# Big Data Frameworks

Michele Beretta

michele.beretta@unibg.it

## Tutoring Schedule

We have 18 hours of tutoring

- 18/11 08:30–10:30
- 25/11 08:30–10:30
- 26/11 16:00–19:00
- 28/11 10:30–13:30
- 02/12 08:30–10:30
- 03/12 16:00–19:00
- 05/12 10:30–13:30

## Outline

- Big Data
- Hadoop Distributed File System
- MapReduce
- Pig & Hive
- Apache Spark
- TBC

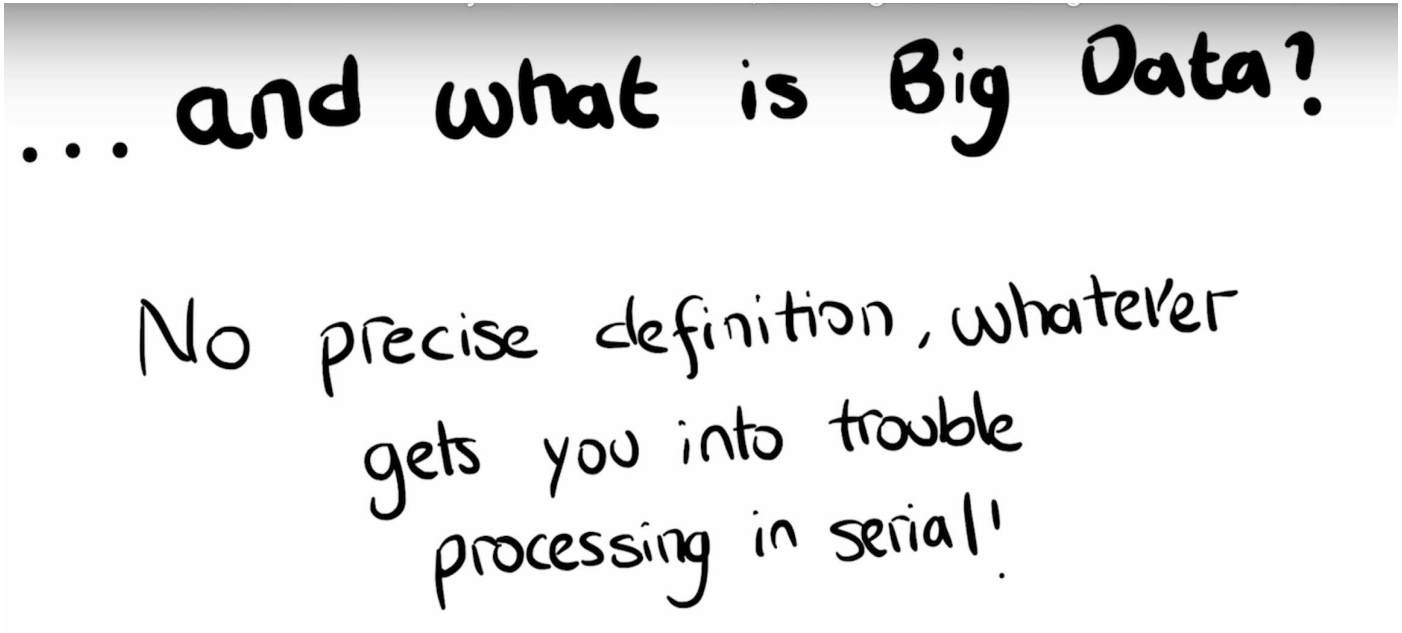
# What is Big Data?

# What is Big Data?


"**Big Data**" is the catch-all term for massive amounts of data as well as for frameworks and R&D initiatives aimed at working with them efficiently.

Image source: erpinnews.com

# A short definition of Big Data




A nice definition from this year's PRACE Summer of HPC presentation "Convergence of HPC and Big Data".

# The three V's of Big Data

It is customary to define Big Data in terms of three V's:

- Volume (the sheer volume of data)
- Velocity (rate of flow of the data and processing speed needs)
- Variety (different sources and formats)

# The three V's of Big Data

Data arise from disparate sources and come in many sizes and formats. Velocity refers to the speed of data generation as well as to processing speed requirements.

| **Volume** | **Velocity** | **Variety** |
|---|---|---|
| MB | batch | table |
| GB | periodic | database |
| TB | near-real time | multimedia |
| PB | real time | unstructured |
| ... | ... | ... |

# Reference: metric prefixes

| | | | | |
|---|---|---|---|---|
| 1000000000000000000000000 | $10^{24}$ | yotta | Y | septillion |
| 1000000000000000000000 | $10^{21}$ | zetta | Z | sextillion |
| 1000000000000000000 | $10^{18}$ | exa | E | quintillion |
| 1000000000000000 | $10^{15}$ | peta | P | quadrillion |
| 1000000000000 | $10^{12}$ | tera | T | trillion |
| 1000000000 | $10^{9}$ | giga | G | billion |
| 1000000 | $10^{6}$ | mega | M | million |
| 1000 | $10^{3}$ | kilo | k | thousand |

**Note:** 1 Gigabyte (GB) is $10^9$ bytes. Sometimes GB is also used to denote $1024^3$ or $2^30$ bytes, which is actually one *gibibyte* (GiB).

# Structured vs. unstructured data

By *structured* data one refers to highly organized data that are usually stored in relational databases or data warehouses. Structured data are easy to search but unflexible in terms of the three "V"s.

*Unstructured* data come in mixed formats, usually require pre-processing, and are difficult to search. Structured data are usually stored in noSQL databases or in *data lakes* (these are scalable storage spaces for raw data of mixed formats).

# Examples of structured/unstructured data

| Industry | Structured data | Unstructured data |
|---|---|---|
| e-commerce | • products & prices<br>• customer data<br>• transactions | • reviews<br>• phone transcripts<br>• social media mentions |
| banking | • financial transactions<br>• customer data | • customer communication<br>• regulations & compliance<br>• financial news |
| healthcare | • patient data<br>• medical billing data | • clinical reports<br>• radiology imagery |

# Big Data in 2025

| Data Phase | Astronomy | Twitter | YouTube | Genomics |
|---|---|---|---|---|
| **Acquisition** | 25 zetta-bytes/year | 0.5–15 billion tweets/year | 500–900 million hours/year | 1 zetta-bases/year |
| **Storage** | 1 EB/year | 1–17 PB/year | 1–2 EB/year | 2–40 EB/year |
| **Analysis** | In situ data reduction | Topic and sentiment mining | Limited requirements | Heterogeneous data and analysis |
| | Real-time processing | Metadata analysis | | Variant calling, ~2 trillion central processing unit (CPU) hours |
| | Massive volumes | | | All-pairs genome alignments, ~10,000 trillion CPU hours |
| **Distribution** | Dedicated lines from antennae to server (600 TB/s) | Small units of distribution | Major component of modern user's bandwidth (10 MB/s) | Many small (10 MB/s) and fewer massive (10 TB/s) data movement |

doi:10.1371/journal.pbio.1002195.t001

This table[1] shows the projected annual storage and computing needs in four domains (astronomy, social media, genomics)

[1] Stephens ZD et al. "Big Data: Astronomical or Genomical?" In: *PLoS Biol* (2015).

# The three V's of Big Data: additional dimensions

Three more "V"s to be pondered:

- Veracity (quality or trustworthiness of data)
- Value (economic value of the data)
- Variability (general variability in any of the aforementioned characteristics)

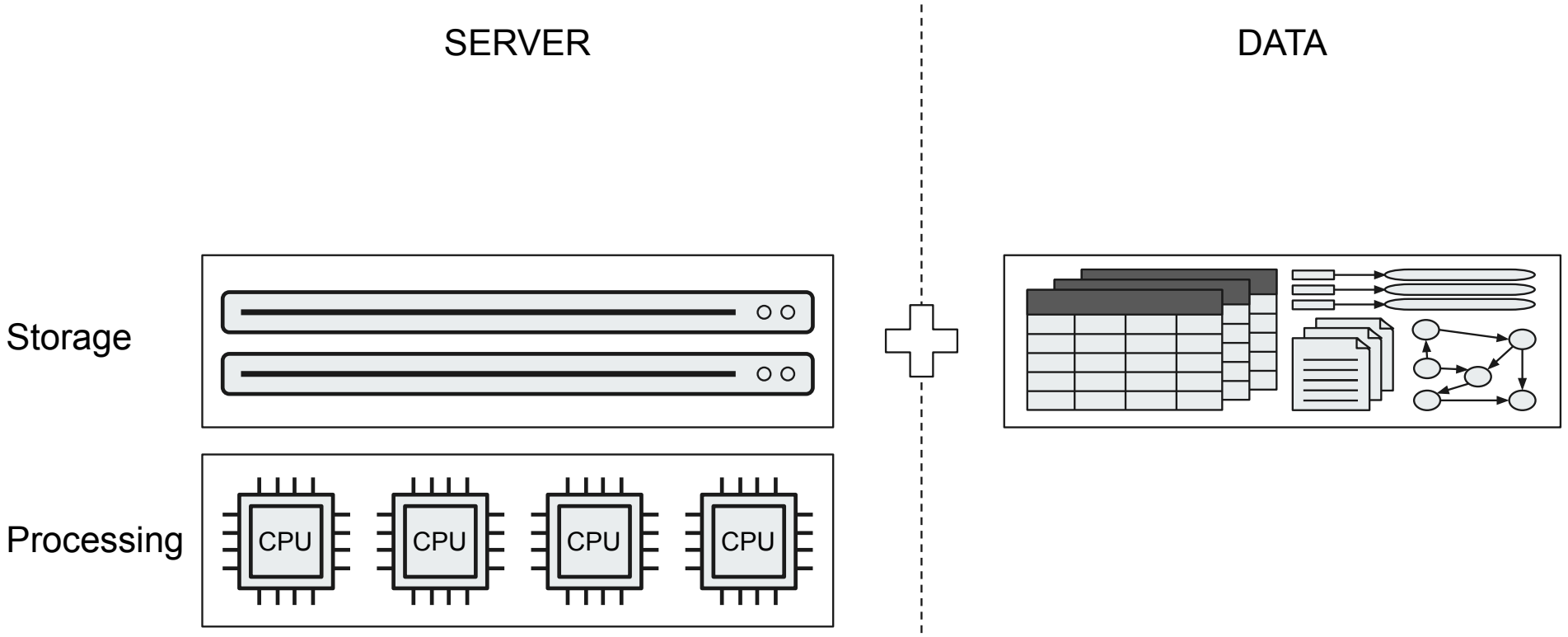
# The challenges of Big Data

Anyone working with large amounts of data will sooner or later be confronted with one or more of these challenges:

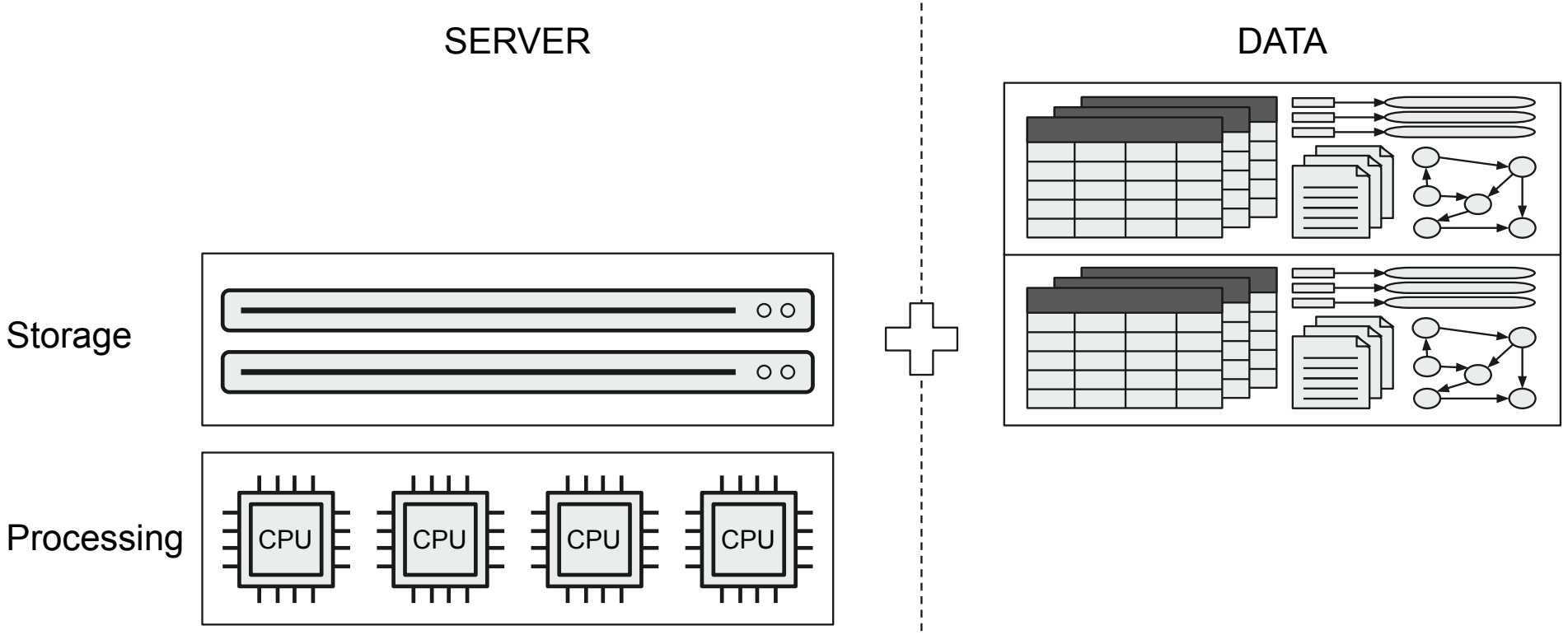
- disk and memory space
- processing speed
- hardware faults
- network capacity and speed
- need to optimize resources use

# But how should we scale our systems?

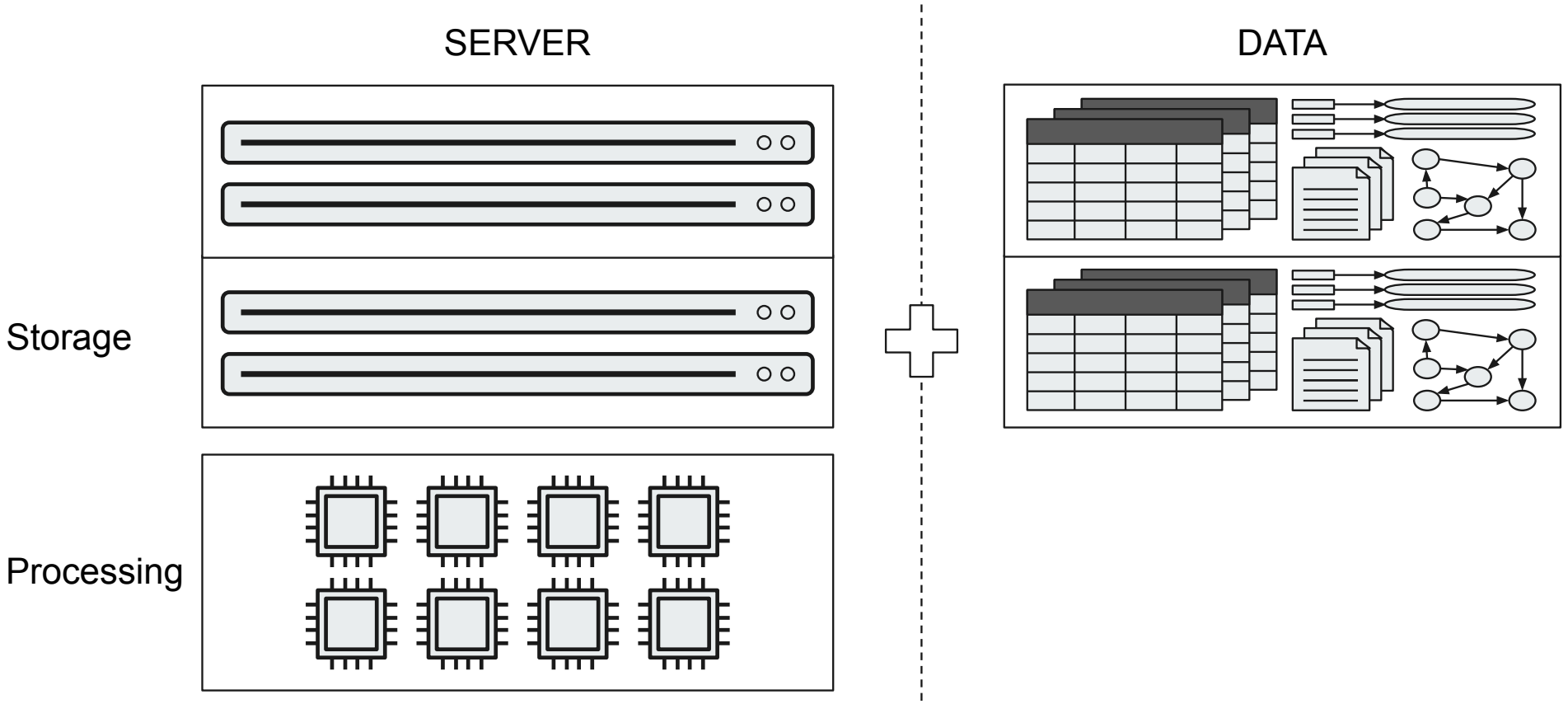
# Initial setup



# What if the data volume doubles?



# Vertical scaling



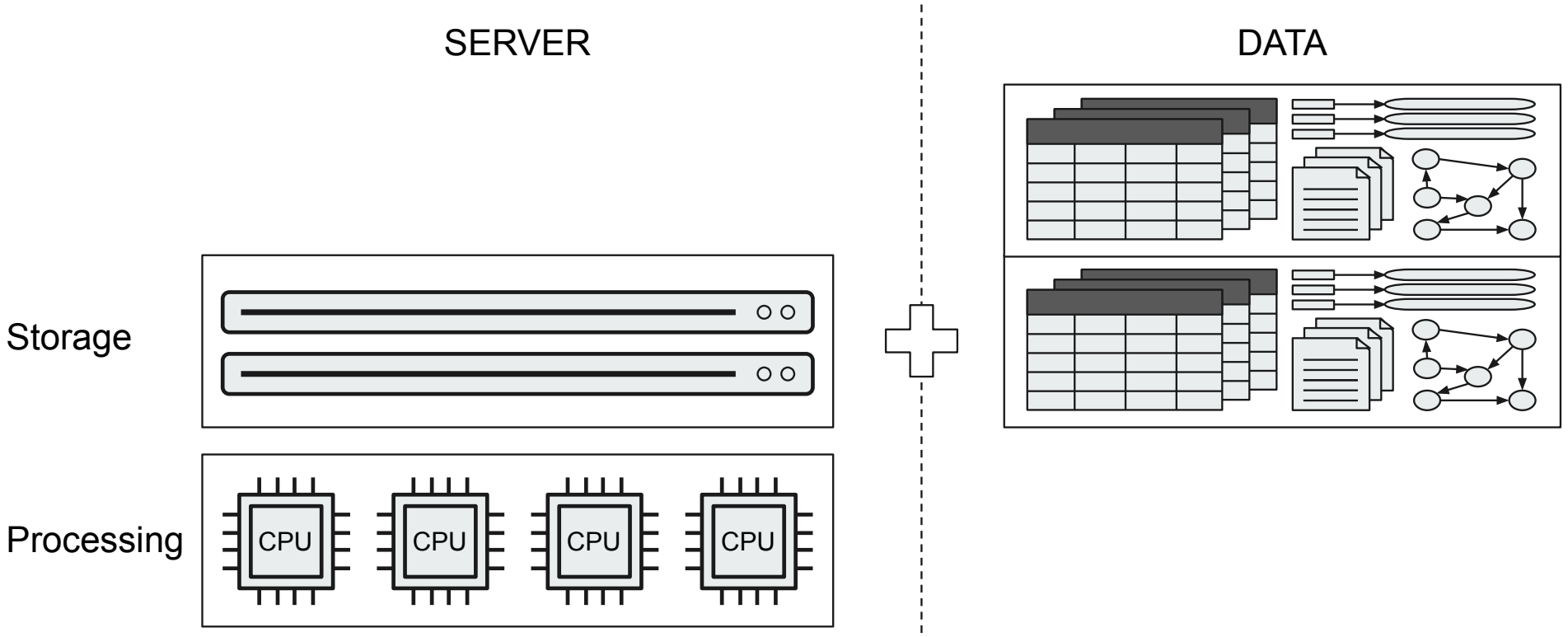
# Limitations

- **Maintenance window with downtime**
  Unless you have a backup server that can handle operations and requests, you will need some considerable downtime to upgrade your machine

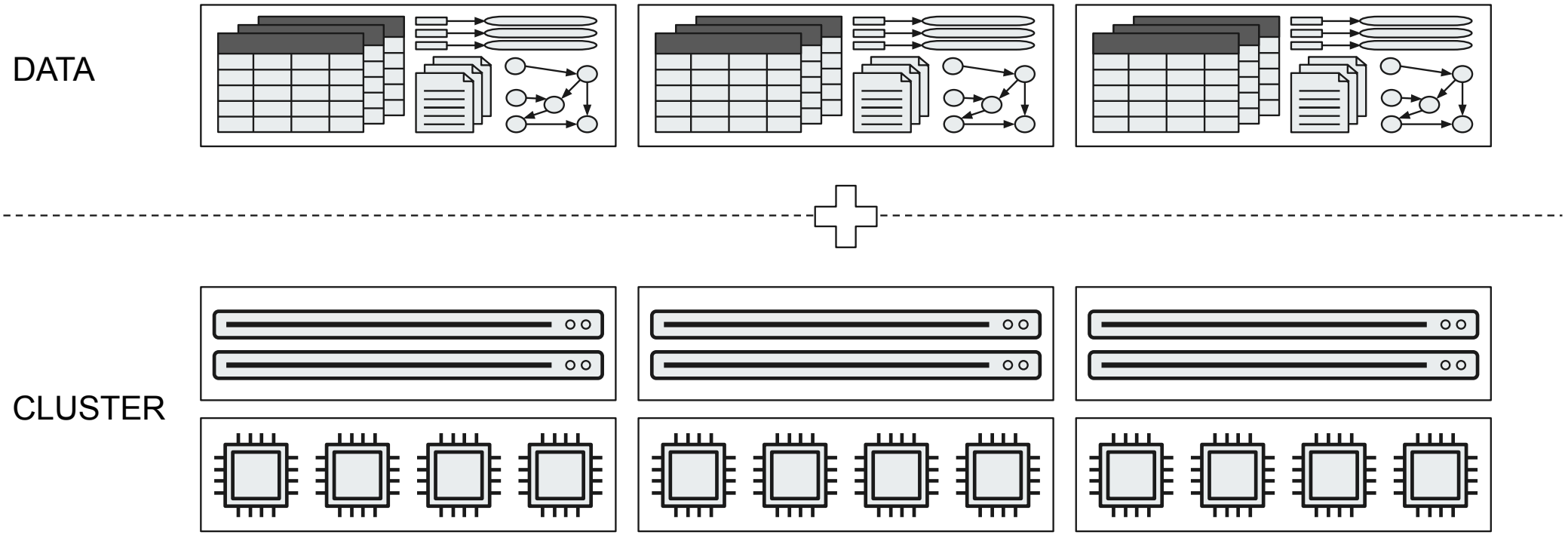
- **Single point of failure**
  Having all your operations on a single server increases the risk of losing all your data if a hardware or software failure were to occur

- **Upgrade limitations**
  There is a limitation to how much you can upgrade a machine

# What is the alternative?



# Horizontal scaling



# ~~Limitations~~ → Advantages

- ~~**Maintenance window with downtime**~~ → **Upgradability and flexibility**
  If you run your application layer on separate machines (horizontally scaled), they are easier to decouple and upgrade without downtime

- ~~**Single point of failure**~~ → **Reliability and availability**
  Horizontal scaling can provide you with a more reliable system. It increases redundancy and ensures that you are not dependent on one machine

- ~~**Upgrade limitations**~~ → **The only limit is $$$**
  There is no practical limit to how many machine you may put together, the only limit is the financial ability to buy more servers

# Limitations

- **Complexity of maintaining and operating the architecture**
  Distributed systems are complex to design, program and debug. You have to take into account the network limits (latency and bandwidth) and make the system reliable to partial failures

- **Limited software support**
  This problem is less and less existing as most of the development these days is oriented towards the development of microservices and cloud native solutions

# The Hadoop distributed computing architecture

# Hadoop for distributed data processing

Hadoop is a framework for running jobs on clusters of computers that provides a good abstraction of the underlying hardware and software.

"*Stripped to its core, the tools that Hadoop provides for building distributed systems—for data storage, data analysis, and coordination—are simple. If there's a common theme, it is about raising the level of abstraction—to create building blocks for programmers who just happen to have lots of data to store, or lots of data to analyze, or lots of machines to coordinate, and who don't have the time, the skill, or the inclination to become distributed systems experts to build the infrastructure to handle it.* [2]"

[2] White T. *Hadoop: The Definitive Guide*. O'Reilly, 1988.

# Hadoop: some facts

Hadoop[3] is an open-source project of the Apache Software Foundation. The project was created to facilitate computations involving massive amounts of data.

- its core components are implemented in Java
- initially released in 2006. Last stable version is 3.3.1 from June 2021
- originally inspired by Google's MapReduce[4] and the proprietary GFS (Google File System)

[3] Apache Software Foundation. *Hadoop*. url: https://hadoop.apache.org.

[4] J. Dean and S. Ghemawat. "MapReduce: Simplified data processing on large clusters." In: *Proceedings of Operating Systems Design and Implementation (OSDI)*. 2004. url: https://www.usenix.org/legacy/publications/library/proceedings/osdi04/tech/full_papers/dean/dean.pdf .

# Hadoop's features

Hadoop's features addressing the challenges of Big Data:

- scalability
- fault tolerance
- high availability
- distributed cache/data locality
- cost-effectiveness as it does not need high-end hardware
- provides a good abstraction of the underlying hardware
- easy to learn
- data can be queried trough SQL-like endpoints (Hive, Cassandra)

# Mini-glossary of Hadoop's distinguishing features

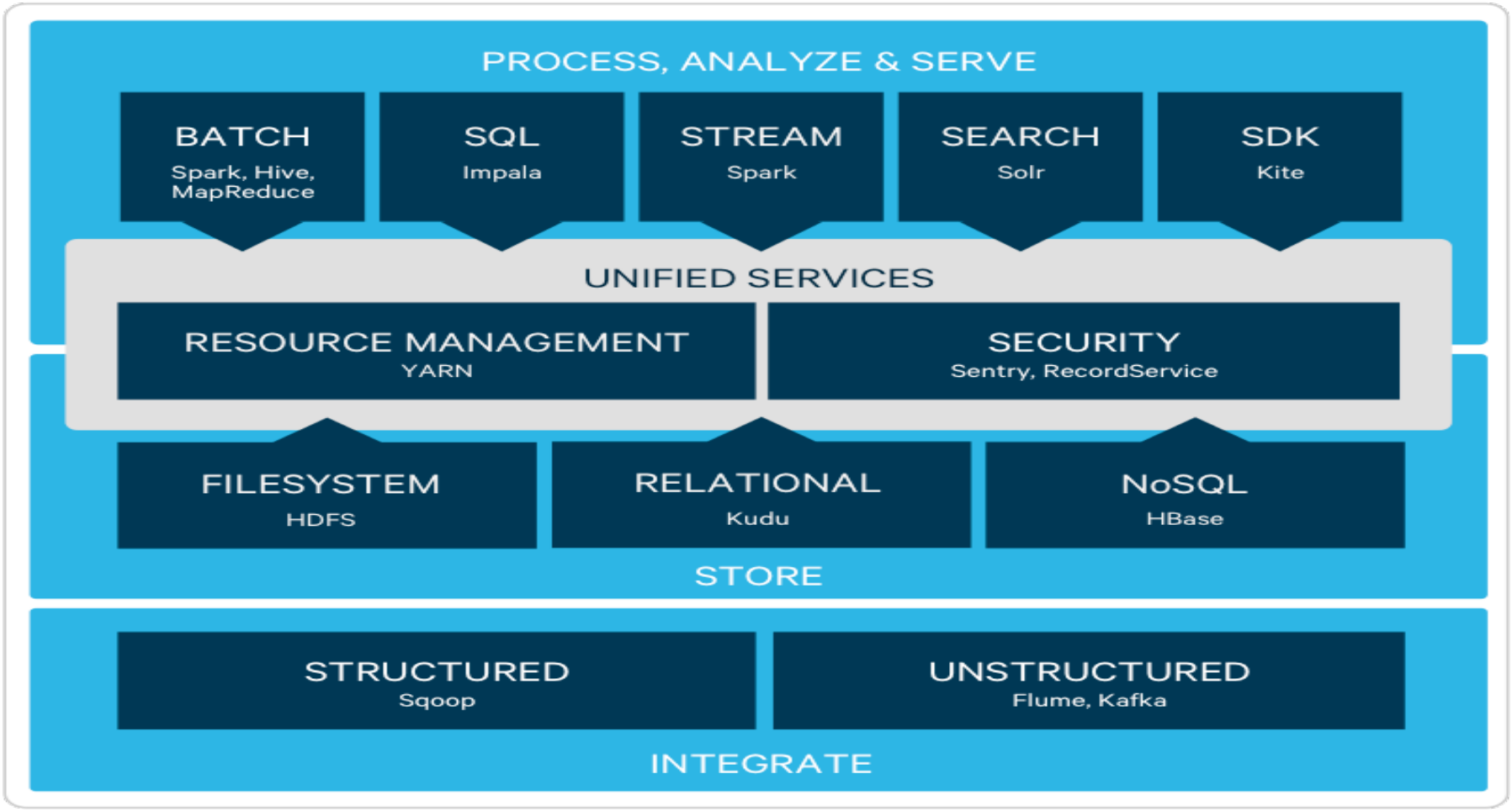
- ***fault tolerance***: the ability to withstand hardware or network failures (also: *resilience*)
- ***high availability***: this refers to the system minimizing downtimes by eliminating single points of failure
- ***data locality***: task are run on the node where data are located, in order to reduce the cost of moving data around

# The Hadoop core

The core of Hadoop consists of:

- Hadoop common, the core libraries
- HDFS, the Hadoop Distributed File System
- MapReduce
- the YARN (Yet Another Resource Negotiator) resource manager

# The Hadoop ecosystem




There's a whole constellation of open source components for collecting, storing, and processing big data that integrate with Hadoop.

Image source: Cloudera

# The Hadoop Distributed File System (HDFS)

HDFS stands for Hadoop Distributed File System and it takes care of partitioning data across a cluster.

In order to prevent data loss and/or task termination due to hardware failures HDFS uses either

- replication (creating multiple copies —usually 3— of the data)
- erasure coding

Data redundancy (obtained through replication or erasure coding) is the basis of Hadoop's fault tolerance.

# Replication vs. Erasure Coding

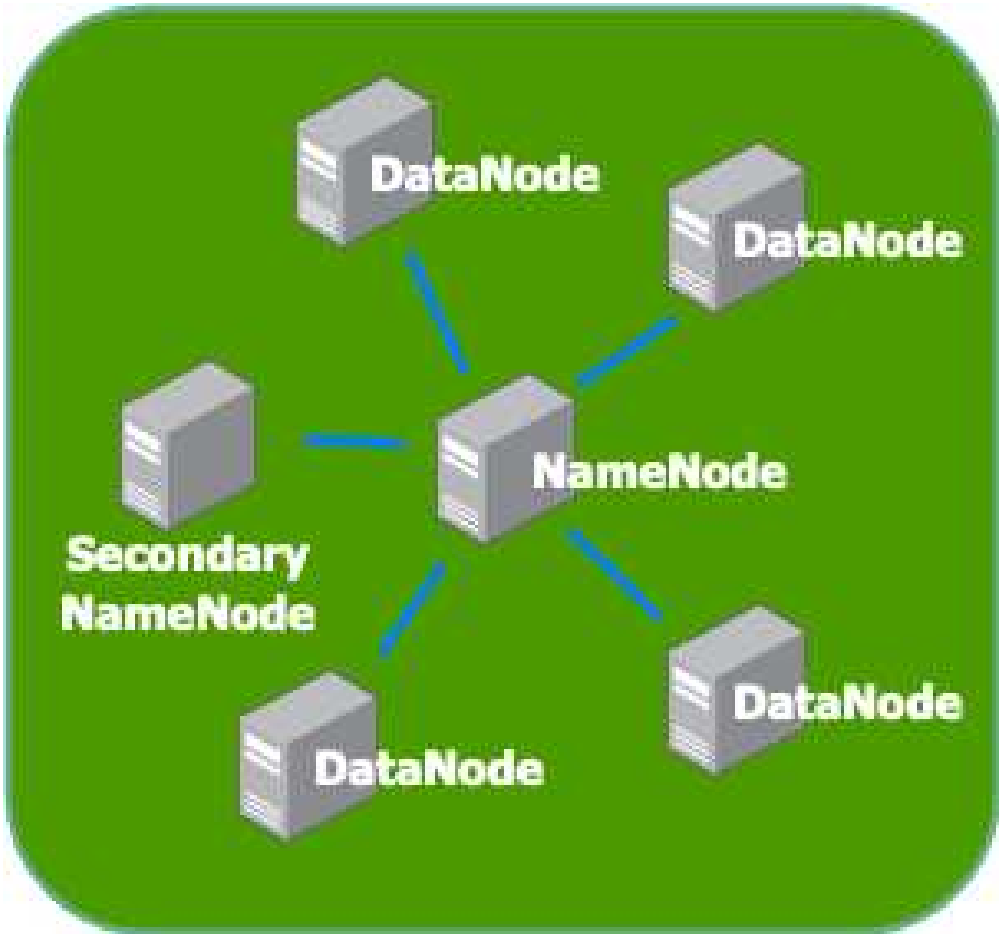
In order to provide protection against failures one introduces:

- data redundancy
- a method to recover the lost data using the redundant data

*Replication is the simplest method for coding data by making n copies of the data. n-fold replication guarantees the availability of data for at most* $n-1$ *failures and it has a storage overhead of 200% (this is equivalent to a storage efficiency of 33%).*

*Erasure coding provides a better storage efficiency (up to to 71%) but it can be more costly than replication in terms of performance.*

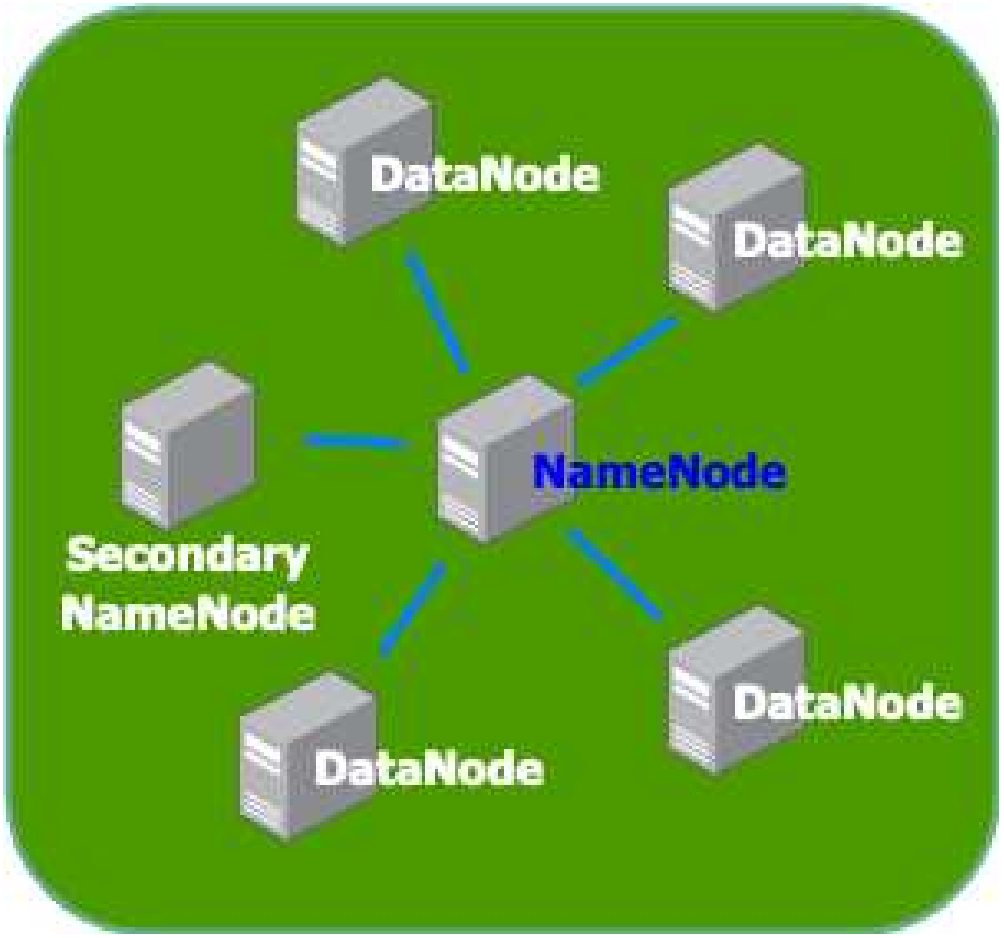
# HDFS architecture




A typical Hadoop cluster installation consists of:

- a NameNode
- a secondary NameNode
- multiple DataNodes

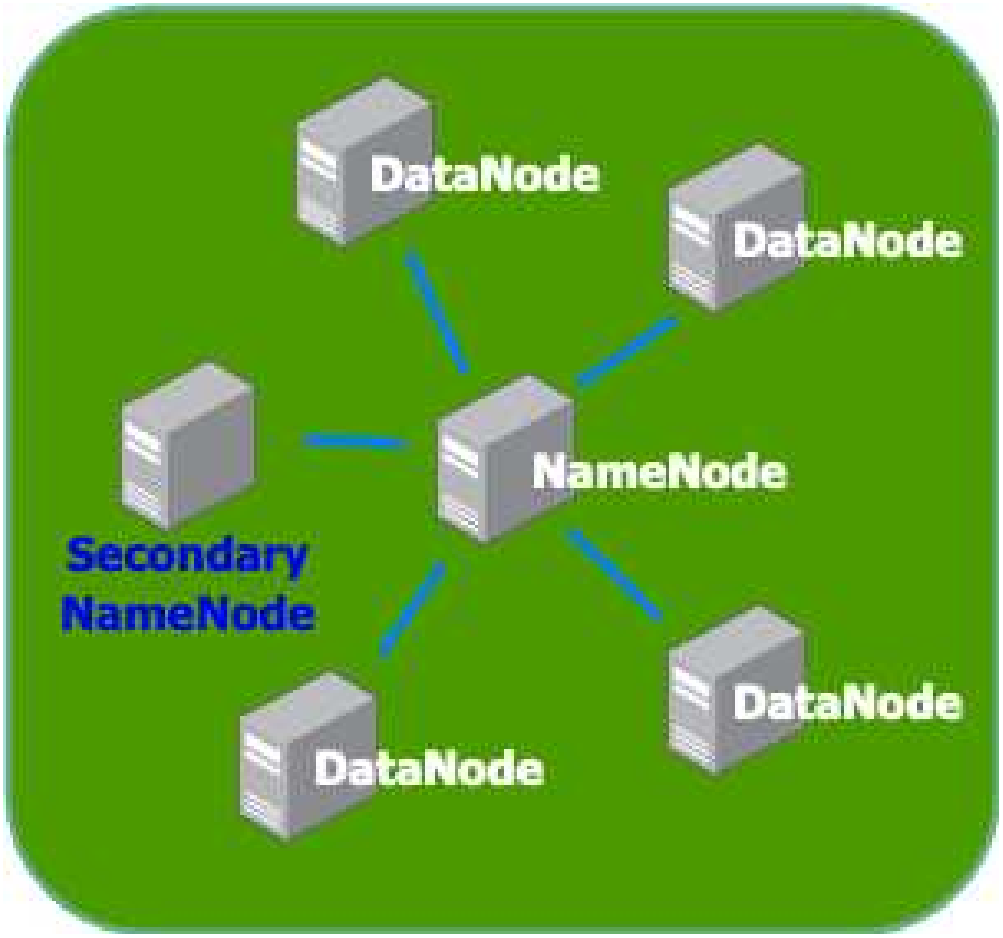
# HDFS architecture: NameNode




## NameNode

The NameNode is the main point of access of a Hadoop cluster. It is responsible for the bookkeeping of the data partitioned across the DataNodes, manages the whole filesystem metadata, and performs load balancing

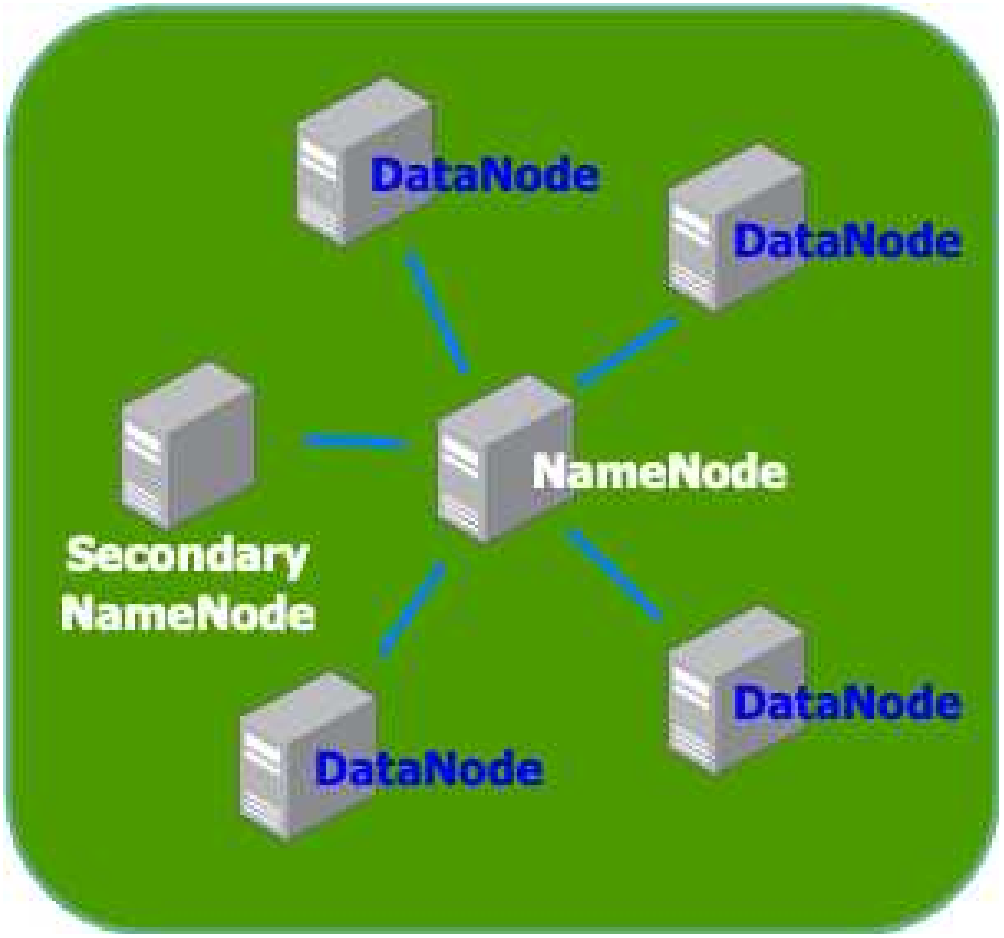
# HDFS architecture: Secondary NameNode




## Secondary NameNode

Keeps track of changes in the NameNode performing regular snapshots, thus allowing quick startup.
An additional *standby node* is needed to guarantee high availability (since the NameNode is a single point of failure).

# HDFS architecture: DataNode




## DataNode

Here is where the data is saved and the computations take place (data nodes should actually be called "data and worker nodes")

# HDFS architecture: internal data representation

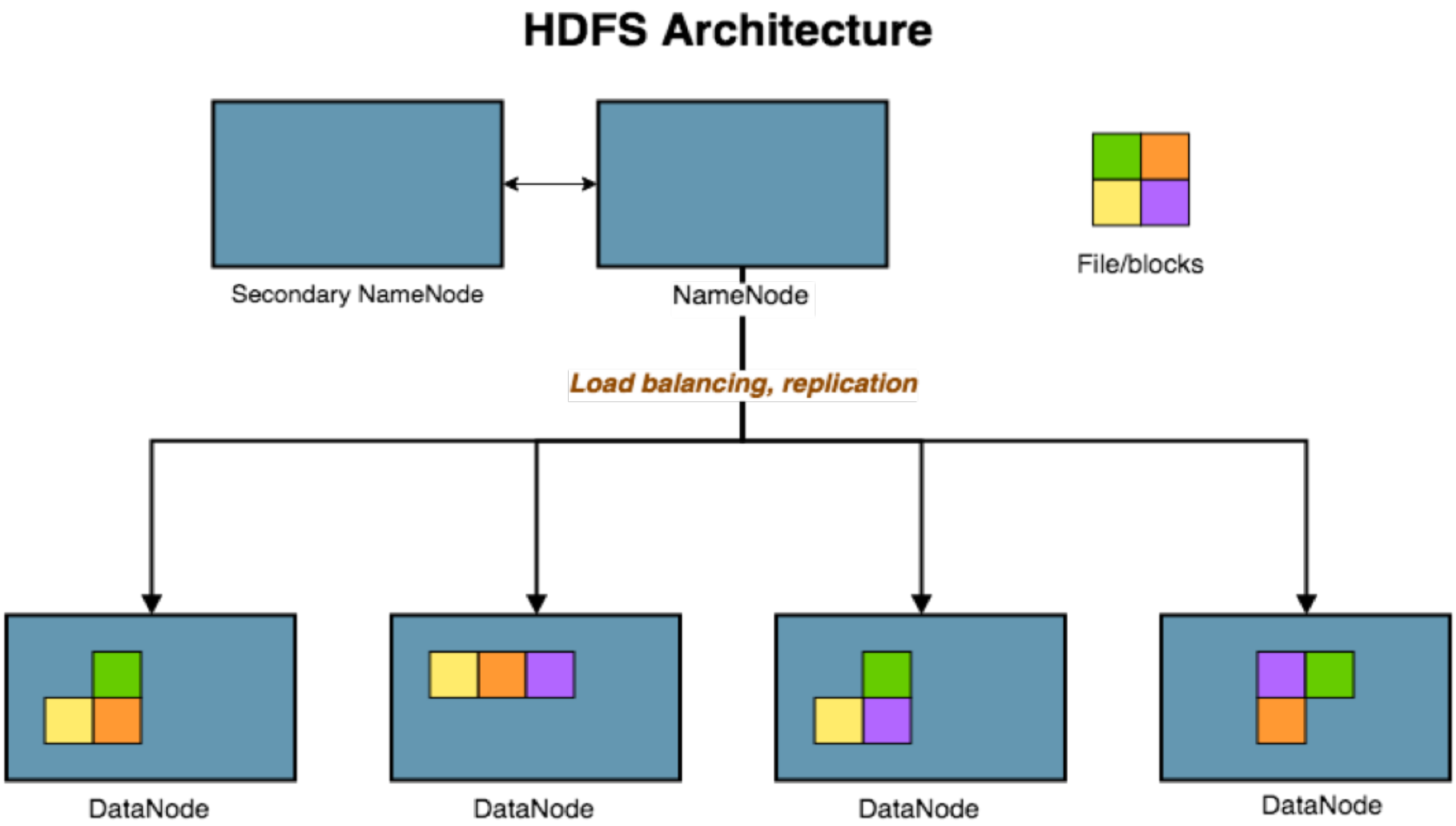
HDFS supports working with very large files.

Internally, data are split into *blocks*. One of the reason for splitting data into blocks is that in this way block objects all have the same size.

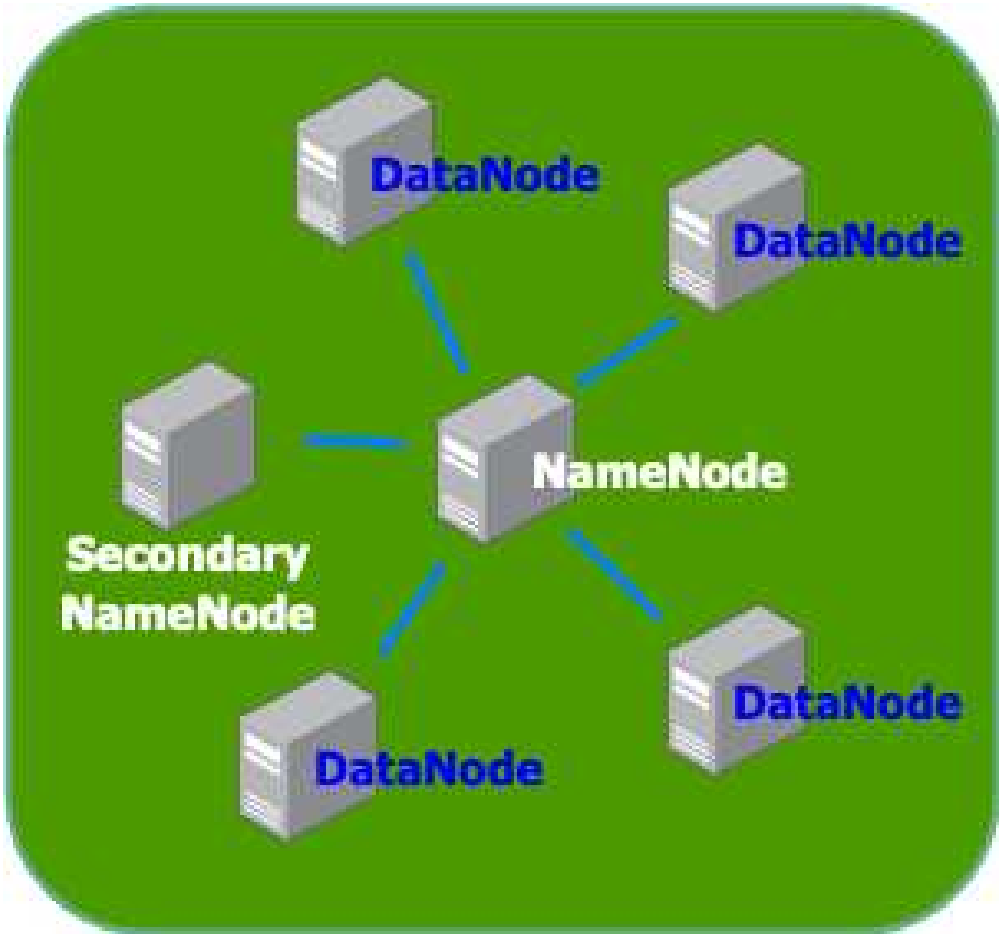
The block size in HDFS can be configured at installation time and it is by default 128MiB (approximately 134MB).

**Note:** Hadoop sees data as a bunch of records and it processes multiple files the same way it does with a single file. So, if the input is a directory instead of a single file, it will process all files in that directory.

# HDFS architecture



# DataNode failures




Each DataNode sends a Heartbeat message to the NameNode periodically. Whenever a DataNode becomes unavailable (due to network or hardware failure), the NameNode stops sending requests to that node and creates new replicas of the blocks stored on that node.

# The WORM principle of HDFS

The Hadoop Distributed File System relies on a simple design principle for data known as Write Once Read Many (WORM).

"*A file once created, written, and closed need not be changed except for appends and truncates. Appending the content to the end of the files is supported but cannot be updated at arbitrary point. This assumption simplifies data coherency issues and enables high throughput data access.* [5]"

The data immutability paradigm is also discussed in Chapter 2 of "Big Data".[6]

[5] Apache Software Foundation. *Hadoop*. url: https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html.

[6] Warren J. and Marz N. *Big Data*. Manning publications, 1988.

# HDFS hands-on exercises

# Basic HDFS filesystem commands

One can regard HDFS as a regular file system, in fact many HDFS shell commands are inherited from the corresponding bash commands.

To run a command on an Hadoop filesystem use the prefix `hdfs dfs`, for instance use:

```
hdfs dfs -mkdir myDir
```

to create a new directory `myDir` on HDFS.

**Note:** One can use interchangeably `hadoop` or `hdfs dfs` when working on a HDFS file system. The command `hadoop` is more generic because it can be used not only on HDFS but also on other file systems that Hadoop supports (such as Local FS, WebHDFS, S3 FS, and others).

# Basic HDFS filesystem commands

## Basic HDFS filesystem commands that also exist in bash

| | |
|---|---|
| `hdfs dfs -mkdir` | create a directory |
| `hdfs dfs -ls` | list files |
| `hdfs dfs -cp` | copy files |
| `hdfs dfs -cat` | print files |
| `hdfs dfs -tail` | output last part of a file |
| `hdfs dfs -rm` | remove files |

# Basic HDFS filesystem commands

Here's three basic commands that are specific to HDFS.

| | |
|---|---|
| `hdfs dfs -put` | Copy single src, or multiple srcs from local file system to the destination file system |
| `hdfs dfs -get` | Copy files to the local file system |
| `hdfs dfs -usage` | get help on hadoop fs |

# Basic HDFS filesystem commands

To get more help on a specific `hdfs` command use: `hdfs -help <command>`

```
$ hdfs dfs -help tail
# -tail [-f] <file> :
#   Show the last 1KB of the file.

#   -f  Shows appended data as the file grows.
```

# Some things to try

```
# create a new directory called "input" on HDFS
hdfs dfs -mkdir input
# copy local file wiki_1k_lines to input on HDFS
hdfs dfs -put wiki_1k_lines input/
# list contents of directory ("-h" = human)
hdfs dfs -ls -h input
# disk usage
hdfs dfs -du -h input
# get help on "du" command
hdfs dfs -help du
# remove directory
hdfs dfs -rm -r input
```

# Some things to try

What is the size of the file `wiki_1k_lines`? What is its disk usage?

```
# show the size of wiki_1k_lines on the regular filesystem
ls -lh wiki_1k_lines
# show the size of wiki_1k_lines on HDFS
hdfs dfs -put wiki_1k_lines
hdfs dfs -ls -h wiki_1k_lines

# disk usage of wiki_1k_lines on the regular filesystem
du -h wiki_1k_lines
# disk usage of wiki_1k_lines on HDFS
hdfs dfs -du -h wiki_1k_lines
```

# Disk usage on HDFS

The command `hdfs dfs -help du` will tell you that the output is of the form:

`size disk space consumed filename.`

You'll notice that the space on disk is larger than the file size (38.6MB versus 19.3MB):

```
hdfs dfs -du -h wiki_1k_lines
# 19.3 M  38.6 M  wiki_1k_lines
```

This is due to replication. You can check the replication factor using:

```
hdfs dfs -stat 'Block size: %o Blocks: %b Replication: %r'
    input/wiki_1k_lines
# Block size: 134217728 Blocks: 20250760 Replication: 2
```

# Disk usage on HDFS

From the previous output:

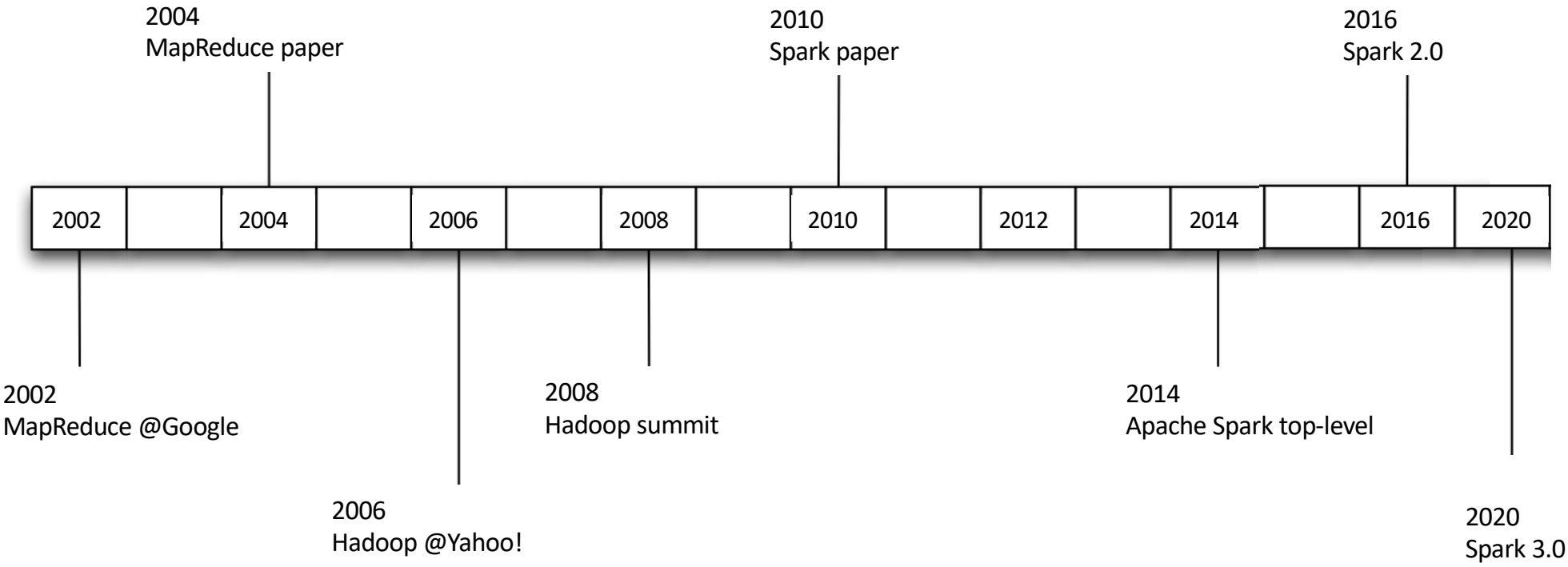
```
Block size: 134217728 Blocks: 20250760 Replication: 2
```

we can see that the HDFS filesystem currently supports a replication factor of 2.

Note that the Hadoop block size is defined in terms of *mebibytes*, in fact 134217728 bytes corresponds to 128MiB and 134MB. One MiB is larger than a MB since one MiB is $1024^2 = 2^{20}$ bytes, while one MB is $10^6$ bytes.

# From Map Reduce & Apache Hadoop to Apache Spark

# A Brief History



# Why interest on Map-reduce?


- MapReduce offers a **scalable distributed** computing platform for handling **large volumes** of data and mining petabytes of **unstructured information**

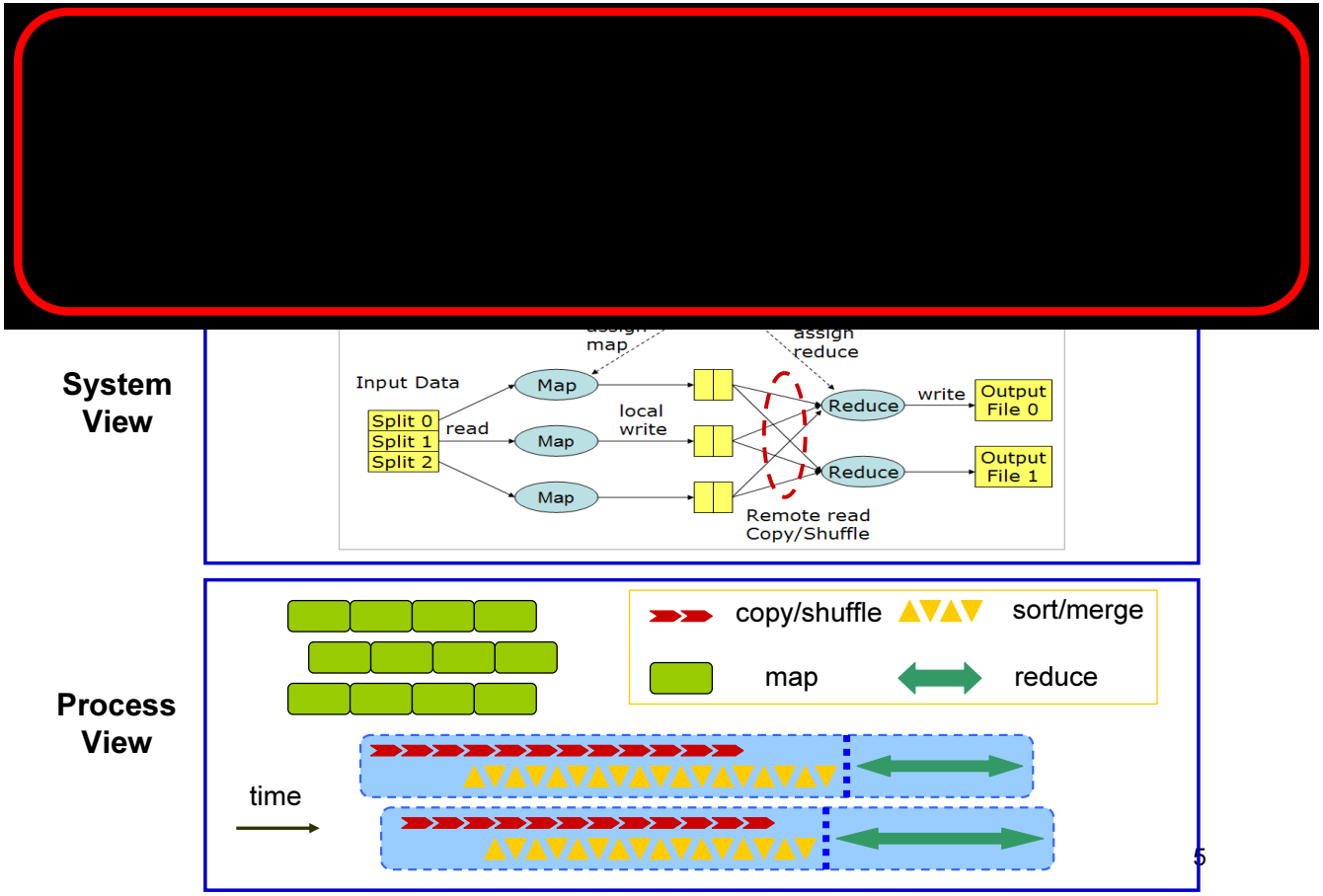
- It has been used to support **data analytics** applications, e.g., **sentiment**, **clickstream**, **sensors**, **geo-location**, **server logs** analyses

- We take a historical perspective, the core ideas are seminal also for modern frameworks

# Early days motivations for MapReduce

- Google need for large-Scale Data Processing:
  - Want to use 1000s of CPUs
  - But don't want hassle of ***managing*** things
- Map-reduce Architecture provides:
  - **Automatic** parallelization & distribution
  - **Fault tolerance**
  - I/O **scheduling**
  - **Monitoring** & status updates

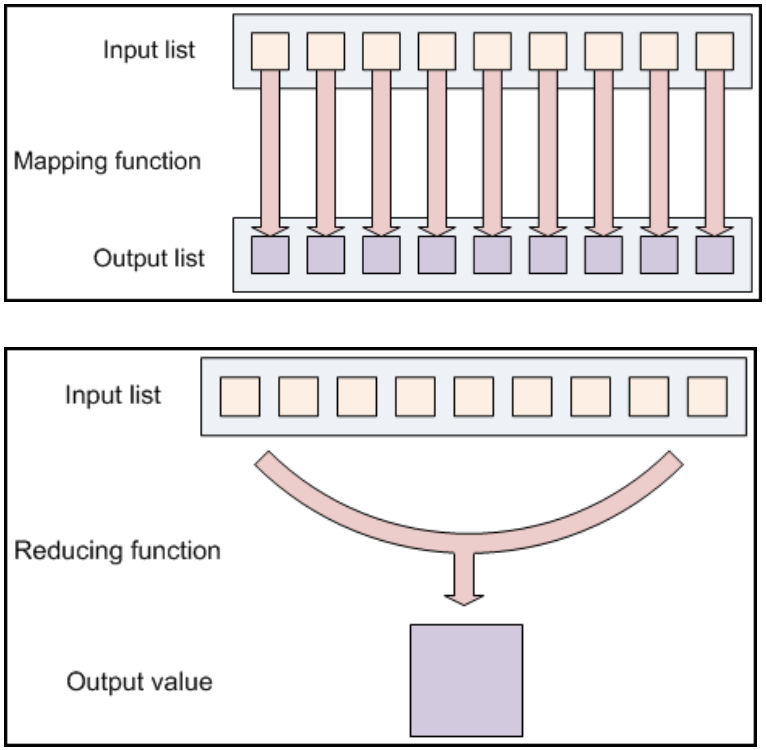
# Map-reduce overview




Courtesy of IBM

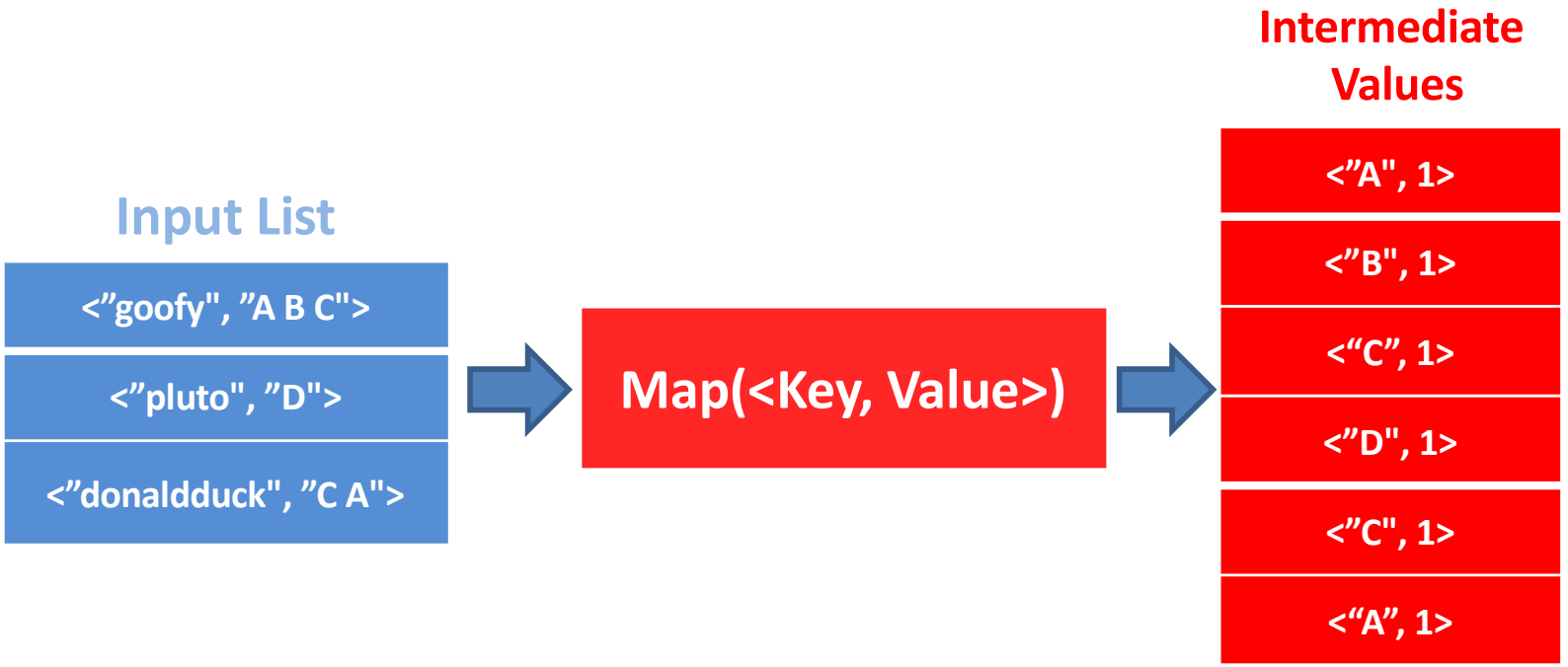
# Function view

- Input: a **set** of **key/value pairs**
- User supplies two functions:
  - **map(k,v) → list(k1,v1)**
  - **reduce(k1, list(v1)) → v2**
- **(k1,v1)** is an **intermediate** key/value pair
- **Output** is the **set** of **(k1,v2)** pairs

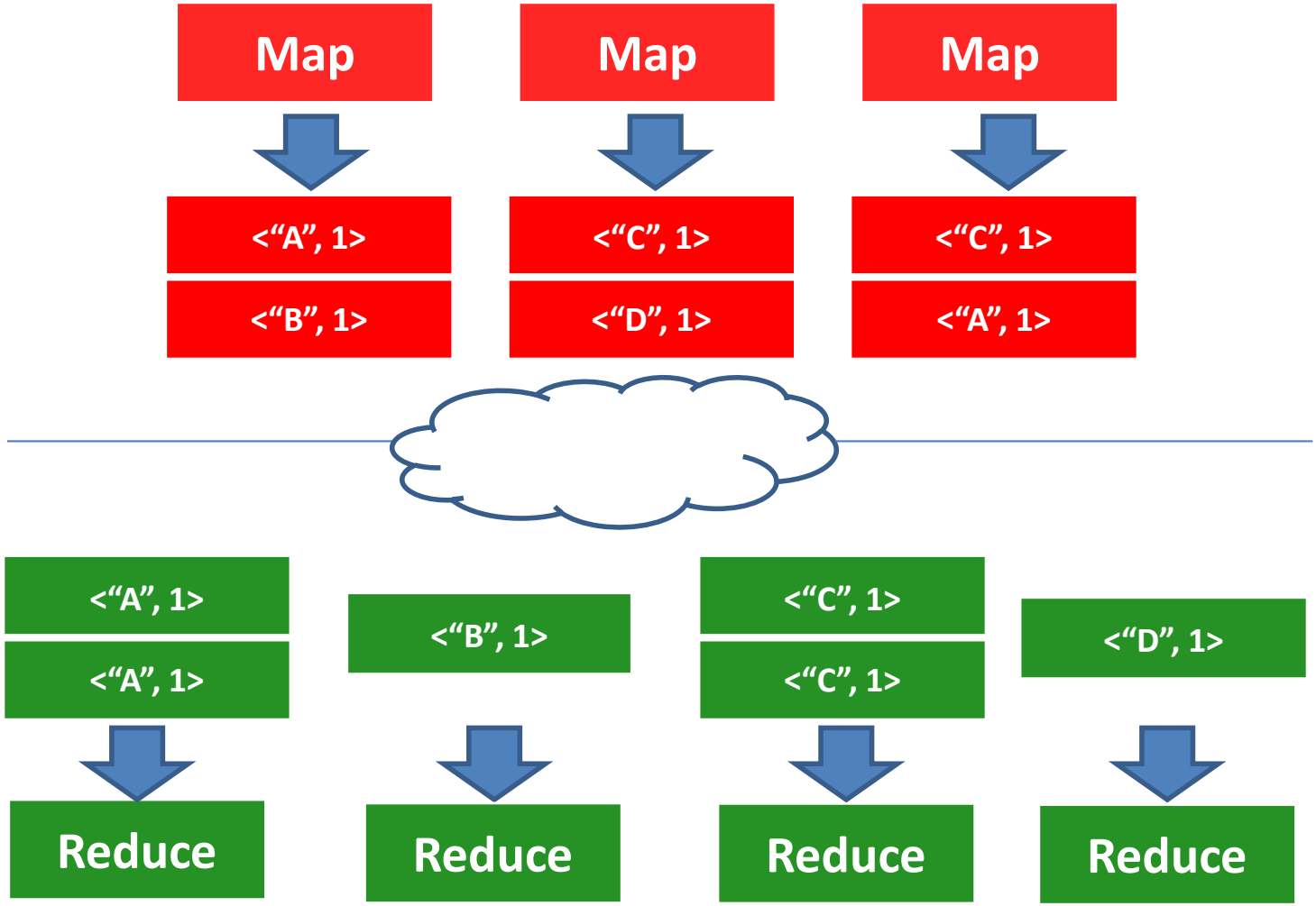


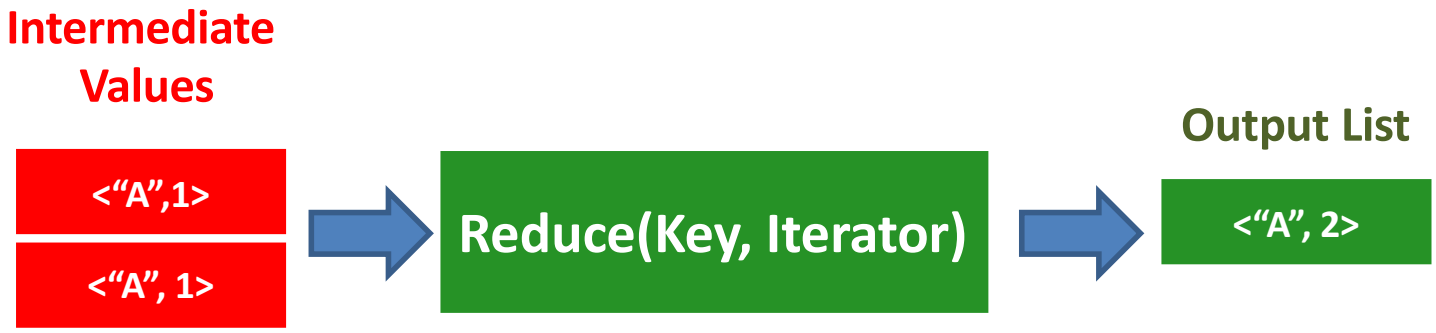
# An example: Word Count


- We have many large files of words
- **Count** the number of times each **distinct word** appears in the files
- *Sample application*: analyze web server logs to find **popular URLs**



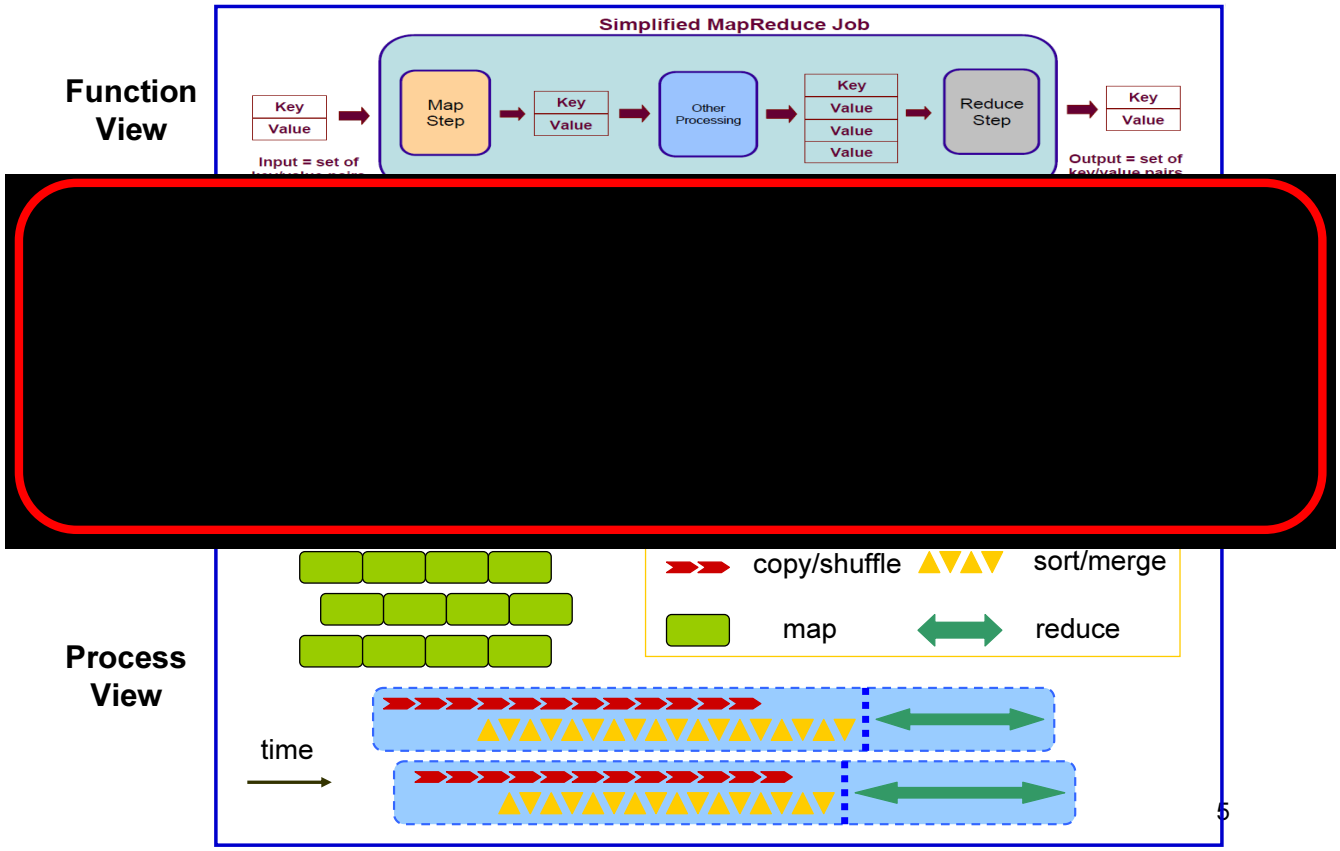

```
let map(String document_content)=
foreach Word word in document_content :
        emit(word, 1)
```


Map
Map
Map
<“A”, 1>
<“B”, 1>
<“C”, 1>
<“D”, 1>
<“C”, 1>
<“A”, 1>
<“A”, 1>
<“A”, 1>
<“B”, 1>
<“C”, 1>
<“C”, 1>
<“D”, 1>
Reduce
Reduce
Reduce
Reduce



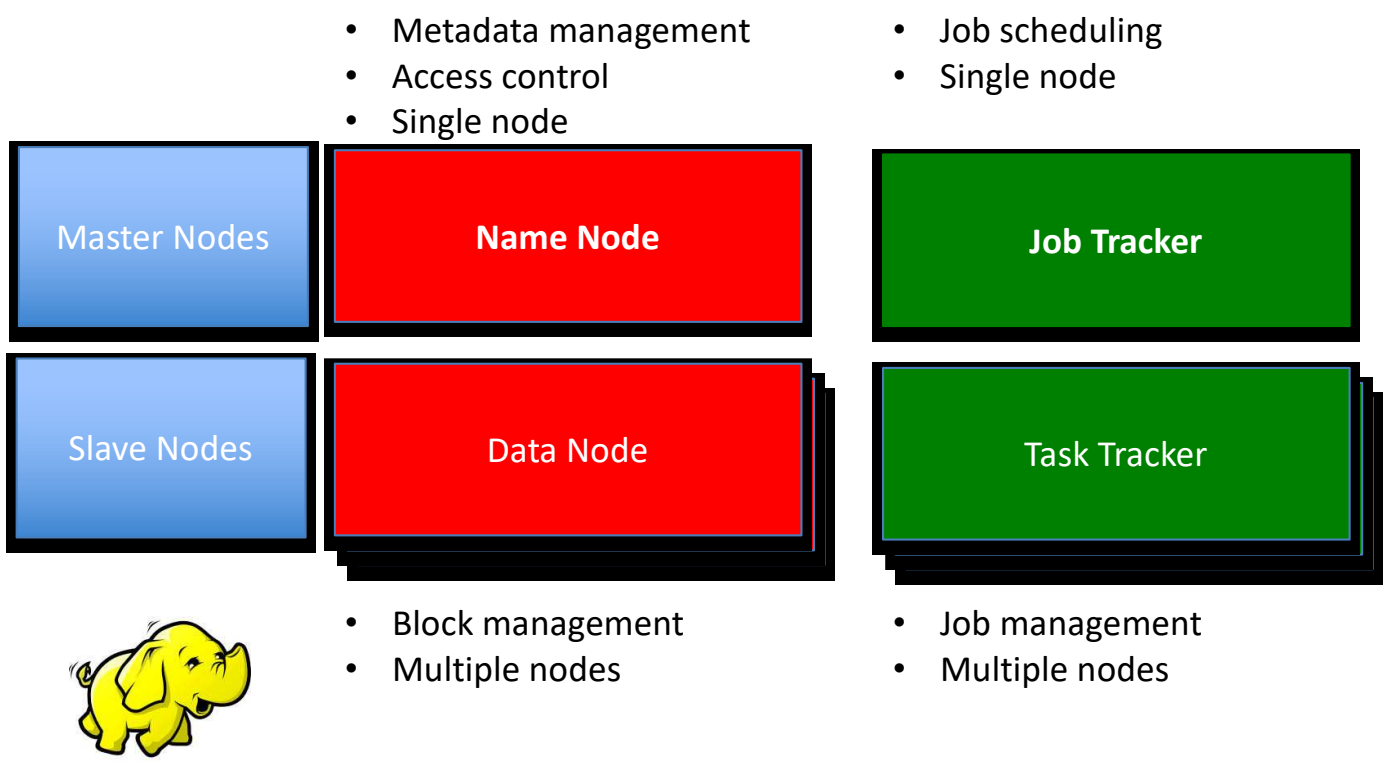

```
let reduce(Word word, Iterator<int> occurences) =
int total_occurences = 0;
foreach int o in occurences :
        total_occurences += o;
emit(word, total_occurences);
```

# Map-reduce overview

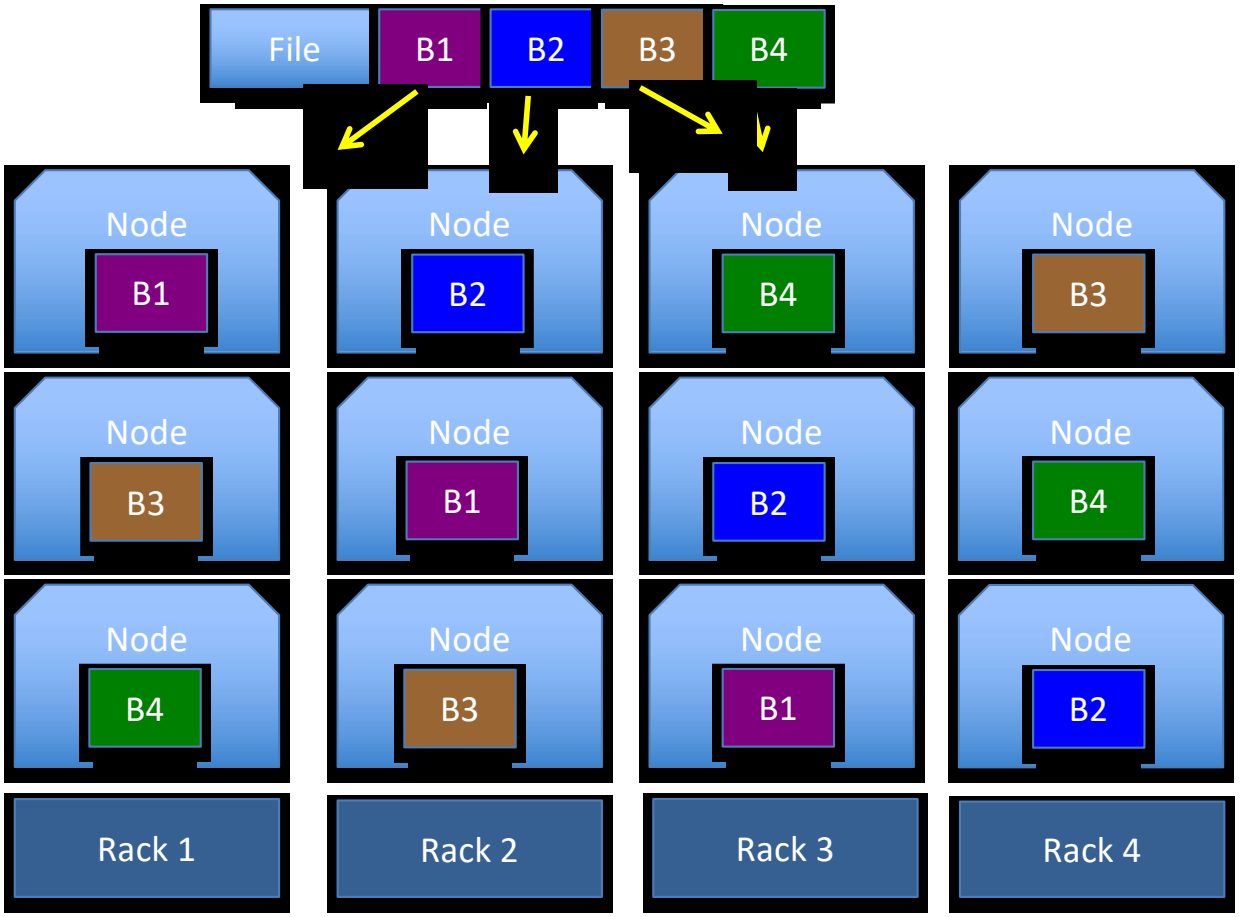



Courtesy of IBM

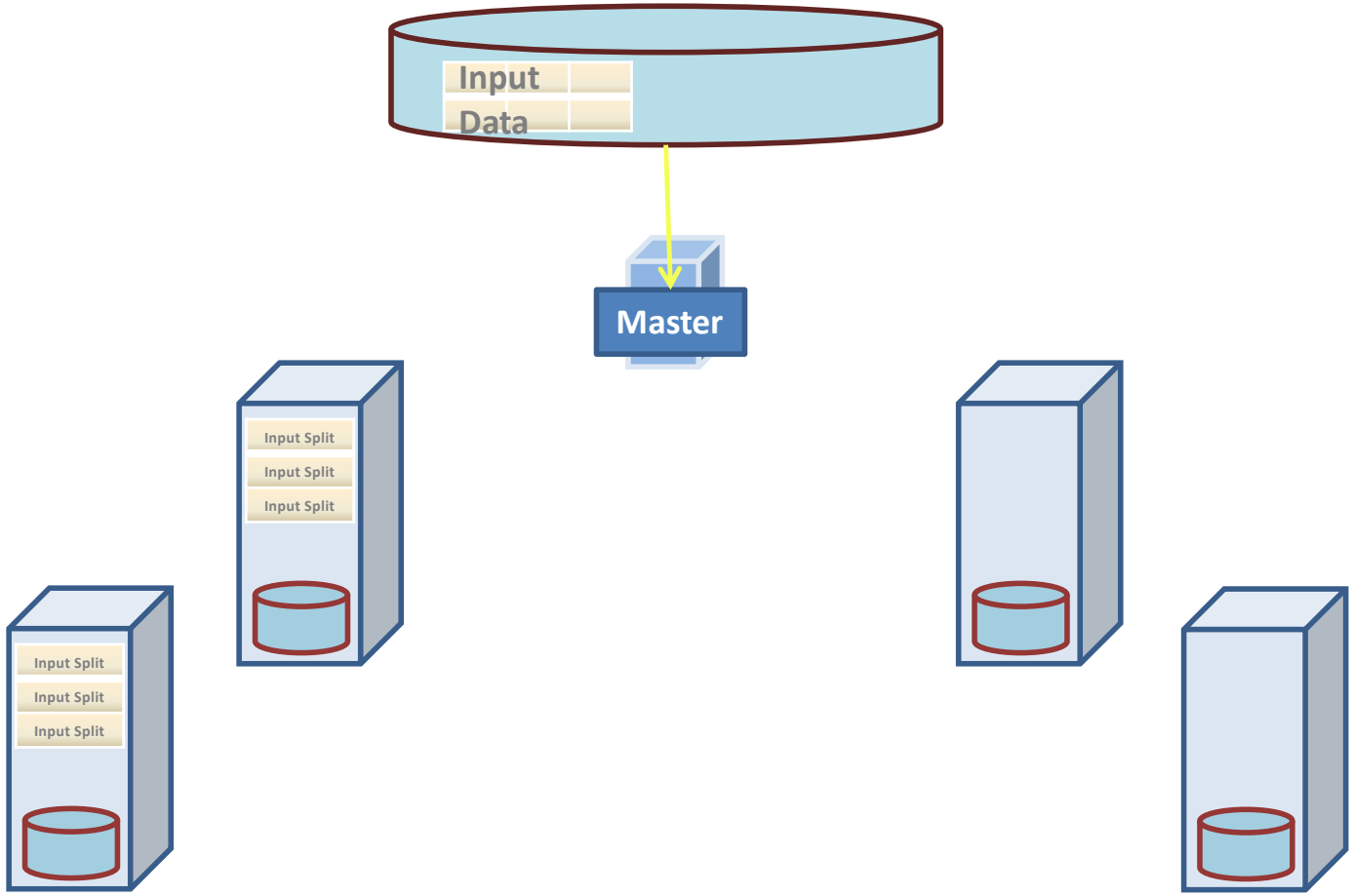
# System view: Hadoop 1.0 implementation



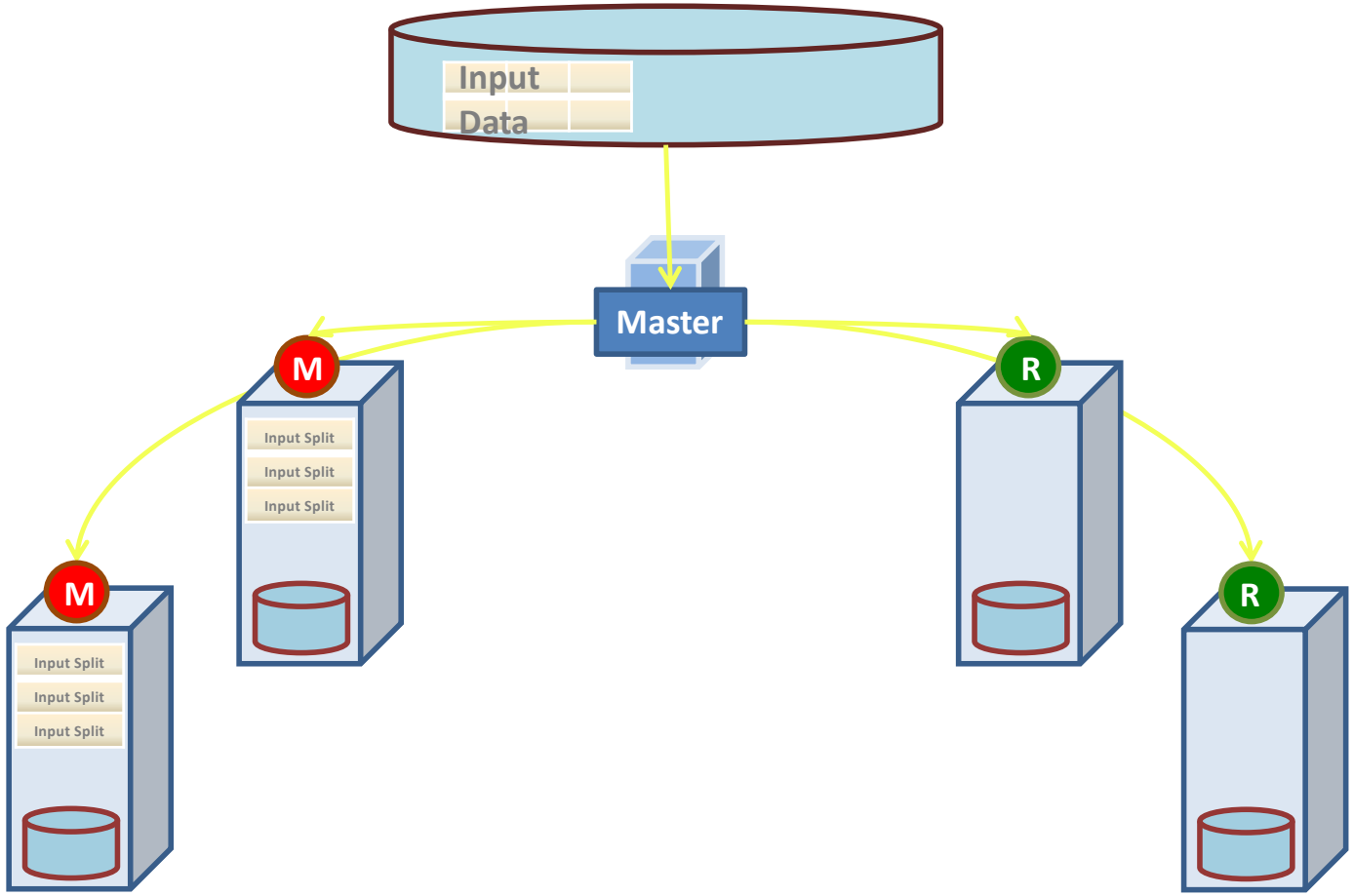
# Storing data: HDFS



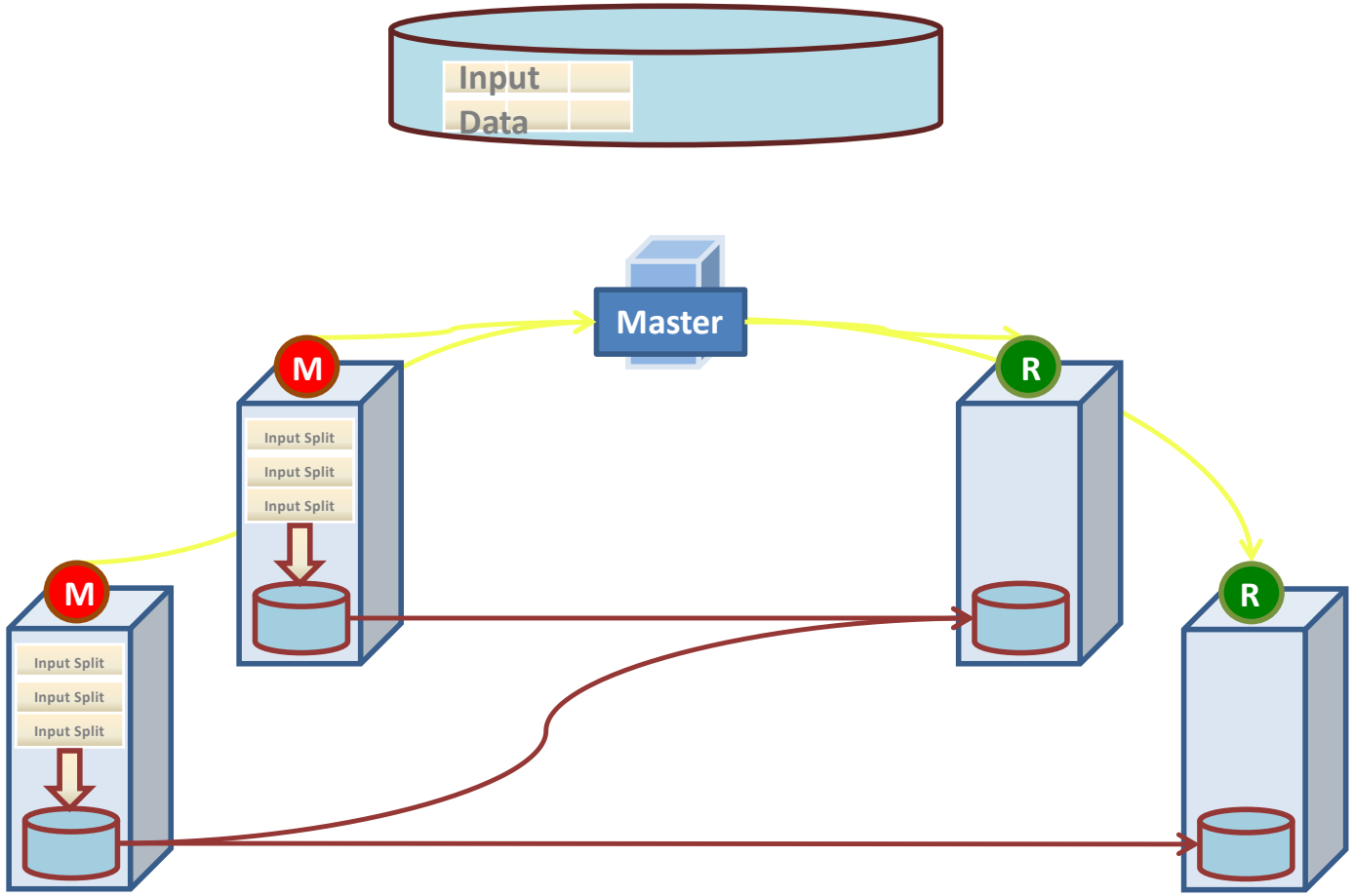
# System view – Execution (Hadoop 1.0)



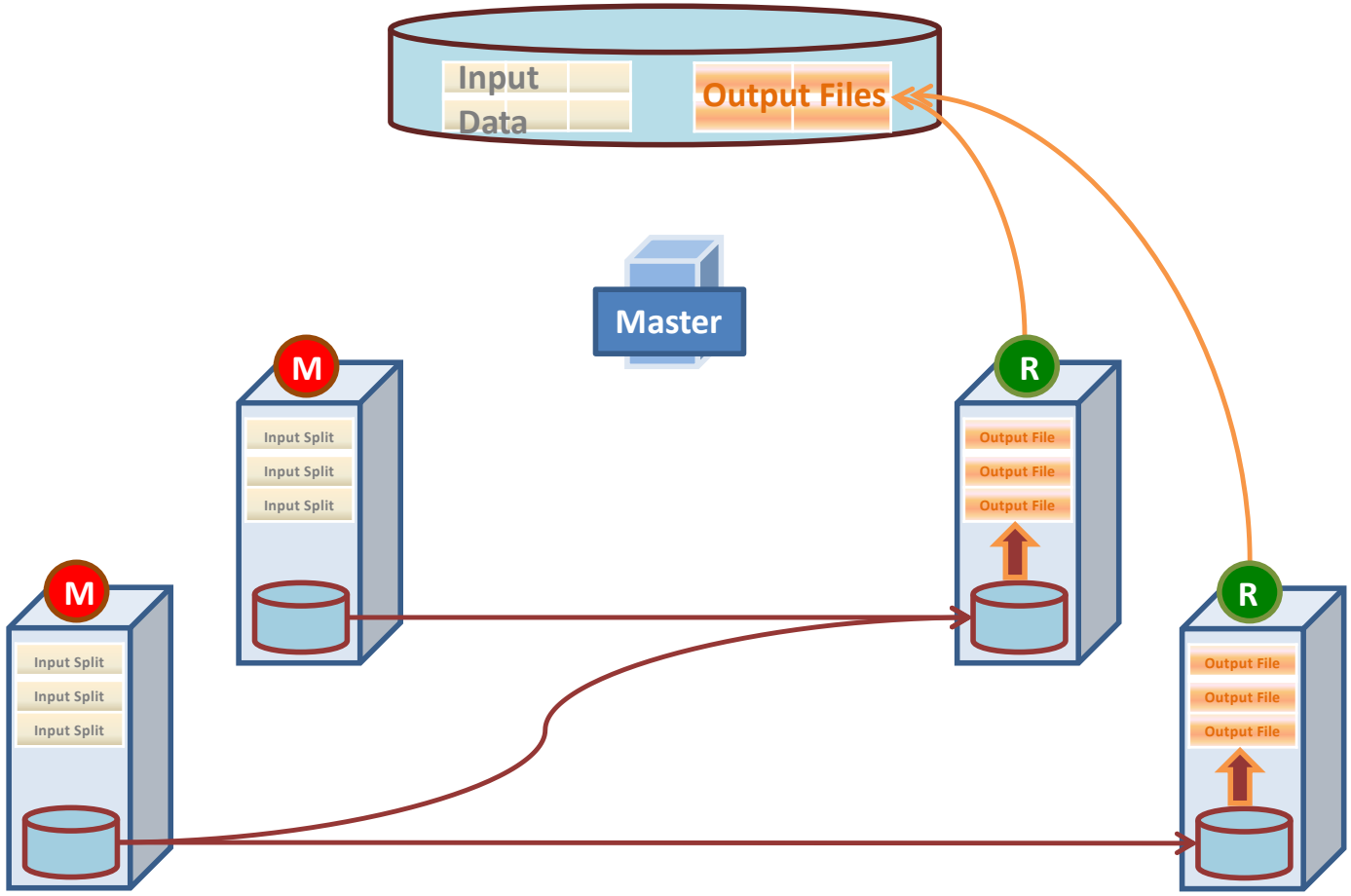
# System view – Execution (Hadoop 1.0)



# System view – Execution (Hadoop 1.0)



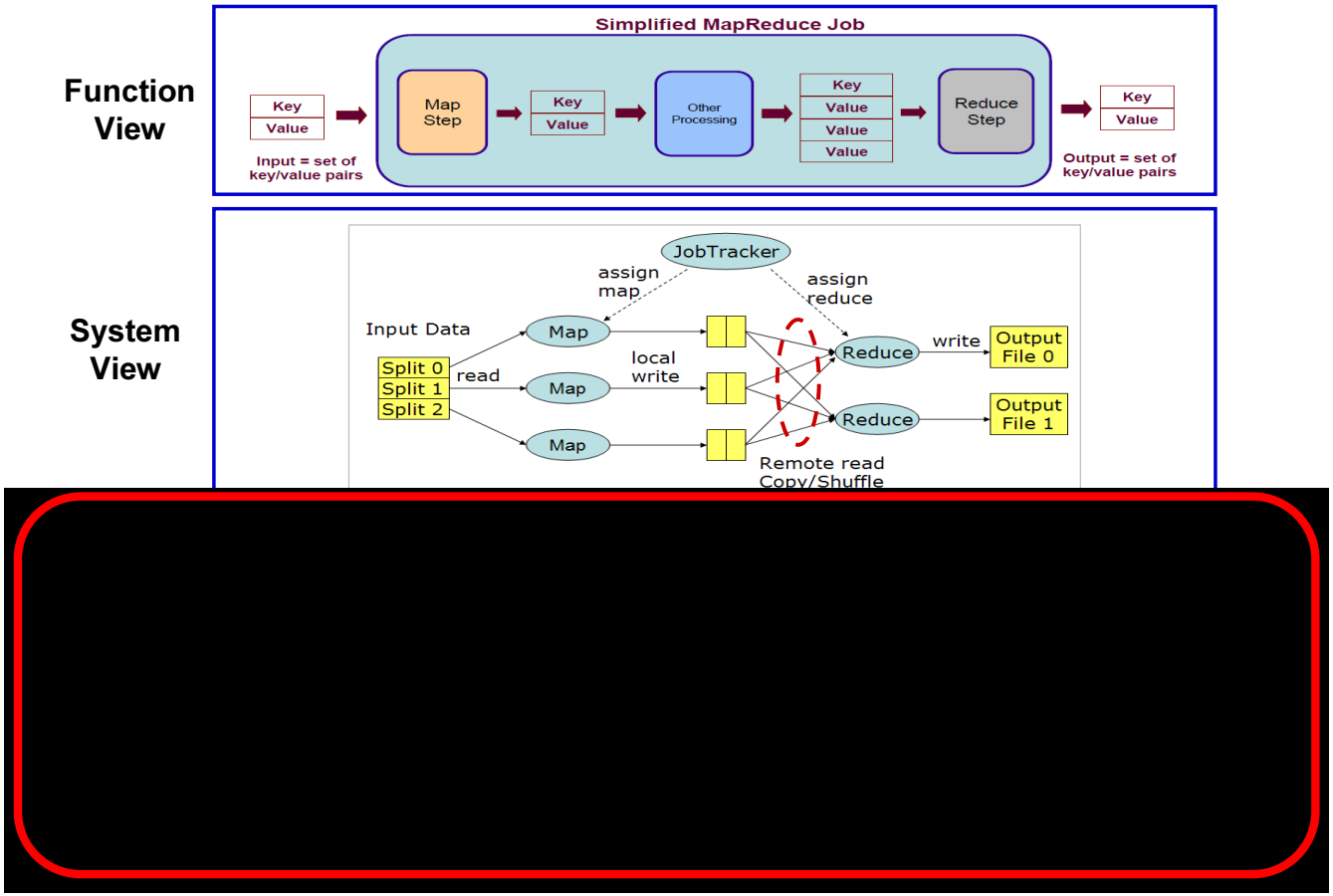
# System view – Execution (Hadoop 1.0)



# Failures

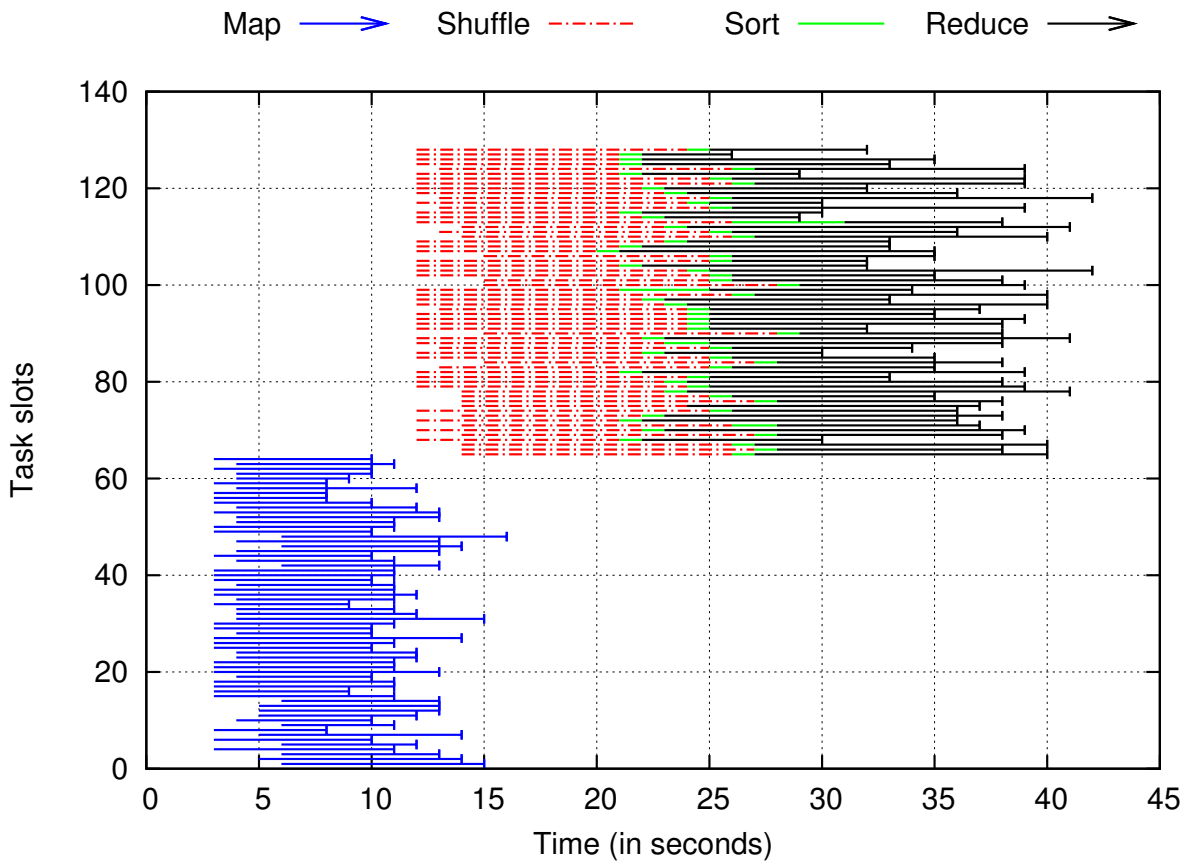
- Master **pings workers** periodically to detect failures
- Map worker failure:
  - Possible intermediate data loss
  - Map tasks restarted on another node with the same input splits
- Reduce worker failure:
  - Reducer tasks are restarted on another node
- **Highly fault-tolerant framework**

# Map-reduce overview

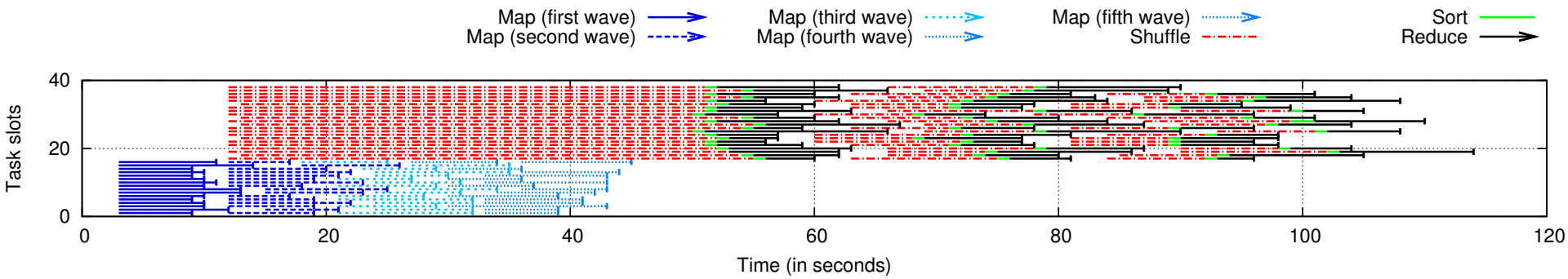



Courtesy of IBM

# Process view



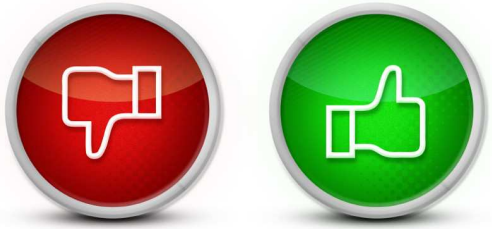
# Process view




Source: A. Verma, L. Cherkasova, R.H. Campbell.
ARIA: Automatic Resource Inference and Allocation for mapreduce environments, ICAC 2011.

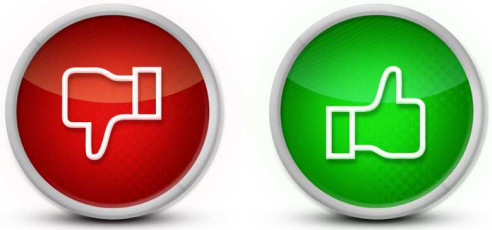
# Hadoop Pros & Cons

# Hadoop 1.0 pros



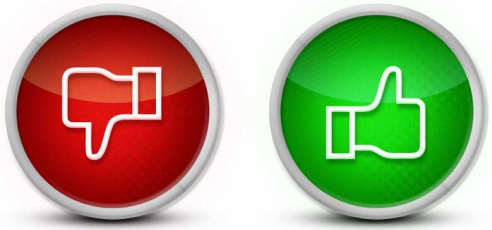
- "Simple" and easy to use
  - A programmer defines his job with only map and reduce functions, **without** having to **specify physical distribution** of his **job** across nodes
- Fault tolerance
  - MapReduce is **highly fault-tolerant** (continue to work in spite of an average of 1.2 failures per analysis job at Google)
- Flexible
  - **No dependency on data model** and schema (good for irregular or unstructured data)

# Hadoop 1.0 cons



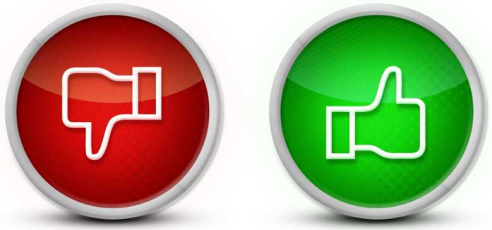
- A **single fixed dataflow**
  - The dataflow is fixed, many **complex algorithms** are **hard** to **implement** in a single job
  - Some algorithms that **require multiple inputs are not well supported** since the dataflow of MapReduce is originally designed to read a single input and generate a single output
- No high-level language
  - **No declarative language** like SQL in DBMS and any query optimization technique
- No schema and **no index**
  - Each item is parsed at reading input and transform it into data objects for data processing, causing performance degradation

# Hadoop 1.0 cons



- Low efficiency
  - With fault-tolerance and scalability as its primary goals, MapReduce operations are **not** always **optimized for I/O efficiency**. In addition, Map and Reduce are **blocking** operations
  - **No** specific **execution** plans and **does not optimize** plans like DBMS does to minimize **data transfer** across nodes

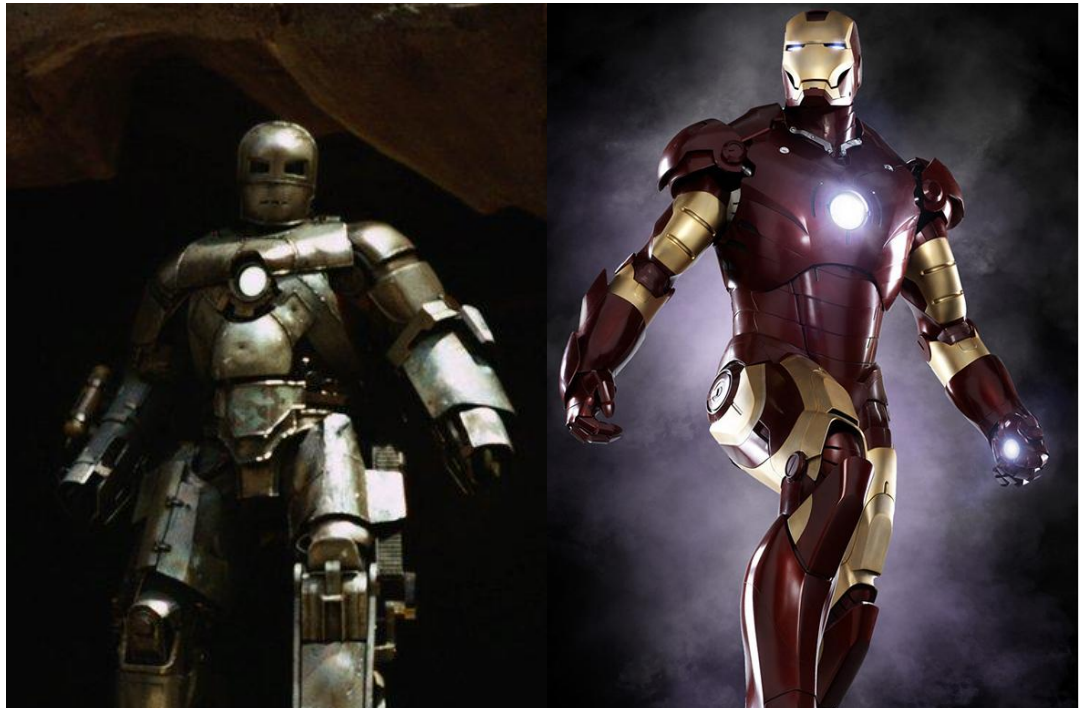
# Comparing Hadoop with commercial data warehouses



- Hadoop **2~50 times slower**, except in data loading
- Hadoop 1.0 was scalable but achieved **very low efficiency per node**, around 5MB/s processing rate, repeating a mistake that previous studies on HPC did *focusing on scalability but missing efficiency*

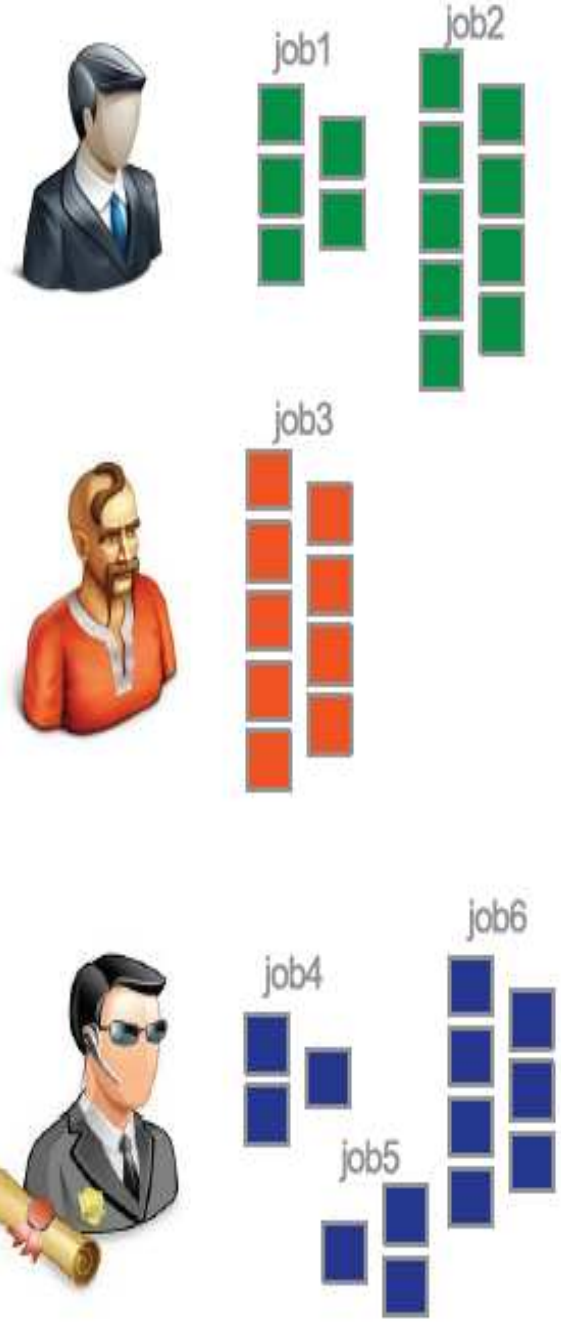
Source: K. H. Lee, Y. J. Lee, H. Choi, Y. D. Chung, and B. Moon. Parallel data processing with MapReduce: a survey. *SIGMOD Rec.* 40(4), 2011.

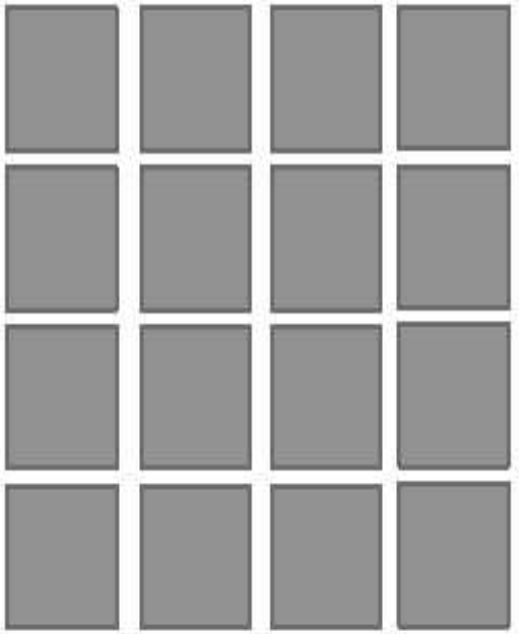
# Hadoop evolution



Hadoop 1.0 → Hadoop 3.2.1

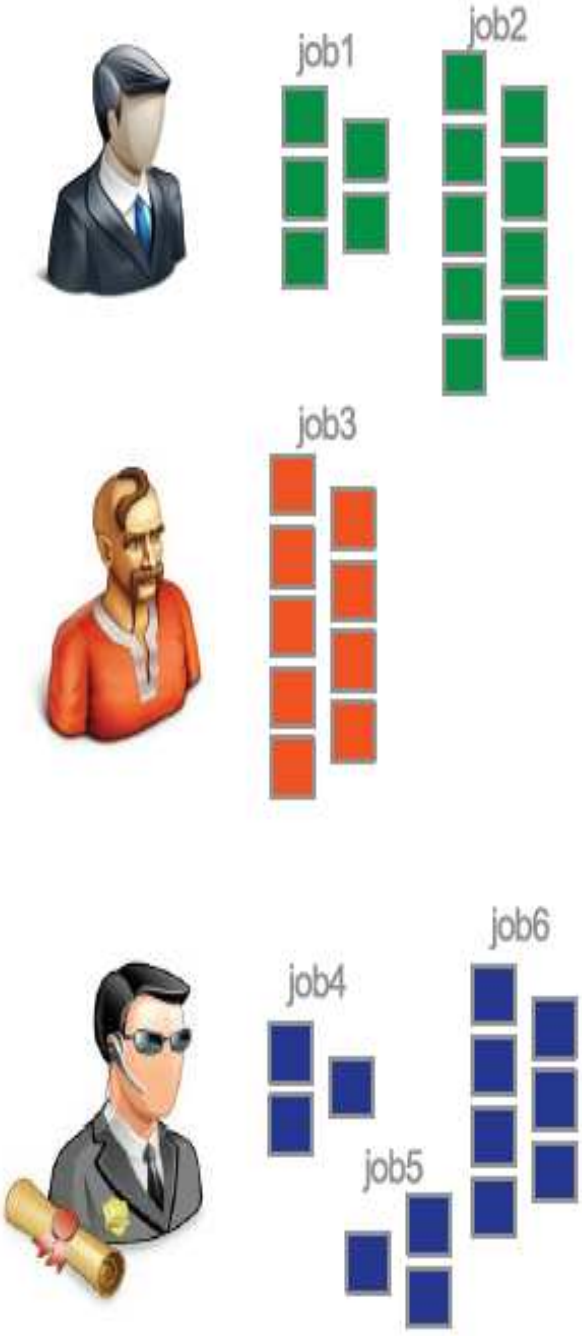
# Hadoop 1.0




Courtesy of Microsoft

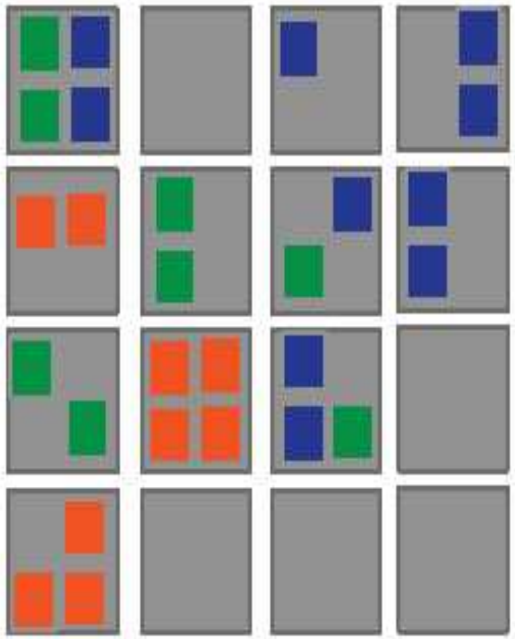
# Hadoop 1.0




Problems to solve

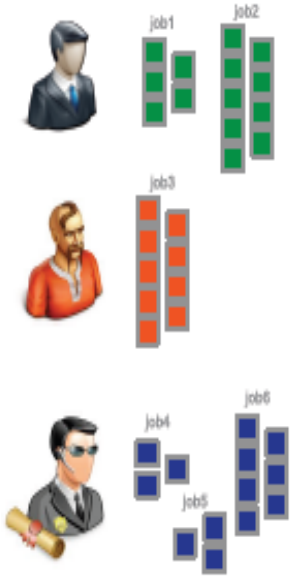
- Who runs?
- Where?
- How much resources?
- Order of execution
- Monitor progress
- Handle failures

Large Shared Cluster



Courtesy of Microsoft

# Hadoop 2.x




Resource Management

- Who runs?
- Where?
- How much resources?

→ Goals

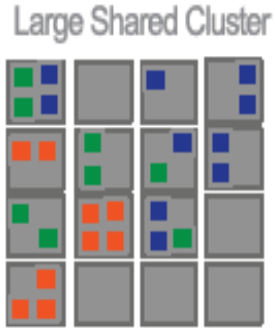
- Cluster throughput
- Scheduling Invariants
- Efficiency (locality)

Application Workflow

- Order of execution
- Monitor progress
- Handle failures

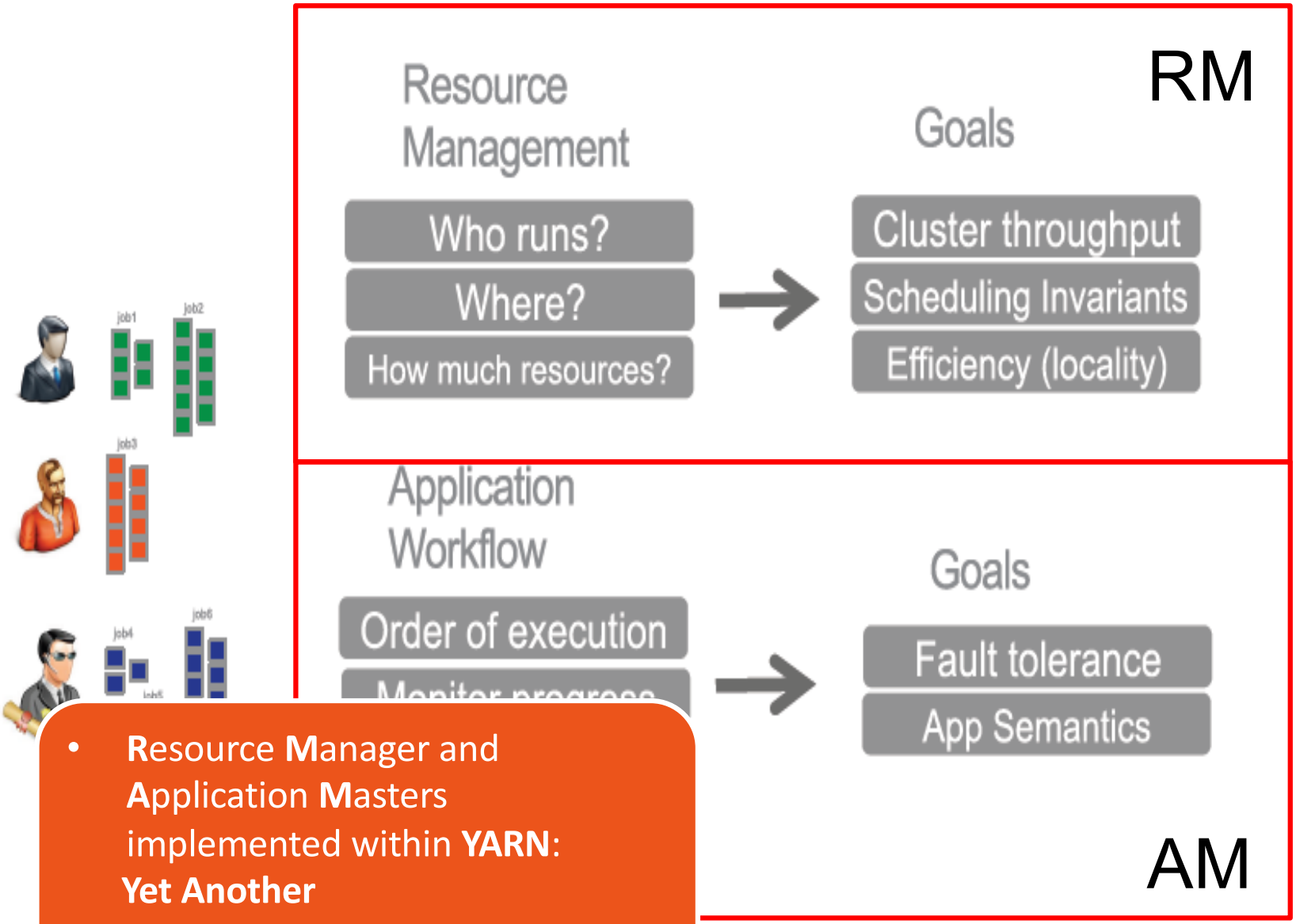
→ Goals

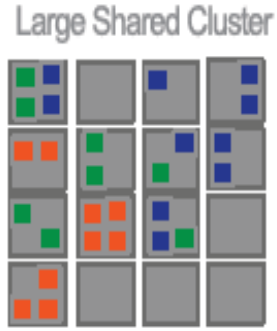
- Fault tolerance
- App Semantics




Courtesy of Microsoft

# Hadoop 2.x



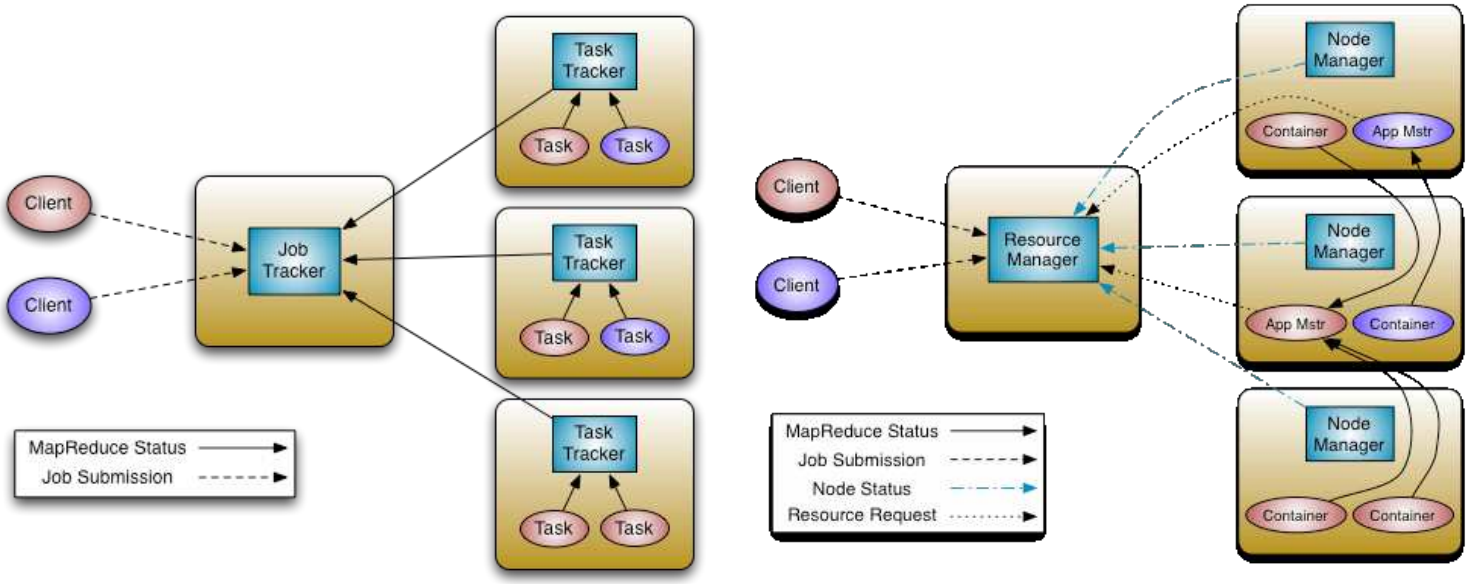




- **R**esource **M**anager and **A**pplication **M**asters implemented within **YARN**: **Yet Another Resource Negotiator**

Courtesy of Microsoft

# Computational nodes architecture

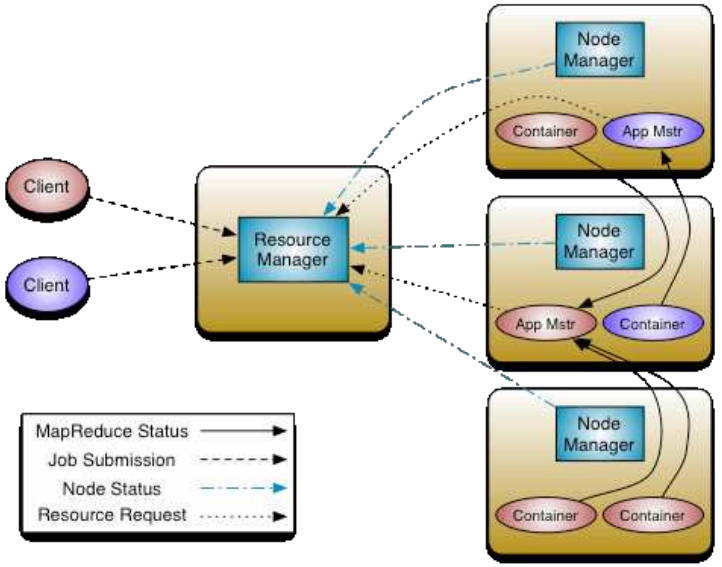
- Job and task trackers have been replaced



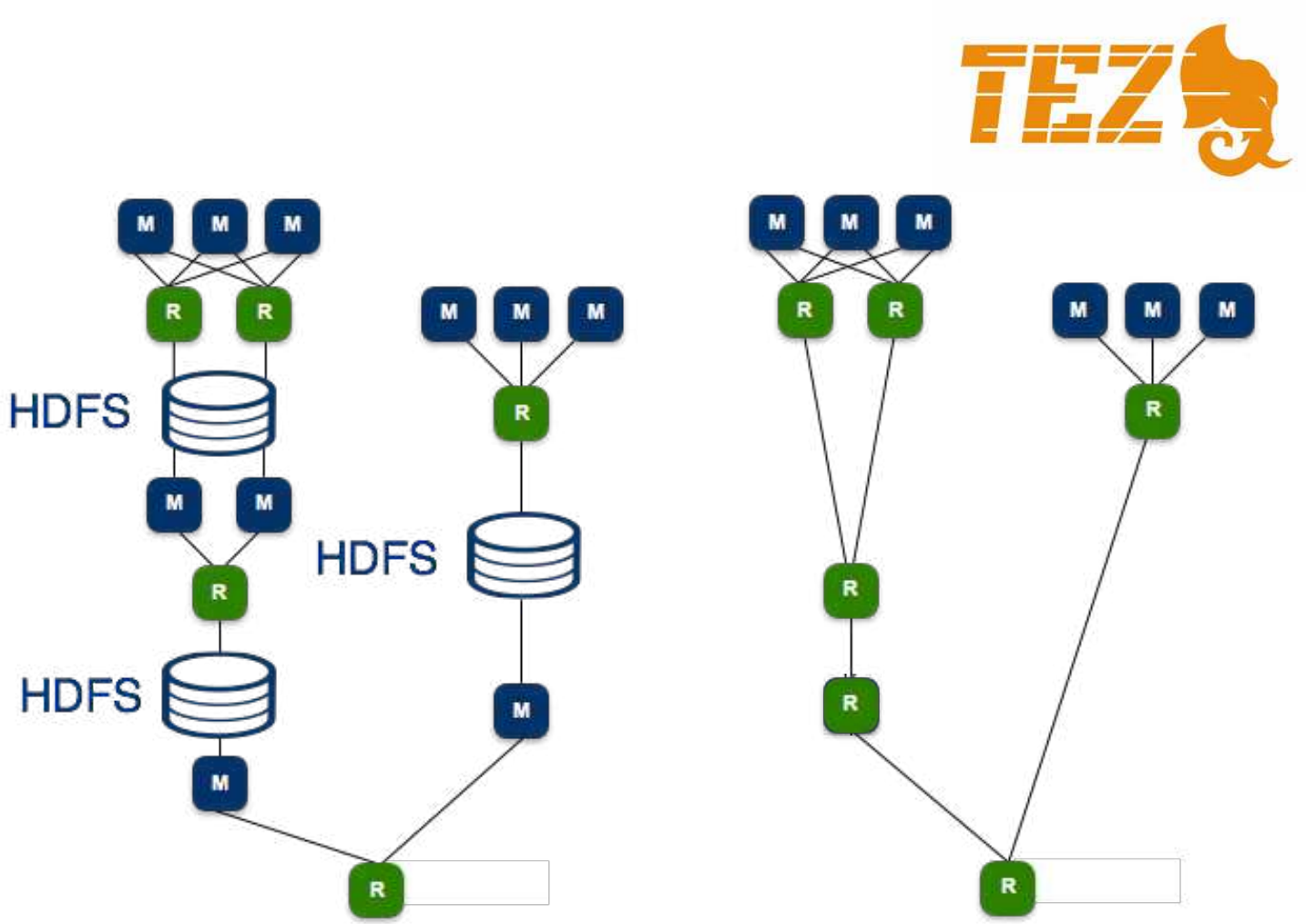
# Computational nodes architecture

- Job and task trackers have been replaced

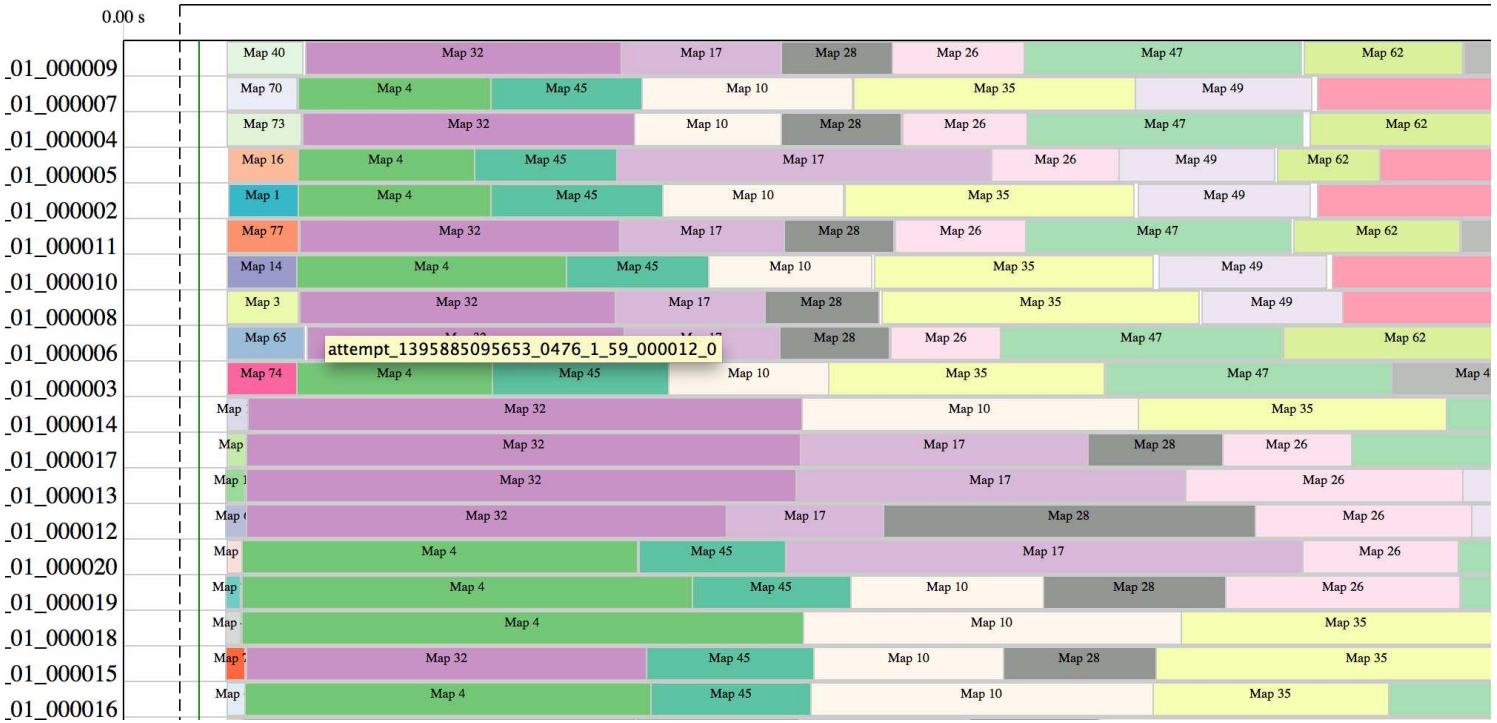
- Resource Manager:
  - Global resource scheduler
  - Hierarchical queues
- Application Master:
  - Per-application
  - Manages application scheduling and task execution
- Node Manager:
  - Per-machine agent
  - Manages the life-cycle of containers
  - Container resource monitoring



# Generic DAGs (Apache Tez)



# YARN Dynamic resource allocation




Bikas Saha Apache Tez: Accelerating Hadoop Data Processing
http://www.slideshare.net/hortonworks/apache-tez-accelerating-hadoop-query-processing

# Hadoop 2.x vs. 3.x

- HDFS Fault Tolerance:
  - Hadoop 2.x: handled by replication (which is wastage of space)
  - Hadoop 3.x: handled by erasure coding, **50% instead of 200%** overhead
- Containers:
  - Hadoop 3.x provides Docker support in Linux containers executors
- GPUs:
  - Hadoop 3.x enables scheduling of additional resources, such as disks and GPUs for better support of deep learning & machine learning applications
  - Deep learning models (in TensorFlow) can run on Hadoop YARN cluster where data resides

# Performance degradation

- The network is a bottleneck:
  - Copy shuffle phase limit jobs execution performance
  - Network optimizations
    - load balancing across multiple paths
    - traffic prioritization and isolation
    - data aggregation and compression
- The disk is a bottleneck:
  - Caching data in memory
- Stragglers:
  - Tasks that take much longer to complete than other tasks in the job
  - Outliers caused by poorly performing machines that cause tasks scheduled on them to take longer
  - Work may have been unevenly divided across tasks, skews

# Pig & Hive

# Need for High-Level Languages

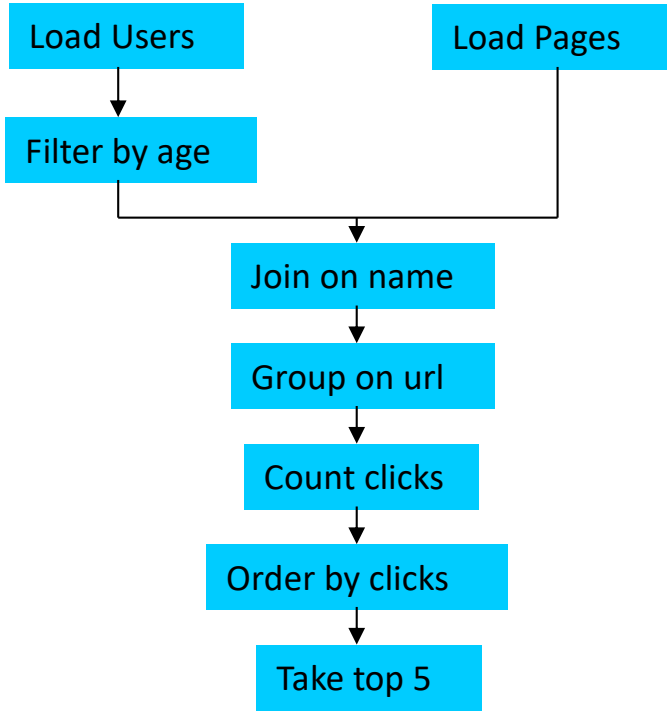

- Hadoop is great for large-data processing!
  - But writing Java programs for everything is verbose and slow
  - Not everyone wants to (or can) write Java code

- Solution: develop higher-level data processing languages
  - Pig: Pig Latin is a bit like Perl
  - Hive: HQL is like SQL

- Common idea:
  - Higher-level language “compiles down” to Hadoop jobs

# Motivation by example

Suppose you have user data in one file, website data in another, and you need to find the top 5 most visited pages by users aged 18 - 25.



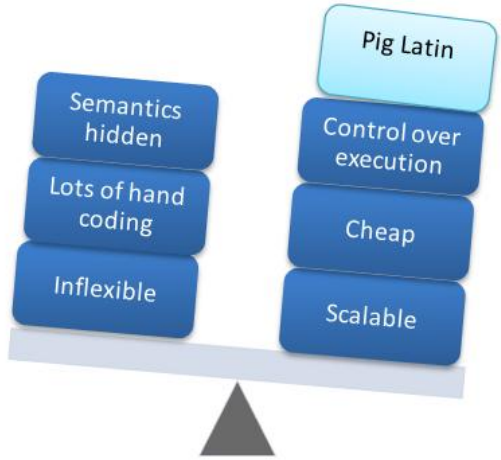
# In Map Reduce

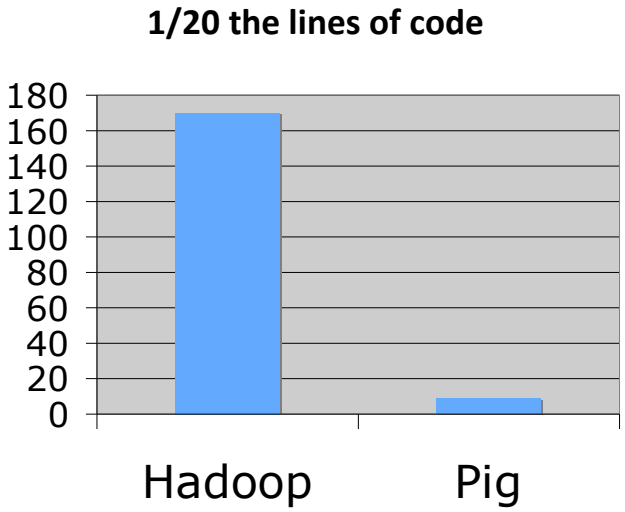
```
import java.io.IOException;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.io.Writable;
import org.apache.hadoop.io.WritableComparable;
import org.apache.hadoop.mapred.FileInputFormat;
import org.apache.hadoop.mapred.FileOutputFormat;
import org.apache.hadoop.mapred.JobConf;
import org.apache.hadoop.mapred.KeyValueTextInputFormat;
import org.apache.hadoop.mapred.Mapper;
import org.apache.hadoop.mapred.MapReduceBase;
import org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.mapred.RecordReader;
import org.apache.hadoop.mapred.Reducer;
import org.apache.hadoop.mapred.Reporter;
import org.apache.hadoop.mapred.SequenceFileInputFormat;
import org.apache.hadoop.mapred.SequenceFileOutputFormat;
import org.apache.hadoop.mapred.TextInputFormat;
import org.apache.hadoop.mapred.jobcontrol.Job;
import org.apache.hadoop.mapred.jobcontrol.JobControl;
import org.apache.hadoop.mapred.lib.IdentityMapper;

public class MRExample {
    public static class LoadPages extends MapReduceBase
        implements Mapper<LongWritable, Text, Text, Text> {

        public void map(LongWritable k, Text val,
                OutputCollector<Text, Text> oc,
                Reporter reporter) throws IOException {
            // Pull the key out
            String line = val.toString();
            int firstComma = line.indexOf(',');
            String key = line.substring(0, firstComma);
            String value = line.substring(firstComma + 1);
            Text outKey = new Text(key);
            // Prepend an index to the value so we know which file
            // it came from.
            Text outVal = new Text("1" + value);
            oc.collect(outKey, outVal);
        }
    }
    public static class LoadAndFilterUsers extends MapReduceBase
        implements Mapper<LongWritable, Text, Text, Text> {

        public void map(LongWritable k, Text val,
                OutputCollector<Text, Text> oc,
                Reporter reporter) throws IOException {
            // Pull the key out
            String line = val.toString();
            int firstComma = line.indexOf(',');
            String value = line.substring(firstComma + 1);
            int age = Integer.parseInt(value);
            if (age < 18 || age > 25) return;
            String key = line.substring(0, firstComma);
            Text outKey = new Text(key);
            // Prepend an index to the value so we know which file
            // it came from.
            Text outVal = new Text("2" + value);
            oc.collect(outKey, outVal);
        }
    }
    public static class Join extends MapReduceBase
        implements Reducer<Text, Text, Text, Text> {

        public void reduce(Text key,
                Iterator<Text> iter,
                OutputCollector<Text, Text> oc,
                Reporter reporter) throws IOException {
            // For each value, figure out which file it's from and
store it
            // accordingly.
            List<String> first = new ArrayList<String>();
            List<String> second = new ArrayList<String>();

            while (iter.hasNext()) {
                Text t = iter.next();
                String value = t.toString();
                if (value.charAt(0) == '1')
first.add(value.substring(1));
                else second.add(value.substring(1));
                reporter.setStatus("OK");
            }

            // Do the cross product and collect the values
            for (String s1 : first) {
                for (String s2 : second) {
                    String outval = key + "," + s1 + "," + s2;
                    oc.collect(null, new Text(outval));
                    reporter.setStatus("OK");
                }
            }
        }
    }
    public static class LoadJoined extends MapReduceBase
        implements Mapper<Text, Text, Text, LongWritable> {

        public void map(
                Text k,
                Text val,
                OutputCollector<Text, LongWritable> oc,
                Reporter reporter) throws IOException {
            // Find the url
            String line = val.toString();
            int firstComma = line.indexOf(',');
            int secondComma = line.indexOf(',', firstComma);
            String key = line.substring(firstComma, secondComma);
            // drop the rest of the record, I don't need it anymore,
            // just pass a 1 for the combiner/reducer to sum instead.
            Text outKey = new Text(key);
            oc.collect(outKey, new LongWritable(1L));
        }
    }
    public static class ReduceUrls extends MapReduceBase
        implements Reducer<Text, LongWritable, WritableComparable,
Writable> {

        public void reduce(
                Text key,
                Iterator<LongWritable> iter,
                OutputCollector<WritableComparable, Writable> oc,
                Reporter reporter) throws IOException {
            // Add up all the values we see

            long sum = 0;
            while (iter.hasNext()) {
                sum += iter.next().get();
                reporter.setStatus("OK");
            }

            oc.collect(key, new LongWritable(sum));
        }
    }
    public static class LoadClicks extends MapReduceBase
        implements Mapper<WritableComparable, Writable, LongWritable,
Text> {

        public void map(
                WritableComparable key,
                Writable val,
                OutputCollector<LongWritable, Text> oc,
                Reporter reporter) throws IOException {
            oc.collect((LongWritable)val, (Text)key);
        }
    }
    public static class LimitClicks extends MapReduceBase
        implements Reducer<LongWritable, Text, LongWritable, Text> {

        int count = 0;
        public void reduce(
            LongWritable key,
            Iterator<Text> iter,
            OutputCollector<LongWritable, Text> oc,
            Reporter reporter) throws IOException {

            // Only output the first 100 records
            while (count < 100 && iter.hasNext()) {
                oc.collect(key, iter.next());
                count++;
            }
        }
    }
    public static void main(String[] args) throws IOException {
        JobConf lp = new JobConf(MRExample.class);
        lp.setJobName("Load Pages");
        lp.setInputFormat(TextInputFormat.class);
        lp.setOutputKeyClass(Text.class);
        lp.setOutputValueClass(Text.class);
        lp.setMapperClass(LoadPages.class);
        FileInputFormat.addInputPath(lp, new
Path("/user/gates/pages"));
        FileOutputFormat.setOutputPath(lp,
            new Path("/user/gates/tmp/indexed_pages"));
        lp.setNumReduceTasks(0);
        Job loadPages = new Job(lp);

        JobConf lfu = new JobConf(MRExample.class);
        lfu.setJobName("Load and Filter Users");
        lfu.setInputFormat(TextInputFormat.class);
        lfu.setOutputKeyClass(Text.class);
        lfu.setOutputValueClass(Text.class);
        lfu.setMapperClass(LoadAndFilterUsers.class);
        FileInputFormat.addInputPath(lfu, new
Path("/user/gates/users"));
        FileOutputFormat.setOutputPath(lfu,
            new Path("/user/gates/tmp/filtered_users"));
        lfu.setNumReduceTasks(0);
        Job loadUsers = new Job(lfu);

        JobConf join = new JobConf(MRExample.class);
        join.setJobName("Join Users and Pages");
        join.setInputFormat(KeyValueTextInputFormat.class);
        join.setOutputKeyClass(Text.class);
        join.setOutputValueClass(Text.class);
        join.setMapperClass(IdentityMapper.class);
        join.setReducerClass(Join.class);
        FileInputFormat.addInputPath(join, new
Path("/user/gates/tmp/indexed_pages"));
        FileInputFormat.addInputPath(join, new
Path("/user/gates/tmp/filtered_users"));
        FileOutputFormat.setOutputPath(join, new
Path("/user/gates/tmp/joined"));
        join.setNumReduceTasks(50);
        Job joinJob = new Job(join);
        joinJob.addDependingJob(loadPages);
        joinJob.addDependingJob(loadUsers);

        JobConf group = new JobConf(MRExample.class);
        group.setJobName("Group URLs");
        group.setInputFormat(KeyValueTextInputFormat.class);
        group.setOutputKeyClass(Text.class);
        group.setOutputValueClass(LongWritable.class);
        group.setOutputFormat(SequenceFileOutputFormat.class);
        group.setMapperClass(LoadJoined.class);
        group.setCombinerClass(ReduceUrls.class);
        group.setReducerClass(ReduceUrls.class);
        FileInputFormat.addInputPath(group, new
Path("/user/gates/tmp/joined"));
        FileOutputFormat.setOutputPath(group, new
Path("/user/gates/tmp/grouped"));
        group.setNumReduceTasks(50);
        Job groupJob = new Job(group);
        groupJob.addDependingJob(joinJob);

        JobConf top100 = new JobConf(MRExample.class);
        top100.setJobName("Top 100 sites");
        top100.setInputFormat(SequenceFileInputFormat.class);
        top100.setOutputKeyClass(LongWritable.class);
        top100.setOutputValueClass(Text.class);
        top100.setOutputFormat(SequenceFileOutputFormat.class);
        top100.setMapperClass(LoadClicks.class);
        top100.setCombinerClass(LimitClicks.class);
        top100.setReducerClass(LimitClicks.class);
        FileInputFormat.addInputPath(top100, new
Path("/user/gates/tmp/grouped"));
        FileOutputFormat.setOutputPath(top100, new
Path("/user/gates/top100sitesforusers18to25"));
        top100.setNumReduceTasks(1);
        Job limit = new Job(top100);
        limit.addDependingJob(groupJob);

        JobControl jc = new JobControl("Find top  100 sites for users
18 to 25");
        jc.addJob(loadPages);
        jc.addJob(loadUsers);
        jc.addJob(joinJob);
        jc.addJob(groupJob);
        jc.addJob(limit);
        jc.run();
    }
}
```

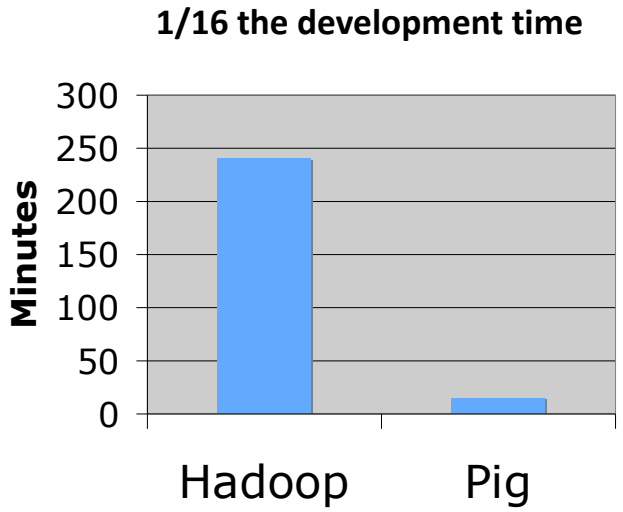
# In Pig Latin

```
Users = load 'users' as (name, age);
Fltrd = filter Users by
        age >= 18 and age <= 25;
Pages = load 'pages' as (user, url);
Jnd = join Fltrd by name, Pages by user;
Grpd = group Jnd by url;
Smmd = foreach Grpd generate group,
       COUNT(Jnd) as clicks;
Srtd = order Smmd by clicks desc;
Top5 = limit Srtd 5;
store Top5 into 'top5sites';
```

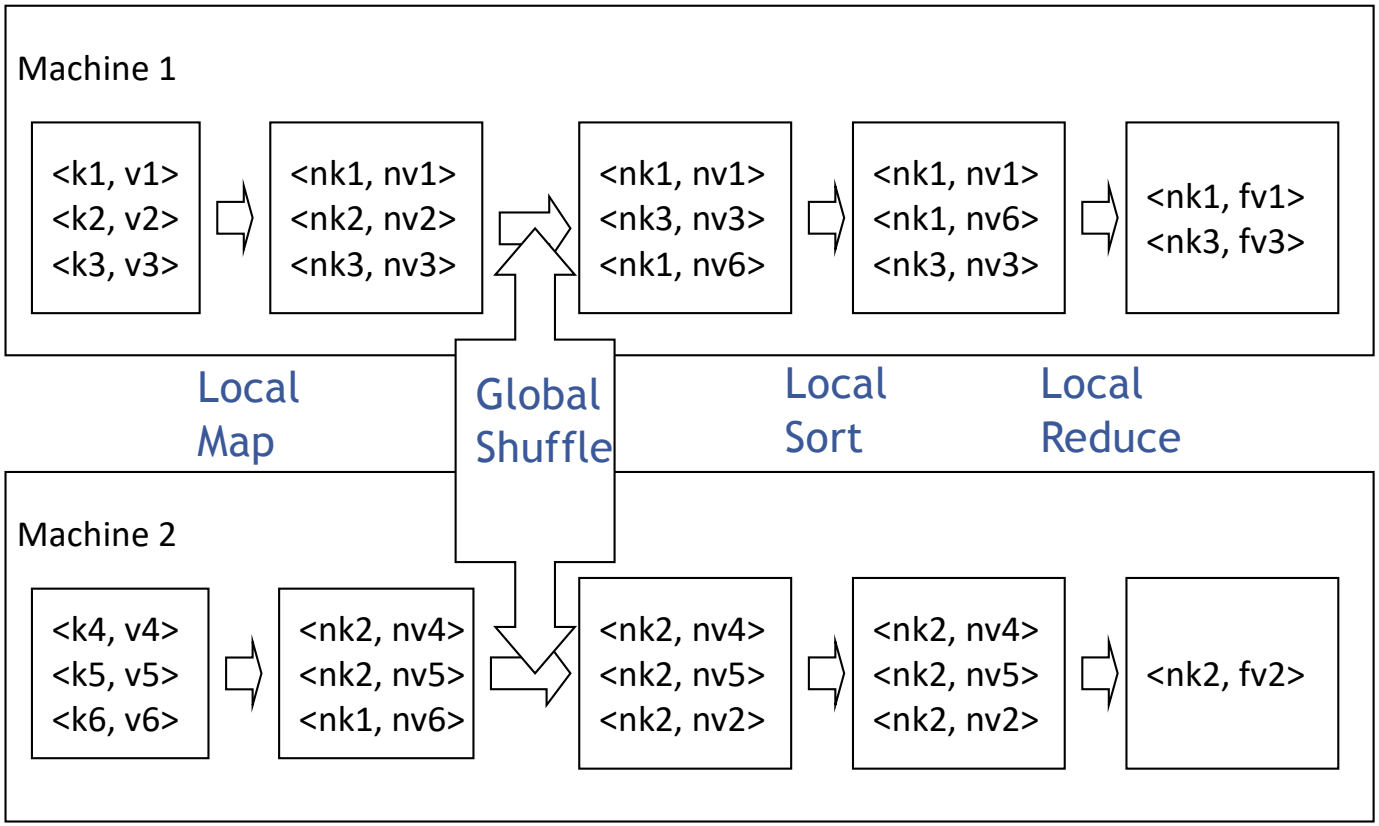
# Java vs. Pig Latin



# (Simplified) Map Reduce



# Hive QL – Join

page_view

| pageid | **userid** | time |
|---|---|---|
| 1 | **111** | 9:08:01 |
| 2 | **111** | 9:08:13 |
| 1 | **222** | 9:08:14 |

X

user

| **userid** | age | gender |
|---|---|---|
| **111** | 25 | female |
| **222** | 32 | male |

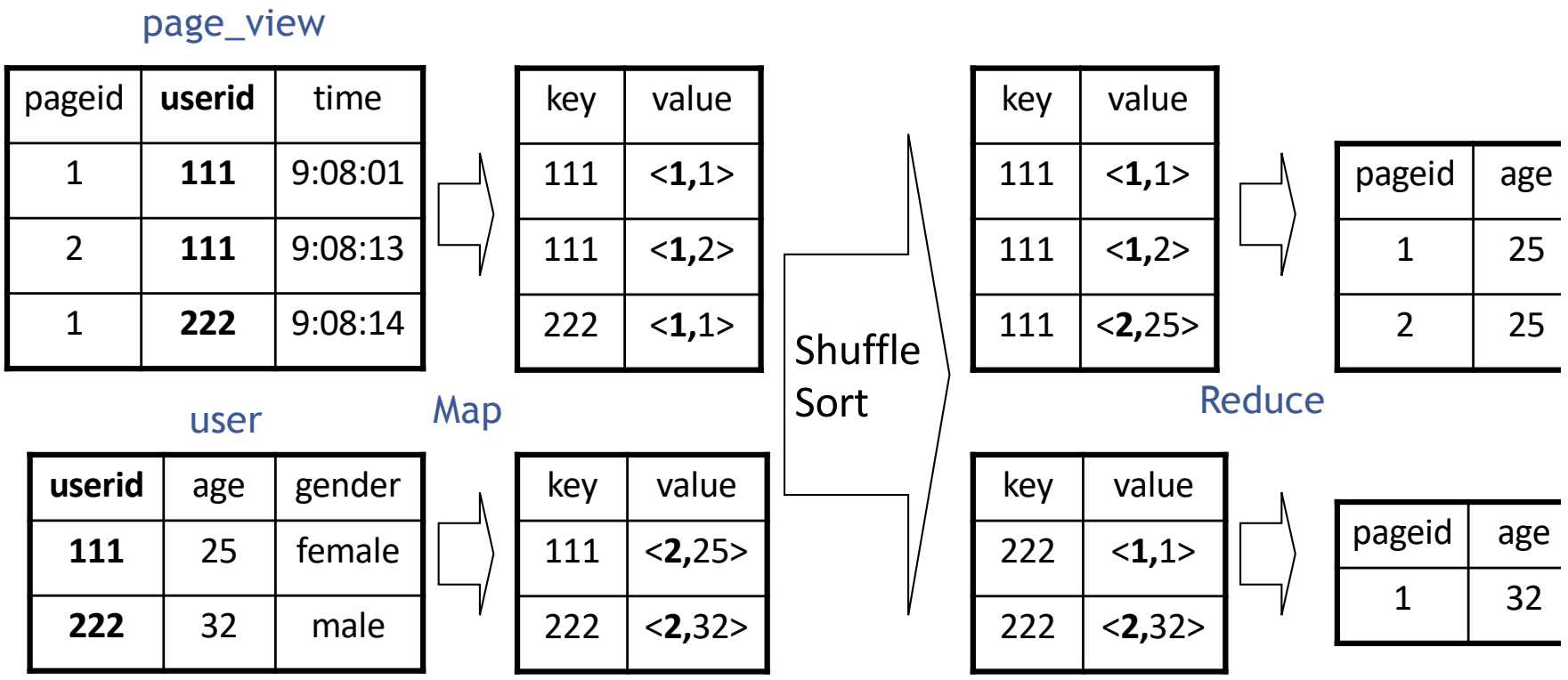
=

pv_users

| pageid | age |
|---|---|
| 1 | 25 |
| 2 | 25 |
| 1 | 32 |

```
INSERT INTO TABLE pv_users
SELECT pv.pageid, u.age
FROM page_view pv JOIN user u
        ON (pv.userid = u.userid);
```

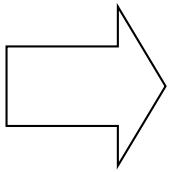
# Hive QL – Join in Map Reduce



# Hive QL – Group By

pv_users

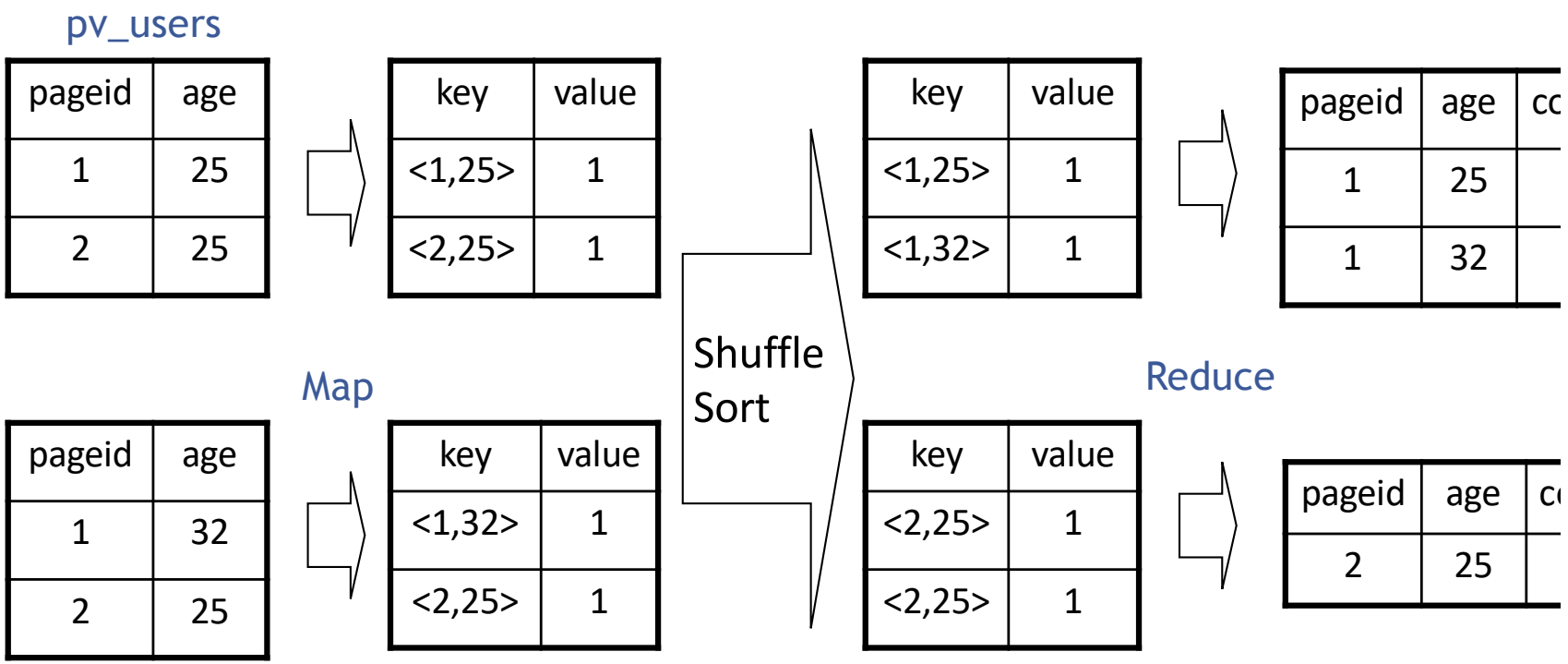
| pageid | age |
|---|---|
| 1 | 25 |
| 2 | 25 |
| 1 | 32 |
| 2 | 25 |



| pageid | age | count |
|---|---|---|
| 1 | 25 | 1 |
| 2 | 25 | 2 |
| 1 | 32 | 1 |

```
SELECT pageid, age, count(1)
FROM pv_users
GROUP BY pageid, age;
```

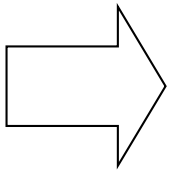
# Hive QL - Group By in Map Reduce



# Hive QL – Group By with Distinct

page_view

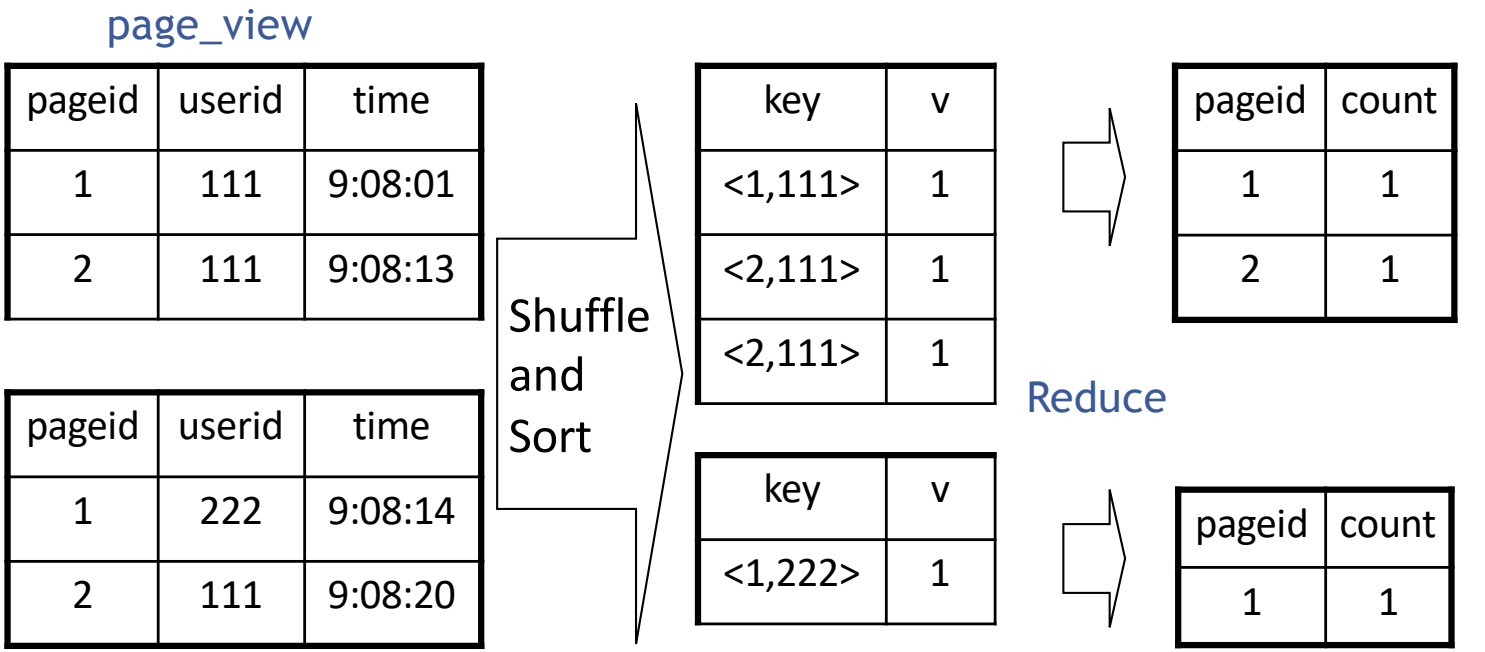
| pageid | userid | time |
|---|---|---|
| 1 | 111 | 9:08:01 |
| 2 | 111 | 9:08:13 |
| 1 | 222 | 9:08:14 |
| 2 | 111 | 9:08:20 |

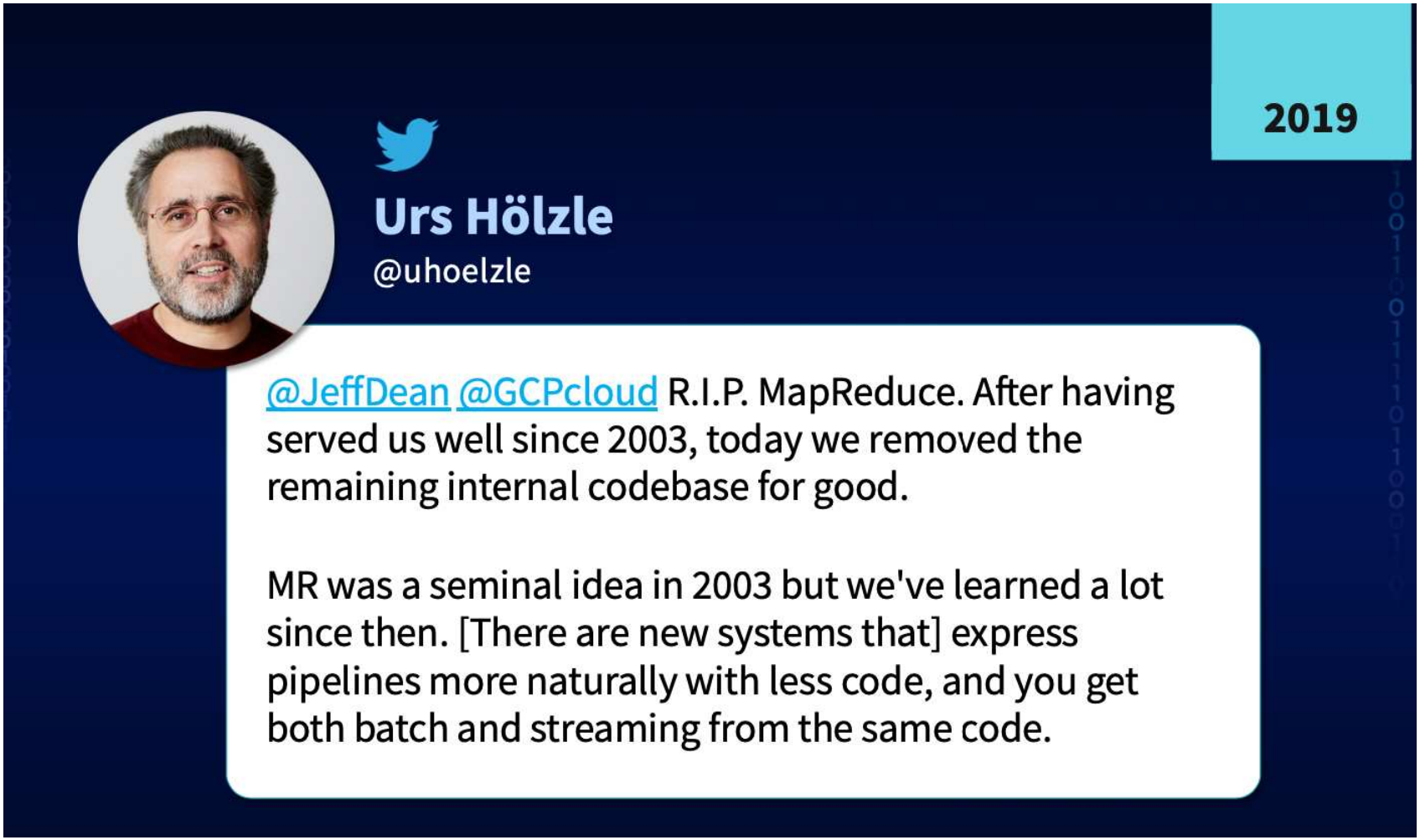


| pageid | count_distinct_userid |
|---|---|
| 1 | 2 |
| 2 | 1 |

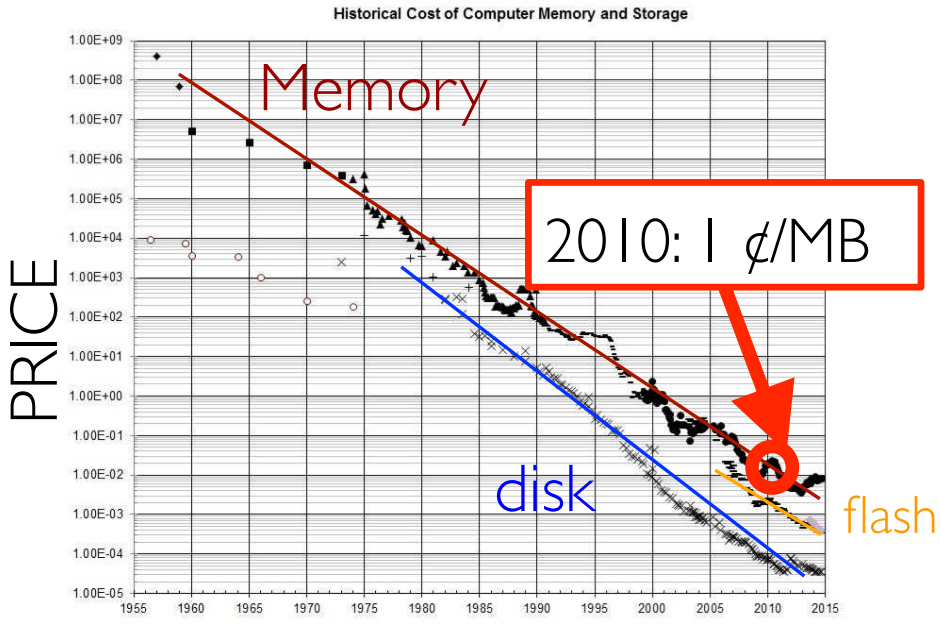
```
SELECT pageid, COUNT(DISTINCT userid)
FROM page_view GROUP BY pageid
```

# Hive QL – Group By with Distinct in Map Reduce




2019
Urs Hölzle
@uhoelzle
@JeffDean @GCPcloud R.I.P. MapReduce. After having served us well since 2003, today we removed the remaining internal codebase for good.
MR was a seminal idea in 2003 but we've learned a lot since then. [There are new systems that] express pipelines more naturally with less code, and you get both batch and streaming from the same code.

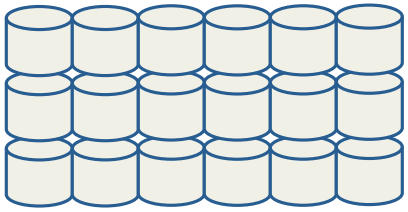
# Tech Trend: Cost of memory



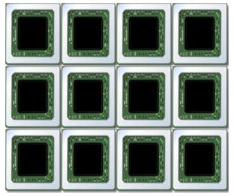

Lower cost means can put more memory in each server

http://www.jcmit.com/mem2014.htm

# Hardware for Big Data

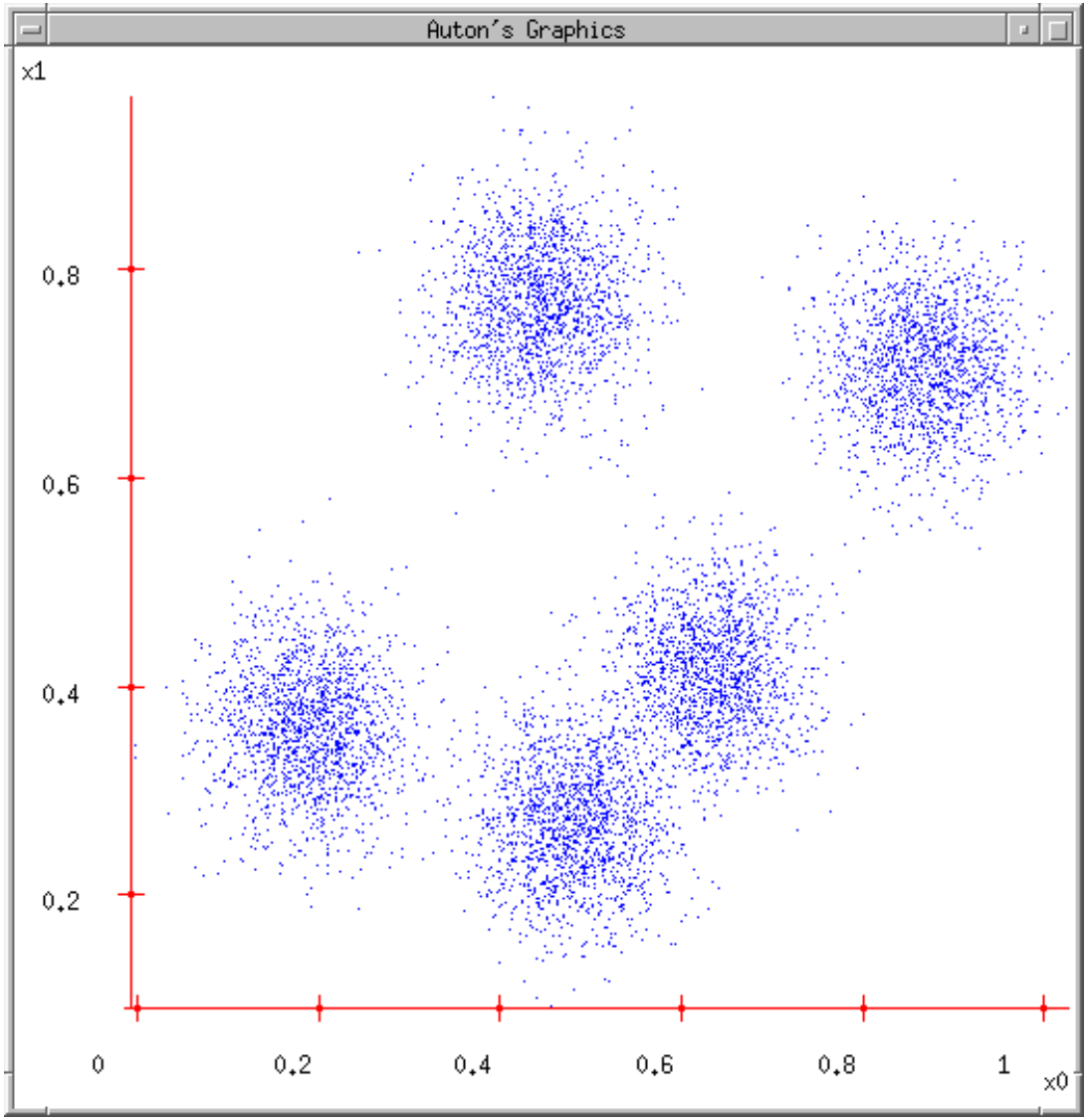


Lots of hard drives



… and CPUs

# Machine learning algorithms are iterative in nature. An example: K-Means

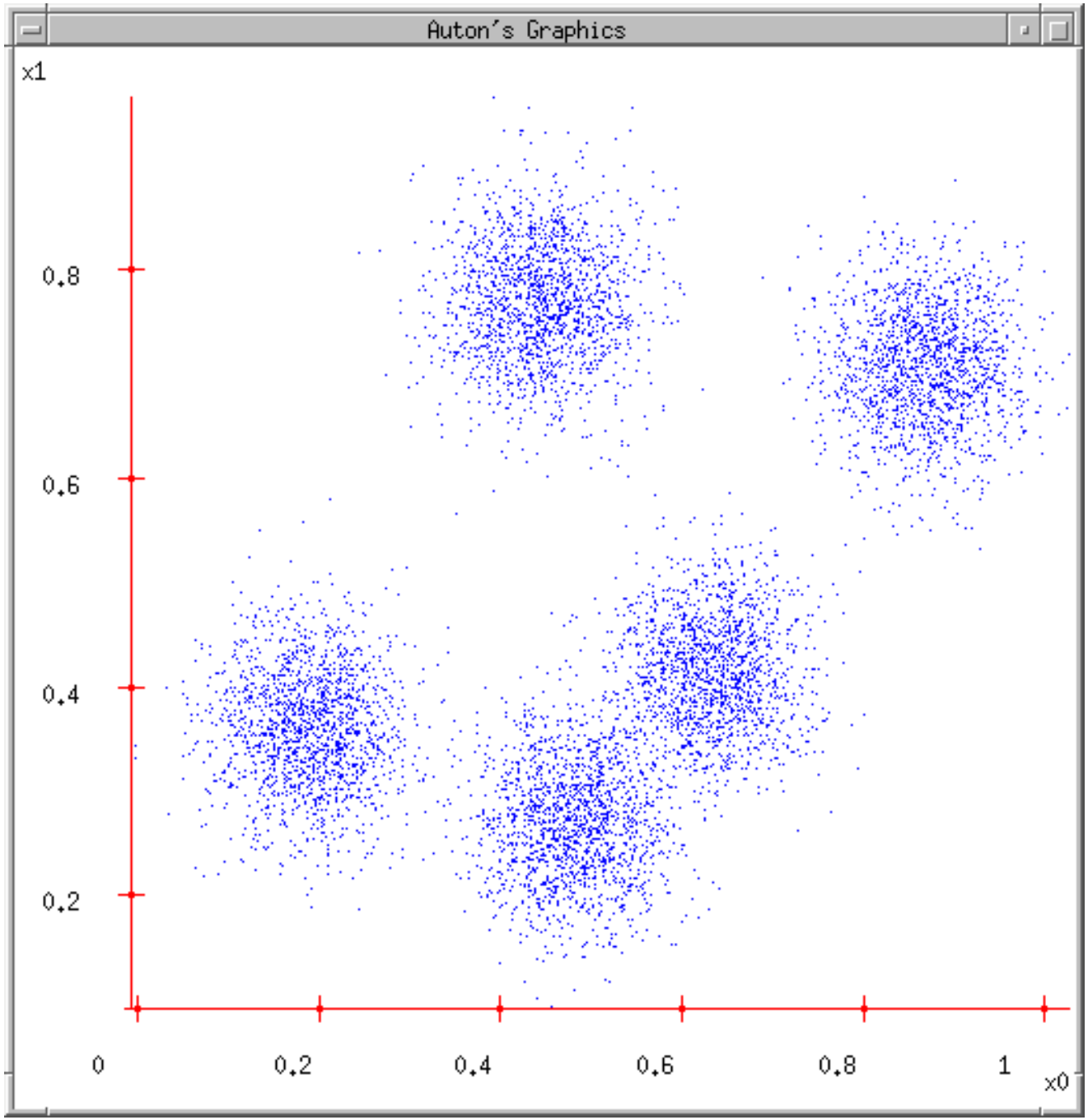


# K-means clustering

- Input: K, set of points
- Place centroids $c_1$, $c_2$, …, $c_k$ at random locations (alternatively add random lables to points)
- Repeat until convergences:
  - for each cluster j=1…K:
    - new centroid $c_j$ = mean of all points $P_i$ assigned to cluster j
  - for each point $P_i$:
    - find nearest centrod $c_j$
    - assing point $P_i$ to cluster j
- Stop when none of the cluster assignments change

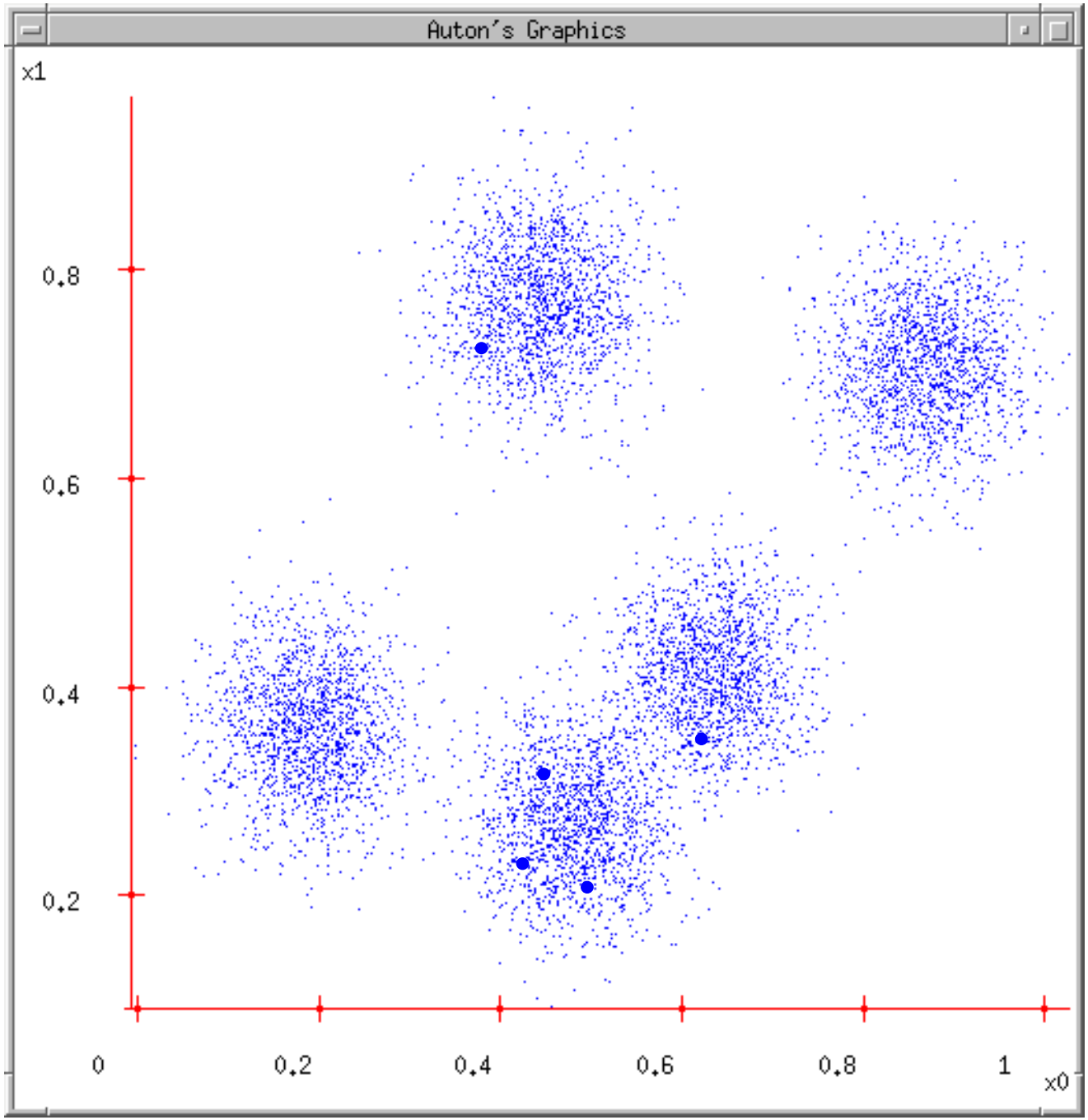
# K-means

1. Ask user how many clusters they'd like. *(e.g. k=5)*



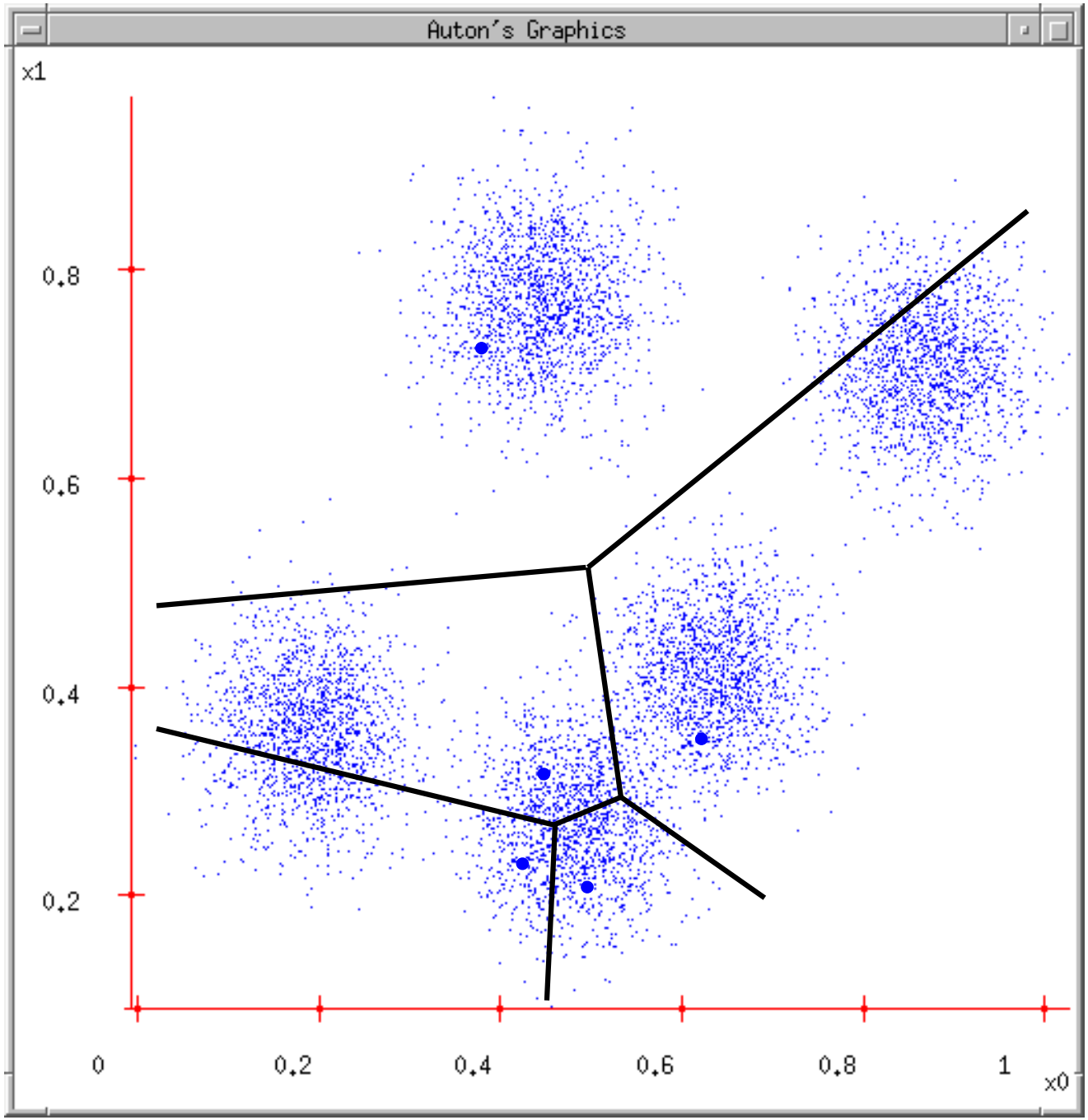
# K-means

1. Ask user how many clusters they'd like. *(e.g. k=5)*
2. Randomly guess k cluster Center locations



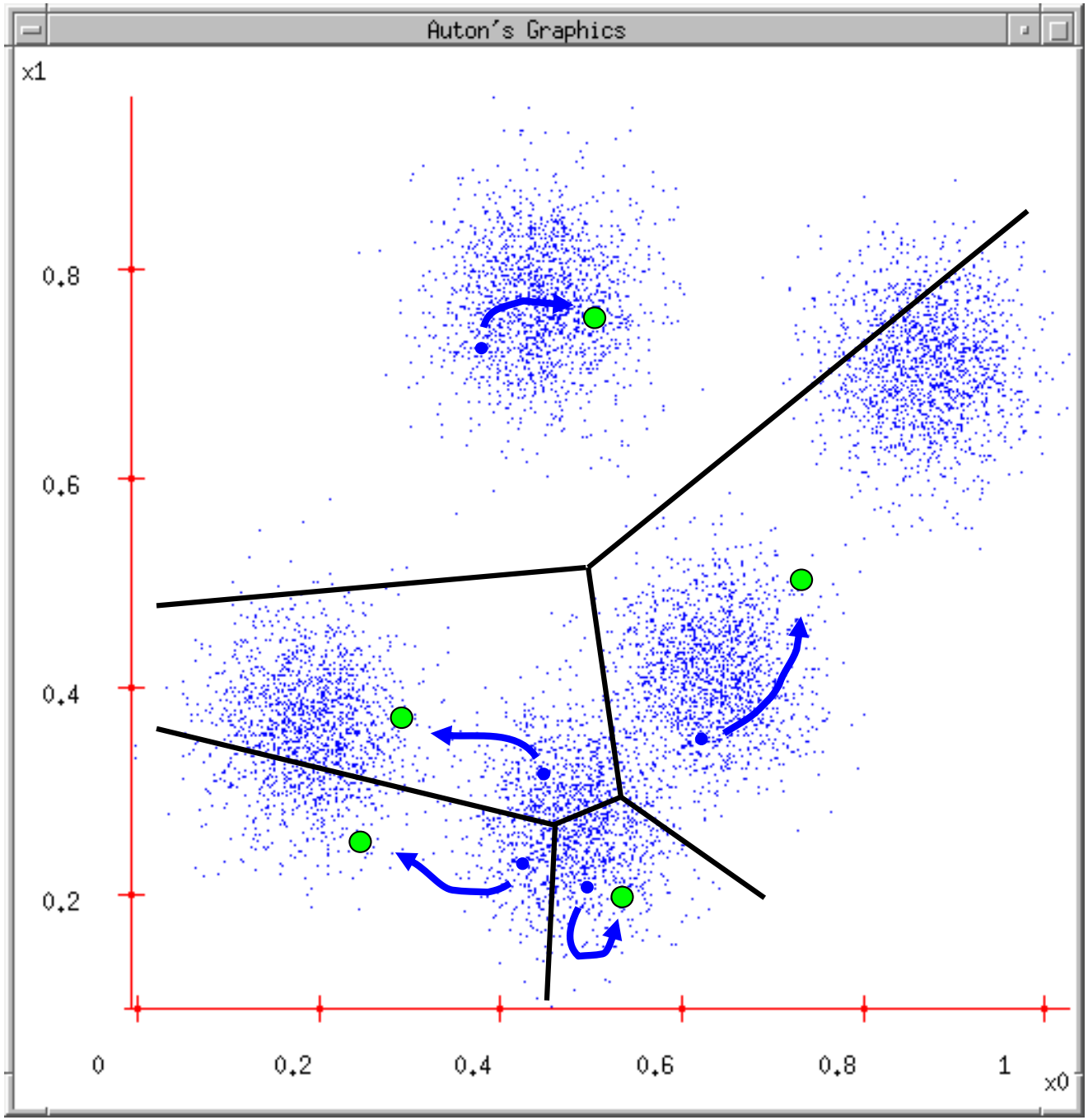
# K-means

1. Ask user how many clusters they'd like. *(e.g. k=5)*
2. Randomly guess k cluster Center locations
3. Each datapoint finds out which Center it's closest to. (Thus each Center "owns" a set of datapoints)



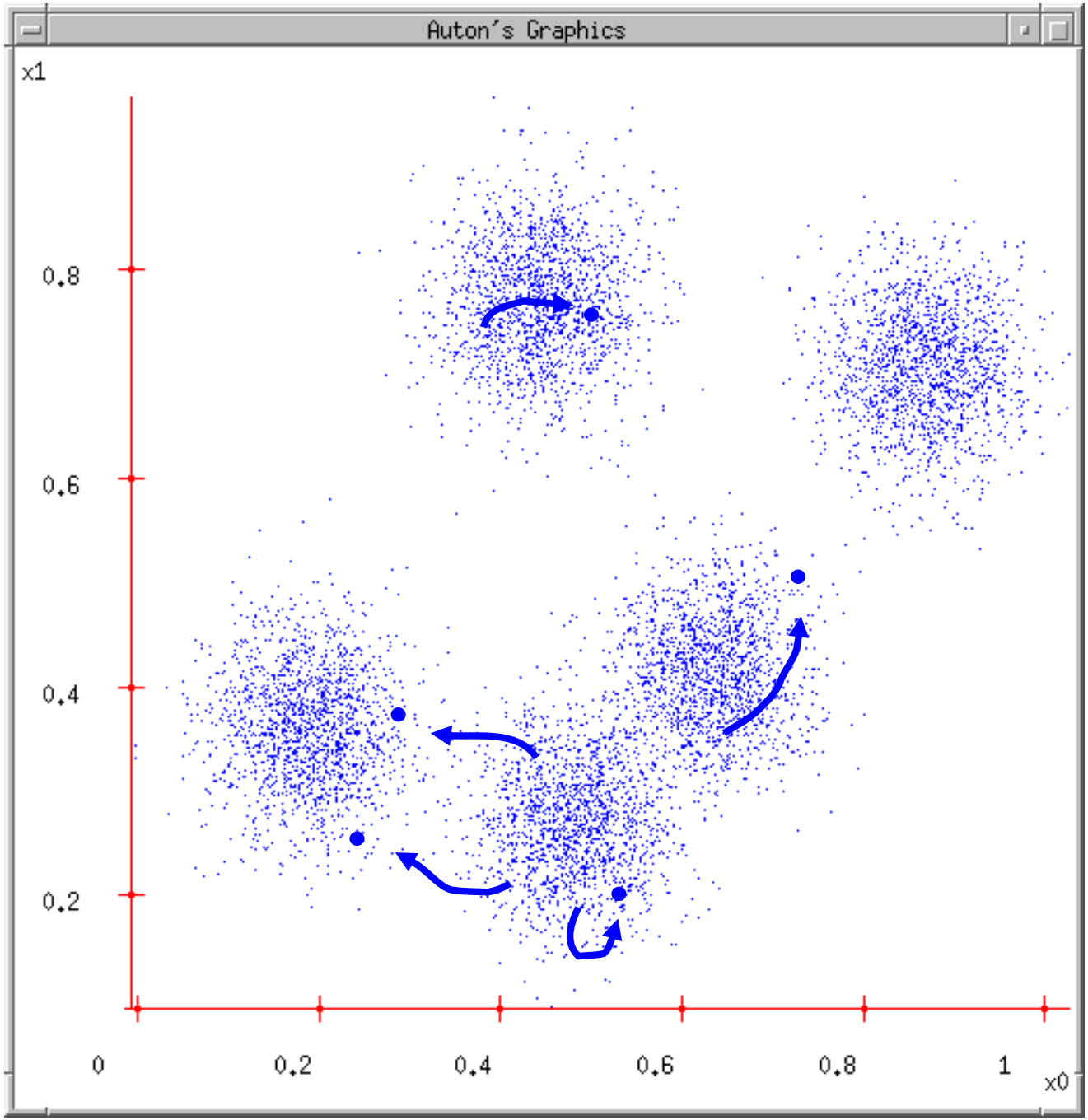
# K-means

1. Ask user how many clusters they'd like. *(e.g. k=5)*
2. Randomly guess k cluster Center locations
3. Each datapoint finds out which Center it's closest to.
4. Each Center finds the centroid of the points it owns

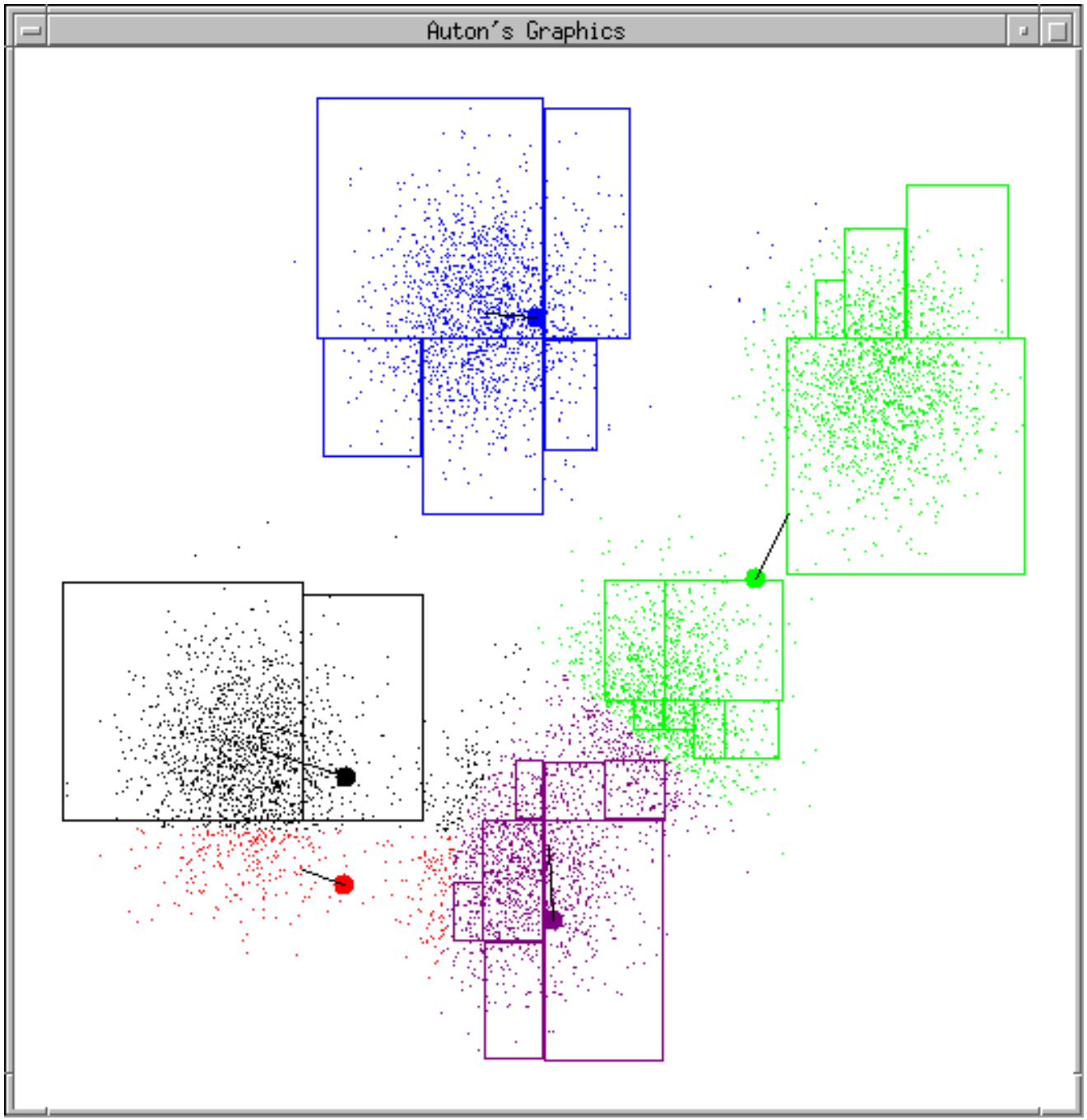


# K-means

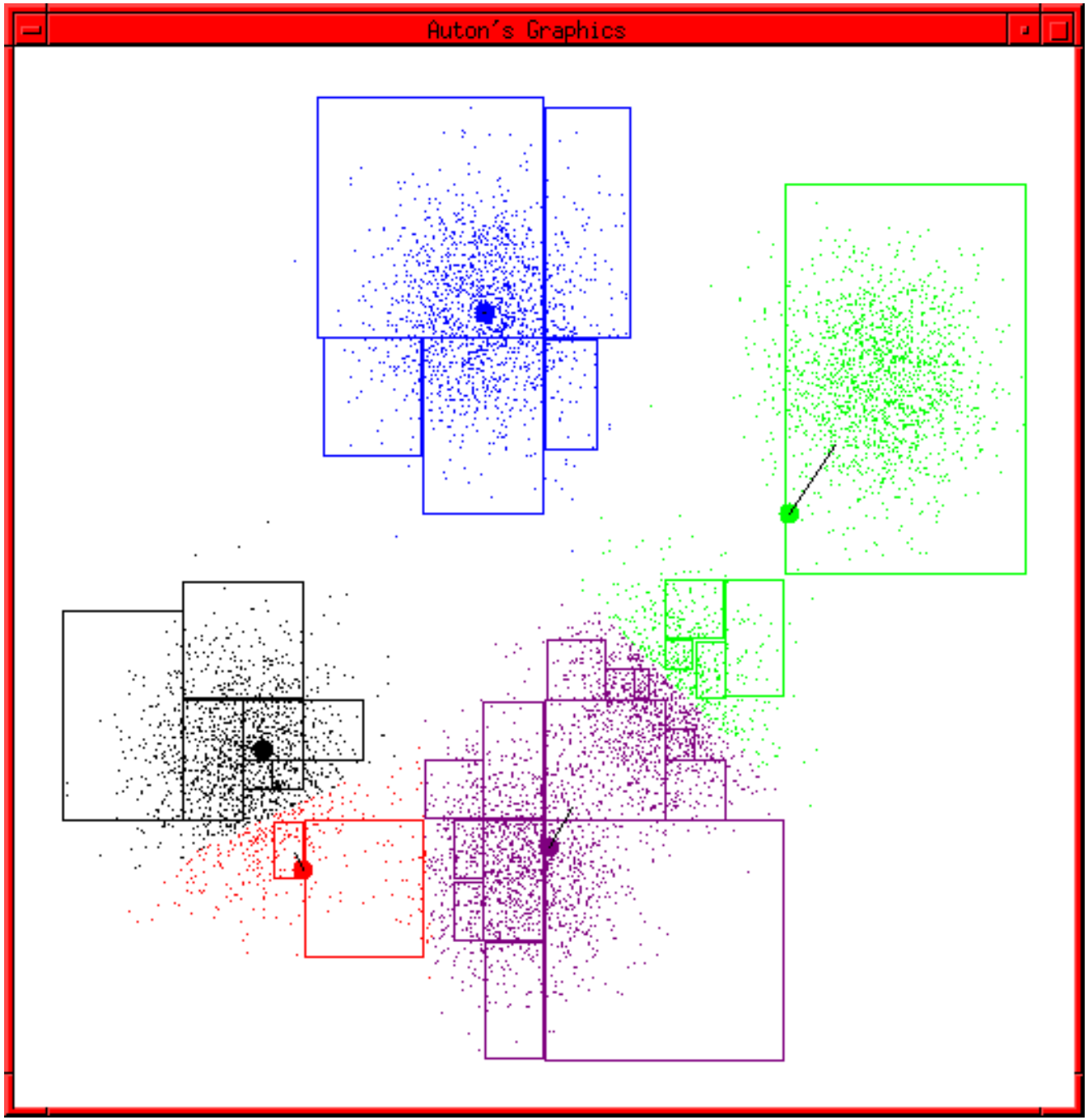
1. Ask user how many clusters they'd like. *(e.g. k=5)*
2. Randomly guess k cluster Center locations
3. Each datapoint finds out which Center it's closest to.
4. Each Center finds the centroid of the points it owns…
5. …and jumps there
6. …Repeat until terminated!



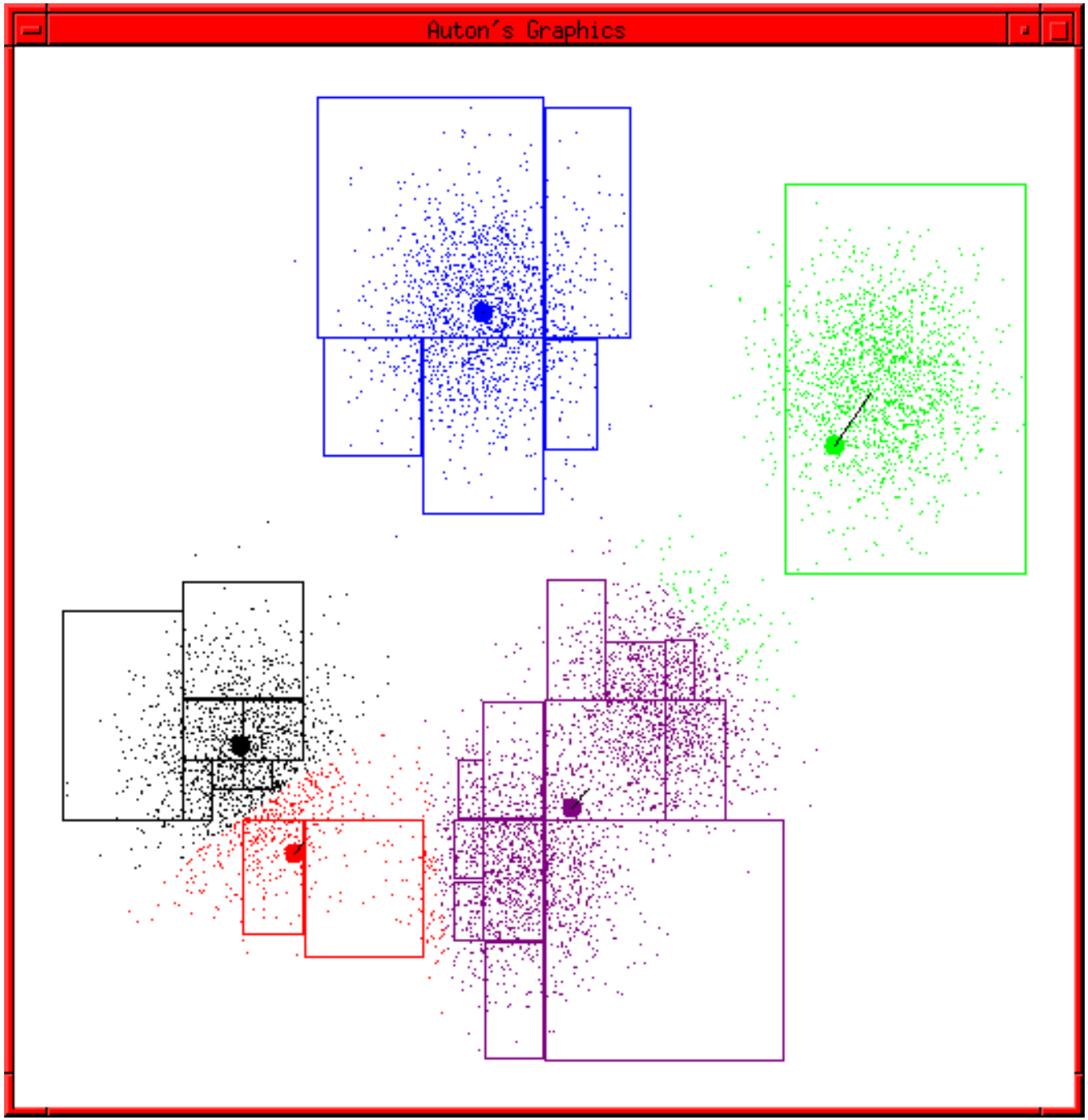
# K-means continues…



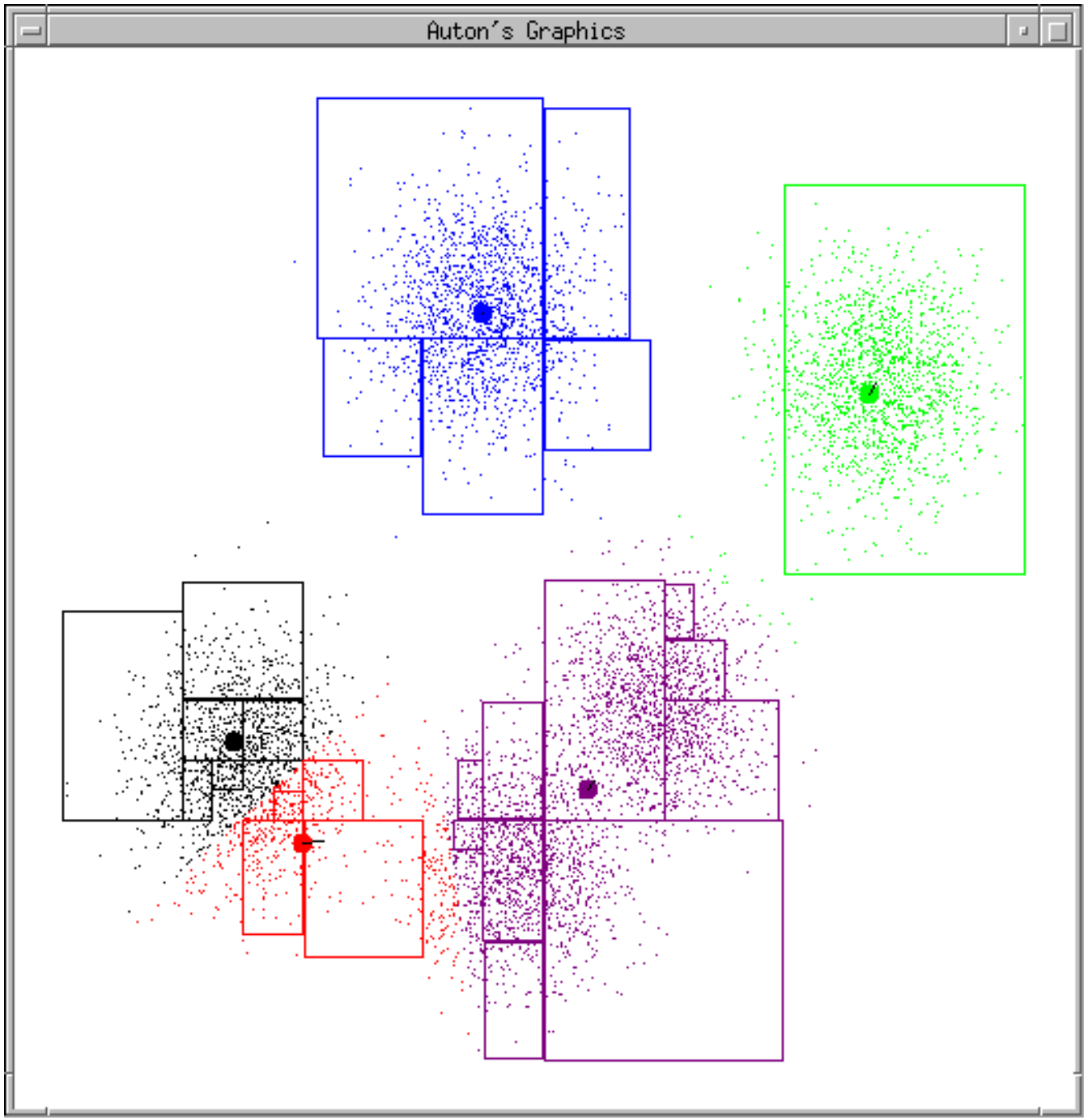
# K-means continues…



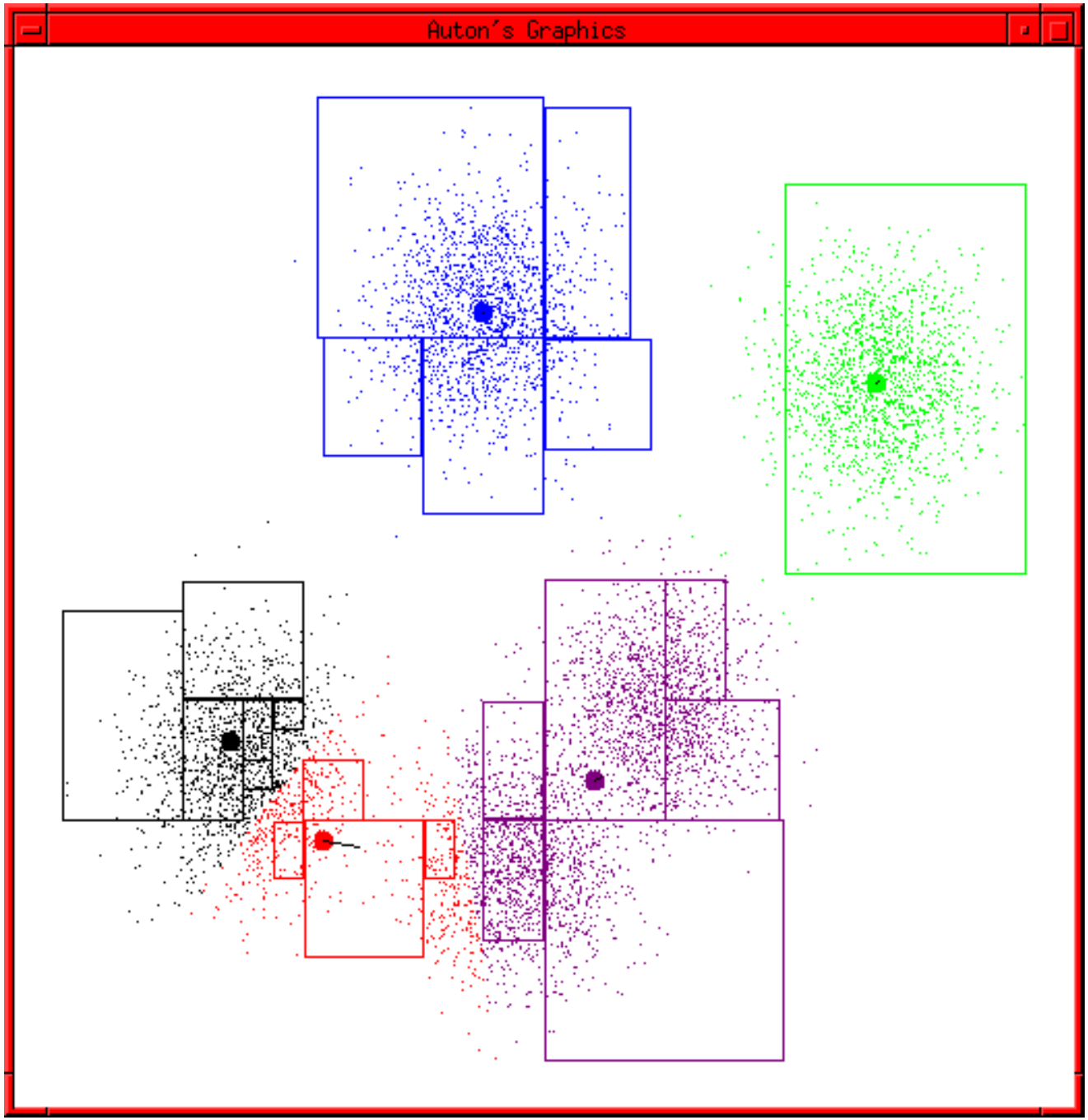
# K-means continues…



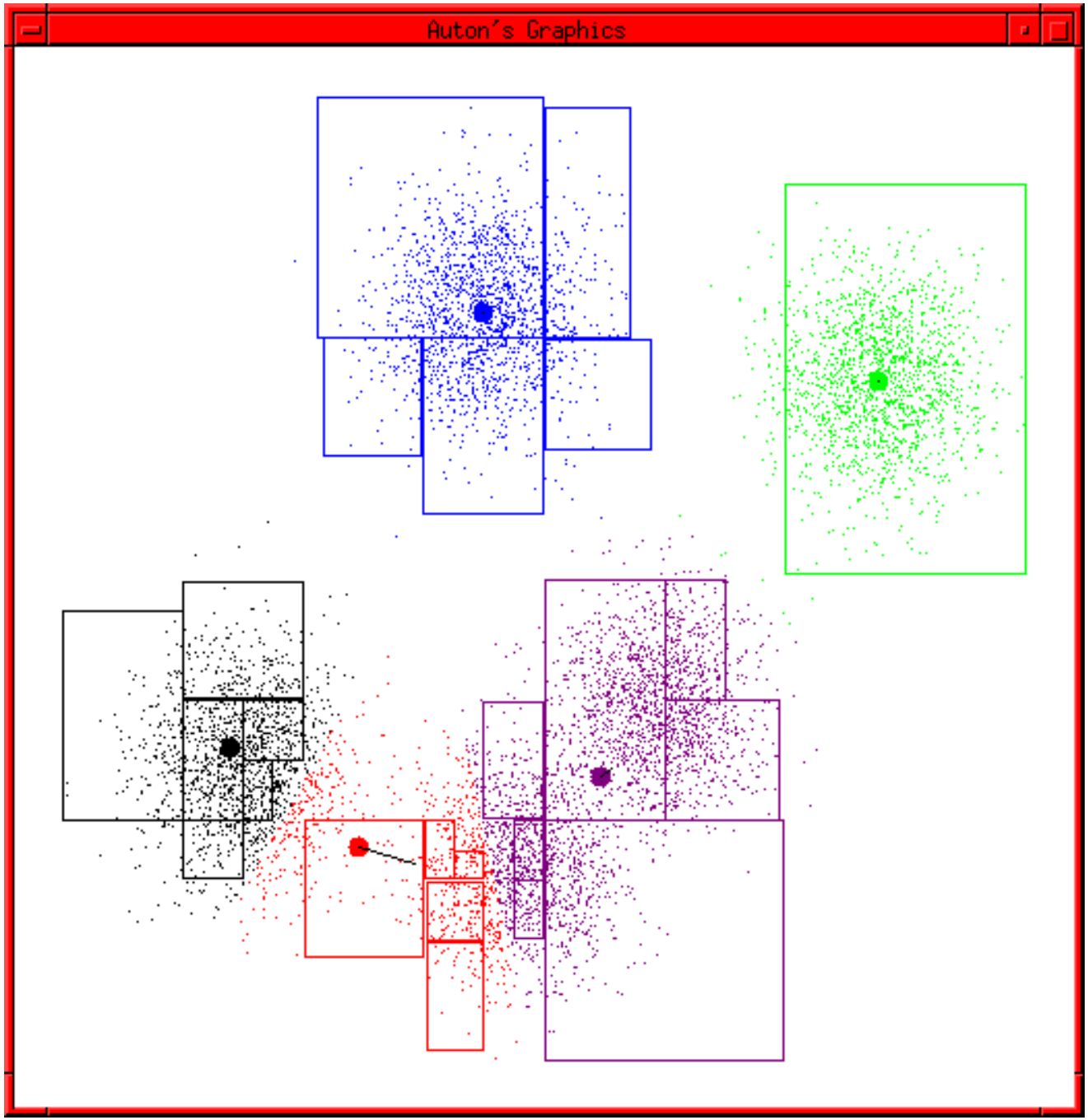
# K-means continues…



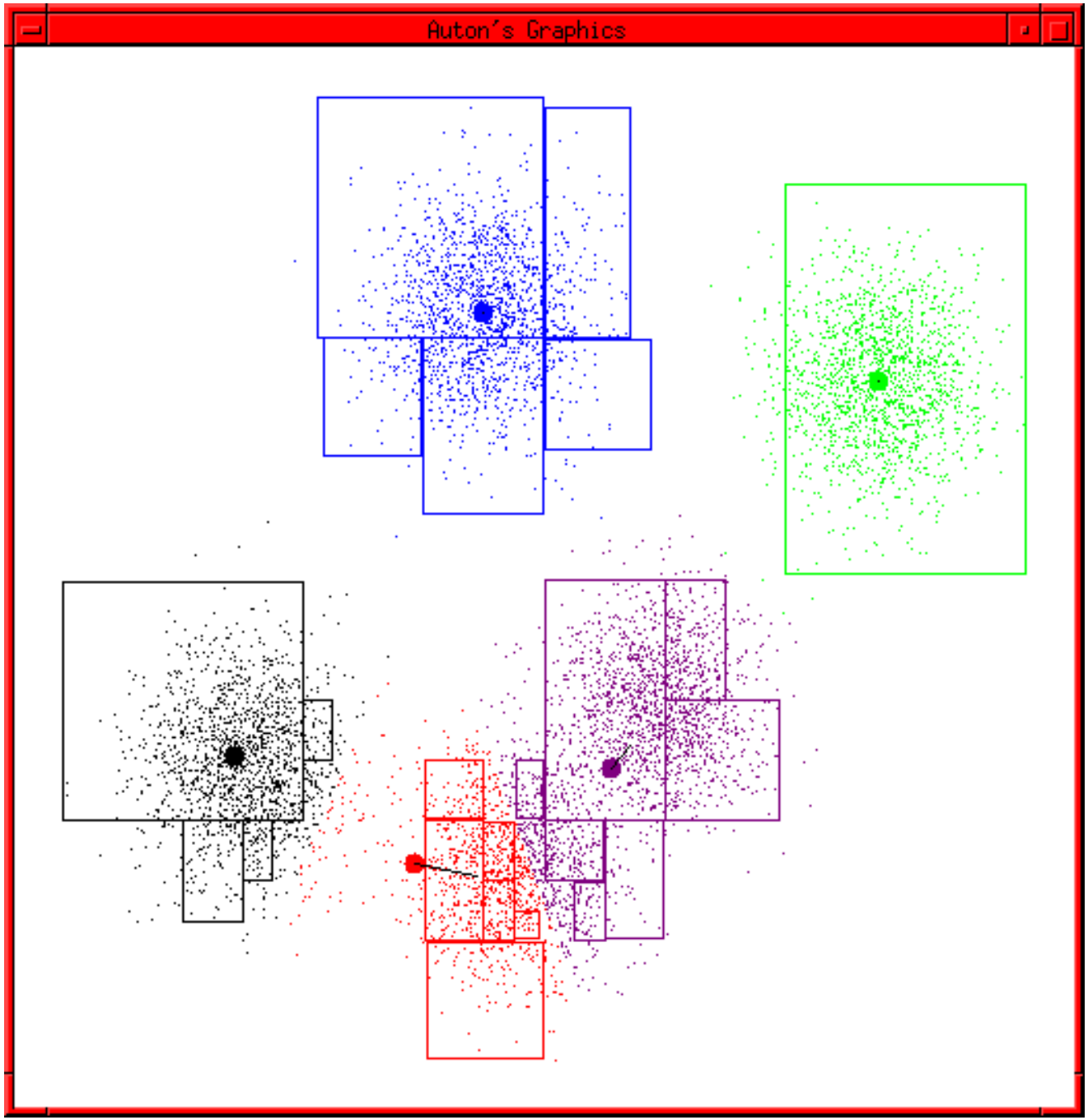
# K-means continues…



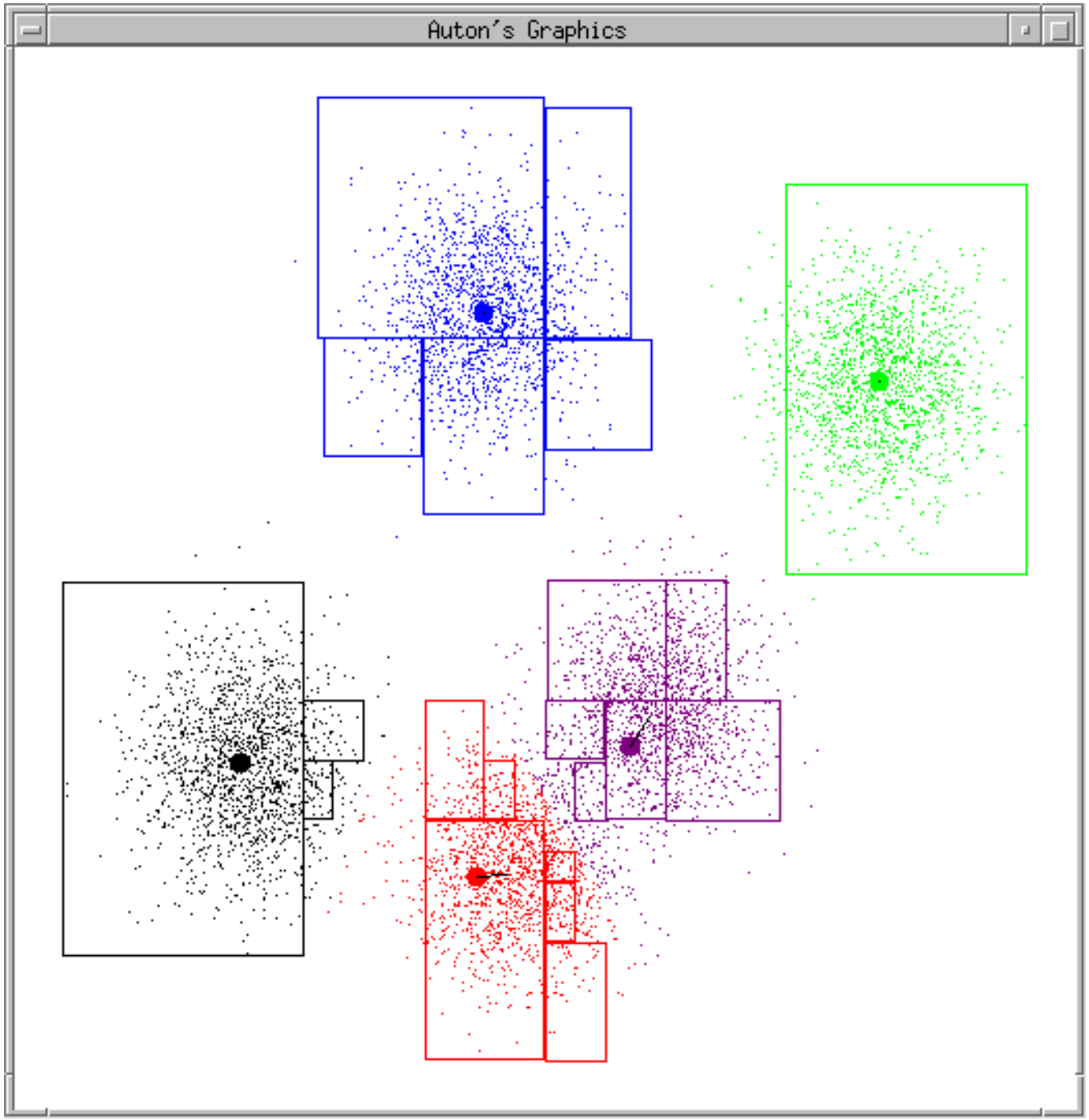
# K-means continues…



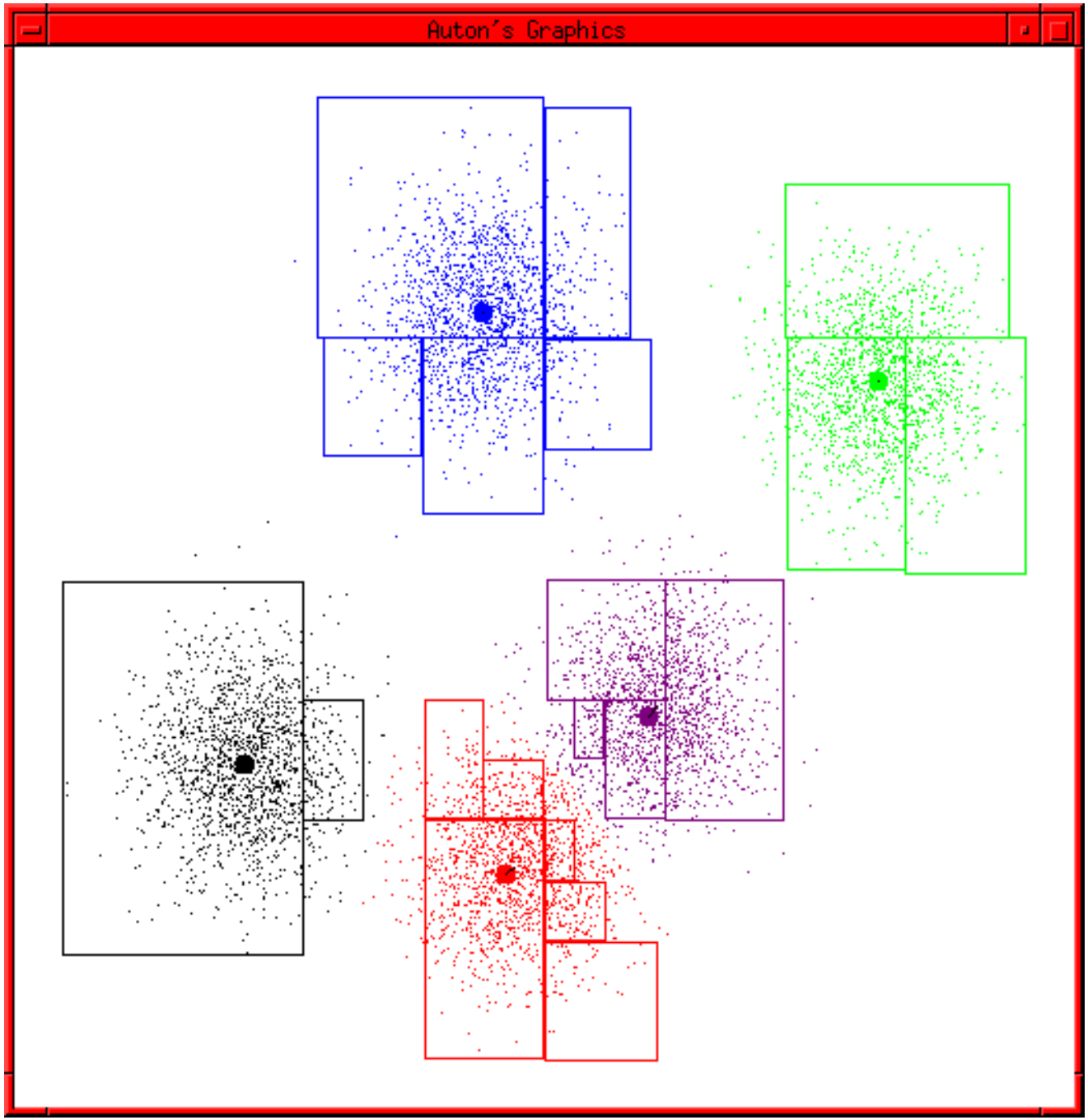
# K-means continues…



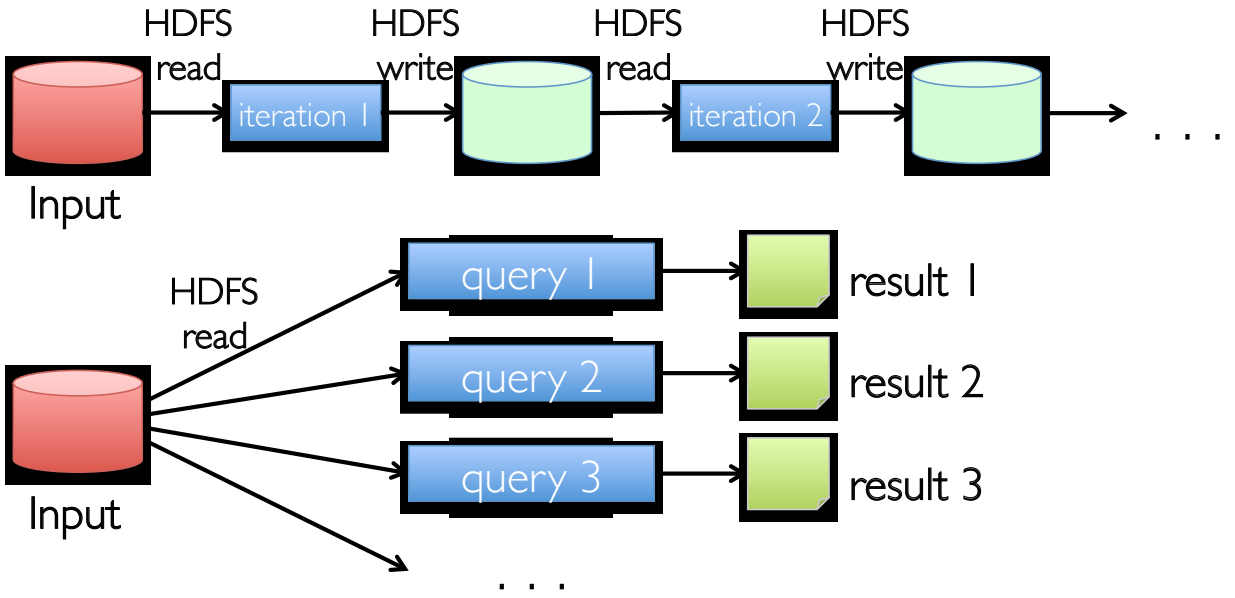
# K-means continues…



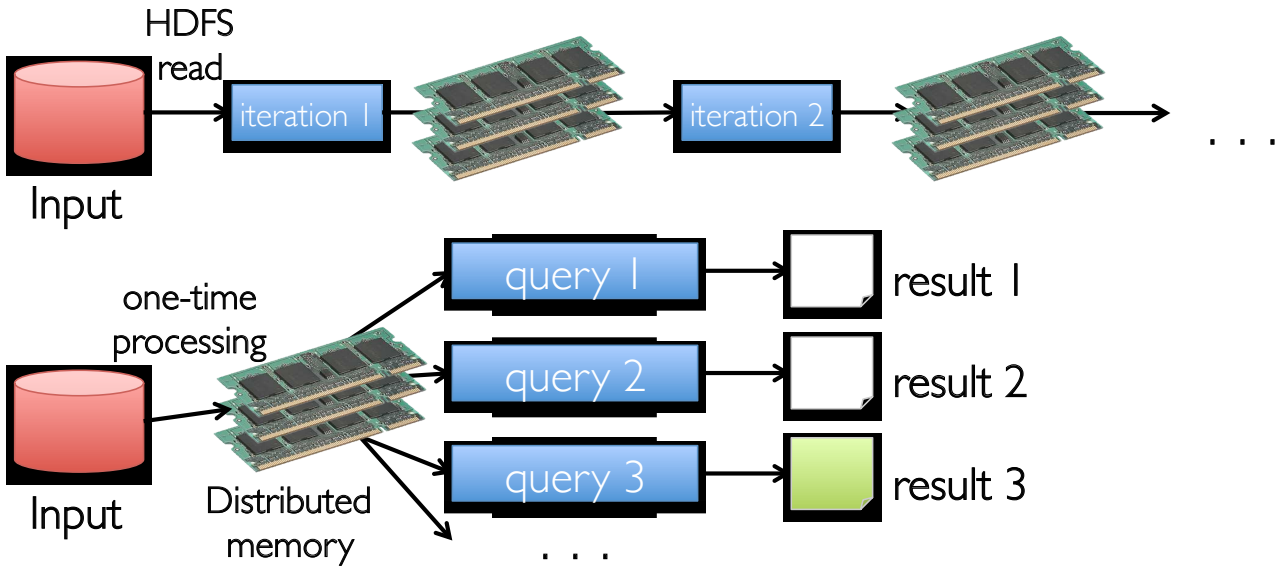
# K-means terminates



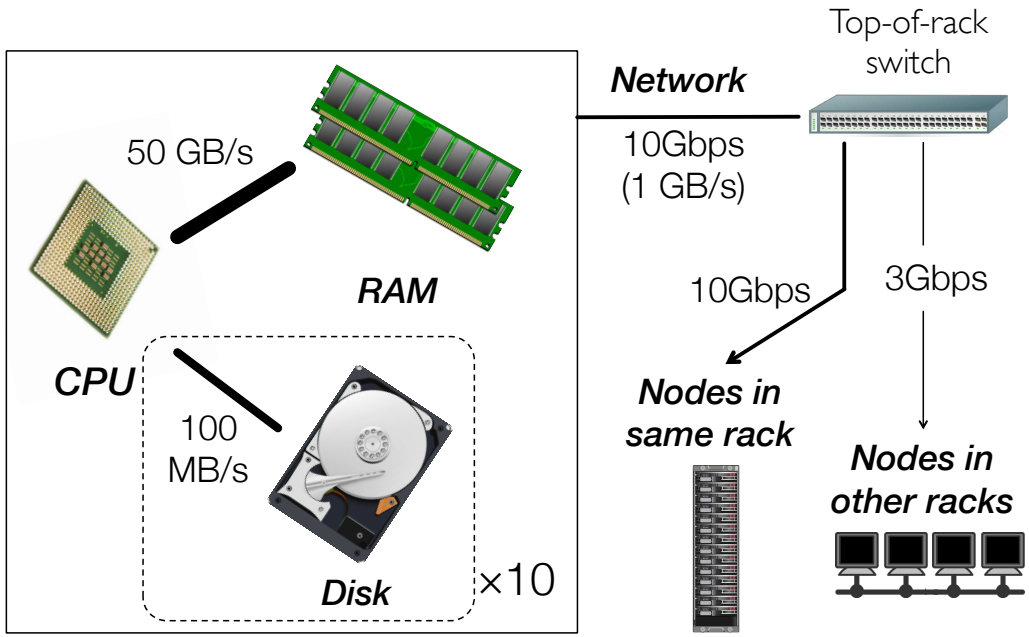
# Use memory instead of disks



# Use memory instead of disks



# Use memory instead of disks



# What is Spark?


- Fast, expressive cluster computing system compatible with Apache Hadoop
  - Works with any Hadoop-supported storage system (HDFS, S3, Avro, …)
- Improves **efficiency** through:
  - In-memory computing primitives → Up to 100× faster
  - General computation graphs
- Improves **usability** through:
  - Rich APIs in Java, Scala, **Python**, R → Often 2-10× less code
  - Interactive shell

# On-Disk Sort Record:

Time to sort 100TB

2013 Record: Hadoop

2100 machines

72 minutes

2014 Record: Spark

207 machines

23 minutes

Also sorted 1PB in 4 hours

databricks

**Source: Daytona GraySort benchmark, sortbenchmark.org**

# Word count


```
public class WordCount {
  public static class TokenizerMapper
       extends Mapper<Object, Text, Text, IntWritable>{

    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(Object key, Text value, Context context
                    ) throws IOException, InterruptedException {
      StringTokenizer itr = new StringTokenizer(value.toString());
      while (itr.hasMoreTokens()) {
        word.set(itr.nextToken());
        context.write(word, one);
      }
    }
  }

  public static class IntSumReducer
       extends Reducer<Text,IntWritable,Text,IntWritable> {
    private IntWritable result = new IntWritable();

    public void reduce(Text key, Iterable<IntWritable> values,
                       Context context
                       ) throws IOException, InterruptedException {
      int sum = 0;
      for (IntWritable val : values) {
        sum += val.get();
      }
      result.set(sum);
      context.write(key, result);
    }
  }

  public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    String[] otherArgs = new GenericOptionsParser(conf, args).getRemainingArgs();
    if (otherArgs.length < 2) {
      System.err.println("Usage: wordcount <in> [<in>...] <out>");
      System.exit(2);
    }
    Job job = new Job(conf, "word count");
    job.setJarByClass(WordCount.class);
    job.setMapperClass(TokenizerMapper.class);
    job.setCombinerClass(IntSumReducer.class);
    job.setReducerClass(IntSumReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    for (int i = 0; i < otherArgs.length - 1; ++i) {
      FileInputFormat.addInputPath(job, new Path(otherArgs[i]));
    }
    FileOutputFormat.setOutputPath(job,
      new Path(otherArgs[otherArgs.length - 1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```

WordCount in 50+ lines of Java MR

# Word count

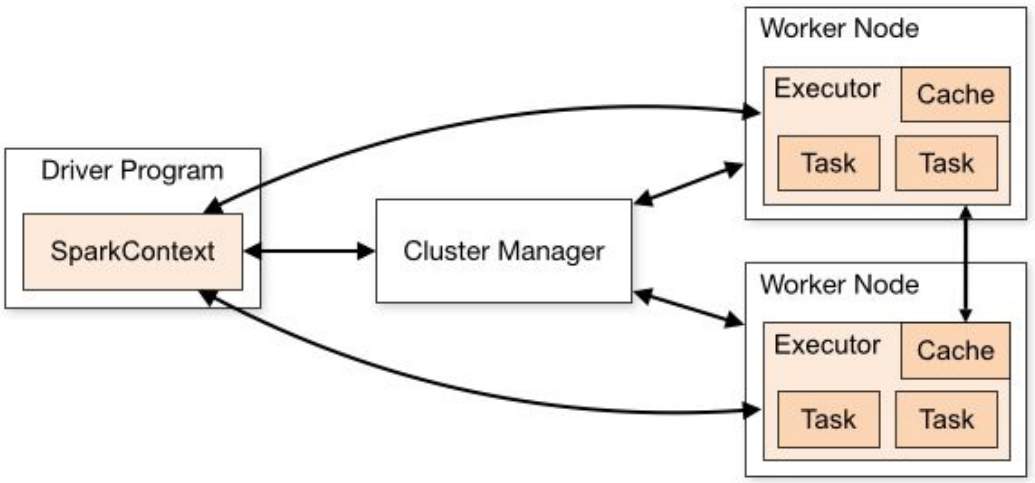

```
public class WordCount {
  public static class TokenizerMapper
       extends Mapper<Object, Text, Text, IntWritable>{

    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(Object key, Text value, Context context
                    ) throws IOException, InterruptedException {
      StringTokenizer itr = new StringTokenizer(value.toString());
      while (itr.hasMoreTokens()) {
        word.set(itr.nextToken());
        context.write(word, one);
      }
    }
  }

  public static class IntSumReducer
       extends Reducer<Text,IntWritable,Text,IntWritable> {
    private IntWritable result = new IntWritable();

    public void reduce(Text key, Iterable<IntWritable> values,
                       Context context
                       ) throws IOException, InterruptedException {
      int sum = 0;
      for (IntWritable val : values) {
        sum += val.get();
      }
      result.set(sum);
      context.write(key, result);
    }
  }

  public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    String[] otherArgs = new GenericOptionsParser(conf, args).getRemainingArgs();
    if (otherArgs.length < 2) {
      System.err.println("Usage: wordcount <in> [<in>...] <out>");
      System.exit(2);
    }
    Job job = new Job(conf, "word count");
    job.setJarByClass(WordCount.class);
    job.setMapperClass(TokenizerMapper.class);
    job.setCombinerClass(IntSumReducer.class);
    job.setReducerClass(IntSumReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    for (int i = 0; i < otherArgs.length - 1; ++i) {
      FileInputFormat.addInputPath(job, new Path(otherArgs[i]));
    }
    FileOutputFormat.setOutputPath(job,
      new Path(otherArgs[otherArgs.length - 1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```

WordCount in 50+ lines of Java MR

```
text_file = sc.textFile("hdfs://...")
counts = text_file.flatMap(lambda line: line.split(" ")) \
                  .map(lambda word: (word, 1)) \
                  .reduceByKey(lambda a, b: a + b)
counts.saveAsTextFile("hdfs://...")
```

WordCount in 3 lines of Spark

# Cluster Mode (1/2)

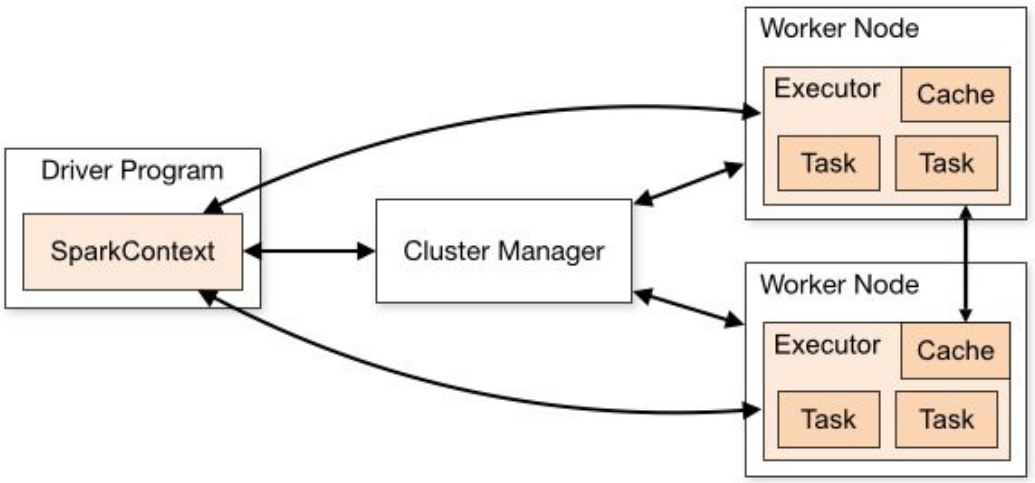



A Spark application runs coordinated by the **SparkContext** in the main program (called the *driver program*)

The SparkContext:

1. Acquires **executors** on nodes in the cluster, which are processes that run computations and store data for your application
2. Sends your application code (defined by JAR or Python files) to the executors
3. Sends **tasks** to the executors to run

# Cluster Mode (2/2)



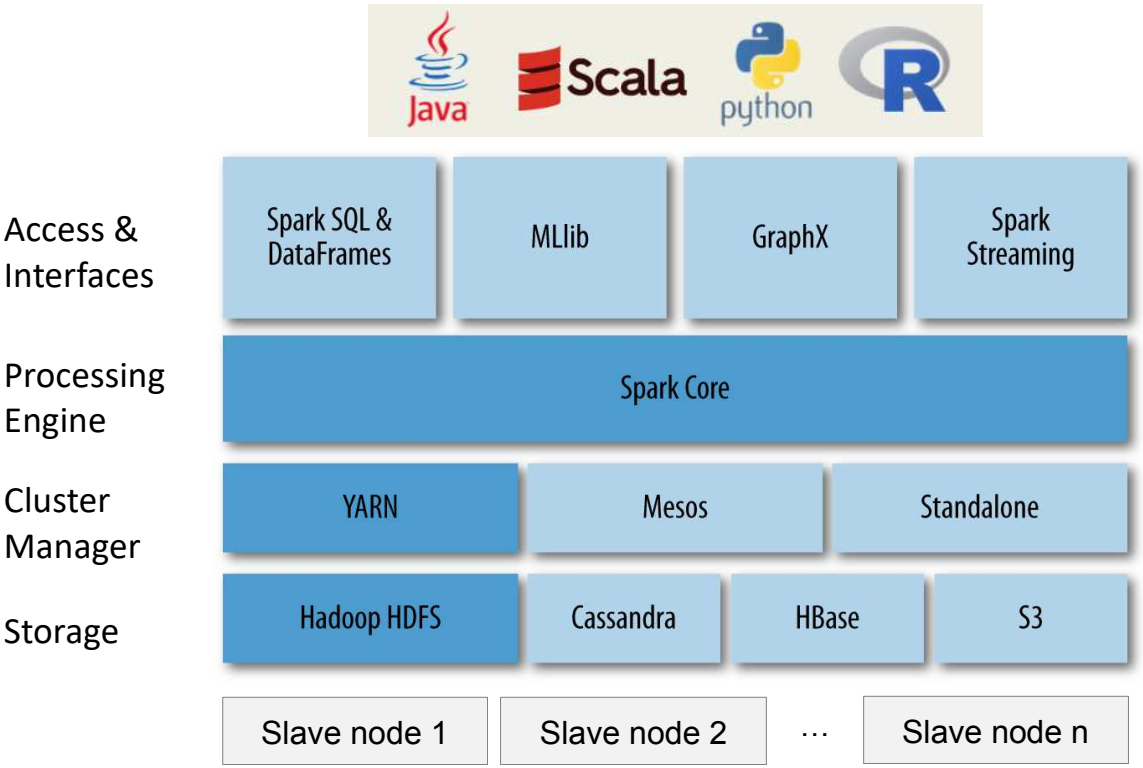

Each application gets its own executor processes

Benefit: Isolation of applications from each other (both scheduling and executor side)
Disadvantage: Data cannot be shared across different Spark applications without writing it to external storage

Spark is agnostic to the underlying cluster manager

Because the driver schedules tasks on the cluster, it should be run close to the worker nodes

# Spark Stack




https://www.safaribooksonline.com/library/view/data-analytics-with/9781491913734/ch04.html

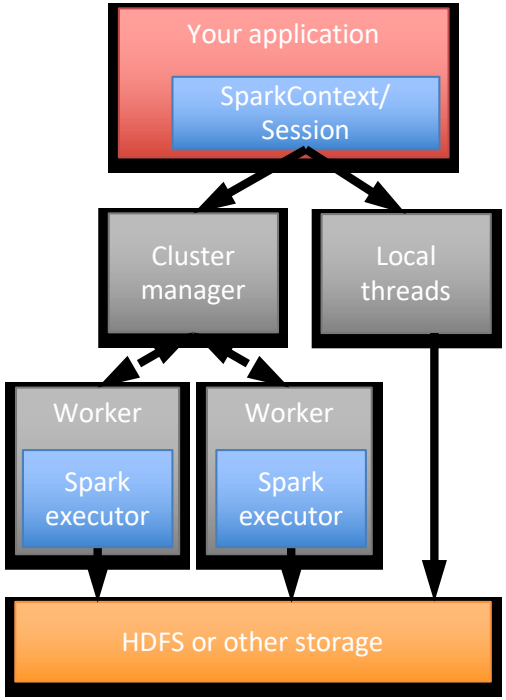
# Spark core
# Resilient Distributed Datasets (RDDs)

# Key Idea

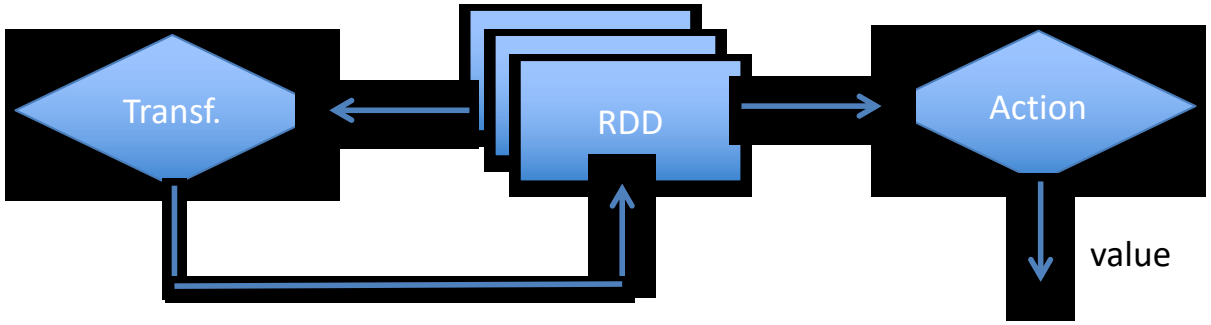
- Resilient distributed datasets (RDDs)
  - Immutable collections of objects spread across a cluster (partitions)
  - Built through parallel transformations (map, filter, etc)
  - Automatically rebuilt on failure
  - Controllable persistence (e.g., caching in RAM)

# Software Components

- Spark runs as a library in your program (one instance per app)
- The starting point of any Spark program is the SparkContext/Session
  - Contains various methods to manipulate RDD/DataFrames
- Runs tasks locally or on a cluster
  - Standalone, Mesos or YARN cluster
- Accesses storage via Hadoop InputFormat API
  - Can use HBase, HDFS, S3, …



# Operations




- Transformations (e.g., map, filter, groupBy, join)
  - Lazy operations to build RDDs/DataFrames from other RDD/DataFrames
- Actions (e.g., count, collect, save)
  - Return a result or write it to storage

# Why lazy operations are good?

- If you tell Spark to operate on a set of data:
  - it listens to what you ask it to do
  - it writes down some shorthand for it so it doesn't forget, and then does absolutely nothing
  - it will continue to do nothing, until you ask it for the final answer

- Often work magically goes away, postpone operations for optimization:
  - You first ask Spark to filter a petabyte of data to find all the point of sale records for the Milan store
  - Then you ask to give you just the first result that comes back
  - Postponing operations, Spark will just find the first Milan POS record
    - much easier than first filtering everything, then picking out only the first line

# Example: Mining Console Logs

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

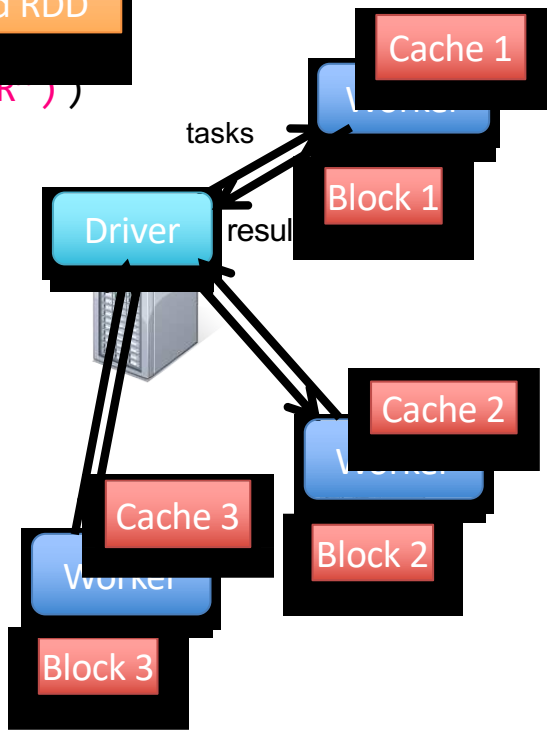
```
logLines = spark.textFile(“hdfs://...”)
errorsRDD = logLines.filter(lambda s: s.startswith(‘ERROR’))
messagesRDD = errorsRDD.map(lambda s: s.split(‘\t’)[2])
messagesRDD.cache()
```

Base RDD

Transformed RDD

Action

```
messagesRDD.filter(lambda s: “foo” in s).count()
messagesRDD.filter(lambda s: “bar” in s).count()
. . .
```



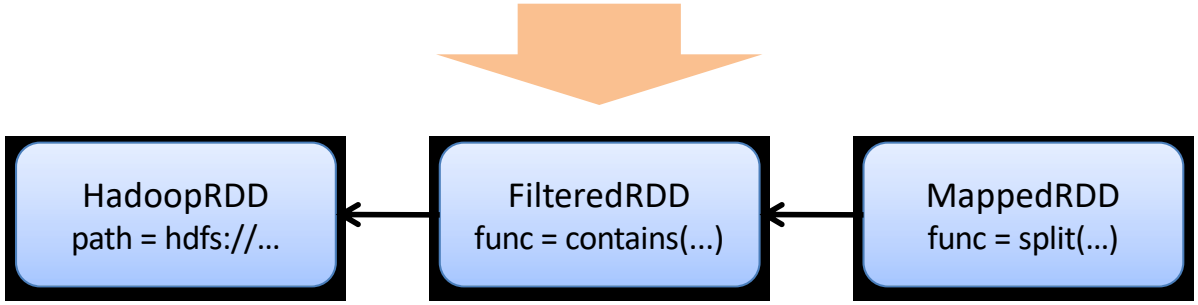

**Result:** full-text search of Wikipedia in <1 sec
(vs 20 sec for on-disk data)

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

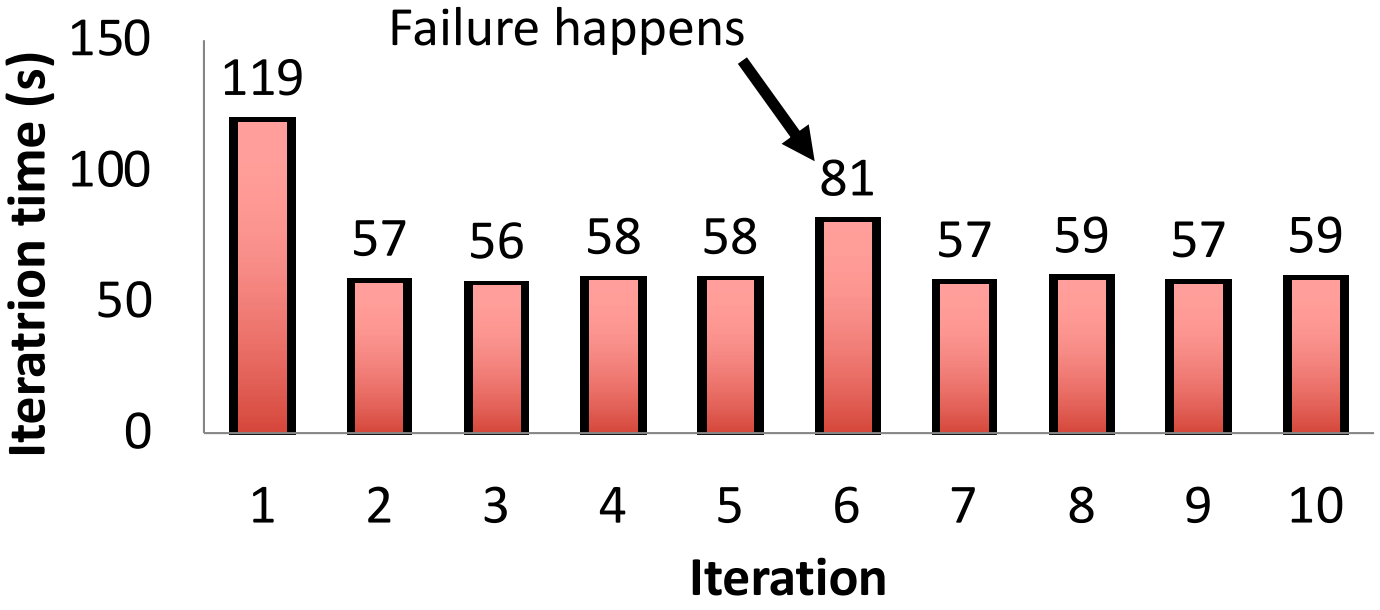
# RDD Fault Tolerance

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

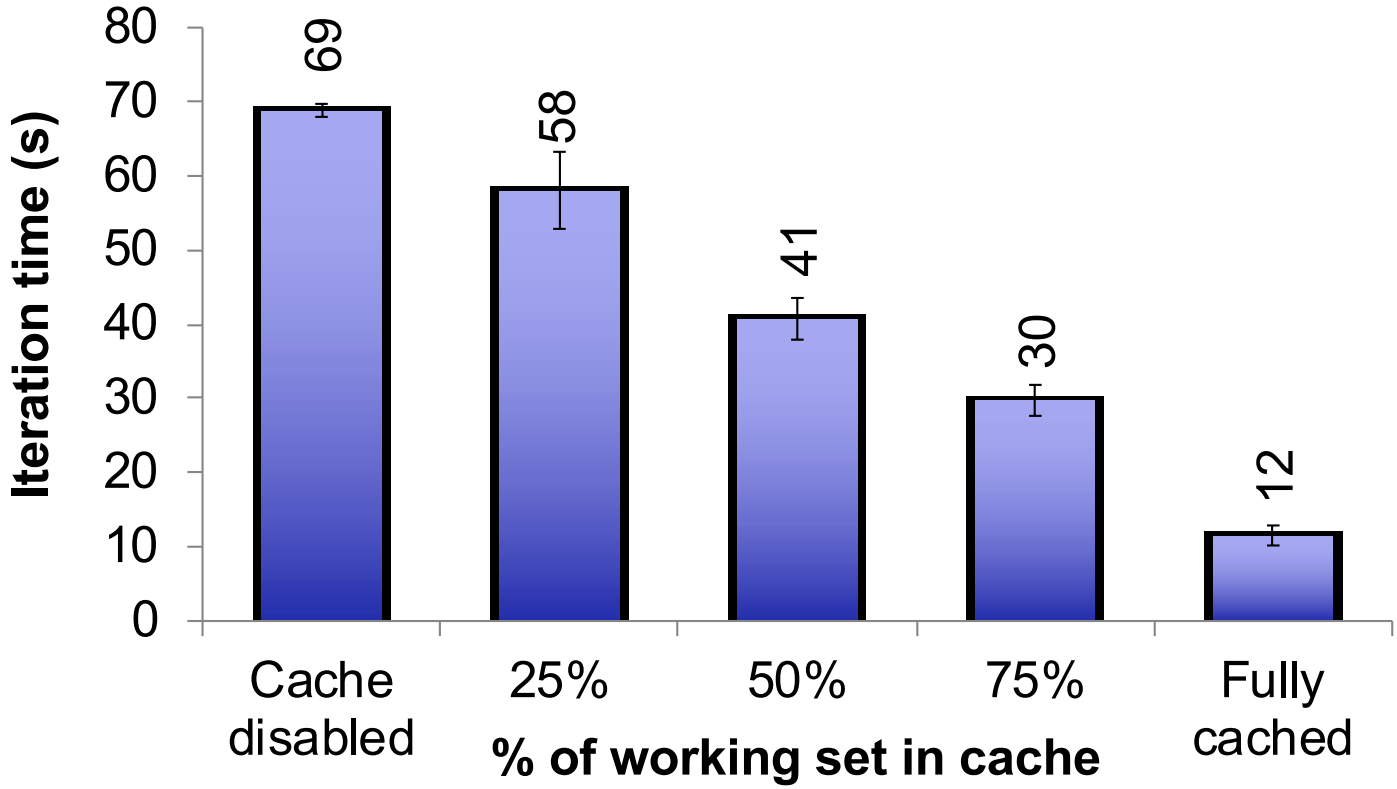
E.g:
```
messages = textFile(...).filter(lambda s: s.contains("ERROR"))
                        .map(lambda s: s.split('\t')[2])
```



# Fault Recovery Test



# Behavior with Less RAM



# Pig and HIVE?


- The latest release of Pig and Hive compile into Spark jobs and not only MapReduce

- High level languages and integration with Machine Learning libraries, etc.

```
pig -x spark
```

```
hive> set hive.execution.engine=spark;
```

# Performance degradation?

- Network optimizations:
  - Can only reduce job completion time by a median of at most 2%
  - Not a bottleneck because much less data is sent over the network than is transferred to and from disk
  - Network I/O is mostly even on 1Gbps networks

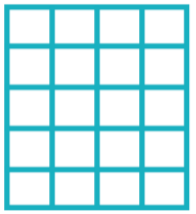
- Disk access optimization or elimination
  - Can only reduce job completion time by a median of at most 19%
  - CPU utilization is typically much higher than disk utilization
  - Use more sophisticated serialization and compression techniques

- Optimizing stragglers:
  - Can only reduce job completion time by a median of at most 10%
  - Java garbage collection and disk data transfer time

K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, B. G. Chun.
Making Sense of Performance in Data Analytics Frameworks. USENIX 2015

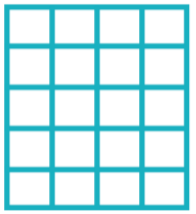
# DataFrames

# From RDDs to DataFrames



- Most data is structured (e.g., CSV, ...)
  - Programming RDDs inevitably ends up with a lot of tuples (x[0],x[1], ...)
- Functional transformations (e.g., map/reduce) are not as intuitive

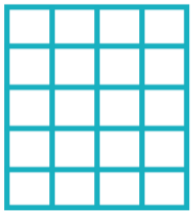
# From RDDs to DataFrames



```
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()
```

databricks

# From RDDs to DataFrames

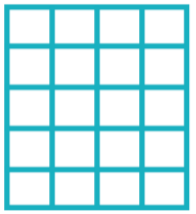


```
pdata.map(lambda x: (x.dept, [x.age, 1])) \
  .reduceByKey(lambda x, y: [x[0] + y[0], x[1] + y[1]]) \
  .map(lambda x: [x[0], x[1][0] / x[1][1]]) \
  .collect()
```

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

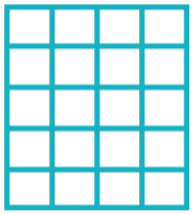
databricks

# DataFrames in Spark



- Distributed collection of data grouped into named columns (i.e., RDD with schema)
- Provides abstraction for selecting, filtering, aggregating and plotting structured data
  - Selecting required columns
  - Joining different data sources
  - Aggregation (count, sum, average, etc)
- Domain-specific functions designed for common tasks
  - Metadata
  - Sampling
  - User Defined Functions (UDFs)
- Available in Python, Scala, Java, and R (via SparkR)

# DataFrames in Spark



- Immutable once constructed
- Track lineage
- Enable distributed computations

- How to construct DataFrames
  - Reading from file(s)
  - Transforming an existing DFs (Spark or Pandas) or RDDs
  - Parallelizing a python collection list
  - Applying transformations and actions

# Create a DataFrame from a csv file

```
In [1]: df_data_1 = spark.read.format('org.apache.spark.sql.execution.datasources.csv.CSVFileFormat')\
                    .option('header', 'true').load('2015-summary.csv')
        df_data_1.take(5)
```

```
Out[1]: [Row(DEST_COUNTRY_NAME='United States', ORIGIN_COUNTRY_NAME='Romania', count='15'),
         Row(DEST_COUNTRY_NAME='United States', ORIGIN_COUNTRY_NAME='Croatia', count='1'),
         Row(DEST_COUNTRY_NAME='United States', ORIGIN_COUNTRY_NAME='Ireland', count='344'),
         Row(DEST_COUNTRY_NAME='Egypt', ORIGIN_COUNTRY_NAME='United States', count='15'),
         Row(DEST_COUNTRY_NAME='United States', ORIGIN_COUNTRY_NAME='India', count='62')]
```

https://www.rita.dot.gov/bts/help_with_data/aviation/index.html

# Create a DataFrame from a csv file

- Getting the schema

```
df_data_1.printSchema()
```

```
root
 |-- DEST_COUNTRY_NAME: string (nullable = true)
 |-- ORIGIN_COUNTRY_NAME: string (nullable = true)
 |-- count: string (nullable = true)
```

# Filter – US outgoing flights

```
us_from_flights=df_data_1.filter(“ORIGIN_COUNTRY_NAME =='United States'”)

# Alternatively, using Pandas-like syntax
us_from_flights=df_data_1[df_data_1.ORIGIN_COUNTRY_NAME =='United States']
us_from_flights.show()
```

| DEST_COUNTRY_NAME | ORIGIN_COUNTRY_NAME | count |
|---|---|---|
| Egypt | United States | 15 |
| Costa Rica | United States | 588 |
| Senegal | United States | 40 |
| Moldova | United States | 1 |
| Guyana | United States | 64 |
| Malta | United States | 1 |
| Anguilla | United States | 41 |
| Bolivia | United States | 30 |

# Filter – US outgoing flights

- Select only destination and number of flights

```
us_from_flights=df_data_1.filter(“ORIGIN_COUNTRY_NAME
=='United States’”).select("DEST_COUNTRY_NAME", "COUNT")

us_from_flights.show()
```

```
+--------------------+-----+
|   DEST_COUNTRY_NAME|COUNT|
+--------------------+-----+
|               Egypt|   15|
|          Costa Rica|  588|
|             Senegal|   40|
|             Moldova|    1|
|              Guyana|   64|
|               Malta|    1|
|            Anguilla|   41|
```

# Filter – US outgoing flights

- Select only destination and number of flights

```
us_from_flights=df_data_1.where("ORIGIN_COUNTRY_NAME
=='United States'").select("DEST_COUNTRY_NAME", "COUNT")

us_from_flights.show()
```

```
+--------------------+-----+
|   DEST_COUNTRY_NAME|COUNT|
+--------------------+-----+
|               Egypt|   15|
|          Costa Rica|  588|
|             Senegal|   40|
|             Moldova|    1|
|              Guyana|   64|
|               Malta|    1|
|            Anguilla|   41|
```

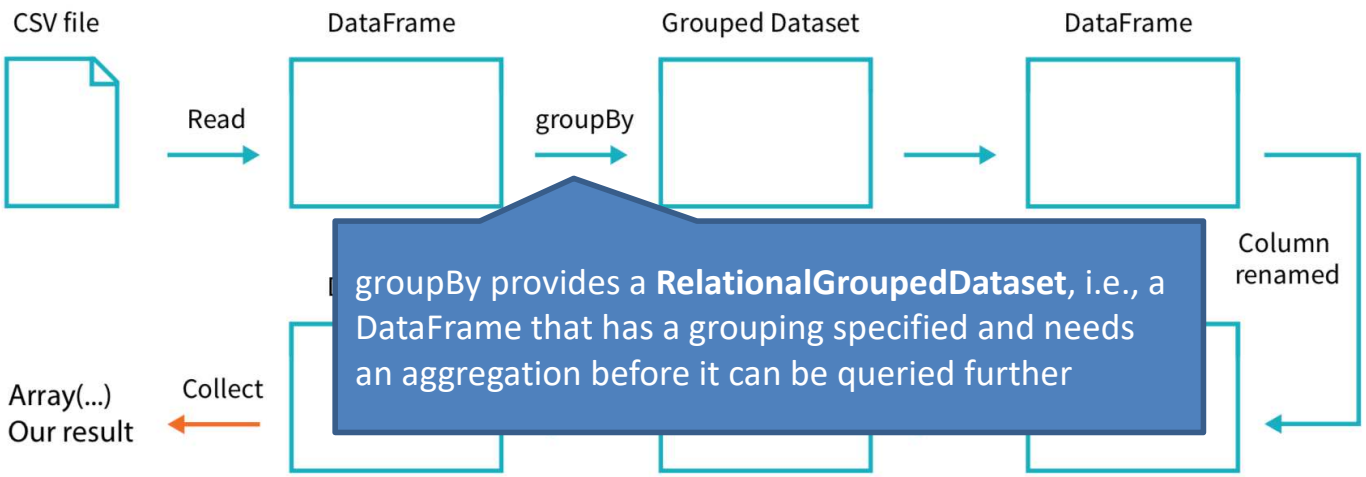
# Filter – Applying types

- Find out the top five destination countries

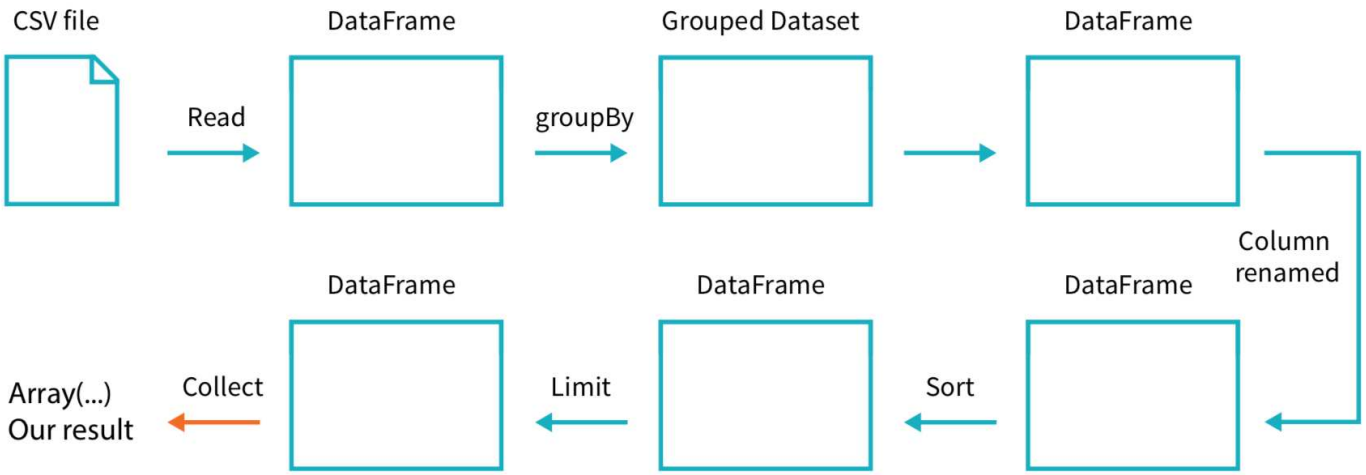
```
from pyspark.sql.types import DoubleType
from pyspark.sql.functions import desc
df_flights2015=df_data_1.withColumn(“count”,df_data_1[“count”].cast(“double”))

df_flights2015.groupBy(“dest_country_name”).\
sum(“count”).\
withColumnRenamed(“sum(count)”,”destination_total”).\
sort(desc(“destination_total”)).limit(5).collect()
```

```
[Row(DEST_COUNTRY_NAME='United States', destination_total=411352.0),
 Row(DEST_COUNTRY_NAME='Canada', destination_total=8399.0),
 Row(DEST_COUNTRY_NAME='Mexico', destination_total=7140.0),
 Row(DEST_COUNTRY_NAME='United Kingdom', destination_total=2025.0),
 Row(DEST_COUNTRY_NAME='Japan', destination_total=1548.0)]
```

# Filter – Lineage



# Filter – Lineage



# Joining two DataFrames

- Let's evaluate the flights delta between 2015 and 2014

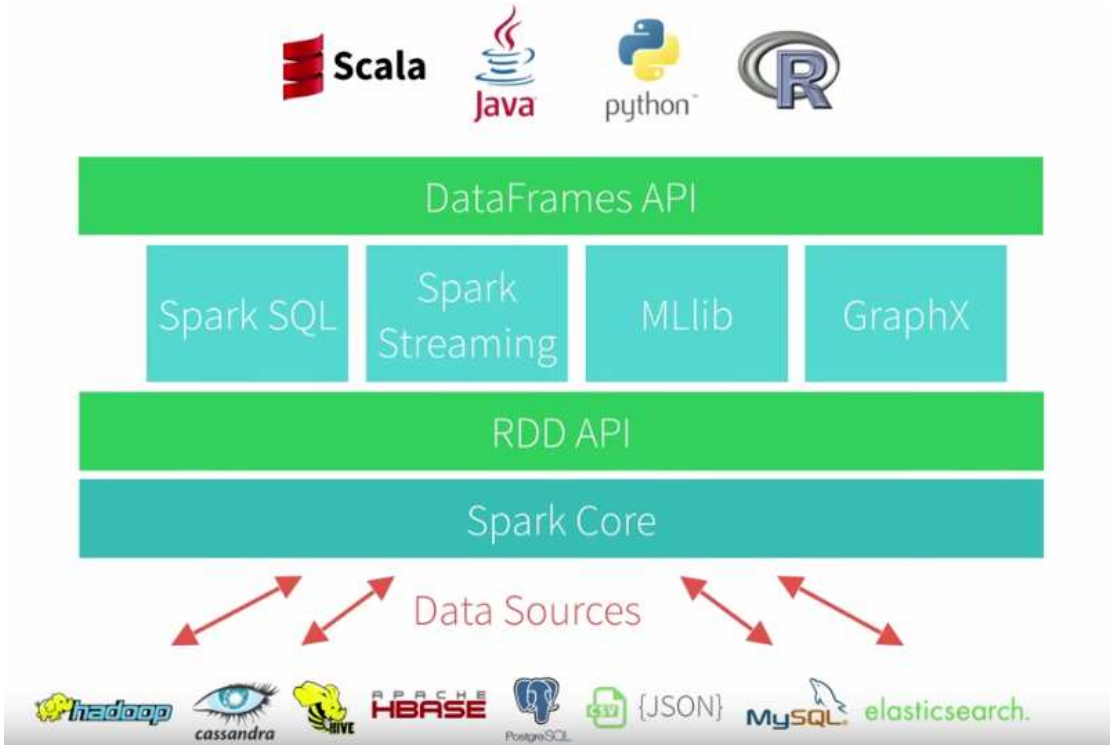
```
df_data_2 = spark.read\
.format('org.apache.spark.sql.execution.datasources.csv.CSVFileFormat')\
.option('header', 'true')\
.load(cos.url('2014-summary.csv',
'apccourse2017180c7db5912f304fcd8f4cf0d147b8b65c'))
df_data_2.take(5)
df_flights2014=df_data_2.withColumn("COUNT",
df_data_2["COUNT"].cast("double")).withColumnRenamed("COUNT","COUNT2014")
```

# Joining two DataFrames

```
df_join=df_flights2014\
.join(df_flights2015,(df_flights2014.DEST_COUNTRY_NAME==\
                      df_flights2015.DEST_COUNTRY_NAME)&\
                      (df_flights2014.ORIGIN_COUNTRY_NAME==\
                      df_flights2015.ORIGIN_COUNTRY_NAME))\
.select(df_flights2014.DEST_COUNTRY_NAME,\
        df_flights2014.ORIGIN_COUNTRY_NAME,"COUNT2014","COUNT")

df_join.take(5)
```

```
[Row(DEST_COUNTRY_NAME='United States', ORIGIN_COUNTRY_NAME='Romania', COUNT2014=12.0, COUNT=15.0),
 Row(DEST_COUNTRY_NAME='United States', ORIGIN_COUNTRY_NAME='Croatia', COUNT2014=2.0, COUNT=1.0),
 Row(DEST_COUNTRY_NAME='United States', ORIGIN_COUNTRY_NAME='Ireland', COUNT2014=291.0, COUNT=344.0),
 Row(DEST_COUNTRY_NAME='United States', ORIGIN_COUNTRY_NAME='India', COUNT2014=62.0, COUNT=62.0),
 Row(DEST_COUNTRY_NAME='Egypt', ORIGIN_COUNTRY_NAME='United States', COUNT2014=11.0, COUNT=15.0)]
```

# Where is Spark today?

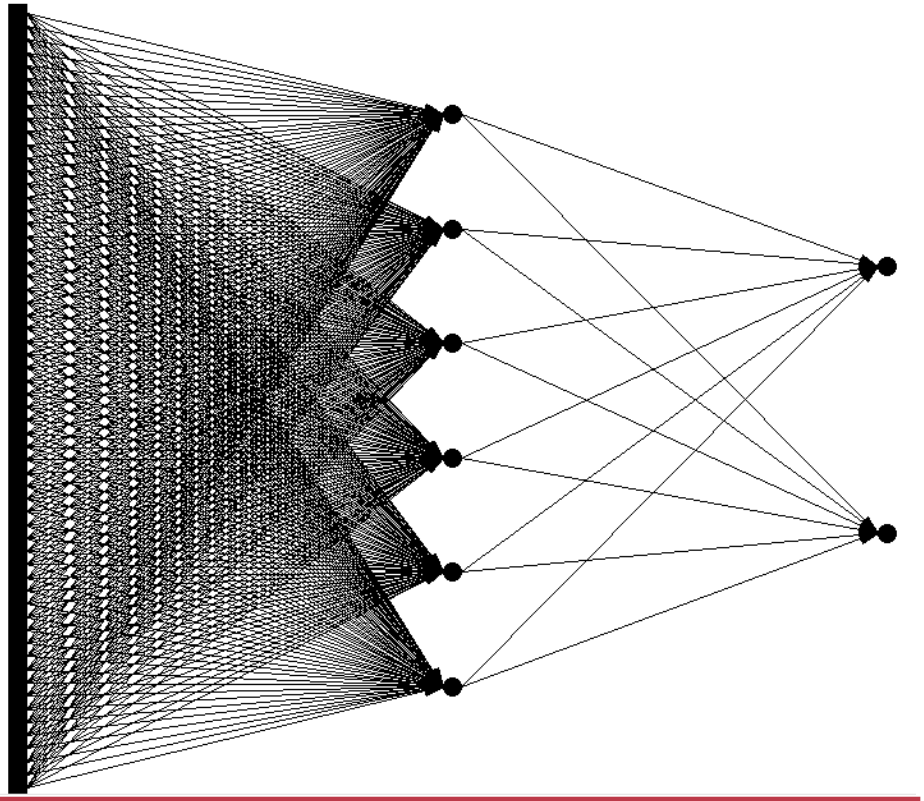



- Strengthening streaming
- Extend MLlib to deep learning

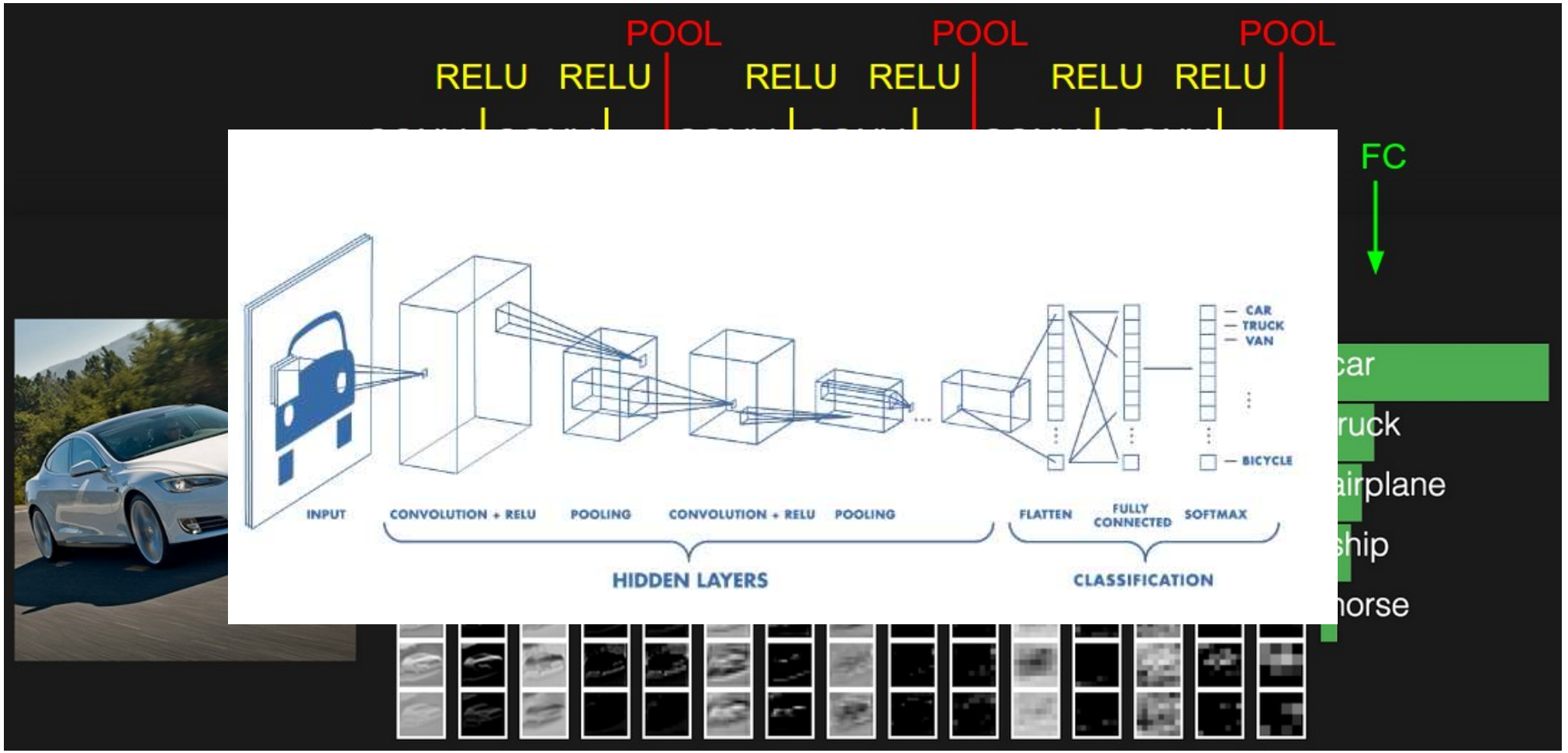
# ANNs – Feedforward Networks

- Network architecture
  - 60 input (one for each frequency bin)
  - 6 hidden
  - 2 output (0-1 for “Steve”, 1-0 for “David”)


“Hello”



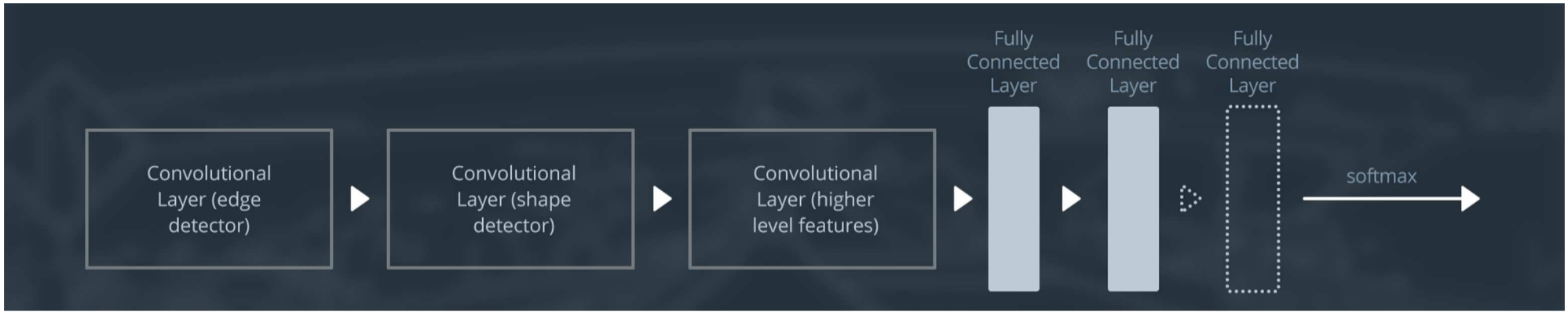
# Convolutional Neural Networks and Deep Learning



# Scale drives Deep Learning progress

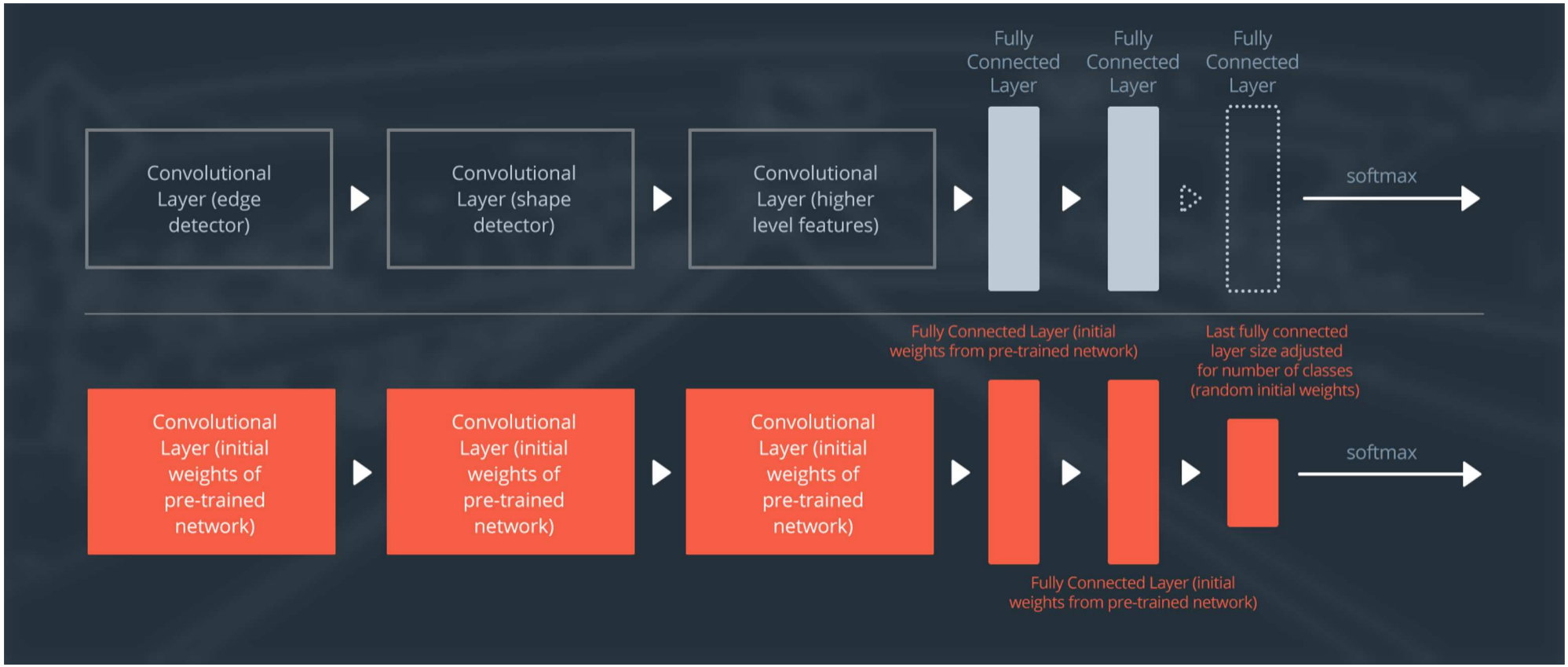

- Same ideas as in the in 1940s:
  - Labeled datasets were thousands of times too small
  - Computers were millions of times too slow

# SparkDL – Transfer learning

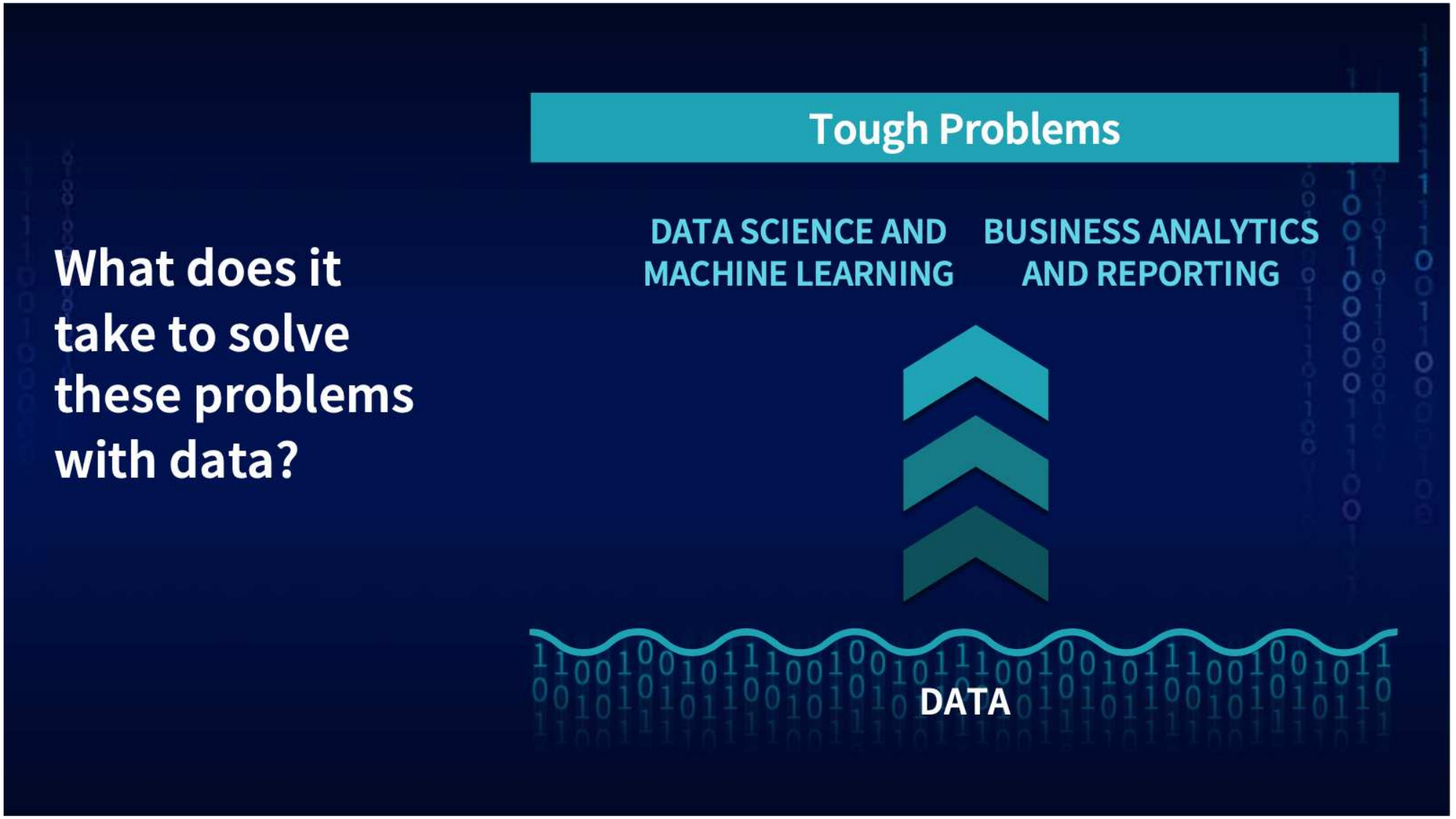

Source: https://www.yuchao.us/2017/04/self-driving-car-nd-a3.html

# SparkDL – Transfer learning

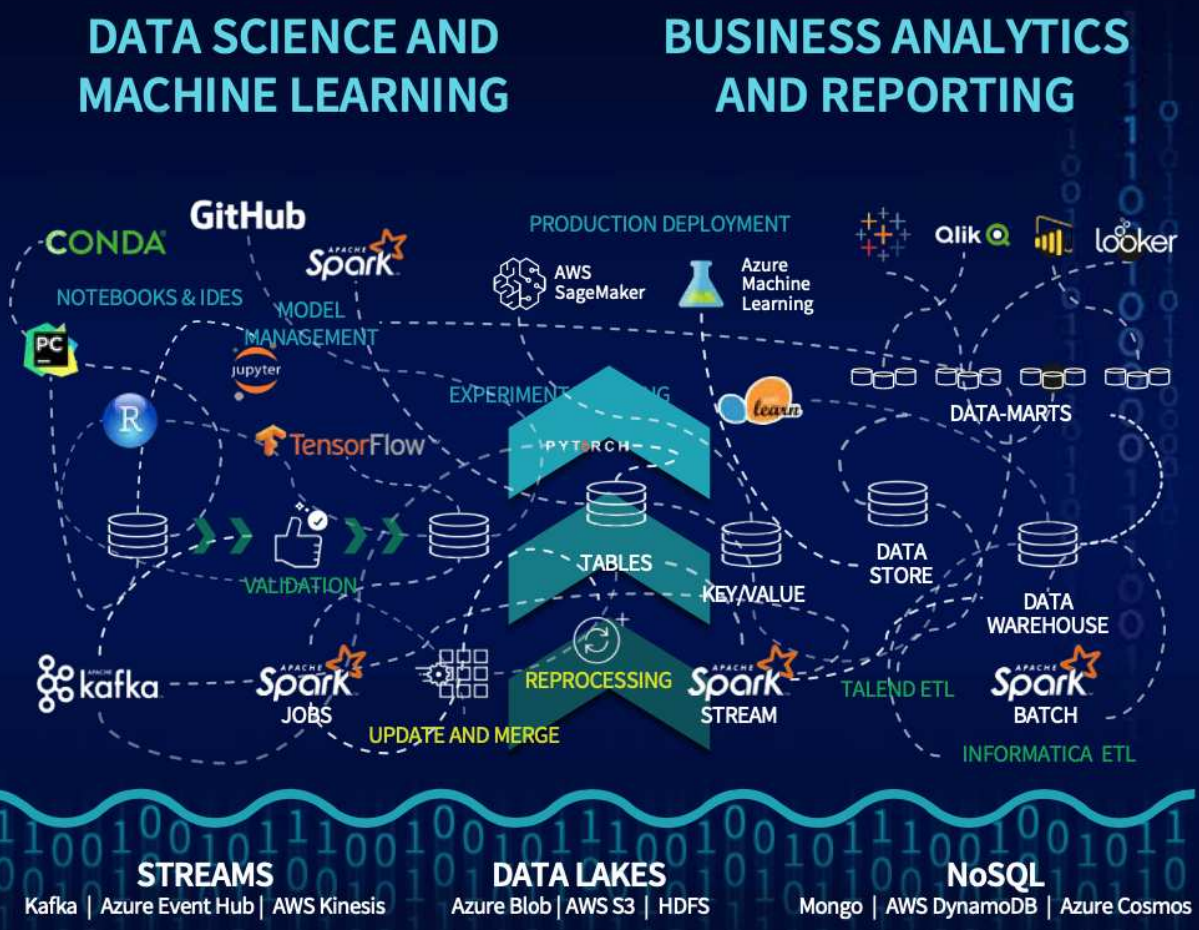

Source: https://www.yuchao.us/2017/04/self-driving-car-nd-a3.html

# New projects at databricks



# New projects at databricks

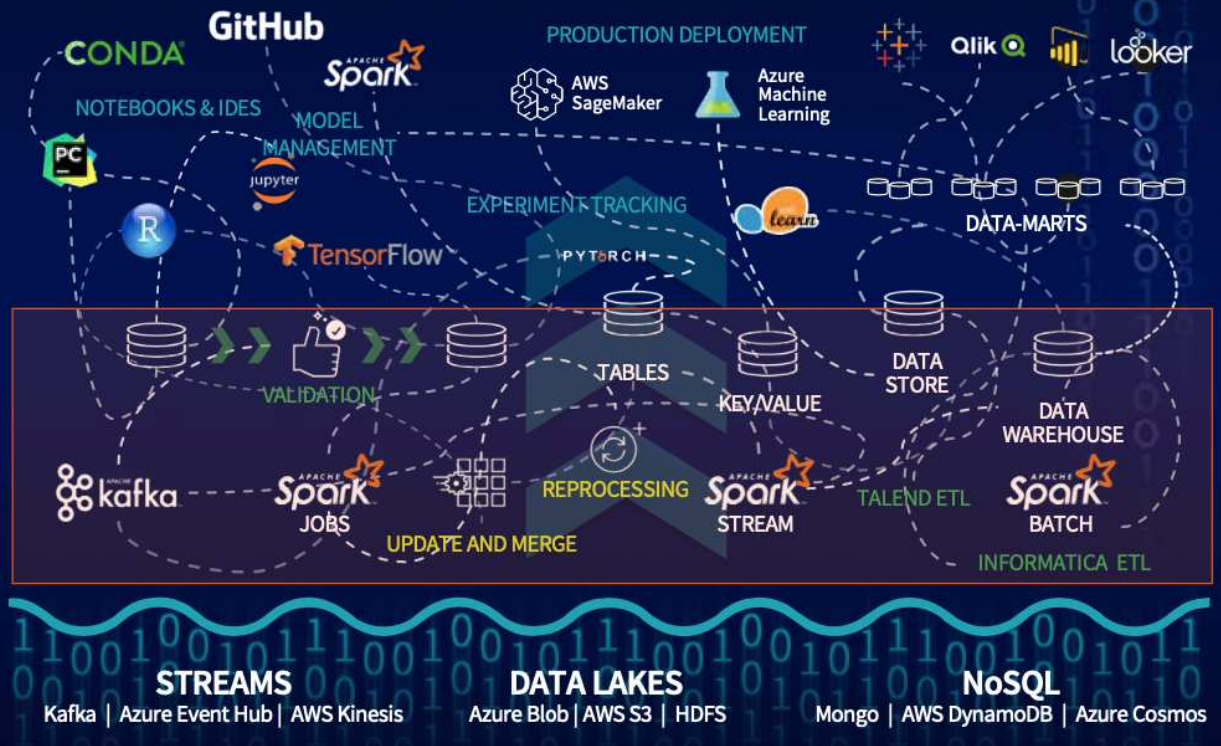
**Organizations are failing to solve these problems due to data, tech and people silos**



# New projects at databricks

**Why do organizations fail to unlock business value?**

**Data quality and reliability**
Multiple copies of inconsistent data built with unreliable pipelines

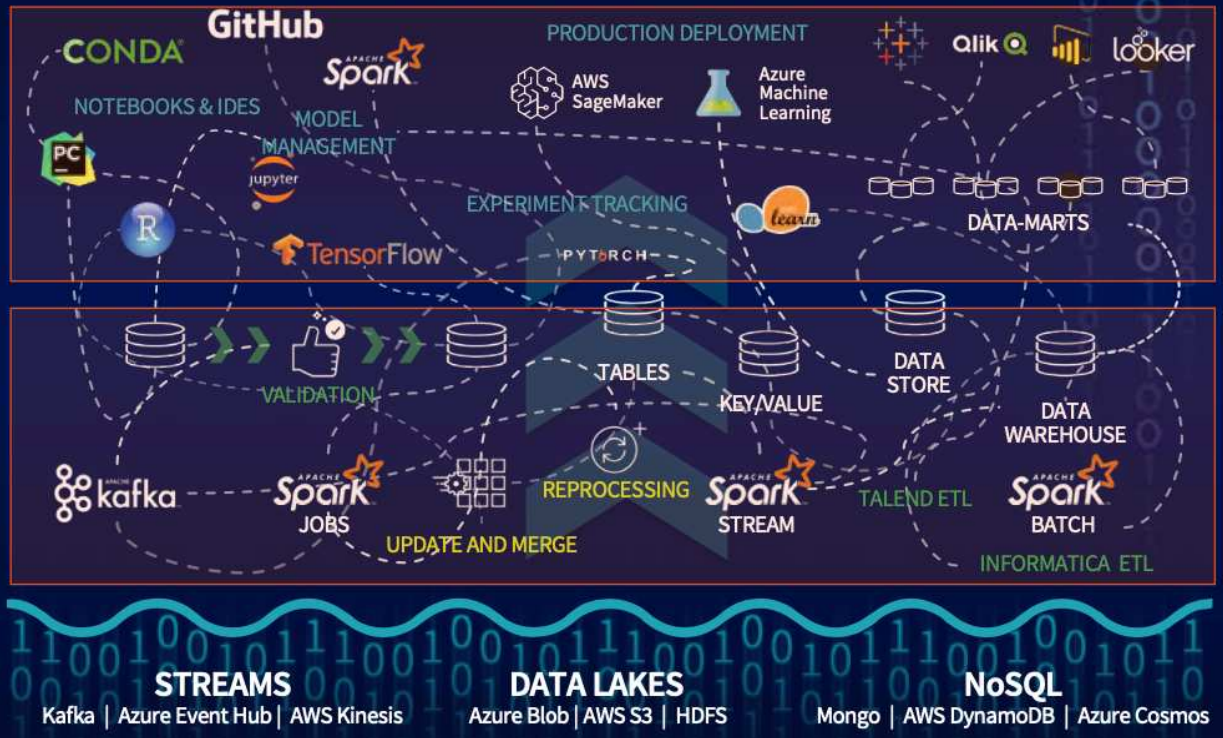


# New projects at databricks

**Why do organizations fail to unlock business value?**

**Disparate technologies**
Must stitch together dozens of software frameworks

**Data quality and reliability**
Multiple copies of inconsistent data built with unreliable pipelines



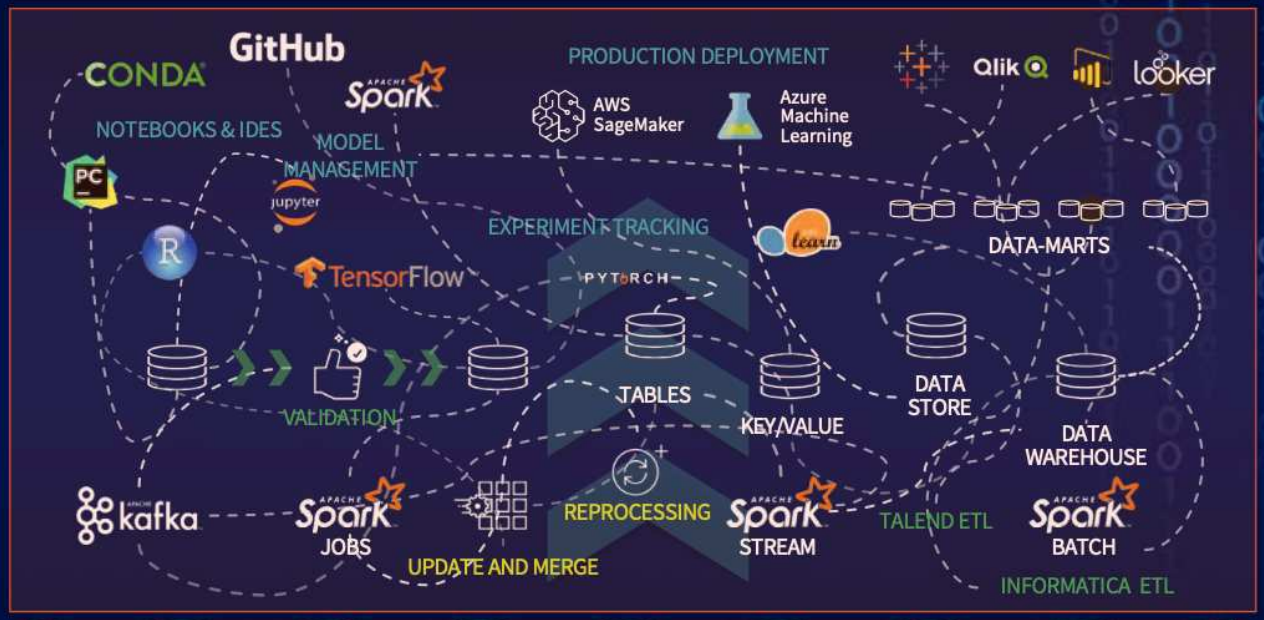
# New projects at databricks

**Why do organizations fail to unlock business value?**

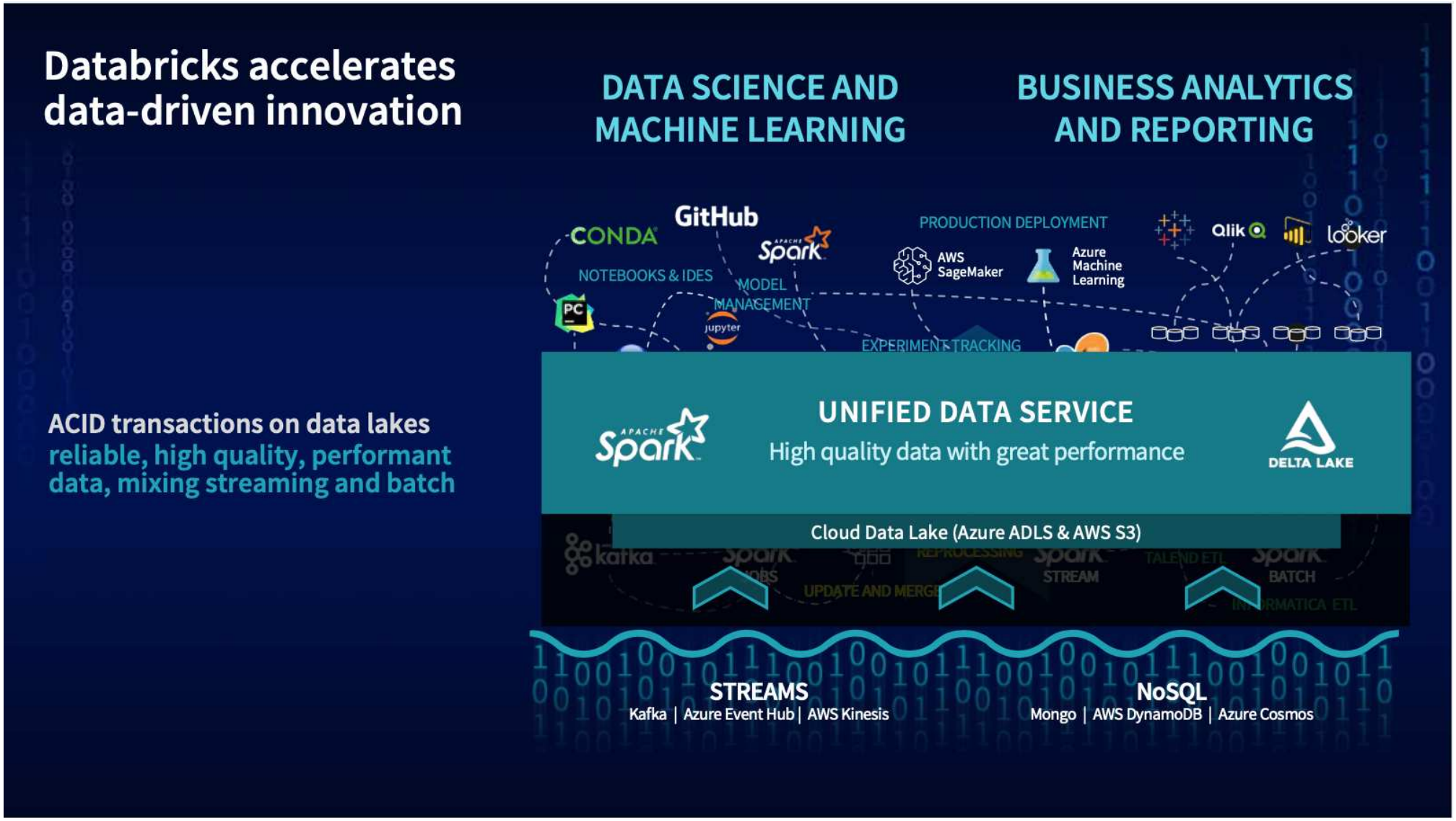
**Disparate technologies**
Must stitch together dozens of software frameworks

**Data quality and reliability**
Multiple copies of inconsistent data built with unreliable pipelines

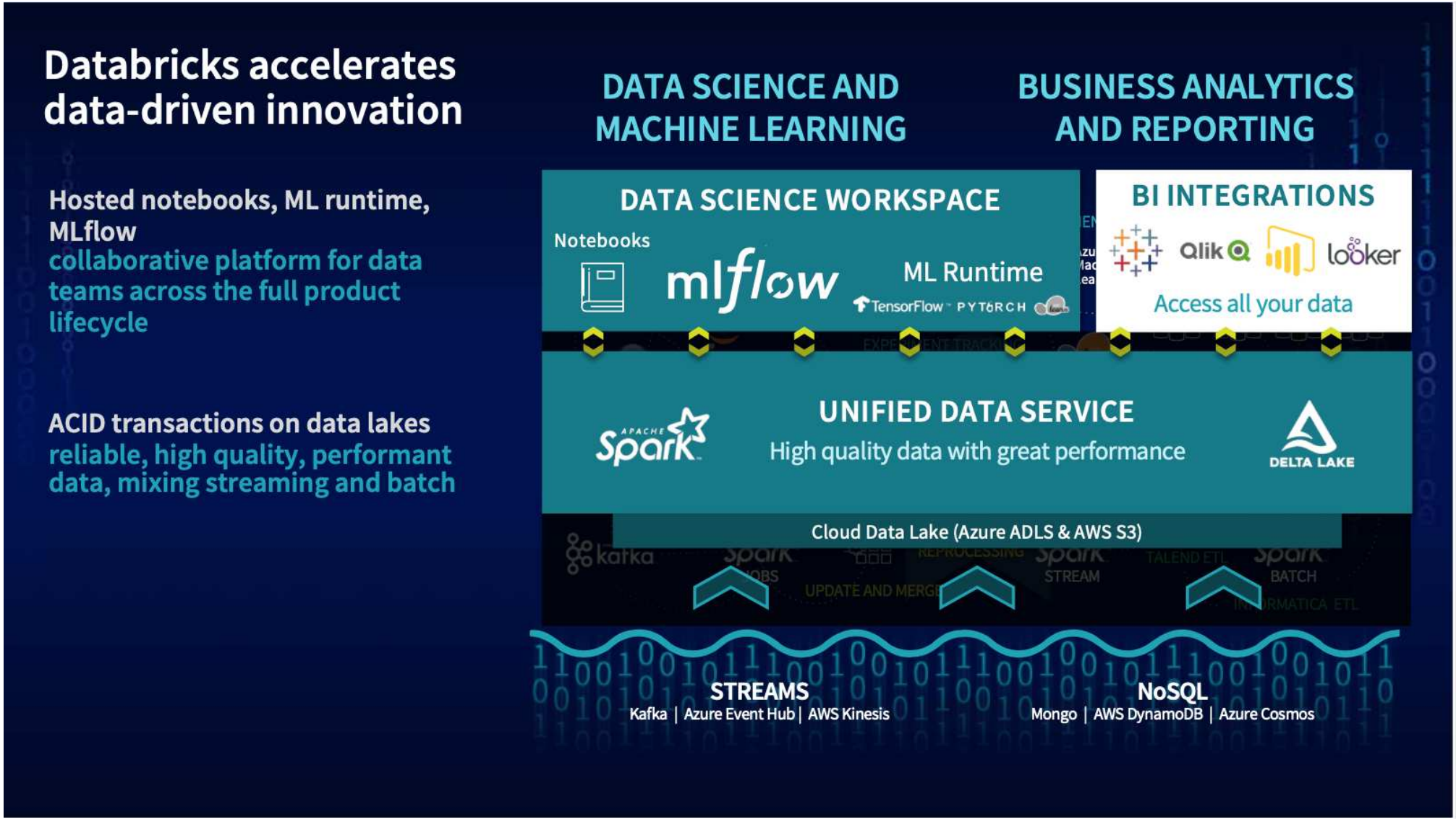
**Fragmented security**
End-to-end security and enterprise SLAs are a nightmare



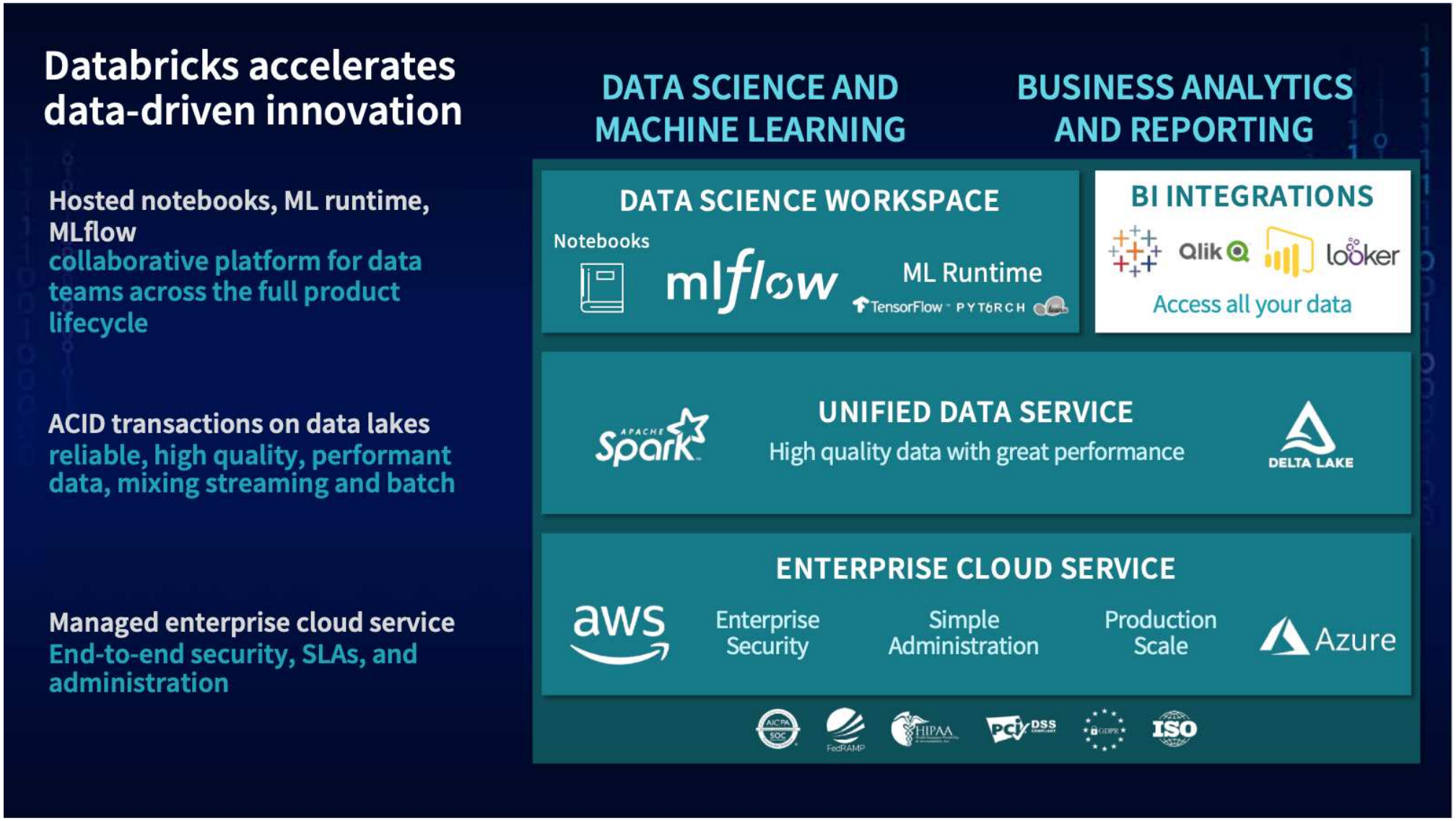
# New projects at databricks



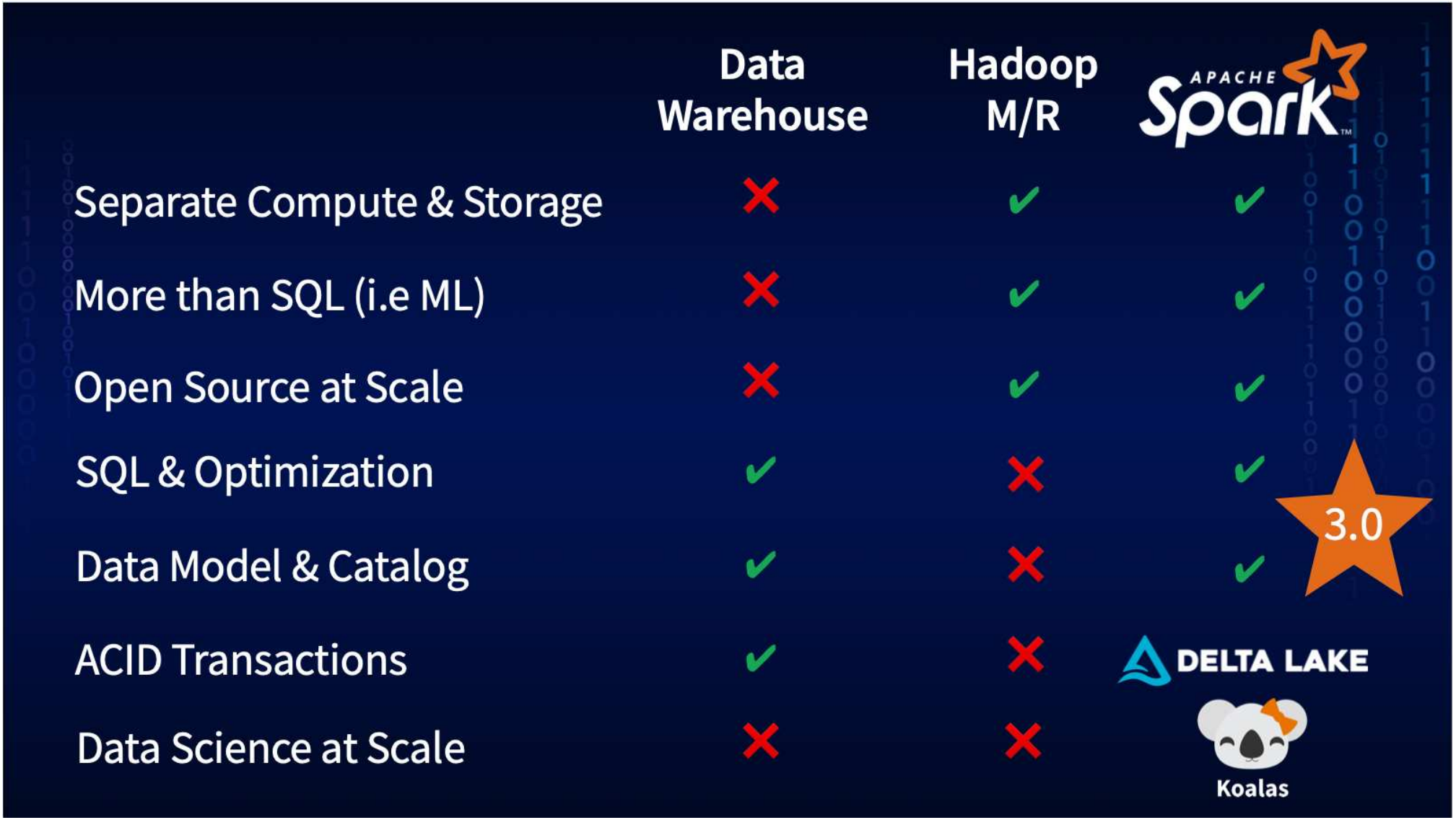
# New projects at databricks



# New projects at databricks



# Spark 3.0


Data Warehouse
Hadoop M/R
APACHE Spark
Separate Compute & Storage
More than SQL (i.e ML)
Open Source at Scale
SQL & Optimization
Data Model & Catalog
ACID Transactions
Data Science at Scale
3.0
DELTA LAKE
Koalas

# Spark 3.0

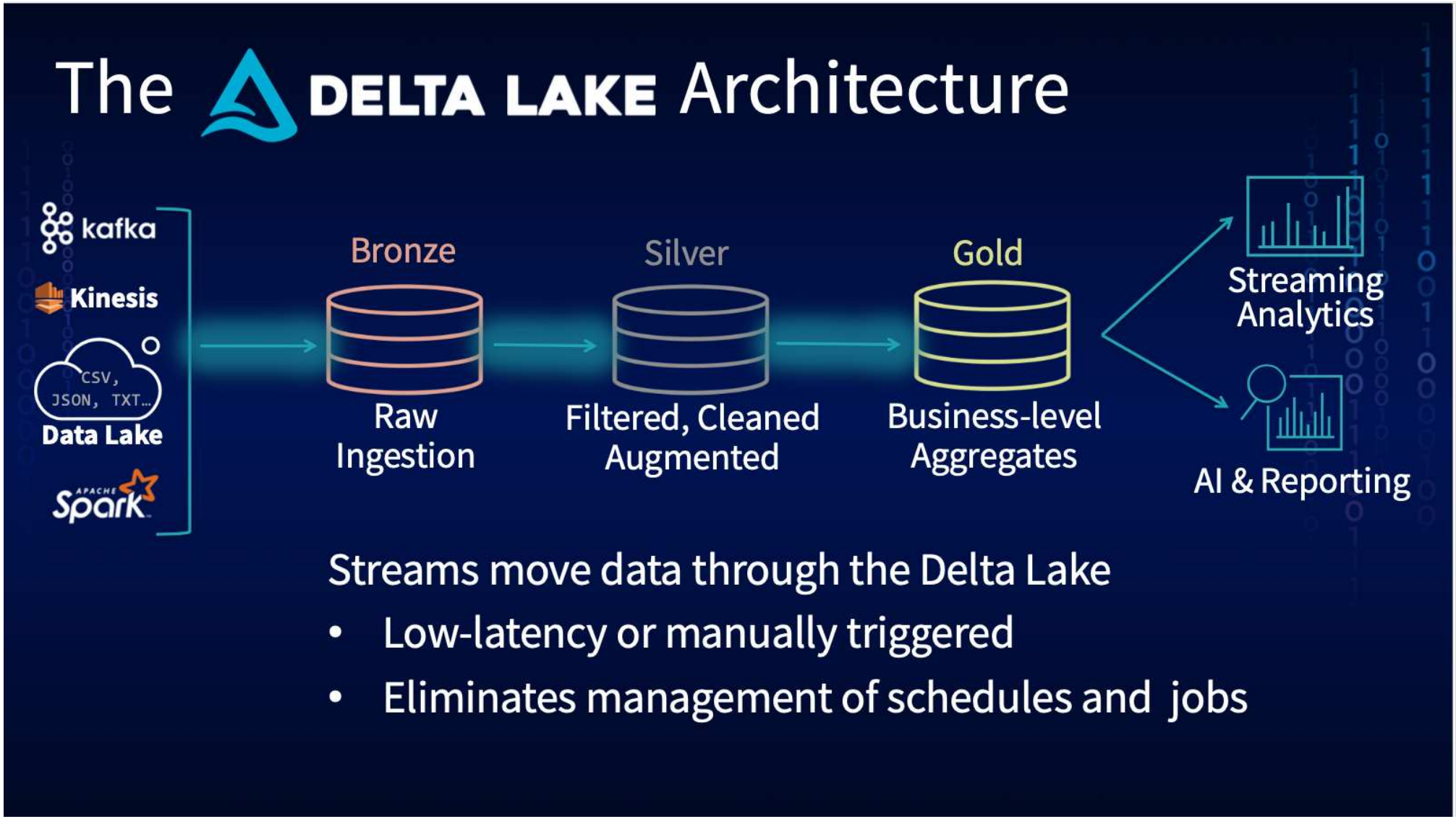
## Spark 3.0: Pluggable Data Catalog

**DataSourceV2**

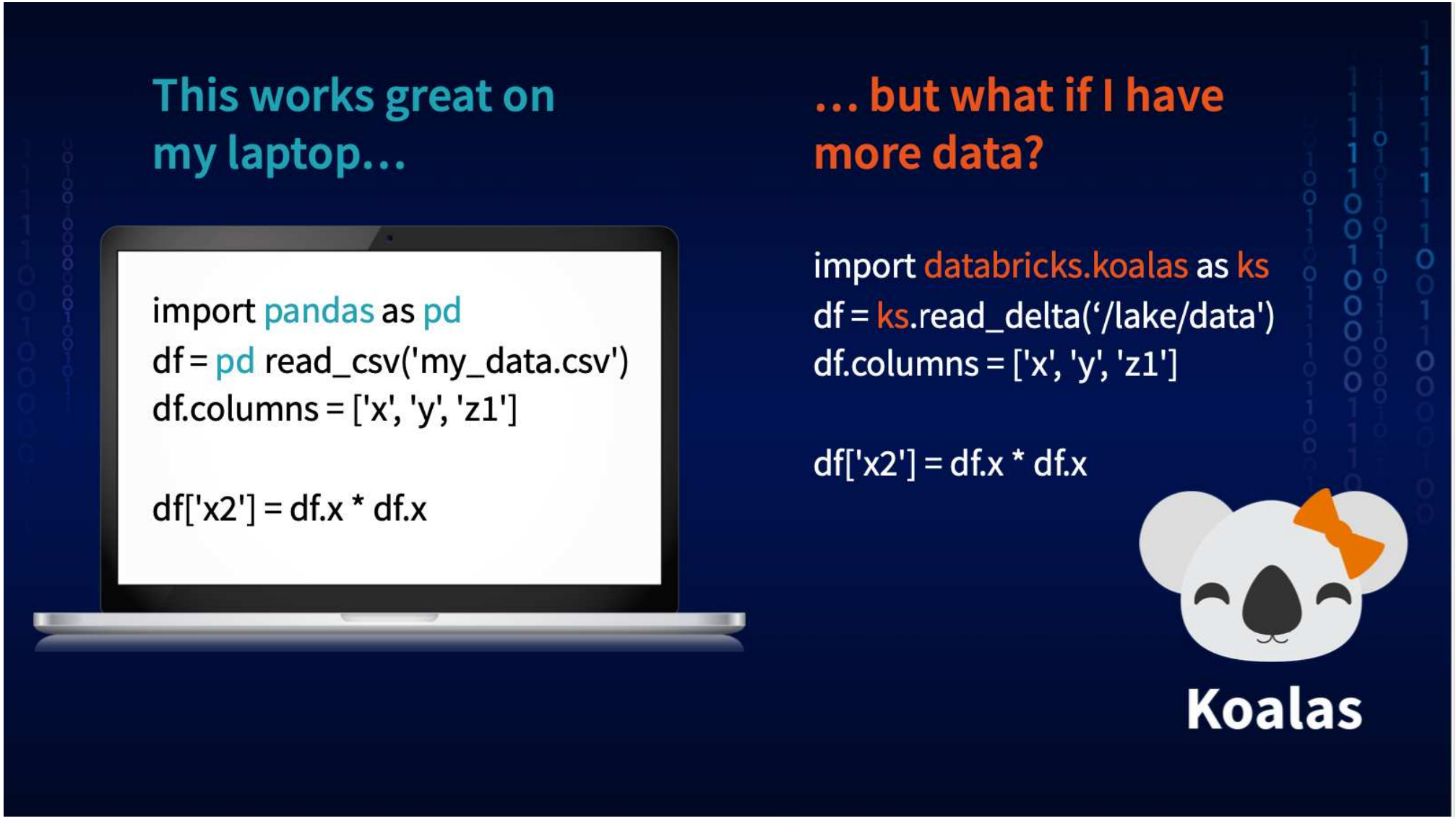
- Pluggable catalog integration
- Improved pushdown
- Unified APIs for streaming and batch

```
df.writeTo("catalog.db.table")
  .overwrite($"year" === "2019")
```

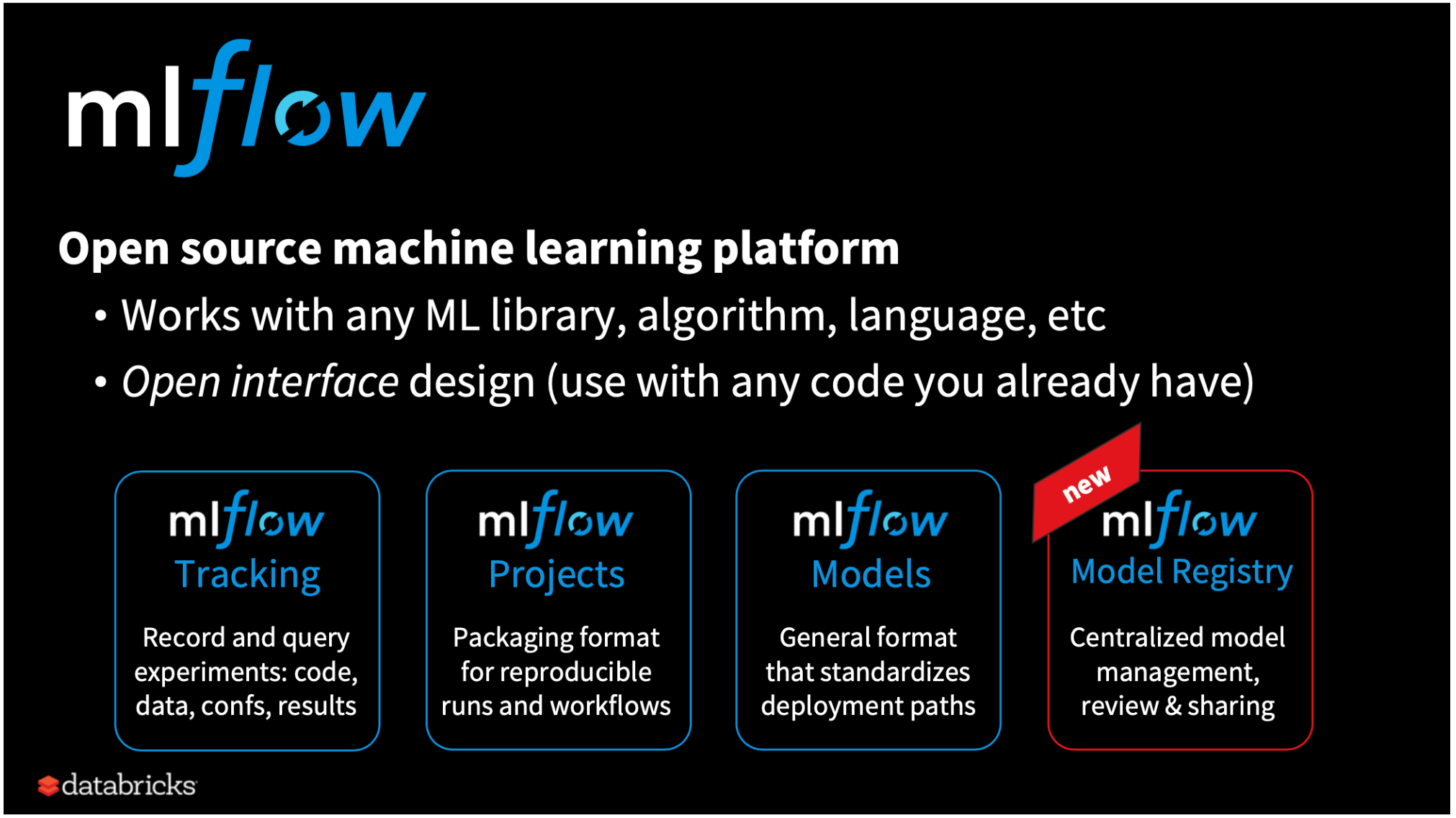
# Spark 3.0


The DELTA LAKE Architecture
kafka
Kinesis
CSV, JSON, TXT…
Data Lake
APACHE Spark
Bronze
Raw Ingestion
Silver
Filtered, Cleaned Augmented
Gold
Business-level Aggregates
Streaming Analytics
AI & Reporting
Streams move data through the Delta Lake
• Low-latency or manually triggered
• Eliminates management of schedules and jobs

# Spark 3.0

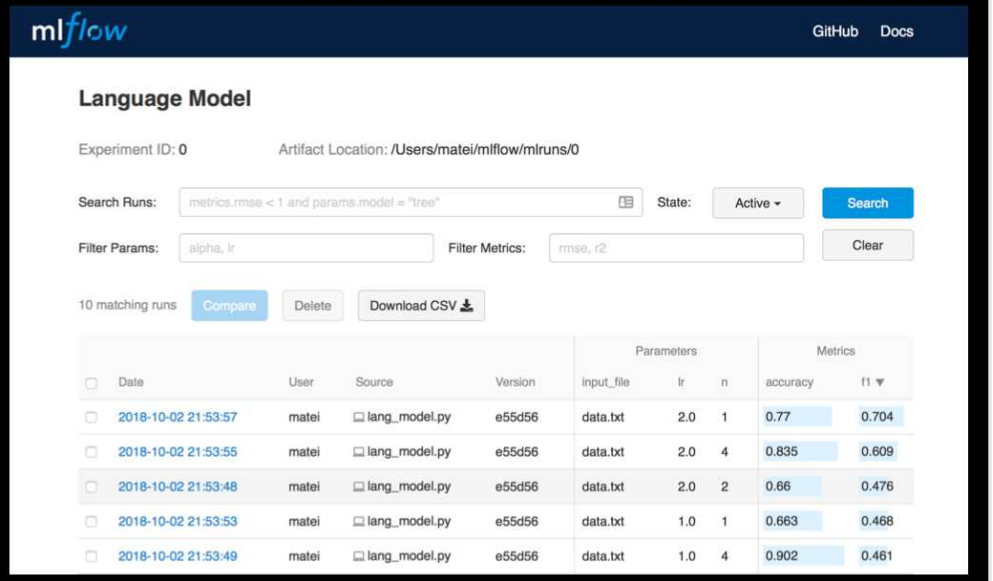

This works great on my laptop…
import pandas as pd
df = pd read_csv('my_data.csv')
df.columns = ['x', 'y', 'z1']
df['x2'] = df.x * df.x
… but what if I have more data?
import databricks.koalas as ks
df = ks.read_delta('/lake/data')
df.columns = ['x', 'y', 'z1']
df['x2'] = df.x * df.x
Koalas

# Spark 3.0


mlflow
Open source machine learning platform
• Works with any ML library, algorithm, language, etc
• Open interface design (use with any code you already have)
mlflow Tracking
Record and query experiments: code, data, confs, results
mlflow Projects
Packaging format for reproducible runs and workflows
mlflow Models
General format that standardizes deployment paths
new
mlflow Model Registry
Centralized model management, review & sharing
databricks

# MLflow Tracking

```
data    = load_text(file)
ngrams = extract_ngrams(data, N=n)
model   = train_model(ngrams,
                learning_rate=lr)
score   = compute_accuracy(model)

mlflow.log_param("data_file", file)
mlflow.log_param("n", n)
mlflow.log_param("learning_rate", lr)
mlflow.log_metric("score", score)

mlflow.keras.log_model(model)
```




Track parameters, metrics, output files & code version

databricks

# Thanks for your attention...


...any questions?

# References


- Hadoop:
  - T.K. Prasad. MapReduce architecture: Map Reduce Architecture. http://www.cs.wright.edu/~tkprasad/courses/cs707L06MapReduce.ppt
  - J. Dean, S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters.http://research.google.com/archive/mapreduce.html
  - S. Ghemawat, H. Gobioff, S.T. Leung. The Google File System. http://research.google.com/archive/gfs.html
  - V. K. Vavilapalli et al. Apache Hadoop YARN: Yet Another Resource Negotiator
  - Apache Tez. http://hortonworks.com/hadoop/tez/
  - Alan F. Gates. Making Hadoop Easy. https://cwiki.apache.org/HadoopDayAug2010.pptx

# References


- Big Data:
  - A. Sathi. Big Data Analytics
  - Big Data and Hadoop Simple and Easy
  - Big Data: the next frontier for innovation, competition, and productivity. McKinsey Global Institute. May, 2011. http://www.mckinsey.com/insights/business_technology/big_data_the_next_frontier_for_innovation
  - Analytics: The real-world use of big data. IBM Institute for Business Value In collaboration with Saïd Business School at the University of Oxford. 2012. http://www-03.ibm.com/systems/hu/resources/the_real_word_use_of_big_data.pd
  - Big Data & Analytics: Next Generation Architecture and Capabilities. Marc Andrews, 2014. https://www.ibm.com/partnerworld/wps/servlet/RedirectServlet?cmsId=isv_ast_smp_ecosystem-webcasts&attachmentName=Data_Warehouse_deck.pdf
  - Hype Cycle for Big Data, 2012. Gartner, 2012. https://www.gartner.com/doc/2100215/hype-cycle-big-data-
  - Big Data Landscape, version 3.0. Matt Turck, and Sutian Dong, 2014. http://pt.slideshare.net/mjft01/big-data-landscape-matt-turck-may-2014

# References


- MapReduce:
  - https://developer.yahoo.com/hadoop/tutorial/module4.html
  - M. T. Jones. Scheduling in Hadoop. http://www.ibm.com/developerworks/opensource/library/os-hadoop-scheduling/

- Spark:
  - M. Zaharia, M. C., T. Das, A. Dave, J. Ma, M. McCauley, M. J Franklin, S. Shenker, and I.Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In Proceedings of Networked Systems Design and Implementation Conference, 2012

# Credits

Part of these slides are based on the following courses:

- Giovanna Roda. Introduction to Hadoop. PRACE Autumn School ’21, 27–28 September 2021
- Danilo Ardagna. From MapReduce & Apache Hadoop to Apache Spark. PhD course on Data Management For Large-scale Analytics