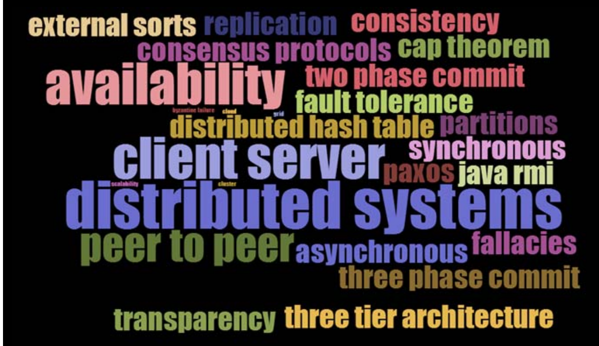
CC5212-1
Procesamiento Masivo de Datos
Otoño 2015

Lecture 4: DFS & MapReduce I

Aidan Hogan
aidhog@gmail.com

Fundamentals of Distributed Systems



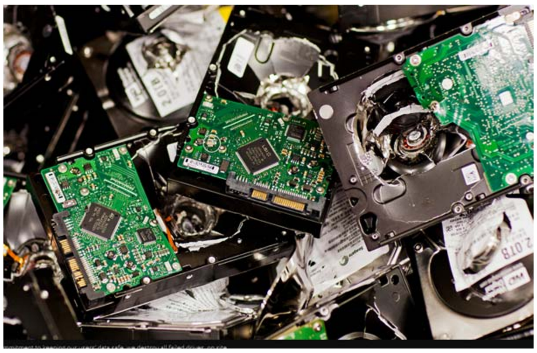
MASSIVE DATA PROCESSING (THE GOOGLE WAY ...)

Inside Google circa 1997/98

Inside Google circa 2015

Google's Cooling System

Google's Recycling Initiative

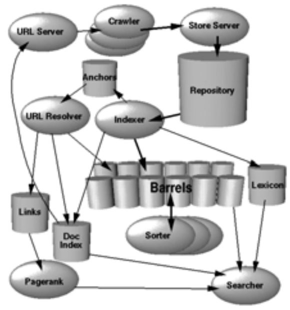


Google Architecture (ca. 1998)

Information Retrieval

- Crawling
- Inverted indexing
- Word-counts
- Link-counts
- greps/sorts
- PageRank
- Updates

…



Google Engineering

- Massive amounts of data
- Each task needs communication protocols
- Each task needs fault tolerance
- Multiple tasks running concurrently

Ad hoc solutions would repeat the same code

Google Engineering

- **Google File System**
  - Store data across multiple machines
  - Transparent Distributed File System
  - Replication / Self-healing
- **MapReduce**
  - Programming abstraction for distributed tasks
  - Handles fault tolerance
  - Supports many "simple" distributed tasks!
- **BigTable, Pregel, Percolator, Dremel …**

Google *Re*-Engineering

Google File System (GFS) → HDFS

MapReduce → hadoop

BigTable → Apache HBase

GOOGLE FILE SYSTEM (GFS)

What is a File-System?

- Breaks files into chunks (or clusters)
- Remembers the sequence of clusters
- Records directory/file structure
- Tracks file meta-data
  - File size
  - Last access
  - Permissions
  - Locks

What is a Distributed File-System

- Same thing, but distributed

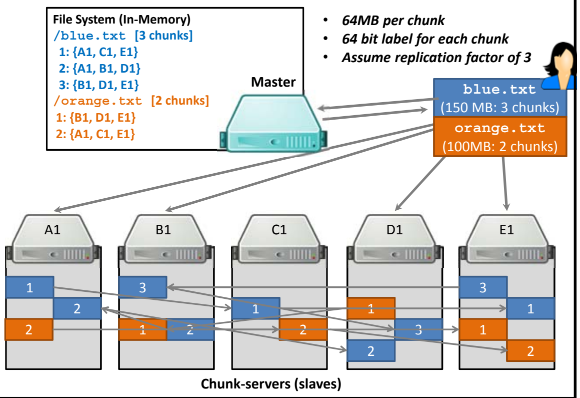
What would transparency / flexibility / reliability / performance / scalability mean for a distributed file system?

- Transparency: Like a normal file-system
- Flexibility: Can mount new machines
- Reliability: Has replication
- Performance: Fast storage/retrieval
- Scalability: Can store a lot of data / support a lot of machines

Google File System (GFS)

- Files are huge
- Files often read or appended
  - Writes in the middle of a file not (really) supported
- Concurrency important
- Failures are frequent
- Streaming important

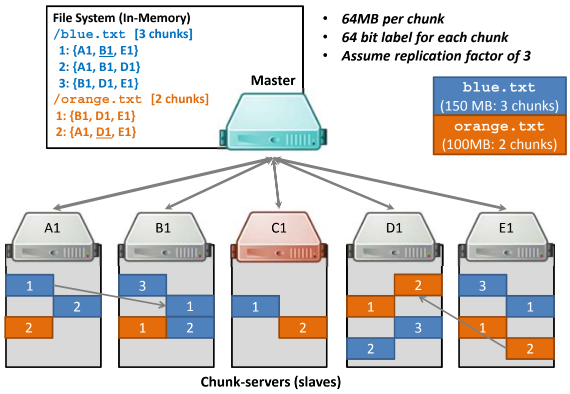
GFS: Pipelined Writes



GFS: Pipelined Writes (In Words)

1. Client asks Master to write a file
2. Master returns a primary chunkserver and secondary chunkservers
3. Client writes to primary chunkserver and tells it the secondary chunkservers
4. Primary chunkserver passes data onto secondary chunkserver, which passes on …
5. When finished, message comes back through the pipeline that all chunkservers have written
   - Otherwise client sends again

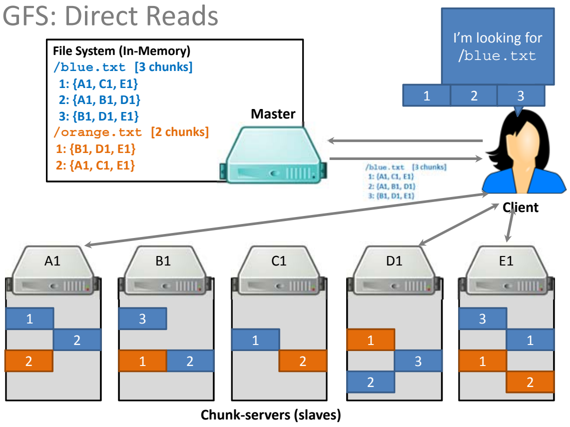
GFS: Fault Tolerance



GFS: Fault Tolerance (In Words)

- Master sends regular "Heartbeat" pings
- If a chunkserver doesn't respond
  1. Master finds out what chunks it had
  2. Master assigns new chunkserver for each chunk
  3. Master tells new chunkserver to copy from a specific existing chunkserver
- Chunks are prioritised by number of remaining replicas, then by demand

GFS: Direct Reads



GFS: Direct Reads (In Words)

1. Client asks Master for file
2. Master returns location of a chunk
   – Returns a ranked list of replicas
3. Client reads chunk directly from chunkserver
4. Client asks Master for next chunk

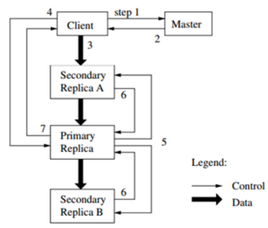
Software makes transparent for client!

GFS: Modification Consistency

Masters assign leases to one replica: a "primary replica"

Client wants to change a file:

1. Client asks Master for the replicas (incl. primary)
2. Master returns replica info to the client
3. Client sends change data
4. Client asks primary to execute the changes
5. Primary asks secondaries to change
6. Secondaries acknowledge to primary
7. Primary acknowledges to client

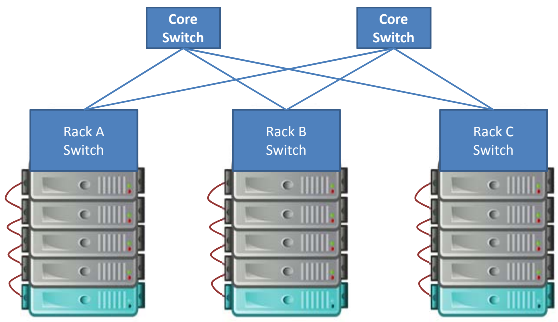


Remember: Concurrency!

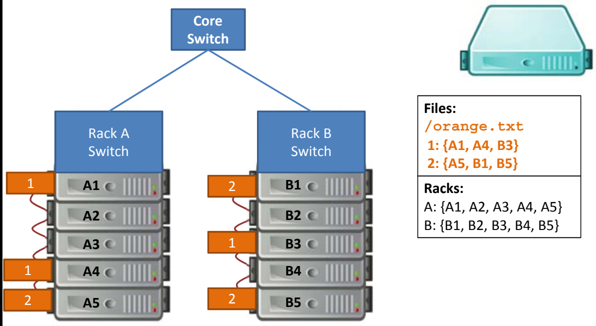
Data & Control Decoupled

GFS: Rack Awareness

GFS: Rack Awareness



GFS: Rack Awareness



GFS: Rack Awareness (In Words)

- Make sure replicas not on same rack
  - In case rack switch fails!
- But communication can be slower:
  - Within rack: pass one switch (rack switch)
  - Across racks: pass three switches (two racks and a core)
- (Typically) pick two racks with low traffic
  - Two nodes in same rack, one node in another rack
    - (Assuming 3x replication)
- Only necessary if more than one rack! ☺

GFS: Other Operations

Rebalancing: Spread storage out evenly

Deletion:

- Just rename the file with hidden file name
  - To recover, rename back to original version
  - Otherwise, three days later will be wiped

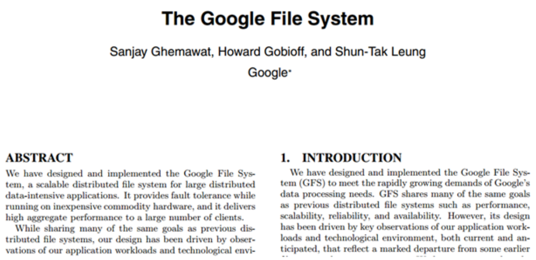
Monitoring Stale Replicas: Dead slave reappears with old data: master keeps version info and will recycle old chunks

GFS: Weaknesses?

What do you see as the core weaknesses of the Google File System?

- Master node single point of failure
  - Use hardware replication
  - Logs and checkpoints!
- Master node is a bottleneck
  - Use more powerful machine
  - Minimise master node traffic
- Master-node metadata kept in memory
  - Each chunk needs 64 bytes
  - Chunk data can be queried from each slave

GFS: White-Paper



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 envi-

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

HADOOP DISTRIBUTED FILE SYSTEM (HDFS)

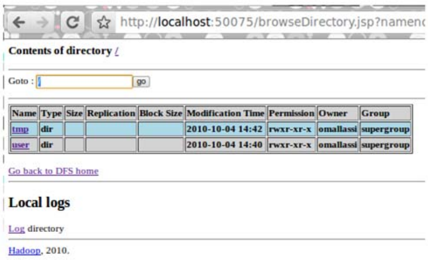
Google *Re*-Engineering

Google File System (GFS)

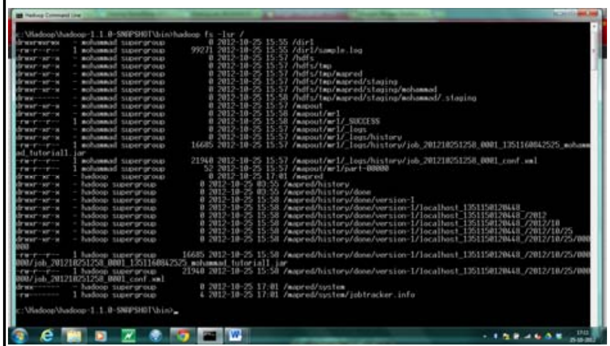
HDFS

- HDFS-to-GFS
  - Data-node = Chunkserver/Slave
  - Name-node = Master
- HDFS does not support modifications
- Otherwise pretty much the same except …
  - GFS is proprietary (hidden in Google)
  - HDFS is open source (Apache!)

HDFS Interfaces



HDFS Interfaces



GOOGLE'S MAP-REDUCE

Google *Re*-Engineering

Google File System (GFS)

MapReduce

MapReduce in Google

- Divide & Conquer:

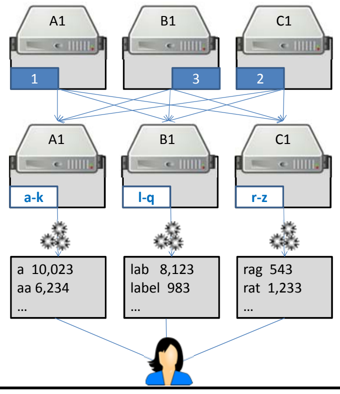
1. Word count

   How could we do a distributed top-*k* word count?

2. Total searches per user
3. PageRank
4. Inverted-indexing

MapReduce: Word Count

Better partitioning method?



Input
Distr. File Sys.

Map()

(Partition/Shuffle)
(Distr. Sort)

Reduce()

Output

MapReduce (in more detail)

1. **Input:** Read from the cluster (e.g., a DFS)
   - Chunks raw data for mappers
   - Maps raw data to initial $(\mathrm{key_{in}}, \mathrm{value_{in}})$ pairs

   What might **Input** do in the word-count case?

2. **Map:** For each $(\mathrm{key_{in}}, \mathrm{value_{in}})$ pair, generate zero-to-many $(\mathrm{key_{map}}, \mathrm{value_{map}})$ pairs
   - $\mathrm{key_{in}}$ /$\mathrm{value_{in}}$ can be diff. type to $\mathrm{key_{map}}$ /$\mathrm{value_{map}}$

   What might **Map** do in the word-count case?

MapReduce (in more detail)

3. **Partition:** Assign sets of $\mathrm{key_{map}}$ values to reducer machines

   How might **Partition** work in the word-count case?

4. **Shuffle:** Data are moved from mappers to reducers (e.g., using DFS)

5. **Comparison/Sort:** Each reducer sorts the data by key using a comparison function
   - *Sort is taken care of by the framework*

MapReduce

6. **Reduce:** Takes a bag of $(\mathrm{key_{map}}, \mathrm{value_{map}})$ pairs with the same $\mathrm{key_{map}}$ value, and produces zero-to-many outputs for each bag
   - Typically zero-or-one outputs

   How might **Reduce** work in the word-count case?

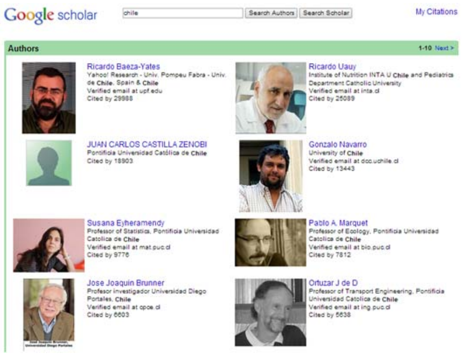
7. **Output:** Merge-sorts the results from the reducers / writes to stable storage

MapReduce: Word Count PseudoCode

```
function map(String name, String document):
  // name: document name
  // document: document contents
  for each word w in document:
    emit (w, 1)

function reduce(String word, Iterator partialCounts):
  // word: a word
  // partialCounts: a list of aggregated partial counts
  sum = 0
  for each pc in partialCounts:
    sum += ParseInt(pc)
  emit (word, sum)
```

MapReduce: Scholar Example



MapReduce: Scholar Example

Assume that in Google Scholar we have inputs like:

$$\text{paper}_{A1} \text{ citedBy } \text{paper}_{B1}$$

How can we use MapReduce to count the total incoming citations per paper?

MapReduce as a Dist. Sys.

- Transparency: Abstracts physical machines
- Flexibility: Can mount new machines; can run a variety of types of jobs
- Reliability: Tasks are monitored by a master node using a heart-beat; dead jobs restart
- Performance: Depends on the application code but exploits parallelism!
- Scalability: Depends on the application code but can serve as the basis for massive data processing!

MapReduce: Benefits for Programmers

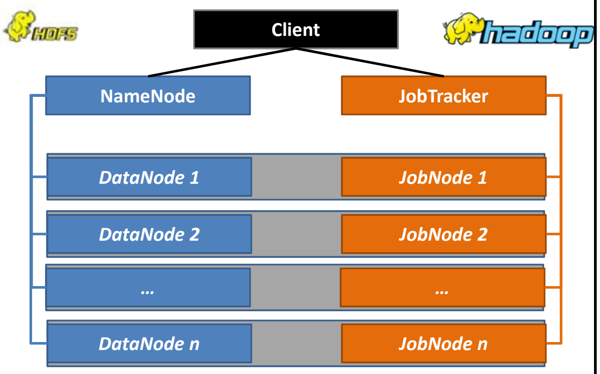
- **Takes care of low-level implementation:**
  - Easy to handle inputs and output
  - No need to handle network communication
  - No need to write sorts or joins
- **Abstracts machines (transparency)**
  - Fault tolerance (through heart-beats)
  - Abstracts physical locations
  - Add / remove machines
  - Load balancing

MapReduce: Benefits for Programmers

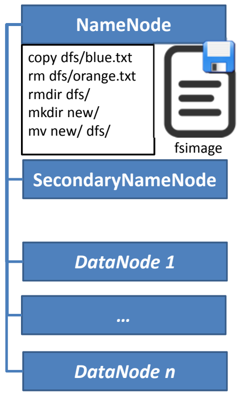
Time for more important things …

HADOOP OVERVIEW

Hadoop Architecture



HDFS: Traditional / SPOF



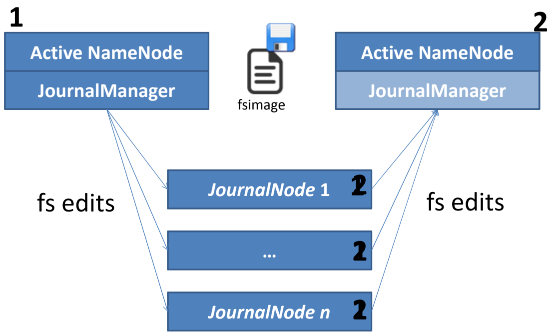
1. NameNode appends edits to log file
2. SecondaryNameNode copies log file and image, makes checkpoint, copies image back
3. NameNode loads image on start-up and makes remaining edits

SecondaryNameNode not a backup NameNode

What is the secondary name-node?

- Name-node quickly logs all file-system actions in a sequential (but messy) way
- Secondary name-node keeps the main `fsimage` file up-to-date based on logs
- When the primary name-node boots back up, it loads the `fsimage` file and applies the remaining log to it
- Hence secondary name-node helps make boot-ups faster, helps keep file system image up-to-date and takes load away from primary

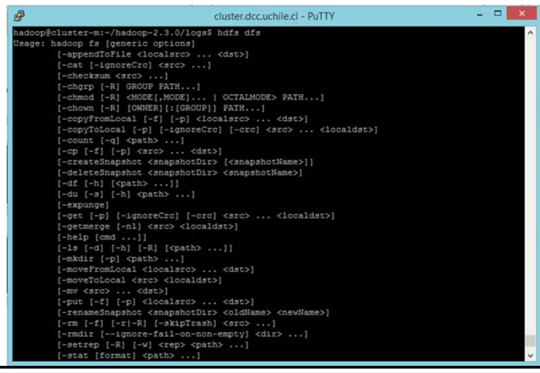
Hadoop: High Availability



PROGRAMMING WITH HADOOP

1. Input/Output (cmd)

\> hdfs dfs




```
hadoop@cluster-m:~/hadoop-2.3.0/logs$ hdfs dfs
Usage: hadoop fs [generic options]
        [-appendToFile <localsrc> ... <dst>]
        [-cat [-ignoreCrc] <src> ...]
        [-checksum <src> ...]
        [-chgrp [-R] GROUP PATH...]
        [-chmod [-R] <MODE[,MODE]... | OCTALMODE> PATH...]
        [-chown [-R] [OWNER][:[GROUP]] PATH...]
        [-copyFromLocal [-f] [-p] <localsrc> ... <dst>]
        [-copyToLocal [-p] [-ignoreCrc] [-crc] <src> ... <localdst>]
        [-count [-q] <path> ...]
        [-cp [-f] [-p] <src> ... <dst>]
        [-createSnapshot <snapshotDir> [<snapshotName>]]
        [-deleteSnapshot <snapshotDir> <snapshotName>]
        [-df [-h] [<path> ...]]
        [-du [-s] [-h] <path> ...]
        [-expunge]
        [-get [-p] [-ignoreCrc] [-crc] <src> ... <localdst>]
        [-getmerge [-nl] <src> <localdst>]
        [-help [cmd ...]]
        [-ls [-d] [-h] [-R] [<path> ...]]
        [-mkdir [-p] <path> ...]
        [-moveFromLocal <localsrc> ... <dst>]
        [-moveToLocal <src> <localdst>]
        [-mv <src> ... <dst>]
        [-put [-f] [-p] <localsrc> ... <dst>]
        [-renameSnapshot <snapshotDir> <oldName> <newName>]
        [-rm [-f] [-r|-R] [-skipTrash] <src> ...]
        [-rmdir [--ignore-fail-on-non-empty] <dir> ...]
        [-setrep [-R] [-w] <rep> <path> ...]
        [-stat [format] <path> ...]
```

1. Input/Output (Java)

```java
public class HDFSHelloWorld {

    public static final String theFilename = "hello.txt";
    public static final String message = "Hello, world!\n";

    public static void main (String [] args) throws IOException {

        Configuration conf = new Configuration();
        FileSystem fs = FileSystem.get(conf);

        Path filenamePath = new Path(theFilename);

        try {
            if (fs.exists(filenamePath)) {
                // remove the file first
                fs.delete(filenamePath, false);
            }

            FSDataOutputStream out = fs.create(filenamePath);
            out.writeUTF(message);
            out.close();

            FSDataInputStream in = fs.open(filenamePath);
            String messageIn = in.readUTF();
            System.out.print(messageIn);
            in.close();
        } catch (IOException ioe) {
            System.err.println("IOException during operation: " + ioe.toString());
            System.exit(1);
        }
    }
}
```

Creates a file system for default configuration

Check if the file exists; if so delete

Create file and write a message

Open and read back

1. Input (Java)

InputFormat:	Description:	Key:	Value:
TextInputFormat	Default format; reads lines of text files	The byte offset of the line	The line contents
KeyValueInputFormat	Parses lines into key, val pairs	Everything up to the first tab character	The remainder of the line
SequenceFileInputFormat	A Hadoop-specific high-performance binary format	user-defined	user-defined

2. Map

```java
package ejemplo;

import java.io.IOException;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapred.MapReduceBase;
import org.apache.hadoop.mapred.Mapper;
import org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.mapred.Reporter;

public class CitationCountMapper extends MapReduceBase
 implements Mapper<Object, Text, Text, IntWritable> {

    private final IntWritable one = new IntWritable(1);
    private Text cited = new Text();

    @Override
    public void map(Object key, Text value,
            OutputCollector<Text, IntWritable> output, Reporter reporter)
                throws IOException {
        String line = value.toString();
        String[] citeOutIn = line.split("\t");
        cited.set(citeOutIn[1]);
        output.collect(cited,one);
    }
}
```

Mapper<InputKeyType, InputValueType, MapKeyType, MapValueType>

OutputCollector will collect the Map key/value pairs

"Reporter" can provide counters and progress to client

Emit output

(Writable *for values*)

```java
package ejemplo;

import java.io.DataInput;
import java.io.DataOutput;
import java.io.IOException;

import org.apache.hadoop.io.Writable;

public class WritableCitation implements Writable {
    public String citingPaper;
    public String citingVenue;
    public int mentions;

    public WritableCitation(String citingPaper, String citingVenue, int mentions) {
        this.citingPaper = citingPaper;
        this.citingVenue = citingVenue;
        this.mentions = mentions;
    }

    public void write(DataOutput out) throws IOException {
        out.writeUTF(citingPaper);
        out.writeUTF(citingVenue);
        out.writeInt(mentions);
    }

    public void readFields(DataInput in) throws IOException {
        citingPaper = in.readUTF();
        citingVenue = in.readUTF();
        mentions = in.readInt();
    }

    public String toString() {
        return citingPaper +"\t" + citingVenue + "\t" + mentions;
    }
}
```

Same order

(not needed in the running example)

(WritableComparable *for keys/values*)

```java
public class WritableComparableCitation implements WritableComparable<WritableComparableCitation> {
    public String citingPaper;
    public String citingVenue;
    public int mentions;

    public WritableComparableCitation(String citingPaper, String citingVenue, int mentions) {
    public void write(DataOutput out) throws IOException {
    public void readFields(DataInput in) throws IOException {
    public String toString() {

    public int compareTo(WritableComparableCitation other) {
        int comp = citingPaper.compareTo(other.citingPaper);
        if(comp==0){
            comp = citingVenue.compareTo(other.citingVenue);
            if(comp == 0){
                comp = Integer.compare(mentions, other.mentions);
            }
        }
        return comp;
    }

    public boolean equals(Object o) {
        if(o==null) return false;
        if(o==this) return true;
        if (!(o instanceof WritableComparableCitation)) return false;
        WritableComparableCitation wcp = (WritableComparableCitation)o;
        return citingPaper.equals(wcp.citingPaper) && this.citingVenue.equals(wcp.citingVenue)
                && this.mentions == wcp.mentions;
    }

    public int hashCode() {
        return citingPaper.hashCode() ^ citingVenue.hashCode() ^ mentions;
    }
}
```

New Interface

Same as before

Needed to sort keys

Needed for default partition function

(not needed in the running example)

3. Partition

PartitionerInterface

```java
package ejemplo;

import org.apache.hadoop.mapred.JobConf;

public class PartitionCites<E> implements Partitioner<WritableComparableCitation, E> {

    @Override
    public int getPartition(WritableComparableCitation key, E val, int machines) {
        return Math.abs(key.hashCode() % machines);
    }

    @Override
    public void configure(JobConf arg0) {
    }

}
```

(This happens to be the default partition method!)

(not needed in the running example)

4. Shuffle

5. Sort/Comparision

```java
public class WritableComparableCitation implements WritableComparable<WritableComparableCitation> {
    public String citingPaper;
    public String citingVenue;
    public int mentions;

    public WritableComparableCitation(String citingPaper, String citingVenue, int mentions) {
    public void write(DataOutput out) throws IOException {
    public void readFields(DataInput in) throws IOException {
    public String toString() {

    public int compareTo(WritableComparableCitation other) {
        int comp = citingPaper.compareTo(other.citingPaper);
        if(comp==0){
            comp = citingVenue.compareTo(other.citingVenue);
            if(comp == 0){
                comp = Integer.compare(mentions, other.mentions);
            }
        }
        return comp;
    }

    public boolean equals(Object o) {
        if(o==null) return false;
        if(o==this) return true;
        if (!(o instanceof WritableComparableCitation)) return false;
        WritableComparableCitation wcp = (WritableComparableCitation)o;
        return citingPaper.equals(wcp.citingPaper) && this.citingVenue.equals(wcp.citingVenue)
                && this.mentions == wcp.mentions;
    }

    public int hashCode() {
        return citingPaper.hashCode() ^ citingVenue.hashCode() ^ mentions;
    }
}
```

Methods in WritableComparator

(not needed in the running example)

6. Reduce

Reducer<MapKey, MapValue, OutputKey, OutputValue>

```java
package ejemplo;

import java.io.IOException;

public class CitationCountReducer extends MapReduceBase
 implements Reducer<Text, IntWritable, Text, IntWritable> {

    @Override
    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();
        }
        reporter.incrCounter("citations", key.toString().substring(0, 1), sum);
        output.collect(key, new IntWritable(sum));
    }
}
```

Key, Iterator over all values for that key, output key–value pair collector, reporter

Write to output

7. Output / Input (Java)

```java
public class HDFSHelloWorld {

    public static final String theFilename = "hello.txt";
    public static final String message = "Hello, world!\n";

    public static void main (String [] args) throws IOException {

        Configuration conf = new Configuration();
        FileSystem fs = FileSystem.get(conf);

        Path filenamePath = new Path(theFilename);

        try {
            if (fs.exists(filenamePath)) {
                // remove the file first
                fs.delete(filenamePath, false);
            }

            FSDataOutputStream out = fs.create(filenamePath);
            out.writeUTF(message);
            out.close();

            FSDataInputStream in = fs.open(filenamePath);
            String messageIn = in.readUTF();
            System.out.print(messageIn);
            in.close();
        } catch (IOException ioe) {
            System.err.println("IOException during operation: " + ioe.toString());
            System.exit(1);
        }
    }
}
```

Creates a file system for default configuration

Check if the file exists; if so delete

Create file and write a message

Open and read back

7. Output (Java)

OutputFormat:	Description
TextOutputFormat	Default; writes lines in "key \t value" form
SequenceFileOutputFormat	Writes binary files suitable for reading into subsequent MapReduce jobs
NullOutputFormat	Disregards its inputs

Control Flow

```java
package ejemplo;

import org.apache.hadoop.fs.Path;

public class CitationCount {

    public static void main(String[] args) {
        JobClient client = new JobClient();
        JobConf conf = new JobConf(CitationCount.class);

        conf.setOutputKeyClass(Text.class);
        conf.setOutputValueClass(IntWritable.class);

        FileInputFormat.addInputPath(conf, new Path("input"));
        FileOutputFormat.setOutputPath(conf, new Path("output"));

        conf.setMapperClass(CitationCountMapper.class);

        conf.setReducerClass(CitationCountReducer.class);
//      conf.setCombinerClass(CitationCountReducer.class);

        client.setConf(conf);
        try {
            JobClient.runJob(conf);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Create a JobClient, a JobConf and pass it the main class

Set the type of output key and value in the configuration

Set input and output paths

Set the mapper class

Set the reducer class (and optionally "**combiner**")

Pass the configuration to the client and run

More in Hadoop: Combiner

- Map-side “mini-reduction”
- Keeps a fixed-size buffer in memory
- Reduce within that buffer
  - e.g., count words in buffer
  - Lessens bandwidth needs
- In Hadoop: can simply use Reducer class ☺

More in Hadoop: Reporter

```
package ejemplo;

import java.io.IOException;

public class CitationCountReducer extends MapReduceBase
 implements Reducer<Text, IntWritable, Text, IntWritable> {

    @Override
    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();
        }
        reporter.incrCounter("citations", key.toString().substring(0, 1), sum);
        output.collect(key, new IntWritable(sum));
    }
}
```

Reporter has a group of maps of counters

More in Hadoop: Chaining Jobs

- Sometimes we need to chain jobs
- In Hadoop, can pass a set of Jobs to the client
- `x.addDependingJob(y)`

More in Hadoop: Distributed Cache

- Some tasks need “global knowledge”
  - For example, a white-list of conference venues and journals that should be considered in the citation count
  - Typically small
- Use a distributed cache:
  - Makes data available locally to all nodes

RECAP

Distributed File Systems

- Google File System (GFS)
  - **Master and Chunkslaves**
  - Replicated pipelined writes
  - Direct reads
  - Minimising master traffic
  - Fault-tolerance: self-healing
  - Rack awareness
  - Consistency and modifications
- Hadoop Distributed File System
  - **NameNode and DataNodes**

MapReduce

1. **Input**
2. **Map**
3. **Partition**
4. **Shuffle**
5. **Comparison/Sort**
6. **Reduce**
7. **Output**

MapReduce/GFS Revision

- GFS: distributed file system
  - Implemented as HDFS
- MapReduce: distributed processing framework
  - Implemented as Hadoop

Hadoop

- `FileSystem`
- `Mapper<InputKey,InputValue,MapKey,MapValue>`
- `OutputCollector<OutputKey,OutputValue>`
- `Writable, WritableComparable<Key>`
- `Partitioner<KeyType,ValueType>`
- `Reducer<MapKey,MapValue,OutputKey,OutputValue>`
- `JobClient/JobConf`

…