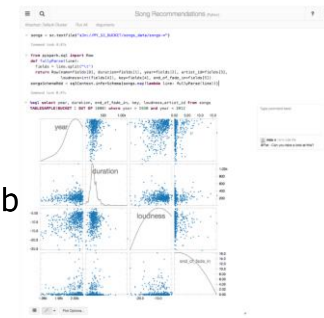
SoICT VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

18

Databricks

databricks
making big data simple

- Founded in late 2013
- by the creators of Apache Spark
- Original team from UC Berkeley AMPLab
- Raised $47 Million in 2 rounds



Databricks Cloud:
"A unified platform for building Big Data pipelines – from ETL to Exploration and Dashboards, to Advanced Analytics and Data Products."

VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

19

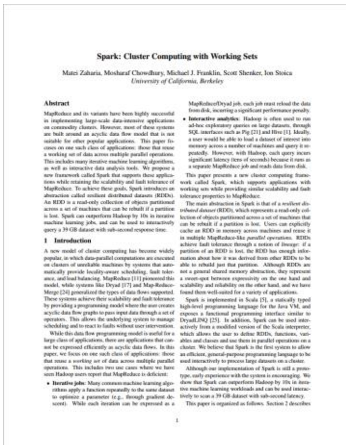
The Databricks team contributed more than **75%** of the code added to Spark in the 2014

databricks

VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

20

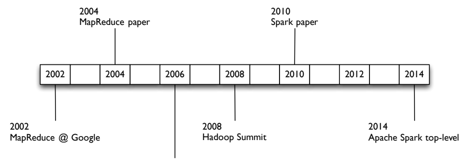
History of Spark



Spark: Cluster Computing with Working Sets

Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, Ion Stoica

University of California, Berkeley



Spark: Cluster Computing with Working Sets
Matei Zaharia, Mosharaf Chowdhury,
Michael J. Franklin, Scott Shenker, Ion Stoica
USENIX HotCloud (2010)
people.csail.mit.edu/ matei/ papers/ 2010/ hotcloud_spark.pdf

Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing
Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica NSDI (2012)
usenix.org/ system/ files/ conference/ nsdi12/ nsdi12-final138.pdf

VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

21

History of Spark

Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing

Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica

University of California, Berkeley

"We present Resilient Distributed Datasets (RDDs), a distributed memory abstraction that lets programmers perform in-memory computations on large clusters in a fault-tolerant manner.

RDDs are motivated by two types of applications that current computing frameworks handle inefficiently: iterative algorithms and interactive data mining tools.

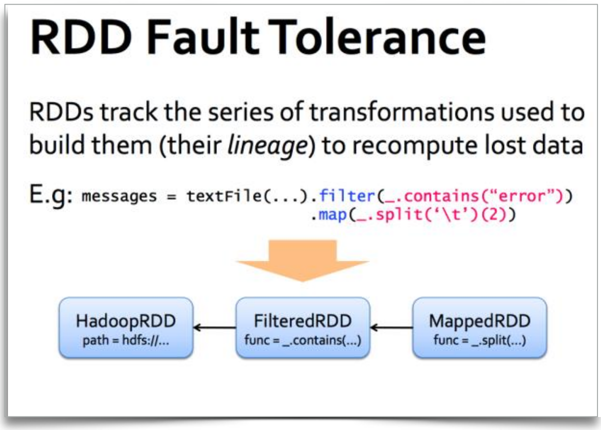
In both cases, keeping data in memory can improve performance by an order of magnitude."

April 2012

VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

22

History of Spark

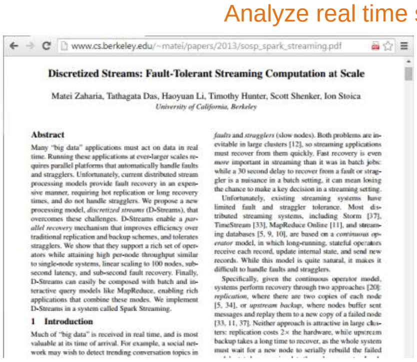


The State of Spark, and Where We're Going Next
Matei Zaharia
Spark Summit (2013)
youtu.be/nU6vO2EJAb4

History of Spark

Spark STREAMING

Analyze real time streams of data in ½ second intervals


www.cs.berkeley.edu/~matei/papers/2013/sosp_spark_streaming.pdf

Discretized Streams: Fault-Tolerant Streaming Computation at Scale

Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, Ion Stoica
University of California, Berkeley

Abstract

Many "big data" applications must act on data in real time. Running these applications at ever-larger scales requires parallel platforms that automatically handle faults and stragglers. Unfortunately, current distributed stream processing models provide fault recovery in an expensive manner, requiring hot replication or long recovery times, and do not handle stragglers. We propose a new processing model, *discretized streams* (D-Streams), that overcomes these challenges. D-Streams enable a *parallel recovery* mechanism that improves efficiency over traditional replication and backup schemes, and tolerates stragglers. We show that they support a rich set of operators while attaining high per-node throughput similar to single-node systems, linear scaling to 100 nodes, sub-second latency, and sub-second fault recovery. Finally, D-Streams can easily be composed with batch and interactive query models like MapReduce, enabling rich applications that combine these modes. We implement D-Streams in a system called Spark Streaming.

1 Introduction

Much of "big data" is received in real time, and is most valuable at its time of arrival. For example, a social network may wish to detect trending conversation topics in

faults and *stragglers* (slow nodes). Both problems are inevitable in large clusters [12], so streaming applications must recover from them quickly. Fast recovery is even *more* important in streaming than it was in batch jobs: while a 30 second delay to recover from a fault or straggler is a nuisance in a batch setting, it can mean losing the chance to make a key decision in a streaming setting.

Unfortunately, existing streaming systems have limited fault and straggler tolerance. Most distributed streaming systems, including Storm [37], TimeStream [33], MapReduce Online [11], and streaming databases [5, 9, 10], are based on a *continuous operator* model, in which long-running, stateful operators receive each record, update internal state, and send new records. While this model is quite natural, it makes it difficult to handle faults and stragglers.

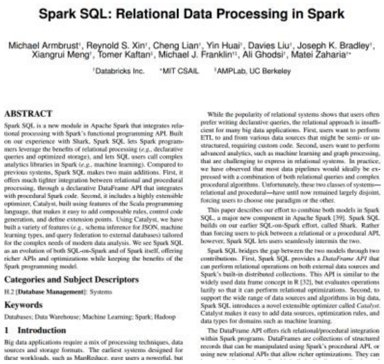
Specifically, given the continuous operator model, systems perform recovery through two approaches [20]: *replication*, where there are two copies of each node [5, 34], or *upstream backup*, where nodes buffer sent messages and replay them to a new copy of a failed node [33, 11, 37]. Neither approach is attractive in large clusters: replication costs 2× the hardware, while upstream backup takes a long time to recover, as the whole system must wait for a new node to serially rebuild the failed

```
TwitterUtils.createStream(...)
    .filter(_.getText.contains("Spark"))
    .countByWindow(Seconds(5))
```

History of Spark

Spark SQL

Seemlessly mix SQL queries with Spark programs.


Spark SQL: Relational Data Processing in Spark

Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, Matei Zaharia

Databricks Inc. MIT CSAIL AMPLab, UC Berkeley

ABSTRACT

Categories and Subject Descriptors

Keywords

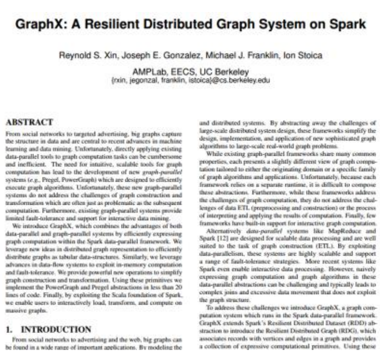
1 Introduction

```
sqlCtx = new HiveContext(sc)
results = sqlCtx.sql(
  "SELECT * FROM people")
names = results.map(lambda p:
p.name)
```


History of Spark

Spark GRAPHX

Analyze networks of nodes and edges using graph processing


GraphX: A Resilient Distributed Graph System on Spark

Reynold S. Xin, Joseph E. Gonzalez, Michael J. Franklin, Ion Stoica

AMPLab, EECS, UC Berkeley

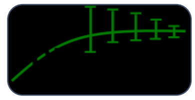
ABSTRACT

1. INTRODUCTION

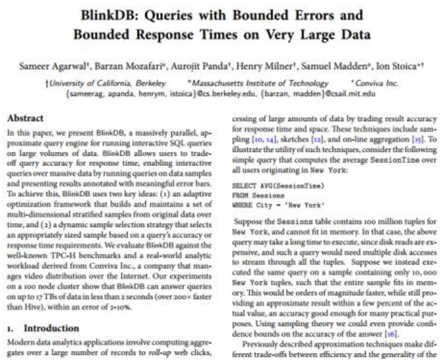
```
graph = Graph(vertices, edges)
messages =
spark.textFile("hdfs://...")
graph2 =
graph.joinVertices(messages) {
  (id, vertex, msg) => ...
}
```

https://amplab.cs.berkeley.edu/wp-content/uploads/2013/05/grades-graphx_with_fonts.pdf

History of Spark



SQL queries with Bounded Errors and Bounded Response Times



BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data

Sameer Agarwal†, Barzan Mozafari°, Aurojit Panda†, Henry Milner†, Samuel Madden°, Ion Stoica*†

†University of California, Berkeley °Massachusetts Institute of Technology *Conviva Inc.
{sameerag, apanda, henrym, istoica}@cs.berkeley.edu, {barzan, madden}@csail.mit.edu

Abstract

In this paper, we present BlinkDB, a massively parallel, approximate query engine for running interactive SQL queries on large volumes of data. BlinkDB allows users to trade-off query accuracy for response time, enabling interactive queries over massive data by running queries on data samples and presenting results annotated with meaningful error bars. To achieve this, BlinkDB uses two key ideas: (1) an adaptive optimization framework that builds and maintains a set of multi-dimensional stratified samples from original data over time, and (2) a dynamic sample selection strategy that selects an appropriately sized sample based on a query's accuracy or response time requirements. We evaluate BlinkDB against the well-known TPC-H benchmarks and a real-world analytic workload derived from Conviva Inc., a company that manages video distribution over the Internet. Our experiments on a 100 node cluster show that BlinkDB can answer queries on up to 17 TBs of data in less than 2 seconds (over 200× faster than Hive), within an error of 2-10%.

1. Introduction

Modern data analytics applications involve computing aggregates over a large number of records to *roll-up* web clicks, cessing of large amounts of data by trading result accuracy for response time and space. These techniques include sampling [10, 14], sketches [12], and on-line aggregation [15]. To illustrate the utility of such techniques, consider the following simple query that computes the average SessionTime over all users originating in New York:

```
SELECT AVG(SessionTime)
FROM Sessions
WHERE City = 'New York'
```

Suppose the Sessions table contains 100 million tuples for New York, and cannot fit in memory. In that case, the above query may take a long time to execute, since disk reads are expensive, and such a query would need multiple disk accesses to stream through all the tuples. Suppose we instead executed the same query on a sample containing only 10,000 New York tuples, such that the entire sample fits in memory. This would be orders of magnitude faster, while still providing an approximate result within a few percent of the actual value, an accuracy good enough for many practical purposes. Using sampling theory we could even provide confidence bounds on the accuracy of the answer [16].

Previously described approximation techniques make different trade-offs between efficiency and the generality of the

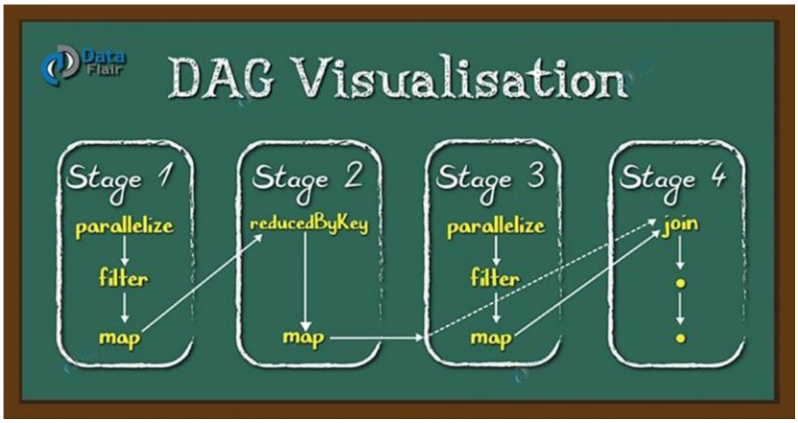
Queries with Time Bounds	Queries with Error Bounds
SELECT avg(sessionTime) **FROM** Table **WHERE** city='San Francisco' **WITHIN** 2 SECONDS	**SELECT** avg(sessionTime) **FROM** Table **WHERE** city='San Francisco' **ERROR** 0.1 **CONFIDENCE** 95.0%

https://www.cs.berkeley.edu/~sameerag/blinkdb_eurosys13.pdf

History of Spark

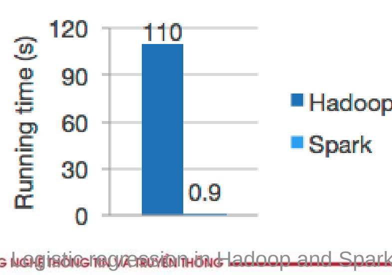
- Unlike the various specialized systems, Spark's goal was to *generalize* MapReduce to support new apps within same engine
- Two reasonably small additions are enough to express the previous models:
  - *fast data sharing*
  - *general DAGs*
- This allows for an approach which is more efficient for the engine, and much simpler for the end users

Directed Acyclic Graph - DAG



What is Apache Spark

- Spark is a unified analytics engine for large-scale data processing
- Speed: run workloads 100x faster
  - High performance for both batch and streaming data
  - Computations run in memory



What is Apache Spark

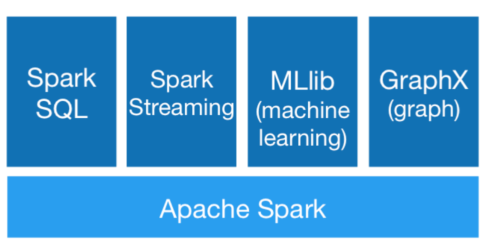
- Ease of Use: write applications quickly in Java, Scala, Python, R, SQL
  - Offer over 80 high-level operators
  - Use them interactively form Scala, Python, R, and SQL

```
df = spark.read.json("logs.json")
df.where("age > 21")
select("name.first").show()
```

Spark's Python DataFrame API
Read JSON files with automatic schema inference

What is Apache Spark

- Generality: combine SQL, Streaming, and complex analytics
  - Provide libraries including SQL and DataFrames, Spark Streaming, MLib, GraphX
  - Wide range of workloads e.g., batch applications, interactive algorithms, interactive queries, streaming



What is Apache Spark

- Run Everywhere:
  - run on Hadoop, Apache Mesos, Kubernetes, standalone or in the cloud.
  - access data in HDFS, Aluxio, Apache Cassandra, Apache Hbase, Apache Hive, etc.

Comparison between Hadoop and Spark

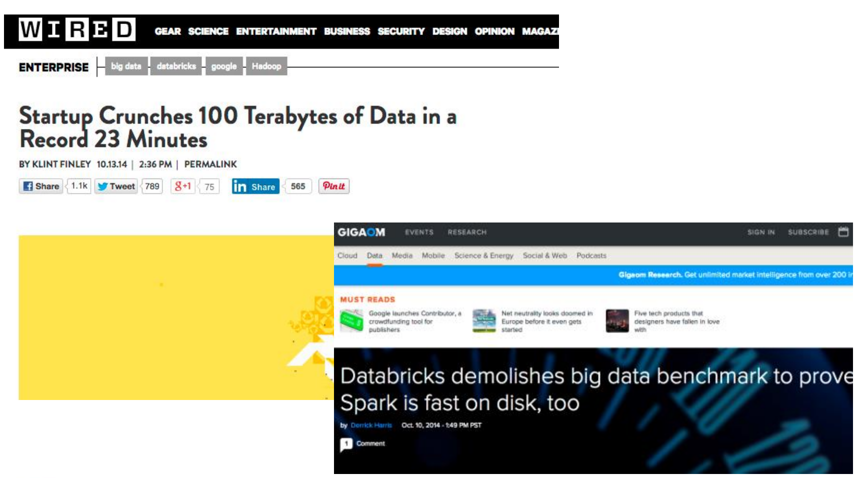
	hadoop MapReduce	Spark
Strengths	▪ Can collect any data ▪ Limitless in size	▪ Can work off any Hadoop collection ▪ Runs on Hadoop, or other clusters ▪ In-memory processing makes it very fast ▪ Supports Java, Scala, Python, and R*, and can be used with SQL.
Used for	▪ Initial data ingestion ▪ Data curation ▪ Large-scale "boil the ocean" analytics ▪ Data archiving	▪ Complex query processing of large amounts of data quickly ▪ Can handle ad hoc queries
Limitations	▪ MapReduce is hard to program ▪ Disk-based batch nature limits speed, agility.	▪ Limited only by processor speed, available memory, cores, and cluster size.

100TB Daytona Sort Competition

	Hadoop MR Record	Spark Record	Spark 1 PB
Data Size	102.5 TB	100 TB	1000 TB
Elapsed Time	72 mins	23 mins	234 mins
# Nodes	2100	206	190
# Cores	50400 physical	6592 virtualized	6080 virtualized
Cluster disk throughput	3150 GB/s (est.)	618 GB/s	570 GB/s
Sort Benchmark Daytona Rules	Yes	Yes	No
Network	dedicated data center, 10Gbps	virtualized (EC2) 10Gbps network	virtualized (EC2) 10Gbps network
Sort rate	**1.42 TB/min**	**4.27 TB/min**	**4.27 TB/min**
Sort rate/node	**0.67 GB/min**	**20.7 GB/min**	**22.5 GB/min**

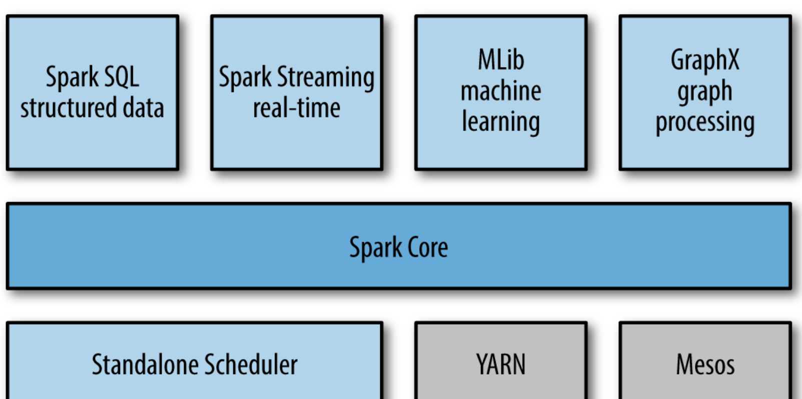
Spark sorted the same data **3X faster** using **10X fewer machines** than Hadoop MR in 2013.

All the sorting took place on disk (HDFS) without using Spark's in-memory cache!



Components of Stack

The Spark stack



The Spark stack

- Spark Core:
  - contain basic functionality of Spark including task scheduling, memory management, fault recovery, etc.
  - provide APIs for building and manipulating RDDs
- SparkSQL
  - allow querying structured data via SQL, Hive Query Language
  - allow combining SQL queries and data manipulations in Python, Java, Scala

The Spark stack

- Spark Streaming: enables processing of live streams of data via APIs
- Mlib:
  - contain common machine language functionality
  - provide multiple types of algorithms: classification, regression, clustering, etc.
- GraphX:
  - library for manipulating graphs and performing graph-parallel computations
  - extend Spark RDD API

The Spark stack

- Cluster Managers
  - Hadoop Yarn
  - Apache Mesos, and
  - Standalone Schedular (simple manager in Spark).

Resilient Distributed Dataset – RDD

- RDD Basics
- Creating RDDs
- RDD Operations
- Common Transformation and Actions
- Persistence (Caching)

RDD Basics

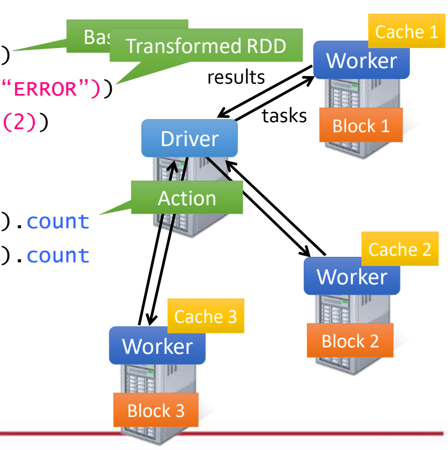
- RDD:
  - Immutable distributed collection of objects
  - Split into multiple partitions => can be computed on different nodes
- All work in Spark is expressed as
  - creating new RDDs
  - transforming existing RDDs
  - calling actions on RDDs

Example

Load error messages from a log into memory, then interactively search for various patterns

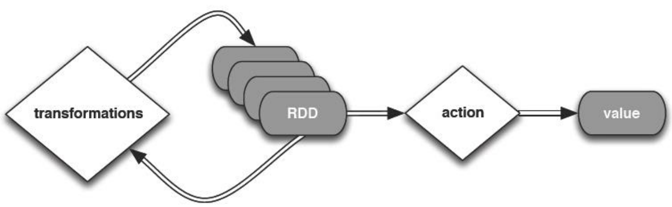
```
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
messages = errors.map(_.split('\t')(2))
cachedMsgs = messages.cache()

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



RDD Basics

- Two types of operations: transformations and actions
- Transformations: construct a new RDD from a previous one e.g., filter data
- Actions: compute a result base on an RDD e.g., count elements, get first element



45

Transformations

- Create new RDDs from existing RDDs
- Lazy evaluation
  - See the whole chain of transformations
  - Compute just the data needed
- Persist contents:
  - persist an RDD in memory to reuse it in future
  - persist RDDs on disk is possible

Typical works of a Spark program

1. Create some input RDDs form external data
2. Transform them to define new RDDs using transformations like filter()
3. Ask Spark to persist() any intermediate RDDs that will need to be reused
4. Launch actions such as count(), first() to kick off a parallel computation

Resilient Distributed Dataset – RDD

- RDD Basics
- Creating RDDs
- RDD Operations
- Common Transformation and Actions
- Persistence (Caching)

Two ways to create RDDs

1. Parallelizing a collection: uses parallelize()

- Python

```
lines = sc.parallelize(["pandas", "i like pandas"])
```

- Scala

```
val lines = sc.parallelize(List("pandas", "i like
pandas"))
```

- Java

```
JavaRDD<String> lines =
sc.parallelize(Arrays.asList("pandas", "i like
pandas"));
```


Two ways to create RDDs

2. Loading data from external storage

- Python

```
lines = sc.textFile("/path/to/README.md")
```

- Scala

```
val lines = sc.textFile("/path/to/README.md")
```

- Java

```
JavaRDD<String> lines =
sc.textFile("/path/to/README.md");
```

Resilient Distributed Dataset – RDD

- RDD Basics
- Creating RDDs
- RDD Operations
- Common Transformation and Actions
- Persistence (Caching)

RDD Operations

- Two types of operations
  - Transformations: operations that return a new RDDs e.g., map(), filter()
  - Actions: operations that return a result to the driver program or write it to storage such as count(), first()
- Treated differently by Spark
  - Transformation: lazy evaluation
  - Action: execution at any time

Transformation

- Example 1. Use filter()
- Python

```
inputRDD = sc.textFile("log.txt")
errorsRDD = inputRDD.filter(lambda x: "error" in x)
```

- Scala

```
val inputRDD = sc.textFile("log.txt")
val errorsRDD = inputRDD.filter(line =>
line.contains("error"))
```

- Java

```
JavaRDD<String> inputRDD = sc.textFile("log.txt");
JavaRDD<String> errorsRDD = inputRDD.filter(
new Function<String, Boolean>() {
public Boolean call(String x) {
       return x.contains("error"); }}
  });
```


Transformation

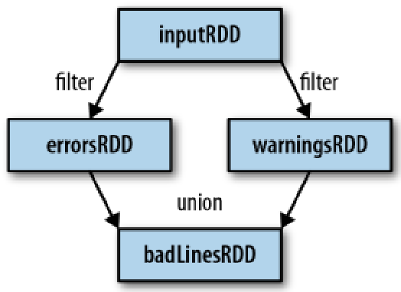
- filter()
  - does not change the existing *inputRDD*
  - returns a pointer to an entirely new RDD
  - *inputRDD* still can be reused
- union()

```
errorsRDD = inputRDD.filter(lambda x: "error" in x)
warningsRDD=inputRDD.filter(lambda x: "warning" in x)
badLinesRDD = errorsRDD.union(warningsRDD)
```

- transformations can operate on any number of input RDDs

Transformation

- Spark keeps track dependencies between RDDs, called the lineage graph
- Allow recovering lost data



Actions

- Example. count the number of errors
- Python

```
print "Input had " + badLinesRDD.count() + " concerning lines"
print "Here are 10 examples:"
for line in badLinesRDD.take(10):
print line
```

- Scala

```
println("Input had " + badLinesRDD.count() + " concerning lines")
println("Here are 10 examples:")
badLinesRDD.take(10).foreach(println)
```

- Java

```
System.out.println("Input had " + badLinesRDD.count() + " concerning
lines")
System.out.println("Here are 10 examples:")
for (String line: badLinesRDD.take(10)) {
System.out.println(line);
}
```

Resilient Distributed Dataset – RDD

- RDD Basics
- Creating RDDs
- RDD Operations
- Common Transformation and Actions
- Persistence (Caching)

RDD Basics

Transformations	Actions
map	reduce
flatMap	collect
filter	count
sample	save
union	lookupKey
groupByKey	…
reduceByKey	
join	
cache	
…	

Transformations

transformation	description
map(*func*)	return a new distributed dataset formed by passing each element of the source through a function *func*
filter(*func*)	return a new dataset formed by selecting those elements of the source on which *func* returns true
flatMap(*func*)	similar to map, but each input item can be mapped to 0 or more output items (so *func* should return a Seq rather than a single item)
sample(*withReplacement, fraction, seed*)	sample a fraction *fraction* of the data, with or without replacement, using a given random number generator *seed*
union(*otherDataset*)	return a new dataset that contains the union of the elements in the source dataset and the argument
distinct([*numTasks*]))	return a new dataset that contains the distinct elements of the source dataset

Transformations

transformation	description
groupByKey([*numTasks*])	when called on a dataset of `(K, V)` pairs, returns a dataset of `(K, Seq[V])` pairs
reduceByKey(*func*, [*numTasks*])	when called on a dataset of `(K, V)` pairs, returns a dataset of `(K, V)` pairs where the values for each key are aggregated using the given reduce function
sortByKey([*ascending*], [*numTasks*])	when called on a dataset of `(K, V)` pairs where `K` implements `Ordered`, returns a dataset of `(K, V)` pairs sorted by keys in ascending or descending order, as specified in the boolean ascending argument
join(*otherDataset*, [*numTasks*])	when called on datasets of type `(K, V)` and `(K, W)`, returns a dataset of `(K, (V, W))` pairs with all pairs of elements for each key
cogroup(*otherDataset*, [*numTasks*])	when called on datasets of type `(K, V)` and `(K, W)`, returns a dataset of `(K, Seq[V], Seq[W])` tuples – also called `groupWith`
cartesian(*otherDataset*)	when called on datasets of types `T` and `U`, returns a dataset of `(T, U)` pairs (all pairs of elements)

SOICT VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

Actions

action	description
reduce(*func*)	aggregate the elements of the dataset using a function *func* (which takes two arguments and returns one), and should also be commutative and associative so that it can be computed correctly in parallel
collect()	return all the elements of the dataset as an array at the driver program – usually useful after a filter or other operation that returns a sufficiently small subset of the data
count()	return the number of elements in the dataset
first()	return the first element of the dataset – similar to *take(1)*
take(*n*)	return an array with the first *n* elements of the dataset – currently not executed in parallel, instead the driver program computes all the elements
takeSample(*withReplacement*, *fraction*, *seed*)	return an array with a random sample of *num* elements of the dataset, with or without replacement, using the given random number generator seed

SOICT VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

61

Actions

action	description
saveAsTextFile(*path*)	write the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem, HDFS or any other Hadoop-supported file system. Spark will call `toString` on each element to convert it to a line of text in the file
saveAsSequenceFile(*path*)	write the elements of the dataset as a Hadoop `SequenceFile` in a given path in the local filesystem, HDFS or any other Hadoop-supported file system. Only available on RDDs of key-value pairs that either implement Hadoop's `Writable` interface or are implicitly convertible to `Writable` (Spark includes conversions for basic types like `Int`, `Double`, `String`, etc).
countByKey()	only available on RDDs of type `(K, V)`. Returns a \`Map\` of `(K, Int)` pairs with the count of each key
foreach(*func*)	run a function *func* on each element of the dataset – usually done for side effects such as updating an accumulator variable or interacting with external storage systems

62

Resilient Distributed Dataset – RDD

- RDD Basics
- Creating RDDs
- RDD Operations
- Common Transformation and Actions
- Persistence (Caching)

Persistence levels

Level	Space used	CPU time	In memory	On disk	Comments
MEMORY_ONLY	High	Low	Y	N	
MEMORY_ONLY_SER	Low	High	Y	N	
MEMORY_AND_DISK	High	Medium	Some	Some	Spills to disk if there is too much data to fit in memory.
MEMORY_AND_DISK_SER	Low	High	Some	Some	Spills to disk if there is too much data to fit in memory. Stores serialized representation in memory.
DISK_ONLY	Low	High	N	Y	

Persistence

- Example

```
val result = input.map(x => x * x)
result.persist(StorageLevel.DISK_ONLY)
println(result.count())
println(result.collect().mkString(","))
```


Acknowledgement and References

Books:

- Holden Karau, Andy Konwinski, Patrick Wendell & Matei Zaharia. Learning Spark. Oreilly
- TutorialsPoint. Spark Core Programming

Slides:

- Paco Nathan. Intro to Apache Spark
- Harold Liu. Berkely Data Analytics Stack
- DataBricks. Intro to Spark Development

Q&A

67

ĐẠI HỌC BÁCH KHOA

25 YEARS ANNIVERSARY

SOICT

HA NOI UNIVERSITY OF SCIENCE AND TECHNOLOGY
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

68

VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

Thank you for your attentions!

soict.hust.edu.vn/ fb.com/groups/soict

69

ĐẠI HỌC BÁCH KHOA HÀ NỘI
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

70