College of Computing

CS 8803 /CS 4365 Big Data Systems and Analytics

Big Data Processing Platforms

Ling Liu

Professor
School of Computer Science
College of Computing, Georgia Tech

Three Most Popular Big Data Processing Platforms

- Hadoop
  - Hadoop Distributed File System (HDFS)
  - Hadoop YARN
  - Hadoop MapReduce
- SPARK
- MPI
  - **MPICH** is a high performance and widely portable implementation of the **Message Passing Interface (MPI)** standard.

Hadoop MapReduce

- Cluster Computing and Hadoop Architecture
- MapReduce Programming Model and Execution
- Fault tolerance in MapReduce
- Sample applications
- Getting started with Hadoop
- Higher-level languages over Hadoop: Pig and Hive

Inherent Problems and Challenges

Potential Topics for Course Projects

What is MapReduce?

- Data-parallel programming model for clusters of commodity machines
- Pioneered by Google
  - Processes 20 PB of data per day on an average cluster of 400 nodes in 2007
- Popularized in 2007-2008 by open-source Hadoop projects
  - Used by Yahoo!, Facebook, Amazon, …

Outline

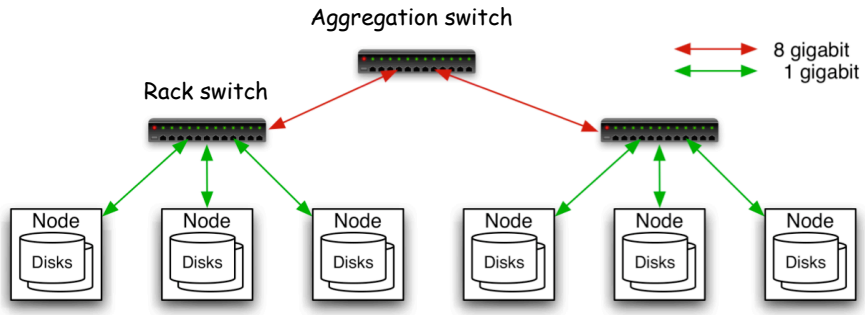
- MapReduce architecture
- Sample applications
- Getting started with Hadoop
- Higher-level languages on top of Hadoop: Pig and Hive
- MapReduce Runtime (Execution Engine)
- Fault tolerance in MapReduce

MapReduce Design Goals

1. **Scalability** to large data volumes:
   - Scan 100 TB on 1 node @ 50 MB/s = 23 days
   - Scan on 1000-node cluster = 33 minutes

2. **Cost-efficiency:**
   - Commodity nodes (cheap, but unreliable)
   - Commodity network
   - Automatic fault-tolerance (fewer admins)
   - Easy to use (fewer programmers)

Typical Hadoop Cluster



- 40 nodes/rack, 1000-4000 nodes in cluster
- 1 Gbps bandwidth within rack, 8 Gbps out of rack
- Node specs (Yahoo terasort):
  8 x 2GHz cores, 8 GB RAM, 4 disks (= 4 TB?)

Image from http://wiki.apache.org/hadoop-data/attachments/HadoopPresentations/attachments/YahooHadoopIntro-apachecon-us-2008.pdf

Typical Hadoop Cluster

Image from http://wiki.apache.org/hadoop-data/attachments/HadoopPresentations/attachments/aw-apachecon-eu-2009.pdf

Cluster Computing: Challenges

1. **Cheap nodes fail, especially if 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 into system

2. **Commodity network = low bandwidth**
   - Solution: Push computation to the data

3. **Programming distributed systems is hard**
   - Solution: Data-parallel programming model: users write "map" & "reduce" functions, system distributes work and handles faults

Cluster Computing Platforms

- **Hadoop Distributed Computing Platform**
  - HDFS
  - Single namespace for entire cluster
  - Replicates data 3x for fault-tolerance

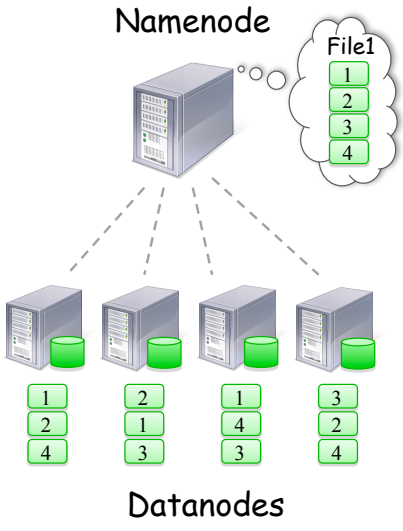
- **Spark Distributed Computing Platform**
  - In Memory Computing
  - Streaming Distributed Computing
  - Run on external distributed file system, e.g., HDFS

Hadoop Components

- **Distributed file system (HDFS)**
  - Single namespace for entire cluster
  - Replicates data 3x for fault-tolerance
- **MapReduce Programming framework**
  - Executes user jobs specified as "map" and "reduce" functions
  - Manages work distribution & fault-tolerance

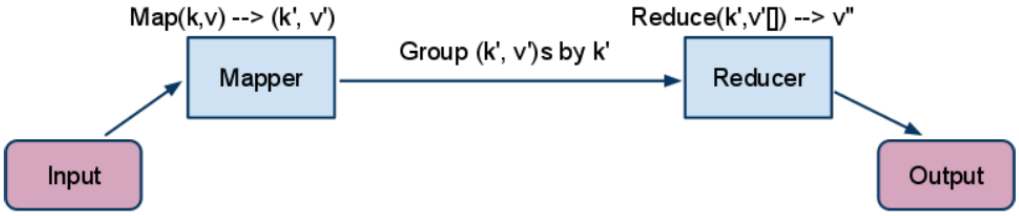
Hadoop Distributed File System

- Files split into 128MB *blocks*
- Blocks replicated across several *datanodes* (usually 3 → Triple Modularity)
- Single *namenode* stores metadata (file names, block locations, etc)
- Optimized for large files, sequential reads
- Files are append-only



MapReduce

- A simple programming model for processing large dataset on a large computer cluster
- Map/Reduce works like a Unix pipeline:
  - cat input | grep | sort | uniq -c | cat > output



- Focus on problem, and let the library deal with the messy detail

MapReduce Programming Model

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

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

- Reduce function:

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

MapReduce Programming

- Programmers focus on programming your jobs using Map tasks and Reduce tasks
- MapReduce system takes care of
  - distributing data and code across the compute cluster;
  - scaling MapReduce program to big data

Warm up: Word Count

- We have a large file of words, one word per line
- Count the number of times each distinct word appears in the file
- *Sample application*: analyze web server logs to find popular URLs

Word Count: Three scenarios

- Case 1: Entire file fits in memory
- Case 2: File too large for mem, but all <word, count> pairs fit in mem
- Case 3: File on disk, too many distinct words to fit in memory
  - **`sort datafile | uniq -c`**

Example: Word Count

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

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

Word Count using MapReduce

```
map(key, value):
// key: document name; value: text of
   document
   for each word w in value:
       emit(w, 1)

reduce(key, values):
// key: a word; value: an iterator over counts
       result = 0
       for each count v in values:
             result += v
       emit(result)
```

See Word Count Map Reduce in Java/Python on slides 50-53

Word Count in Java (Map)

```
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 in Java (Reduce)

```
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 in Java (Main Prog)

```
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);
 }
```

Word Count in Python with Hadoop Streaming

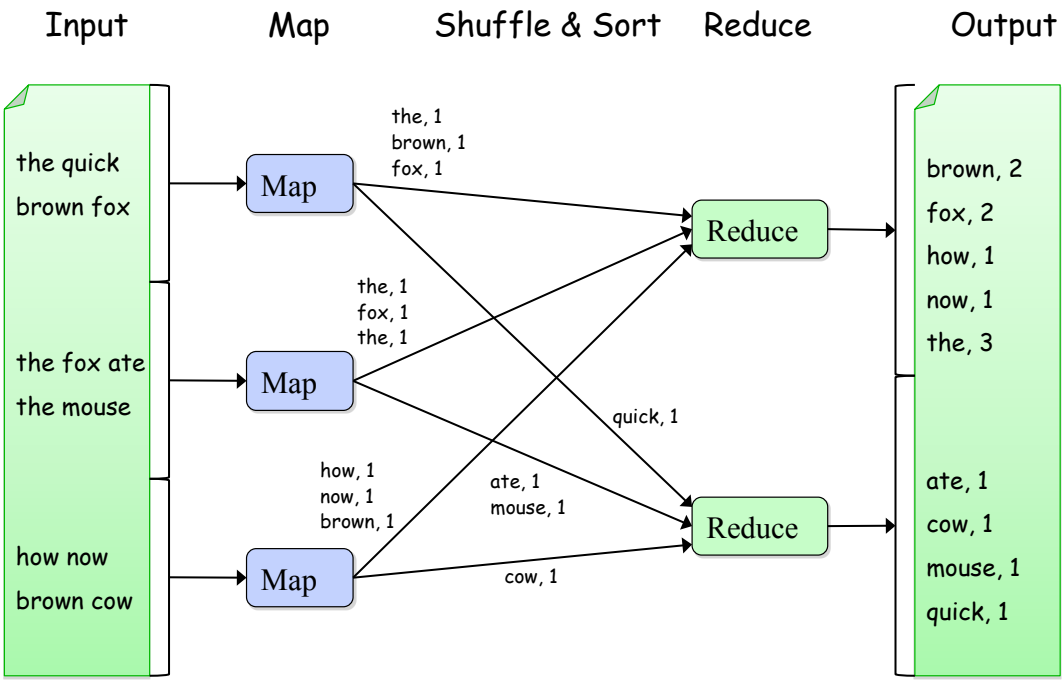
Mapper.py:

```
import sys
for line in sys.stdin:
  for word in line.split():
    print(word.lower() + "\t" + 1)
```

Reducer.py:

```
import sys
counts = {}
for line in sys.stdin:
  word, count = line.split("\t")
    dict[word] = dict.get(word, 0) + int(count)
for word, count in counts:
  print(word.lower() + "\t" + 1)
```


Word Count Execution



Outline

- MapReduce architecture
- Sample applications
- Getting started with Hadoop
- Higher-level languages on top of Hadoop: Pig and Hive
- MapReduce Runtime (Execution Engine)
- Fault tolerance in MapReduce

Summary (cont.)

- Map/Reduce
  - Programming model for organizing computations
  - an OS for using clusters (Cloud Computing)

	Master	Slave
MapReduce	jobtracker	tasktracker
DFS	namenode	datanode

- **Design weakness of current Hadoop MapReduce**
  - Memory Utilization
  - Configuration Management

1. (Parallel) Search

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

- **Map:**

```
if(line matches pattern):
   output(line)
```

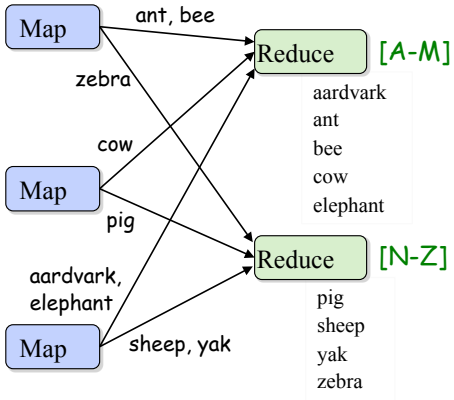
- **Reduce:** identify function
  - Alternative: no reducer (map-only job)

2. (Parallel) Sort

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

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

- **Trick:** Pick partitioning function h such that $k_1<k_2 \Rightarrow h(k_1)<h(k_2)$



3. Inverted Index

Input: (filename, text) records

Output: the list of files containing each word

Map:

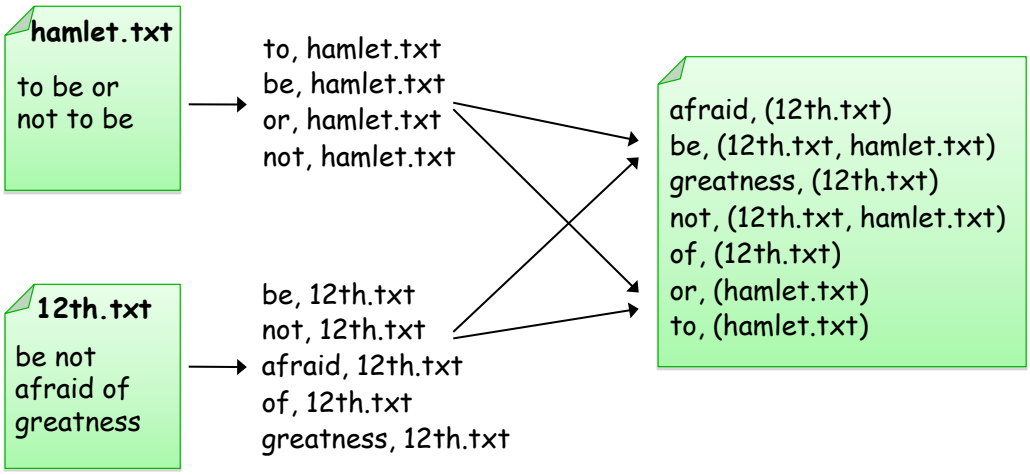
```
foreach word in text.split():
    output(word, filename)
```

Combine: uniquify filenames for each word

Reduce:

```
def reduce(word, filenames):
output(word, sort(filenames))
```


Inverted Index Example



4. Most Popular Words

- **Input:** (filename, text) records
- **Output:** top 100 words occurring in the 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
  - Count files in job 1's reducer rather than job 2's mapper
  - Estimate count distribution in advance and drop rare words

Terabyte Sort Benchmark

- Started by Jim Gray at Microsoft in 1998
- Sorting 10 billion 100 byte records
  - The sort benchmark specifies the input data (10 billion 100 byte records), which must be completely sorted and written to disk.
- Previous records was 297 seconds
- Hadoop won the general category in 209 seconds
  - 910 nodes
  - 2 quad-core Xeons @ 2.0Ghz / node
  - 4 SATA disks / node
  - 8 GB ram / node
  - 1 gb ethernet / node
  - 40 nodes / rack
  - 8 gb ethernet uplink / rack
- The sort used 1800 maps and 1800 reduces and allocated enough memory to buffers to hold the intermediate data in memory.
- Only hard parts were:
  - Getting a total order
  - Converting the data generator to map/reduce

Outline

- MapReduce architecture
- Sample applications
- Getting started with Hadoop
- Higher-level languages on top of Hadoop: Pig and Hive
- MapReduce Runtime (Execution Engine)
- Fault tolerance in MapReduce

Getting Started with Hadoop

- Download from hadoop.apache.org
- To install locally, unzip and set `JAVA_HOME`
- Details: hadoop.apache.org/core/docs/current/quickstart.html

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

Programming Assignment 1

Word Count in Java (Map)

```
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 in Java (Reduce)

```
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 in Java (Main Prog)

```
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);
 }
```


Word Count in Python with Hadoop Streaming

Mapper.py:

```
import sys
for line in sys.stdin:
  for word in line.split():
    print(word.lower() + "\t" + 1)
```

Reducer.py:

```
import sys
counts = {}
for line in sys.stdin:
  word, count = line.split("\t")
    dict[word] = dict.get(word, 0) + int(count)
for word, count in counts:
  print(word.lower() + "\t" + 1)
```

Homework & Project

- Homework
  - Programming Assignment:
    - Install Hadoop MapReduce on your laptop, run its wordcount MR program and report runtime for two sizes of datasets.
  - Reading Assignment: read two papers in the topic area

- **Interesting Projects:**
  - How to solve real world problems using Hadoop MapReduce Cloud/architecture (Application Level)
  - How to improve the performance of MapReduce program execution (Distributed OS/Middleware Level)

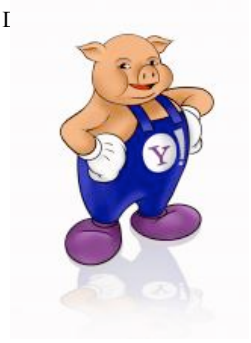
Outline

- MapReduce architecture
- Sample applications
- Getting started with Hadoop
- Higher-level languages on top of Hadoop: Pig and Hive
- MapReduce Runtime (Execution Engine)
- Fault tolerance in MapReduce

Higher-level languages over Hadoop: Pig and Hive (Motivation)

- Many parallel algorithms can be expressed by a series of MapReduce jobs
- But MapReduce is fairly low-level: must think about keys, values, partitioning, etc
- Can we capture common "job building blocks"?

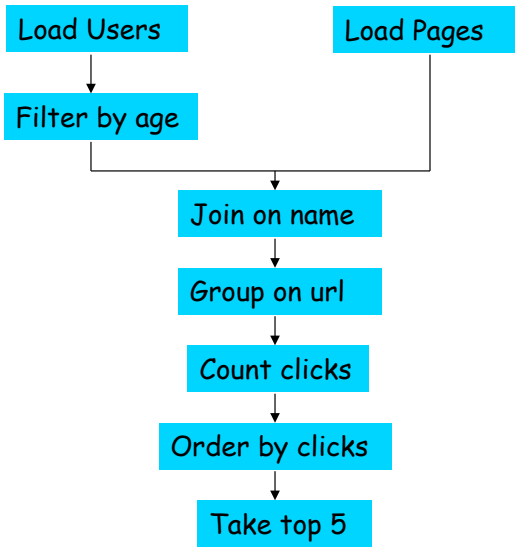
Pig



- Started at Yahoo! Research
- Runs about 30% 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 Pen development environment for Eclipse

An Example Problem

Suppose you have user data in one file, page view data in another, and you need to 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 Pig Latin

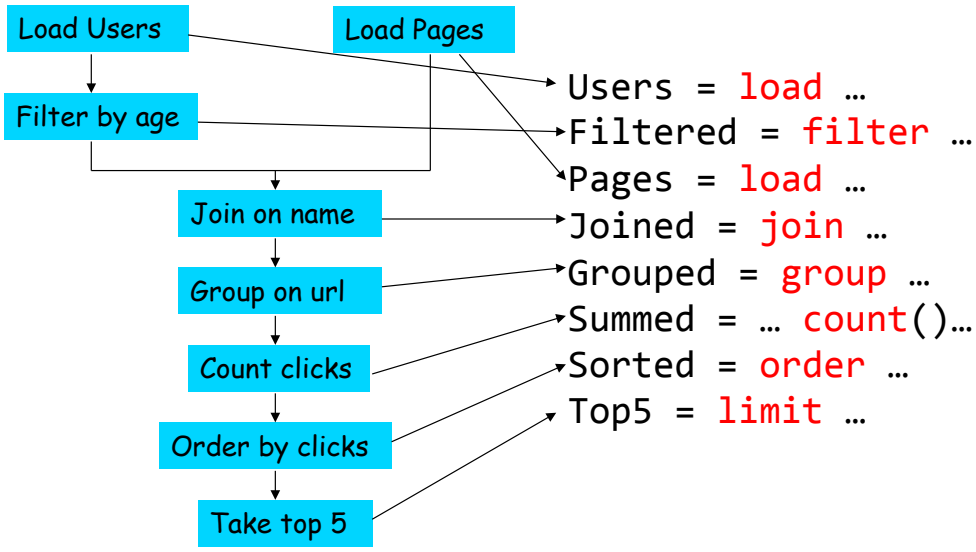
```
Users    = load 'users' as (name, age);
Filtered = filter Users by
                  age >= 18 and age <= 25;
Pages    = load 'pages' as (user, url);
Joined   = join Filtered by name, Pages by user;
Grouped  = group Joined by url;
Summed   = foreach Grouped generate group,
                   count(Joined)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

Ease of Translation

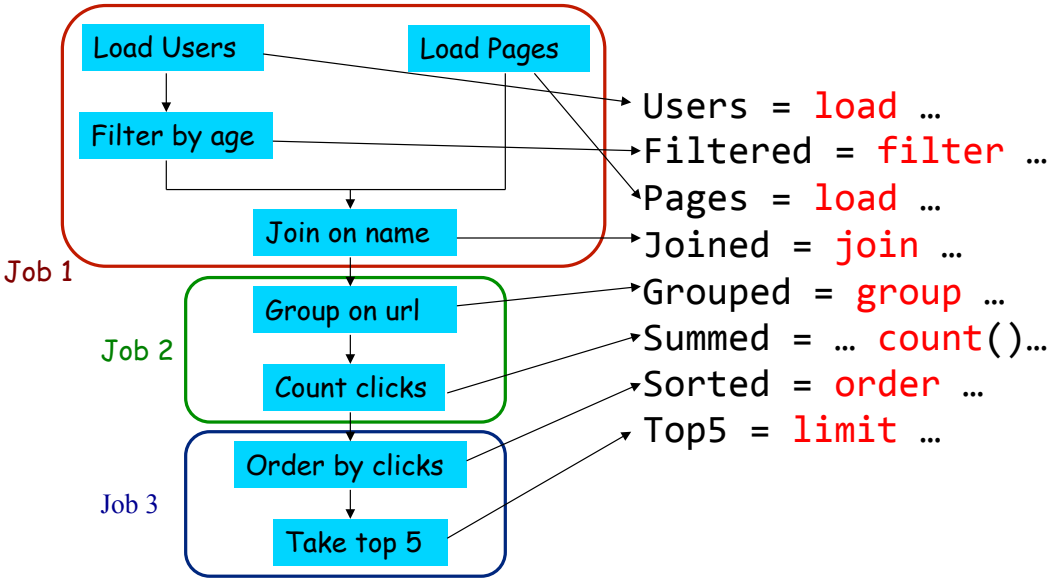
Notice how naturally the components of the job translate into Pig Latin.



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

Ease of Translation

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



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.Job;
import org.apache.hadoop.mapred.jobcontrol.JobC
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.toSt
                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 ke
                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_pag
        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.cla
        FileInputFormat.addInputPath(lfu, new
Path("/user/gates/users"));
        FileOutputFormat.setOutputPath(lfu,
                new Path("/user/gates/tmp/filtered_us
        lfu.setNumReduceTasks(0);
        Job loadUsers = new Job(lfu);

        JobConf join = new JobConf(MRExample.class);
        join.setJobName("Join Users and Pages");
        join.setInputFormat(KeyValueTextInputForm
        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(KeyValueTextInputFor
        group.setOutputKeyClass(Text.class);
        group.setOutputValueClass(LongWritable.cl
        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.cl
        top100.setJobName("Top 100 sites");
        top100.setInputFormat(SequenceFileInputFo
        top100.setOutputKeyClass(LongWritable.cla
        top100.setOutputValueClass(Text.class);
        top100.setOutputFormat(SequenceFileOutput
        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
18 to 25");
        jc.addJob(loadPages);
        jc.addJob(loadUsers);
        jc.addJob(joinJob);
        jc.addJob(groupJob);
        jc.addJob(limit);
        jc.run();
    }
}
```

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

Hive

- Developed at Facebook
- Used for majority of Facebook jobs
- "Relational database" built on Hadoop
  - Maintains list of table schemas
  - SQL-like query language (HQL)
  - Can call Hadoop Streaming scripts from HQL
  - Supports table partitioning, clustering, complex data types, some optimizations

Creating a Hive Table

```
CREATE TABLE page_views(viewTime INT, userid BIGINT,
                    page_url STRING, referrer_url STRING,
                    ip STRING COMMENT 'User IP address')
COMMENT 'This is the page view table'
PARTITIONED BY(dt STRING, country STRING)
STORED AS SEQUENCEFILE;
```

- Partitioning breaks table into separate files for each (`dt, country`) pair

  Ex: `/hive/page_view/dt=2008-06-08,country=US`
  `/hive/page_view/dt=2008-06-08,country=CA`

Simple Query

- Find all page views coming from xyz.com during March (or on March 31st):

```
SELECT page_views.*
FROM page_views
WHERE page_views.date >= '2008-03-01'
AND page_views.date <= '2008-03-31'
AND page_views.referrer_url like '%xyz.com';
```

- Hive only reads partition `2008-03-01,*` (March 2008 data) instead of scanning entire table

Aggregation and Joins

- Count users who visited each page by gender:

```
SELECT pv.page_url, u.gender, COUNT(DISTINCT u.id)
FROM page_views pv JOIN user u ON (pv.userid = u.id)
GROUP BY pv.page_url, u.gender
WHERE pv.date = '2008-03-03';
```

- Sample output:

page_url	gender	count(userid)
home.php	MALE	12,141,412
home.php	FEMALE	15,431,579
photo.php	MALE	23,941,451
photo.php	FEMALE	21,231,314

Sample Hive Queries

Find top 5 pages visited by users aged 18-25:

```
SELECT p.url, COUNT(1) as clicks
FROM users u JOIN page_views p ON (u.name = p.user)
WHERE u.age >= 18 AND u.age <= 25
GROUP BY p.url
ORDER BY clicks
LIMIT 5;
```

Filter page views through Python script:

```
SELECT TRANSFORM(p.user, p.date)
USING 'map_script.py'
AS dt, uid CLUSTER BY dt
FROM page_views p;
```

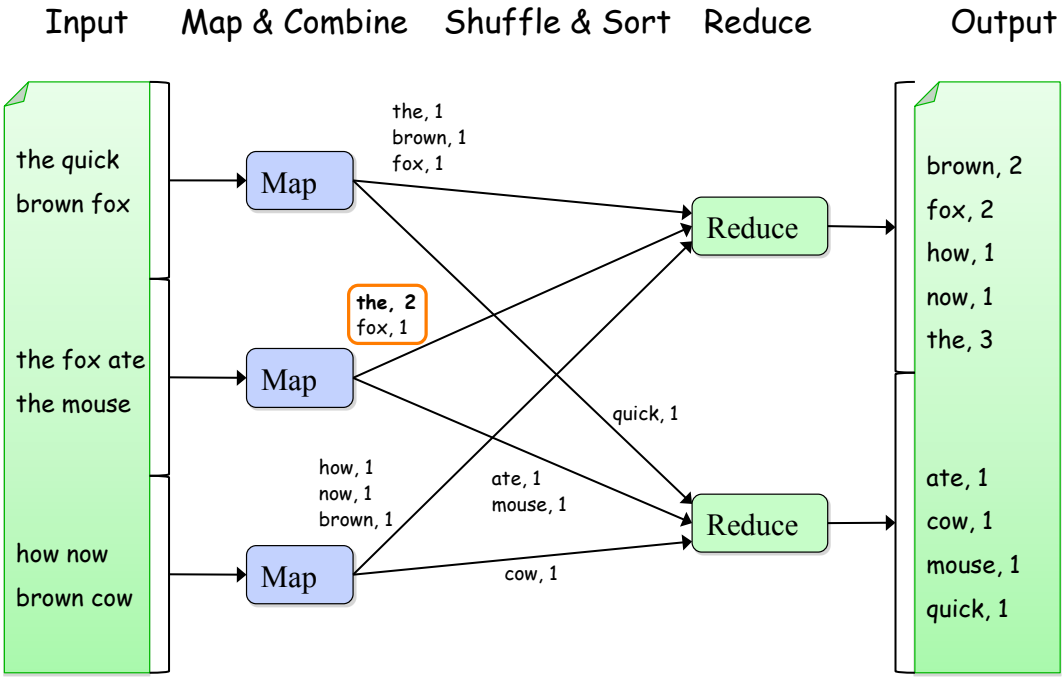
Using a Hadoop Streaming Mapper Script

```
SELECT TRANSFORM(page_views.userid,
                 page_views.date)
USING 'map_script.py'
AS dt, uid CLUSTER BY dt
FROM page_views;
```


MapReduce Optimization

- Optimization to the MapReduce Programming Model
  - Example: locality based optimization (local combine before shuffle)
  - Other possible optimization can be introduced …

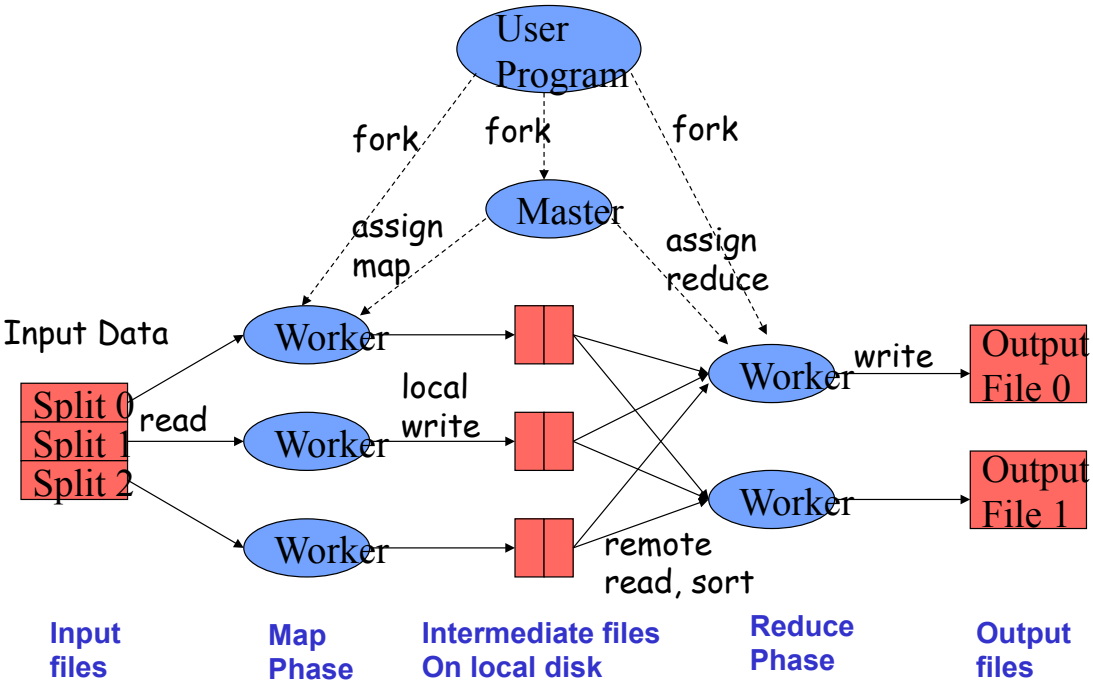
Word Count with Combiner



Outline

- MapReduce architecture
- Sample applications
- Getting started with Hadoop
- Higher-level languages on top of Hadoop: Pig and Hive
- MapReduce Runtime (Execution Engine)
- Fault tolerance in MapReduce

Distributed Execution Overview



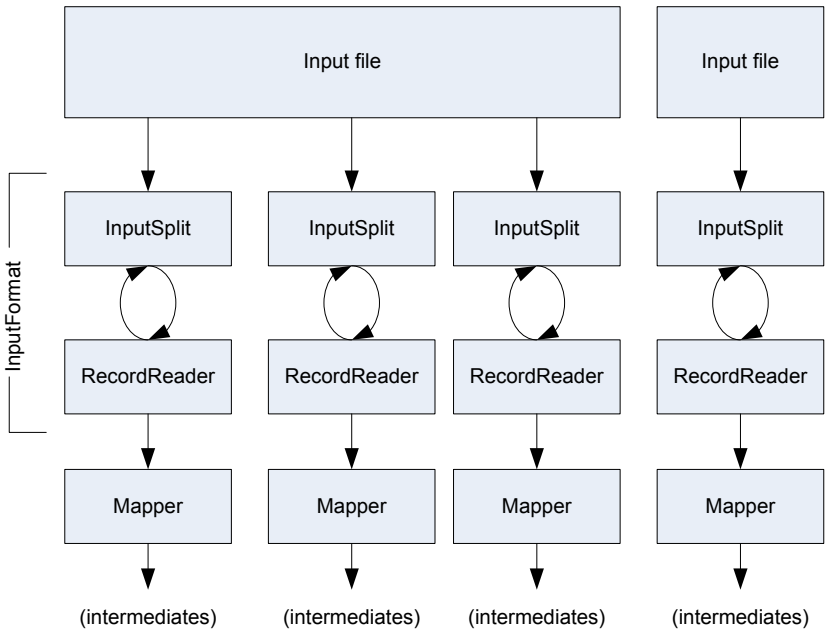
MapReduce Execution Engine

- Mapper-Reducer Coordination: Data Flow
- Master-Slave Coordination: Execution Flow
- Execution Support for Scaling to Big Data

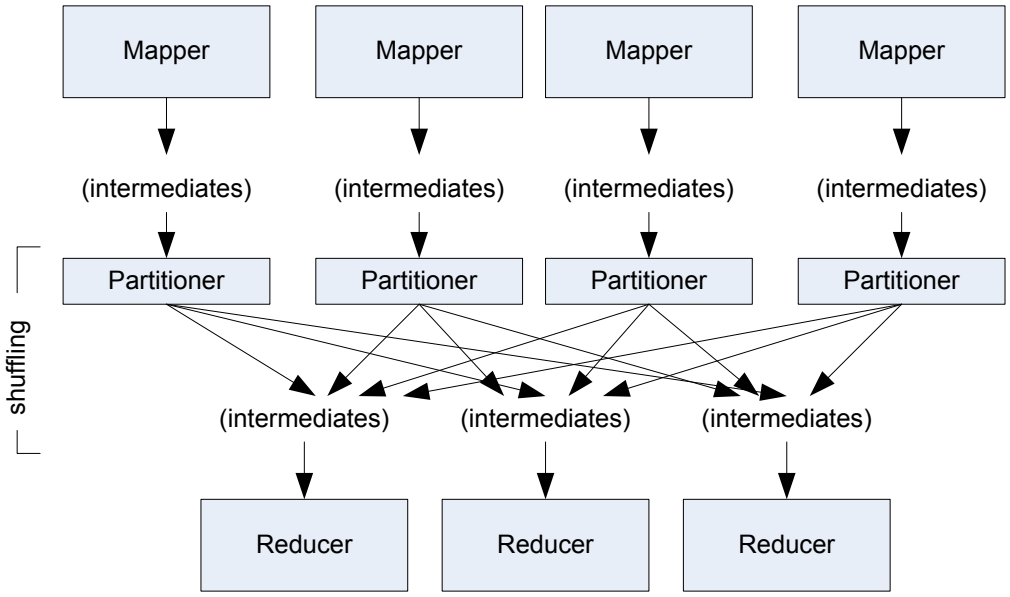
Data flow

- Input, final output are stored on a distributed file system
  - Scheduler tries to schedule map tasks “close” to physical storage location of input data
- Intermediate results are stored on local FS of map and reduce workers
- Output is stored in designated storage locations and triple replicated
  - often input to another map reduce task

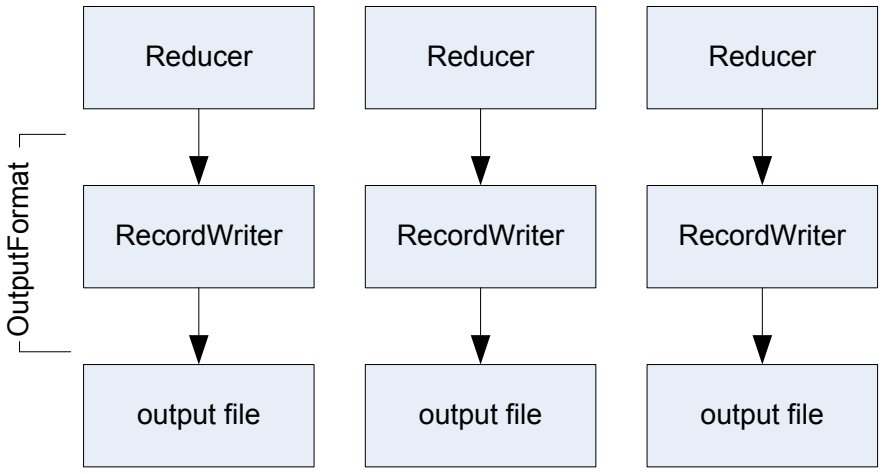
Getting Data To The Mapper



Partition And Shuffle



Finally: Writing The Output



Triple Replication

MapReduce Execution Summary

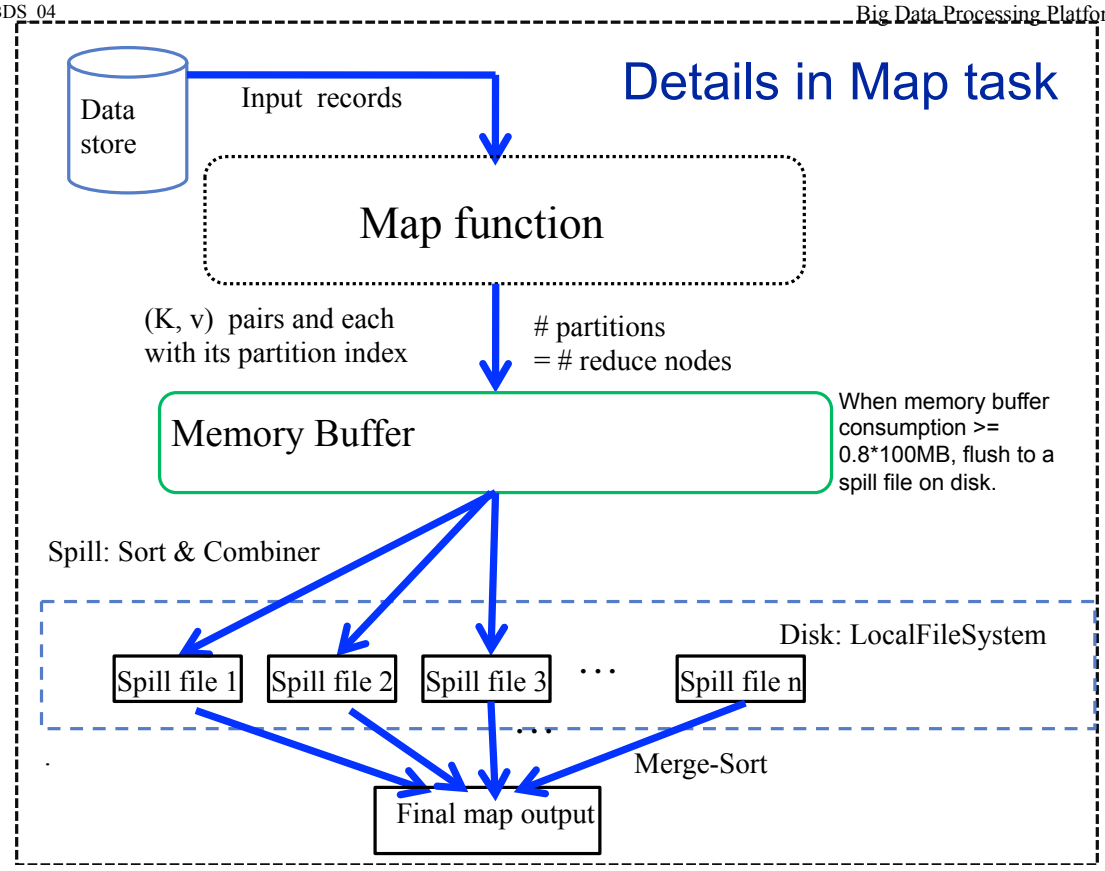
- How is the Map/Reduce Computation is distributed (and parallelized)?
  1. **Map phase**: Partition input key/value pairs into chunks, run map() tasks in parallel
  2. **Shuffle phase**: After all map()s are complete, consolidate all emitted values for each unique emitted key
  3. **Reduce phase**: Now partition space of output map keys, and run reduce() in parallel
- If map() or reduce() fails, reexecute!

MapReduce Execution Details

- Single *master* controls job execution on multiple *slaves*
- Mappers preferentially placed on same node or same rack as their input block
  - Minimizes network usage
- Mappers save outputs to local disk before serving them to reducers
  - Allows recovery if a reducer crashes
  - Allows having more reducers than nodes

How does MapReduce scale to big data

- Input file splits into chunks of equal size and stored in HDFS with triple replication
- Each map takes one chunk (inputsplit) as an input dataset.
- When input dataset is too big to fit into the allocated map slot memory, spill files will be generated.



Execution overview

1. The MapReduce library in the user program first splits input files into M pieces of typically 16 MB to 64 MB/piece. It then starts up many copies of the program on a cluster of machines.
2. One of the copies of the program is the master. The rest are workers that are assigned work by the master. There are M map tasks and R reduce tasks to assign. The master picks idle workers and assigns each one a map task or a reduce task.
3. A worker who is assigned a map task reads the contents of the assigned input split. It parses key/value pairs out of the input data and passes each pair to the user-defined Map function. The intermediate key/value pairs produced by the Map function are buffered in memory.
4. The locations of these buffered pairs on the local disk are passed back to the master, who forwards these locations to the reduce workers.

Execution overview (cont.)

5. When a reduce worker is notified by the master about these locations, it uses RPC remote procedure calls to read the buffered data from the local disks of the map workers. When a reduce worker has read all intermediate data, it sorts it by the intermediate keys so that all occurrences of the same key are grouped together.
6. The reduce worker iterates over the sorted intermediate data and for each **unique intermediate key** encountered, it passes the key and the corresponding set of intermediate values to the user's *Reduce function*. The output of the *Reduce function is appended* to a final output file for this reduce partition.
7. When all map tasks and reduce tasks have been completed, the master wakes up the user program---the MapReduce call in the user program returns back to the user code. The output of the mapreduce execution is available in the R output files (one per reduce task).

Coordination

- Master data structures
  - Task status: (idle, in-progress, completed)
  - Idle tasks get scheduled as workers become available
  - When a map task completes, it produces its map output file and sends the master the location and sizes of its R intermediate files, one for each reducer
  - Master pushes this info to reducers
- Failure handling
  - Master pings workers periodically to detect failures
- Speculation
  - When a map slot is done and no more map task is left, the map slot is available for speculation task (helping stragglers)

Lecture Summary

- MapReduce's data-parallel programming model hides complexity of distribution and fault tolerance
- Principal philosophies:
  - *Make it scale*, so you can throw hardware at problems
  - *Make it cheap*, saving hardware, programmer and administration costs (but requiring fault tolerance)
- Hive and Pig further simplify programming
- MapReduce is not suitable for all problems, but when it works, it may save you a lot of time

Outline

- MapReduce architecture
- Sample applications
- Getting started with Hadoop
- Higher-level languages on top of Hadoop: Pig and Hive
- MapReduce Runtime (Execution Engine)
- Fault tolerance in MapReduce

Fault Tolerance in MapReduce

- Task crashes
- Node crashes
- Speculative Execution
  - If a task is going slowly (straggler)
- Master crashes

Fault Tolerance in MapReduce

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

➢ Note: For this and the other fault tolerance features to work, *your map and reduce tasks must be side-effect-free*

Fault Tolerance in MapReduce

2. If a node crashes:
   - Re-launch its current tasks on other nodes
   - Re-launch any maps the node previously ran
     - Necessary because their output files were lost along with the crashed node

Fault Tolerance in MapReduce

3. If a task is going slowly (straggler):
   - Launch the second copy of task on another node (speculative execution)
   - Take the output of whichever copy finishes first, and kill the other one

- Critical for performance in large clusters: stragglers occur frequently due to failing hardware, bugs, misconfiguration, etc

Failures

Q&A: Why is a completed map task discarded?

- Map worker failure
  - Map tasks completed or in-progress at worker are reset to idle
  - Reduce workers are notified when task is rescheduled on another worker
- Reduce worker failure
  - Only in-progress tasks are reset to idle
- Master failure
  - MapReduce task is aborted and client is notified

Summary

- By providing a data-parallel programming model, MapReduce can control job execution in useful ways:
  - Automatic division of job into tasks
  - Automatic placement of computation near data
  - Automatic load balancing
  - Recovery from failures & stragglers
- Users can focus on application, not on complexities of distributed computing

Outline

Part I

- Hadoop Architecture
- MapReduce Programming Model
- Higher-level languages over Hadoop: Pig and Hive
- MapReduce Execution Engine

Part II

- Inherent Problems and Challenges
  - Intermediate result data loss due to failure
  - MapReduce stage barrier
  - Processing big data with strong or evolving dependencies
  - MapReduce Task Scheduling
  - MapReduce Job scheduling

Questions