

Cloud Computing – Part 2 of 3

Advanced Computer Networks
Summer Semester 2014



Cloud Computing Overview

- Last week:
 - Introduction
 - Basic principles and characteristics
 - Virtualization, Load Balancing, ...
- Last week & today:
 - Main standards
 - E.g., SDN, OpenFlow, MapReduce, Hadoop, ...
- Next week:
 - Advances in research

Cloud Computing Overview

- 22/5 lecture:
 - Shortened to ~45 minutes
 - Afterwards: A guest talk by Ruediger Geib from Deutsche Telekom

What's in the Cloud Ecosystem?

- Data sharing
 - File systems like Google File System (GFS), Hadoop File System (HDFS), ...
- Data analysis & programming abstractions
 - MapReduce, PIG, Hive, Spark, ...
- Multiplexing of resources & coordination
 - Mesos
- (DataBases
 - Cassandra (No-SQL without single point of failure))

Google Cloud Infrastructure

- Google File System (GFS) [1], 2003

- Distributed File System for entire cluster
- Single namespace

- Google MapReduce (MR) [7], 2004

- Runs queries/jobs on data
- Manages work distribution & fault-tolerance
- Colocated with file system

- Apache open source versions Hadoop DFS and Hadoop MR

The Google File System

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

ABSTRACT

We have designed and implemented the Google File System, a scalable distributed file system for large distributed data-intensive applications. It provides fault tolerance while running on inexpensive commodity hardware, and it delivers high aggregate performance to a large number of clients. While sharing many of the same goals as previous distributed file systems, our design has been driven by observations of our application workloads and technological environment, both current and anticipated, that reflect a marked departure from some earlier file system assumptions. This has led us to reexamine traditional choices and explore radically different design points. The file system has successfully met our storage needs. It is widely deployed within Google as the storage platform

1. INTRODUCTION

We have designed and implemented the Google File System (GFS) to meet the rapidly growing demands of Google's data processing needs. GFS shares many of the same goals as previous distributed file systems such as performance, scalability, reliability, and availability. However, its design has been driven by key observations of our application workloads and technological environment, both current and anticipated, that reflect a marked departure from some earlier file system design assumptions. We have reexamined traditional choices and explored radically different points in the design space. First, component failures are the norm rather than the exception. The file system consists of hundreds or even thousands of storage machines built from inexpensive com-

MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat
jeff@google.com, sanjay@google.com
Google, Inc.

Abstract

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

Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the details of partitioning the input data, scheduling the pro-

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

As a reaction to this complexity, we designed a new abstraction that allows us to express the simple computations we were trying to perform but hides the messy details of parallelization, fault-tolerance, data distribution and load balancing in a library. Our abstraction is in-

GFS/HDFS overview

- *Petabyte* storage
 - Large files (multi GB) containing many application objects (e.g., web documents)
 - Reason: unwieldy to manage millions/billions of KB sized files
 - Files split into large blocks (128 MB) and replicated across several nodes
 - Big blocks allow high throughput sequential reads/writes
- Data *striped* on hundreds/thousands of servers
 - Scan 100 TB on 1 node @ 50 MB/s = 24 days
 - Scan on 1000-node cluster = 35 minutes

GFS/HDFS overview (2)

- *Failures* will be the norm
 - Mean time between failures for 1 node = 3 years
 - Mean time between failures for 1000 nodes = 1 day
- Use *commodity* hardware
 - Failures are the norm anyway, buy cheaper hardware
- No complicated consistency models
 - Single writer, append-only data

GFS/HDFS overview (3)

- Most files are modified by *append* rather than *overwrite*
- *Random writes* are basically non-existent within a file
- Files are very often read *sequentially*
 - E.g., *MapReduce*
- As a result: Focus on optimization of appending rather than caching at client

Hadoop in detail

- Multiple components arranged in a *cluster*
 - Master node, backup node, client nodes
- Each of these runs file system and MapReduce elements
 - Master node: NameNode and JobTracker
 - Backup node: Secondary NameNode
 - Client nodes: DataNode and Tasktracker

Hadoop in detail – NameNode

- Run by master node: keeps directory tree of all files in the distributed file system (DFS)
 - There is one single master node in the DFS
- Tracks *where* the data is stored
 - Doesn't store data itself!
- Client nodes can ask for location of data
- Client nodes *have* to ask if they want to copy/modify/delete/add a file

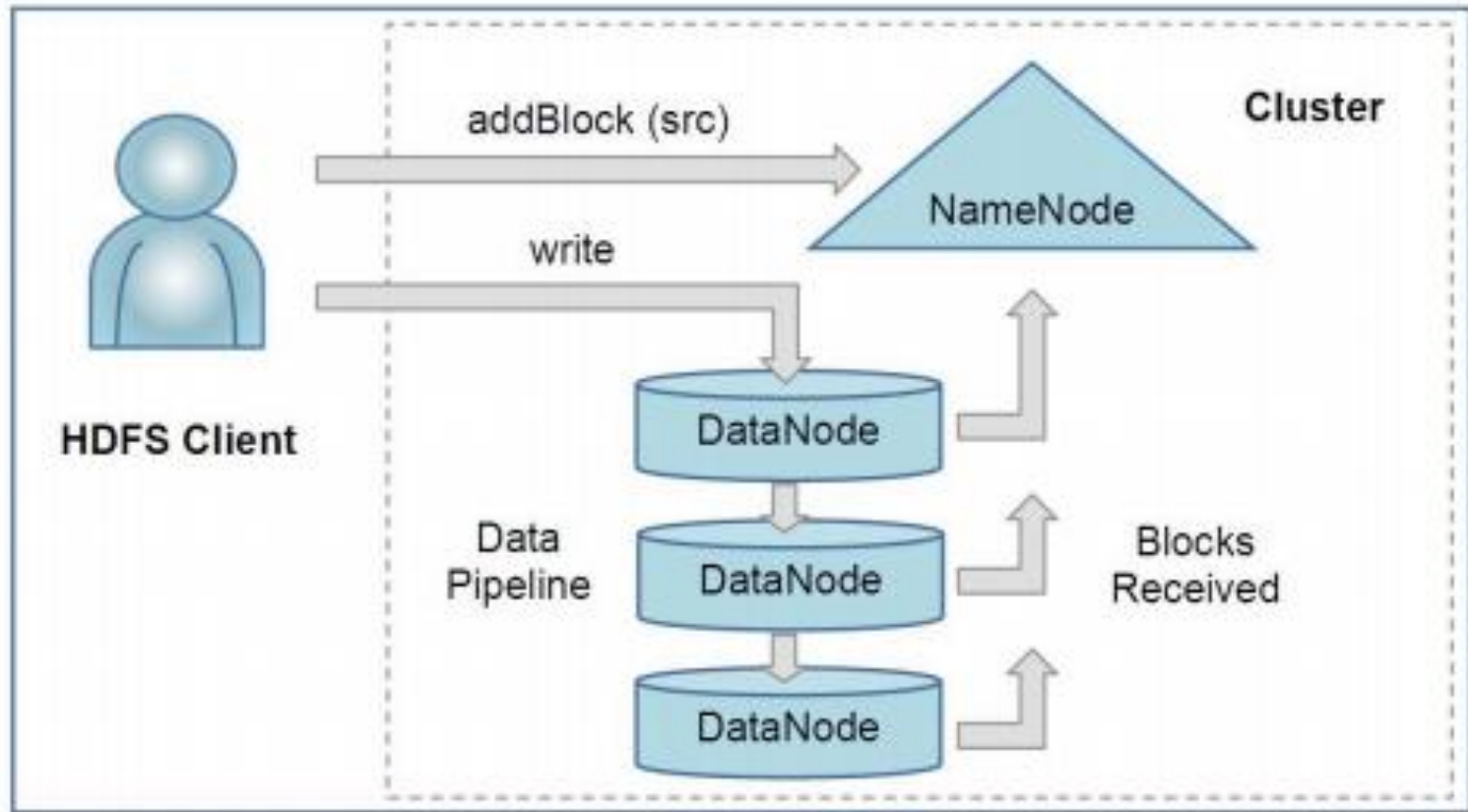
Hadoop in detail – Secondary Name Node

- NameNode: single point of failure
- Secondary NameNode creates checkpoints of NameNode
- Backup in case of failure of NameNode

Hadoop in detail – DataNode

- Stores the actual data in HDFS
- Data is replicated among multiple DataNodes
 - Typically: three
- Provides an interface for communication with NameNode and client applications (*directly!*)
- Any new DataNode is allowed to join the cluster at any time
 - Can get storage ID from NameNode
 - Elasticity!

Hadoop in detail – Adding a File [2]



Hadoop in detail – Adding a File

- Pipeline: setup, data streaming, close (cf TCP?)
- Data streaming: handles packets (typically 64KB)
 - Packets are ACKed as in typical networking protocols
- After adding a file, it can only be appended, not modified
 - I.e., if you want to modify, you have to rewrite the whole file!
 - Or: have, e.g., database entries appended instead of modified (common practice)
 - Also: FS like GFS/HDFS typically expect long-term storage applications

Hadoop in detail – Reading a File

- Client asks NameNode for a list of replicas
 - Tries closest replica (distance) first (lowest latency)
- Read can fail: DataNode failure, DataNode not host of replica anymore, data corrupted
 - Corruption detection: checksum on each data block

Hadoop in detail – Trackers

- JobTracker:
 - Run at master node
 - Distributes *MapReduce* tasks to client nodes (ideally those who have the data or are in the same rack – overhead?)
- TaskTracker
 - Run at client node
 - Accepts *map, reduce, and sort/shuffle tasks* from the JobTracker

MapReduce - Overview

- Parallel data processing model for processing and analysing massive amounts of data
- Observation/Motivation: Operations almost always parallel
- Typically run on large clusters of commodity hardware (i.e., cloud)
- Takes care of partitioning of data, scheduling of jobs, and the communication between nodes in the cluster
 - Programmer does not have to worry about that anymore!

MapReduce Model

- Data type: key-value **records**

- **Map** function:

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

- Group all identical K_{inter} values and pass to reducer

- **Reduce** function:

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

MapReduce Model

- **Optional: Combine** function:
 - Same as the reduce function, but executed on the map nodes
 - Can reduce network overhead (see example)
- W.r.t. network overhead: MapReduce tries to schedule jobs close to where the data is stored in GFS/HDFS (replicas at DataNodes in the latter)
- Functions have to be implemented by the coder

MapReduce Model

- Typical problem size:
 - $M = 200,000$ (Map Tasks / # of input partitions)
 - $R = 5,000$ (Reduce Tasks / # of output files)
 - $P = 2,000$ (Processing elements)

Example: Word Count

Input: key is filename, value is a line in input file

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

Intermediate: key is a word, value is 1

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

Word Count Execution

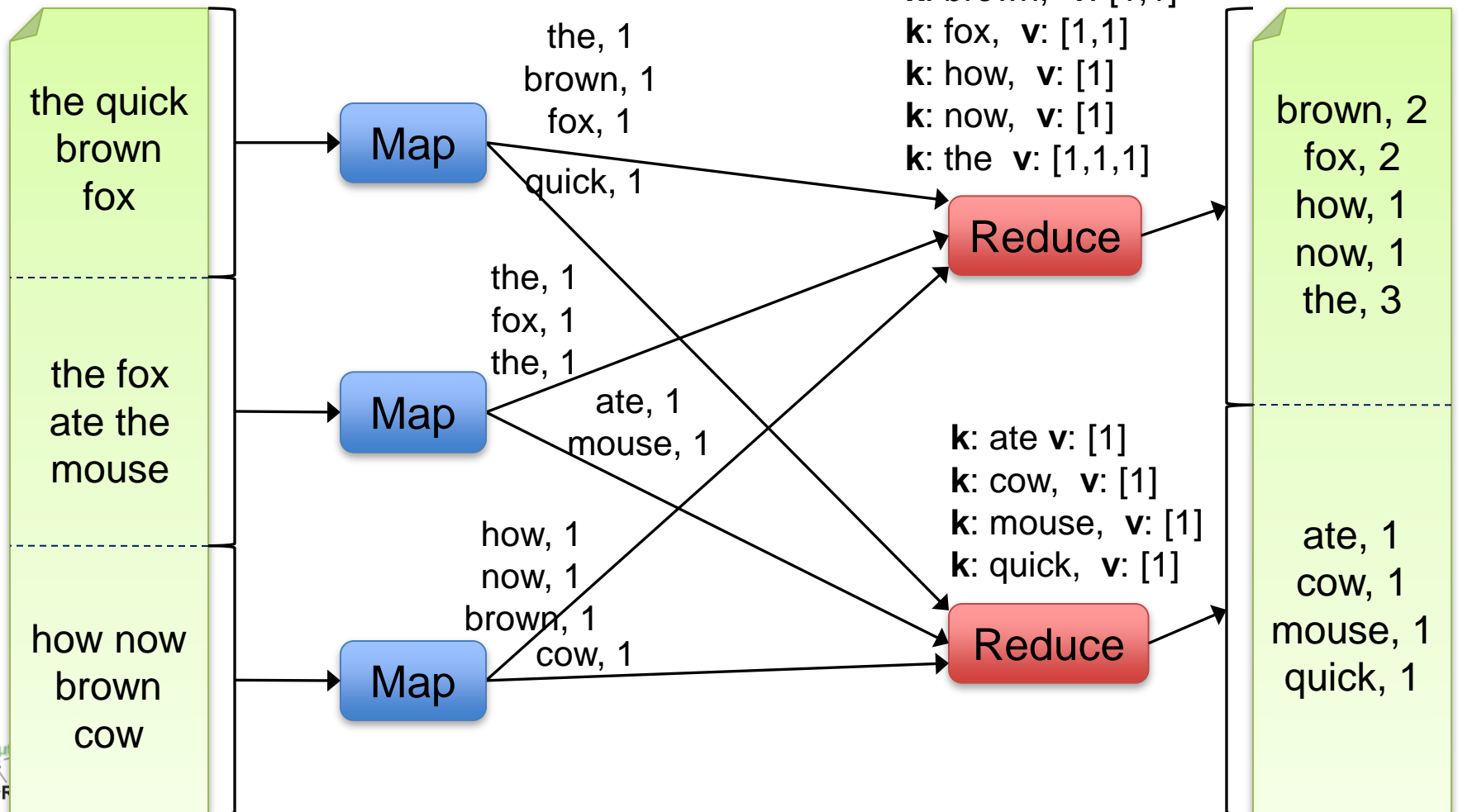
Input

Map

Shuffle & Sort

Reduce

Output



Another Example

- Google Maps: Road intersections?
- Large data sets of geographical data, how to process?
- *Input*: List of roads and intersections
- *Map function*: Creates pairs of connected points (road, intersection) or (road, road)
- *Sort*: sort by key
- *Reduce function*: Get list of pairs with same key
- *Output*: List of all points that connect to a particular road)

What is MapReduce Used For?

- At **Google**:
 - Index building for Google Search
 - Article clustering for Google News
 - Statistical machine translation
- At **Facebook**:
 - Data mining
 - Ad optimization
 - Spam detection
 - 1100 and 300 machines clusters: 0.5 PetaByte of data every day!
- Non-tech companies:
 - *New York Times*: MapReduce used in 100 EC2 instances to convert articles since 1852 (4TB TIFF images) to PDF within 24 hours – cost: \$240

MapReduce Insights

- Restricted key-value model
 - Same **fine-grained operation** (Map & Reduce) repeated on big data
 - Operations must be **deterministic**
 - Operations must be **idempotent/no side effects**
 - Only communication is through the shuffle
 - Operation (Map & Reduce) output saved (on disk)

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

MapReduce Pros

- Automatic **scaling**
 - As operations as side-effect free, they can be distributed to any number of machines dynamically
- Automatic **load-balancing**
 - Move tasks and handle slow tasks (*stragglers*)
 - If MapReduce operation is nearly completed, speculatively execute duplicate copies of stragglers
 - Reasons for straggling: machine shared with other jobs, network congestion, ...

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
 - *Pregel* for graph-processing

PIG / HIVE: Motivation

- Limitation of MR
 - Have to use Map/Reduce model
 - Not Reusable
 - Error prone
 - For complex jobs:
 - Multiple stages of Map/Reduce functions

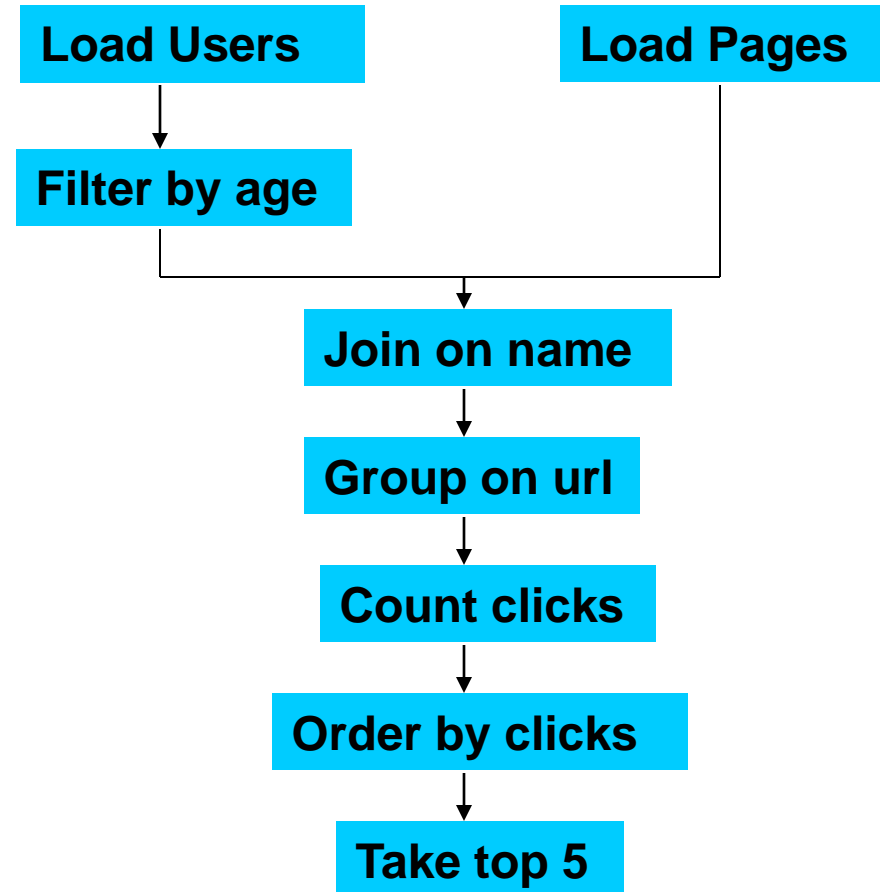
Pig

- High-level language and execution environment:
 - Produces sequences of MapReduce jobs
 - Provides relational (SQL) operators (JOIN, GROUP BY, etc)
 - Easy to plug in Java functions
- Started at Yahoo! Research
 - Runs about 50% of Yahoo!'s jobs



Example Problem

Given *user data* in one file, and *website data* in another, find the *top 5 most visited pages by users aged 18-25*



Example from <http://wiki.apache.org/pig-data/attachments/PigTalksPapers/attachments/ApacheConEurope09.ppt>

In MapReduce

```
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;
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
            // 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.setOutputFormat(TextOutputFormat.class);
    lp.setOutputKeyClass(Text.class);
    lp.setOutputValueClass(LongWritable.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

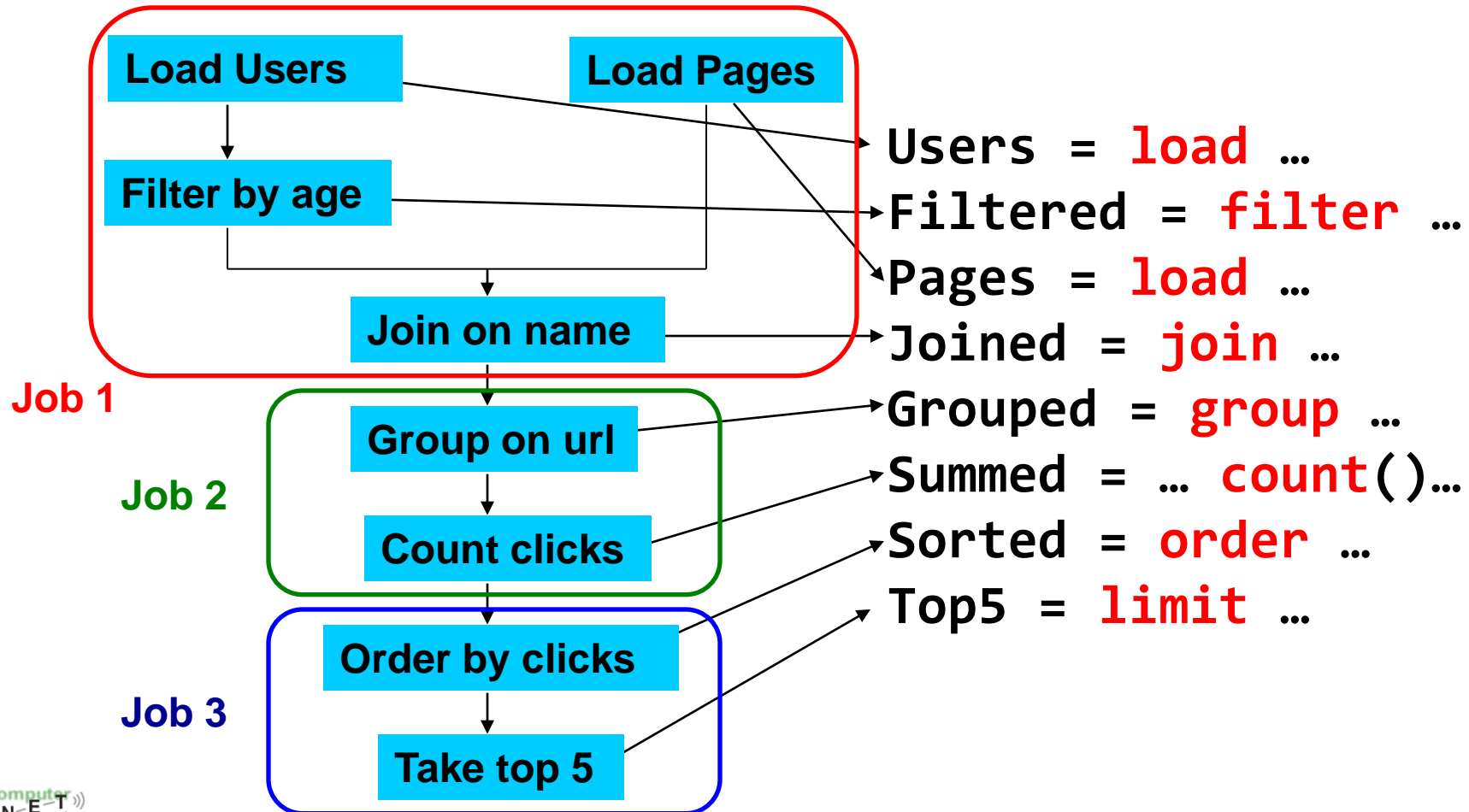
```
Users      = load 'users' as (name, age);
Filtered   = filter Users by
              age >= 18 and age <= 25;
Pages      = load 'pages' as (user, url);
Joined     = join Filtered by name, Pages by user;
Grouped    = group Joined by url;
Summed     = foreach Grouped generate group,
              count(Joined)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

store Top5 into 'top5sites';
```

Example from <http://wiki.apache.org/pig-data/attachments/PigTalksPapers/attachments/ApacheConEurope09.ppt>

Translation to MapReduce

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



Hive [4]

- Relational database built on Hadoop
 - Maintains table schemas
 - SQL-like query language (which can also call Hadoop Streaming scripts, *HiveQL*)
 - Supports table partitioning, complex data types, sampling, some query optimization
- Developed at Facebook



Hive

- Started at Facebook in 2008
- Now used for 95% of Hadoop jobs at Facebook
- ~200 engineers/data analysts use Hive per month



Overview

- Hive is a data warehousing system to store structured data on Hadoop file system
 - Hive Warehouse: 4800 cores,
 - 5.5 PB storage (12TB per node)
- Provides an easy query on these data by executing Hadoop MapReduce *plans*

Overview

- Intuition
 - Make the unstructured data look like tables regardless how it is really layed out
 - SQL based query can be directly against these tables (*HiveQL*)
 - Generate specify *execution plan* for this query

Hive Components



- Interfaces: CLI, GUI, JDBC/ODBC
- Thrift Server: interoperability service (support for multiple languages)
- Driver: Manages lifecycle of a HiveQL query, maintains session
 - Compiler: compiles HiveQL to a sequence of MapReduce tasks
 - Optimizer: optimizes MR task sequence
 - Execution Engine: executes tasks provided by compiler in proper order

Architecture

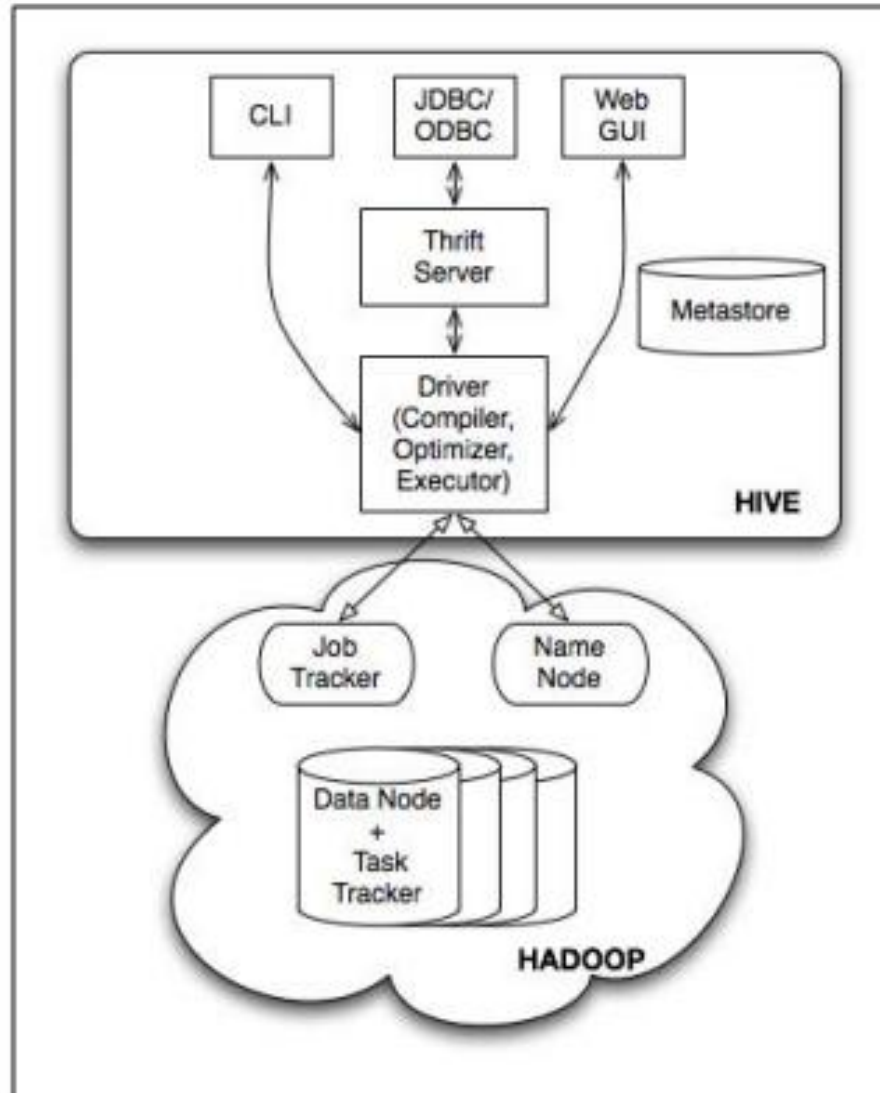


Figure 1: Hive Architecture

Architecture

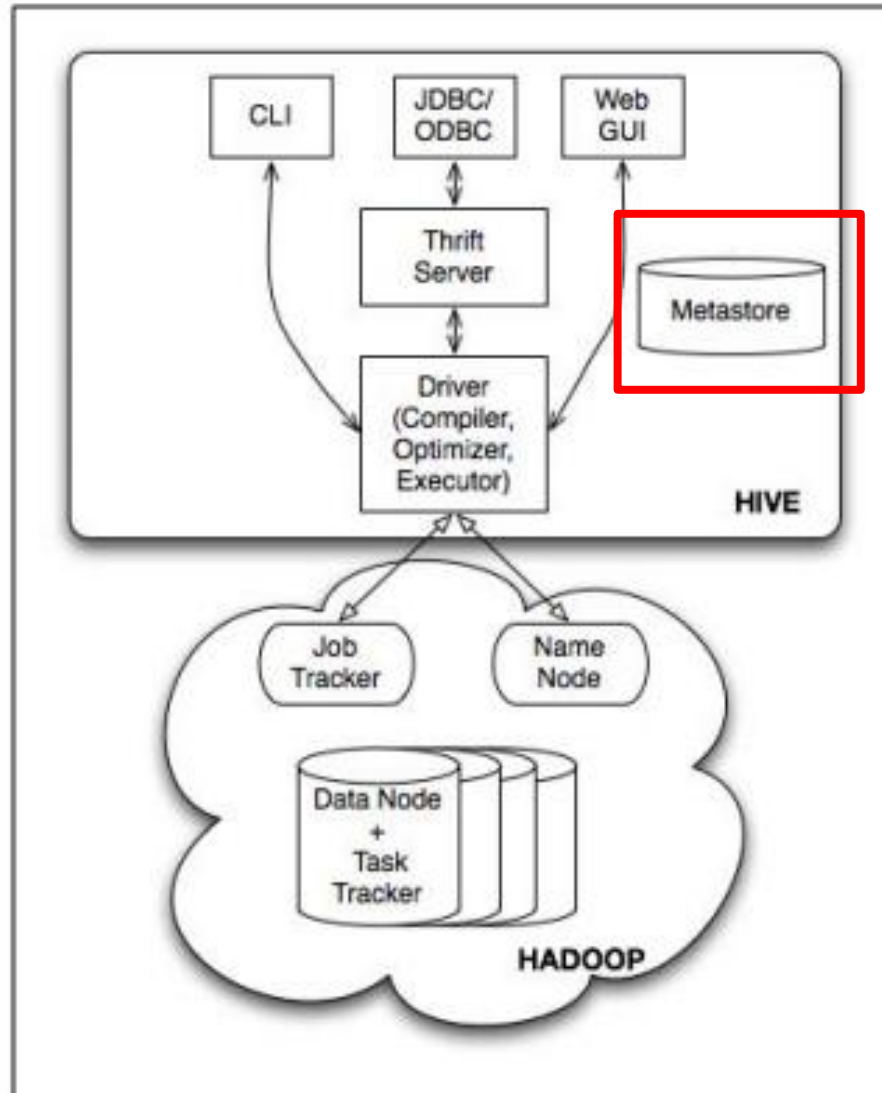
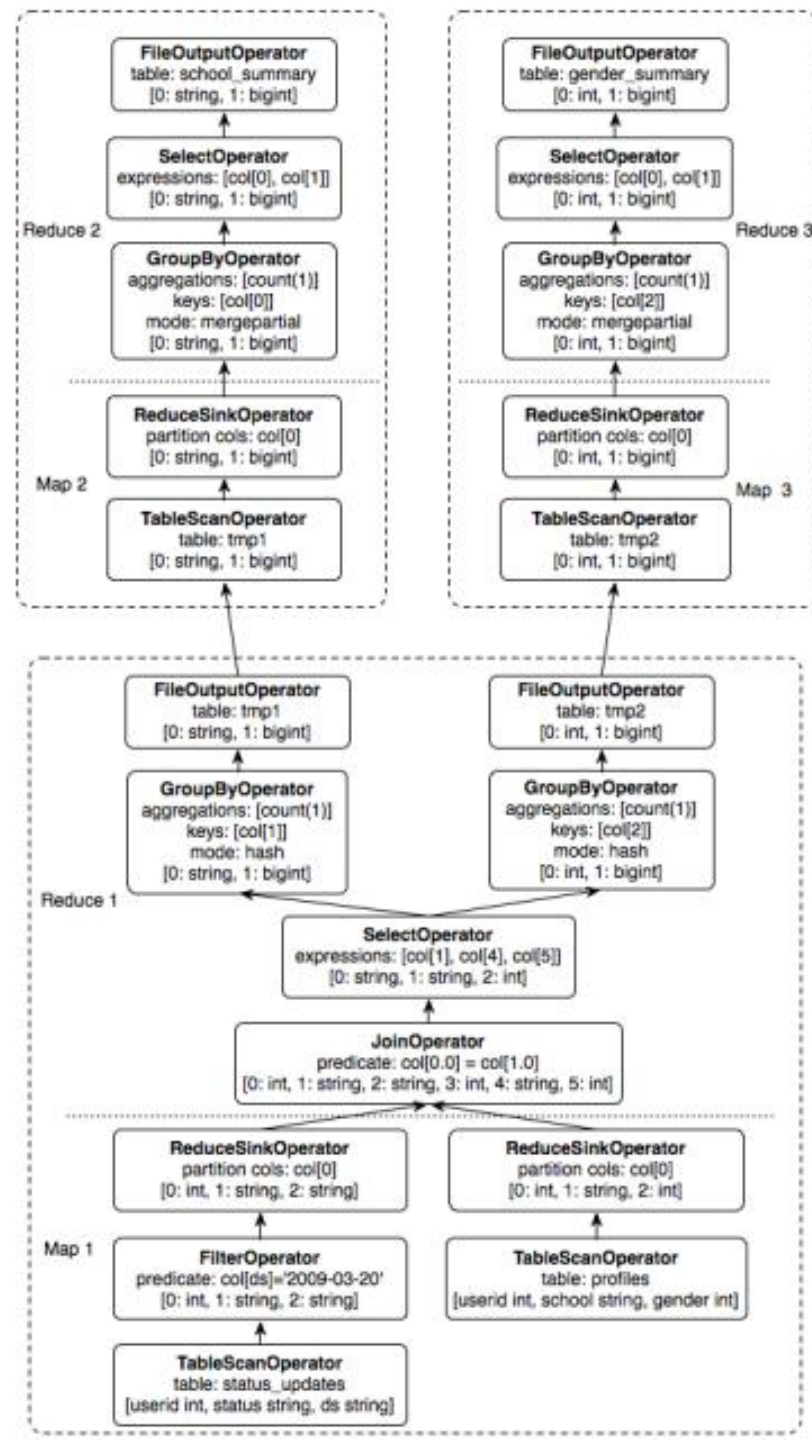


Figure 1: Hive Architecture

Mapping to MapReduce

- Very similar to PIG:
 - Mapping = selection of data from table(s)
 - Reduce: JOIN or GROUP operations



Where to store the tables?

- *Metastore*:
 - System catalog maintaining meta-data about the stored tables
 - Meta-data is created on table creation
 - Contains, for each table:
 - List of columns and their types, owner, storage, serialization/deserialization (SerDe) information, user supplied key/value data
- Note: This information is not accessed sequentially but more or less random
 - HDFS/GFS not an optimal solution, use standard file systems (NTFS, AFS, Oracle,...)

Metadata

- Database namespace
- Table definitions
 - schema info, physical location In HDFS
- Partitions
 - possible to split tables into parts (PARTITION BY)

Execution

- GROUP BY operation
 - Efficient execution plans based on:
 - *Data skew:*
 - how evenly distributed data across a number of physical nodes
 - bottleneck / load balance?
 - *Partial aggregation:*
 - Group the data with the same group by value as soon as possible
 - In memory hash-table for mapper
 - Earlier than combiner

Execution

- JOIN operation
 - Traditional Map-Reduce Join
 - Early Map-side Join
 - very efficient for joining a small table with a large table
 - Keep smaller table data in memory first
 - Join with a chunk of larger table data each time
 - Space complexity for time complexity

Serialization

- Ser/De
 - Describe how to load the data from the file into a representation that make it looks like a table;
- Lazy load
 - Create the field object when necessary
 - Reduce the overhead to create unnecessary objects in Hive
 - Java is expensive to create objects
 - Increase performance

Hive – Performance

Date	SVN Revision	Major Changes	Query A	Query B	Query C
2/22/2009	746906	Before Lazy Deserialization	83 sec	98 sec	183 sec
2/23/2009	747293	Lazy Deserialization	40 sec	66 sec	185 sec
3/6/2009	751166	Map-side Aggregation	22 sec	67 sec	182 sec
4/29/2009	770074	Object Reuse	21 sec	49 sec	130 sec
6/3/2009	781633	Map-side Join *	21 sec	48 sec	132 sec
8/5/2009	801497	Lazy Binary Format *	21 sec	48 sec	132 sec

- QueryA: SELECT count(1) FROM t;
- QueryB: SELECT concat(concat(concat(a,b),c),d) FROM t;
- QueryC: SELECT * FROM t;
- map-side time only (incl. GzipCodec for comp/decompression)
- * These two features need to be tested with other queries.

<http://www.slideshare.net/cloudera/hw09-hadoop-development-at-facebook-hive-and-hdfs>

Hive – Example: CLI

```
hive> SELECT sales.*, things.* FROM sales LEFT OUTER JOIN things ON (sales.id = things.id);
```

```
Total MapReduce jobs = 1
```

```
Launching Job 1 out of 1
```

```
Job 0: Map: 2 Reduce: 1 Cumulative CPU: 6.49 sec HDFS Read: 439 HDFS Write: 63 SUCCESS
```

```
Total MapReduce CPU Time Spent: 6 seconds 490 msec
```

```
OK
```

```
Ali      0      NULL  NULL
```

```
Joe      2      2      Tie
```

```
Hank     2      2      Tie
```

```
Eve      3      3      Hat
```

```
Hank     4      4      Coat
```

```
Time taken: 85.943 seconds
```

```
hive> █
```

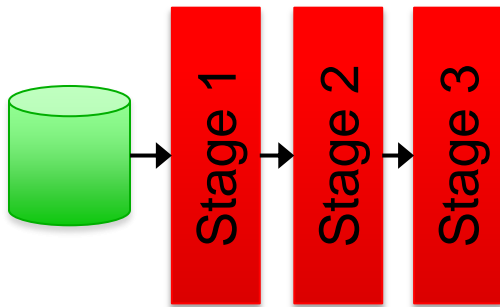
Hive - Limitations

- Not good for unstructured data
- Not good for small datasets (high latency)
- Not comparable to Opera etc. in terms of performance
 - No updates, transactions, etc.
- Performance tradeoff (see previous slide) when compared to MR
 - But keep in mind: easier expression, development effort is reduced!

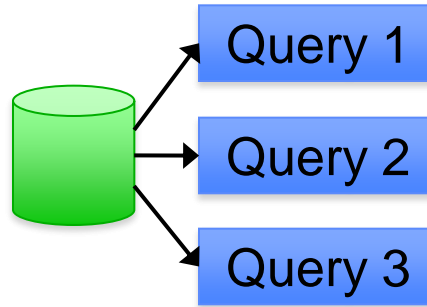
Spark [5]

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

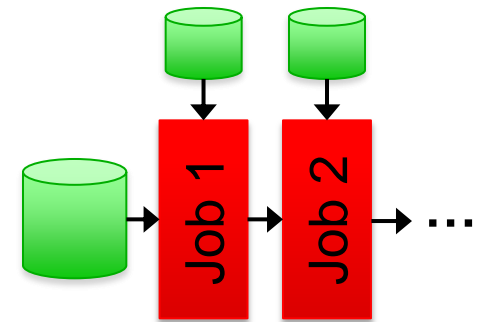
Efficient primitives for **data sharing**



Iterative job



Interactive mining




Stream processing

Spark Motivation

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

Efficient primitives for **data sharing**



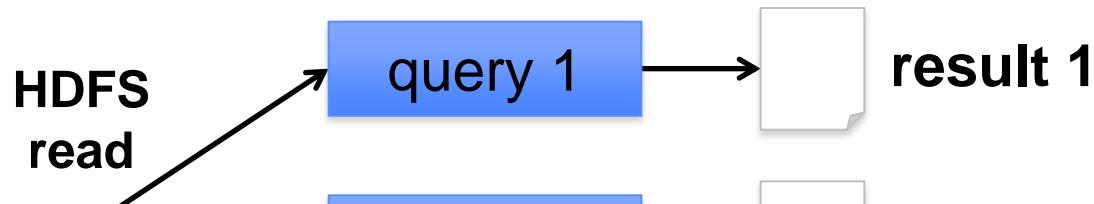
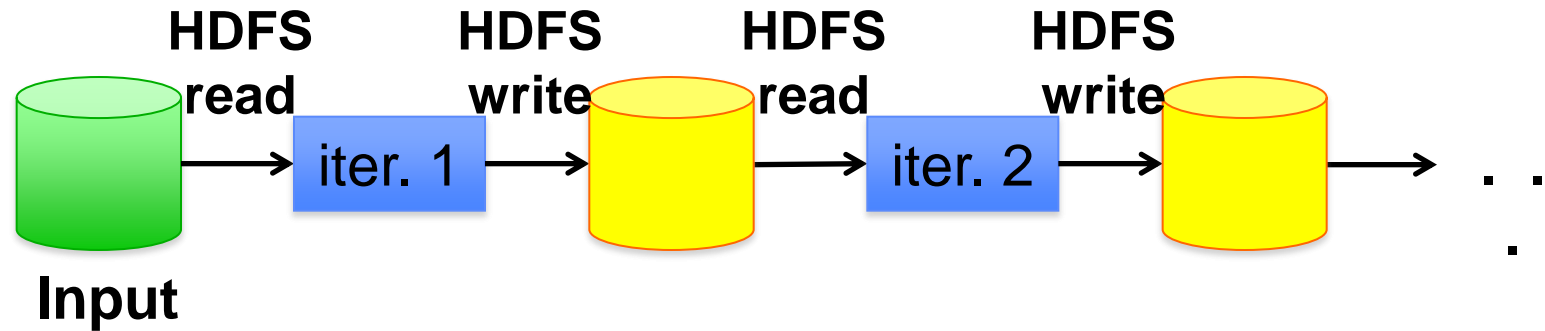
Problem: in MR, the only way to share data across jobs is using stable storage (e.g. file system) → slow!

Iterative job

Interactive mining

Stream processing

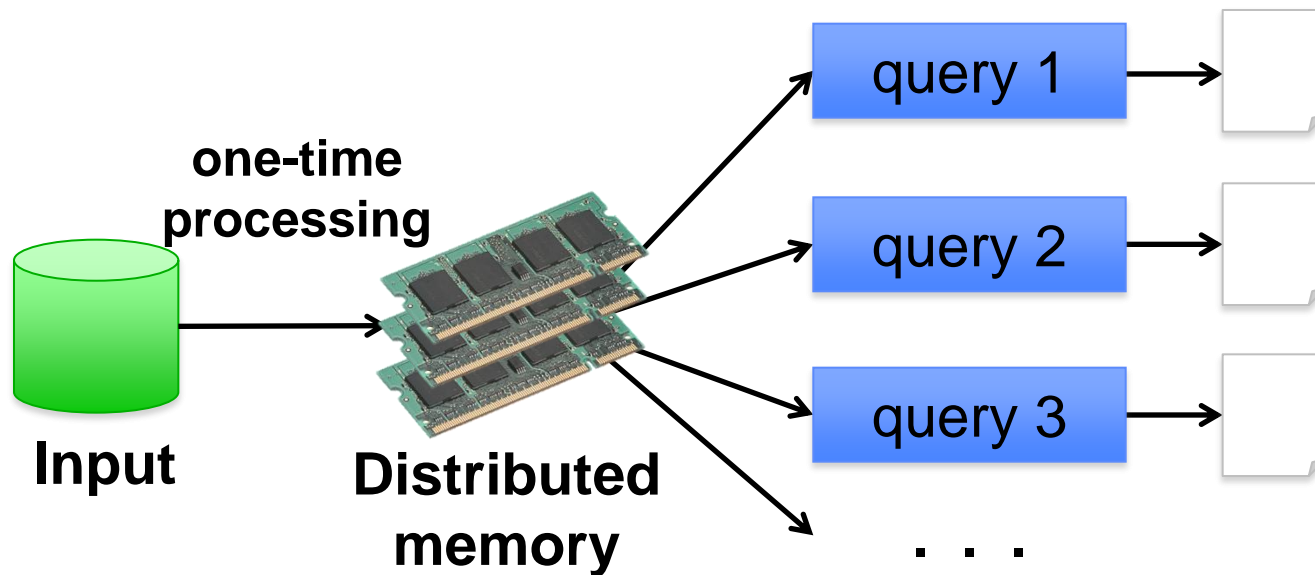
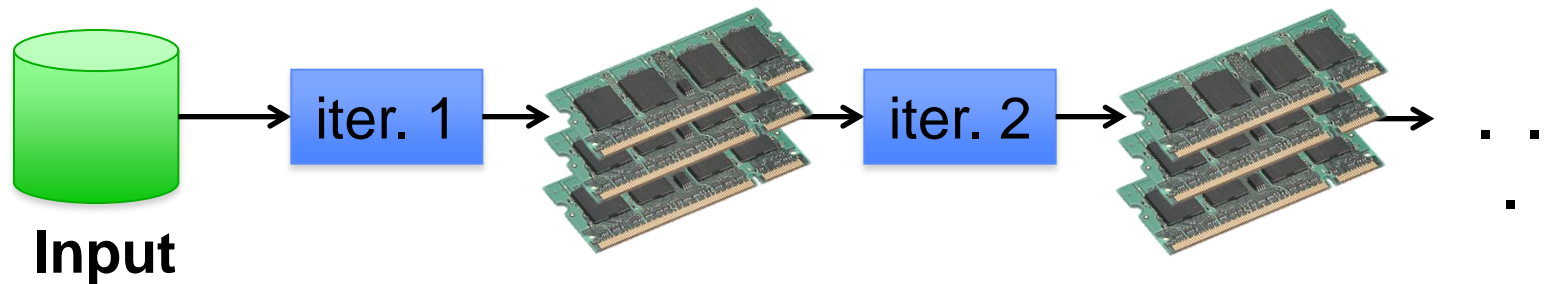
Examples



Opportunity: DRAM is getting cheaper → use main memory for intermediate results instead of disks

...

Goal: In-Memory Data Sharing



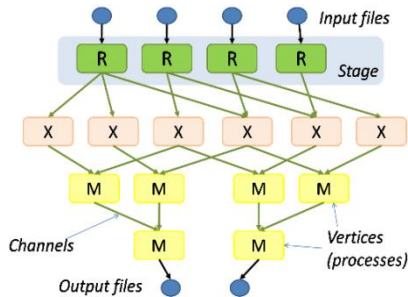
10-100 × faster than network and disk

Spark Solution: Resilient Distributed Datasets (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 (its lineage) to recompute lost data
- <http://www.spark-project.org/>

Datacenter Scheduling Problem

- Rapid innovation in datacenter computing frameworks
- **No single framework optimal for all applications**
- **Want to run multiple frameworks in a single datacenter**
 - ...to maximize utilization
 - ...to share data between frameworks



CIEL

S4 distributed stream computing platform



Percolator

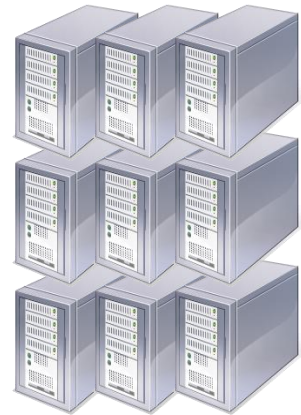
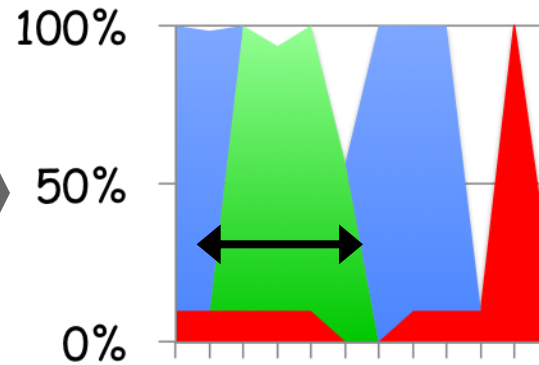
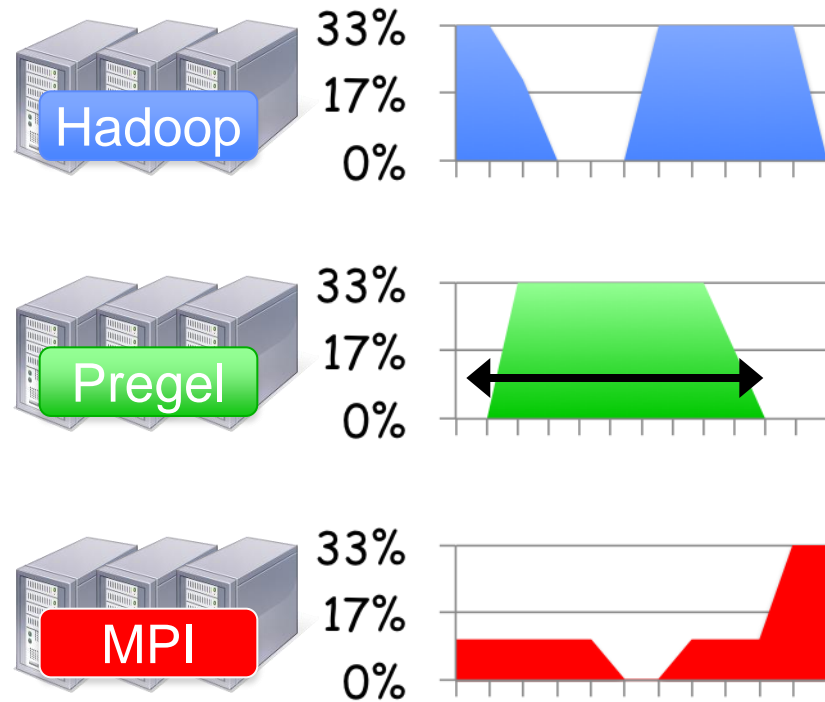


Dryad

Where We Want to Go

Today: static partitioning

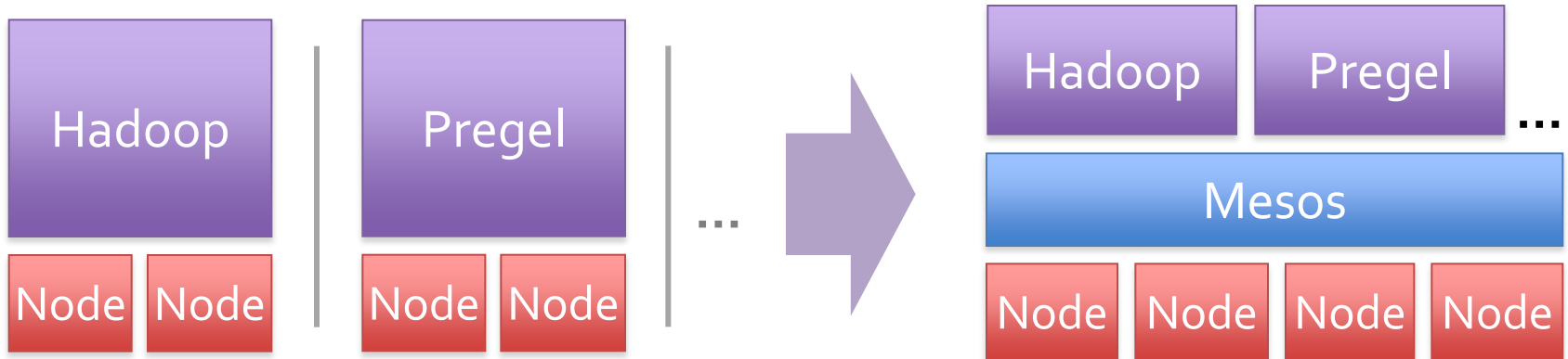
Dynamic sharing



Shared cluster

Solution: Apache Mesos [6]

- Mesos is a common resource sharing layer over which diverse frameworks can run



- Run multiple instances of the *same* framework
 - Isolate production and experimental jobs
 - Run multiple versions of the framework concurrently
- Build *specialized frameworks* targeting particular problem domains
 - Better performance than general-purpose abstractions

Mesos Goals

- **High utilization** of resources
- **Support diverse frameworks** (current & future)
- **Scalability** to 10,000's of nodes
- **Reliability** in face of failures

<http://incubator.apache.org/mesos/>

Resulting design: Small microkernel-like core that pushes scheduling logic to frameworks

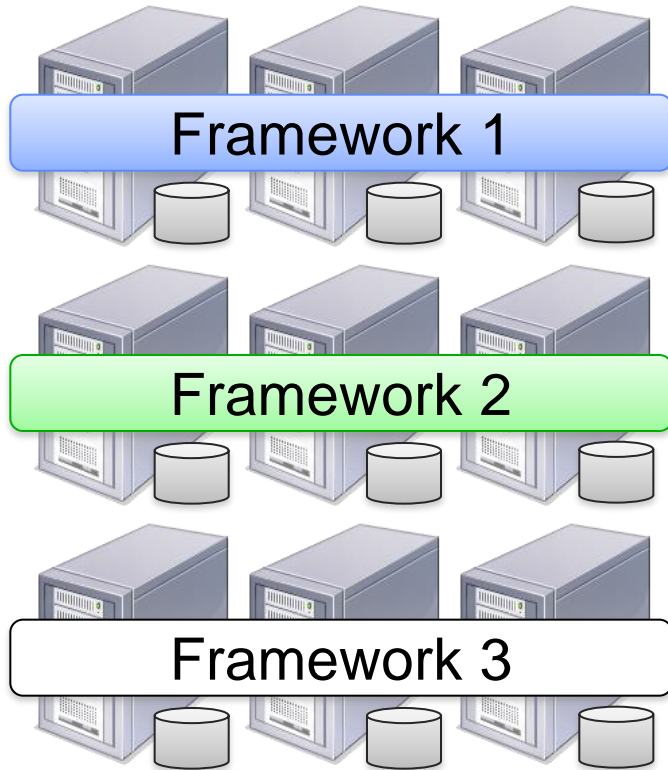
Mesos Design Elements

- Fine-grained sharing:
 - Allocation at the level of *tasks* within a job
 - Improves utilization, latency, and data locality

- Resource offers:
 - Simple, scalable application-controlled scheduling mechanism

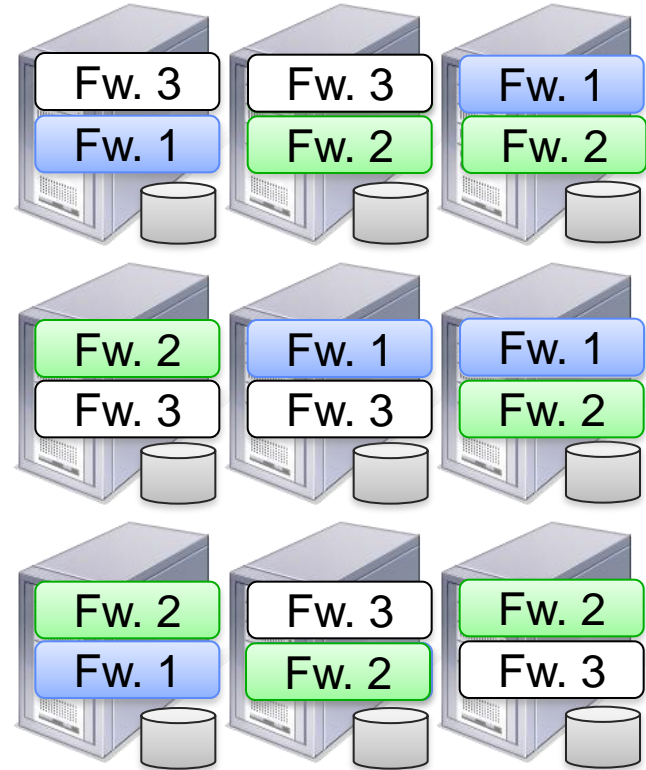
Element 1: Fine-Grained Sharing

Coarse-Grained Sharing (HPC):



Storage System (e.g. HDFS)

Fine-Grained Sharing (Mesos):



Storage System (e.g. HDFS)

+ Improved utilization, responsiveness, data locality

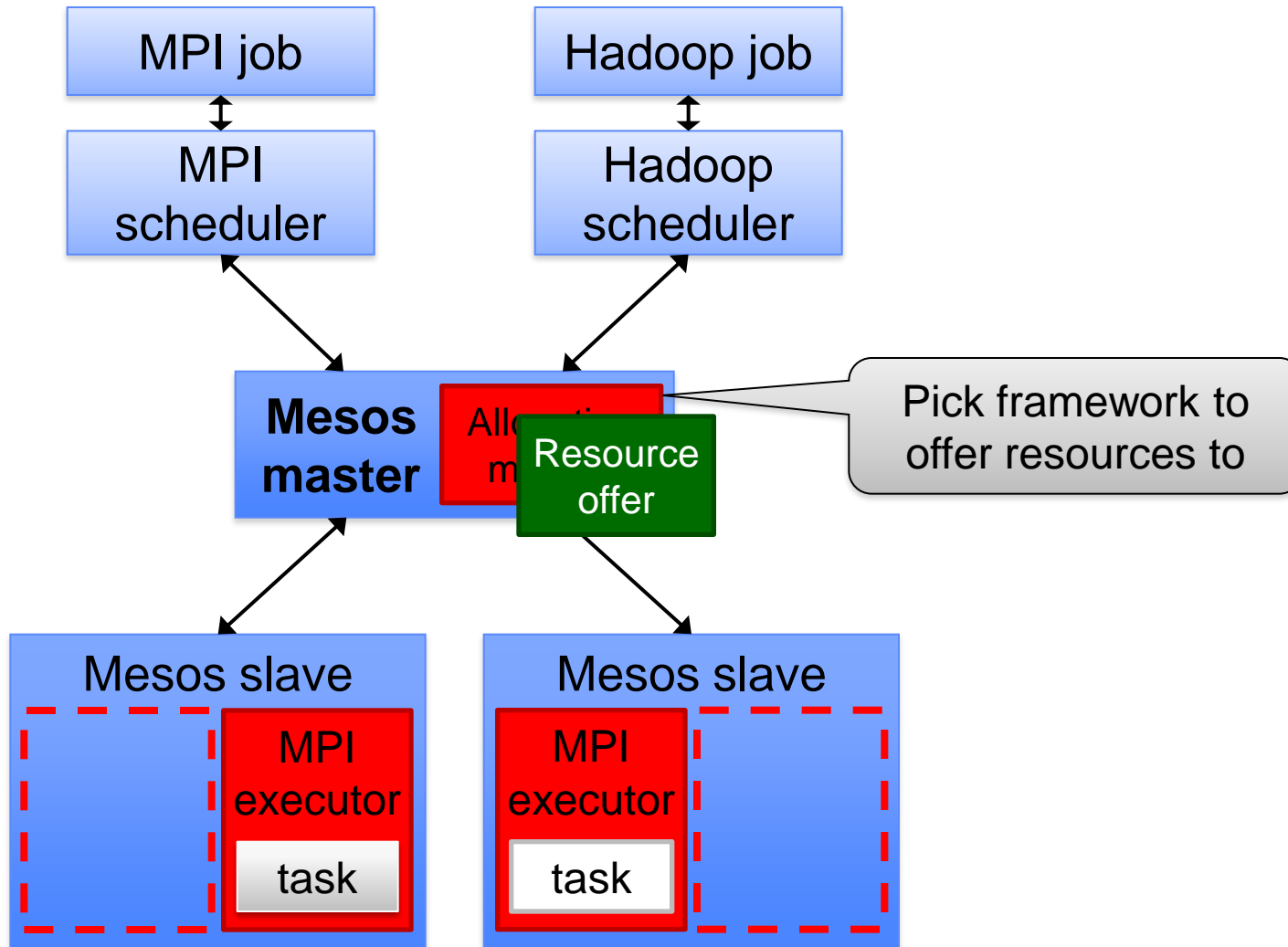
Element 2: Resource Offers

- Option: Global scheduler
 - Frameworks express needs in a specification language, global scheduler matches them to resources
 - + Can make optimal decisions
 - Complex: language must support all framework needs
 - Difficult to scale and to make robust
 - Future frameworks may have unanticipated needs

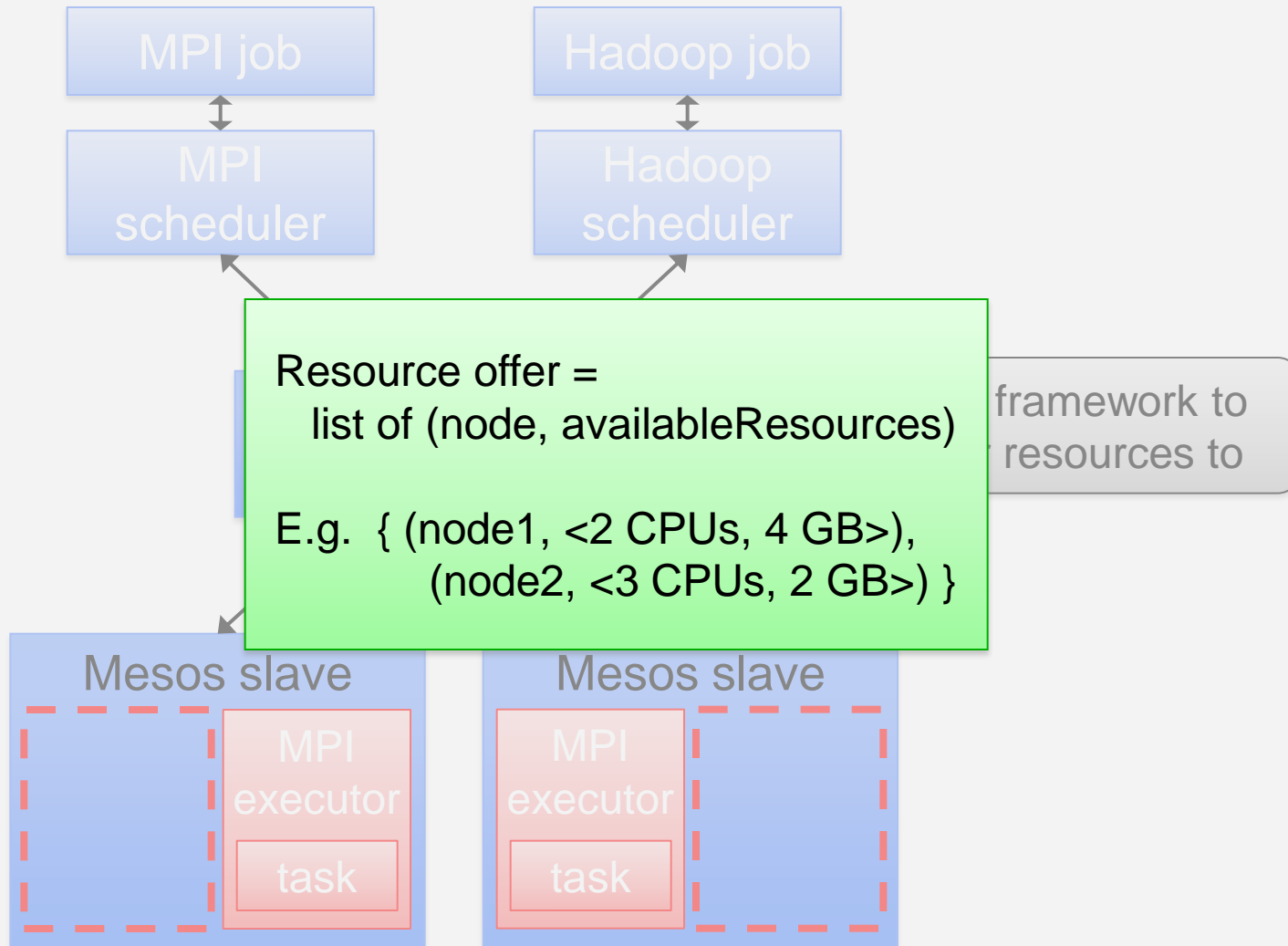
Element 2: Resource Offers

- Mesos: Resource offers
 - Offer available resources to frameworks, let them pick which resources to use and which tasks to launch
 - + Keeps Mesos simple, lets it support future frameworks
 - Decentralized decisions might not be optimal

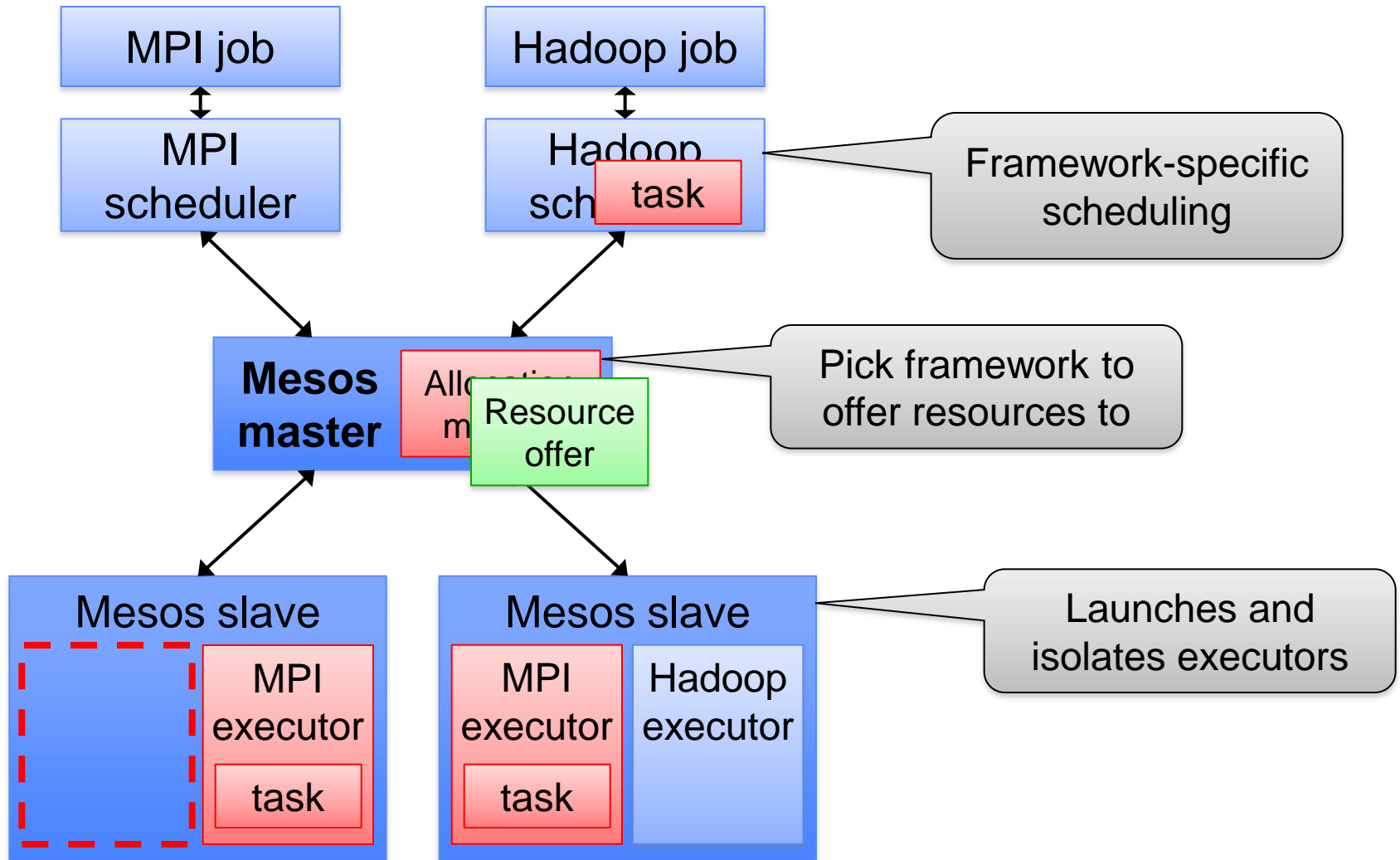
Mesos Architecture



Mesos Architecture



Mesos Architecture



Summary

- Cloud computing/datacenters are the new computer
 - Emerging “Datacenter/Cloud Operating System” appearing
 - Main concepts: virtualization, load balancing, ...
- Pieces of the DC/Cloud OS
 - Networks (SDN, OpenFlow)
 - High-throughput filesystems (GFS/HDFS)
 - Job frameworks (MapReduce, Spark, Pregel)
 - High-level query languages (Pig, Hive)
 - Cluster scheduling (Apache Mesos)

References

- [1] S. Ghemawat, H. Gobioff and S. Leung. The Google File System, Proceedings of the ACM SOSP 2003
- [2] K. Shvachko, H. Kuang, S. Radia and R. Chansler. The Hadoop Distributed File System, Proceedings of the 26th IEEE Symposium on Mass Storage Systems and Technologies
- [3] Olston, Christopher, et al. "Pig latin: a not-so-foreign language for data processing." Proceedings of the 2008 ACM SIGMOD international conference on Management of data. ACM, 2008.
- [4] Thusoo, Ashish, et al. "Hive: a warehousing solution over a map-reduce framework." *Proceedings of the VLDB Endowment* 2.2 (2009): 1626-1629.
- [5] Zaharia, Matei, et al. "Spark: cluster computing with working sets." *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*. 2010.
- [6] Hindman, Benjamin, et al. "Mesos: A platform for fine-grained resource sharing in the data center." *Proceedings of the 8th USENIX NSDI*. 2011.
- [7] Dean, Jeffrey, and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters." *Communications of the ACM* 51.1 (2008): 107-113.