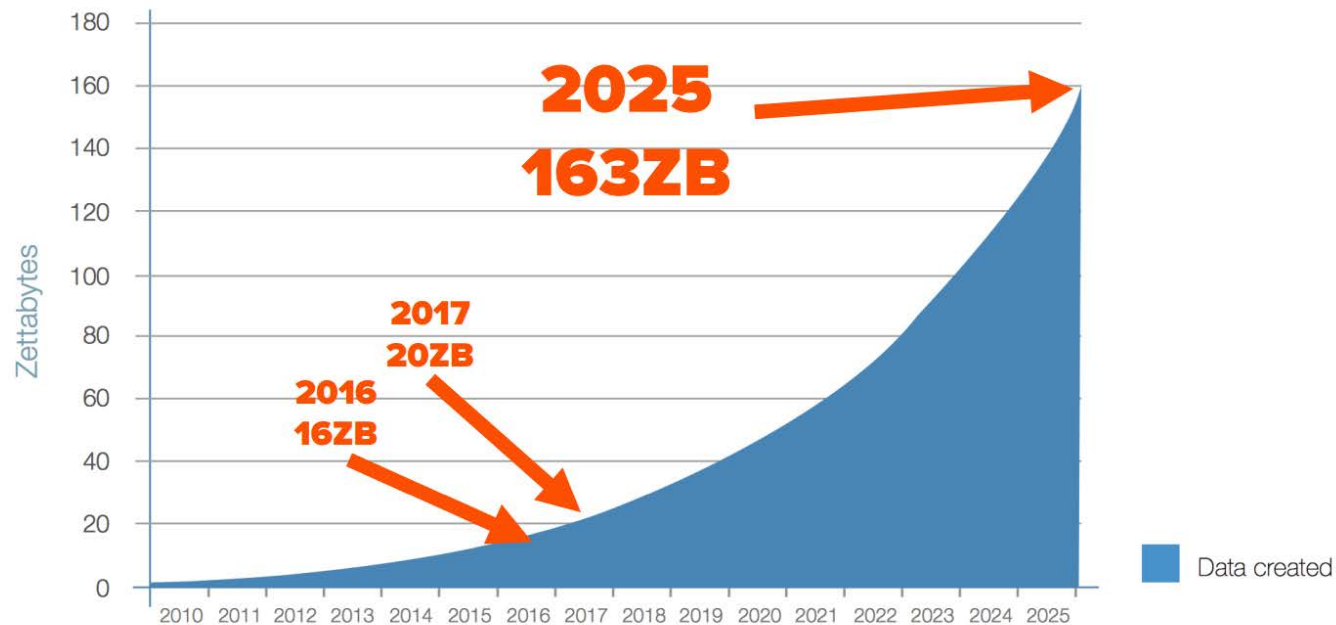# Hadoop

Data Deluge

- Billions of users connected through the Internet
  - WWW, FB, twitter, cell phones, …
  - 80% of the data on FB was produced last year
- Storage getting cheaper
  - Store more data!

2025
163ZB

2017
20ZB

2016
16ZB

Zettabytes

180
160
140
120
100
80
60
40
20
0

2010 2011 2012 2013 2014 2015 2016 2017 2018 2019 2020 2021 2022 2023 2024 2025

Data created

Source: IDC's Data Age 2025 study, sponsored by Seagate, April 2017

Why Hadoop

- Drivers:
    - 500M+ unique users per month
    - Billions of interesting events per day
    - Data analysis is key issue
- Need massive scalability
    - PB's of storage, millions of files, 1000's of nodes
- Need cost effectively
    - Use commodity hardware
    - Share resources among multiple projects
    - Provide scale when needed
- Need reliable infrastructure
    - Must be able to deal with failures – hardware, software, networking
        - Failure is expected rather than exceptional
    - Transparent to applications
        - very expensive to build reliability into each application

The Hadoop infrastructure provides these capabilities

Introduction to Hadoop

- Apache Hadoop
  - Open Source – Apache Foundation project
    - Yahoo! is apache platinum sponsor
- History
  - Started in 2005 by Doug Cutting
  - Yahoo! became the primary contributor in 2006
    - They have scaled it from 20 node clusters to 10,000 node+ clusters today
  - They deployed large scale science clusters in 2007
  - They began running major production jobs in Q1, 2008
- Portable
  - Written in Java
  - Runs on commodity hardware
  - Linux, Mac OS/X, Windows, and Solaris

4

Growing Hadoop Ecosystem

- **Hadoop Core**
  - Distributed File System
  - MapReduce Framework

- **Pig (initiated by Yahoo!)**
  - Parallel Programming Language and Runtime

- **Hbase (initiated by Powerset)**
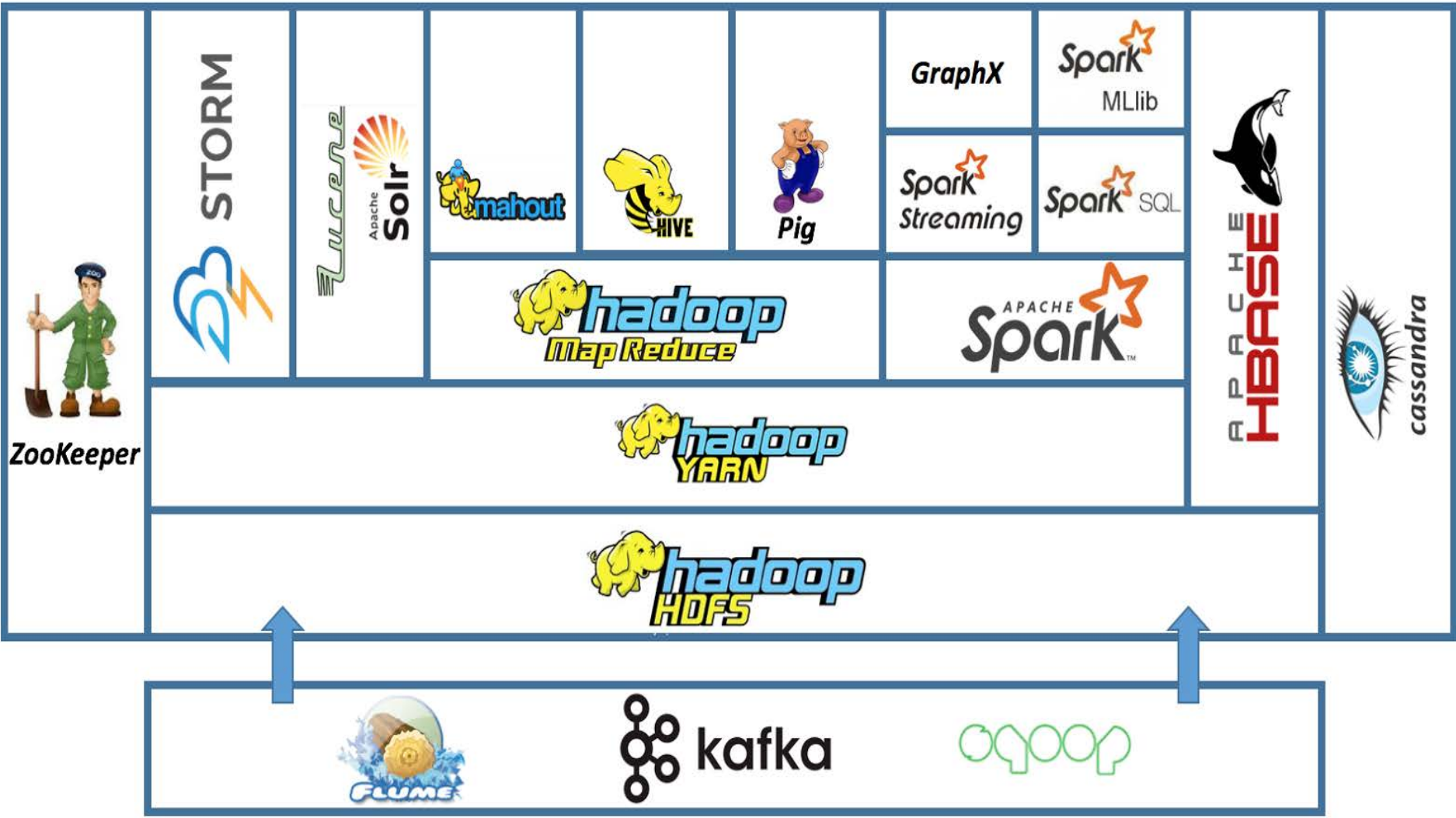 - Table storage for semi-structured data

- **Zookeeper (initiated by Yahoo!)**
  - Coordinating distributed systems

- **Hive (initiated by Facebook)**
  5 SQL-like query language and metastore

M45 (open cirrus cluster)

- Collaboration with Major Research Universities (via open cirrus)
  - Carnegie Mellon University
  - The University of California at Berkeley
  - Cornell University
  - The University of Massachusetts at Amherst joined
- Seed Facility: Datacenter in a Box (DiB)
  - 500 nodes, 4000 cores, 3TB RAM, 1.5PB disk
  - High bandwidth connection to Internet
  - Located on Yahoo! corporate campus
- Runs Hadoop
- Has been used for Ten years

Hadoop Community

Apache Hadoop Community

- Hadoop is owned by the Apache Foundation
  - Provides legal and technical framework for **collaboration**
  - All code and intellectual property (IP) owned by non-profit foundation
- Anyone can join Apache's meritocracy
  - Users
  - Contributors
    - write patches
  - Committers
    - can commit patches
  - Project Management Committee
    - vote on new committers and releases
    - represent from many organizations
- Use, contribution, and diversity are growing
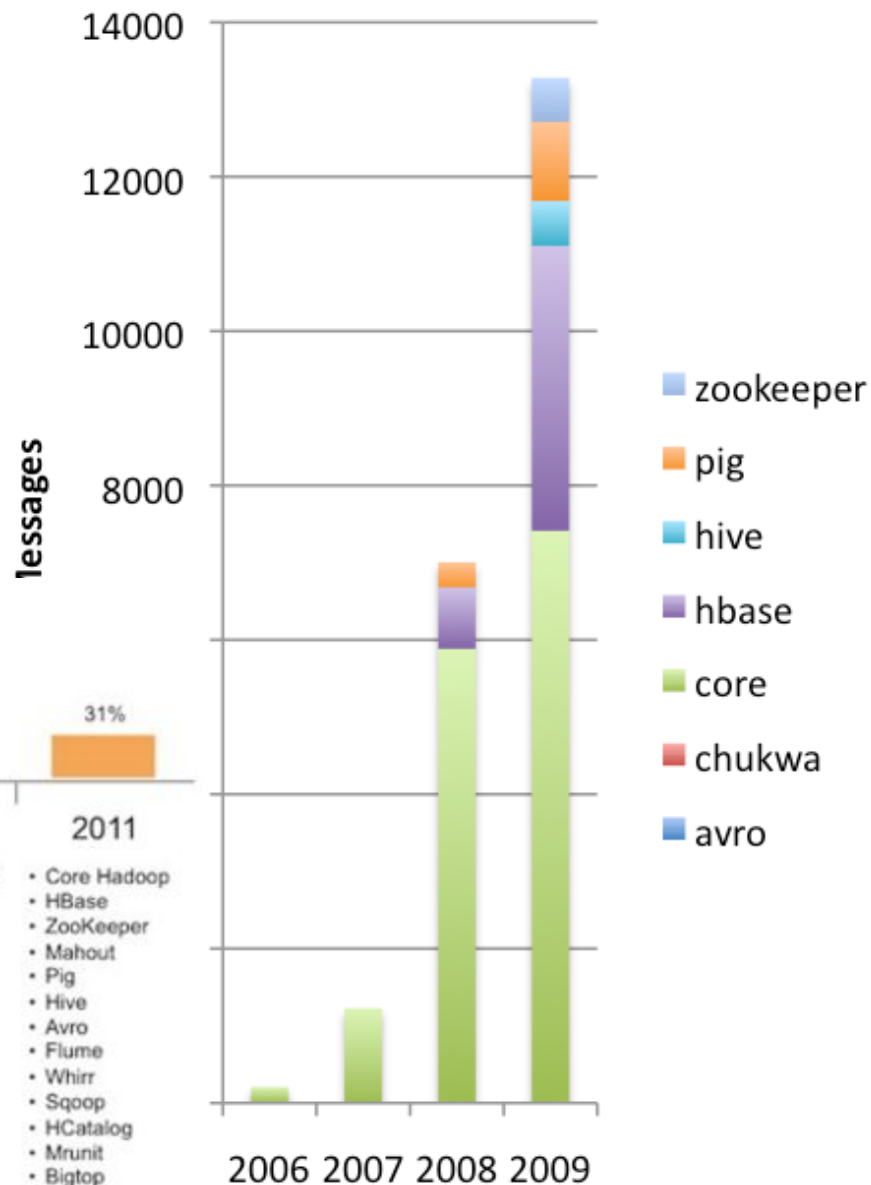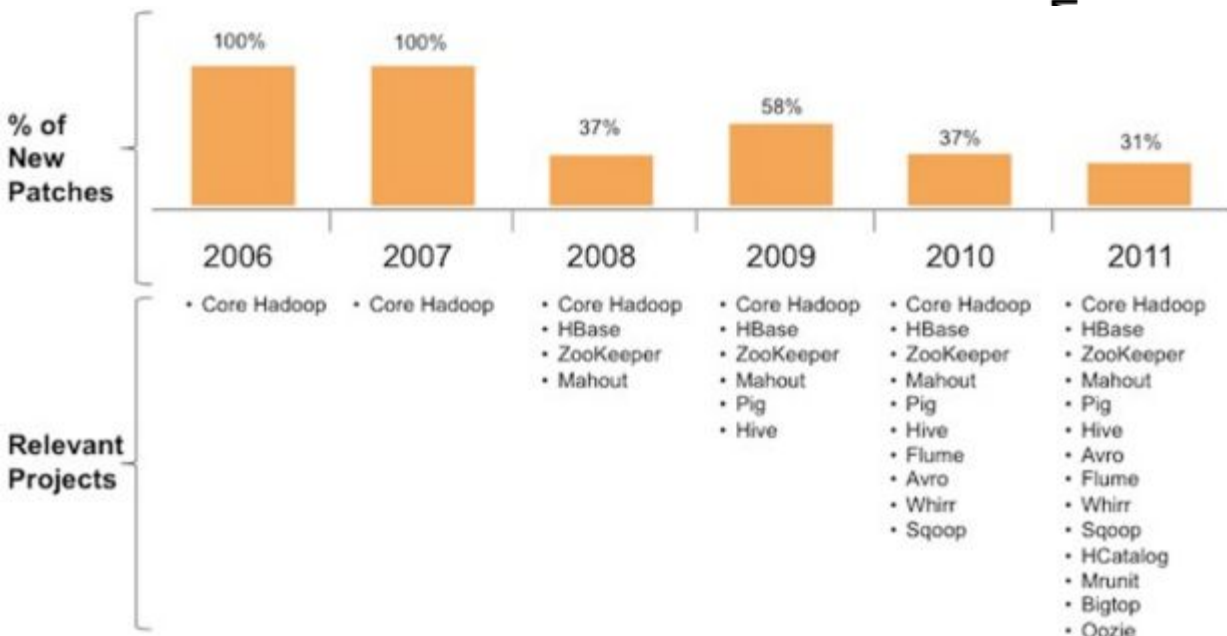 - But they need and want more!

Contributions to Hadoop

- Each contribution is a patch
- Divided by subproject
  - Core (includes HDFS and Map/Red)
  - Avro, Chukwa, HBase, Hive, Pig, and Zookeeper
- 2009 Non-Core > Core
- Core Contributors
  - 185 people (30% from Yahoo!)
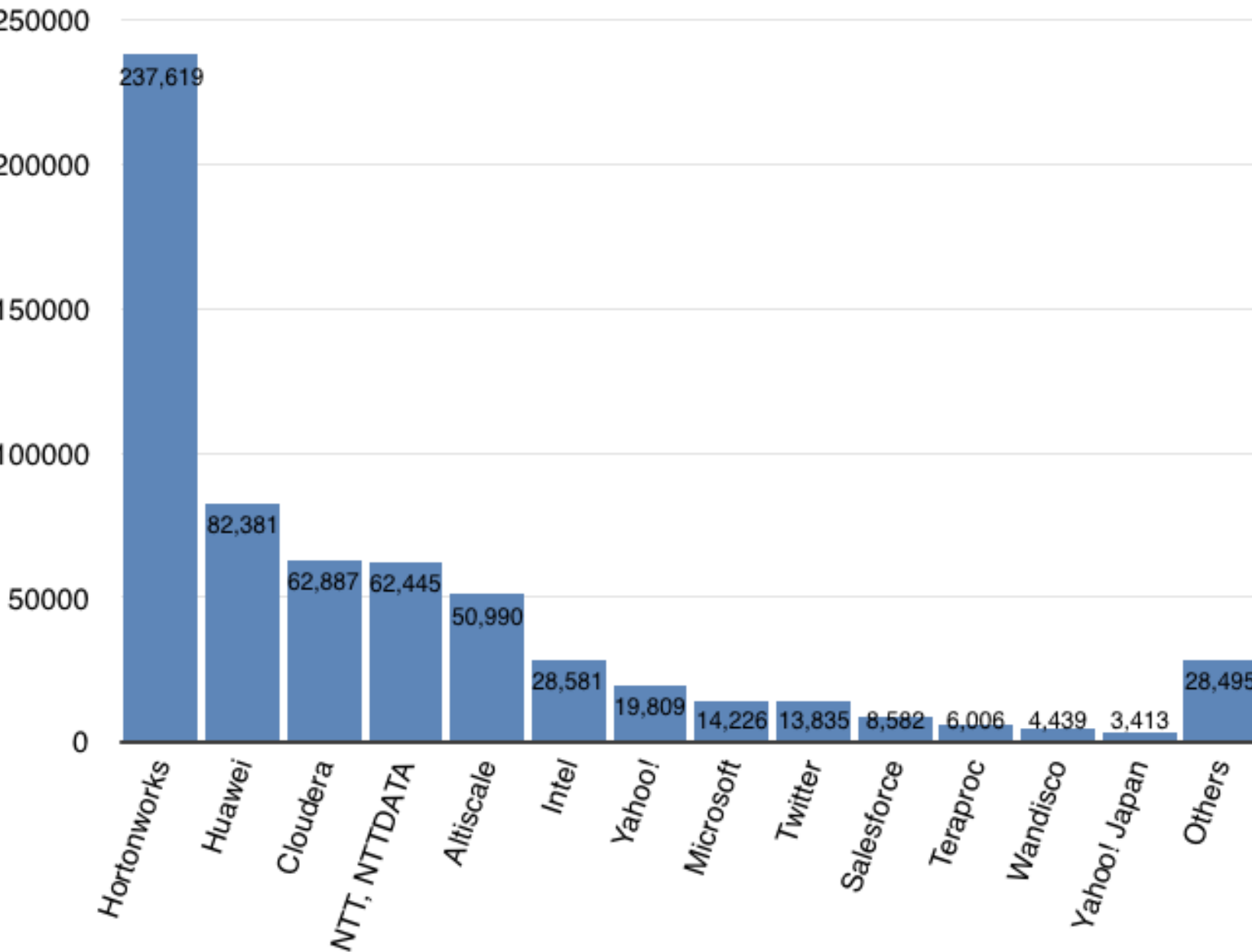 - 72% of patches from Yahoo!
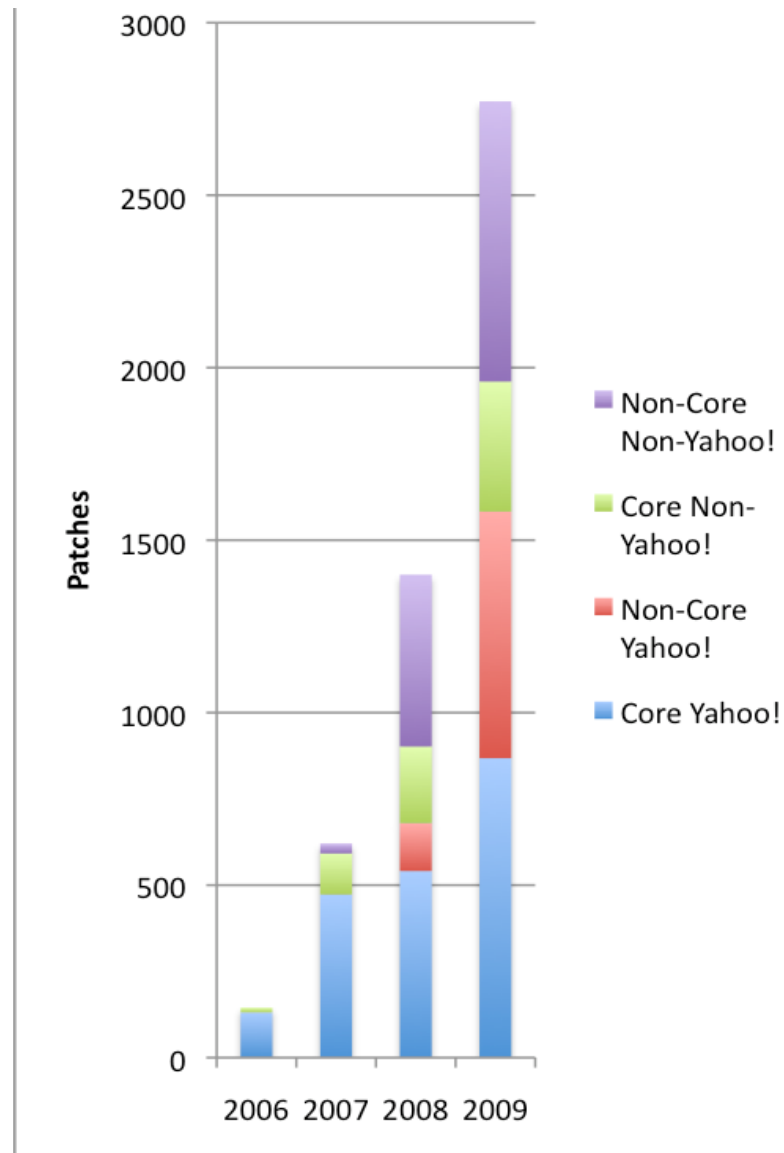
Growing Sub-Project

- User list traffic is best indicator of usage.
- Only Core, Pig, and HBase have existed > 12 months
- All sub-projects are growing

zookeeper
pig
hive
hbase
core
chukwa
avro

2006 2007 2008 2009

Number of lines of code changed in 2015

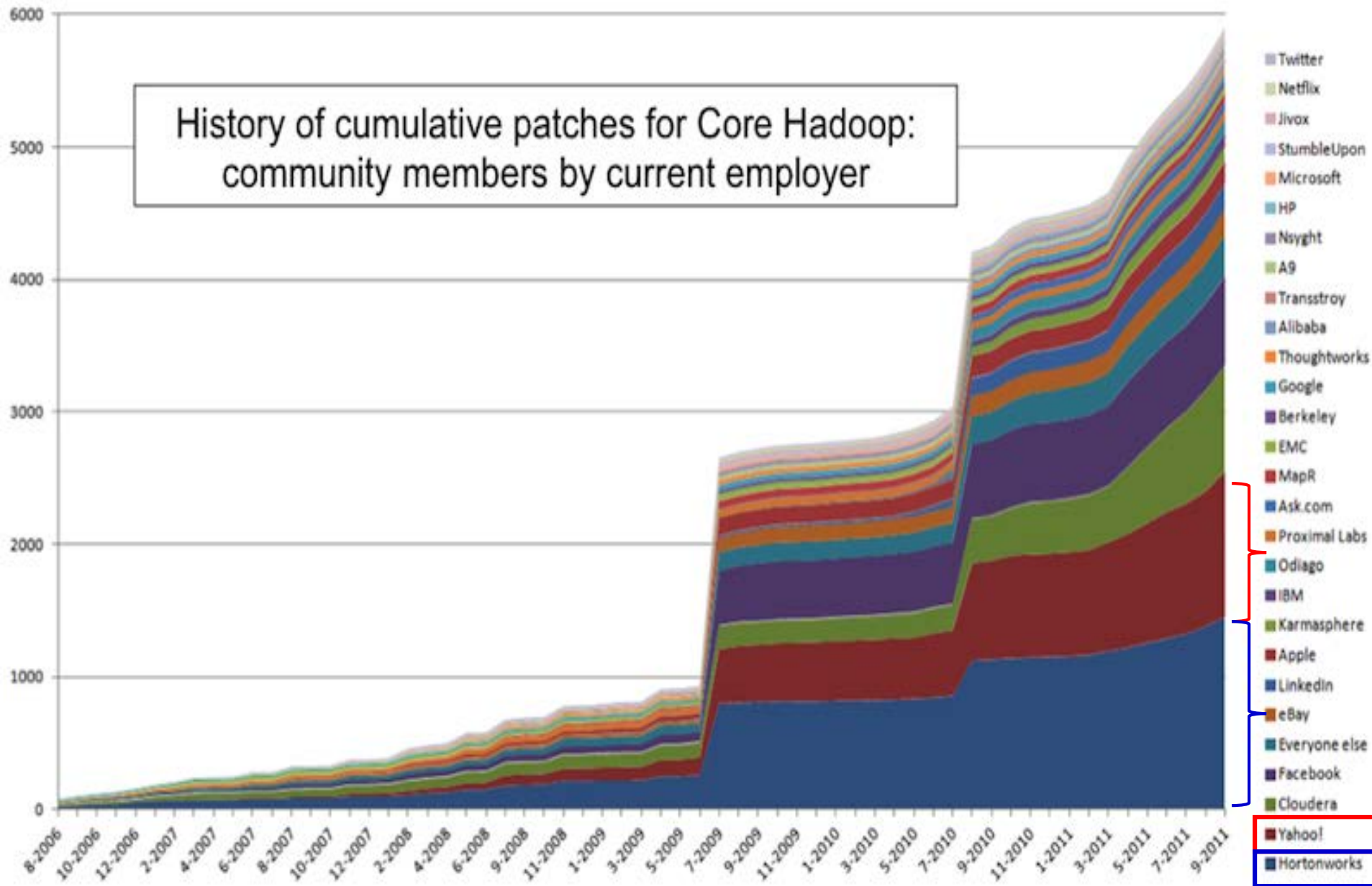| Company	Lines
Hortonworks	237,619
Huawei	82,381
Cloudera	62,887
NTT, NTTDATA	62,445
Altiscale	50,990
Intel	28,581
Yahoo!	19,809
Microsoft	14,226
Twitter	13,835
Salesforce	8,582
Teraproc	6,006
Wandisco	4,439
Yahoo! Japan	3,413
Others	28,495

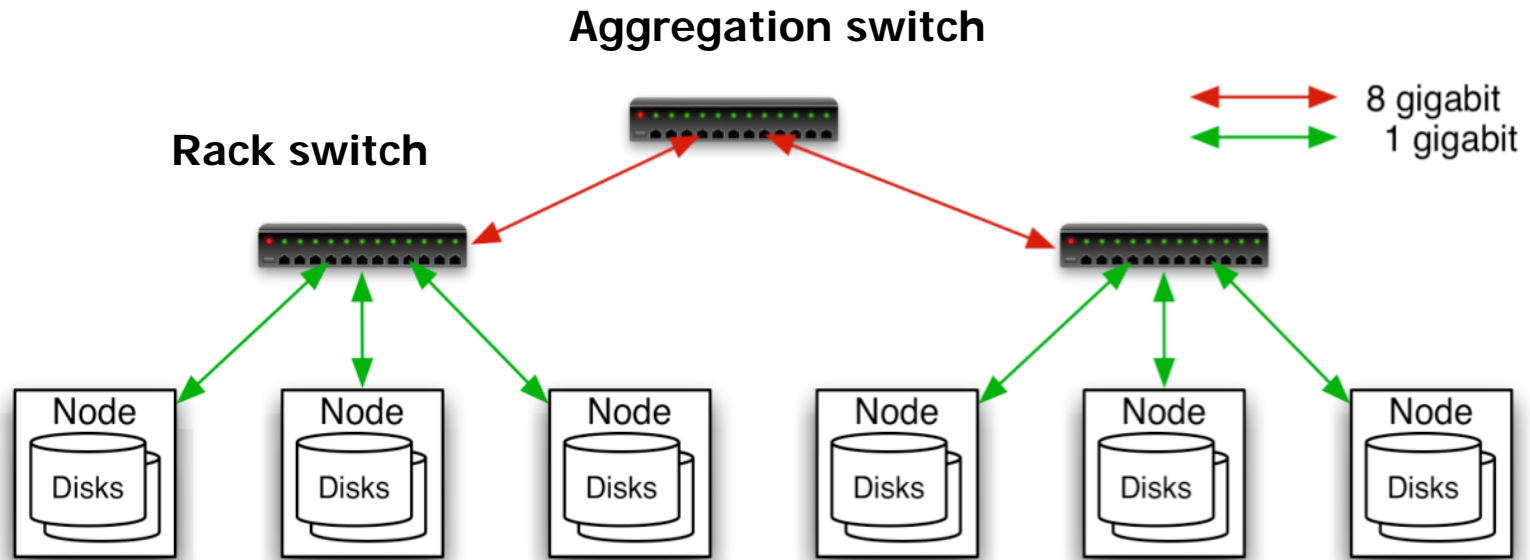Global Hadoop market is expected to reach $5.24 Billion in 2020

The Community Effect

History of cumulative patches for Core Hadoop: community members by current employer

Hadoop Architecture

Typical Hadoop Cluster (Facebook)

Aggregation switch

Rack switch

8 gigabit
1 gigabit

Node
Disks

Node
Disks

Node
Disks

Node
Disks

Node
Disks

Node
Disks

- 40 nodes/rack, 1000-4000 nodes in a cluster
- 1 Gbps bandwidth in rack, 8 Gbps out of rack
- Node specs (Facebook):
  8-16 cores, 32 GB RAM, 8×1.5 TB disks, no RAID

Typical Hadoop Cluster
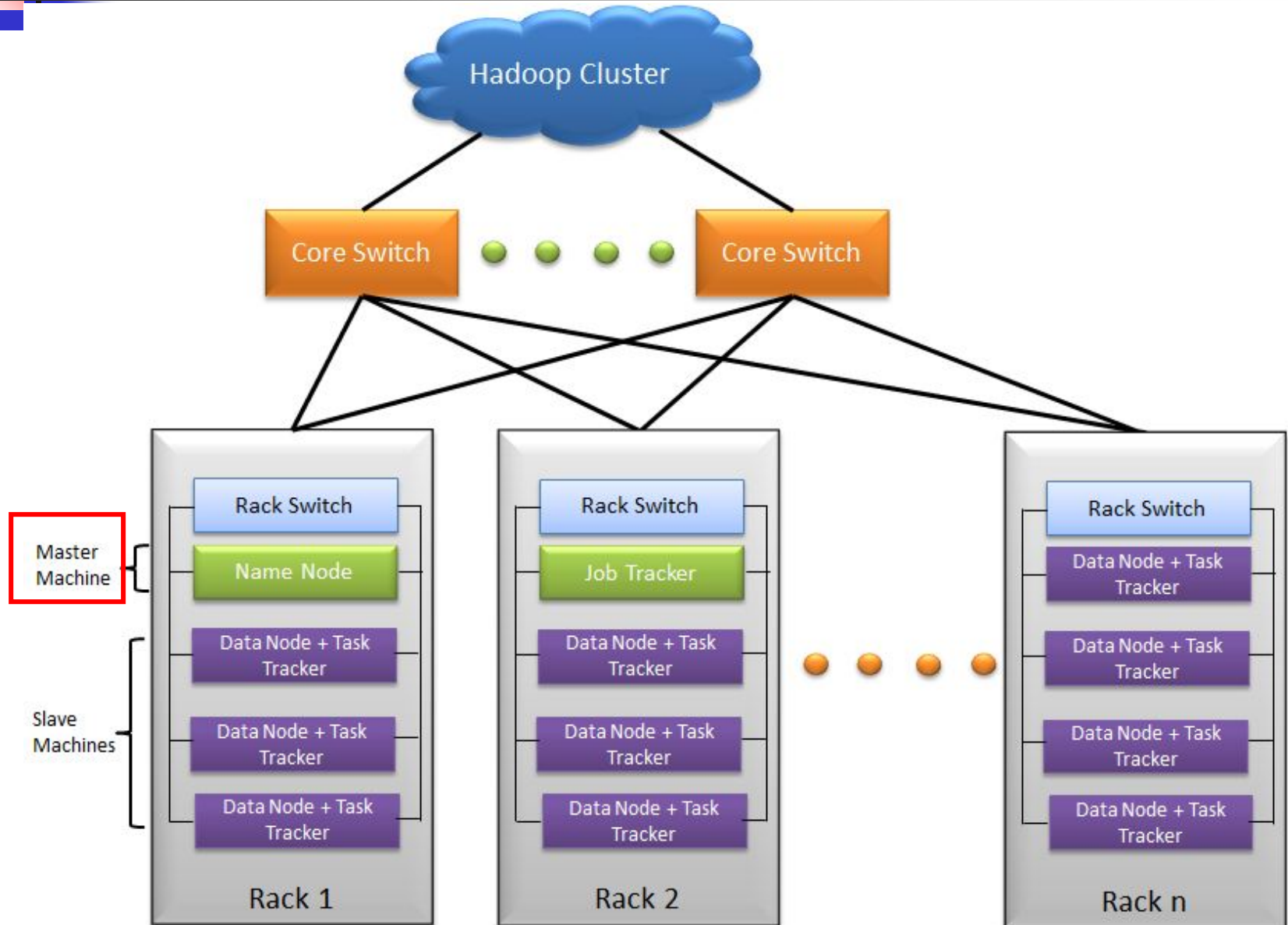
Challenges of Cloud Environment

- Cheap nodes encounter failure, especially when you have many
  - Mean time between failures for 1 node = 3 years
  - Mean time between failures for 1000 nodes = 1 day
- **Solution:** Build fault tolerance in the system

- Commodity network implies low bandwidth
- **Solution:** Effectively computer the data to save the data size

- Programming distributed system is hard
- **Solution:** Restricted programming model: users write data-parallel "map" and "reduce" functions, system handles work distribution and failures

Enter the World of Distributed Systems

- Distributed Systems/Computing
  - *Loosely coupled* set of computers, communicating through message passing, solving a common goal

- Distributed computing is *challenging*
  - Dealing with *partial failures* (examples?)
  - Dealing with *asynchrony* (examples?)

    network and computers

- Distributed Computing versus Parallel Computing?
  - distributed computing = parallel computing + partial failures

Dealing with Distribution

- We have seen several of the tools that help with distributed programming
    - Message Passing Interface (MPI)
    - Distributed Shared Memory (DSM)
    - Remote Procedure Calls (RPC)

- But, distributed programming is still very hard
    - Programming for scale, fault-tolerance, consistency, …

The Datacenter is the new Computer

The Datacenter as a Computer
An Introduction to the Design of Warehouse-Scale Machines

MORGAN & CLAYPOOL PUBLISHERS

Luiz Andre Barroso
Urs Hölzle

SYNTHESIS LECTURES ON COMPUTER ARCHITECTURE
Mark D. Hill, *Series Editor*

- *"Program"* == Web search, email, map/reduce, …

- *"Computer"* == 10,000's computers, storage, network

- Warehouse-sized facilities and workloads

- *Built from <u>less reliable components</u> than traditional datacenters*

Distributed File System

- Single petabyte file system for entire cluster
  - Managed by **a single namenode**.
  - Files are written, read, renamed, deleted, append-only.
  - Optimized for streaming reads of large files.
- Files are divided into large blocks.
  - Transfer to the client
  - CRC 32 is used in data with checksum
  - For reliability, **replicated to several datanodes**,
- Client library talks to both **namenode** and **datanodes**
  - Data is not sent through the namenode.
  - Throughput of file system scales nearly linearly.
- Access from Java, C, or command line.

Hadoop Components

- Distributed file system (HDFS)
  - Single namespace for entire cluster
  - Replicates data **3x** for fault-tolerance

- MapReduce framework
  - Runs jobs submitted by users
  - Manages work distribution & fault-tolerance
  - Collocated with file system (i.e., allocate the jobs to the file system)

Hadoop Distributed File System

- Files split into 128MB blocks
- Blocks replicated across several datanodes (often 3)
- Namenode stores metadata (file names, locations, etc)
- Optimized for large files, sequential reads
- Files are append-only

Namenode

File1

1
2
3
4

1
2
4

2
1
3

1
4
3

3
2
4

Datanodes
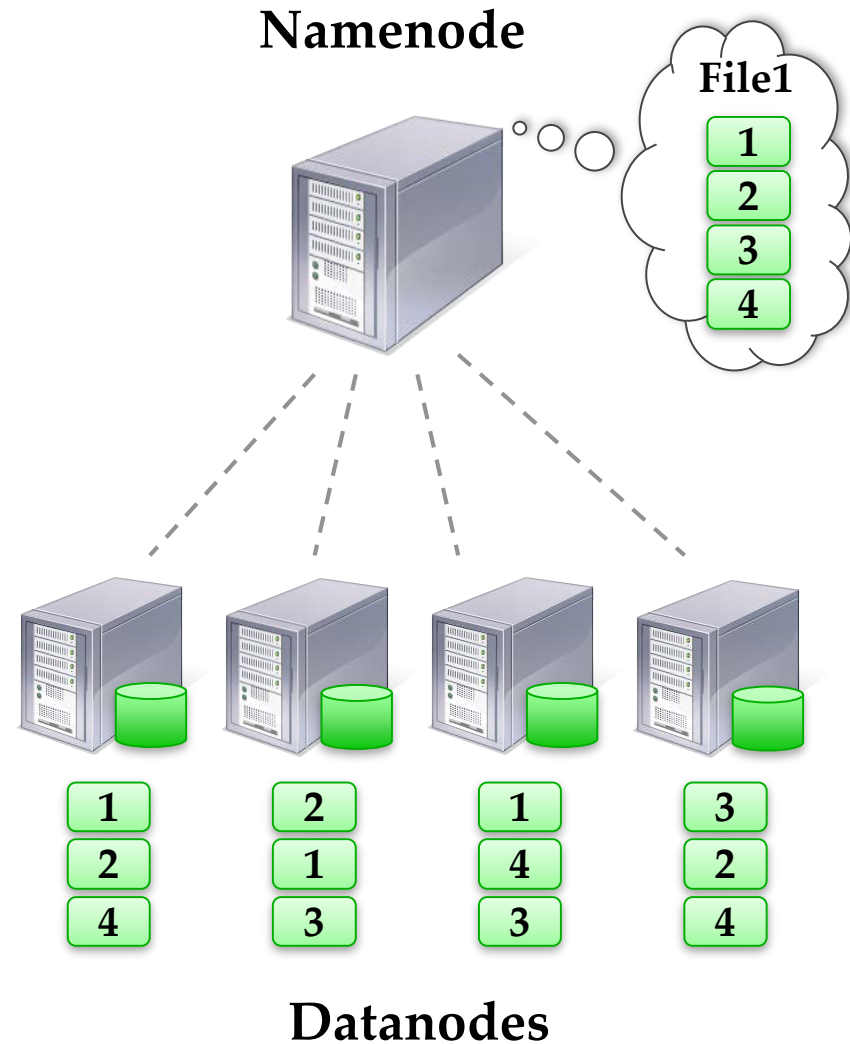
What is MapReduce?

- **MapReduce** is a programming model for processing large data sets.
  - Programming model for data-intensive computing on commodity clusters
  - MapReduce is typically used to do distributed computing on clusters of computers
- Pioneered by Google
  - Processes 20 PB of data per day
- Popularized by Apache Hadoop project
  - Used by Yahoo!, Facebook, Amazon, …

Map/Reduce features

Java, C++, and text-based APIs

- Java and C++ use object concept
- Text-based (streaming) APIs for scripting or legacy apps
- Higher level interfaces: Pig, Hive, Jaql
- Automatic re-execution on failure
  - In a large cluster, some nodes are always slow or flaky
  - Framework re-executes failed tasks
- Locality optimizations
  - For large data, bandwidth is a problem for transmission data
  - Map-Reduce queries HDFS considering locations of input data
  - Map tasks are scheduled close to the inputs as possible

MapReduce Insights

- Restricted key-value model
  - Same **fine-grained operation** (Map & Reduce) repeated on big data
  - Operations must be **deterministic**
  - Operations must be **no side effects**
  - Only communication is through the **shuffle**
    - Data from the mapper tasks is prepared and moved to the nodes where the reducer tasks will be run.
  - Operation (Map & Reduce) outputs are saved on disk.

MapReduce Insights

- The mapper is applied to every **key-value pair** in the input which is originally stored on the underlying distributed file system.

- The result of mapper is an arbitrary number of intermediate key-value pairs, and then these pairs will be sorted and grouped by the same key, finally be passed to reducer (reduce function) as input.

- Shuffle can strongly affects the efficiency of MapReduce tasks.

shuffle

Who Use MapReduce?

Industry
- Google:
  - Index building for Google Search
  - Article clustering for Google News
  - Statistical machine translation
- Yahoo!:
  - Index building for Yahoo! Search
  - Spam detection for Yahoo! Mail
- Facebook:
  - Data mining
  - Advertising optimization
  - Spam detection

Who Use MapReduce?

Academic

- For research:
  - Analyzing Wikipedia conflicts (PARC)
  - Natural language processing (CMU)
  - Climate simulation (Washington)
  - Bioinformatics (Maryland)
  - Particle physics (Nebraska)
  - …

Google Cloud Infrastructure

- Google File System (GFS), 2003
  - Distributed File System for entire cluster
  - Single namespace

- Google MapReduce (MR), 2004
  - Runs queries/jobs on data
  - Manages work distribution & fault-tolerance
  - Colocated with file system

- Apache open source versions Hadoop DFS and Hadoop MR

GFS/HDFS Insights

- **Petabyte** storage
  - Files split into large blocks (128 MB) and replicated across several nodes
  - Big blocks allow high throughput sequential reads/writes
- Use **commodity** hardware
  - Failures are the norm anyway because buy cheaper hardware
- No complicated consistency models
  - Single writer, append-only data

MapReduce Pros

- Distribution is completely **transparent**
  - Not a single line of distributed programming (ease, correctness)
- Automatic **fault-tolerance**
  - Determinism enables running failed tasks somewhere else again
  - Saved intermediate data enables just re-running failed reducers
- Automatic **scaling**
  - As operations as side-effect free, they can be distributed to any number of machines dynamically
- Automatic **load-balancing**
  - Move tasks and speculatively execute duplicate copies of slow tasks (*stragglers*)

MapReduce Cons

- Restricted programming model
  - Not always natural to express problems in this model
  - Low-level coding necessary
  - Little support for iterative jobs (lots of disk access)
  - High-latency (batch processing)
- Addressed by follow-up research
  - **Pig** and **Hive** for high-level coding
  - **Spark** for iterative and low-latency jobs

HiveQL (~SQL)   PigLatin

Hive   Pig

MR

HDFS
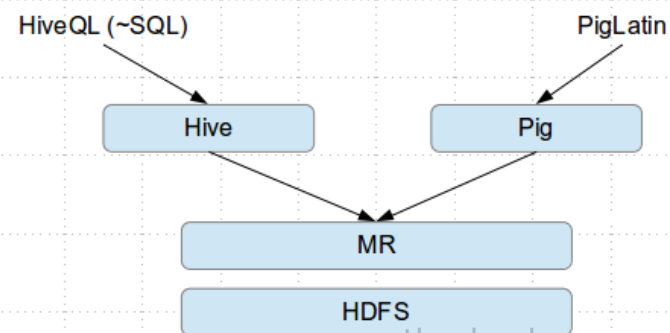
MapReduce Goals

- <u>Scalability</u> process large data volumes:
  - Scan 100 TB on 1 node at 50 MB/s = 24 days
  - Using 1000-node cluster to scan = 35 minutes

- Cost-efficiency:
  - Commodity nodes (cheap, but unreliable)
  - Commodity network (low bandwidth)
  - Automatic fault-tolerance (fewer administration)
  - Easy to use (fewer programmers)

MapReduce Programming Model

- Data type: key-value records

- Map function:

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

- Reduce function:

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

Hadoop Programming – Map/Reduce
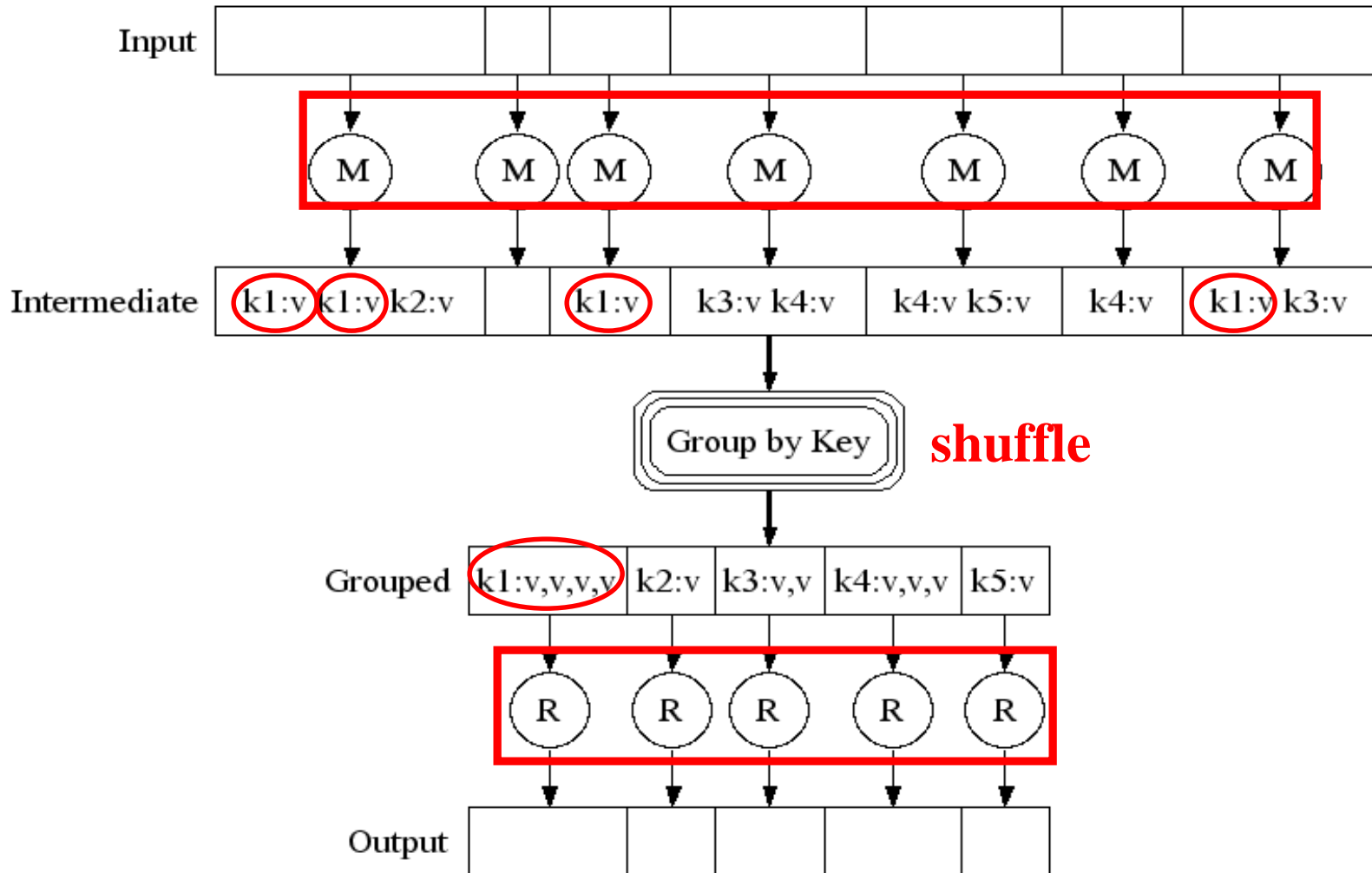
Map / Reduce
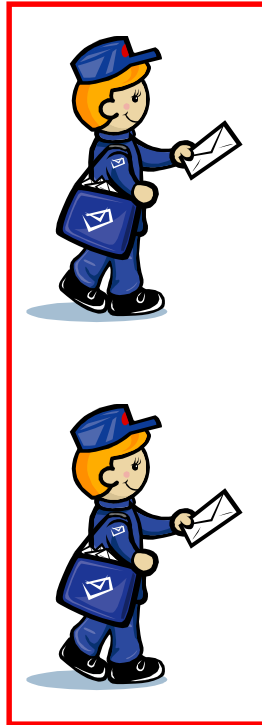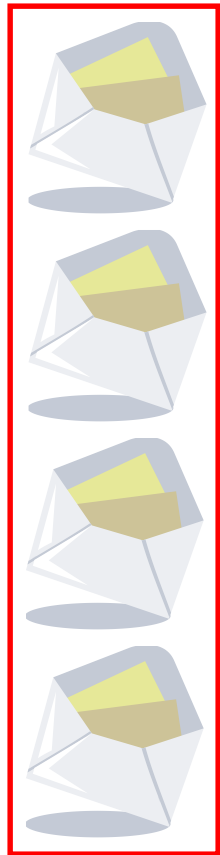
mappers
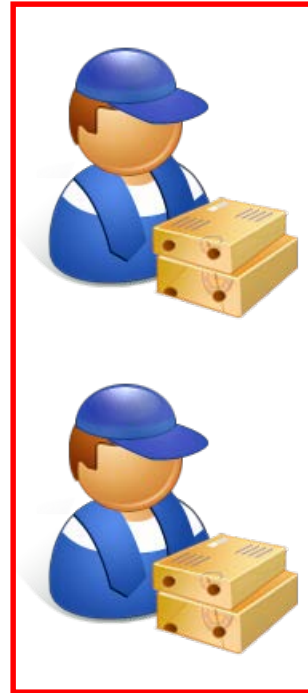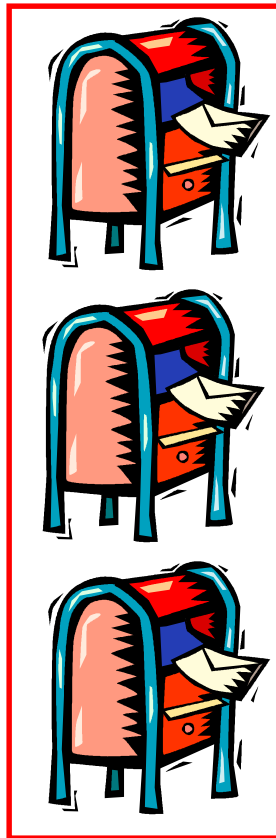
reducers

100
.
.
3
7

220
.
.
2
8

Example: Word Count (Python)

```python
def mapper(line):
    foreach word in line.split():
        output(word, 1)


def reducer(key, values):
    output(key, sum(values))
```

Word Count Execution

Input	Map	Shuffle & Sort	Reduce	Output

Input:
- the quick brown fox
- the fox ate the mouse
- how now brown cow

Map (box)

the, 1
brown, 1
fox, 1

quick, 1

the, 1
fox, 1
the, 1

Map (box)

how, 1
now, 1
brown, 1

ate, 1
mouse, 1

Map (box)

cow, 1

Reduce (box)

Reduce (box)

Output:

brown, 2
fox, 2
how, 1
now, 1
the, 3

ate, 1
cow, 1
mouse, 1
quick, 1
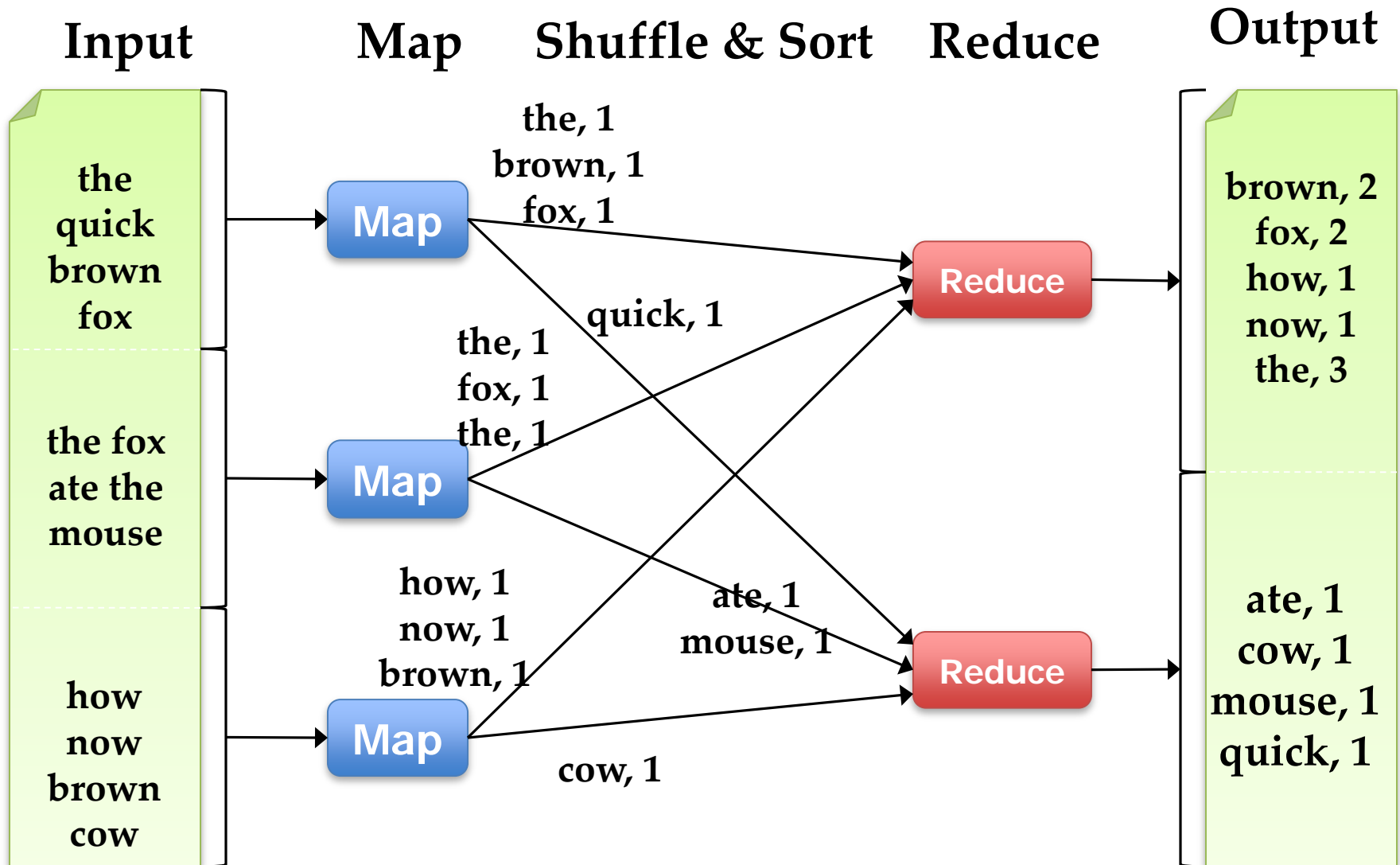
An Optimization using the Combiner

- Local reduce function for repeated keys produced by same map
- For associative options like sum, count, max.
- Decreases amount of intermediate data
- Example: local counting for Word Count:

```
def combiner(key, values):
    output(key, sum(values))
```

Word Count with Combiner

Input	Map	Shuffle & Sort	Reduce	Output

Input
the
quick
brown
fox

the fox
ate the
mouse

how
now
brown
cow

Map
Map

the, 2
fox, 1

Map

Map

Shuffle & Sort
the, 1
brown, 1
fox, 1

quick, 1

how, 1
now, 1
brown, 1

ate, 1
mouse, 1

cow, 1

Reduce
Reduce

Reduce

Output
brown, 2
fox, 2
how, 1
now, 1
the, 3

ate, 1
cow, 1
mouse, 1
quick, 1
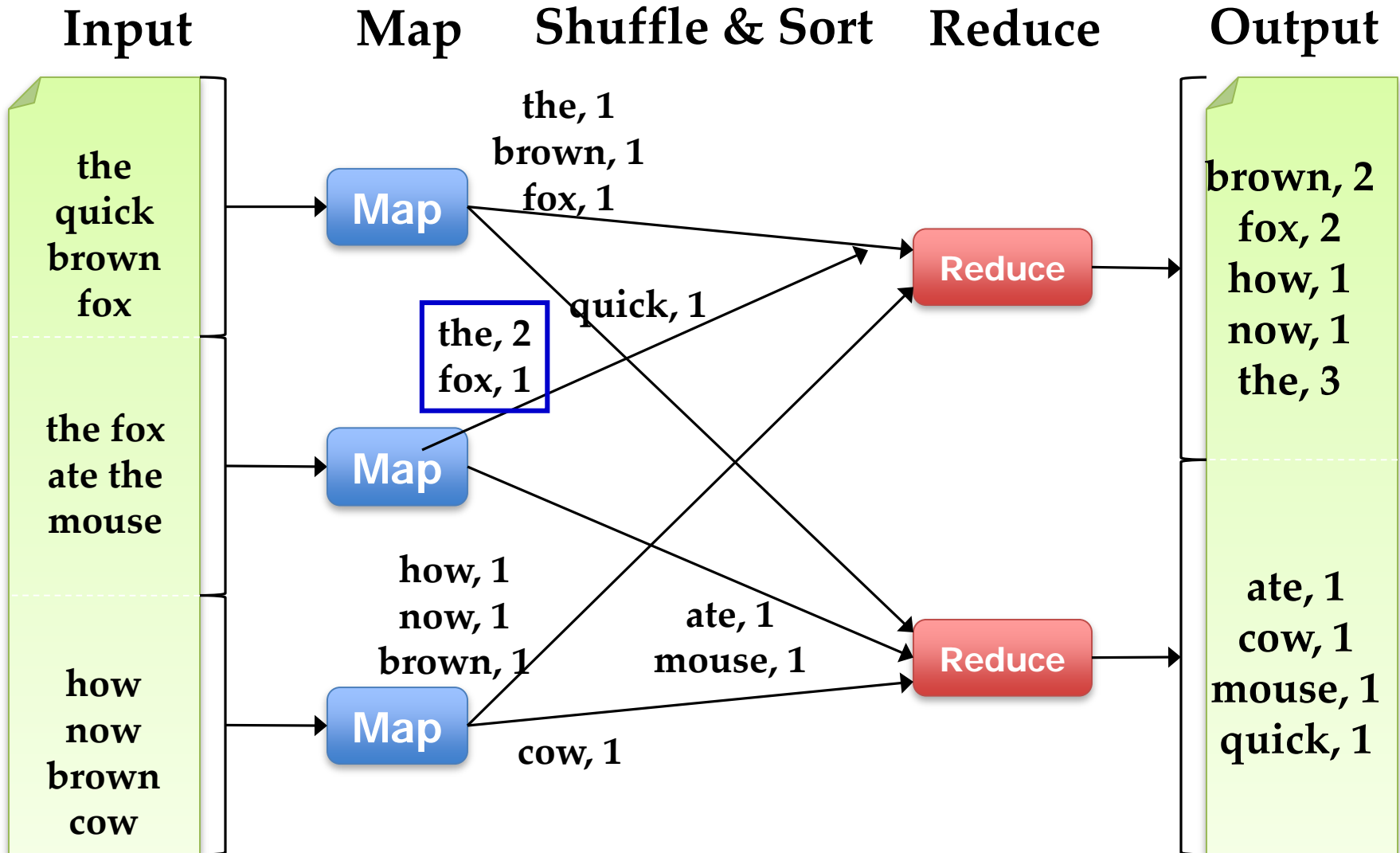
MapReduce Execution Details

- Mappers preferentially scheduled on same node or same rack as their input block
  - Minimize bandwidth use to improve performance

- Mappers <u>save outputs to local disk</u> before serving to reducers
  - Allows recovery if a reducer crashes
  - Allows running more reducers than number of nodes

Fault Tolerance in MapReduce

1. If a **task** crashes:
   - Retry on another node
     - Find for the map because it had no dependencies
     - Find for the reduce because outputs of map are on disk
   - If the same task repeatedly fails, fail for the job or ignore that input block

➢ **Note: For the fault tolerance in work, user tasks must be deterministic and side-effect-free**

Fault Tolerance in MapReduce

2. If a **node** crashes:
   - Relaunch its current tasks on other nodes
   - Relaunch any maps in which the node previously ran
     - Note that their output files were lost along with the crashed node

3. If a **task** is going slowly (straggler):
   - Launch second copy of the task on another node
   - Take the output of whichever copy finishes first, and kill the other one
   - This action is critical for performance in large clusters (many possible causes of stragglers)

Some issues

- By providing a restricted data-parallel programming model, MapReduce can control job execution in useful ways:
  - Automatic division of job into tasks
  - Be placed near data for computing
  - Load balancing
  - Recovery from failures & stragglers

Outline

- MapReduce architecture
- Sample applications
- Introduction to Hadoop
- Higher-level query languages: Pig & Hive
- Current research

1. Search

- **Input:** (lineNumber, line) records
- **Output:** lines matching a given pattern

- **Map:**
  ```
  if(line matches pattern):
      output(line)
  ```

- **Reduce:** identity function
  - Alternative: no reducer (map-only job)

2. Sort

- **Input:** (key, value) records
- **Output:** same records and sorted by key

- **Map:** identity function
- **Reduce:** identify function

- **Trick:** Pick <u>partitioning function</u> p such that
  $$k_1 < k_2 => p(k_1) < p(k_2)$$

Map — ant, bee → Reduce [A-M]
Map — zebra
cow
Map — pig
aardvark, elephant
Map — sheep, yak → Reduce [N-Z]

Reduce [A-M]:
aardvark
ant
bee
cow
elephant

Reduce [N-Z]:
pig
sheep
yak
zebra
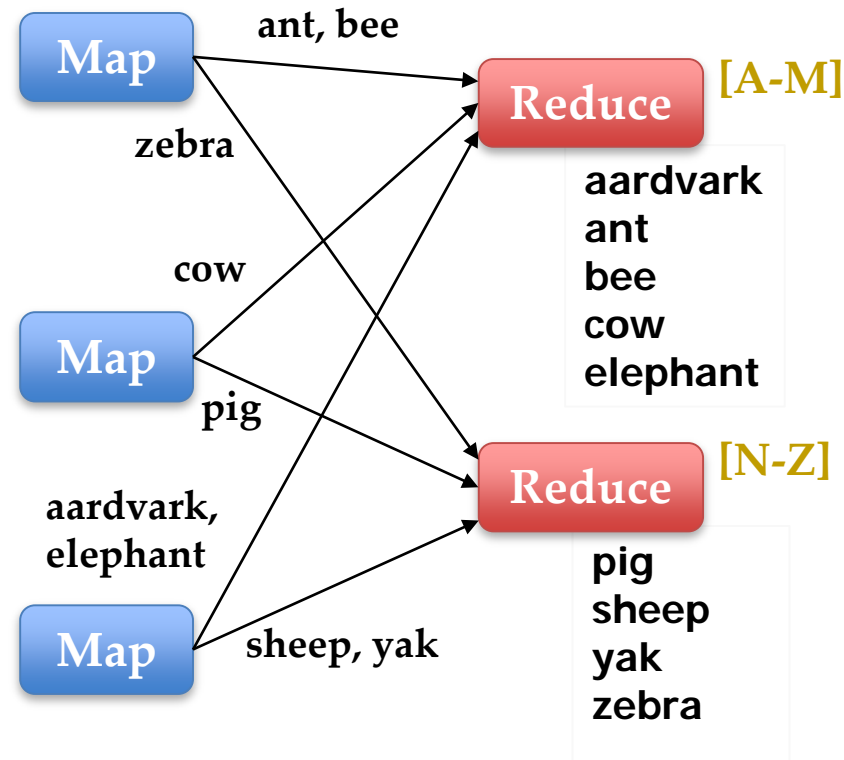
3. Inverted Index

- **Input:** (filename, text) records
- **Output:** list of files containing each word

- **Map:**
  ```
  foreach word in text.split():
      output(word, filename)
  ```

- **Combine:** unique filenames for each word

- **Reduce:**
  ```
  def reduce(word, filenames):
      output(word, sort(filenames))
  ```

Inverted Index Example

hamlet.txt

to be or
not to be

→ to, hamlet.txt
be, hamlet.txt
or, hamlet.txt
not, hamlet.txt

12th.txt

be not
afraid of
greatness

→ be, 12th.txt
not, 12th.txt
afraid, 12th.txt
of, 12th.txt
greatness, 12th.txt

afraid, (12th.txt)
be, (12th.txt, hamlet.txt)
greatness, (12th.txt)
not, (12th.txt, hamlet.txt)
of, (12th.txt)
or, (hamlet.txt)
to, (hamlet.txt)
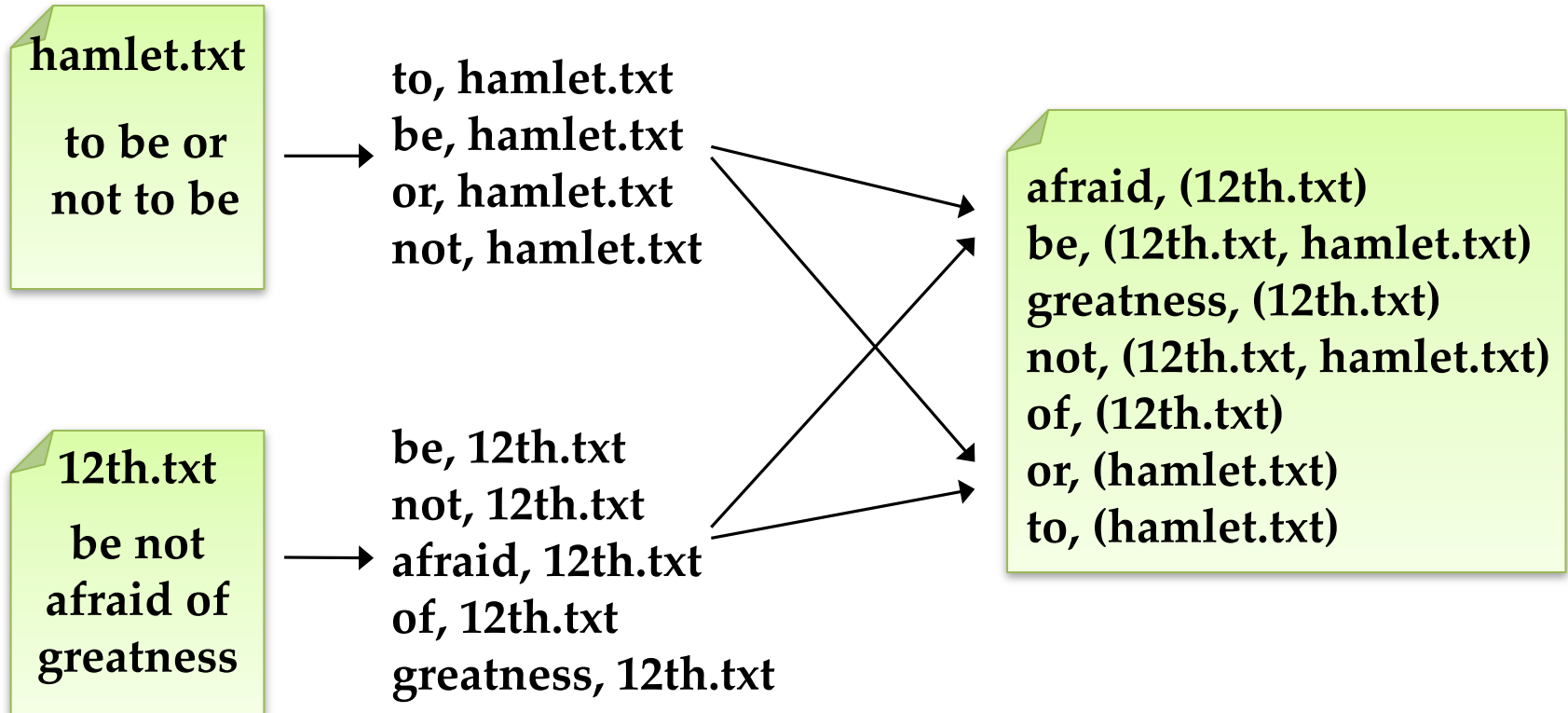
4. Most Popular Words

- **Input:** (filename, text) records
- **Output:** the 100 words occurring in most files

- Two-stage solution:
  - **Job 1:**
    - Create inverted index, giving (word, list(file)) records
  - **Job 2:**
    - Map each (word, list(file)) to (count, word)
    - Sort these records by count as in sort job

- Optimizations:
  - Map to (word, 1) instead of (word, file) in Job 1
  - Estimate count distribution in advance by sampling

5. Numerical Integration (積分)

- **Input:** (start, end) records for sub-ranges to integrate
  - Can implement using custom InputFormat
- **Output:** integral of **f(x)** over entire range

- **Map:**
```
def map(start, end):
 sum = 0
 for(x = start; x < end; x += step):
     sum += f(x) * step
 output("", sum)
```

- **Reduce:**
```
def reduce(key, values):
 output(key, sum(values))
```

Outline

- MapReduce architecture
- Sample applications
- Introduction to Hadoop
- Higher-level query languages: Pig & Hive
- Current research

Introduction to Hadoop

- Download from hadoop.apache.org
- To install locally, unzip and set `JAVA_HOME`
- Docs: hadoop.apache.org/common/docs/current

- Three ways to write jobs:
  - Java API
  - Hadoop Streaming (for Python, Perl, etc)
  - Pipes API (C++)

Word Count Using Map in Java

```java
public static class MapClass extends MapReduceBase
    implements Mapper<LongWritable, Text, Text, IntWritable> {

    private final static IntWritable ONE = new IntWritable(1);

    public void map(LongWritable key, Text value,
                    OutputCollector<Text, IntWritable> output,
                    Reporter reporter) throws IOException {
      String line = value.toString();
      StringTokenizer itr = new StringTokenizer(line);
      while (itr.hasMoreTokens()) {
        output.collect(new Text(itr.nextToken()), ONE);
      }
    }
}
```

Word Count Using Reduce in Java

```java
public static class Reduce extends MapReduceBase
    implements Reducer<Text, IntWritable, Text, IntWritable> {

    public void reduce(Text key, Iterator<IntWritable> values,
                       OutputCollector<Text, IntWritable> output,
                       Reporter reporter) throws IOException {
        int sum = 0;
        while (values.hasNext()) {
            sum += values.next().get();
        }
        output.collect(key, new IntWritable(sum));
    }
}
```

Word Count (Main Function)
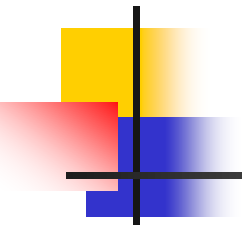
```java
public static void main(String[] args) throws Exception {
    JobConf conf = new JobConf(WordCount.class);
    conf.setJobName("wordcount");

    conf.setMapperClass(MapClass.class);
    conf.setCombinerClass(Reduce.class);
    conf.setReducerClass(Reduce.class);

    FileInputFormat.setInputPaths(conf, args[0]);
    FileOutputFormat.setOutputPath(conf, new Path(args[1]));

    conf.setOutputKeyClass(Text.class); // out keys are words (strings)
    conf.setOutputValueClass(IntWritable.class); // values are counts

    JobClient.runJob(conf);
}
```

```java
import java.io.IOException;
import java.util.StringTokenizer;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class HadoopWordCountSample {
    public static class WordCountMapper
            extends Mapper<Object, Text, Text, IntWritable>{

        //計數使用，設定為1。每當找到相同的字就會+1。
        private final static IntWritable plugOne  = new IntWritable(1);
        private Text word = new Text();

        @Override
        public void map(Object key, Text value, Context context
        ) throws IOException, InterruptedException {
            //使用StringTokenizer效能會比使用split好。預設使用空白，tab或是換行當
作分隔符號。
            StringTokenizer st = new StringTokenizer(value.toString());
            while (st.hasMoreTokens()) {
                word.set(st.nextToken());
                context.write(word, plugOne);
            }
        }
    }
}
```

64

```java
public static class WordCountReducer
        extends Reducer<Text,IntWritable,Text,IntWritable> {

    private IntWritable result = new IntWritable();

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

public static void main(String[] args) throws Exception {
    Configuration config = new Configuration();
    Job job = Job.getInstance(config, "hadoop word count example");
    job.setJarByClass(HadoopWordCountSample.class);
    job.setReducerClass(WordCountReducer.class);
    job.setMapperClass(WordCountMapper.class);
    //設定setCombinerClass後，每個mapper會在sorting後，對結果先做一次reduce
    job.setCombinerClass(WordCountReducer.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    //執行程式時，第一個參數（agrs[0]）為欲計算檔案路徑
    FileInputFormat.addInputPath(job, new Path(args[0]));
    //第二個參數（agrs[1]）為計算結果存放路徑
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
    System.exit(job.waitForCompletion(true) ? 0 : 1);
}
}
```
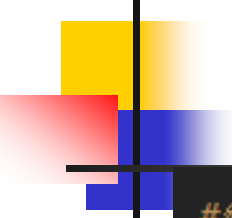
65

```
#編輯~/.bashrc檔案
sudo vi ~/.bashrc
#加入HADOOP_CLASSPATH。注意！記得放在JAVA_HOME後面得行數，否則會出錯。
export HADOOP_CLASSPATH=${JAVA_HOME}/lib/tools.jar
```

```
#Compile
hadoop com.sun.tools.javac.Main HadoopWordCountSample.java
#Package
jar cf hwcs.jar HadoopWordCountSample*.class
```

- wordcount_target1

```
I am Jack
I am the king of the world
```

- wordcount_target2

```
I am Rose
I am looking for Jack
```

```
#建立資料夾
hadoop fs -mkdir -p /tmp/wordcount_target
#上傳
hadoop fs -put wordcount_target1 wordcount_target2 /tmp/wordcount_target
t
```

```
hadoop jar hwcs.jar HadoopWordCountSample /tmp/wordcount_target /tmp/wordcount
_result
#說明
hwcs.jar:                要執行計算的MapReduce jar檔案路徑（需要包含的檔案名稱）
HadoopWordCountSample: 欲執行的class name
/tmp/wordcount_target: 欲計算檔案路徑
/tmp/wordcount_result: 計算結果存放路徑
```

```
hadoop fs -cat /tmp/wordcount_result/part-r-00000
#結果
am 4
for 1
king 1
looking 1
of 1
the 2
world 1
I 4
Jack 2
Rose 1
```

Word Count in Python

- Mapper.py

mapper.py

```python
#!/usr/bin/env python

import sys

# input comes from STDIN (standard input)
for line in sys.stdin:
    # remove leading and trailing whitespace
    line = line.strip()
    # split the line into words
    words = line.split()
    # increase counters
    for word in words:
        # write the results to STDOUT (standard output);
        # what we output here will be the input for the
        # Reduce step, i.e. the input for reducer.py
        #
        # tab-delimited; the trivial word count is 1
        print '%s\t%s' % (word, 1)</pre>
```

68

```python
#!/usr/bin/env python

from operator import itemgetter
import sys

current_word = None
current_count = 0
word = None

# input comes from STDIN
for line in sys.stdin:
    # remove leading and trailing whitespace
    line = line.strip()

    # parse the input we got from mapper.py
    word, count = line.split('\t', 1)

    # convert count (currently a string) to int
    try:
        count = int(count)
    except ValueError:
        # count was not a number, so silently
        # ignore/discard this line
        continue

    # this IF-switch only works because Hadoop sorts map output
    # by key (here: word) before it is passed to the reducer
    if current_word == word:
        current_count += count
    else:
        if current_word:
            # write result to STDOUT
            print '%s\t%s' % (current_word, current_count)
        current_count = count
        current_word = word

# do not forget to output the last word if needed!
if current_word == word:
    print '%s\t%s' % (current_word, current_count)
```

Results

```
# very basic test
hduser@ubuntu:~$ echo "foo foo quux labs foo bar quux" | /home/hduser/mapper.py
foo     1
foo     1
quux    1
labs    1
foo     1
bar     1
quux    1

hduser@ubuntu:~$ echo "foo foo quux labs foo bar quux" | /home/hduser/mapper.py | sort -k1,1 | /home/hduser/reducer.py
bar     1
foo     3
labs    1
quux    2
```

HDFS Running

- hduser@ubuntu:/usr/local/hadoop$ bin/hadoop jar contrib/streaming/hadoop-*streaming*.jar -mapper /home/hduser/mapper.py -reducer /home/hduser/reducer.py -input /user/hduser/gutenberg/* -output /user/hduser/gutenberg-output
additionalConfSpec_:null
null=@@@userJobConfProps_.get(stream.shipped.hadoopstreaming
packageJobJar: [/app/hadoop/tmp/hadoop-unjar54543/]
[] /tmp/streamjob54544.jar tmpDir=null
[…] INFO mapred.FileInputFormat: Total input paths to process : 7
[…] INFO streaming.StreamJob: getLocalDirs(): [/app/hadoop/tmp/mapred/local]
[…] INFO streaming.StreamJob: Running job: job_201510011615_0021
[…]
[…] INFO streaming.StreamJob: map 0% reduce 0%
[…] INFO streaming.StreamJob: map 43% reduce 0%
[…] INFO streaming.StreamJob: map 86% reduce 0%
[…] INFO streaming.StreamJob: map 100% reduce 0%
[…] INFO streaming.StreamJob: map 100% reduce 33%
[…] INFO streaming.StreamJob: map 100% reduce 70%
[…] INFO streaming.StreamJob: map 100% reduce 77%
[…] INFO streaming.StreamJob: map 100% reduce 100%
[…] INFO streaming.StreamJob: Job complete: job_201510011615_0021
[…] INFO streaming.StreamJob: Output: /user/hduser/gutenberg-output
hduser@ubuntu:/usr/local/hadoop$

Results

- hduser@ubuntu:/usr/local/hadoop$ bin/hadoop dfs -ls /user/hduser/gutenberg-output
  Found 1 items
  /user/hduser/gutenberg-output/part-00000   <r 1>   903193 2016-09-20 13:00
  hduser@ubuntu:/usr/local/hadoop$
  hduser@ubuntu:/usr/local/hadoop$ bin/hadoop dfs -cat /user/hduser/gutenberg-output/part-00000
  "(Lo)cra"   1
  "1490   1
  "1498," 1
  "35"   1
  "40,"   1
  "A   2
  "AS-IS".   2
  "A_   1
  "Absoluti   1
  […]
  hduser@ubuntu:/usr/local/hadoop$

Outline

- MapReduce architecture
- Sample applications
- Introduction to Hadoop
- Higher-level query languages: Pig & Hive
- Current research

Motivation

- MapReduce is powerful: many algorithms can be expressed as a series of MapReduce jobs

- But it's <u>fairly low-level</u>: must think about keys, values, partitioning, etc.

- Can we capture common "job patterns"?

Pig

- **Apache Pig** is a high-level platform for creating programs that run on Apache Hadoop.
  - The language is called **Pig Latin**.

- Pig can execute its Hadoop jobs in MapReduce, Apache Tez, or Apache Spark.

- Pig Latin abstracts the programming from the Java MapReduce idiom into a notation which makes MapReduce programming high level, similar to that of SQL for relational database management systems.

Pig

- Pig Latin can be extended using <u>user-defined functions</u> (UDFs) which the user can write in Java, <u>Python</u>, <u>JavaScript</u>, <u>Ruby</u> or <u>Groovy</u> and then call directly from the language.

Pig

- Started at Yahoo's research

- Runs about 50% of Yahoo!'s jobs

- Features:
  - Expresses sequences of MapReduce jobs
  - Data model: nested "bags" of items
  - Provides relational (SQL) operators (JOIN, GROUP BY, etc.)
  - Easy to plug in Java functions

Pig script → MapReduce

- Do not understand below MapReduce operations
- Pig transfers logical plan to MR plan

(SQL)

automatic rewrite + optimize

Pig

Logical Plan

Physical Plan

MR Plan

Map-Reduce

cluster
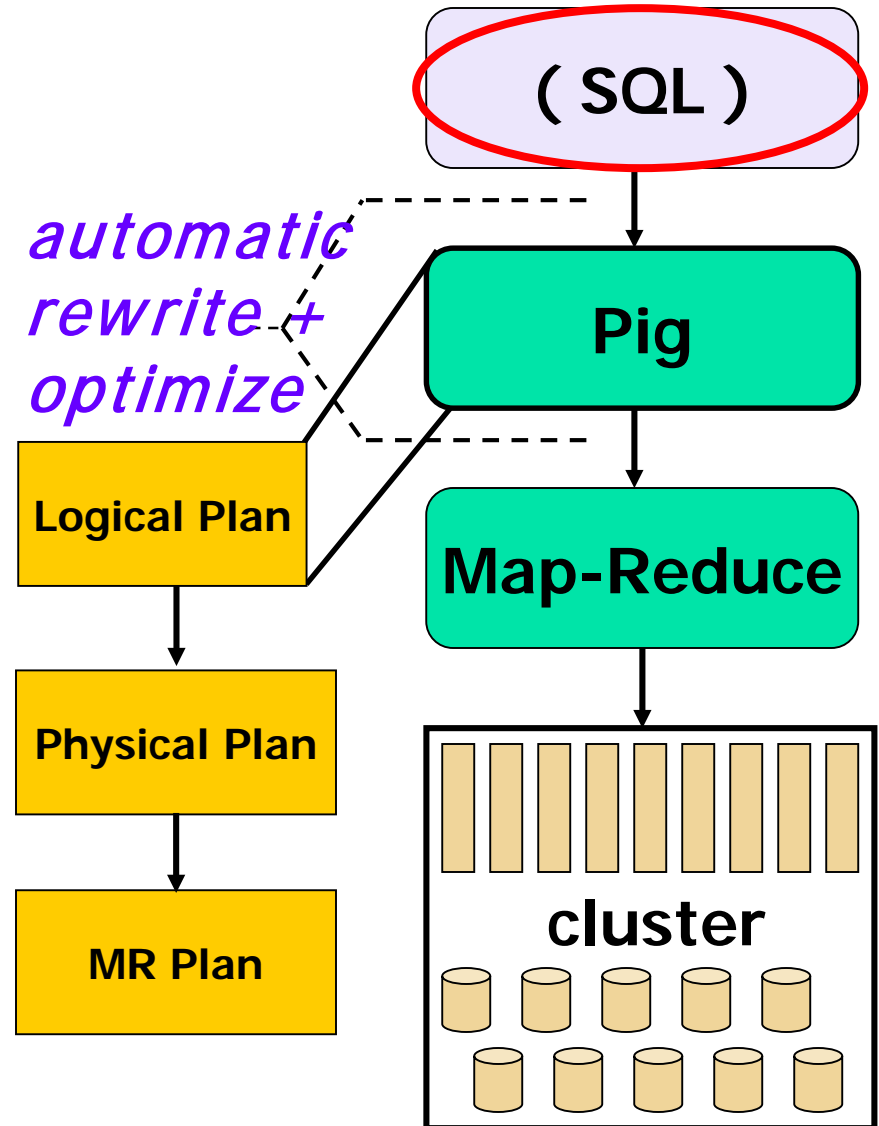
Pig Example

- Show users aged 18-25

```
Users = LOAD 'users.txt'
        USING PigStorage(',') AS (name, age);
Fltrd = FILTER Users
        BY age >= 18 AND age <= 25;
Names = FOREACH Fltrd GENERATE name;

STORE Names INTO 'names.out';
```

How to execute

- ## Local:
  - `pig` **`-x local`** `foo.pig`

- ## Hadoop (HDFS):
  - `pig foo.pig`
  - `pig` **`-Dmapred.job.queue.name=xxx`** `foo.pig`
    - `hadoop queue -showacls`

How to execute

- Interactive pig shell
  - `$ pig`
  - grunt> _

Load Data

```
Users = LOAD 'users.txt'
        USING PigStorage(',') AS (name, age);
```

- LOAD … AS …
- PigStorage(',') to specify separator

John,18
Mary,20
Bob,30

→

name	age
John	18
Mary	20
Bob	30

Filter

```
Fltrd = FILTER Users
        BY age >= 18 AND age <= 25;
```

- FILTER ... BY ...
    - constraints can be composite

name	age
John	18
Mary	20
Bob	30

→

name	age
John	18
Mary	20

Generate / Project

```
Names = FOREACH Fltrd GENERATE name;
```

- FOREACH … GENERATE

name	age
John	18
Mary	20

→

name
John
Mary

Store Data

```
STORE Names INTO 'names.out';
```

- STORE … INTO …
  - PigStorage(',') to specify separator if multiple fields

Command - JOIN

```
Users = LOAD 'users' AS (name, age);
Pages = LOAD 'pages' AS (user, url);
Jnd   = JOIN Users BY name, Pages BY user;
```

name	age
John	18
Mary	20
Bob	30

user	url
John	yaho
Mary	goog
Bob	bing

name	age	user	url
John	18	John	yaho
Mary	20	Mary	goog
Bob	30	Bob	bing

Command - GROUP

```
Grpd  = GROUP Jnd by url;
describe Grpd;
```

name	age	url
John	18	yhoo
Mary	20	goog
Dee	25	yhoo
Kim	40	bing
Bob	30	bing

→

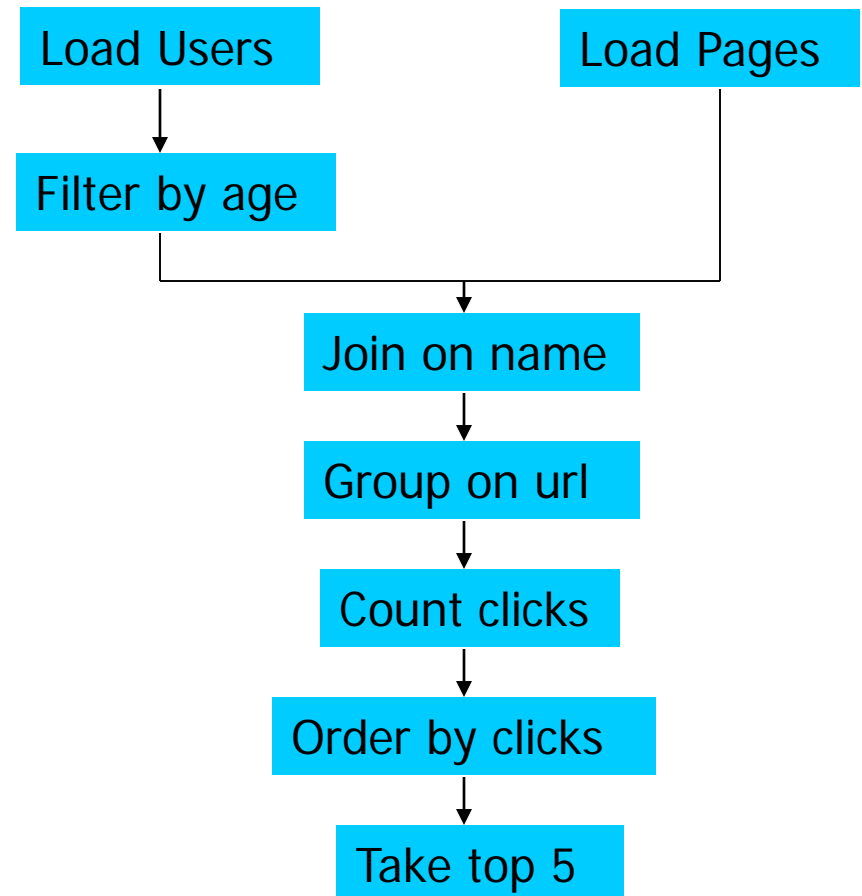|	
yhoo	(John, 18, yhoo) (Dee, 25, yhoo)
goog	(Mary, 20, goog)
bing	(Kim, 40, bing) (Bob, 30, bing)

Other Commands

- `PARALLEL` – controls #reducer

- `ORDER` – sort by a field

- `COUNT` – eval: count #elements

- `COGROUP` – structured JOIN

- More at http://hadoop.apache.org/pig/docs/r0.5.0/piglatin_reference.html

An Example Problem

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

```
Load Users            Load Pages
    |
    v
Filter by age
    |
    +---------+   +-----+
              v   v
           Join on name
                |
                v
           Group on url
                |
                v
           Count clicks
                |
                v
           Order by clicks
                |
                v
            Take top 5
```
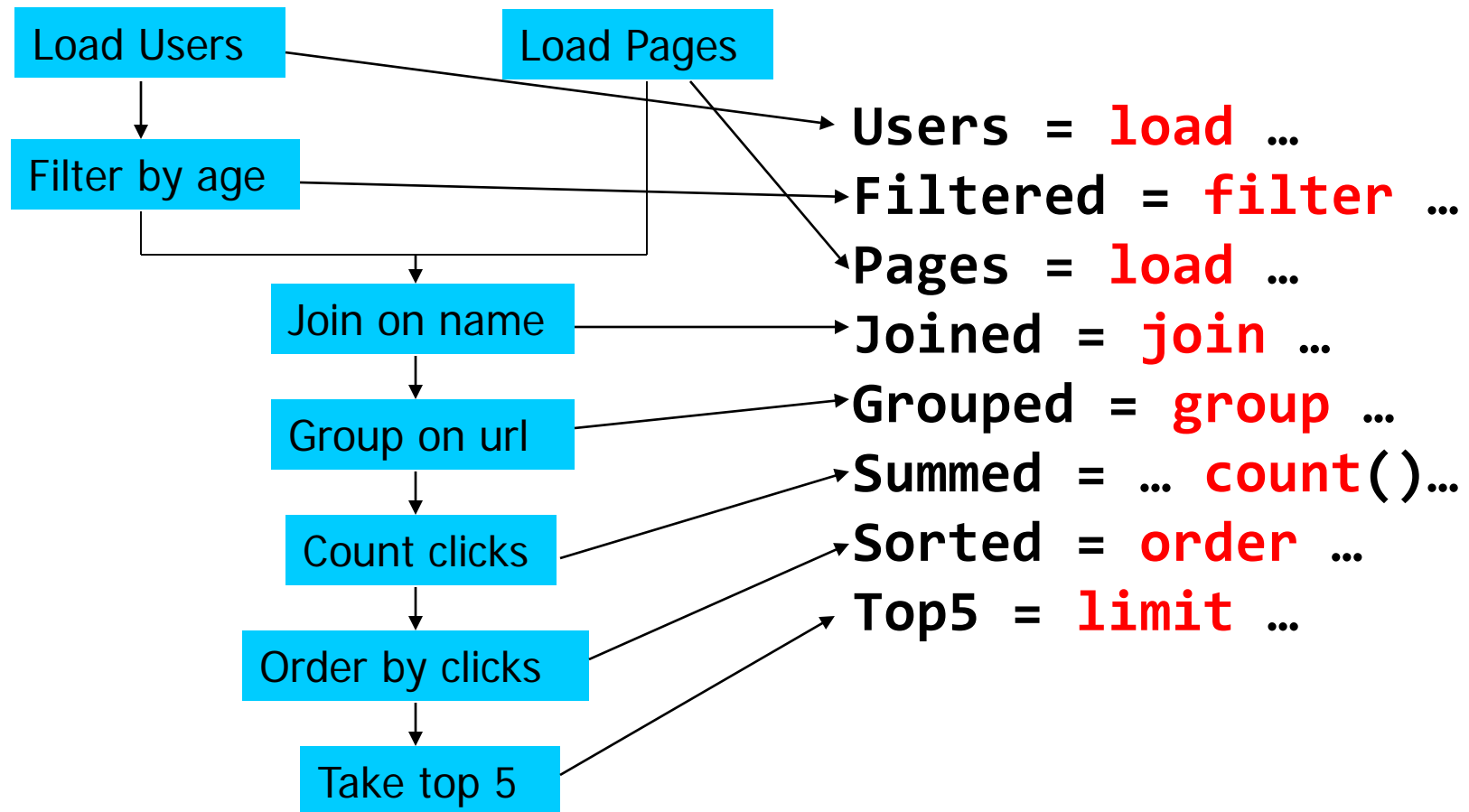
Pig Latin

```
Users    = load 'users' as (name, age);
Filtered = filter Users by
                        age >= 18 and age <= 25;
Pages    = load 'pages' as (user, url);
Joined   = join Filtered by name, Pages by user;
Grouped  = group Joined by url;
Summed   = for each Grouped generate group,
                        count(Joined) as clicks;
Sorted   = order Summed by clicks desc;
Top5     = limit Sorted 5;

store Top5 into 'top5sites';
```
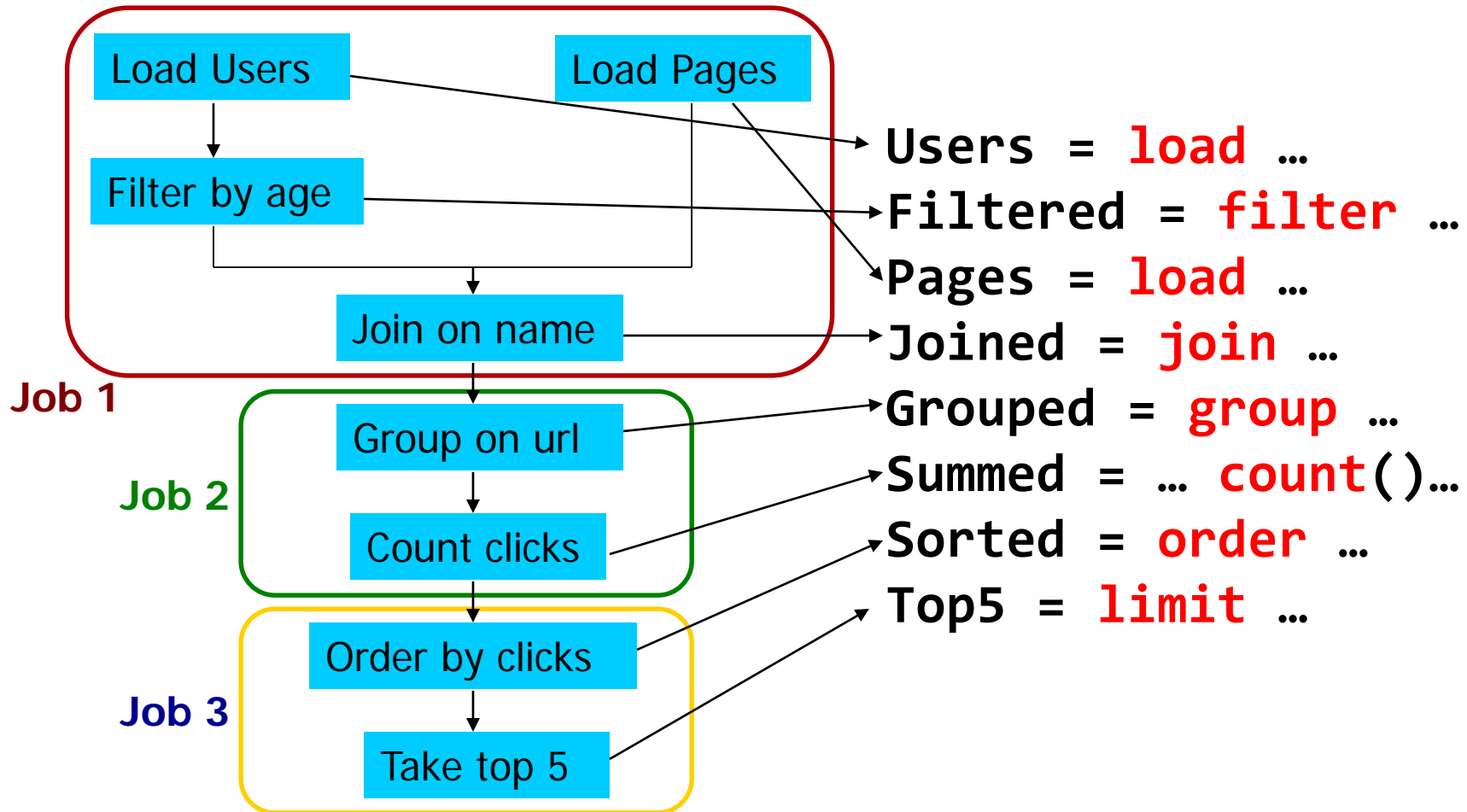
Translation to MapReduce

Notice how naturally the components of the job translate into Pig Latin.

Flow	Pig Latin
Load Users → Filter by age	**Users = load …**
Load Pages	**Filtered = filter …**
	Pages = load …
Join on name	**Joined = join …**
Group on url	**Grouped = group …**
Count clicks	**Summed = … count()…**
Order by clicks	**Sorted = order …**
Take top 5	**Top5 = limit …**

Translation to MapReduce

Notice how naturally the components of the job translate into Pig Latin.

Load Users	Load Pages
Filter by age	
Join on name	

Job 1

| Group on url |
| Count clicks |

Job 2

| Order by clicks |
| Take top 5 |

Job 3

Users = load …
Filtered = filter …
Pages = load …
Joined = join …
Grouped = group …
Summed = … count()…
Sorted = order …
Top5 = limit …

Hive

- **Apache Hive** is a data warehouse software project built on top of Apache Hadoop for providing data query and analysis.

- Hive gives an SQL-like interface to query data stored in various databases and file systems that integrate with Hadoop.

Hive

- Traditional SQL queries must be implemented in the <u>MapReduce</u> Java API to execute SQL applications and queries over distributed data.

- Hive provides the necessary SQL abstraction to integrate SQL-like queries (<u>HiveQL</u>) into the underlying Java without the need to implement queries in the low-level Java API.

Hive

- **Developed at Facebook**
- **Used for most jobs of Facebook**
- **Relational database built on Hadoop**
  - Maintains table schema
  - SQL-like query language (which can also call Hadoop streaming scripts)
  - Supports table partitioning, complex data types, sampling, some query optimization

Hive Query Language

- **Basic SQL**
  - From clause sub-query
  - ANSI JOIN (equi-join only)
  - Multi-Table insert
  - Multi group-by
  - Sampling
  - Objects traversal
- **Extensibility**
  - Pluggable Map-reduce scripts using TRANSFORM

HiveQL (~SQL)   PigLatin

Hive   Pig

MR

HDFS

thecloudavenue.com
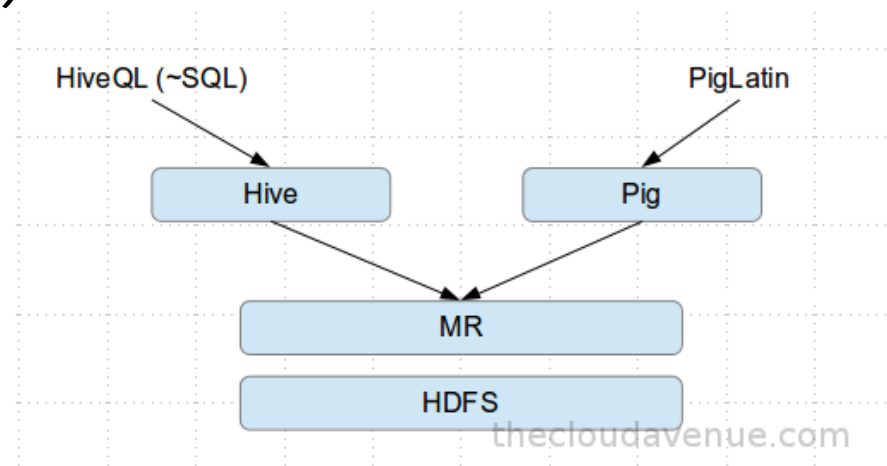
Hive Query Language

- JOIN

  SELECT t1.a1 as c1, t2.b1 as c2

  FROM t1 JOIN t2 ON (t1.a2 = t2.b2);

- INSERTION

  INSERT OVERWRITE TABLE t1

  SELECT * FROM t2;

Hive Query Language

- Insertion

INSERT OVERWRITE TABLE sample1 '/tmp/hdfs_out'
   SELECT * FROM sample WHERE ds='2017-09-24';

INSERT OVERWRITE DIRECTORY '/tmp/hdfs_out'
   SELECT * FROM sample WHERE ds='2017-09-24';

INSERT OVERWRITE LOCAL DIRECTORY '/tmp/hive-
   sample-out'
   SELECT * FROM sample;

Example

> INSERT INTO TABLE student01 VALUES ('Ball', 30, 40, 50), ('Tom', 60, 70, 80), ('EQ', 90, 95, 100);

```
hive> INSERT INTO TABLE student01
    > VALUES ('Ball', 30, 40, 50), ('Tom', 60, 70, 80), ('EQ', 90, 95, 100);
Query ID = hadoop_20180204231613_fe55e560-20ea-46d6-bd5e-bc92aef6098d
Total jobs = 3
Launching Job 1 out of 3
Number of reduce tasks is set to 0 since there's no reduce operator
Job running in-process (local Hadoop)
2018-02-04 23:16:19,592 Stage-1 map = 100%,  reduce = 0%
Ended Job = job_local1734397660_0001
Stage-4 is selected by condition resolver.
Stage-3 is filtered out by condition resolver.
Stage-5 is filtered out by condition resolver.
Moving data to: hdfs://localhost:9000/user/hive/warehouse/school.db/student01/.h
ive-staging_hive_2018-02-04_23-16-13_356_4803938009316820562-1/-ext-10000
Loading data to table school.
Table school.student01 stats:              totalSize=40, rawDataSize=
37]
MapReduce Jobs Launched:
Stage-Stage-1:  HDFS Read: 40 HDFS Write: 152 SUCCESS
Total MapReduce CPU Time Spent: 0 msec
OK
Time taken: 7.655 seconds
hive>
```

> select * from student01;
> select * from student02;

> select * from student01;
> select * from student02;

```
hive> select * from student01;
OK
Ball    30      40      50
Tom     60      70      80
EQ      90      95      100
Time taken: 0.24 seconds, Fetched: 3 row(s)
hive> select * from student02;
OK
Time taken: 0.193 seconds
hive>
```
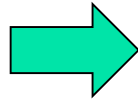
103

Hive Query Language

- Map Reduce

```
FROM (MAP doctext USING 'python wc_mapper.py' AS (word, cnt)
FROM docs
CLUSTER BY word
)
REDUCE word, cnt USING 'python wc_reduce.py';

FROM (FROM session_table
SELECT sessionid, tstamp, data
DISTRIBUTE BY sessionid SORT BY tstamp
)
REDUCE sessionid, tstamp, data USING 'session_reducer.sh';
```
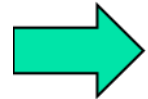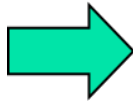
Example

```
$  cd /usr/local/hadoop
$  mkdir input
```

➡️

```
$  cd /usr/local/hadoop/input
$  echo "hello world" > test01.txt
$  echo "hello hadoop" > test02.txt
```

➡️

```
$  hive
```

➡️

```
>  create table docs(line string);
>  load data local inpath '/usr/local/hadoop/input' overwrite into table docs;
```

```
>  select * from word_count;
```

```
hadoop@master: /usr/local/hadoop/input
hive> select * from word_count;
OK
hadoop    1
hello     2
world     1
Time taken: 0.193 seconds, Fetched: 3 row(s)
hive>
```

Summary

- MapReduce's data-parallel programming model hides <u>complexity of distribution</u> and <u>fault tolerance</u>

- Principal philosophies:
    - Scale, so you can throw problems to hardware
    - Cheap, saving hardware, programmer and administration costs but can own fault tolerance

- Hive and Pig further simplify programming

- MapReduce is not suitable for all problems, but it may save you a lot of time.

Outline

- MapReduce architecture
- Sample applications
- Introduction to Hadoop
- Higher-level query languages: Pig & Hive
- Current research

Cloud Programming Research

- More general execution engines
  - **Dryad** (Microsoft): general task directed acyclic graph
  - **S4** (Yahoo!): streaming computation
  - **Pregel** (Google): in-memory iterative graph algs.
  - **Spark** (Berkeley): general in-memory computing

- Language-integrated interfaces
  - Run computations directly from host language
  - **DryadLINQ** (MS), **FlumeJava** (Google), **Spark**

Spark

Fast, Interactive, Language-Integrated Cluster Computing

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

www.spark-project.org

amplab
UC BERKELEY

Project Goals

- Extend the MapReduce model to better support two common classes of analytics apps:
  - **Iterative** algorithms (machine learning, graphs)
  - **Interactive** data mining (R, excel, python)
- Enhance programmability:
  - Integrate into **Scala** programming language
  - Allow interactive use from Scala interpreter

- Acyclic data flow is inefficient for applications that repeatedly reuse a working set of data.

- With current frameworks, apps reload data from stable storage on each query

Motivation

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

Benefits of data flow: runtime can decide where to run tasks and can automatically recover from failures
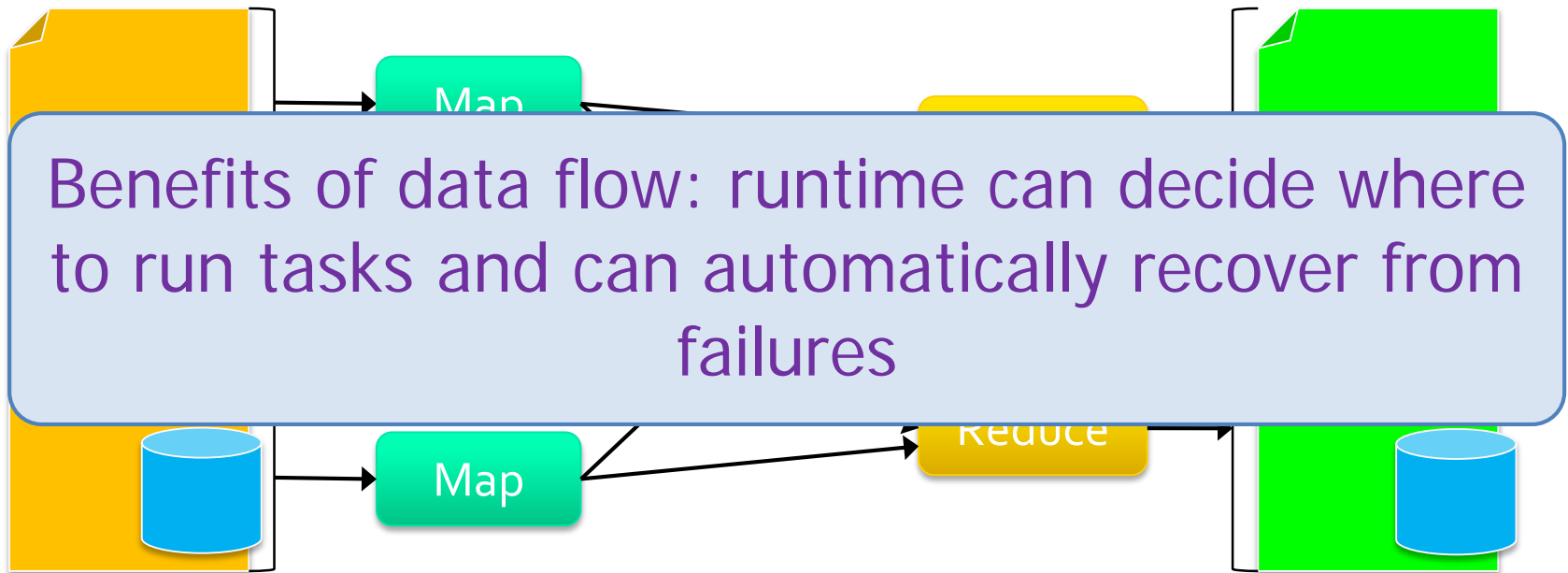
Spark Motivation
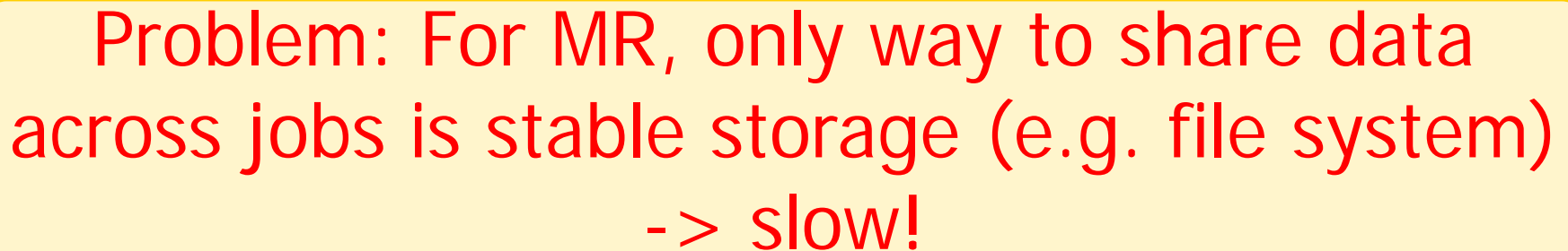
- MapReduce simplified "big data" analysis on large, unreliable clusters

- However, many organizations started using it widely, users wanted more:
  - More complex, multi-stage applications
  - More interactive queries
  - More low-latency online processing

Spark Motivation

■ Complex jobs, interactive queries and online processing all need one thing that MapReduce lacks:

Efficient primitives for **data sharing**

Problem: For MR, only way to share data across jobs is stable storage (e.g. file system) -> slow!

Iterative job   **Interactive mining**   **Stream processing**
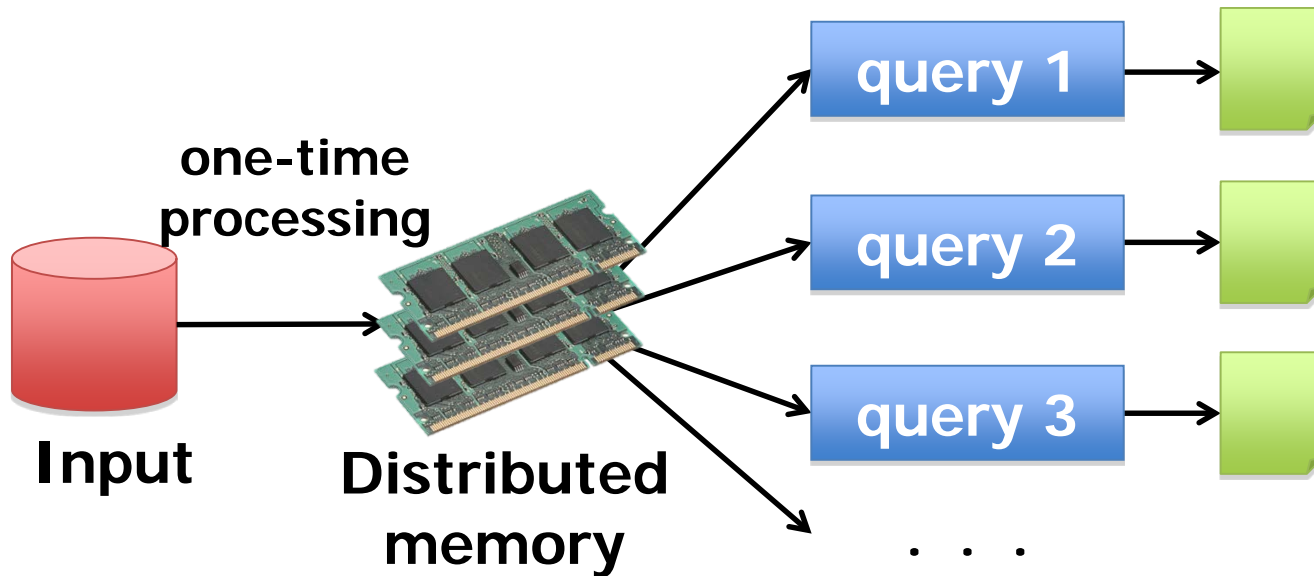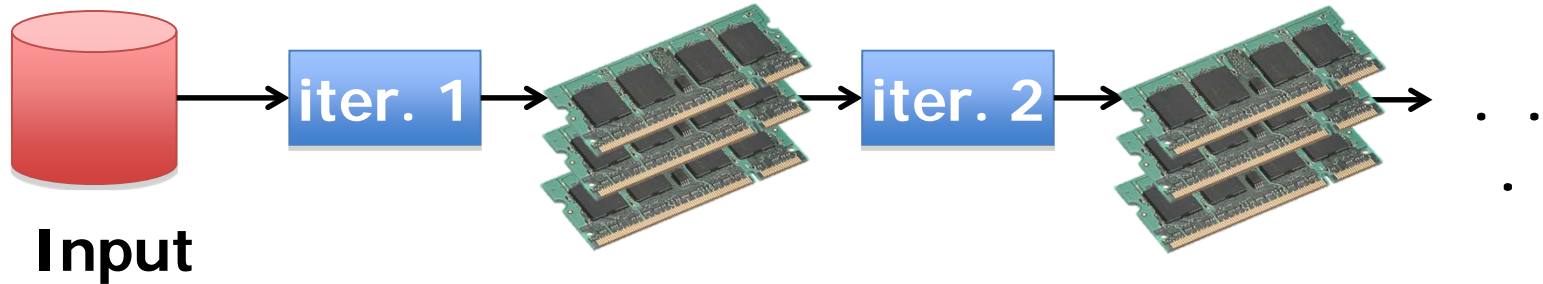
Example: Data Sharing

HDFS read → **iter. 1** → HDFS write → HDFS read → **iter. 2** → HDFS write → . . .

Input

Serial Process

HDFS read → **query 1** → result 1
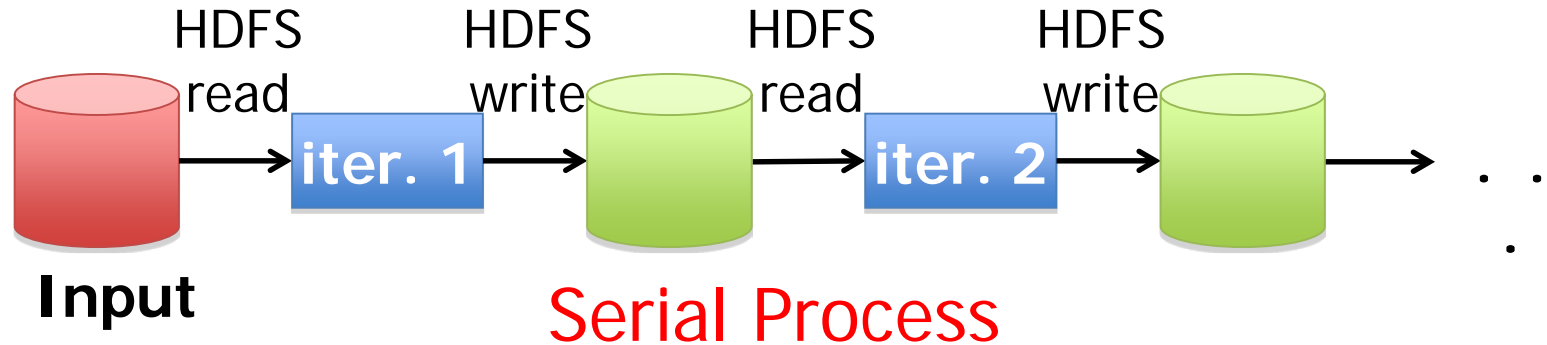
Opportunity: DRAM is getting cheaper ➔ use main memory for intermediate results instead of disks
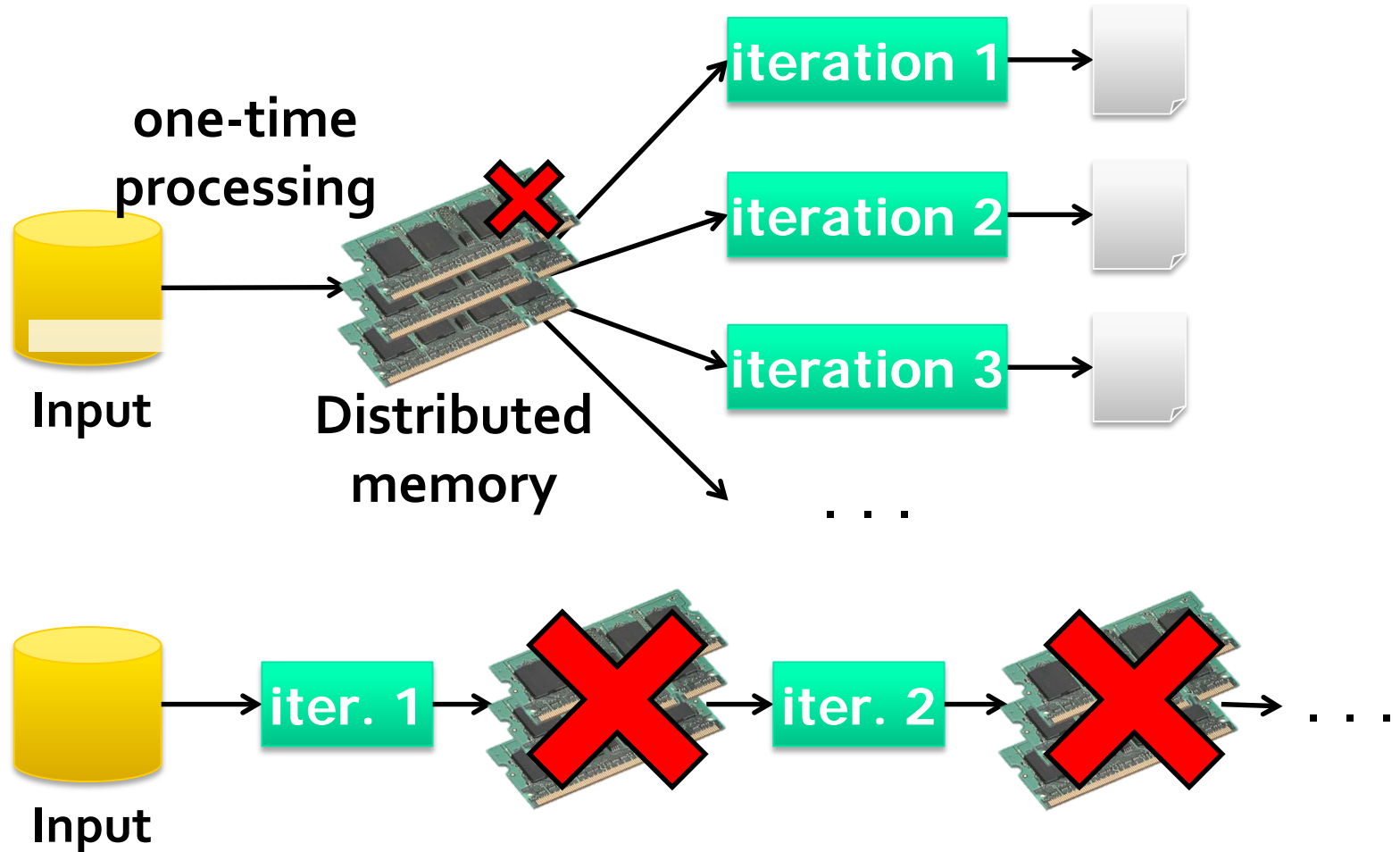
Input

. . .

Parallel Process

Goal: Data Sharing in Memory

Input iter. 1 → iter. 2 → . . .

one-time processing

Input Distributed memory → query 1, query 2, query 3, . . .

10～100× faster than network and disk

Resilient Distributed Datasets (RDDs) Recovery

one-time processing

Input

Distributed memory

iteration 1

iteration 2

iteration 3

. . .

Input

iter. 1

iter. 2

. . .

Resilient Distributed Datasets (RDDs)

- Restricted form of distributed shared memory
  - Read only/ immutable , partitioned collections of records
  - Deterministic
  - From coarse grained operations (map, filter, join, etc.)
  - From stable storage or other RDDs
  - User controlled persistence
  - User controlled partitioning

RDD

- A read-only <u>multiset</u> of data items distributed over a cluster of machines, that is maintained in a <u>fault-tolerant</u> way.

- MapReduce programs read input data from disk, <u>map</u> a function across the data, <u>reduce</u> the results of the map, and store reduce results on disk.

- RDDs function as a <u>working set</u> for distributed programs that offers a (deliberately) restricted form of distributed <u>shared memory</u>

RDD運算類型	說明
「轉換」運算 Transformation	• RDD執行「轉換」運算的結果，會產生另外一個RDD。 • 但是由於RDD的lazy特性，「轉換」運算並不會立刻實際執行，它會等到執行到「動作」運算，才會實際執行。
「動作」運算 Action	• RDD執行「動作」運算後，不會產生另外一個RDD，它會產生數值、陣列或寫入檔案系統。 • RDD執行「動作」運算時，會立刻實際執行，並且連同之前的「轉換」運算一併執行。
「持久化」 Persistence	• 對於那些會重複使用的RDD，可以將RDD「持久化」在記憶體中做為後續使用，以加快執行效能。

RDDs

- Partitioned collections of records that can be stored in memory across the cluster

- Manipulated through a diverse set of transformations (*map*, *filter*, *join*, etc)

- Fault recovery without costly replication
  - Remember the series of transformations that built an RDD (from its lineage) to **recompute** lost data

Programming Model

RDDs

- Immutable, partitioned collections of objects
- Created through parallel *transformations* (map, filter, groupBy, join, ...) on data in stable storage
- Can be *cached* for efficient reuse

Actions on RDDs

- Count, reduce, collect, save, ...

RDD commands

- Create intRDD

  val intRDD = sc.parallelize(List(3,1, 2, 5, 5))

  intRDD.collect()

- Create stringRDD

  val stringRDD = sc.parallelize(List("Apple", "Orange", "Banana","Grape","Apple"))
  stringRDD.collect()

- Map function

- def addOne(x :Int):Int={ return (x+1) }

  intRDD.map(addOne).collect()

- intRDD.map(x => x + 1).collect()

```
hduser@master: ~/workspace/WordCount/data

scala> def addOne(x :Int):Int={
     | return (x+1)
     | }
addOne: (x: Int)Int

scala> intRDD.map(addOne).collect()
res24: Array[Int] = Array(4, 2, 3, 6, 6)

scala> intRDD.map(x => x + 1).collect()
res25: Array[Int] = Array(4, 2, 3, 6, 6)

scala> intRDD.map(_ + 1).collect()
res26: Array[Int] = Array(4, 2, 3, 6, 6)
```

具名函數的語法

匿名函數的語法

匿名函數+匿名參數的語法

124

RDD commands

- ## filter
  stringRDD.filter(x =>x.contains("ra")).collect()

- ## union
  intRDD1.union(intRDD2).union(intRDD3).collect()

  (intRDD1 ++ intRDD2++ intRDD3).collect()

- ## intersection
  intRDD1.intersection(intRDD2).collect()

- ## subtract
  intRDD1.subtract(intRDD2).collect()

- ## Cartesian
  intRDD1.cartesian(intRDD2).collect()

Example: Log Mining

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

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

messages.filter(_.contains("foo")).count
messages.filter(_.contains("bar")).count
. . .
```
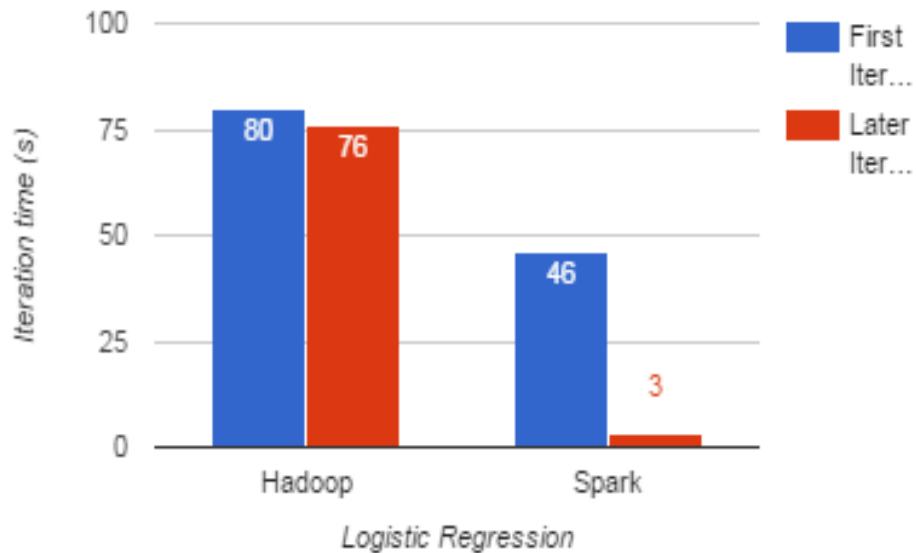
Bas...

Transformed RDD

Cache 1

Worker

results

tasks

Block 1

Driver

Cache 2

Worker

Cache 3

Block 2

Worker

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

Block 3
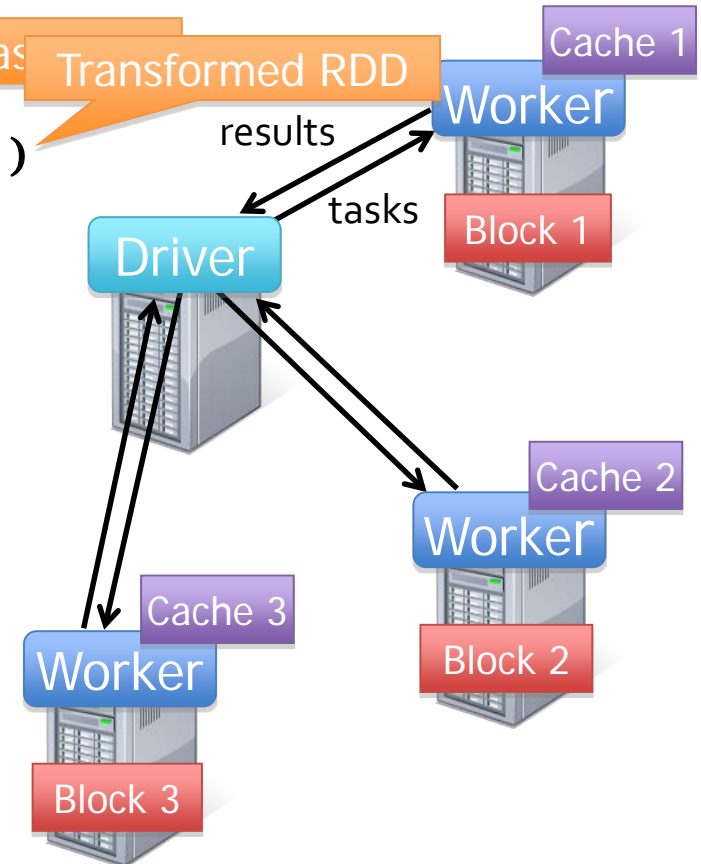
Evaluation

10 iterations on 100GB data using 25-100 machines
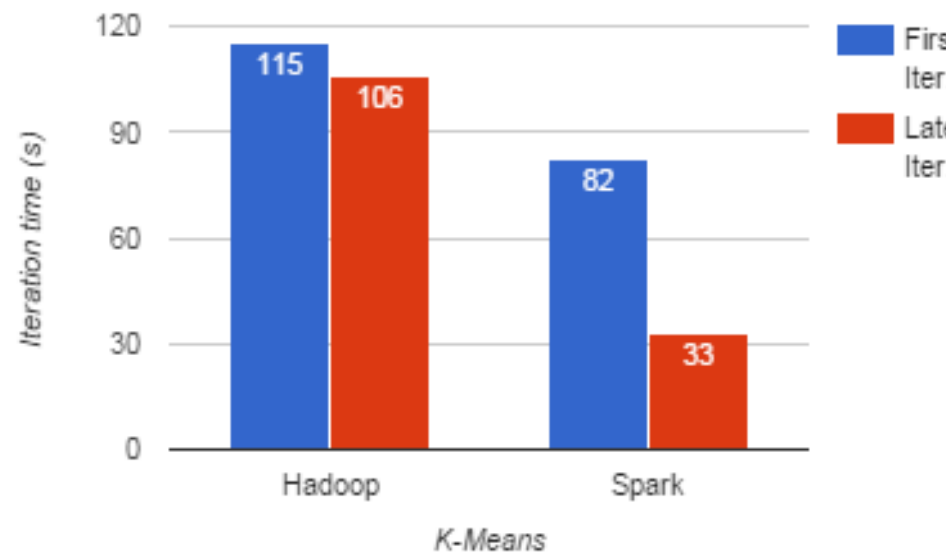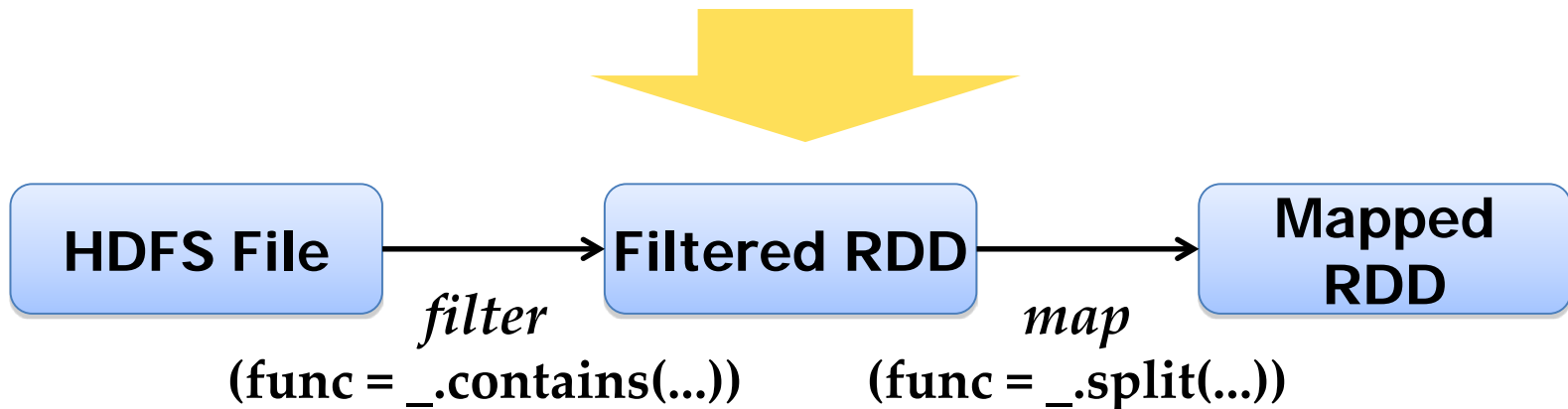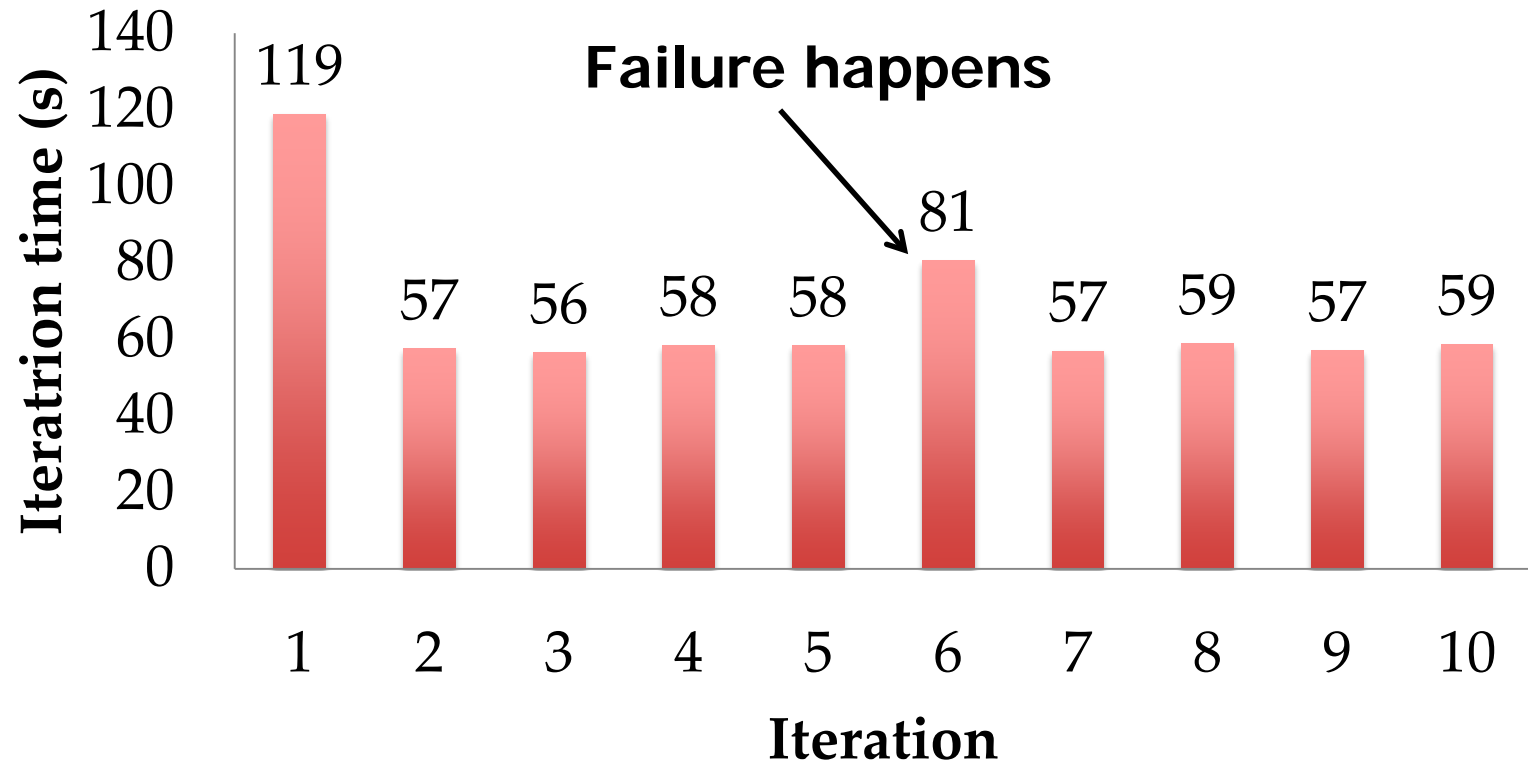
Fault Recovery

RDDs track **lineage** information that can be used to efficiently reconstruct lost partitions

Ex:

```
messages =
textFile(...).filter(_.startsWith("ERROR"))
                    .map(_.split('\t')(2))
```
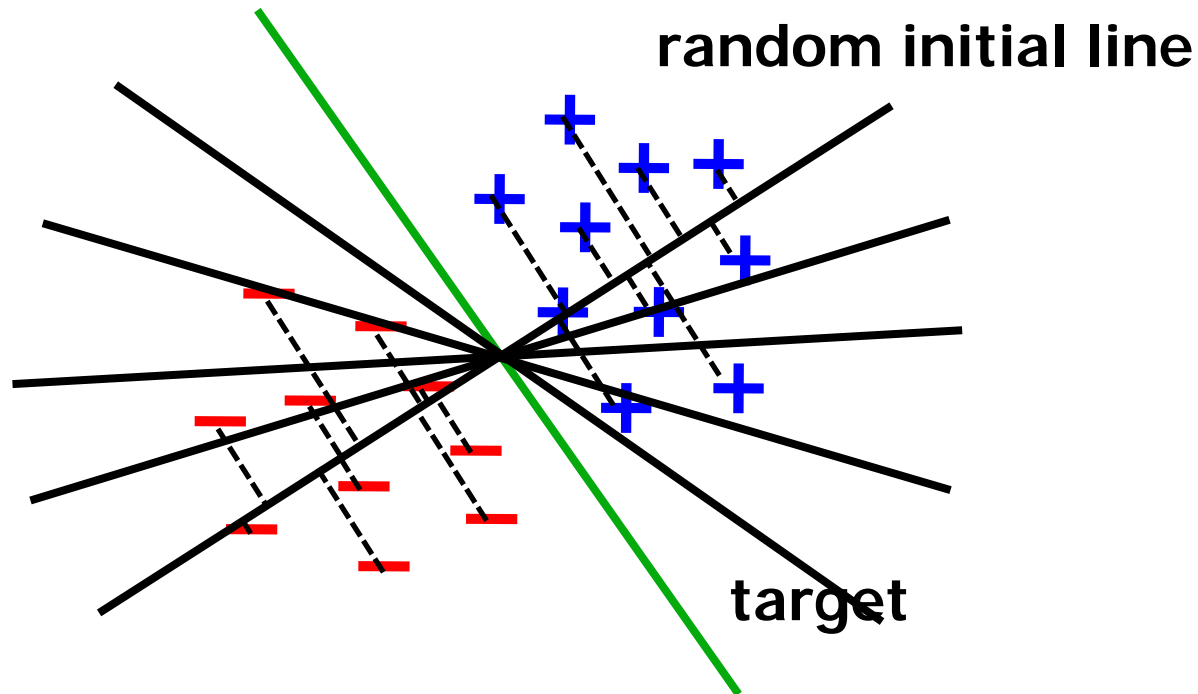

HDFS File		Filtered RDD		Mapped RDD
	filter (func = _.contains(...))		*map* (func = _.split(...))	

Fault Recovery Results

Example: Logistic Regression

Find best line separating two sets of points

random initial line

target

Logistic Regression Code

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

var w = Vector.random(D)

for (i <- 1 to ITERATIONS)
{
  val gradient = data.map(p =>
    (1 / (1 + exp(-p.y*(w dot p.x))) - 1) * p.y * p.x
  ).reduce(_ + _)
  w -= gradient
}

println("Final w: " + w)
```
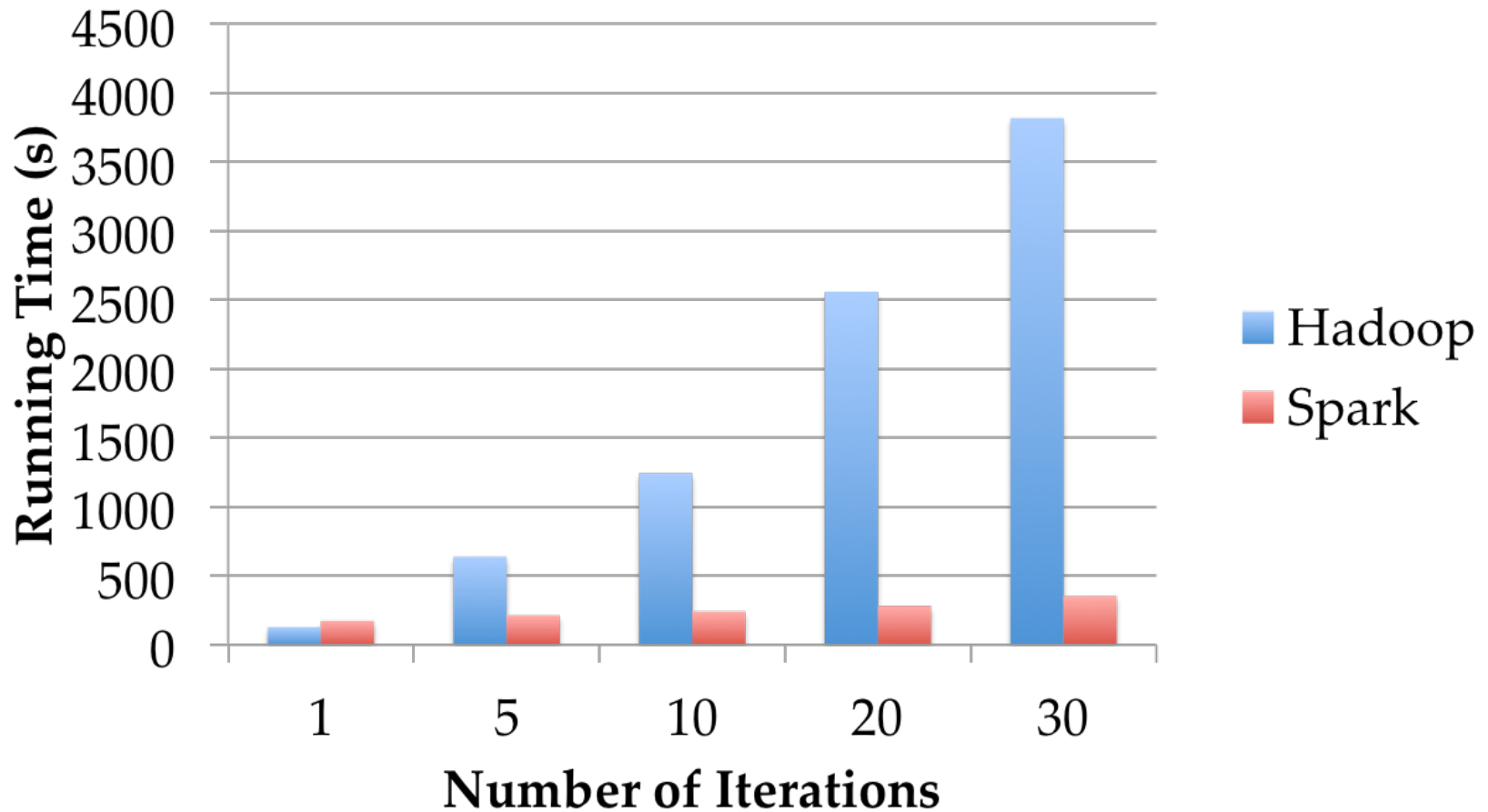
Logistic Regression Performance

Example: Collaborative Filtering

Goal: predict users' movie ratings based on past ratings of other movies

$$R = \begin{pmatrix} 1 & ? & ? & 4 & 5 & ? & 3 \\ ? & ? & 3 & 5 & ? & ? & 3 \\ 5 & ? & 5 & ? & ? & ? & 1 \\ 4 & ? & ? & ? & ? & 2 & ? \end{pmatrix}$$

Users

Movies

Spark Applications

- In-memory data mining on Hive data (Conviva)
- Predictive analytics (Quantified)
- City traffic prediction (Mobile Millennium)
- Twitter spam classification (Monarch)
- Collaborative filtering via matrix factorization
- Time series analysis
- Network simulation

…

Mobile Millennium Project

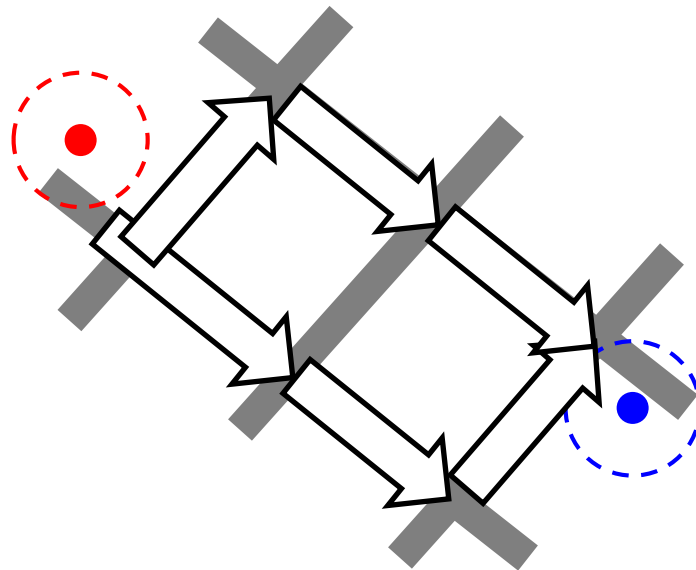Estimate city traffic using GPS observations from probe vehicles
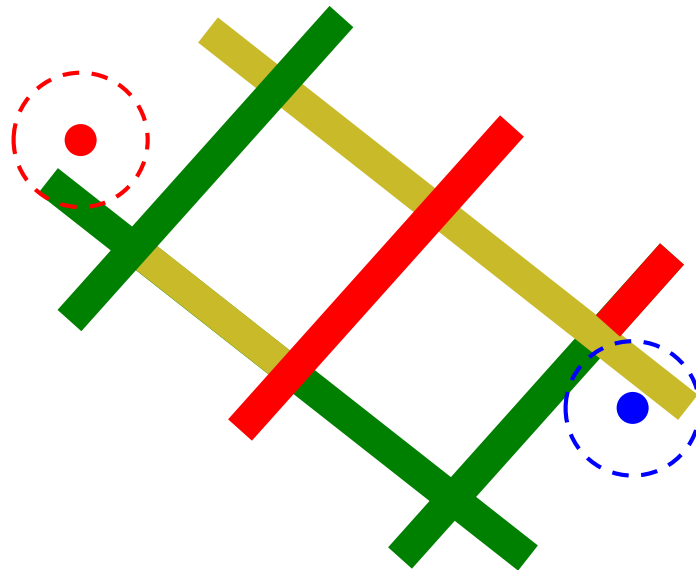(e.g. SF taxis)
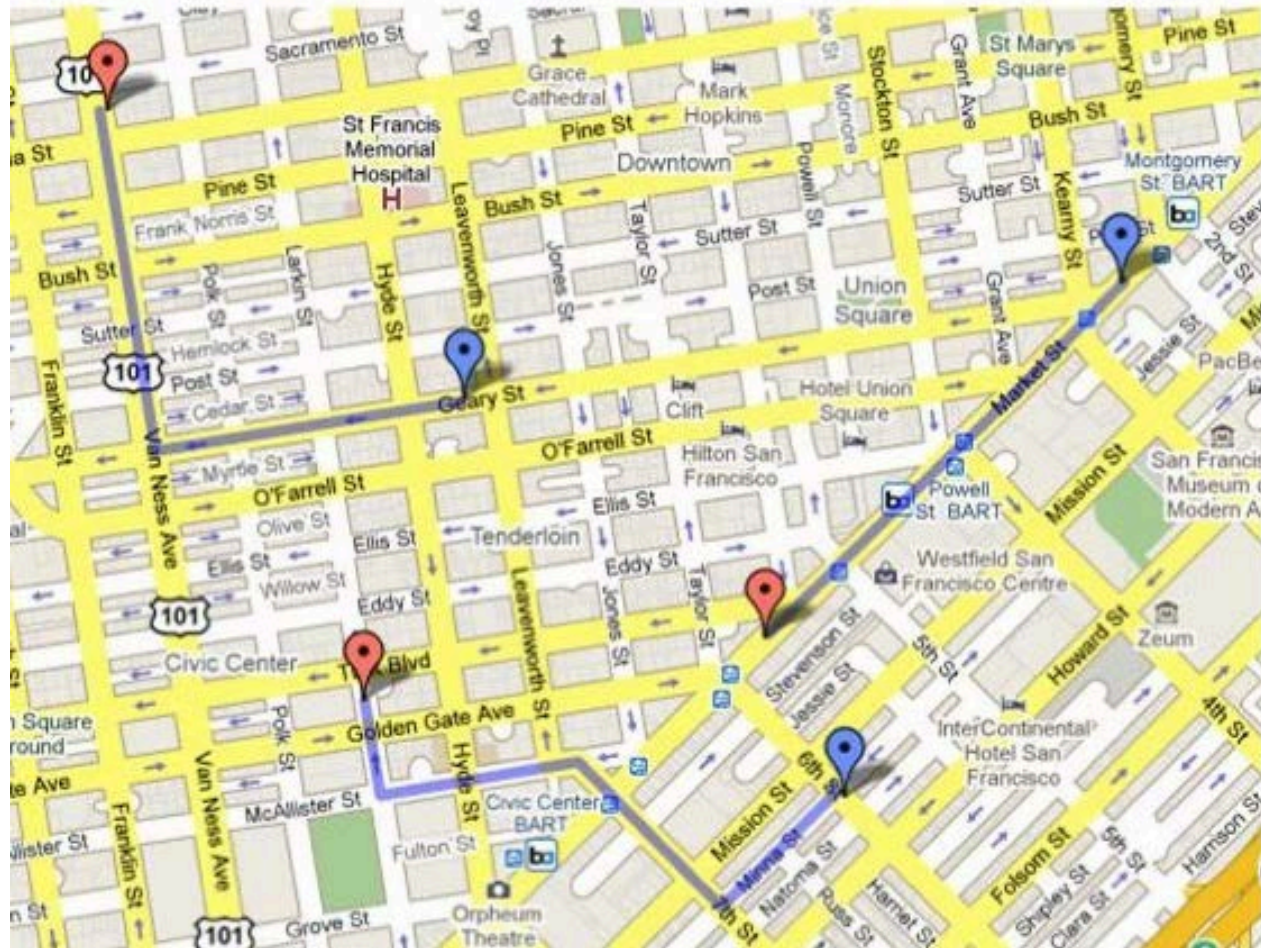
Sample Data

Challenge

- Data is noisy and sparse (1 sample/minute)
- Must infer path taken by each vehicle in addition to travel time distribution on each link

Challenge

- Data is noisy and sparse (1 sample/minute)
- Must infer path taken by each vehicle in addition to travel time distribution on each link

Solution

EM algorithm to estimate paths and travel time distributions simultaneously

observations

flatMap

weighted path samples

groupByKey

link parameters

broadcast
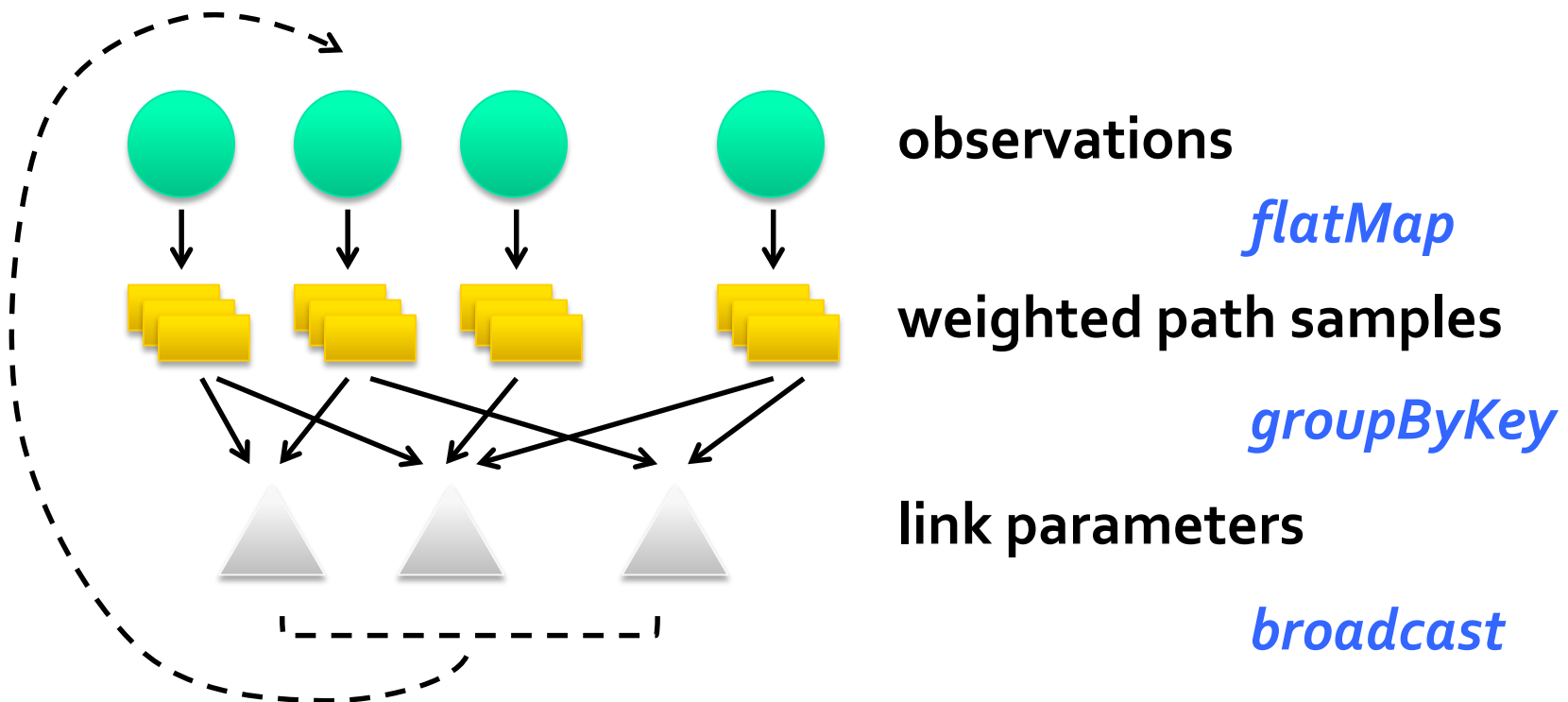
Frameworks Built on Spark

- Pregel on Spark (Bagel)
  - Google message passing model for graph computation
  - 200 lines of code
- Hive on Spark (Shark)
  - 3000 lines of code
  - Compatible with Apache Hive
  - ML operators in <u>Scala</u>

Scala is an <u>object-functional</u> <u>programming</u> and <u>scripting language</u> for general <u>software applications</u>, <u>statically typed</u>, designed to concisely express solutions in an elegant, <u>type-safe</u> and lightweight manner.

Scala_"Hello World" example

- **Edit**

```scala
object HelloWorld extends App {
  println("Hello, World!")
}
```

- **Compiler**

```
$ scalac HelloWorld.scala
```

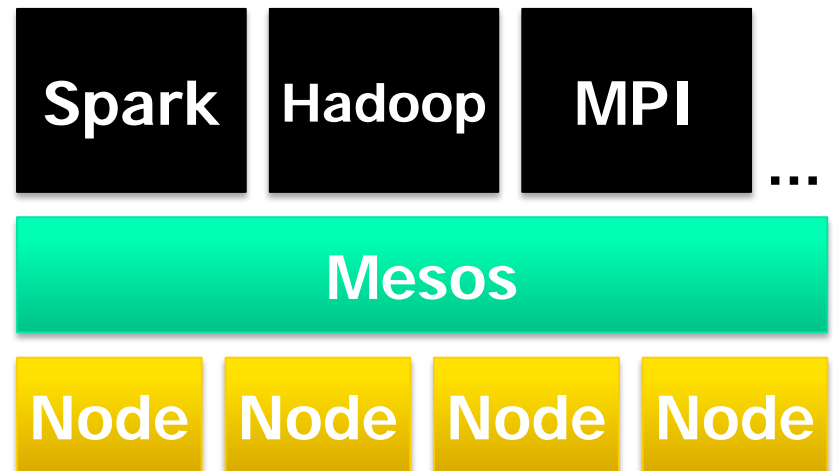- **Run**

```
$ scala HelloWorld
```

Implementation

Runs on <u>Apache Mesos</u>
to share resources with
Hadoop & other apps

Can read from any
Hadoop input source
(e.g. HDFS)
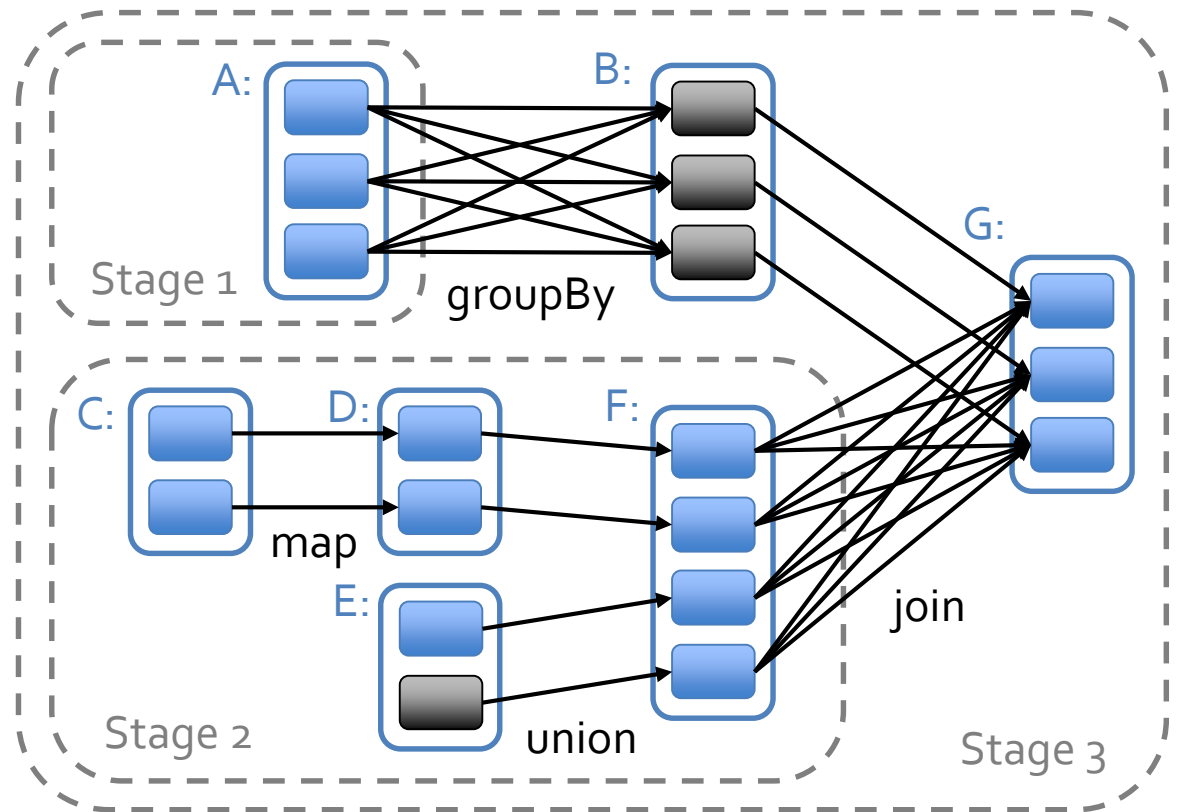
- No changes to Scala compiler

Spark	Hadoop	MPI	...

Mesos

Node	Node	Node	Node

Spark Scheduler

Pipelines functions within a stage

Cache-aware work reuse & locality

Partitioning-aware to avoid shuffles

= cached data partition

If You Want to Try It Out

- **<u>www.spark-project.org</u>**

- To run locally, just need Java installed

- Easy scripts for launching on Amazon EC2

- Can call into any Java library from Scala

Other Resources

- Hadoop: http://hadoop.apache.org/common
- Pig: http://hadoop.apache.org/pig
- Hive: http://hadoop.apache.org/hive
- Spark: http://spark-project.org

- Hadoop video tutorials:
  www.cloudera.com/hadoop-training

- Amazon Elastic MapReduce:
  http://aws.amazon.com/elasticmapreduce/

Q&A

- For more information:
    - http://hadoop.apache.org/
    - http://developer.yahoo.com/hadoop/

- Who uses Hadoop?:
    - http://wiki.apache.org/hadoop/PoweredBy