Big Data, Spark, & Databricks

Andy Konwinski

Nov 14, 2017, CS285 Guest Lecture, Stanford

Many slides re-used from:

Burak Yuvaz, Databricks. Berkeley CS186 Spring 2017
Reynold Xin, Databricks. Berkeley CS186 2016
Matei Zaharia, Databricks. Processing Big Data with Small Programs
Michael Franklin, SQL, NoSQL, NewSQL?. Berkeley CS186 2013

Who Am I

Technical Cofounder at Databricks

Co-creator of Apache Mesos

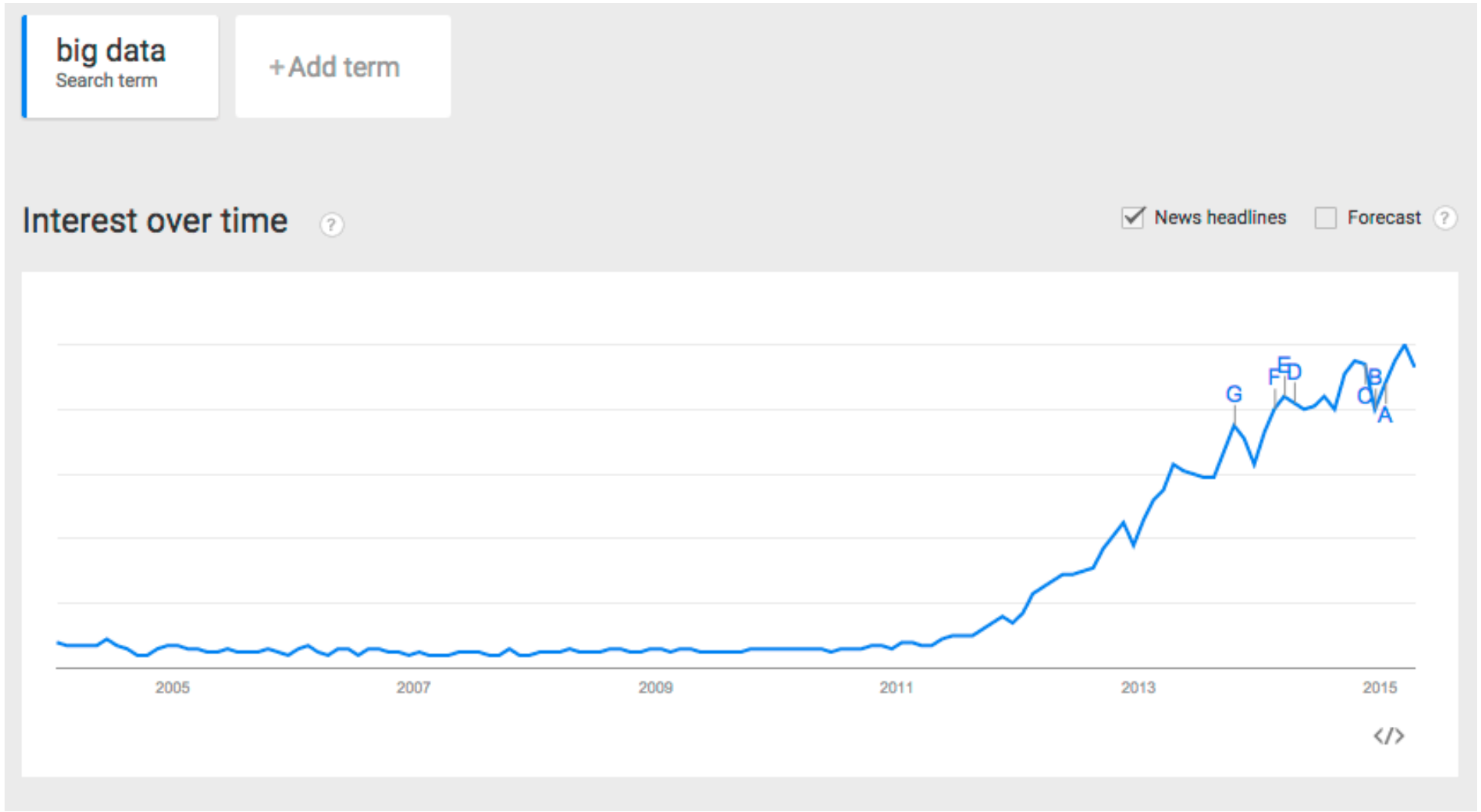
Apache Spark Committer

PhD CS, UC Berkeley; BS CS, U of Wisconsin

Agenda

- What is Big Data? What is it used for?
- Brief history of Big Data analytics
- Apache Spark & Databricks
- Demo & discussion

What is Big Data?


big data
Search term
+ Add term
Interest over time
News headlines
Forecast
G
F
E
D
C
B
A
2005
2007
2009
2011
2013
2015

Gartner's Definition

~~"Big data" is high~~-**volume**, -**velocity** and -**variety** ~~information assets that demand cost-effective, innovative forms of information processing for enhanced insight and decision making.~~

3+1 Vs of Big Data

Volume: data size

Velocity: rate of data coming in

Variety (most important V): data sources, formats, workloads

Veracity: data cleanliness, completeness, accuracy

3+1+N Vs of Big Data

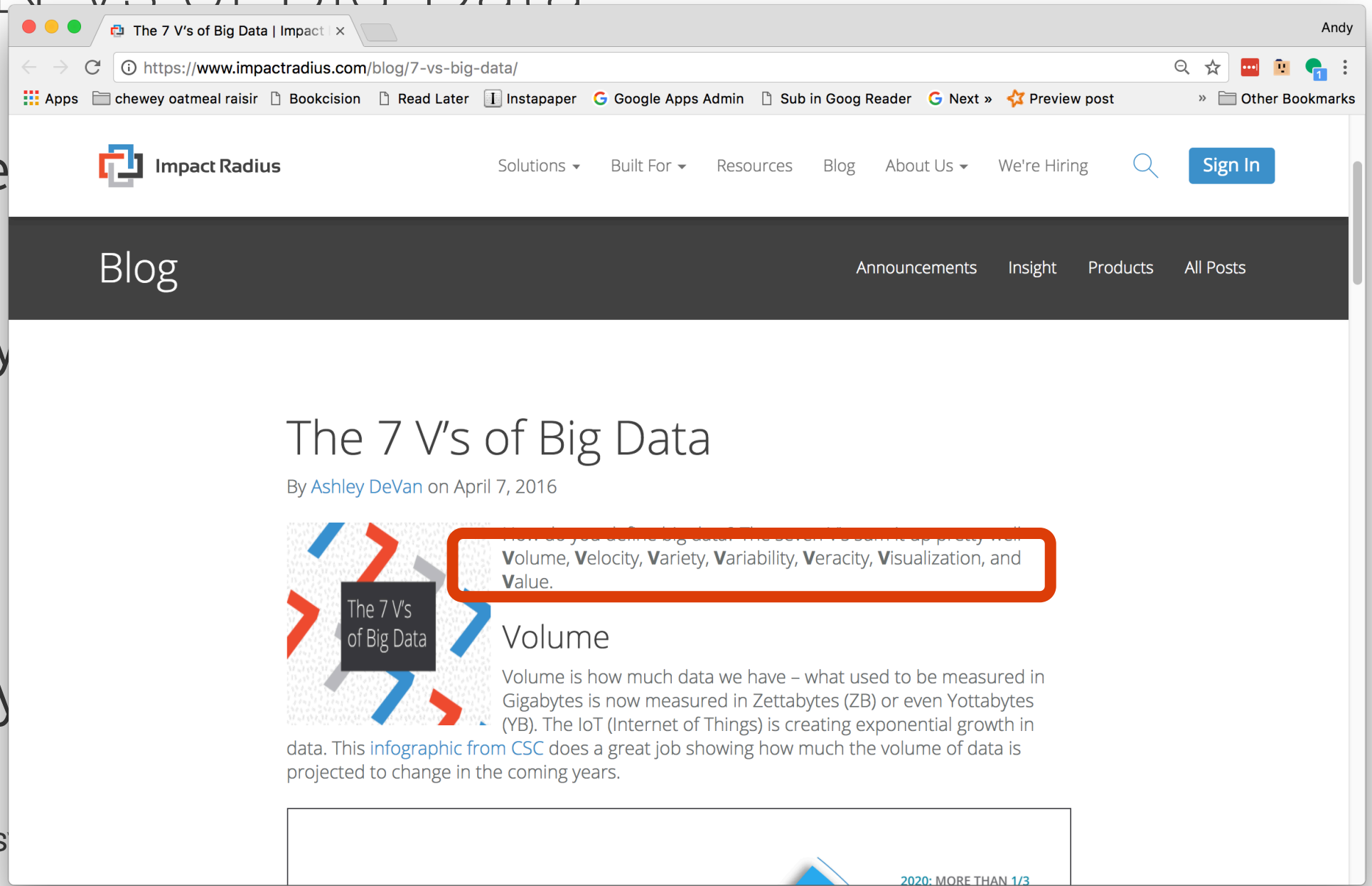
Volume

Velocity

Variety

Veracity

databricks



What is it used for?
Why is it valuable?

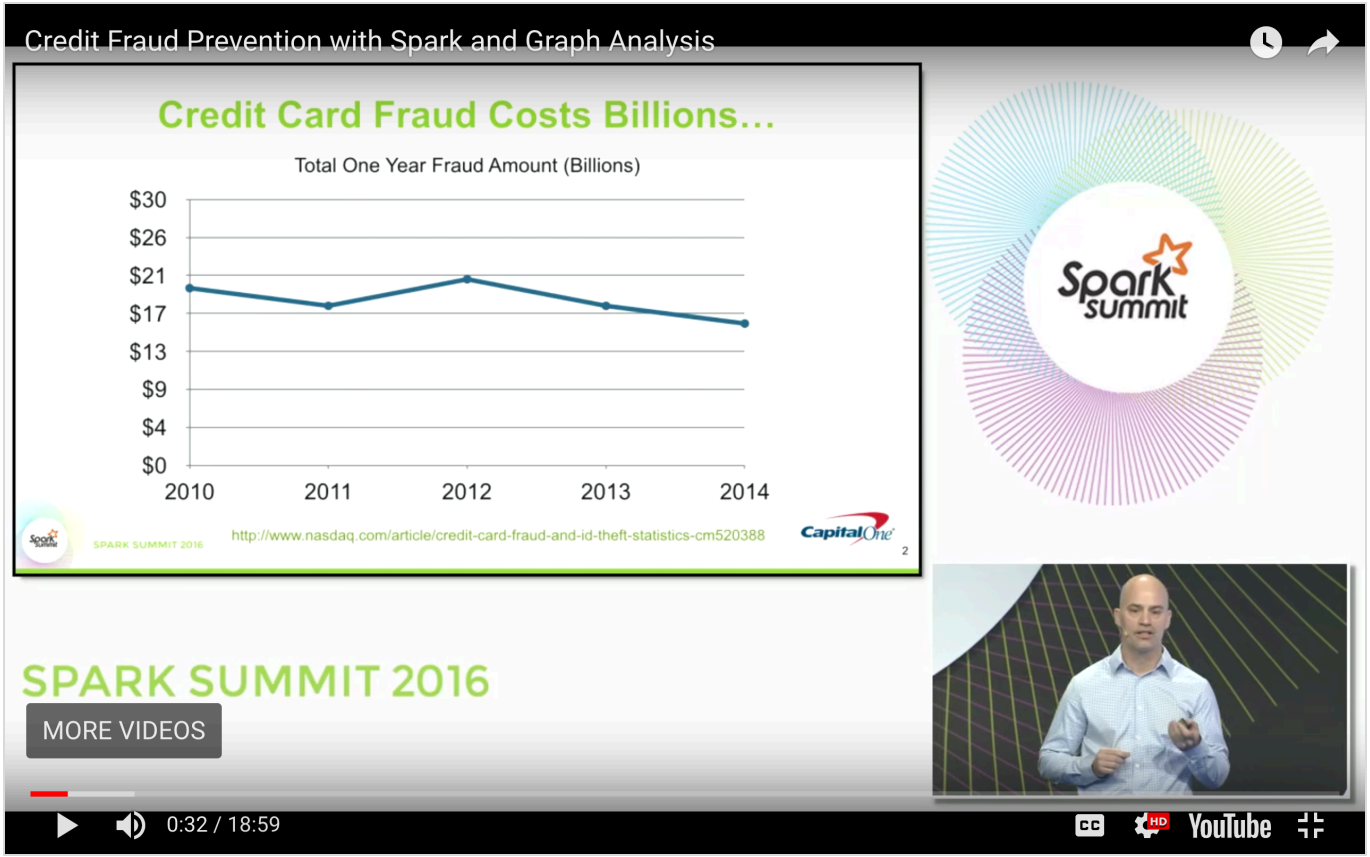
Motivation: Business Use Cases

Security

- Fraud detection
- Anomaly detection for breaches

Examples:

- ~$16B/yr lost to CC fraud
- ~15M victims/yr
- CapitalOne detects fraud in real-time



databricks

Motivation: Business Use Cases

Machine Learning: Recommendation Systems

- Recommend products/social connections/features

Examples:

- Riot Games – League of Legends
  - In-store purchase of outfits/cosmetics for characters. A main profit driver
  - Identifying unsportsperson like conduct in chat rooms & warning or banning
- Hotels.com – recommend hotels/car rentals

Agenda

- What is Big Data? What is it used for?
- **Brief history of Big Data analytics**
- Apache Spark & Databricks
- Demo & discussion

History & Roots

2003 - Google published GFS

2004 - Google published MapReduce

2006 - Hadoop created

2009 - Spark created (as part of Mesos project)

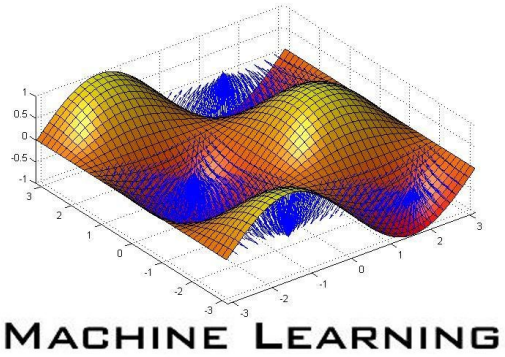
2013 – Databricks created

Challenges Google faced

Data size growing (volume & velocity)

Complexity of analysis increasing (variety)

- Massive ETL (web crawling), Machine learning



Why didn’t Google just use an existing database?

Why didn't Google just use databases?

No databases at the time worked at that **scale**

It **cost** less to build than buy (vendors charge by TB or machine)

A lot of **unstructured data**: web pages, images, videos

SQL not right **programming model** for their tasks

The Big Data solution

Single machine can’t process or store all the data

Only solution is to **distribute** general storage & processing over clusters.

Data-Parallel Models

Restrict the programming interface so that the system can do more automatically

"Here's an operation, run it on all of the data"

- I don't care *where* it runs (you schedule that)
- In fact, feel free to run it *twice* on different nodes
- **Similar to "declarative programming" in databases**

2003

The Google File System

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung
Google*

ABSTRACT

We have designed and implemented the Google File System, a scalable distributed file system for large distributed data-intensive applications. It provides fault tolerance while running on inexpensive commodity hardware, and it delivers high aggregate performance to a large number of clients.

While sharing many of the same goals as previous distributed file systems, our design has been driven by observations of our application workloads and technological environment, both current and anticipated, that reflect a marked departure from some earlier file system assumptions. This has led us to reexamine traditional choices and explore radically different design points.

The file system has successfully met our storage needs. It is widely deployed within Google as the storage platform for the generation and processing of data used by our service as well as research and development efforts that require large data sets. The largest cluster to date provides hundreds of terabytes of storage across thousands of disks on over a thousand machines, and it is concurrently accessed by hundreds of clients.

In this paper, we present file system interface extensions designed to support distributed applications, discuss many aspects of our design, and report measurements from both micro-benchmarks and real world use.

1. INTRODUCTION

We have designed and implemented the Google File System (GFS) to meet the rapidly growing demands of Google's data processing needs. GFS shares many of the same goals as previous distributed file systems such as performance, scalability, reliability, and availability. However, its design has been driven by key observations of our application workloads and technological environment, both current and anticipated, that reflect a marked departure from some earlier file system design assumptions. We have reexamined traditional choices and explored radically different points in the design space.

First, component failures are the norm rather than the exception. The file system consists of hundreds or even thousands of storage machines built from inexpensive commodity parts and is accessed by a comparable number of client machines. The quantity and quality of the components virtually guarantee that some are not functional at any given time and some will not recover from their current failures. We have seen problems caused by application bugs, operating system bugs, human errors, and the failures of disks, memory, connectors, networking, and power supplies. Therefore, constant monitoring, error detection, fault tolerance, and automatic recovery must be integral to the system.

Second, files are huge by traditional standards. Multi-GB

2004

MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

Google, Inc.

Abstract

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a *map* function that processes a key/value pair to generate a set of intermediate key/value pairs, and a *reduce* function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model, as shown in the paper.

Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication. This allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system.

Our implementation of MapReduce runs on a large cluster of commodity machines and is highly scalable:

given day, etc. Most such computations are conceptually straightforward. However, the input data is usually large and the computations have to be distributed across hundreds or thousands of machines in order to finish in a reasonable amount of time. The issues of how to parallelize the computation, distribute the data, and handle failures conspire to obscure the original simple computation with large amounts of complex code to deal with these issues.

As a reaction to this complexity, we designed a new abstraction that allows us to express the simple computations we were trying to perform but hides the messy details of parallelization, fault-tolerance, data distribution and load balancing in a library. Our abstraction is inspired by the *map* and *reduce* primitives present in Lisp and many other functional languages. We realized that most of our computations involved applying a *map* operation to each logical "record" in our input in order to compute a set of intermediate key/value pairs, and then applying a *reduce* operation to all the values that shared the same key, in order to combine the derived data ap-

Pair and share: what is MapReduce?

Problems with MapReduce that led to Spark

1. Programming model
2. Performance

Agenda

- What is Big Data? What is it used for?
- Brief history of Big Data analytics
- **Apache Spark & Databricks**
- Demo & discussion

MapReduce Challenge #1: Programming model

Most real applications require multiple MR steps

- Google indexing pipeline: 21 steps
- Analytics queries (e.g. groupby + count): 2-5 steps
- Iterative algorithms (e.g. PageRank): 10's of steps

Multi-step jobs create spaghetti code

- 21 MR steps -> 21 mapper and reducer classes
- Lots of boilerplate code per step

MR Jobs as execution primitives compiled by higher layer abstractions...

facebook

YAHOO!

In reality, 90+% of MR jobs are generated by Hive SQL

Programming model

Full Google WordCount:

```
#include "mapreduce/mapreduce.h"

// User's map function
class SplitWords: public Mapper {
  public:
  virtual void Map(const MapInput&
input)
  {
    const string& text =
input.value();
    const int n = text.size();
    for (int i = 0; i < n; ) {
      // Skip past leading whitespace
      while (i < n &&
isspace(text[i]))
        i++;
      // Find word end
      int start = i;
      while (i < n &&
!isspace(text[i]))
        i++;
      if (start < i)
        Emit(text.substr(
            start,i-start),"1");
    }
  }
};

REGISTER_MAPPER(SplitWords);
```

```
// User's reduce functi
class Sum: public Reduc
  public:
  virtual void Reduce(R
input)
  {
    // Iterate over all
the
    // same key and add
    int64 value = 0;
    while (!input->done
      value += StringTo
                input->
      input->NextValue();
    }
    // Emit sum for input->key()
    Emit(IntToString(value));
  }
};

REGISTER_REDUCER(Sum);

int main(int argc, char** argv) {
  ParseCommandLineFlags(argc, argv);
```

```
                                    out->set_num_tasks(100);
                                    out->set_format("text");
                                    out->set_reducer_class("Sum");

                                    // Do partial sums within map
                                    out->set_combiner_class("Sum");

                                    // Tuning parameters
                                    spec.set_machines(2000);
                                    spec.set_map_megabytes(100);
                                    spec.set_reduce_megabytes(100);

                                    // Now run it
                                    MapReduceResult result;
                                    if (!MapReduce(spec, &result))
                                  abort();
                                    return 0;
                                  }
```

Spark WordCount:

```
val file = spark.textFile("hdfs://...")
val counts = file.flatMap(line => line.split(" "))
                 .map(word => (word, 1))
                 .reduceByKey(_ + _)

counts.save("out.txt")
```

MapReduce Challenge #2: Performance

Each MR job writes all output to disk

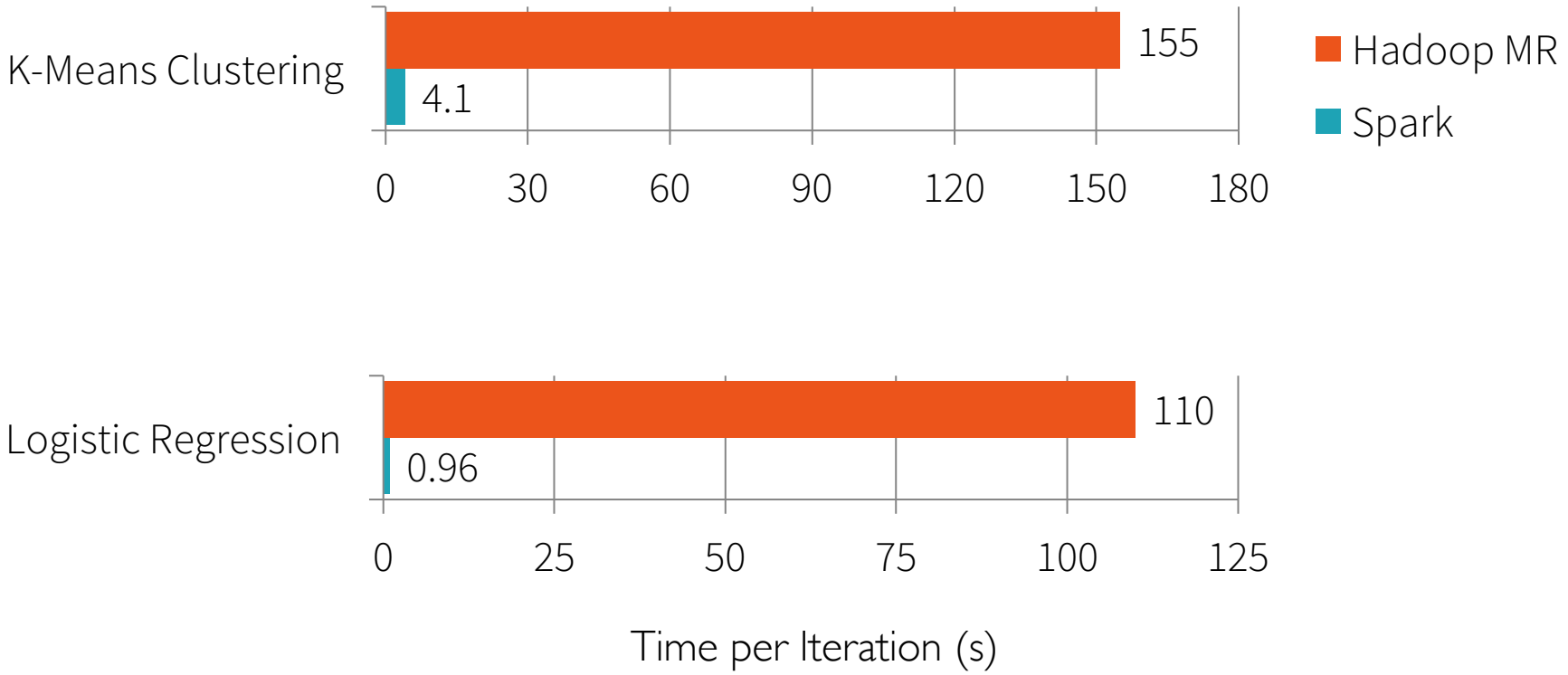
Lack of more primitives such as data broadcast

Spark provides data caching and broadcast primitives

Many many other performance optimizations

- e.g. in-memory columnar storage
- 1.6+ has relational optimizer

Performance



Spark history & overview

Started in Berkeley in 2010; donated to Apache Foundation in 2013

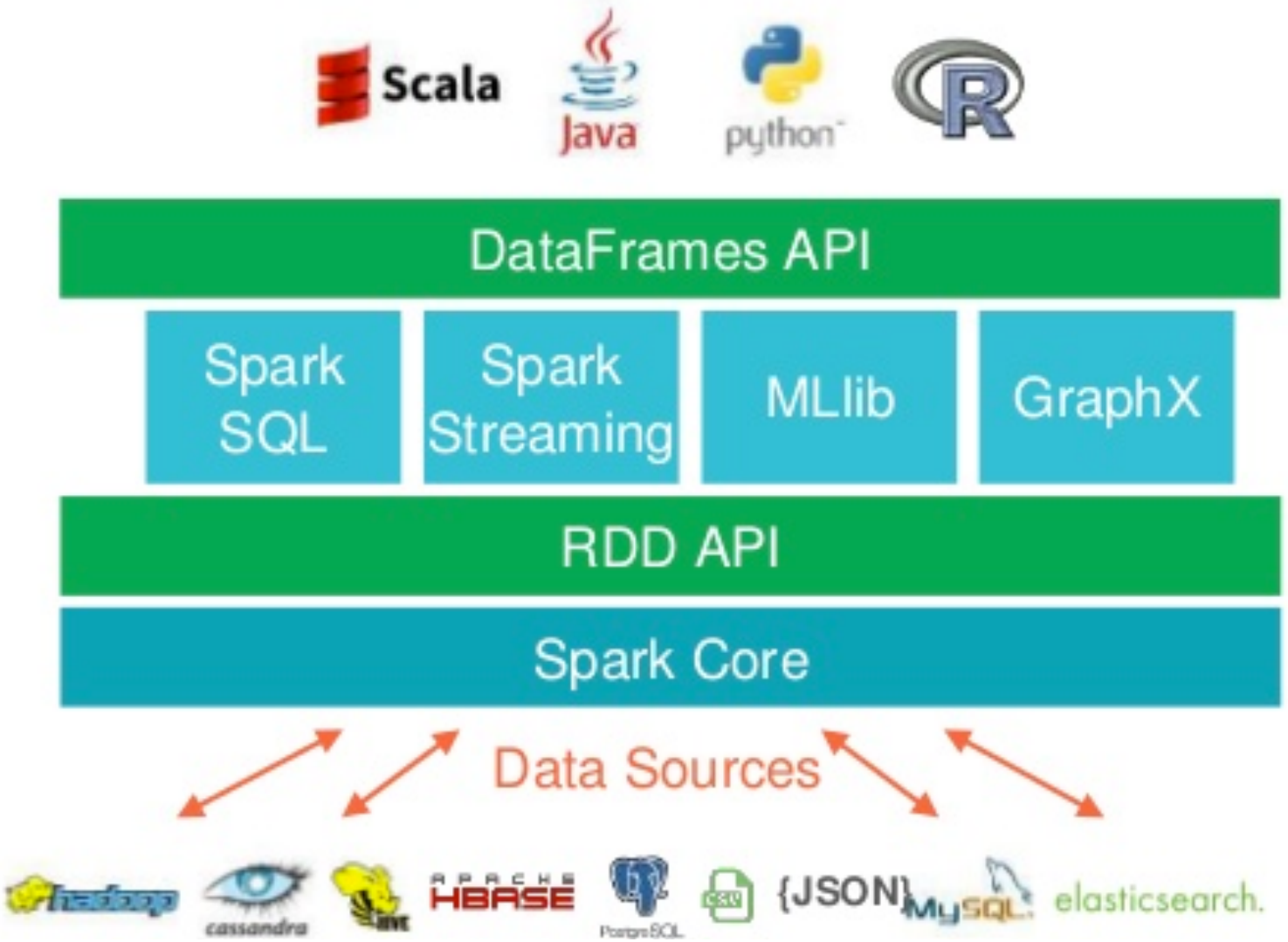
Programmability: Domain Specific Language in Scala / Java / Python / R

- Functional transformations on collections
- 5 – 10X less code than MR
- Interactive use from Scala / Python REPL

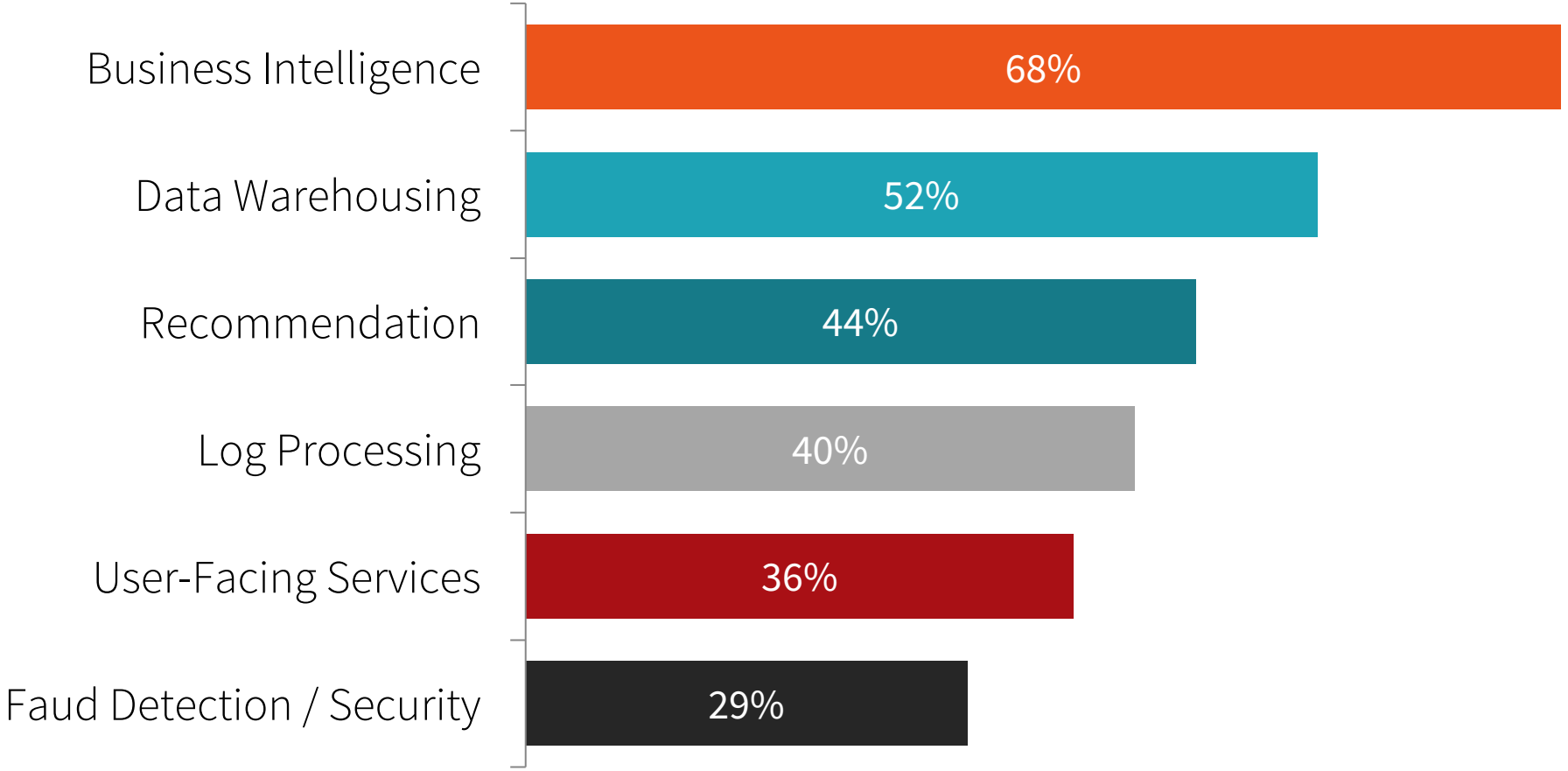
Performance:

- General DAG of tasks (i.e. multi-stage MR)
- Richer primitives: in-memory cache, torrent broadcast, etc
- Can run 10 – 100X faster than MR

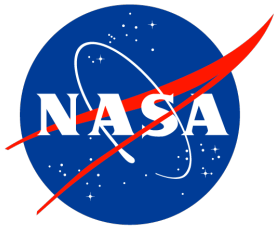
Spark stack

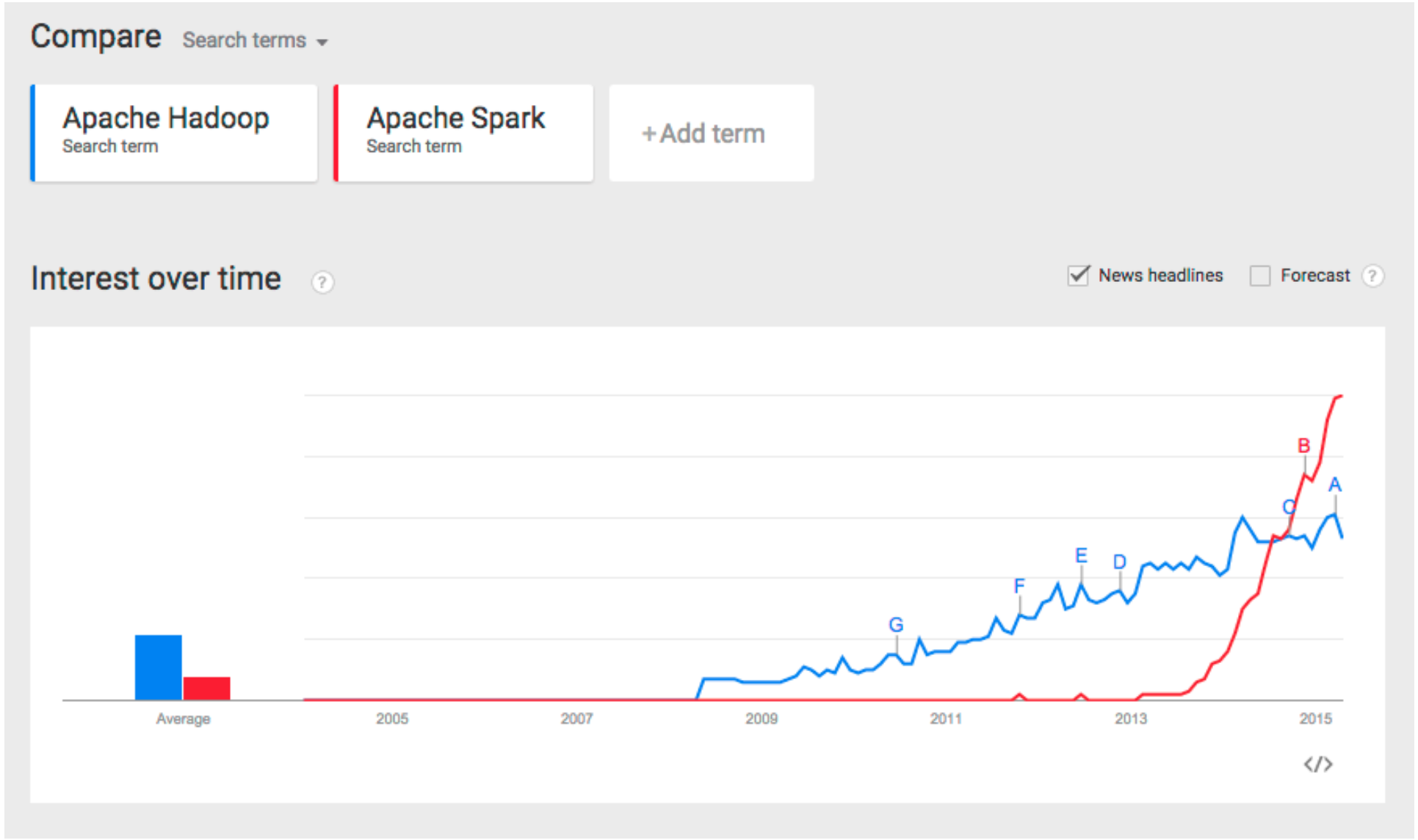

Scala
Java
python
R
DataFrames API
Spark SQL
Spark Streaming
MLlib
GraphX
RDD API
Spark Core
Data Sources
hadoop
cassandra
hive
APACHE HBASE
PostgreSQL
CSV
{JSON}
MySQL
elasticsearch.

Top Applications



Spark has 1000s of users





Note： not a scientific comparison.

“Spark is the Taylor Swift of big data software.”

- Derrick Harris, Fortune

When should I use Spark?

When Should I Use Spark?

I want to know how much of my product was sold per country.

When Should I Use Spark?

I want to know how much of my product was sold per country.

When Should I Use Spark?

I want to serve the user preferences of a customer when they click "Settings" on my website.

When Should I Use Spark?

I want to serve the user preferences of a customer when they click "Settings" on my website.

When Should I Use Spark?

I want to extract features from thousands of newspaper articles and classify whether an article belongs in the "Economy" or "Politics" section.

When Should I Use Spark?

I want to extract features from thousands of newspaper articles and classify whether an article belongs in the "Economy" or "Politics" section.

When Should I Use Spark?

Given a map, I want to figure out the fastest way to get from point A to point B.

When Should I Use Spark?

Given a map, I want to figure out the fastest way to get from point A to point B.

When Should I Use Spark?

I want to centralize all the data my sensors are generating all around the world. I would like to push my data somewhere from these sensors.

When Should I Use Spark?

I want to centralize all the data my sensors are generating all around the world. I would like to push my data somewhere from these sensors.

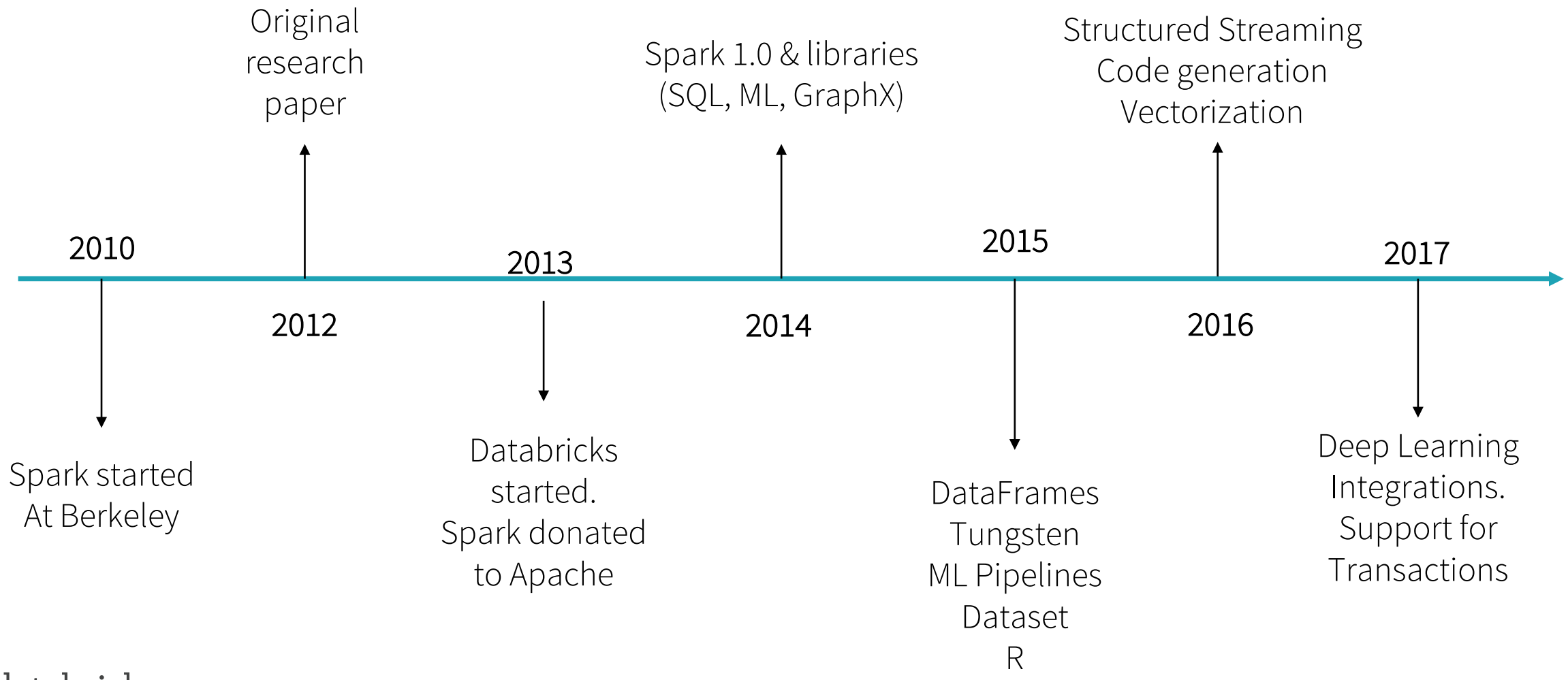
When Should I Use Spark?

I have sensors generating data every 15 seconds. I want to analyze the data but first enrich it with regional information and aggregate per region in 15 minute windows.

When Should I Use Spark?

I have sensors generating data every 15 seconds. I want to analyze the data but first enrich it with regional information and aggregate per region in 15 minute windows.

Spark's life story

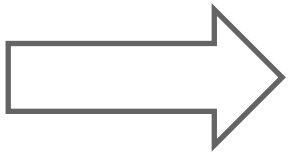


databricks

RDDs and DataFrames
...and how to use them.

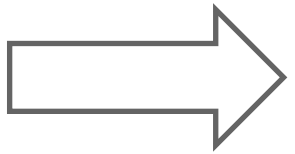
Spark Core API Evolution

Early adopters



Users

Understand
MapReduce
& functional APIs



Data Scientists
Statisticians
R users
PyData

Resilient Distributed Datasets (RDDs)

DataFrames

DataFrames in Spark

Distributed abstraction for tabular data in Java, Python, R, Scala

Similar APIs as single-node tools (Pandas, R), thus easy to learn

```
> head(filter(df, df$waiting < 50))  # an example in R
##  eruptions waiting
##1     1.750      47
##2     1.750      47
##3     1.867      48
```

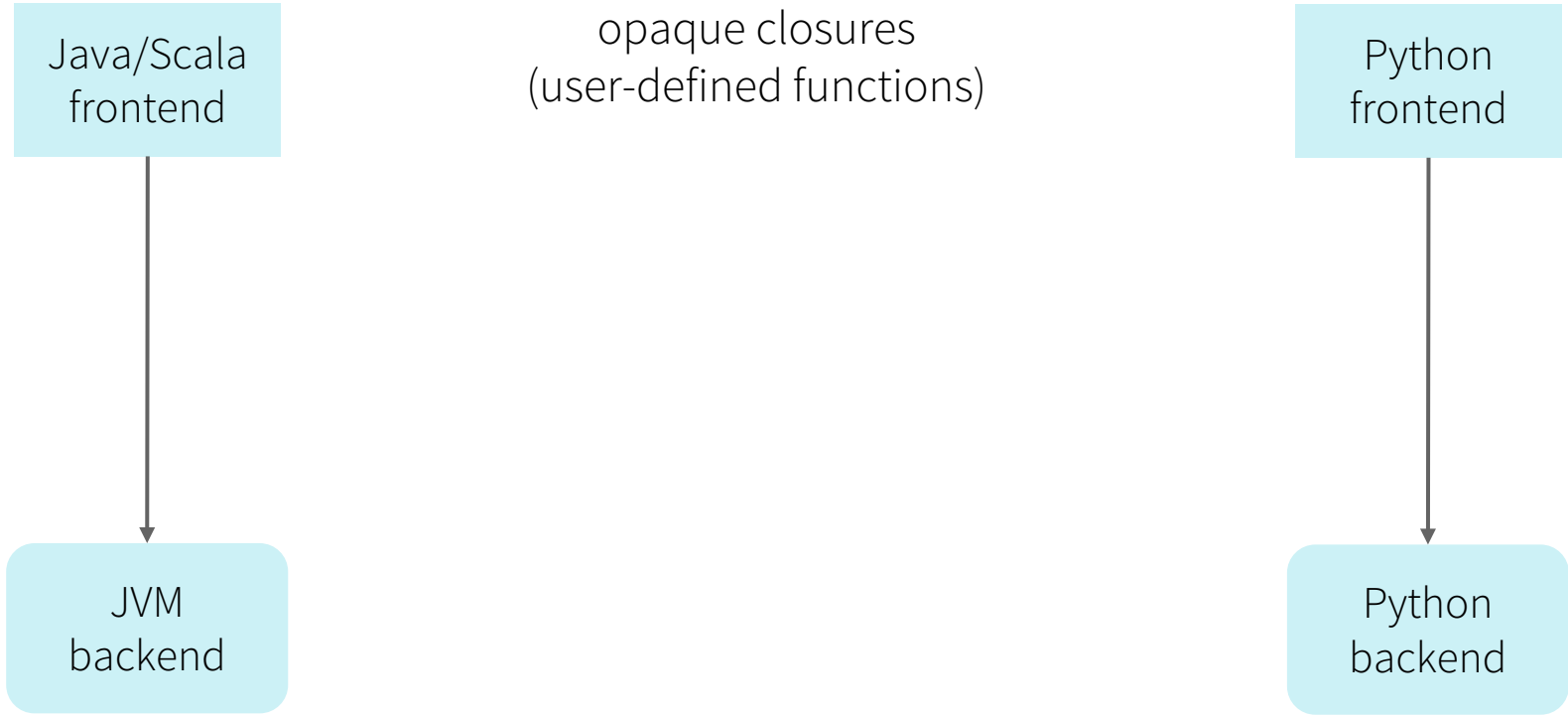
RDD

```
pdata.map(lambda x: (x.dept, [x.age, 1])) \
  .reduceByKey(lambda x, y: [x[0] + y[0], x[1] + y[1]]) \
  .map(lambda x: [x[0], x[1][0] / x[1][1]]) \
  .collect()
```

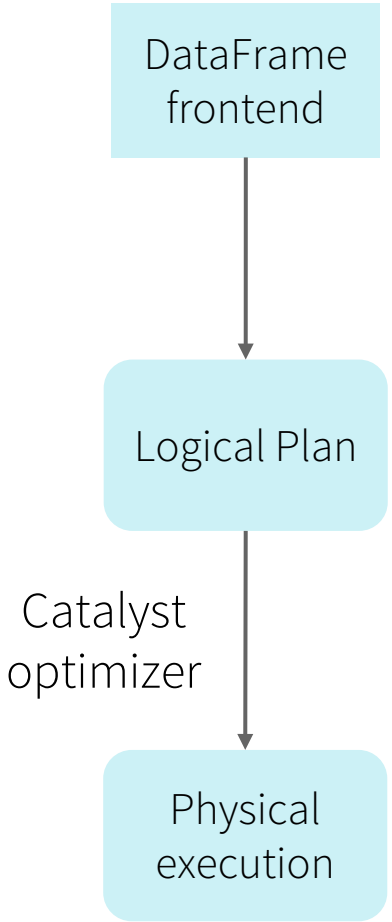
DataFrame

```
data.groupBy("dept").avg("age")
```

Spark RDD Execution

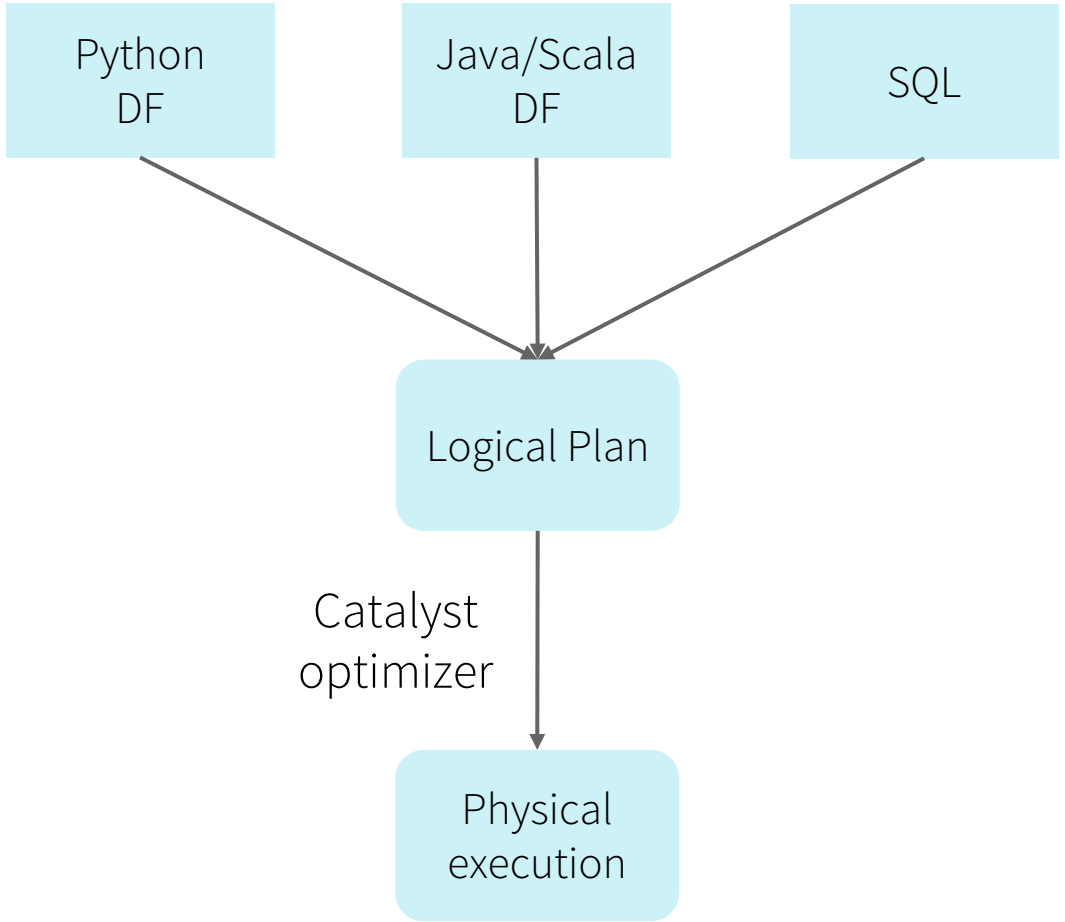


Spark DataFrame Execution



Intermediate representation for computation

Spark DataFrame Execution



Simple wrappers to create logical plan

Intermediate representation for computation

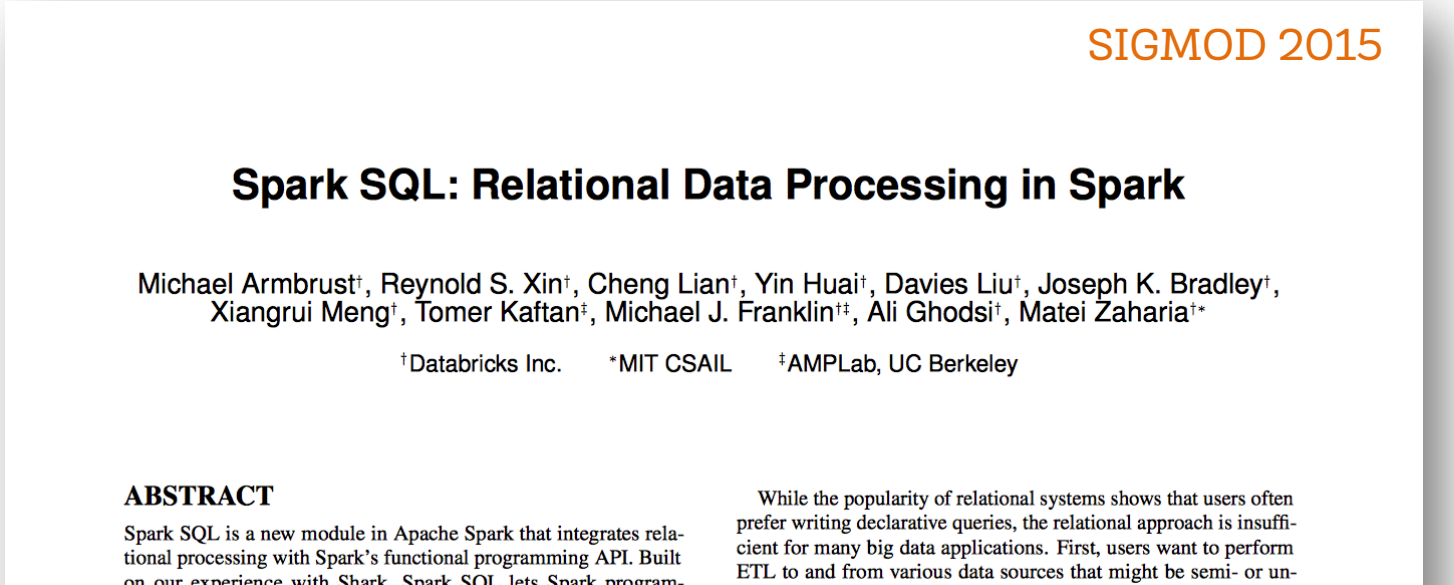
DataFrames and Spark SQL

Efficient library for structured data (data with a known schema)

- Two interfaces: SQL for analysts + apps, DataFrames for programmers

Optimized computation and storage, similar to RDBMS

SIGMOD 2015



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†*

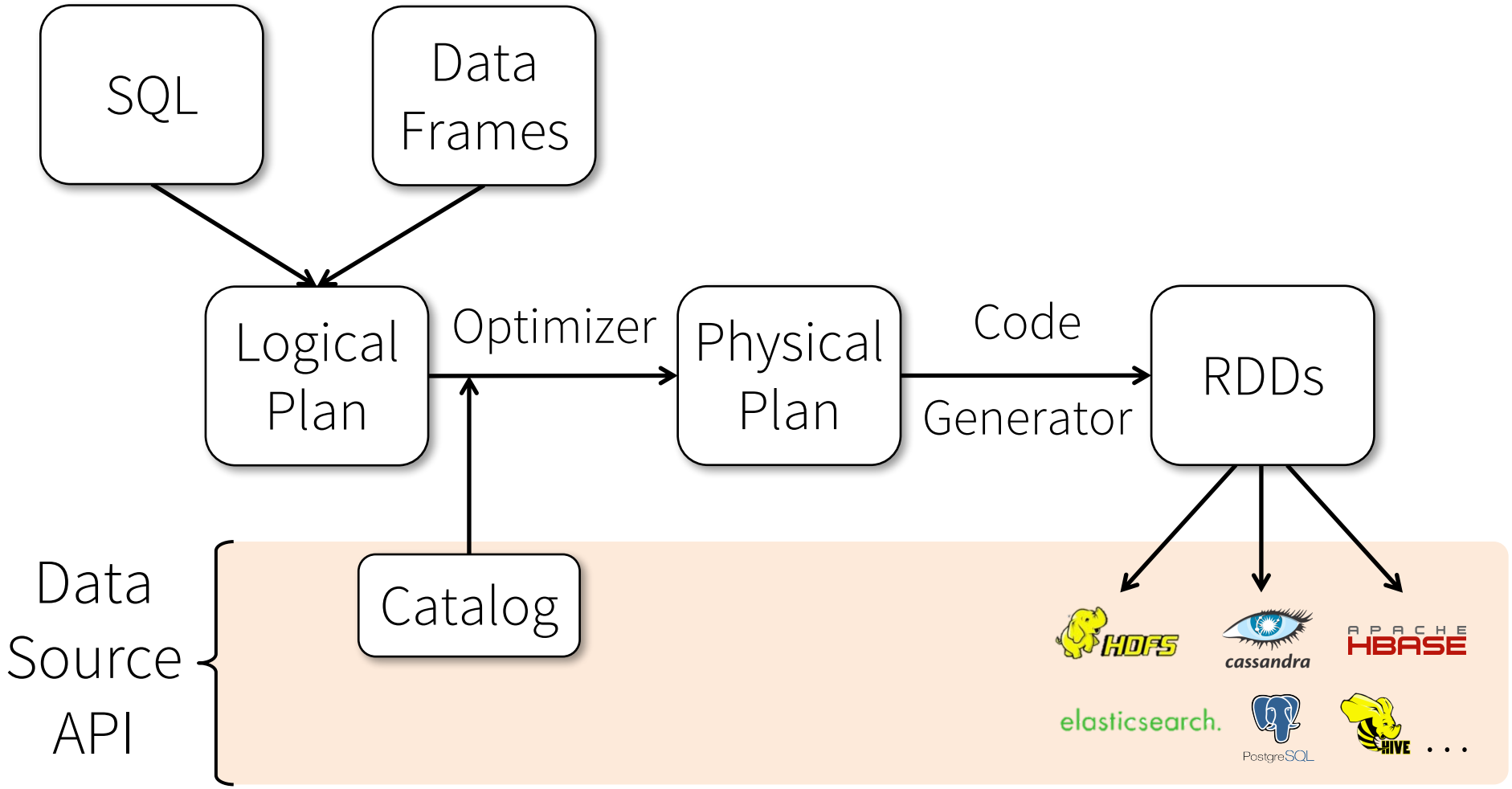
†Databricks Inc. *MIT CSAIL ‡AMPLab, UC Berkeley

ABSTRACT

Spark SQL is a new module in Apache Spark that integrates relational processing with Spark's functional programming API. Built

While the popularity of relational systems shows that users often prefer writing declarative queries, the relational approach is insufficient for many big data applications. First, users want to perform ETL to and from various data sources that might be semi- or un-

Execution Steps



DataFrame API

DataFrames hold rows with a known schema and offer relational operations on them through Spark's DataFrame relational API

```
users = spark.sql("select * from users")

ca_users = users.where(users.state == "CA")

ca_users.count()

ca_users.groupBy("name").avg("age")
```

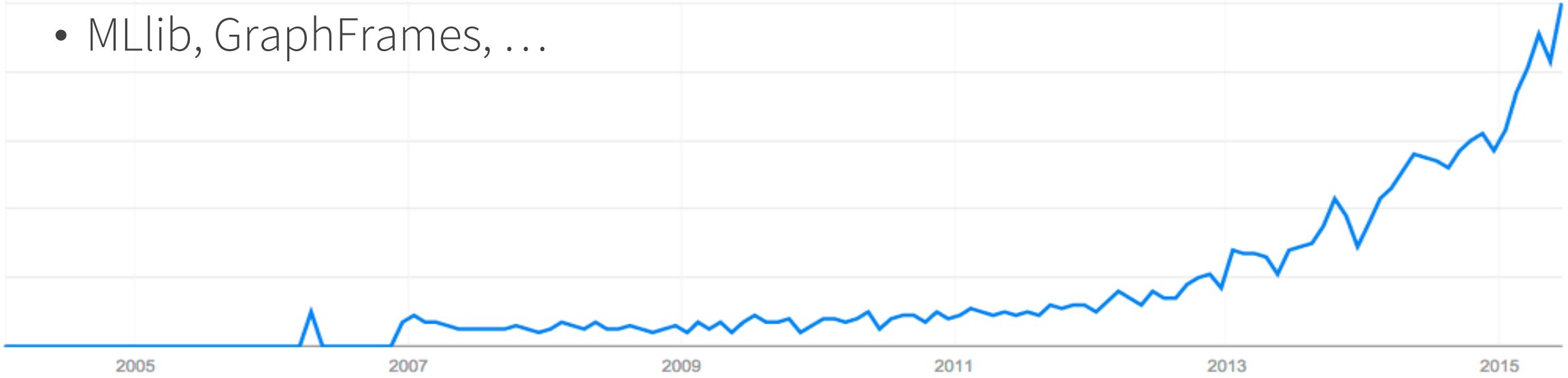
Why DataFrames?

Based on data frame concept in R and Python

- Spark is the first to make this a declarative API

Integrates with other data science libraries

- MLlib, GraphFrames, …

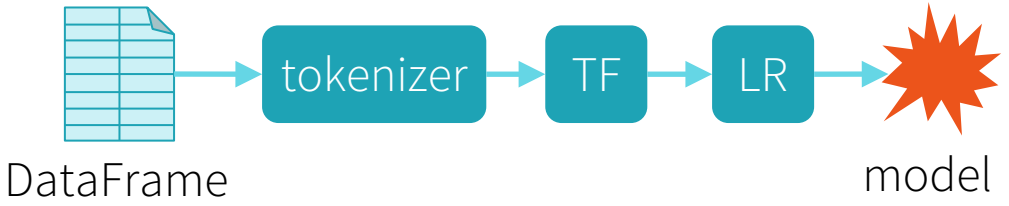


Google trends for “data frame”

Other High-Level APIs

Machine Learning Pipelines

Modular API based on scikit-learn



GraphFrames

Relational + graph operations

Structured Streaming

Declarative streaming API in Spark 2.0

Many high-level data science APIs can be declarative

DEMO

Thanks! Questions?

andy@databricks.com

Many slides re-used from:

Burak Yuvaz, Databricks. Berkeley CS186 Spring 2017
Reynold Xin, Databricks. Berkeley CS186 2016
Matei Zaharia, Databricks. Processing Big Data with Small Programs
Michael Franklin, SQL, NoSQL, NewSQL?. Berkeley CS186 2013

What's really different?

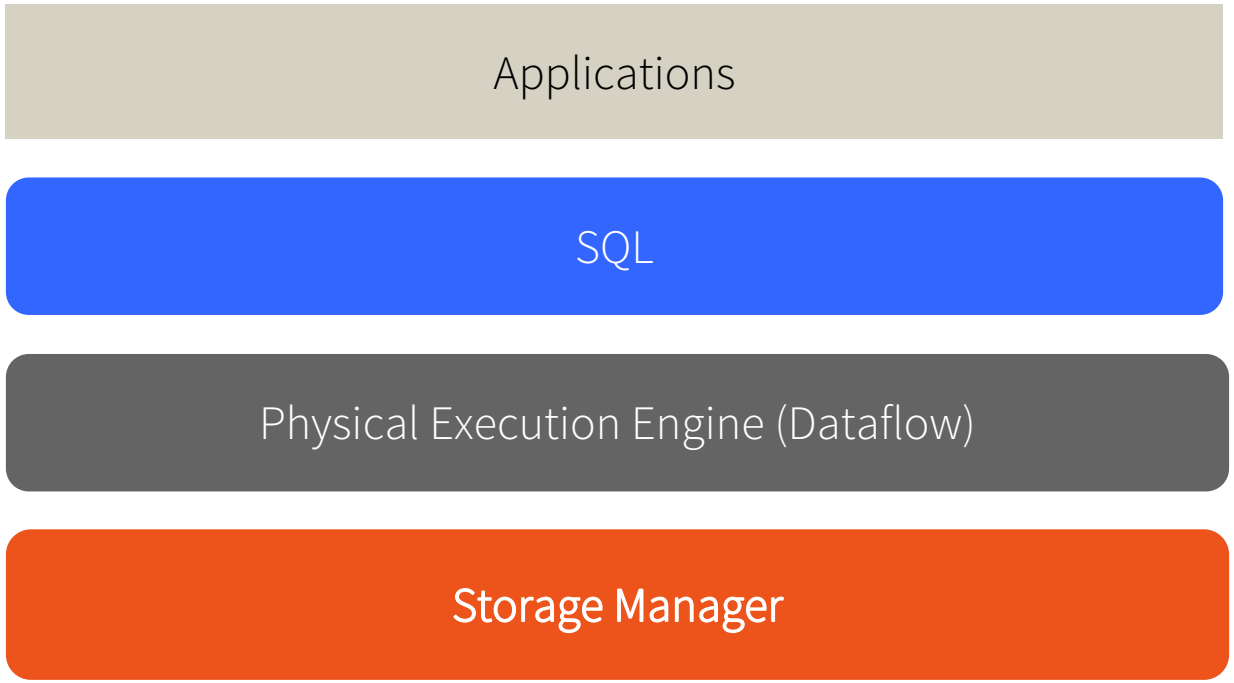
SQL on Big Data (Hadoop/Spark) vs SQL in Databases?

Two perspectives:

1. Flexibility in data and compute model

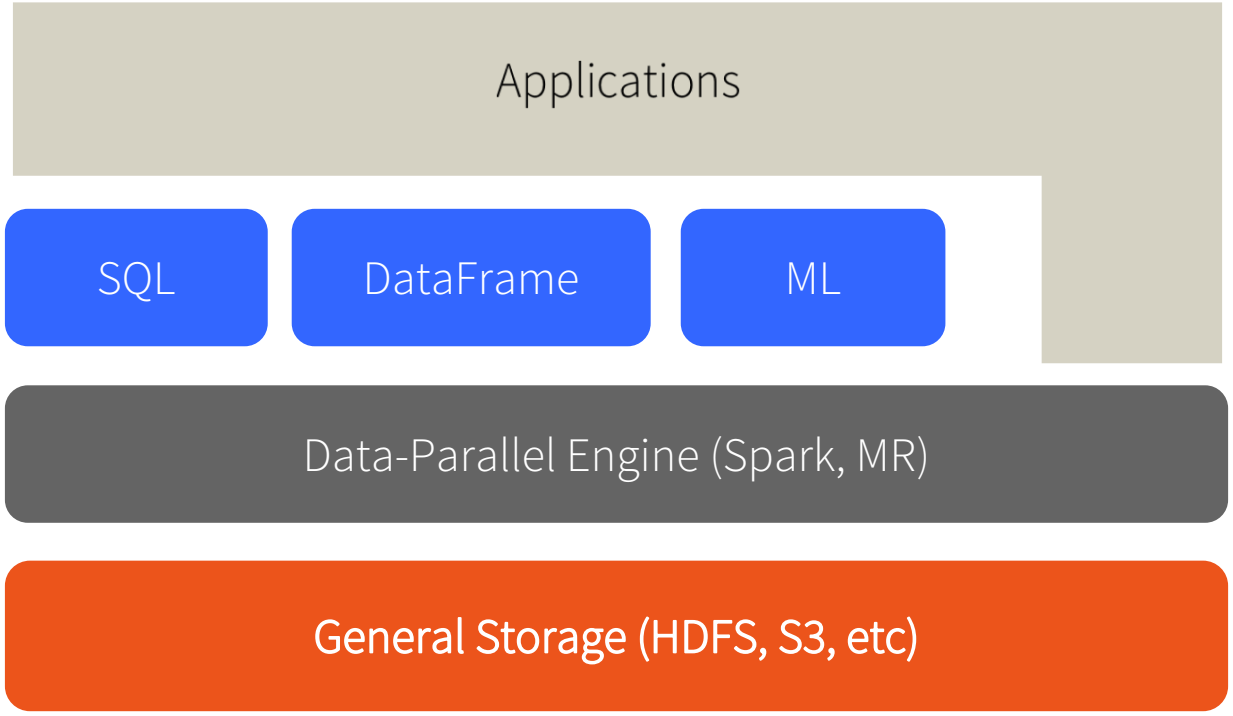
2. Fault-tolerance

Traditional Database Systems (Monolithic)



One way (SQL) in/out and data must be structured

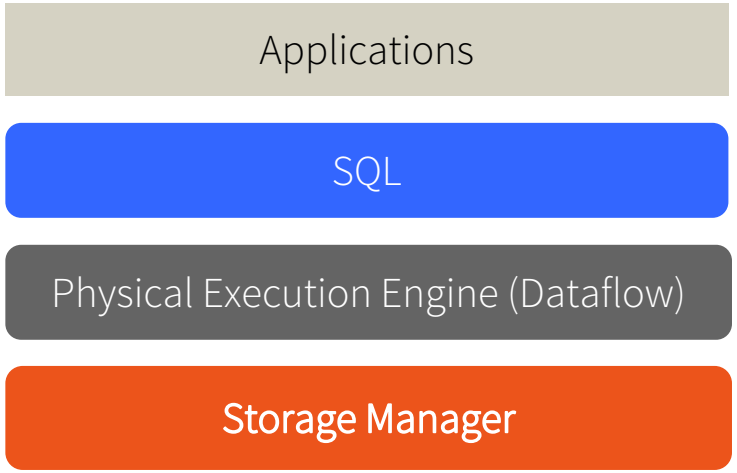
Big Data Systems (Layered)



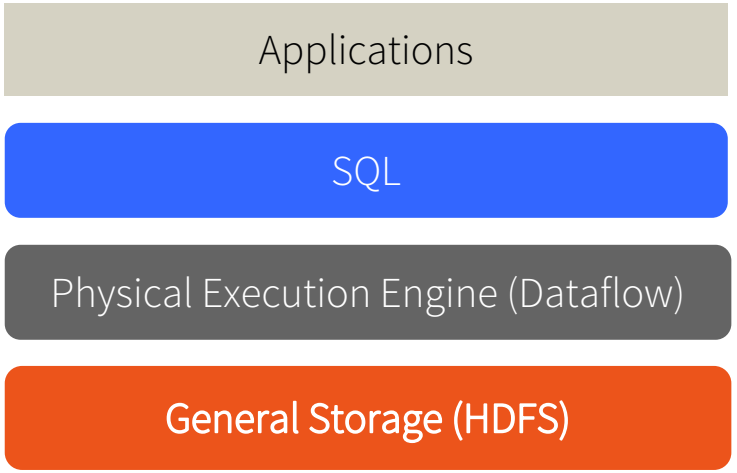
Decoupled storage, low vs high level compute
Structured, semi-structured, unstructured data
Schema on read, schema on write

Evolution of Database Systems Decouple Storage from Compute

Traditional



2014 - 2016



IBM Big Insight
Oracle
EMC Greenplum
…

support for nested data (e.g. JSON)

Perspective 2: Fault Tolerance

Database systems: coarse-grained fault tolerance

- If fault happens, fail the query (or rerun from the beginning)

MapReduce: fine-grained fault tolerance

- Rerun failed tasks, not the entire query

Google | Official Blog

Insights from Googlers into our products, technology, and the Google culture

Sorting 1PB with MapReduce

Posted: Friday, November 21, 2008

At Google we are fanatical about organizing the world's information. As a result, we spend a lot of time finding better ways to sort information using MapReduce, a key component of our software infrastructure that allows us to run multiple processes simultaneously. MapReduce is a perfect solution for many of the computations we run daily trans

We were writing it to 48,000 hard drives (we did not use the full capacity of these disks, though), and **every time we ran our sort, at least one of our disks managed to break** (this is not surprising at all given the duration of the test, the number of disks involved, and the expected lifetime of hard disks).

In ou ed
expe
spirit. You can think of it as an Olympic event for computations. By pushing the boundaries of these types of programs, we learn about the limitations of current technologies as well as the lessons useful in designing next generation computing platforms. This, in turn, should help everyone have faster access to higher-quality information.

MapReduce
Checkpointing-based Fault Tolerance

Checkpoint all intermediate output

- Replicate them to multiple nodes
- Upon failure, recover from checkpoints
- High cost of fault-tolerance (disk and network I/O)

Necessary for PBs of data on thousands of machines

What if I have 20 nodes and my query takes only 1 min?

Spark
Unified Checkpointing and Rerun

Simple idea: remember the lineage to create an RDD, and recompute from last checkpoint.

When fault happens, query still continues.

When faults are rare, no need to checkpoint, i.e. cost of fault-tolerance is low.

BD vs DB: What's Really Different?

Monolithic vs layered storage & compute

- Databases becoming more layered
- Although Big Data still far more flexible than DB

Fault-tolerance

- Databases mostly coarse-grained fault-tolerance, assuming faults are rare
- Big Data mostly fine-grained fault-tolerance, with new strategies in Spark to mitigate faults at low cost

Convergence

Databases evolving towards Big Data

- Decouple storage from compute
- Provide alternative programming models
- Semi-structured data (JSON, XML, etc)

Big Data evolving towards Databases

- Schema beyond key-value
- Separation of logical vs physical plan
- Query optimization
- More optimized storage formats

What we talked about today?

3 Vs of Big Data

GFS & MapReduce & Hadoop

Spark (RDD, DataFrame)

Convergence of Big Data and Databases

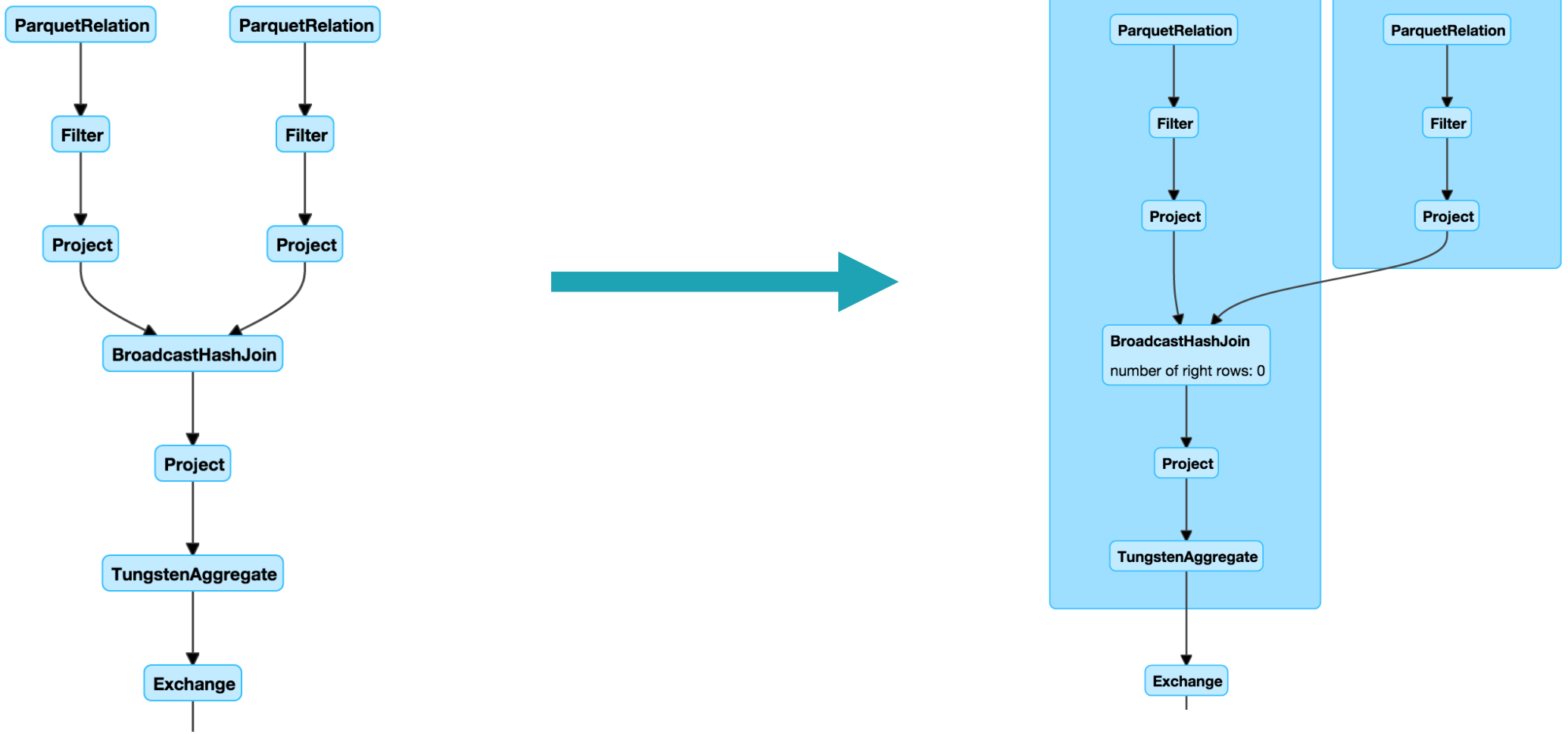
Whole-stage Codegen

Fusing operators together so the generated code looks like hand optimized code:

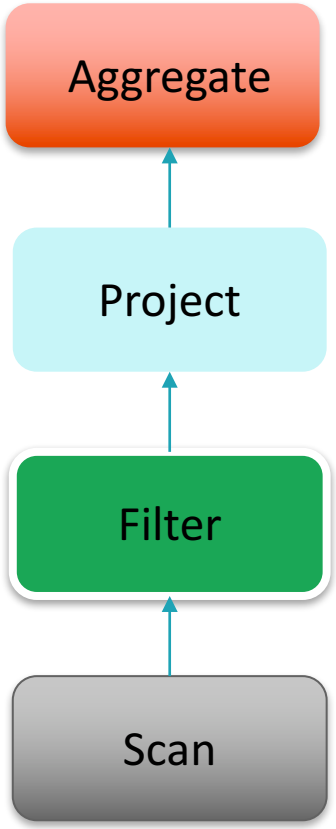
- Identify chains of operators (“stages”)
- Compile each stage into a single function
- Functionality of a general purpose execution engine; performance as if hand built system just to run your query

See paper Efficiently Compiling Efficient Query Plans for Modern Hardware, Neumann, VLDB 2011

Whole-stage Codegen: Planner



Whole-stage Codegen: Spark as a "Compiler"

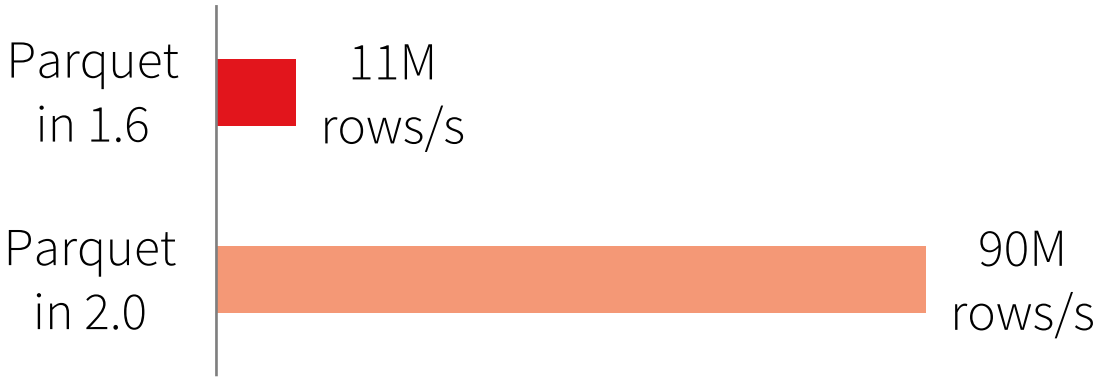


```
long count = 0;
for (ss_item_sk in store_sales) {
  if (ss_item_sk == 1000) {
    count += 1;
  }
}
```

Vectorized Decoding

Vectorized decoding

- Parquet + built-in cache
- Inspired by X100/Vectorwise



War Stories

Pipelines will fail

me	Inbox	URGENT: Internal error during audit log delivery (prod) for runId 2 on 2016-10-20 - Audit log delivery pipeline fa	10/22/16
me	Inbox	URGENT: Internal error during audit log delivery (prod) for runId 1 on 2016-10-20 - Audit log delivery pipeline fa	10/21/16
me	Inbox	URGENT: Internal error during audit log delivery (prod) for runId 3 on 2016-10-10 - Audit log delivery pipeline fa	10/12/16
me	Inbox	URGENT: Internal error during audit log delivery (prod) for runId 1 on 2016-10-11 - Audit log delivery pipeline fa	10/12/16
me	Inbox	URGENT: Internal error during audit log delivery (prod) for runId 2 on 2016-10-08 - Audit log delivery pipeline fa	10/9/16
me	Inbox	URGENT: Internal error during audit log delivery (prod) for runId 1 on 2016-10-08 - Audit log delivery pipeline fa	10/9/16
me	Inbox	URGENT: Internal error during audit log delivery (prod) for runId 3 on 2016-10-07 - Audit log delivery pipeline fa	10/8/16
me	Inbox	URGENT: Internal error during audit log delivery (prod) for runId 2 on 2016-10-06 - Audit log delivery pipeline fa	10/7/16

Troubleshooting

The audit logging pipeline fails several times a month. Usually it succeeds on the retries. Here are some of the common error cases. These error cases can be found in the digest emails sent by the pipeline:

1. `Caused by: java.io.IOException: /path/date=2016-10-25/blah.gz.parquet already exists`

   This is due to S3 eventual consistency. The retry should succeed

2. `IllegalStateException: Cannot call methods on a stopped SparkContext.`

   Something happened to the application. Probably all executors were lost and Spark killed the application. This is an open source bug and should be fixed soon. Retries with on-demand instances should succeed.

3. `Job 25 cancelled part of cancelled job group 31555453419973440370_5242158252298278457_job-84798-run-1-action-88481`

   Job seems to have timed out. May want to increase job timeout or use more instances, as load for this day may be high

4. `java.io.IOException: Failed to delete /path/date=2016-10-10/blah.gz.parquet`

   Another S3 eventual consistency problem. Retry should succeed

5. `Caused by: org.apache.spark.SparkException: Job aborted due to stage failure: Task 256 in stage 76.0 failed 4 times, most recent failure: Lost task 256.3 in stage 76.0 (TID 39492, 10.0.78.66): java.io.IOException: No space left on device`

   Caused by too much shuffle. Increasing EBS volume should help.

6. `Caused by: org.apache.spark.SparkException: Job aborted due to stage failure: Task 92 in stage 5.0 failed 4 times, most recent failure: Lost task 92.3 in stage 5.0 (TID 16616, 10.0.27.18): java.io.IOException: Stream is corrupted`

   Caused by EC2 instances in bad states. Workarounds will be merged to new version of Spark (2.1 or newer). Currently (2016/11/30) we use a customized spark image in prod job which seems fixed the issue.

Bad data problems

One day one table had 8 columns

The next day it had 32 columns

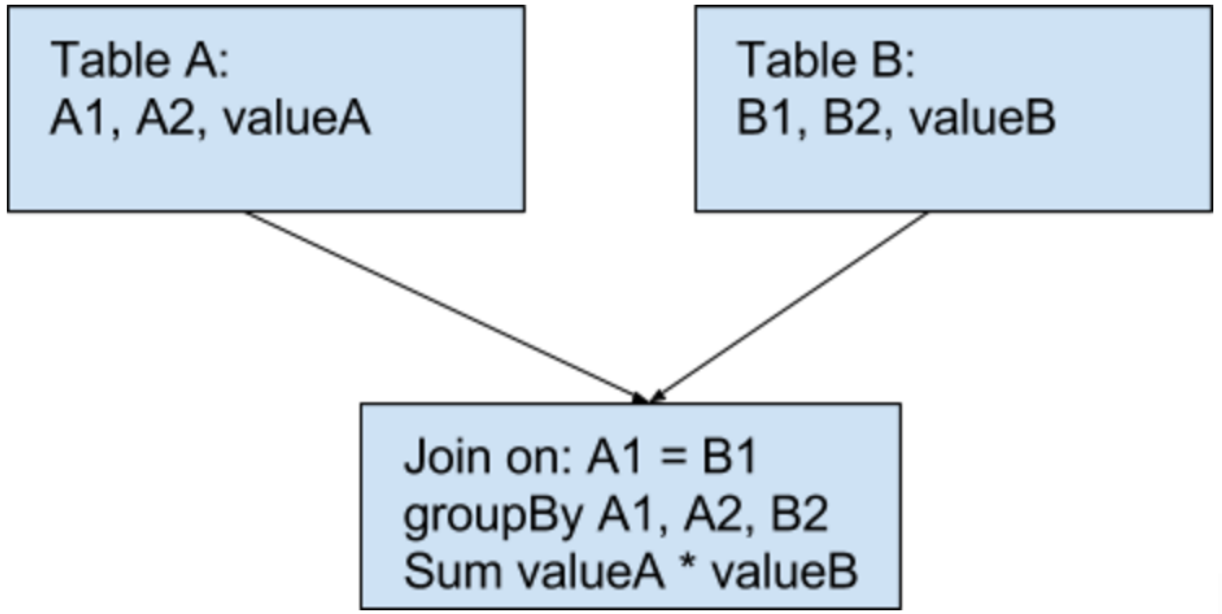
What happened?

Customer Stories

Attempting to join ~400M rows x ~2B rows

Keep hitting out of memory issues

After join get 4 Trillion rows



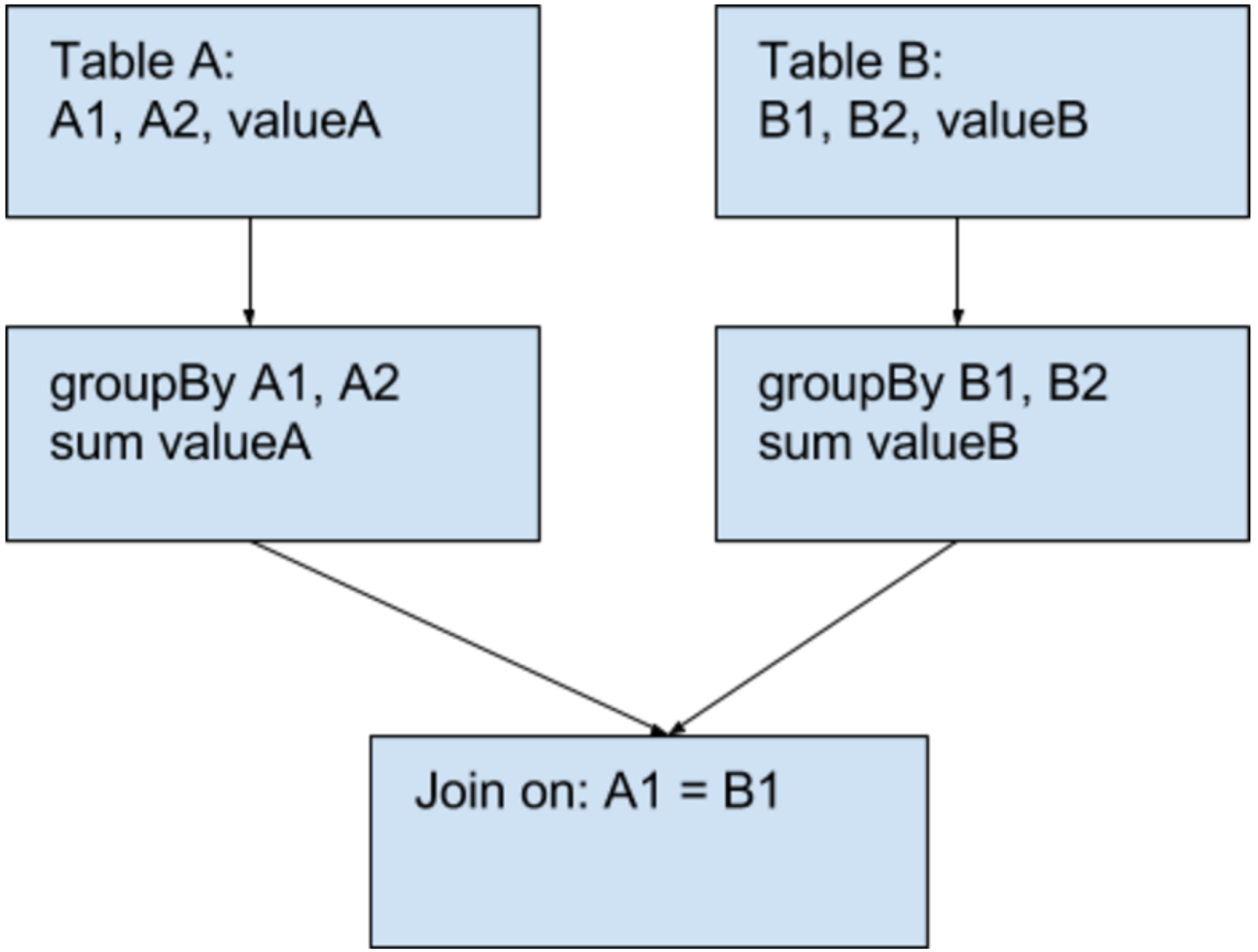


Table A:
A1, A2, valueA
Table B:
B1, B2, valueB
groupBy A1, A2
sum valueA
groupBy B1, B2
sum valueB
Join on: A1 = B1

Customer Stories

Customer job spuriously fails with `ExecutorLostFailure`

```
16/09/29 17:36:44 WARN AkkaRpcEndpointRef: Error sending message [message =
Heartbeat(18,[Lscala.Tuple2;@79d26749,BlockManagerId(18, 10.49.188.112, 37861))] in 1 attempts
org.apache.spark.rpc.RpcTimeoutException: Futures timed out after [120 seconds]. This timeout is controlled by
spark.rpc.askTimeout at
org.apache.spark.rpc.RpcTimeout.org$apache$spark$rpc$RpcTimeout$$createRpcTimeoutException(RpcEnv.scala:214)
```

They are running an iterative ML algorithm. The futures timed out errors all seem to be happening around the "collect" phase

They have 50 workers -> 400 tasks. Each task is trying to send about 25 MB to the driver. Total bytes received by the driver will be around 25 * 400 -> 10 GB.

The driver starts GC'ing after several iterations (as it received 10 GB on the previous iterations). GC + Network saturation prevent the Driver and Master from acknowledging the Executor heartbeats.

Executors get killed because the master didn't receive any heartbeats (even though the executors tried for 2 minutes)

Job fails because they use rdd.localCheckpoint

groupByKey + collect causing large network traffic to Driver

Replace with treeAggregate

Lessons Learned

Unit testing

#1 thing you will do when working in industry

Helps set contracts

Makes sure other people don’t break those contracts

Good to learn frameworks like jUnit, scalatest, python unittest, mocha in JS

Solve problems at the source

Requires clearly defining the problem & understanding the root cause

Sometimes not easy, may be time consuming

Doing it right has better rate of return

Aggregate classes of problems and try building holistic solutions

- Customers will want faster horses. Build them a car instead
- Try solving bigger picture problem rather than manifestations of problem
- Otherwise end up with fragmented solutions that work somewhere but not everywhere

Leverage what's out there

Someone probably has faced an issue you're facing

Someone probably has already solved the issue you're facing

Use that someone's work

Don't reinvent the wheel

Scope work as much as you can ahead of time

Figure out requirements

Provide the simplest possible solution that meets these requirements

KISS (Keep it simple, stupid)

Think about evolvability over time

Extra Spark Slides

Why is it so hard?

Data is a mess

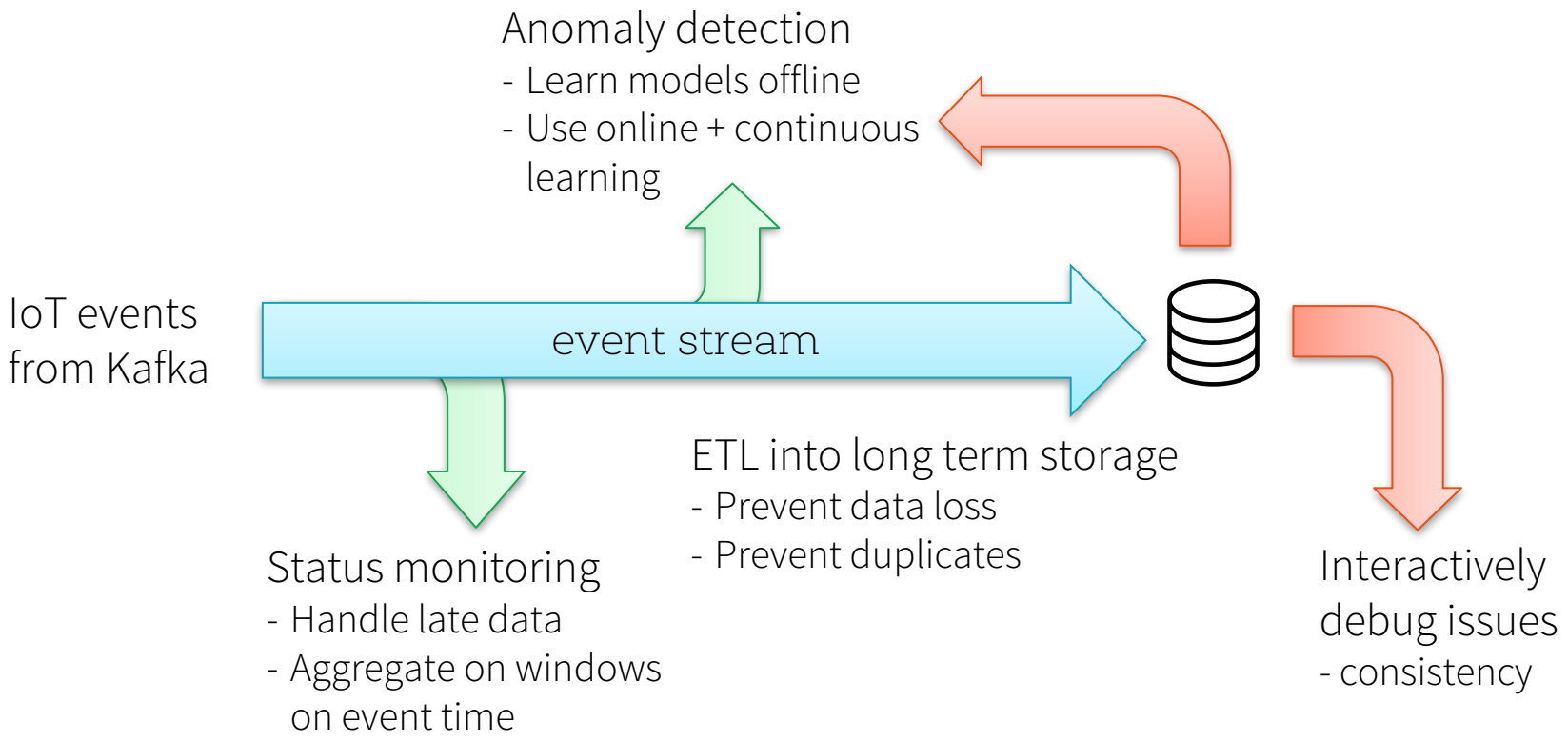
- Siloed across many sources
- Saved in different formats
- Always has “bad” values
- Evolves over time

Scalability is an issue

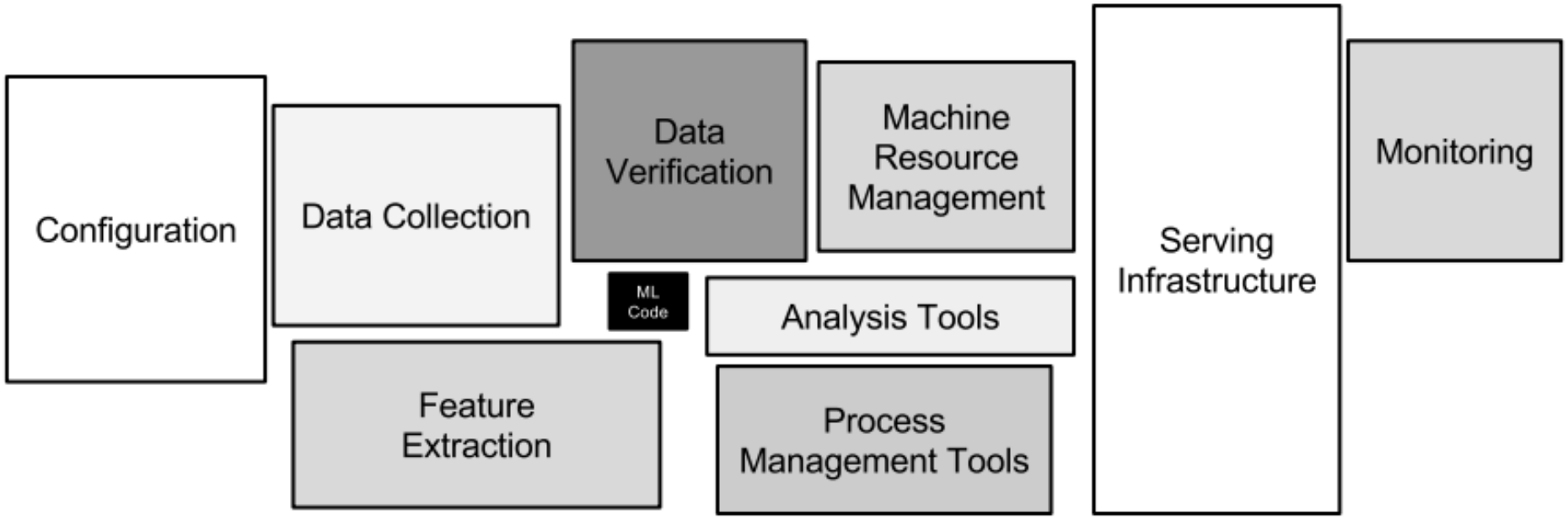
- Require fault tolerance
- Data skew is a pain

Why is it so hard?

Pipelines are growing complex



The hard part of ML



https://papers.nips.cc/paper/5656-hidden-technical-debt-in-machine-learning-systems.pdf

Apache® Spark™ to the rescue

Unified engine for large-scale data processing

- Batch / Offline
- Streaming / Online

Fast

Easy to use

- Great APIs
- Available in Python, Scala, Java, R, SQL

Huge Ecosystem

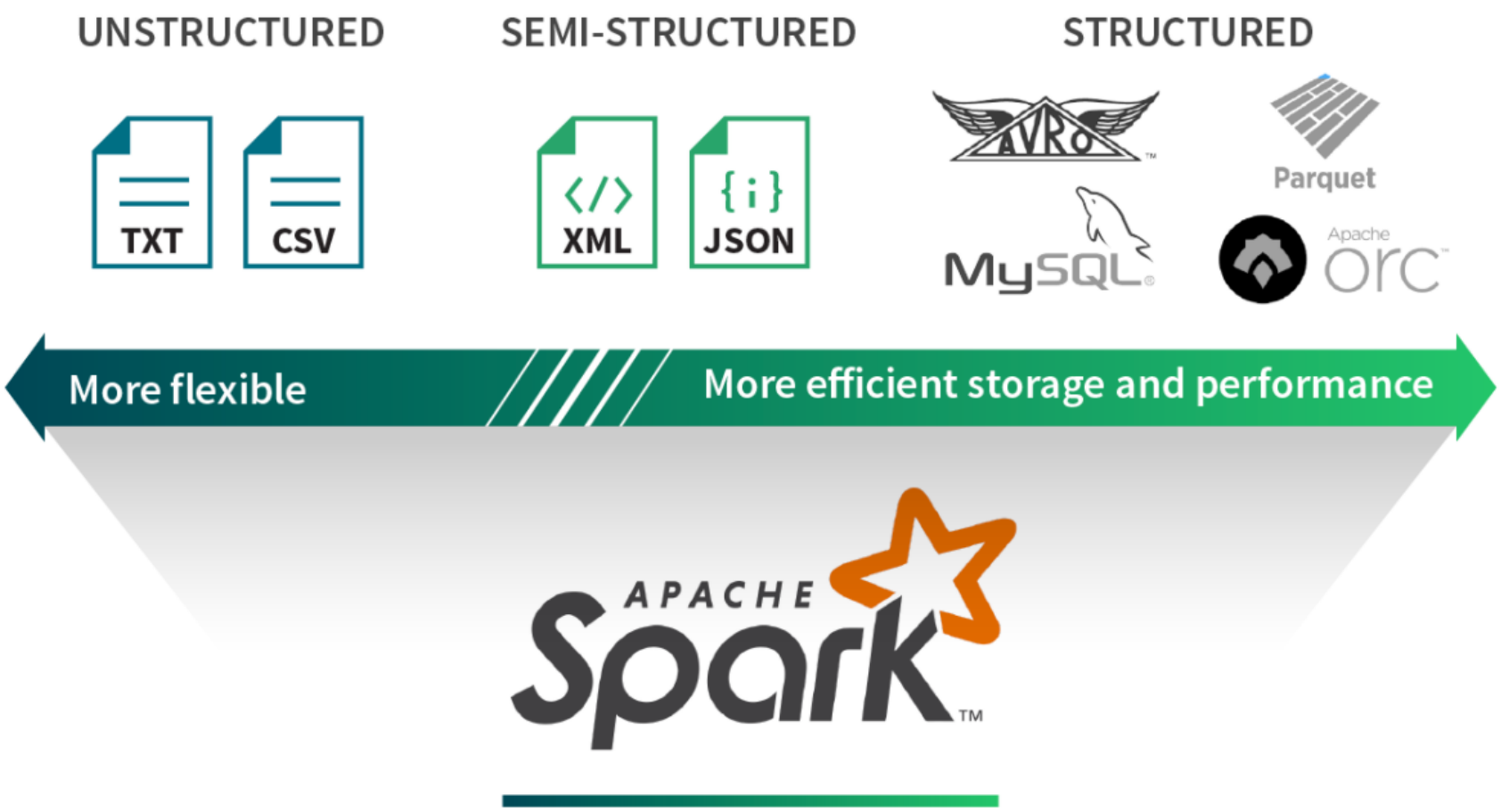
- 1,000+ contributors on GitHub
- Can run Standalone, on Yarn, on Mesos, on the Cloud, on premise
- Can connect to many data sources and consume many data formats
- Spark Packages

Spark has 1000s of users

Cloud	On-Prem
• Elastic • Low cost of ownership • Pay-as-you-go • No up-front infrastructure burden • No infra-maintenance burden	• Rigid • You're in control • Can customize according to niche requirements

Data Sources

Data Formats



Google File System

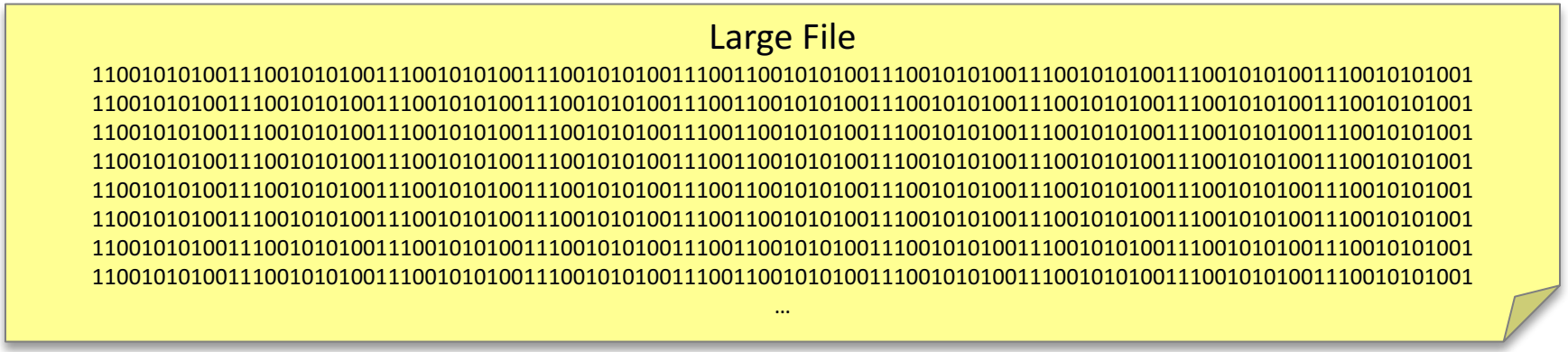
GFS Assumptions

- “Files are huge by traditional standards”
- “Component failures are the norm rather than the exception”
- Files are append-only
  - “Most files are mutated by appending new data rather than overwriting existing data”
  - Why is this ok given our workload types?
  - What are the advantages of this?
  - Alternative techniques & types of storage systems for when updates required

(Quotes are from GFS paper)

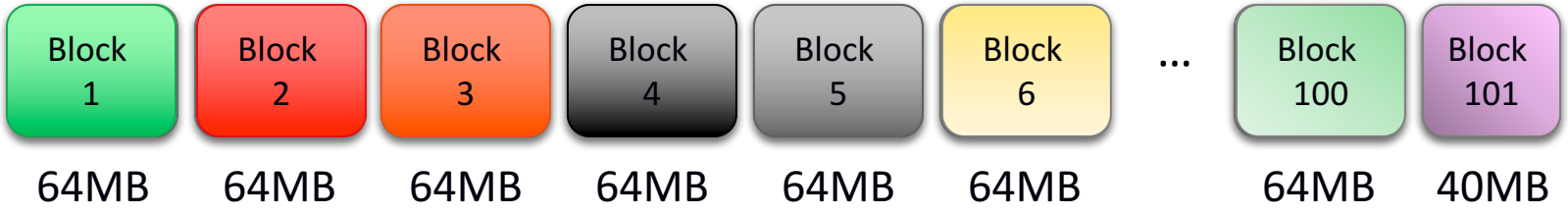
File Splits

Example:



6440MB

Let's color-code them



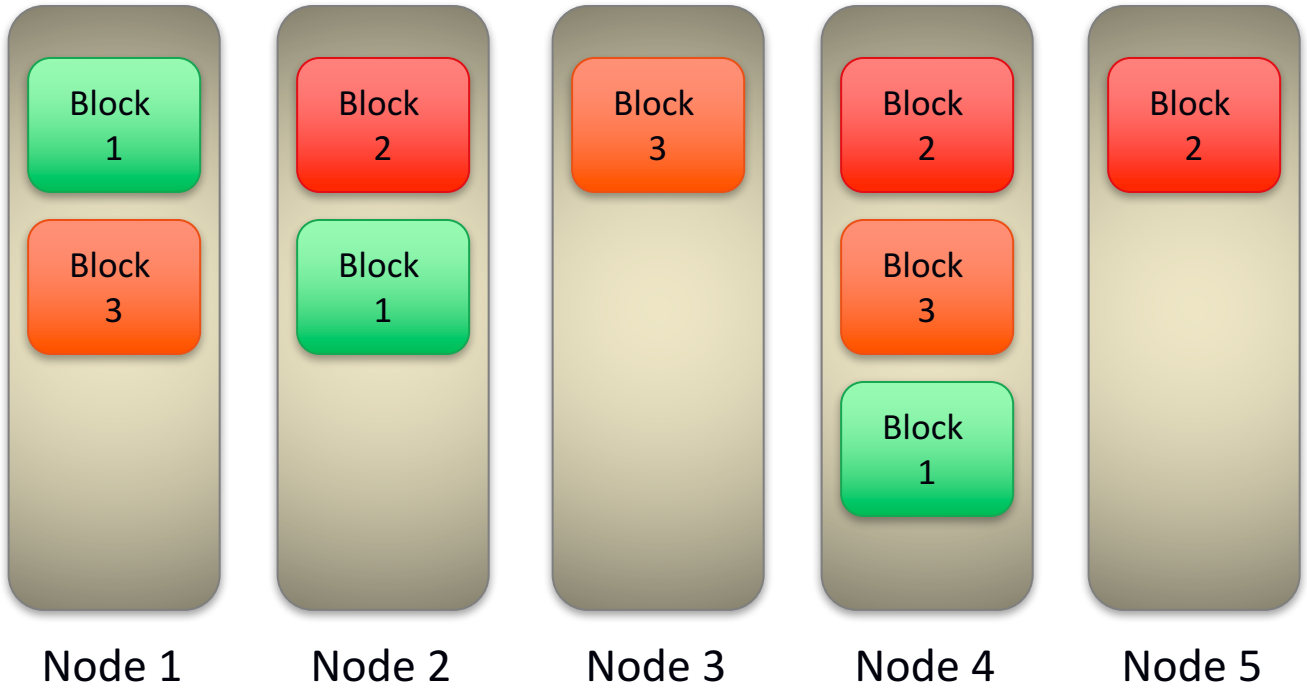
e.g., Block Size = 64MB

Files are composed of set of blocks

- Typically 64MB in size
- Each block is stored as a separate file in the local file system (e.g. NTFS)

Block Placement

Example:



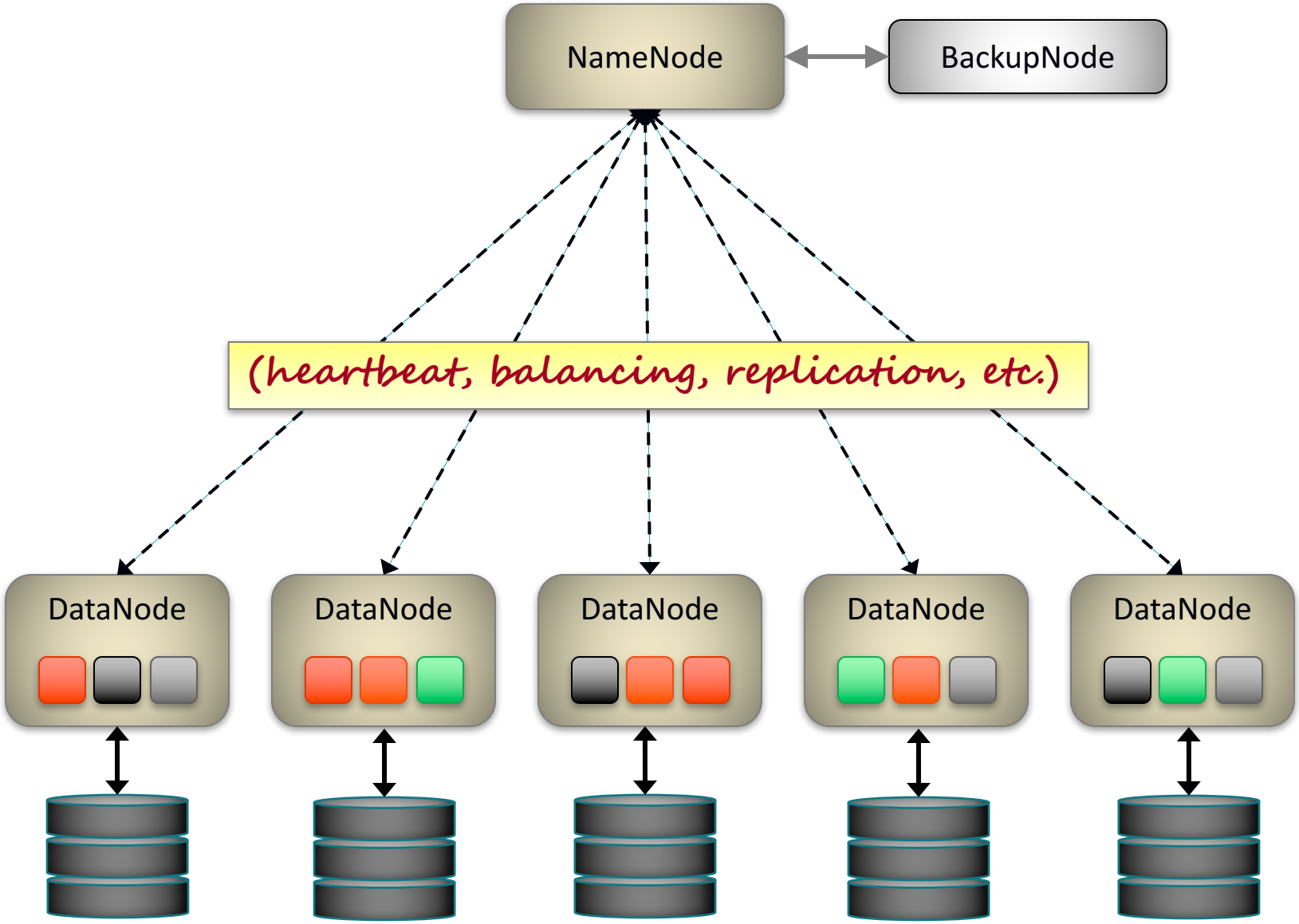
e.g., Replication factor = 3

Default placement policy:

- First copy is written to the node creati… …ently)
- Second copy is writt… …the same rack
  …cross-rack network traffic)
- …y is written to a data node in a different rack
  (*to tolerate switch failures*)

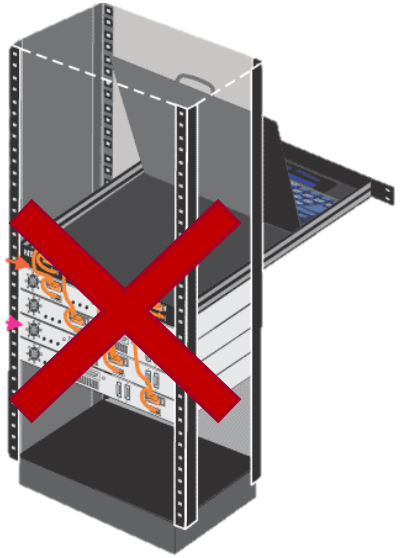
Objectives: load balancing, fast access, fault tolerance

GFS Architecture



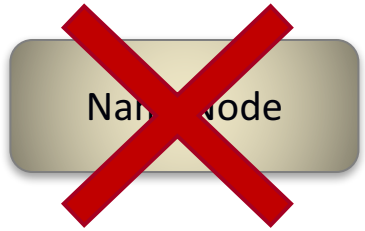
Failures, Failures, Failures

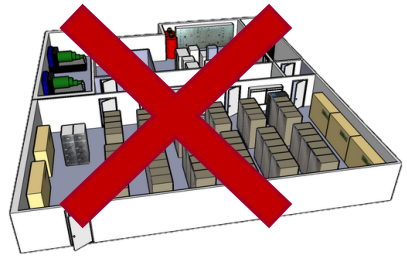
GFS paper: “Component failures are the norm rather than the exception.”



Failure types:

- Disk errors and failures
- DataNode failures
- Switch/Rack failures
- NameNode failures
- Datacenter failures





Open Source version of GFS

There is an open source version called Hadoop Distributed File System (HDFS). More on Hadoop later.

What about indexes?

GFS does not support indexes out of the box!

When is it OK not to have them as an option?

How does it affect workloads?

What about workloads that need them?

When is it OK not

How does it affec

What about work

Motivation

- HBase as a database depends on FileSystem for many things
- HBase has to work over HDFS, linux & windows
- HBase is the most advanced user of HDFS
- For tuning for IO performance, you have to understand how HBase does IO

MapReduce	HBase
Large files	Large files
Few random seek	A lot of random seek
Batch oriented	Latency sensitive
High throughput	
Failure handling at task level	Durability guarantees with sync
Computation moves to data	Computation generates local data
	Large number of open files

https://www.slideshare.net/enissoz/hbase-and-hdfs-understanding-filesystem-usage slide 33

databricks

GFS Summary

- Store large, immutable (append-only) files
- Scalability
- Reliability
- Availability
- Append-only, no indices

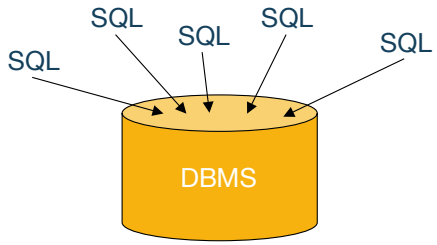
Note on another buzzword:

Cloudera & other industry vendors popularized the buzzword “Data Lake” – basically just another name for HDFS.

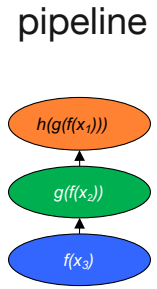
MapReduce

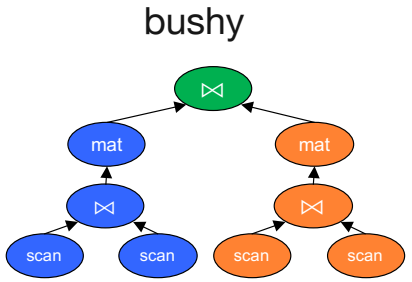
[REVIEW] Summary: Kinds of Parallelism

- Inter-Query



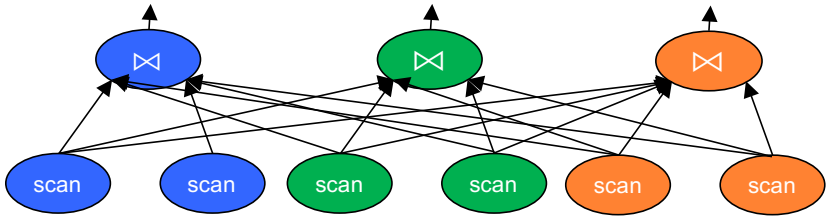
- Intra-Query
  1. Inter-Operator





Most of Big Data (Hadoop, Spark, etc.)

2. Intra-Operator (partitioned)



MapReduce Programming Model

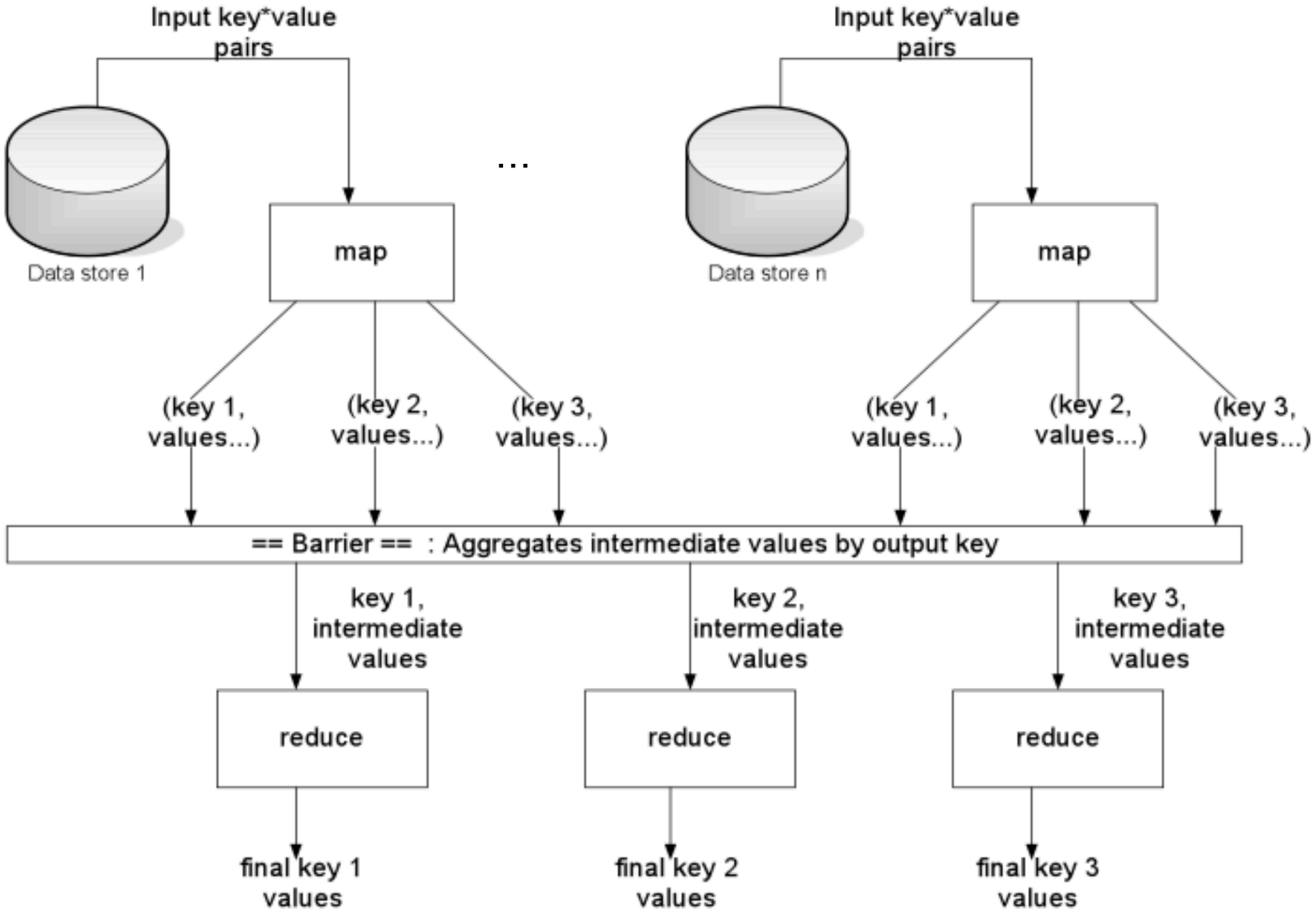
Data type: key-value *records*

Map function:

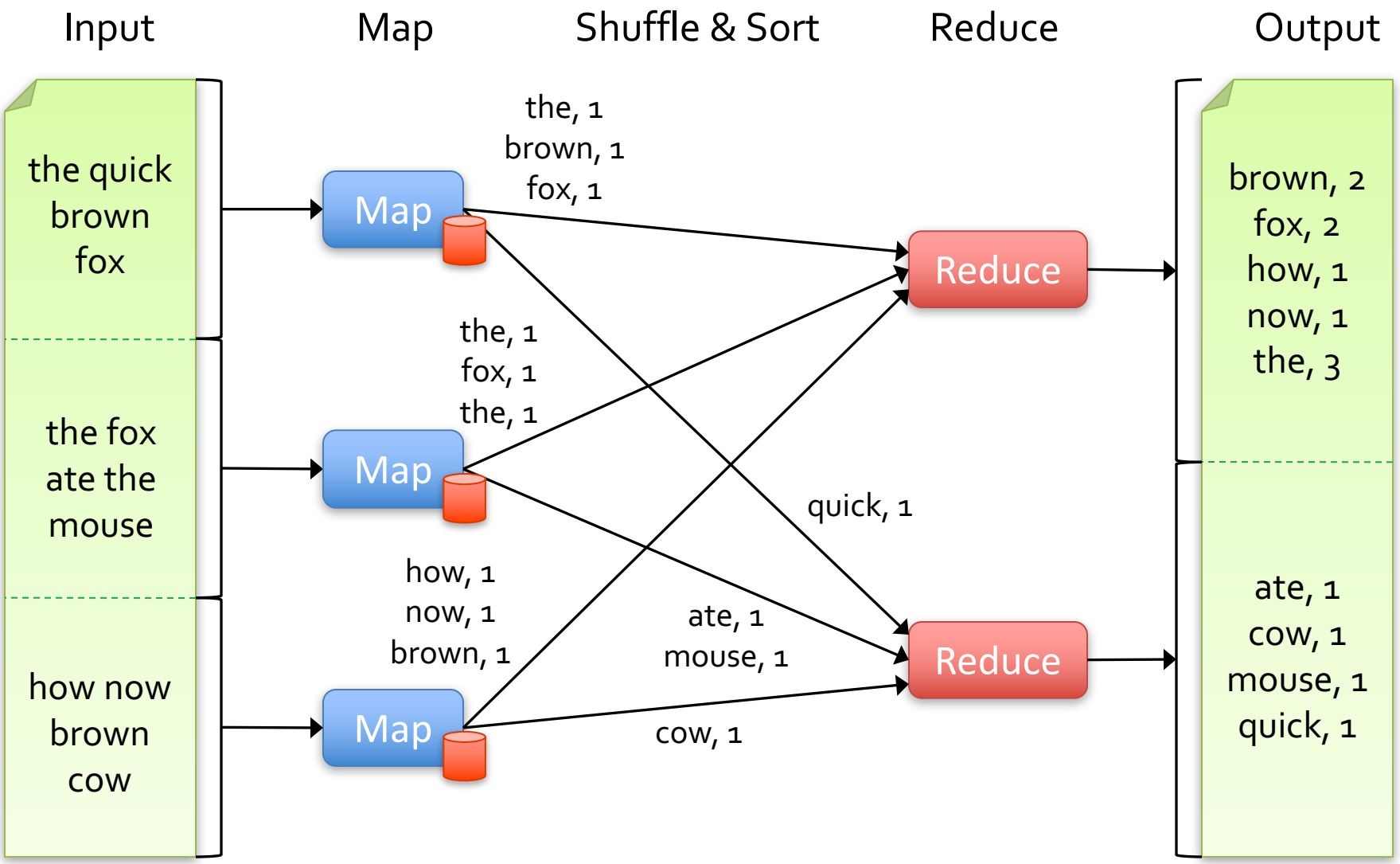
$$(K_{in}, V_{in}) \rightarrow list(K_{inter}, V_{inter})$$

Reduce function:

$$(K_{inter}, list(V_{inter})) \rightarrow list(K_{out}, V_{out})$$


Input key*value pairs
Input key*value pairs
...
Data store 1
Data store n
map
map
(key 1, values...)
(key 2, values...)
(key 3, values...)
(key 1, values...)
(key 2, values...)
(key 3, values...)
== Barrier == : Aggregates intermediate values by output key
key 1, intermediate values
key 2, intermediate values
key 3, intermediate values
reduce
reduce
reduce
final key 1 values
final key 2 values
final key 3 values

Hello World of Big Data: Word Count



MapReduce Execution

Automatically split work into many small tasks

Send map tasks to nodes based on data locality

Load-balance dynamically as tasks finish

MapReduce Fault Recovery

If a task fails, re-run it and re-fetch its input

- Requirement: input is immutable

If a node fails, re-run its map tasks on others

- Requirement: task result is deterministic & side effect is idempotent

Dealing with Stragglers

Misconfigured/broken nodes = slow tasks

What's the fix?

"Backup Tasks" = launch 2nd copy of slowest tasks on another node

44% speedups in MR paper

What are requirements for this to work?

How do you define slow?

Cost? Backup tasks are not free, why?

3.6 Backup Tasks

One of the common causes that lengthens the total time taken for a MapReduce operation is a "straggler": a machine that takes an unusually long time to complete one of the last few map or reduce tasks in the computation. Stragglers can arise for a whole host of reasons. For example, a machine with a bad disk may experience frequent correctable errors that slow its read performance from 30 MB/s to 1 MB/s. The cluster scheduling system may have scheduled other tasks on the machine, causing it to execute the MapReduce code more slowly due to competition for CPU, memory, local disk, or network bandwidth. A recent problem we experienced was a bug in machine initialization code that caused processor caches to be disabled: computations on affected machines slowed down by over a factor of one hundred.

We have a general mechanism to alleviate the problem of stragglers. When a MapReduce operation is close to completion, the master schedules backup executions of the remaining *in-progress* tasks. The task is marked as completed whenever either the primary or the backup execution completes. We have tuned this mechanism so that it typically increases the computational resources used by the operation by no more than a few percent. We have found that this significantly reduces the time to complete large MapReduce operations. As an example, the sort program described in Section 5.3 takes 44% longer to complete when the backup task mechanism is disabled.

Improving MapReduce Performance in Heterogeneous Environments

Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, Ion Stoica
University of California, Berkeley
{matei,andyk,adj,randy,stoica}@cs.berkeley.edu

Abstract

MapReduce is emerging as an important programming model for large-scale data-parallel applications such as web indexing, data mining, and scientific simulation. Hadoop is an open-source implementation of MapReduce enjoying wide adoption and is often used for short jobs where low response time is critical. Hadoop's performance is closely tied to its task scheduler, which implicitly assumes that cluster nodes are homogeneous and tasks make progress linearly, and uses these assumptions to decide when to speculatively re-execute tasks that appear to be stragglers. In practice, the homogeneity assumptions do not always hold. An especially compelling setting where this occurs is a virtualized data center, such as Amazon's Elastic Compute Cloud (EC2). We show that Hadoop's scheduler can cause severe performance degradation in heterogeneous environments. We design a new scheduling algorithm, Longest Approximate Time to End (LATE), that is highly robust to heterogeneity. LATE can improve Hadoop response times by a factor of 2 in clusters of 200 virtual machines on EC2.

1 Introduction

Today's most popular computer applications are Internet services with millions of users. The sheer volume of data

The MapReduce model popularized by Google is very attractive for ad-hoc parallel processing of arbitrary data. MapReduce breaks a computation into small tasks that run in parallel on multiple machines, and scales easily to very large clusters of inexpensive commodity computers. Its popular open-source implementation, Hadoop [2], was developed primarily by Yahoo, where it runs jobs that produce hundreds of terabytes of data on at least 10,000 cores [4]. Hadoop is also used at Facebook, Amazon, and Last.fm [5]. In addition, researchers at Cornell, Carnegie Mellon, University of Maryland and PARC are starting to use Hadoop for seismic simulation, natural language processing, and mining web data [5, 6].

A key benefit of MapReduce is that it automatically handles failures, hiding the complexity of fault-tolerance from the programmer. If a node crashes, MapReduce reruns its tasks on a different machine. Equally importantly, if a node is available but is performing poorly, a condition that we call a *straggler*, MapReduce runs a *speculative copy* of its task (also called a "backup task") on another machine to finish the computation faster. Without this mechanism of *speculative execution*[1], a job would be as slow as the misbehaving task. Stragglers can arise for many reasons, including faulty hardware and misconfiguration. Google has noted that speculative execution can improve job response times by 44% [1].

MapReduce Summary

By providing a data-parallel model, MapReduce greatly simplified cluster computing:

- Automatic division of job into tasks
- Locality-aware scheduling
- Load balancing
- Recovery from failures & stragglers w/ backup tasks

Also flexible enough to model a lot of workloads…

MapReduce Summary (continued)

Focused on intra-operator parallelism

Allows for simple mid-query fault tolerance

Allows for Straggler handling

Hadoop

Open-sourced by Yahoo!

- modeled after the two Google papers

Two components:

- Storage: Hadoop Distributed File System (HDFS)
- Compute: Hadoop MapReduce

Huge investment by VCs: three large, well-funded companies & ~6 other distributions

- Cloudera: $1B in funding, IPO this year
- Hortonworks: $248M in funding, IPO last year
- MapR: $280M in funding
- Distributions (pre-consolidation): Amazon, Microsoft, Intel, Teradata, ~~IBM~~, ~~EMC/Pivotal~~

MapReduce: A major step backwards

By David DeWitt on January 17, 2008 4:20 PM | Permalink | Comments (44) | TrackBacks (1)

[Note: Although the system attributes this post to a single author, it was written by David J. DeWitt and Michael Stonebraker]

On January 8, a Database Column reader asked for our views on new distributed database research efforts, and we'll begin here v
to discuss it, since the recent trade press has been filled with news of the revolution of so-called "cloud computing." This paradigı
processors working in parallel to solve a computing problem. In effect, this suggests constructing a data center by lining up a largı
much smaller number of high-end servers.

For example, IBM and Google have announced plans to make a 1,000 processor cluster available to a few select universities to te
software tool called MapReduce [1]. Berkeley has gone so far as to plan on teaching their freshman how to program using the Ma

As both educators and researchers, we are amazed at the hype that the MapReduce proponents have spread about how it represen
data-intensive applications. MapReduce may be a good idea for writing certain types of general-purpose computations, but to the

1. A giant step backward in the programming paradigm for large-scale data intensive applications
2. A sub-optimal implementation, in that it uses brute force instead of indexing
3. Not novel at all -- it represents a specific implementation of well known techniques developed nearly 25 years ago
4. Missing most of the features that are routinely included in current DBMS