

Big Data and the Cloud

Part II

Big Data Storage on the Cloud

Hadoop

▶ Plan for today

- ▶ **A brief history of Hadoop**
- ▶ **Writing jobs for Hadoop**
 - ▶ Mappers, reducers, drivers
 - ▶ Compiling and running a job
- ▶ **Hadoop Distributed File System (HDFS)**
 - ▶ Node types; read and write operation
 - ▶ Accessing data in HDFS
- ▶ **Hadoop internals**
 - ▶ Dataflow: Input format, partitioner, combiner, ...
 - ▶ Fully distributed mode: Node types, setup
 - ▶ Fault tolerance; speculative execution



▶ 2002-2004: Lucene and Nutch

- ▶ Early 2000s: Doug Cutting develops two open-source search projects:

- ▶ Lucene: Search indexer
 - ▶ Used e.g., by Wikipedia
- ▶ Nutch: A spider/crawler (with Mike Carafella)



▶ Nutch

- ▶ Goal: Web-scale, crawler-based search
- ▶ Written by a few part-time developers
- ▶ Distributed, 'by necessity'
- ▶ Demonstrated 100M web pages on 4 nodes, but true 'web scale' still very distant



2004-2006: GFS and MapReduce

- ▶ 2003/04: GFS, MapReduce papers published
 - ▶ Sanjay Ghemawat, Howard Gobioff, Shun-Tak Leung: "The Google File System", SOSP 2003
 - ▶ Jeffrey Dean and Sanjay Ghemawat: "MapReduce: Simplified Data Processing on Large Clusters", OSDI 2004
 - ▶ Directly addressed Nutch's scaling issues
- ▶ GFS & MapReduce added to Nutch
 - ▶ Two part-time developers over two years (2004-2006)
 - ▶ Crawler & indexer ported in two weeks
 - ▶ Ran on 20 nodes at IA and UW
 - ▶ Much easier to program and run, scales to several 100M web pages, but still far from web scale




2006-2008: Yahoo

- ▶ **2006: Yahoo hires Cutting**
 - ▶ Provides engineers, clusters, users, ...
 - ▶ Big boost for the project; Yahoo spends tens of M\$
 - ▶ Not without a price: Yahoo has a slightly different focus (e.g., security) than the rest of the project; delays result
- ▶ **Hadoop project split out of Nutch**
 - ▶ Finally hit web scale in early 2008
- ▶ **Cutting is now at Cloudera**
 - ▶ Startup; started by three top engineers from Google, Facebook, Yahoo, and a former executive from Oracle
 - ▶ Has its own version of Hadoop; software remains free, but company sells support and consulting services
 - ▶ Was elected chairman of Apache Software Foundation



▶ Who uses Hadoop?

- ▶ Hadoop is running search on some of the Internet's largest sites:
 - ▶ Amazon Web Services: Elastic MapReduce
 - ▶ AOL: Variety of uses, e.g., behavioral analysis & targeting
 - ▶ EBay: Search optimization (532-node cluster)
 - ▶ Facebook: Reporting/analytics, machine learning (1100 m.)
 - ▶ Fox Interactive Media: MySpace, Photobucket, Rotten T.
 - ▶ Last.fm: Track statistics and charts
 - ▶ IBM: Blue Cloud Computing Clusters
 - ▶ LinkedIn: People You May Know (2x50 machines)
 - ▶ Rackspace: Log processing
 - ▶ Twitter: Store + process tweets, log files, other data
 - ▶ Yahoo: >36,000 nodes; biggest cluster is 4,000 nodes

Chapter 16
of your
textbook



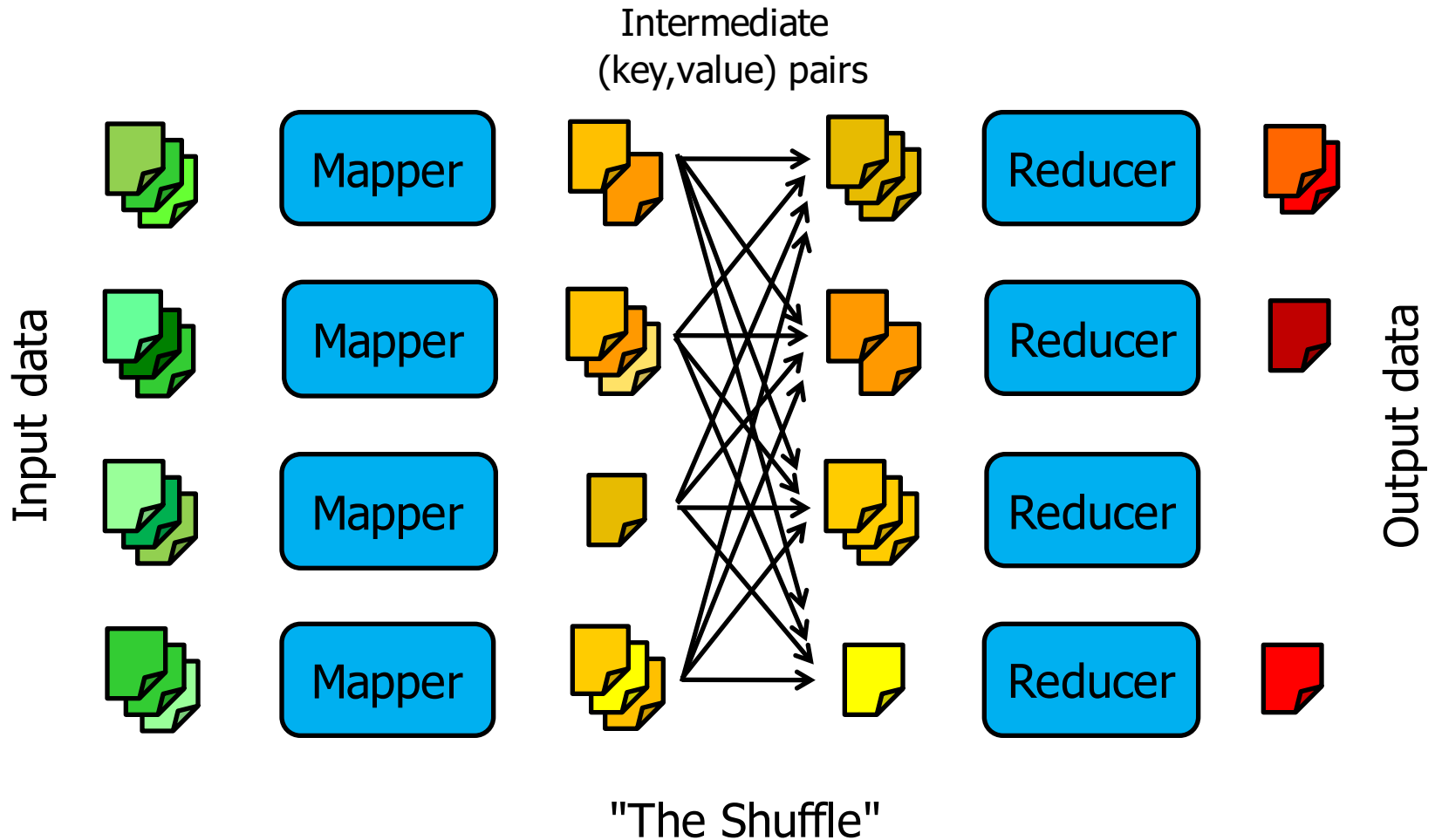
▶ Plan for today

- ▶ A brief history of Hadoop 
- ▶ Writing jobs for Hadoop 
 - ▶ Mappers, reducers, drivers
 - ▶ Compiling and running a job
- ▶ Hadoop Distributed File System (HDFS)
 - ▶ Node types; read and write operation
 - ▶ Accessing data in HDFS
- ▶ Hadoop internals
 - ▶ Dataflow: Input format, partitioner, combiner, ...
 - ▶ Fully distributed mode: Node types, setup
 - ▶ Fault tolerance; speculative execution

Simplified scenario

- ▶ In this section, I will demonstrate how to use Hadoop in standalone mode
 - ▶ Useful for development and debugging (NOT for production)
 - ▶ Single node (e.g., your laptop computer)
 - ▶ No jobtrackers or tasktrackers
 - ▶ Data in local file system, not in HDFS
- ▶ This is how the Hadoop installation in your virtual machine works
- ▶ Later: Fully-distributed mode
 - ▶ Used when running Hadoop on actual clusters

▶ Recap: MapReduce dataflow



▶ What do we need to write?

- ▶ **A mapper**
 - ▶ Accepts (key,value) pairs from the input
 - ▶ Produces intermediate (key,value) pairs, which are then shuffled
- ▶ **A reducer**
 - ▶ Accepts intermediate (key,value) pairs
 - ▶ Produces final (key,value) pairs for the output
- ▶ **A driver**
 - ▶ Specifies which inputs to use, where to put the outputs
 - ▶ Chooses the mapper and the reducer to use
- ▶ **Hadoop takes care of the rest!!**
 - ▶ Default behaviors can be customized by the driver

▶ Hadoop data types

Name	Description	JDK equivalent
IntWritable	32-bit integers	Integer
LongWritable	64-bit integers	Long
DoubleWritable	Floating-point numbers	Double
Text	Strings	String

- ▶ Hadoop uses its own serialization
 - ▶ Java serialization is known to be very inefficient
- ▶ Result: A set of special data types
 - ▶ All implement the 'Writable' interface
 - ▶ Most common types shown above; also has some more specialized types (SortedMapWritable, ObjectWritable, ...)
 - ▶ **Caution:** Behavior somewhat unusual



The Mapper

Input format
(file offset, line)

Intermediate format
can be freely chosen

```
import org.apache.hadoop.mapreduce.*;
import org.apache.hadoop.io.*;

public class FooMapper extends Mapper<LongWritable, Text, Text, Text> {
    public void map(LongWritable key, Text value, Context context) {
        context.write(new Text("foo"), value);
    }
}
```

- ▶ Extends abstract 'Mapper' class
 - ▶ Input/output types are specified as type parameters
- ▶ Implements a 'map' function
 - ▶ Accepts (key,value) pair of the specified type
 - ▶ Writes output pairs by calling 'write' method on context
 - ▶ Mixing up the types will cause problems at runtime (!)



The Reducer

Intermediate format
(same as mapper output)

Output format

```
import org.apache.hadoop.mapreduce.*;
import org.apache.hadoop.io.*;

public class FooReducer extends Reducer<Text, Text, IntWritable, Text> {
    public void reduce(Text key, Iterable<Text> values, Context context)
        throws java.io.IOException, InterruptedException
    {
        for (Text value: values)
            context.write(new IntWritable(4711), value);
    }
}
```

Note: We may get multiple values for the same key!

- ▶ Extends abstract 'Reducer' class
 - ▶ Must specify types again (must be compatible with mapper!)
- ▶ Implements a 'reduce' function
 - ▶ Values are passed in as an 'Iterable'
 - ▶ **Caution:** These are NOT normal Java classes. Do not store them in collections - content can change between iterations!



The Driver

```
import org.apache.hadoop.mapreduce.*;
import org.apache.hadoop.io.*;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

public class FooDriver {
    public static void main(String[] args) throws Exception {
        Job job = new Job();
        job.setJarByClass(FooDriver.class);

        FileInputFormat.addInputPath(job, new Path("in"));
        FileOutputFormat.setOutputPath(job, new Path("out"));

        job.setMapperClass(FooMapper.class);
        job.setReducerClass(FooReducer.class);

        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(Text.class);

        System.exit(job.waitForCompletion(true) ? 0 : 1);
    }
}
```

Mapper&Reducer are
in the same Jar as
FooDriver

Input and Output
paths

Format of the (key,value)
pairs output by the
reducer

- ▶ Specifies how the job is to be executed
 - ▶ Input and output directories; mapper & reducer classes

▶ Plan for today

- ▶ A brief history of Hadoop ✓
- ▶ Writing jobs for Hadoop ✓
 - ▶ Mappers, reducers, drivers ✓
 - ▶ **Compiling and running a job** ← NEXT
- ▶ Hadoop Distributed File System (HDFS)
 - ▶ Node types; read and write operation
 - ▶ Accessing data in HDFS
- ▶ Hadoop internals
 - ▶ Dataflow: Input format, partitioner, combiner, ...
 - ▶ Fully distributed mode: Node types, setup
 - ▶ Fault tolerance; speculative execution



Manual compilation

- ▶ Goal: Produce a JAR file that contains the classes for mapper, reducer, and driver
 - ▶ This can be submitted to the Job Tracker, or run directly through Hadoop

- ▶ **Step #1:** Put `hadoop-core-1.0.3.jar` into classpath:

```
export CLASSPATH=$CLASSPATH:/path/to/hadoop/hadoop-core-1.0.3.jar
```

- ▶ **Step #2:** Compile mapper, reducer, driver:

```
javac FooMapper.java FooReducer.java FooDriver.java
```

- ▶ **Step #3:** Package into a JAR file:

```
jar cvf Foo.jar *.class
```

- ▶ Alternative: "Export..." / "Java JAR file" in Eclipse
-



Compilation with Ant

```
<project name="foo" default="jar" basedir=".">
  <target name="init">
    <mkdir dir="classes"/>
  </target>

  <target name="compile" depends="init">
    <javac srcdir="src" destdir="classes" includes="*.java" debug="true"/>
  </target>

  <target name="jar" depends="compile">
    <jar destfile="foo.jar">
      <fileset dir="classes" includes="**/*.class"/>
    </jar>
  </target>

  <target name="clean">
    <delete dir="classes"/>
    <delete file="foo.jar"/>
  </target>
</project>
```

Directory where
source files
are kept

Makes the JAR
file

Clean up any
derived files

- ▶ Apache Ant: A build tool for Java (~"make")
 - ▶ Run "ant jar" to build the JAR automatically
 - ▶ Run "ant clean" to clean up derived files (like make clean)

Standalone mode installation

- ▶ What is standalone mode?
 - ▶ Installation on a single node
 - ▶ No daemons running (no Task Tracker, Job Tracker)
 - ▶ Hadoop runs as an 'ordinary' Java program
 - ▶ Used for debugging



Running a job in standalone mode

- ▶ **Step #1: Create & populate input directory**
 - ▶ Configured in the Driver via `addInputPath()`
 - ▶ Put input file(s) into this directory (ok to have more than 1)
 - ▶ Output directory must not exist yet
- ▶ **Step #2: Run Hadoop**
 - ▶ As simple as this: `hadoop jar <jarName> <driverClassName>`
 - ▶ Example: `hadoop jar foo.jar upenn.nets212.FooDriver`
 - ▶ In verbose mode, Hadoop will print statistics while running
- ▶ **Step #3: Collect output files**



Recap: Writing simple jobs for Hadoop

- ▶ Write a mapper, reducer, driver
 - ▶ Custom serialization → Must use special data types (Writable)
 - ▶ Explicitly declare all three (key,value) types
- ▶ Package into a JAR file
 - ▶ Must contain class files for mapper, reducer, driver
 - ▶ Create manually (javac/jar) or automatically (ant)
- ▶ Running in standalone mode
 - ▶ `hadoop jar foo.jar FooDriver`
 - ▶ Input and output directories in local file system



Wait a second...

- ▶ Wasn't Hadoop supposed to be very scalable?
 - ▶ Work on Petabytes of data, run on thousands of machines
- ▶ Some more puzzle pieces are needed
 - ▶ Special file system that can a) hold huge amounts of data, and b) feed them into MapReduce efficiently
 - Hadoop Distributed File System (HDFS)
 - ▶ Framework for distributing map and reduce tasks across many nodes, coordination, fault tolerance...
 - Fully distributed mode
 - ▶ Mechanism for customizing dataflow for particular applications (e.g., non-textual input format, special sort...)
 - Hadoop data flow

▶ Plan for today

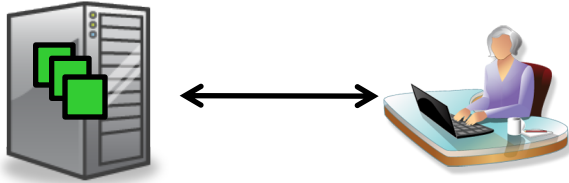
- ▶ A brief history of Hadoop ✓
- ▶ Writing jobs for Hadoop
 - ▶ Mappers, reducers, drivers ✓
 - ▶ Compiling and running a job ✓
- ▶ Hadoop Distributed File System (HDFS) ← NEXT
 - ▶ Node types; read and write operation
 - ▶ Accessing data in HDFS
- ▶ Hadoop internals
 - ▶ Dataflow: Input format, partitioner, combiner, ...
 - ▶ Fully distributed mode: Node types, setup
 - ▶ Fault tolerance; speculative execution



What is HDFS?

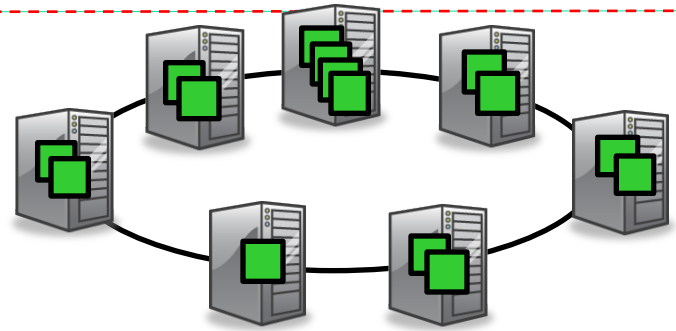
- ▶ HDFS is a distributed file system
 - ▶ Makes some unique tradeoffs that are good for MapReduce
- ▶ What HDFS does well:
 - ▶ Very large read-only or append-only files (individual files may contain Gigabytes/Terabytes of data)
 - ▶ Sequential access patterns
- ▶ What HDFS does not do well:
 - ▶ Storing lots of small files
 - ▶ Low-latency access
 - ▶ Multiple writers
 - ▶ Writing to arbitrary offsets in the file

▶ HDFS versus NFS



Network File System (NFS)

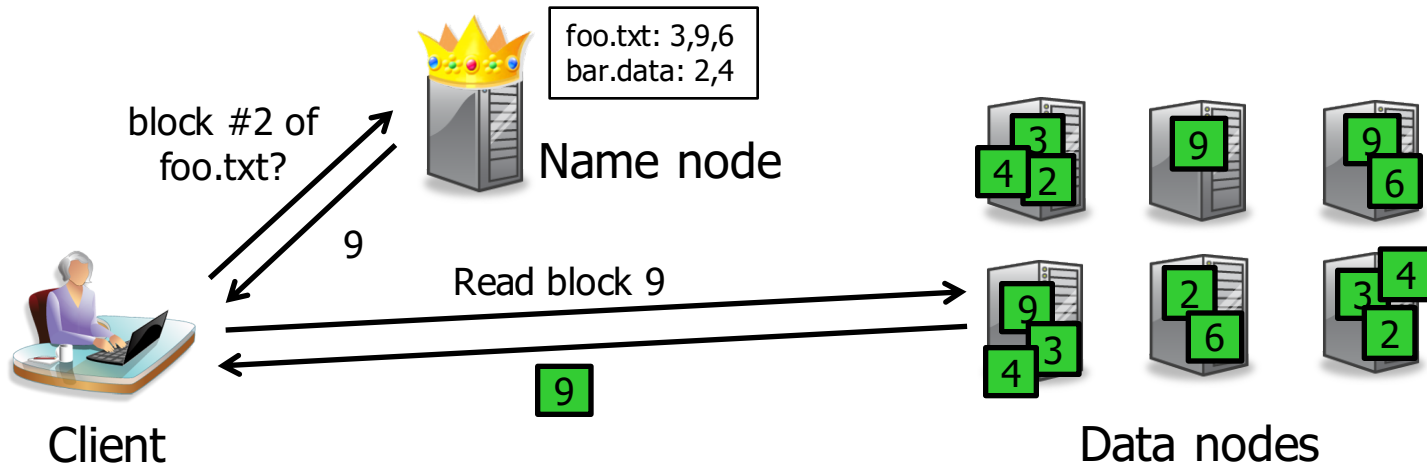
- ▶ Single machine makes part of its file system available to other machines
- ▶ Sequential or random access
- ▶ **PRO:** Simplicity, generality, transparency
- ▶ **CON:** Storage capacity and throughput limited by single server



Hadoop Distributed File System (HDFS)

- Single virtual file system spread over many machines
- Optimized for sequential read and local accesses
- **PRO:** High throughput, high capacity
- **"CON":** Specialized for particular types of applications

▶ How data is stored in HDFS



- ▶ **Files are stored as sets of (large) blocks**
 - ▶ Default block size: 64 MB (ext4 default is 4kB!)
 - ▶ Blocks are replicated for durability and availability
 - ▶ What are the advantages of this design?
- ▶ **Namespace is managed by a single name node**
 - ▶ Actual data transfer is directly between client & data node
 - ▶ Pros and cons of this decision?

▶ The Namenode



Name node

```
foo.txt: 3,9,6  
bar.data: 2,4  
blah.txt: 17,18,19,20  
xyz.img: 8,5,1,11
```

fsimage

```
Created abc.txt  
Appended block 21 to blah.txt  
Deleted foo.txt  
Appended block 22 to blah.txt  
Appended block 23 to xyz.img  
...
```

edits

- ▶ **State stored in two files: fsimage and edits**
 - ▶ fsimage: Snapshot of file system metadata
 - ▶ edits: Changes since last snapshot
- ▶ **Normal operation:**
 - ▶ When namenode starts, it reads fsimage and then applies all the changes from edits sequentially
 - ▶ Pros and cons of this design?

The Secondary Namenode

- ▶ What if the state of the namenode is lost?
 - ▶ Data in the file system can no longer be read!
- ▶ **Solution #1: Metadata backups**
 - ▶ Namenode can write its metadata to a local disk, and/or to a remote NFS mount
- ▶ **Solution #2: Secondary Namenode**
 - ▶ Purpose: Periodically merge the edit log with the fsimage to prevent the log from growing too large
 - ▶ Has a copy of the metadata, which can be used to reconstruct the state of the namenode
 - ▶ But: State lags behind somewhat, so data loss is likely if the namenode fails

▶ Plan for today

- ▶ A brief history of Hadoop ✓
- ▶ Writing jobs for Hadoop
 - ▶ Mappers, reducers, drivers ✓
 - ▶ Compiling and running a job ✓
- ▶ Hadoop Distributed File System (HDFS) ✓
 - ▶ Node types; read and write operation ✓
 - ▶ Accessing data in HDFS ← NEXT
- ▶ Hadoop internals
 - ▶ Dataflow: Input format, partitioner, combiner, ...
 - ▶ Fully distributed mode: Node types, setup
 - ▶ Fault tolerance; speculative execution



Accessing data in HDFS

```
[ahae@carbon ~]$ ls -la /tmp/hadoop-ahae/dfs/data/current/
total 209588
drwxrwxr-x 2 ahae ahae      4096 2013-10-08 15:46 .
drwxrwxr-x 5 ahae ahae      4096 2013-10-08 15:39 ..
-rw-rw-r-- 1 ahae ahae 11568995 2013-10-08 15:44 blk_-3562426239750716067
-rw-rw-r-- 1 ahae ahae   90391 2013-10-08 15:44 blk_-3562426239750716067_1020.meta
-rw-rw-r-- 1 ahae ahae     4 2013-10-08 15:40 blk_5467088600876920840
-rw-rw-r-- 1 ahae ahae    11 2013-10-08 15:40 blk_5467088600876920840_1019.meta
-rw-rw-r-- 1 ahae ahae 67108864 2013-10-08 15:44 blk_7080460240917416109
-rw-rw-r-- 1 ahae ahae  524295 2013-10-08 15:44 blk_7080460240917416109_1020.meta
-rw-rw-r-- 1 ahae ahae 67108864 2013-10-08 15:44 blk_-8388309644856805769
-rw-rw-r-- 1 ahae ahae  524295 2013-10-08 15:44 blk_-8388309644856805769_1020.meta
-rw-rw-r-- 1 ahae ahae 67108864 2013-10-08 15:44 blk_-9220415087134372383
-rw-rw-r-- 1 ahae ahae  524295 2013-10-08 15:44 blk_-9220415087134372383_1020.meta
-rw-rw-r-- 1 ahae ahae    158 2013-10-08 15:40 VERSION
[ahae@carbon ~]$
```

- ▶ HDFS implements a separate namespace
 - ▶ Files in HDFS are not visible in the normal file system
 - ▶ Only the blocks and the block metadata are visible
 - ▶ HDFS cannot be (easily) mounted
 - ▶ Some FUSE drivers have been implemented for it



Accessing data in HDFS

```
[ahae@carbon ~]$ /usr/local/hadoop/bin/hadoop fs -ls /user/ahae
Found 4 items
-rw-r--r--  1 ahae supergroup      1366 2013-10-08 15:46 /user/ahae/README.txt
-rw-r--r--  1 ahae supergroup         0 2013-10-08 15:35 /user/ahae/input
-rw-r--r--  1 ahae supergroup         0 2013-10-08 15:39 /user/ahae/input2
-rw-r--r--  1 ahae supergroup 212895587 2013-10-08 15:44 /user/ahae/input3
[ahae@carbon ~]$
```

- ▶ File access is through the hadoop command
- ▶ Examples:
 - ▶ `hadoop fs -put [file] [hdfsPath]` Stores a file in HDFS
 - ▶ `hadoop fs -ls [hdfsPath]` List a directory
 - ▶ `hadoop fs -get [hdfsPath] [file]` Retrieves a file from HDFS
 - ▶ `hadoop fs -rm [hdfsPath]` Deletes a file in HDFS
 - ▶ `hadoop fs -mkdir [hdfsPath]` Makes a directory in HDFS

▶ Alternatives to the command line

- ▶ Getting data in and out of HDFS through the command-line interface is a bit cumbersome
- ▶ Alternatives have been developed:
 - ▶ FUSE file system: Allows HDFS to be mounted under Unix
 - ▶ WebDAV share: Can be mounted as filesystem on many OSes
 - ▶ HTTP: Read access through namenode's embedded web svr
 - ▶ FTP: Standard FTP interface
 - ▶ ...



Accessing HDFS directly from Java

- ▶ Programs can read/write HDFS files directly
 - ▶ Not needed in MapReduce; I/O is handled by the framework
- ▶ Files are represented as URIs
 - ▶ Example: `hdfs://localhost/user/ahae/example.txt`
- ▶ Access is via the `FileSystem` API
 - ▶ To get access to the file: `FileSystem.get()`
 - ▶ For reading, call `open()` -- returns `InputStream`
 - ▶ For writing, call `create()` -- returns `OutputStream`



What about permissions?

- ▶ Since 0.16.1, Hadoop has rudimentary support for POSIX-style permissions
 - ▶ rwx for users, groups, 'other' -- just like in Unix
 - ▶ 'hadoop fs' has support for chmod, chgrp, chown
- ▶ **But: POSIX model is not a very good fit**
 - ▶ Many combinations are meaningless: Files cannot be executed, and existing files cannot really be written to
- ▶ **Permissions were not really enforced**
 - ▶ Hadoop does not verify whether user's identity is genuine
 - ▶ Useful more to prevent accidental data corruption or casual misuse of information



Where are things today?

- ▶ Since v.20.20x, Hadoop has some security
 - ▶ Kerberos RPC (SASL/GSSAPI)
 - ▶ HTTP SPNEGO authentication for web consoles
 - ▶ HDFS file permissions actually enforced
 - ▶ Various kinds of delegation tokens
 - ▶ Network encryption
 - ▶ For more details, see:
<https://issues.apache.org/jira/secure/attachment/12428537/security-design.pdf>
- ▶ Big changes are coming
 - ▶ Project Rhino (e.g., encrypted data at rest)



Recap: HDFS

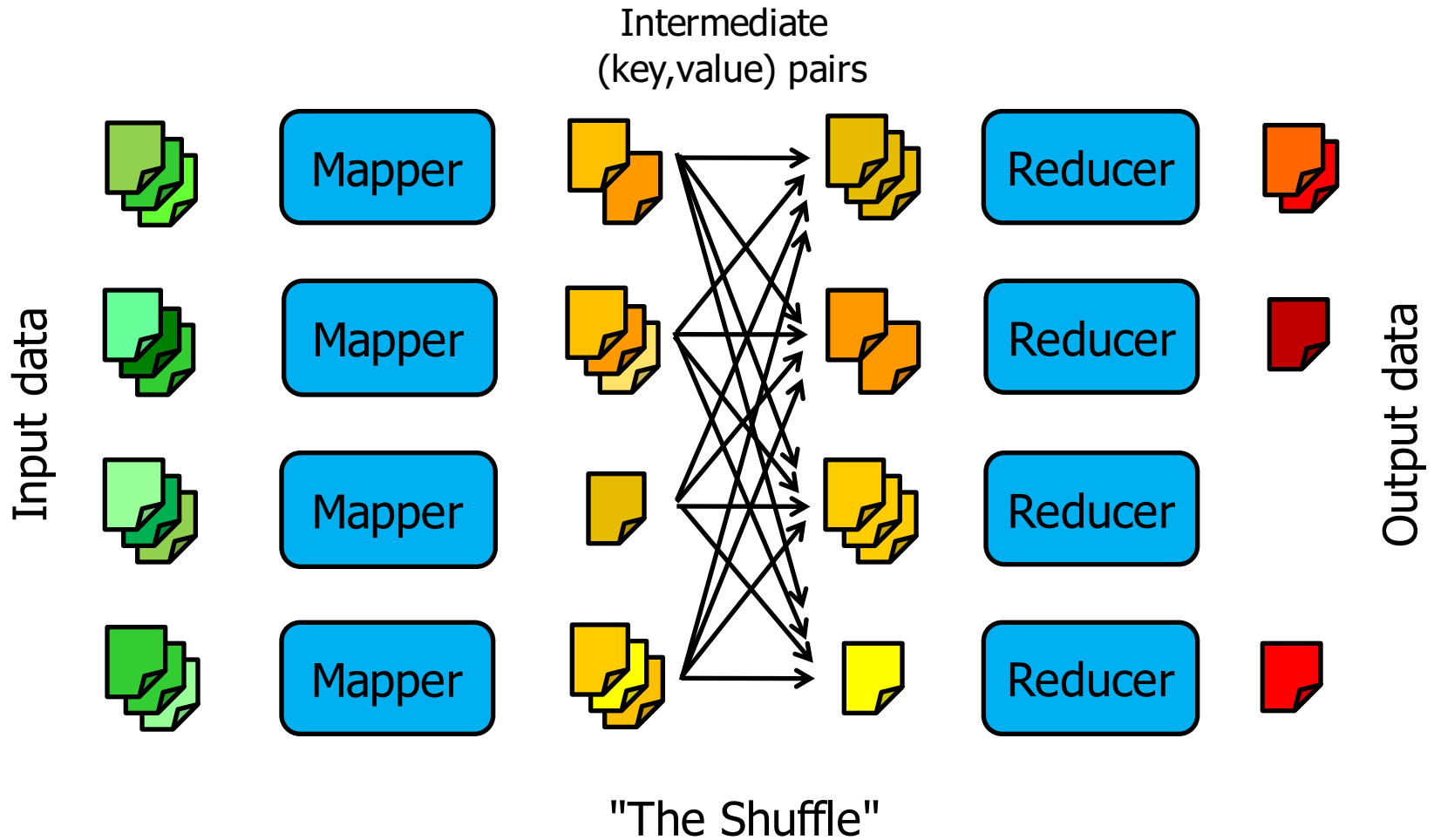
- ▶ **HDFS: A specialized distributed file system**
 - ▶ Good for large amounts of data, sequential reads
 - ▶ Bad for lots of small files, random access, non-append writes
- ▶ **Architecture: Blocks, namenode, datanodes**
 - ▶ File data is broken into large blocks (64MB default)
 - ▶ Blocks are stored & replicated by datanodes
 - ▶ Single namenode manages all the metadata
 - ▶ Secondary namenode: Housekeeping & (some) redundancy
- ▶ **Usage: Special command-line interface**
 - ▶ Example: `hadoop fs -ls /path/in/hdfs`

▶ Plan for today

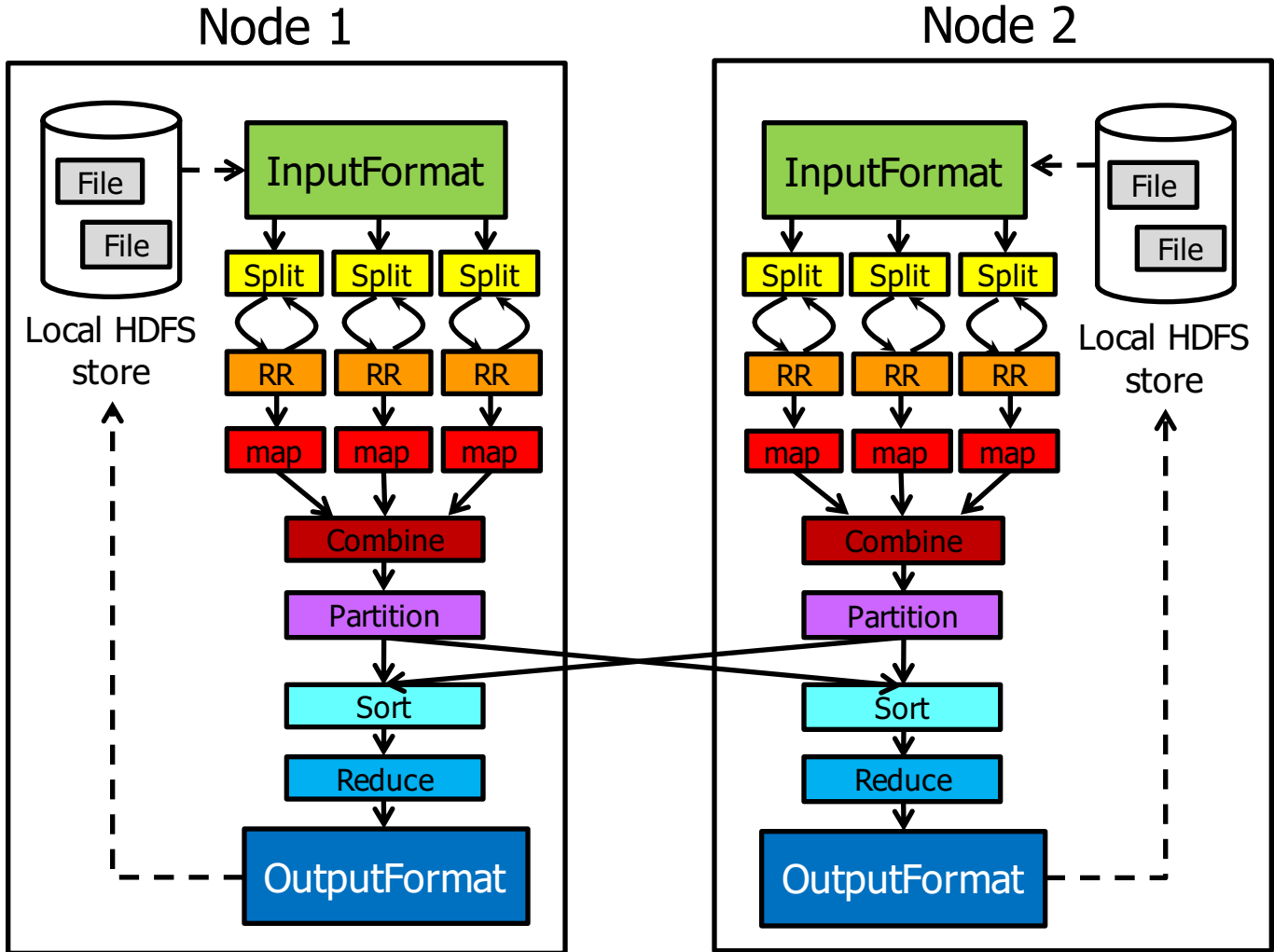
- ▶ A brief history of Hadoop ✓
- ▶ Writing jobs for Hadoop
 - ▶ Mappers, reducers, drivers ✓
 - ▶ Compiling and running a job ✓
- ▶ Hadoop Distributed File System (HDFS)
 - ▶ Node types; read and write operation ✓
 - ▶ Accessing data in HDFS ✓
- ▶ Hadoop internals
 - ▶ Dataflow: Input format, partitioner, combiner, ...
 - ▶ Fully distributed mode: Node types, setup
 - ▶ Fault tolerance; speculative execution



▶ Recap: High-level dataflow

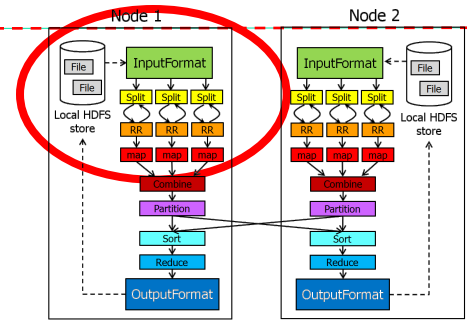


Detailed dataflow in Hadoop



▶ Input Format

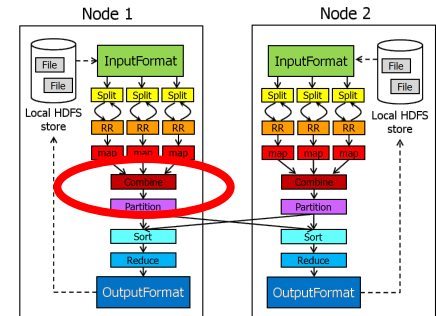
- ▶ Defines which input files should be read, and how
 - ▶ Defaults provided, e.g., TextInputFormat, DBInputFormat, KeyValueTextInputFormat...
- ▶ Defines InputSplits
 - ▶ InputSplits break file into separate tasks
 - ▶ Example: one task for each 64MB block (why?)
- ▶ Provides a factory for RecordReaders
 - ▶ RecordReaders actually read the file into (key,value) pairs
 - ▶ Default format, TextInputFormat, uses byte offset in file as the key, and line as the value
 - ▶ KeyValueInputFormat reads (key,value) pairs from the file directly; key is everything up to the first tab character





Combiners

- ▶ Optional component that can be inserted after the mappers
 - ▶ Input: All data emitted by the mappers on a given node
 - ▶ Output passed to the partitioner

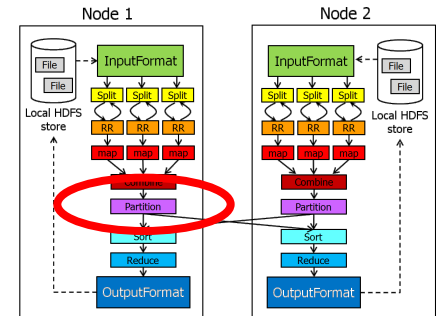


- ▶ Why is this useful?
 - ▶ Suppose your mapper counts words by emitting $(xyz, 1)$ pairs for each word xyz it finds
 - ▶ If a word occurs many times, it is much more efficient to pass (xyz, k) to the reducer, than passing k copies of $(xyz, 1)$



Paritioner

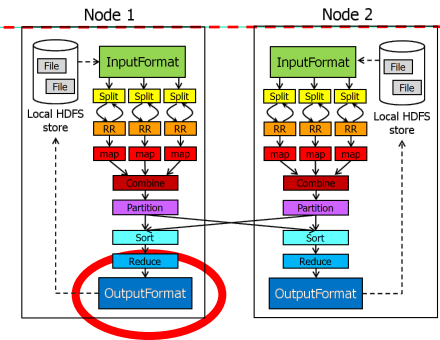
- ▶ Controls which intermediate key-value pairs should go to which reducer



- ▶ Defines a partition on the set of KV pairs
 - ▶ Number of partitions is the same as the number of reducers
- ▶ Default partitioner (HashPartitioner) assigns partition based on a hash of the key

▶ Output Format

- ▶ Counterpart to InputFormat
- ▶ Controls where output is stored, and how
 - ▶ Provides a factory for RecordWriter
- ▶ Several implementations provided
 - ▶ TextOutputFormat (default)
 - ▶ DBOutputFormat
 - ▶ MultipleTextOutputFormat
 - ▶ ...



▶ Recap: Dataflow in Hadoop

- ▶ Hadoop has many components that are usually hidden from the developer
- ▶ Many of these can be customized:
 - ▶ InputFormat: Defines how input files are read
 - ▶ InputSplit: Defines how data portions are assigned to tasks
 - ▶ RecordReader: Reads actual KV pairs from input files
 - ▶ Combiner: Mini-reduce step on each node, for efficiency
 - ▶ Partitioner: Assigns intermediate KV pairs to reducers
 - ▶ Comparator: Controls how KV pairs are sorted after shuffle
- ▶ More details: Chapter 7 of your textbook

▶ Plan for today

- ▶ A brief history of Hadoop ✓
- ▶ Writing jobs for Hadoop
 - ▶ Mappers, reducers, drivers ✓
 - ▶ Compiling and running a job ✓
- ▶ Hadoop Distributed File System (HDFS)
 - ▶ Node types; read and write operation ✓
 - ▶ Accessing data in HDFS ✓
- ▶ Hadoop internals
 - ▶ Dataflow: Input format, partitioner, combiner, ... ✓
 - ▶ Fully distributed mode: Node types, setup ← NEXT
 - ▶ Fault tolerance; speculative execution



Hadoop daemons

- ▶ **TaskTracker**
 - ▶ Runs maps and reduces. One per node.
 - ▶ **JobTracker**
 - ▶ Accepts jobs; assigns tasks to TaskTrackers
 - ▶ **DataNode**
 - ▶ Stores HDFS blocks
 - ▶ **NameNode**
 - ▶ Stores HDFS metadata
 - ▶ **SecondaryNameNode**
 - ▶ Merges edits file with snapshot; "backup" for NameNode
- A single node can run more than one of these!

▶ An example configuration



Small cluster



JobTracker



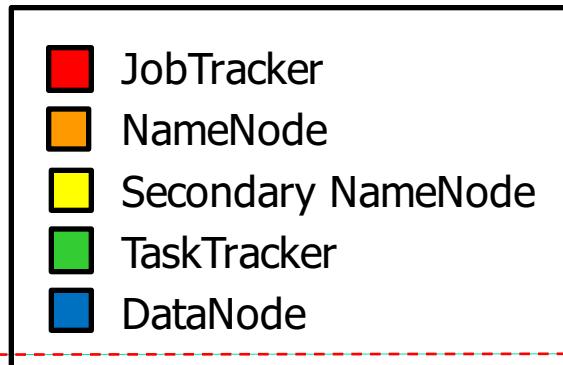
NameNode



Secondary
NameNode



Medium cluster





Fault tolerance

- ▶ What if a node fails during a job?
 - ▶ JobTracker notices that the node's TaskTracker no longer responds; re-executes the failed node's tasks
- ▶ What specifically should be re-executed?
 - ▶ Depends on the phase the job was in
 - ▶ Mapping phase: Re-execute all maps assigned to failed node
 - ▶ Reduce phase: Re-execute all reduces assigned to the node
 - ▶ Is this sufficient?
 - ▶ No! Failed node may also have completed map tasks, and other nodes may not have finished copying out the results
 - ▶ Need to re-execute map tasks on the failed node as well!

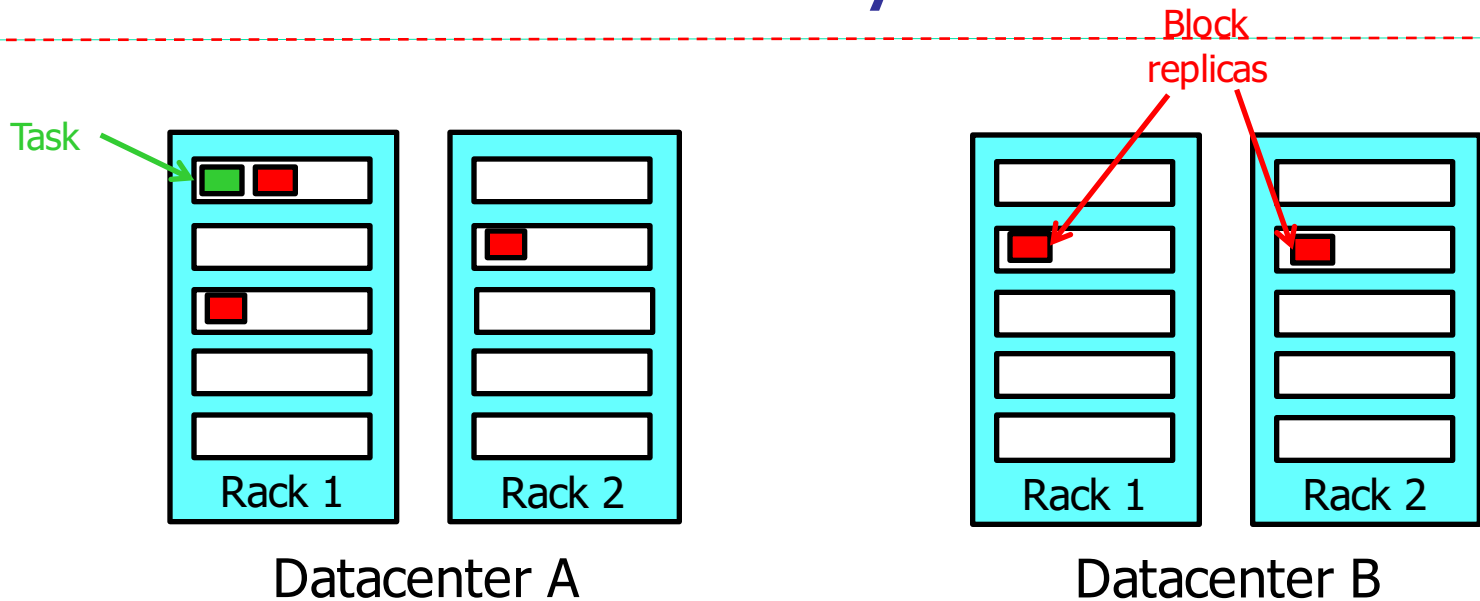


Speculative execution

- ▶ What if some tasks are much harder, or some nodes much slower, than the others?
 - ▶ Entire job is delayed!
- ▶ **Solution: Speculative execution**
 - ▶ If task is almost complete, schedule a few redundant tasks on nodes that have nothing else to do
 - ▶ Whichever one finishes first becomes the definitive copy; the others' results are discarded to prevent duplicates



Placement and locality



- ▶ Which of the replicated blocks should be read?
 - ▶ If possible, pick the closest one (reduces network load)
 - ▶ Distance metric takes into account: Nodes, racks, datacenters
- ▶ Where should the replicas be put?
 - ▶ Tradeoff between fault tolerance and locality/performance




Recap: Distributed mode

- ▶ **Five important daemons:**
 - ▶ MapReduce daemons: JobTracker, TaskTracker
 - ▶ HDFS daemons: DataNode, NameNode, Secondary NameN.
 - ▶ Workers run TaskTracker+DataNode
- ▶ **Special features:**
 - ▶ Transparently re-executes jobs if nodes fail
 - ▶ Speculatively executes jobs to limit impact of stragglers
 - ▶ Rack-aware placement to keep traffic local

Big Data Computations on the cloud

▶ Plan for today

- ▶ Introduction 
 - ▶ Census example
- ▶ MapReduce architecture
 - ▶ Data flow
 - ▶ Execution flow
 - ▶ Fault tolerance etc.



Analogy: National census

- ▶ Suppose we have 10,000 employees, whose job is to collate census forms and to determine how many people live in each city
- ▶ How would you organize this task?

United States Census 2010 U.S. DEPARTMENT OF COMMERCE
Economics and Statistics Administration U.S. CENSUS BUREAU

This is the official form for all the people at this address. It is quick and easy, and your answers are protected by law.

Use a blue or black pen. **Start here**

The Census must count every person living in the United States on April 1, 2010.

Before you answer Question 1, count the people living in this house, apartment, or mobile home using our guidelines.

- Count all people, including babies, who live and sleep here most of the time.

The Census Bureau also conducts counts in institutions and other places, so:

- Do not count anyone living away either at college or in the Armed Forces.
- Do not count anyone in a nursing home, jail, prison, detention facility, etc., on April 1, 2010.
- Leave these people off your form, even if they will return to live here after they leave college, the nursing home, the military, jail, etc. Otherwise, they may be counted twice.

The Census must also include people without a permanent place to stay, so:

- If someone who has no permanent place to stay is staying here on April 1, 2010, count that person. Otherwise, he or she may be missed in the census.

1. How many people were living or staying in this house, apartment, or mobile home on April 1, 2010?
Number of people =

2. Were there any additional people staying here April 1, 2010 that you did not include in Question 1? Mark all that apply.

- Children, such as newborn babies or foster children
- Relatives, such as adult children, cousins, or in-laws
- Nonrelatives, such as roommates or live-in baby sitters
- People staying here temporarily
- No additional people

3. Is this house, apartment, or mobile home — Mark ONE box.

- Owned by you or someone in this household with a mortgage or loan? *Include home equity loans.*
- Owned by you or someone in this household free and clear (without a mortgage or loan)?
- Rented?
- Occupied without payment of rent?

4. What is your telephone number? We may call if we don't understand an answer.
Area Code + Number - -

5. Please provide information for each person living here. Start with a person living here who owns or rents this house, apartment, or mobile home. If the owner or renter lives somewhere else, start with any adult living here. This will be Person 1. What is Person 1's name? Print name below.
Last Name
First Name MI
 Male Female

6. What is Person 1's sex? Mark ONE box.

7. What is Person 1's age and what is Person 1's date of birth? Please report babies as age 0 when the child is less than 1 year old. Print numbers in boxes.
Age on April 1, 2010 Month Day Year of birth

→ NOTE: Please answer BOTH Question 8 about Hispanic origin and Question 9 about race. For this census, Hispanic origins are not races.

8. Is Person 1 of Hispanic, Latino, or Spanish origin?

- No, not of Hispanic, Latino, or Spanish origin
- Yes, Mexican, Mexican Am., Chicano
- Yes, Puerto Rican
- Yes, Cuban
- Yes, another Hispanic, Latino, or Spanish origin — Print origin, for example, Argentinean, Colombian, Dominican, Nicaraguan, Salvadoran, Spaniard, and so on.

9. What is Person 1's race? Mark one or more boxes.

- White
- Black, African Am., or Negro
- American Indian or Alaska Native — Print name of enrolled or principal tribe.
- Asian Indian
- Japanese
- Native Hawaiian
- Chinese
- Korean
- Guamanian or Chamorro
- Filipino
- Vietnamese
- Samoan
- Other Asian — Print race, for example, Hmong, Laotian, Thai, Pakistani, Cambodian, and so on.
- Other Pacific Islander — Print race, for example, Fijian, Tongan, and so on.
- Some other race — Print race.

10. Does Person 1 sometimes live or stay somewhere else?
No Yes — Mark all that apply.

- In college housing
- In the military
- At a seasonal or second residence
- In jail or prison
- In a nursing home or another reason

→ If more people were counted in Question 1, continue with Person 2.

OMB No. 0607-0919-C: Approval Expires 12/31/2011.
Form **D-61** (1-16-2009)

USCENSUSBUREAU

National census "data flow"

▶ Making things more complicated

- ▶ Suppose people take vacations, get sick, work at different rates
- ▶ Suppose some forms are incorrectly filled out and require corrections or need to be thrown away
- ▶ What if the supervisor gets sick?
- ▶ How big should the stacks be?
- ▶ How do we monitor progress?
- ▶ ...



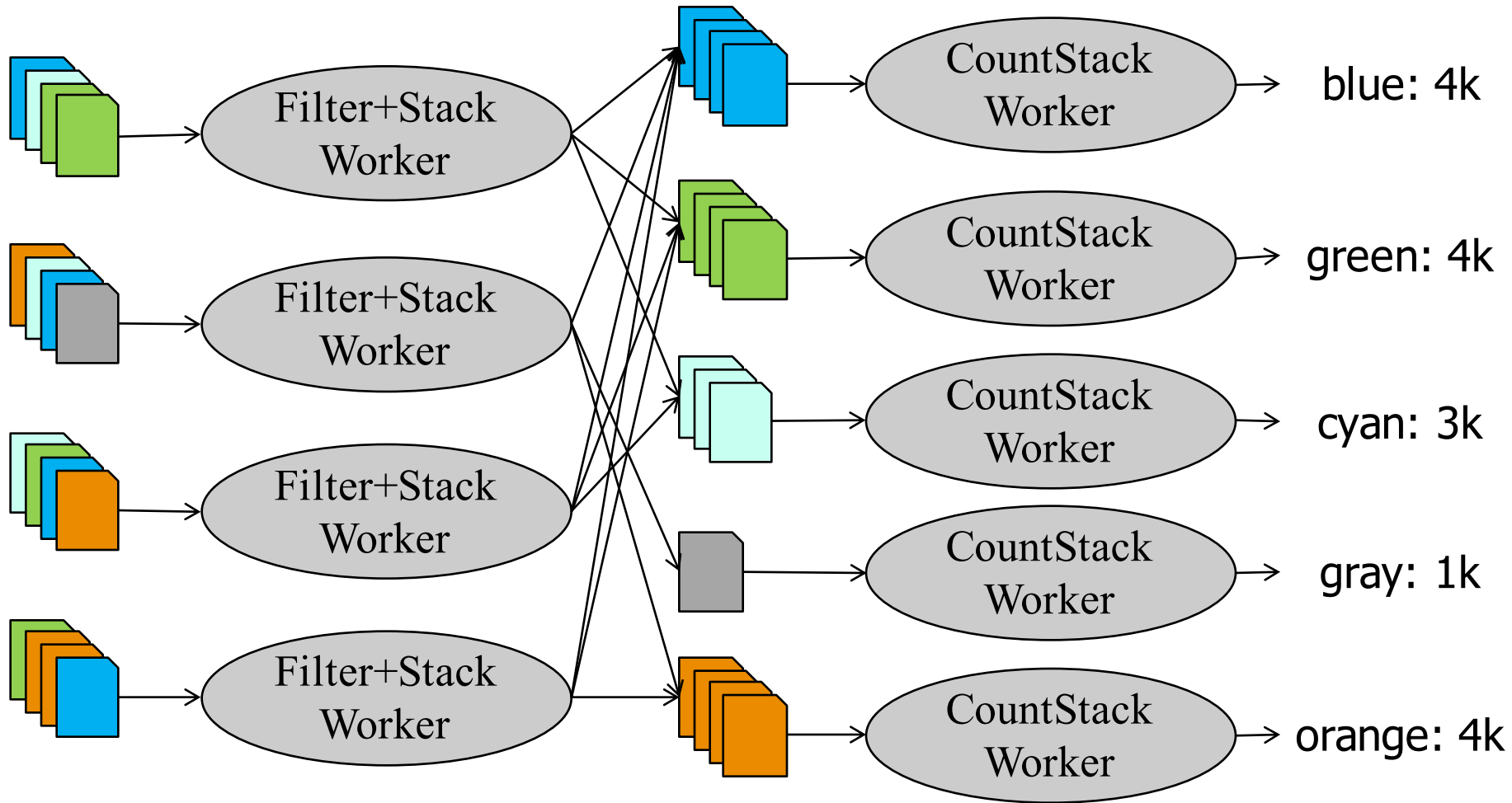
A bit of introspection

- ▶ What is the main challenge?
 - ▶ Are the individual tasks complicated?
 - ▶ If not, what makes this so challenging?
- ▶ How resilient is our solution?
- ▶ How well does it balance work across employees?
 - ▶ What factors affect this?
- ▶ How general is the set of techniques?

▶ I don't want to deal with all this!!!

- ▶ Wouldn't it be nice if there were some system that took care of all these details for you?
- ▶ Ideally, you'd just tell the system what needs to be done
- ▶ That's the MapReduce framework.

▶ Abstracting into a digital data flow





Abstracting once more

- ▶ There are two kinds of workers:
 - ▶ Those that take input data items and produce output items for the “stacks”
 - ▶ Those that take the stacks and **aggregate** the results to produce outputs on a per-stack basis
- ▶ We'll call these:
 - ▶ **map**: takes (item_key, value), produces one or more (stack_key, value') pairs
 - ▶ **reduce**: takes (stack_key, {set of value'}), produces one or more output results – typically (stack_key, agg_value)




We will refer to this key
as the **reduce key**



Why MapReduce?

- ▶ **Scenario:**
 - ▶ You have a huge amount of data, e.g., all the Google searches of the last three years
 - ▶ You would like to perform a computation on the data, e.g., find out which search terms were the most popular
 - ▶ **How would you do it?**
 - ▶ **Analogy to the census example:**
 - ▶ The computation isn't necessarily difficult, but parallelizing and distributing it, as well as handling faults, is challenging
 - ▶ **Idea: A programming language!**
 - ▶ Write a simple program to express the (simple) computation, and let the language runtime do all the hard work
-

▶ Plan for today

- ▶ Introduction 
 - ▶ Census example 
- ▶ MapReduce architecture 
 - ▶ Data flow
 - ▶ Execution flow
 - ▶ Fault tolerance etc.



What is MapReduce?

- ▶ A famous distributed programming model
- ▶ In many circles, considered *the* key building block for much of Google's data analysis
 - ▶ A programming language built on it: Sawzall, <http://labs.google.com/papers/sawzall.html>
 - ▶ ... *Sawzall has become one of the most widely used programming languages at Google. ... [O]n one dedicated Workqueue cluster with 1500 Xeon CPUs, there were 32,580 Sawzall jobs launched, using an average of 220 machines each. While running those jobs, 18,636 failures occurred (application failure, network outage, system crash, etc.) that triggered rerunning some portion of the job. The jobs read a total of 3.2×10^{15} bytes of data (2.8PB) and wrote 9.9×10^{12} bytes (9.3TB).*
 - ▶ Other similar languages: Yahoo's Pig Latin and Pig; Microsoft's Dryad
- ▶ Cloned in open source: Hadoop, <http://hadoop.apache.org/>



The MapReduce programming model

- ▶ Simple distributed functional programming primitives
- ▶ Modeled after Lisp primitives:
 - ▶ `map` (apply function to all items in a collection) and
 - ▶ `reduce` (apply function to set of items with a common key)
- ▶ We start with:
 - ▶ A user-defined function to be applied to all data,
`map`: (key,value) → (key, value)
 - ▶ Another user-specified operation
`reduce`: (key, {set of values}) → result
 - ▶ A set of n nodes, each with data
- ▶ All nodes run `map` on all of their data, producing new data with keys
 - ▶ This data is collected by key, then `shuffled`, and finally `reduced`
 - ▶ Dataflow is through temp files on GFS

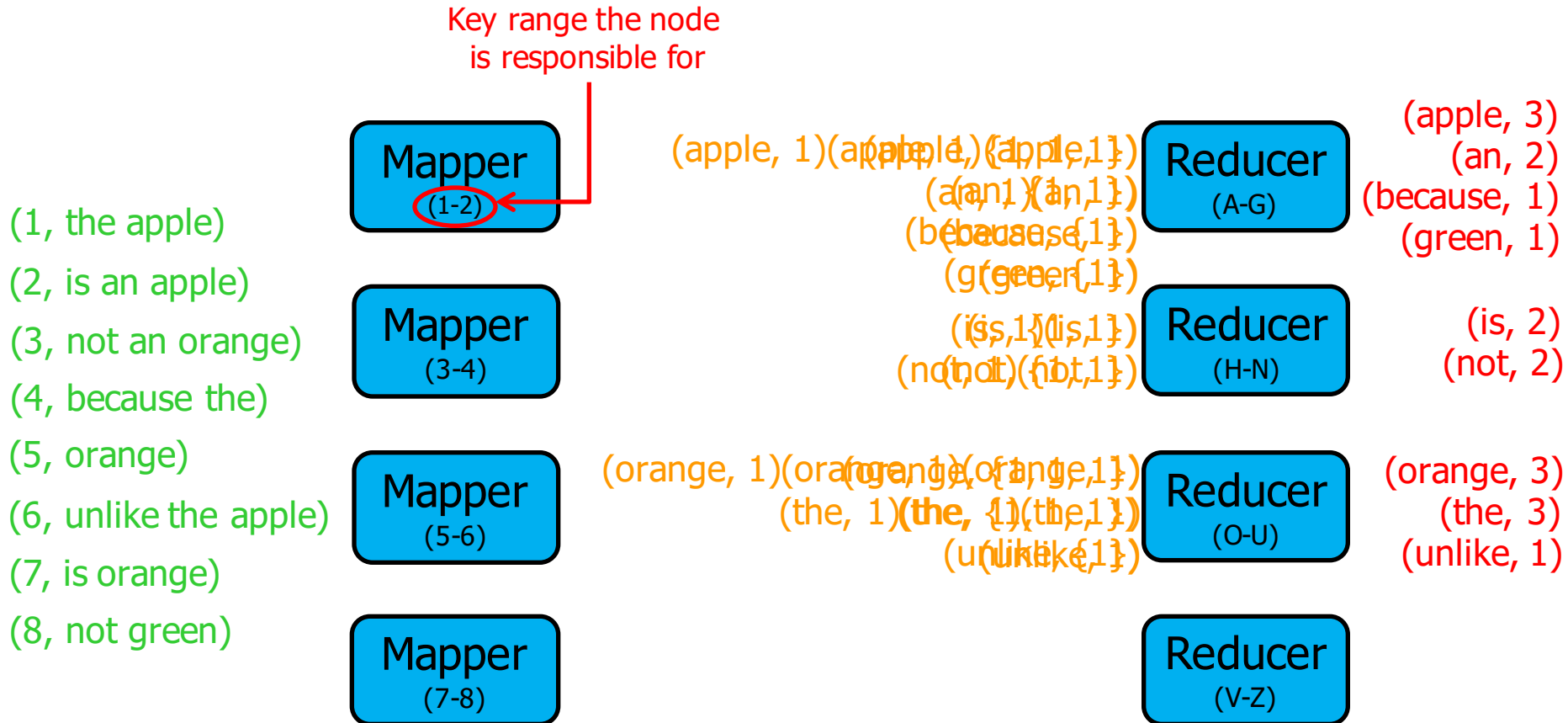
▶ Simple example: Word count

```
map(String key, String value) {  
    // key: document name, line no  
    // value: contents of line  
    for each word w in value:  
        emit(w, "1")  
}
```

```
reduce(String key, Iterator values) {  
    // key: a word  
    // values: a list of counts  
    int result = 0;  
    for each v in values:  
        result += ParseInt(v);  
    emit(key, result)  
}
```

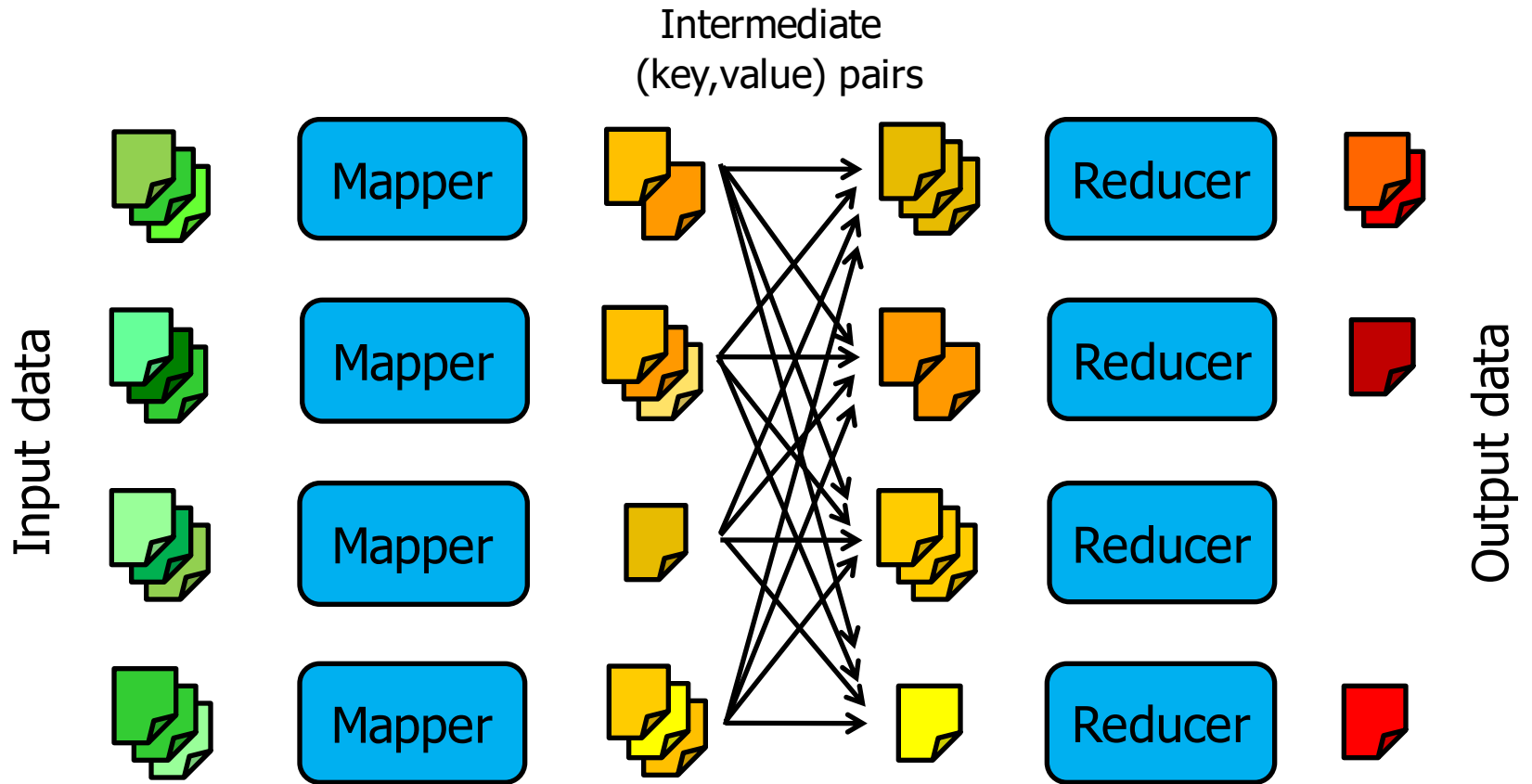
- ▶ Goal: Given a set of documents, count how often each word occurs
 - ▶ Input: Key-value pairs (document:lineNumber, text)
 - ▶ Output: Key-value pairs (word, #occurrences)
 - ▶ What should be the intermediate key-value pairs?

Simple example: Word count



- 1 Each mapper receives some of the KV-pairs as input
- 2 The mappers process the KV-pairs one by one
- 3 Each KV-pair output by the mapper is sent to the reducer that is responsible for it
- 4 The reducers sort their input by key and group it
- 5 The reducers process their input one group at a time

MapReduce dataflow



"The Shuffle"

What is meant by a 'dataflow'?
What makes this so scalable?



More examples

- ▶ Distributed grep – all lines matching a pattern
 - ▶ Map: filter by pattern
 - ▶ Reduce: output set
- ▶ Count URL access frequency
 - ▶ Map: output each URL as key, with count 1
 - ▶ Reduce: sum the counts
- ▶ Reverse web-link graph
 - ▶ Map: output (target,source) pairs when link to target found in source
 - ▶ Reduce: concatenates values and emits (target,list(source))
- ▶ Inverted index
 - ▶ Map: Emits (word,documentID)
 - ▶ Reduce: Combines these into (word,list(documentID))

▶ Common mistakes to avoid

▶ Mapper and reducer should be **stateless**

- ▶ Don't use static variables - after `map` + `reduce` return, they should remember nothing about the processed data!
- ▶ Reason: No guarantees about which key-value pairs will be processed by which workers!

```
HashMap h = new HashMap();  
map(key, value) {  
    if (h.containsKey(key)) {  
        h.add(key, value);  
        emit(key, "X");  
    }  
}
```

Wrong!

▶ Don't try to do your own **I/O!**

- ▶ Don't try to read from, or write to, files in the file system
- ▶ The MapReduce framework does all the I/O for you:

```
map(key, value) {  
    File foo =  
        new File("xyz.txt");  
    while (true) {  
        s = foo.readLine();  
        ...  
    }  
}
```

Wrong!

- ▶ All the incoming data will be fed as arguments to `map` and `reduce`
- ▶ Any data your functions produce should be output via `emit`

▶ More common mistakes to avoid

```
map(key, value) {  
    emit("FOO", key + " " + value);  
}
```

Wrong!

```
reduce(key, value[]) {  
    /* do some computation on  
    all the values */  
}
```

- ▶ Mapper must not map too much data to the same key
 - ▶ In particular, don't map *everything* to the same key!!
 - ▶ Otherwise the reduce worker will be overwhelmed!
 - ▶ It's okay if some reduce workers have more work than others
 - ▶ Example: In WordCount, the reduce worker that works on the key 'and' has a lot more work than the reduce worker that works on 'syzygy'.

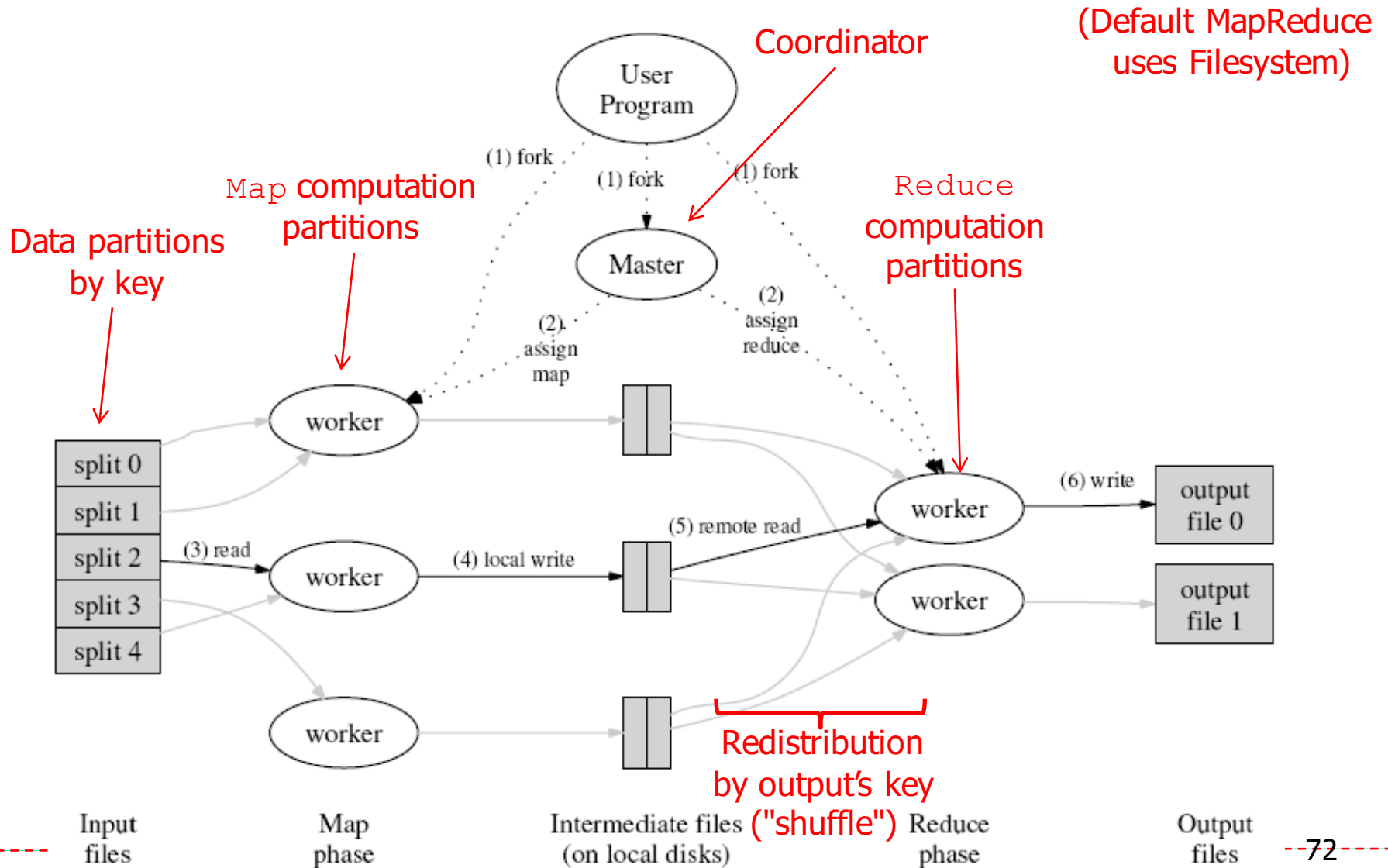


Designing MapReduce algorithms

- ▶ **Key decision: What should be done by `map`, and what by `reduce`?**
 - ▶ `map` can do something to each individual key-value pair, but it can't look at other key-value pairs
 - ▶ Example: Filtering out key-value pairs we don't need
 - ▶ `map` can emit more than one intermediate key-value pair for each incoming key-value pair
 - ▶ Example: Incoming data is text, `map` produces `(word,1)` for each word
 - ▶ `reduce` can aggregate data; it can look at multiple values, as long as `map` has mapped them to the same (intermediate) key
 - ▶ Example: Count the number of words, add up the total cost, ...
 - ▶ **Need to get the intermediate format right!**
 - ▶ If `reduce` needs to look at several values together, `map` must emit them using the same key!
-



More details on the MapReduce data flow





Some additional details

- ▶ To make this work, we need a few more parts...
- ▶ The **file system** (distributed across all nodes):
 - ▶ Stores the inputs, outputs, and temporary results
- ▶ The **driver program** (executes on one node):
 - ▶ Specifies where to find the inputs, the outputs
 - ▶ Specifies what mapper and reducer to use
 - ▶ Can customize behavior of the execution
- ▶ The **runtime system** (controls nodes):
 - ▶ Supervises the execution of tasks
 - ▶ Esp. **JobTracker**



Some details

- ▶ Fewer computation partitions than data partitions
 - ▶ All data is accessible via a distributed filesystem with replication
 - ▶ Worker nodes produce data in key order (makes it easy to merge)
 - ▶ The master is responsible for scheduling, keeping all nodes busy
 - ▶ The master knows how many data partitions there are, which have completed – atomic commits to disk
- ▶ **Locality:** Master tries to do work on nodes that have replicas of the data
- ▶ Master can deal with stragglers (slow machines) by re-executing their tasks somewhere else

▶ What if a worker crashes?

- ▶ We rely on the file system being shared across all the nodes
- ▶ Two types of (crash) faults:
 - ▶ Node wrote its output and then crashed
 - ▶ Here, the file system is likely to have a copy of the complete output
 - ▶ Node crashed before finishing its output
 - ▶ The JobTracker sees that the job isn't making progress, and restarts the job elsewhere on the system
- ▶ (Of course, we have fewer nodes to do work...)
- ▶ But what if the master crashes?



Other challenges

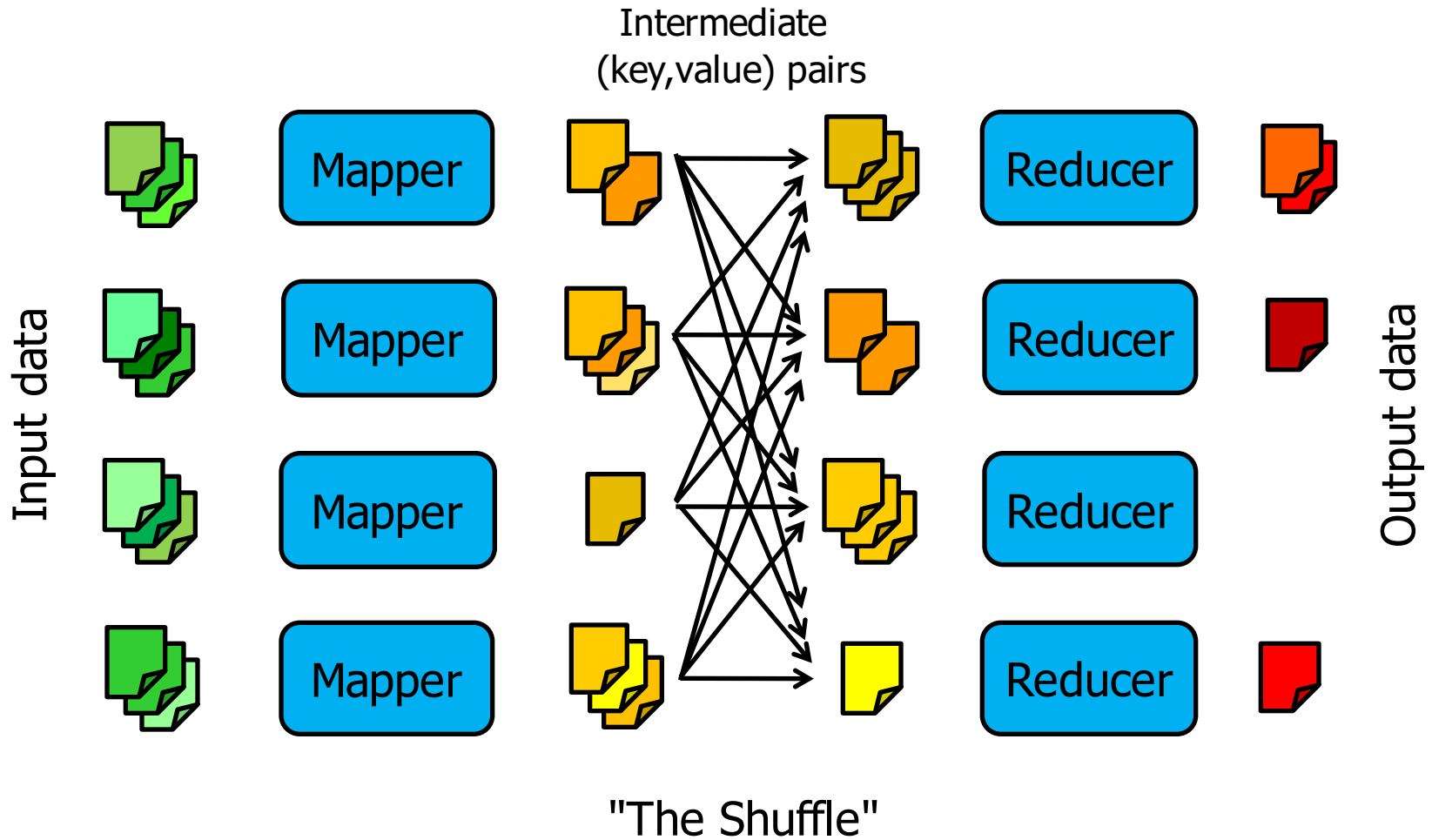
- ▶ **Locality**
 - ▶ Try to schedule map task on machine that already has data
- ▶ **Task granularity**
 - ▶ How many map tasks? How many reduce tasks?
- ▶ **Dealing with stragglers**
 - ▶ Schedule some backup tasks
- ▶ **Saving bandwidth**
 - ▶ E.g., with combiners
- ▶ **Handling bad records**
 - ▶ "Last gasp" packet with current sequence number



Scale and MapReduce

- ▶ From a particular Google paper on a language built over MapReduce:
 - ▶ ... Sawzall has become one of the most widely used programming languages at Google. ...
[O]n one dedicated Workqueue cluster with 1500 Xeon CPUs, there were 32,580 Sawzall jobs launched, using an average of 220 machines each.
While running those jobs, **18,636** failures occurred (application failure, network outage, system crash, etc.) that triggered rerunning some portion of the job. The jobs read a total of **3.2×10^{15} bytes of data (2.8PB)** and wrote **9.9×10^{12} bytes (9.3TB)**.

▶ Recap: MapReduce dataflow





Recap: MapReduce

```
map(key:URL, value:Document)
{
    String[] words = value.split(" ");
    for each w in words
        emit(w, 1);
}

reduce(rkey:String, rvalues:Integer[])
{
    Integer result = 0;
    foreach v in rvalues
        result = result + v;
    emit(rkey, v);
}
```

These types depend on the input data

Produces intermediate key-value pairs that are sent to the reducer

These types can be (and often are) different from the ones in map()

reduce gets all the intermediate values with the same rkey

Both map() and reduce() are stateless: Can't have a 'global variable that is preserved across invocations!

Any key-value pairs emitted by the reducer are added to the final output

Plan for today

- ▶ **Single-pass algorithms in MapReduce** 
 - ▶ Filtering algorithms
 - ▶ Aggregation algorithms
 - ▶ Intersections and joins
 - ▶ Partial Cartesian products
 - ▶ **Sorting**

▶ The basic idea

- ▶ Let's consider single-pass algorithms
- ▶ Need to take the algorithm and break it into filter/collect/aggregate steps
 - ▶ Filter/collect becomes part of the `map` function
 - ▶ Collect/aggregate becomes part of the `reduce` function
- ▶ Note that sometimes we may need multiple map / reduce stages – chains of maps and reduces

- ▶ Let's see some examples



Filtering algorithms

- ▶ Goal: Find lines/files/tuples with a particular characteristic
- ▶ Examples:
 - ▶ grep Web logs for requests to *.upenn.edu/*
 - ▶ find in the Web logs the hostnames accessed by 192.168.2.1
 - ▶ locate all the files that contain the words 'Apple' and 'Jobs'
- ▶ Generally: **map** does most of the work, **reduce** may simply be the identity

▶ Aggregation algorithms

- ▶ Goal: Compute the maximum, the sum, the average, ..., over a set of values
- ▶ Examples:
 - ▶ Count the number of requests to *.upenn.edu/*
 - ▶ Find the most popular domain
 - ▶ Average the number of requests per page per Web site
- ▶ Often: `map` may be simple or the identity



A more complex example

- ▶ **Goal: Billing for a CDN like Amazon CloudFront**
 - ▶ **Input: Log files from the edge servers. Two files per domain:**
 - ▶ `access_log-www.foo.com-20111006.txt`: HTTP accesses
 - ▶ `ssl_access_log-www.foo.com-20111006.txt`: HTTPS accesses
 - ▶ **Example line:**

```
158.130.53.72 - - [06/Oct/2011:16:30:38 -0400] "GET /largeFile.ISO HTTP/1.1" 200 8130928734 "-" "Mozilla/5.0 (compatible; MSIE 5.01; Win2000)"
```
 - ▶ Mapper receives (filename,line) tuples
 - ▶ **Billing policy (simplified):**
 - ▶ Billing is based on a mix of request count and data traffic (why?)
 - ▶ 10,000 HTTP requests cost \$0.0075
 - ▶ 10,000 HTTPS requests cost \$0.0100
 - ▶ One GB of traffic costs \$0.12
 - ▶ **Desired output is a list of (domain, grandTotal) tuples**



Intersections and joins

- ▶ Goal: Intersect multiple different inputs on some shared values
 - ▶ Values can be equal, or meet a certain predicate
- ▶ Examples:
 - ▶ Find all documents with the words “data” and “centric” given an inverted index
 - ▶ Find all professors and students in common courses and return the pairs <professor,student> for those cases



Partial Cartesian products

- ▶ Goal: Find some complex relationship, e.g., based on pairwise distance
- ▶ Examples:
 - ▶ Find all pairs of sites within 100m of each other
- ▶ Generally hard to parallelize
 - ▶ But may be possible if we can divide the input into bins or tiles, or link it to some sort of landmark
 - ▶ Overlap the tiles? (how does this scale?)
 - ▶ Generate landmarks using clustering?

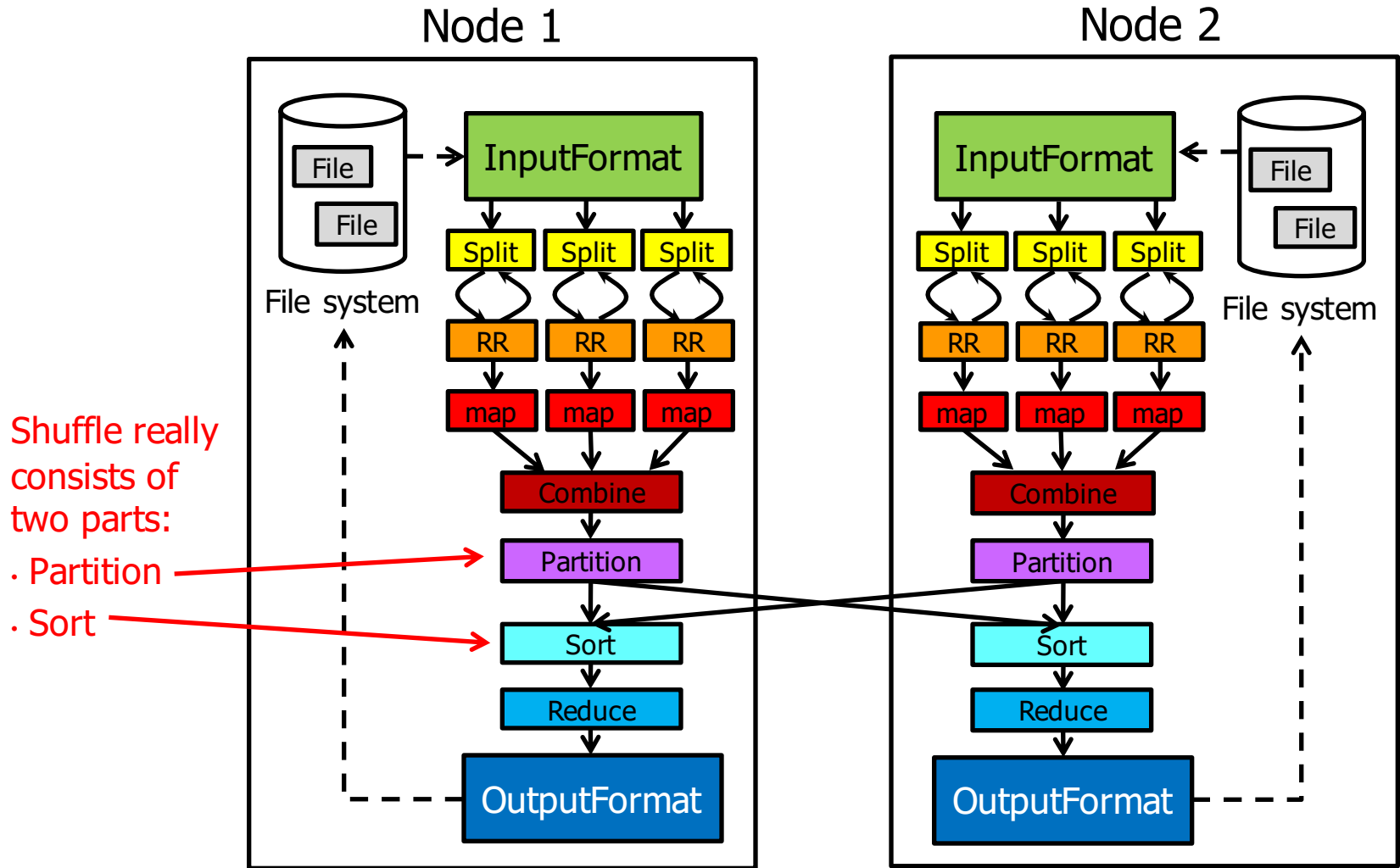
Sorting

- ▶ Goal: Sort input
- ▶ Examples:
 - ▶ Return all the domains covered by Google's index and the number of pages in each, ordered by the number of pages
- ▶ The programming model does not support this *per se*, but the implementations do
 - ▶ Let's take a look at what happens in the Shuffle stage

▶ Plan for today

- ▶ Single-pass algorithms in MapReduce ✓
 - ▶ Filtering algorithms ✓
 - ▶ Aggregation algorithms ✓
 - ▶ Intersections and joins ✓
 - ▶ Partial Cartesian products ✓
 - ▶ **Sorting** ← NEXT

▶ The shuffle stage revisited



▶ Shuffle as a sorting mechanism

- ▶ We can exploit the per-node sorting operation done by the Shuffle stage
 - ▶ If we have a single reducer, we will get sorted output
 - ▶ If we have multiple reducers, we can get partly sorted output (or better – consider an order-preserving hash)
 - ▶ Note it's quite easy to write a last-pass file that merges all of the part-r-000x files
 - ▶ We can use a **heap** to do this
- ▶ Let's see an example!
 - ▶ Return all the domains covered by Google's index and the number of pages in each, ordered by the number of pages

▶ Strengths and weaknesses

- ▶ What problems can you solve well with MapReduce?
 - ▶ ... in a single pass?
 - ▶ ... in multiple passes?
- ▶ Are there problems you cannot solve efficiently with MapReduce?
- ▶ Are there problems it can't solve at all?
- ▶ How does it compare to other ways of doing large-scale data analysis?
 - ▶ Is MapReduce always the fastest/most efficient way?


▶ Recap: MapReduce algorithms

- ▶ A variety of different tasks can be expressed as a single-pass MapReduce program
 - ▶ Filtering and aggregation + combinations of the two
 - ▶ Joins on shared elements
 - ▶ If we allow multiple MapReduce passes or even fixpoint iteration, we can do even more (see later)
- ▶ But it does not work for all tasks
 - ▶ Partial Cartesian product not an ideal fit, but can be made to work with binning and tiling
 - ▶ Sorting doesn't work at all, at least in the abstract model, but the implementations support it

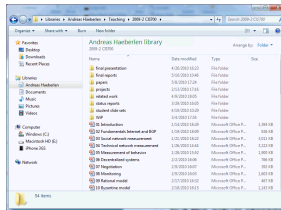
Big Data Storage on the cloud

Key-Value Stores

▶ Plan for today

- ▶ **Key-value stores (KVS)** 
 - ▶ Basic concept; operations
 - ▶ Examples of KVS
 - ▶ KVS and concurrency
 - ▶ Key-multi-value stores; cursors
- ▶ **Key-value stores in the Cloud**
 - ▶ Challenges
 - ▶ Specialized KVS
- ▶ **Two implementations**
 - ▶ Amazon S3
 - ▶ Amazon SimpleDB

▶ Complex service, simple storage



Variable-size files
- read, write, append
- move, rename
- lock, unlock
- ...

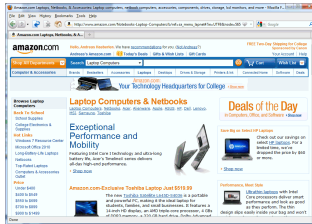
Operating system



Fixed-size blocks
- read
- write

- ▶ PC users see a rich, powerful interface
 - ▶ Hierarchical namespace (directories); can move, rename, append to, truncate, (de)compress, view, delete files, ...
- ▶ But the actual storage device is very simple
 - ▶ HDD only knows how to read and write fixed-size data blocks
- ▶ Translation done by the operating system

▶ Analogy to cloud storage



Shopping carts
Friend lists
User accounts
Profiles
...


Web service

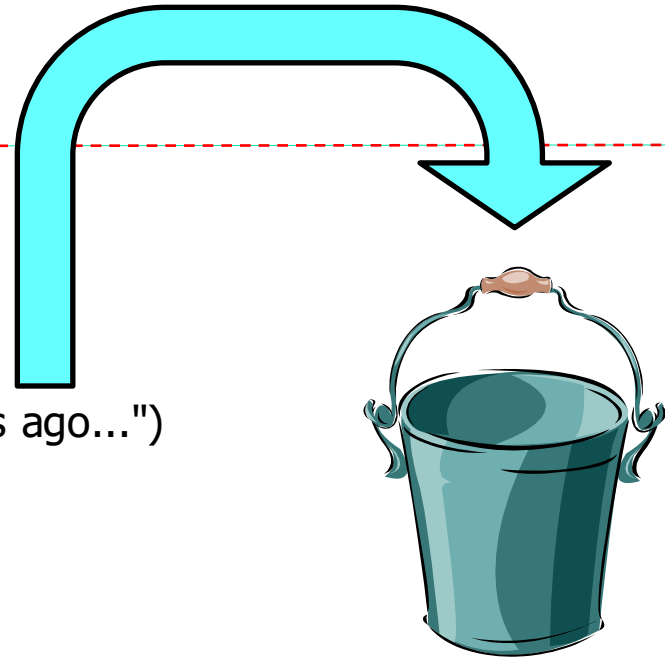


Key/value store
- read, write
- delete

- ▶ Many cloud services have a similar structure
 - ▶ Users see a rich interface (shopping carts, product categories, searchable index, recommendations, ...)
- ▶ But the actual storage service is very simple
 - ▶ Read/write 'blocks', similar to a giant hard disk
- ▶ Translation done by the web service

▶ Key-value stores

Keys Values
↓ ↓
(bob, bschmitt@foo.com)
(gettysburg, "Four score and seven years ago...")
(29ck2dxa1, 0128ckso1\$9#*!!8349e)
(windows, )



- ▶ The **key-value store (KVS)** is a simple **abstraction** for managing persistent state
 - ▶ Data is organized as (key,value) pairs
 - ▶ Only three basic operations:
 - ▶ PUT(key, value)
 - ▶ GET(key) → value
 - ▶ Delete(key)



Examples of KVS

- ▶ Where have you seen this concept before?
- ▶ Conventional examples outside the cloud:
 - ▶ In-memory **associative arrays** and **hash tables** – limited to a single application, only persistent until program ends
 - ▶ On-disk indices (like BerkeleyDB)
 - ▶ "Inverted indices" behind search engines
 - ▶ Database management systems – multiple KVSs++
 - ▶ Distributed hashtables (e.g., on top of Chord/Pastry)

▶ Plan for today

- ▶ Key-value stores (KVS) ✓
 - ▶ Basic concept; operations ✓
 - ▶ Examples of KVS ✓
 - ▶ KVS and concurrency ← NEXT
 - ▶ Key-multi-value stores; cursors
- ▶ Key-value stores in the Cloud
 - ▶ Challenges
 - ▶ Specialized KVS
- ▶ Two implementations
 - ▶ Amazon S3
 - ▶ Amazon SimpleDB

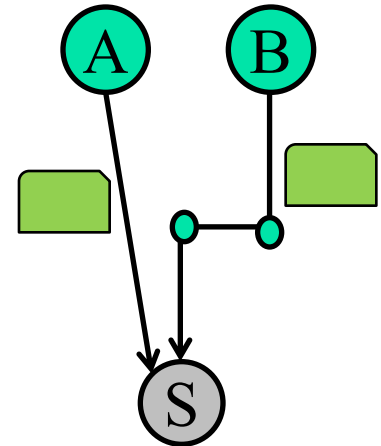
▶ Supporting an Internet service with a KVS

- ▶ We'll do this through a central server, e.g., a Web or application server

- ▶ Two main issues:

1. There may be **multiple concurrent requests** from different clients
 - ▶ These might be GETs, PUTs, DELETES, etc.

2. These requests may come from different parts of the network, with **message propagation delays**
 - ▶ It takes a while for a request to make it to the server!
 - ▶ We'll have to handle requests in the order received (why?)



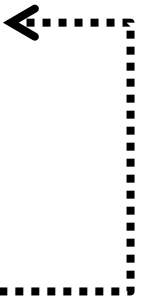
▶ Managing concurrency in a KVS

- ▶ What happens if we do multiple GET operations in parallel?
 - ▶ ... over different keys?
 - ▶ ... over the same key?
- ▶ What if we do multiple PUT operations in parallel? or a GET and a PUT?
- ▶ What is the unit of protection (**concurrency control**) that is necessary here?



Concurrency control

- ▶ Most systems use **locks** on individual items
 - ▶ Each requestor asks for the lock
 - ▶ A **lock manager** processes these requests (typically in FIFO order) as follows:
 - ▶ Lock manager grants the lock to a requestor
 - ▶ Requestor makes modifications
 - ▶ Then releases the lock when it's done
- ▶ There are several kinds of locks, and several other alternatives
 - ▶ Example: S/X lock
 - ▶ See CIS 455 for more details



Limitations of per-key concurrency control

- ▶ Suppose I want to transfer credits from my WoW account to my friend's?
 - ▶ ... while someone else is doing a GET on my (and her) credit amounts to see if they want to trade?
- ▶ This is where one needs a **database management system (DBMS)** or **transaction processing manager (app server)**
 - ▶ Allows for "locking" at a higher level, across keys and possibly even systems (see CIS 330 for more details)
- ▶ Could you implement higher-level locks within the KVS? If so, how?



▶ Plan for today

- ▶ Key-value stores (KVS) ✓
 - ▶ Basic concept; operations ✓
 - ▶ Examples of KVS ✓
 - ▶ KVS and concurrency ✓
 - ▶ Key-multi-value stores; cursors ← NEXT
- ▶ Key-value stores in the Cloud
 - ▶ Challenges
 - ▶ Specialized KVS
- ▶ Two implementations
 - ▶ Amazon S3
 - ▶ Amazon SimpleDB



Key-Multi-Value stores

- ▶ What if I want to have multiple values for the same key in a KVS?
 - ▶ Example: Multiple images with the same search keyword
- ▶ **Option 1:** Make the “value” a collection object like a set
 - ▶ Then PUT really becomes GET → add → PUT
- ▶ **Option 2:** Allow the KVS to store multiple values per key
 - ▶ Requires a **cursor** that scrolls through the matches
 - ▶ Similar to Java's notion of an iterator

▶ Accessing data with a cursor

- ▶ How can we retrieve all the values a particular key maps to?
 - ▶ There could be a very large number of them (remember HW1MS1!)
- ▶ Idea: Use a **cursor**
 - ▶ Follows the following programming pattern:

```
cursor = kvs.getFirstMatch(key);  
  
while (cursor != null) {  
    value = cursor.getValue();  
    cursor = kvs.getNextMatch(key, cursor);  
}
```



Recap: Key-value stores

- ▶ **KVS: A simple abstraction for managing persistent data state**
 - ▶ Interface consists only of PUT and GET (+possibly DELETE)
 - ▶ Some variants allow multiple values per key
 - ▶ Examples: Distributed hash tables, associative arrays, ...
 - ▶ Extremely scalable implementations exist
- ▶ **Challenge: Concurrency control**
 - ▶ From the perspective of the KVS, values for different keys are independent
 - ▶ Difficult to change multiple values atomically
 - ▶ Some applications may require higher-level locking

▶ Plan for today

- ▶ Key-value stores (KVS) ✓
 - ▶ Basic concept; operations ✓
 - ▶ Examples of KVS ✓
 - ▶ KVS and concurrency ✓
 - ▶ Key-multi-value stores; cursors ✓
- ▶ Key-value stores in the Cloud ← NEXT
 - ▶ Challenges
 - ▶ Specialized KVS
- ▶ Two implementations
 - ▶ Amazon S3
 - ▶ Amazon SimpleDB

▶ Key-Value stores on the Cloud

- ▶ Many situations need hosting of large data sets
 - ▶ Examples: Amazon catalog, eBay listings, Facebook pages, ...
- ▶ Ideal: Abstraction of a 'big disk in the clouds', which would have:
 - ▶ Perfect **durability** – nothing would ever disappear in a crash
 - ▶ 100% **availability** – we could always get to the service
 - ▶ Zero **latency** from anywhere on earth – no delays!
 - ▶ Minimal **bandwidth utilization** – we only send across the network what we absolutely need
 - ▶ **Isolation** under concurrent updates – make sure data stays consistent



The inconveniences of the real world

- ▶ Why isn't this feasible?
- ▶ The “cloud” exists over a physical network
 - ▶ Communication takes time, esp. across the globe
 - ▶ Bandwidth is limited, both on the backbone and endpoint
- ▶ The “cloud” has imperfect hardware
 - ▶ Hard disks crash
 - ▶ Servers crash
 - ▶ Software has bugs
- ▶ Can you map these to the previous desiderata?



Finding the right tradeoff

- ▶ In practice, we can't have everything
 - ▶ ... but most applications don't really need 'everything'!
- ▶ Some observations:
 1. **Read-only** (or read-mostly) data is easiest to support
 - ▶ Replicate it everywhere! No concurrency issues!
 - ▶ But only some kinds of data fit this pattern – examples?
 2. **Granularity matters**: “Few large-object” tasks generally tolerate longer latencies than “many small-object” tasks
 - ▶ Fewer requests, often more processing at the client
 - ▶ But it's much more expensive to replicate or to update!
 3. Maybe it makes sense to develop **separate solutions** for large read-mostly objects vs. small read-write objects!
 - ▶ Different requirements → different technical solutions



Specialized KVS

- ▶ Cloud KVS are often specialized for a particular tradeoff or usage scenario

- ▶ Example: Amazon's solutions
 - ▶ **Simple Storage Service (S3):**
 - ▶ large objects – files, virtual machines, etc.
 - ▶ assumes objects change infrequently
 - ▶ objects are opaque to the storage system
 - ▶ **SimpleDB:**
 - ▶ small objects – Java objects, records, etc.
 - ▶ generally updated more frequently; greater need for consistency
 - ▶ generally multiple attributes or properties, which are exposed to the storage system



Recap: KVS on the cloud

- ▶ Ideally, we would simply like the abstraction of a 'big disk in the cloud'
 - ▶ Perfect durability, availability, consistency, throughput, ...
- ▶ Practical constraints require compromises
 - ▶ Propagation delay, unreliable hardware/software, ...
- ▶ Hence, we need to make the right tradeoff
 - ▶ For example, specialize KVS for particular workloads
 - ▶ No one-size-fits-all solution; different solutions are useful in different situations

▶ Plan for today

- ▶ Key-value stores (KVS) ✓
 - ▶ Basic concept; operations ✓
 - ▶ Examples of KVS ✓
 - ▶ KVS and concurrency ✓
 - ▶ Key-multi-value stores; cursors ✓
- ▶ Key-value stores in the Cloud ✓
 - ▶ Challenges ✓
 - ▶ Specialized KVS ✓
- ▶ Two implementations ← NEXT
 - ▶ Amazon S3
 - ▶ Amazon SimpleDB



Big Objects: Amazon S3

- ▶ S3 = Simple Storage System
 - ▶ Think roughly of an Internet file system
- ▶ Stores large **objects** (=values) that may have **access permissions**
 - ▶ Used in “cloud backup” services like Jungle Disk
 - ▶ Used to distribute software packages
 - ▶ Used internally by Amazon to store virtual machines
- ▶ “Up to 99.999999999% durability, 99.99% availability” (“ten nines” and “four nines”)

▶ S3: Key concepts

- ▶ S3 consists of:
 - ▶ **objects** – named items stored in S3
 - ▶ **buckets** of objects – think of these as volumes in a filesystem
 - ▶ the console includes a notion of **folders**, but these are not intrinsic to S3



- ▶ Names within a bucket must uniquely identify a single object
 - ▶ i.e., keys must be unique



S3: Keys and objects

- ▶ What can we use as keys?
 - ▶ Keys can be any string
- ▶ What can we use as objects?
 - ▶ Objects can be from 1 byte to 5 TB, any format
 - ▶ Number of objects is 'unlimited'
- ▶ Where can objects be stored?
 - ▶ Can be assigned to specific geographic regions (Washington, Virginia, California, Ireland, Singapore, Tokyo, ...)
 - ▶ Why is this important? (name at least four reasons!)

low latency to customer
minimize fault correlation

regulatory/legal requirements
low-storage-cost regions

▶ S3: Different ways to access objects

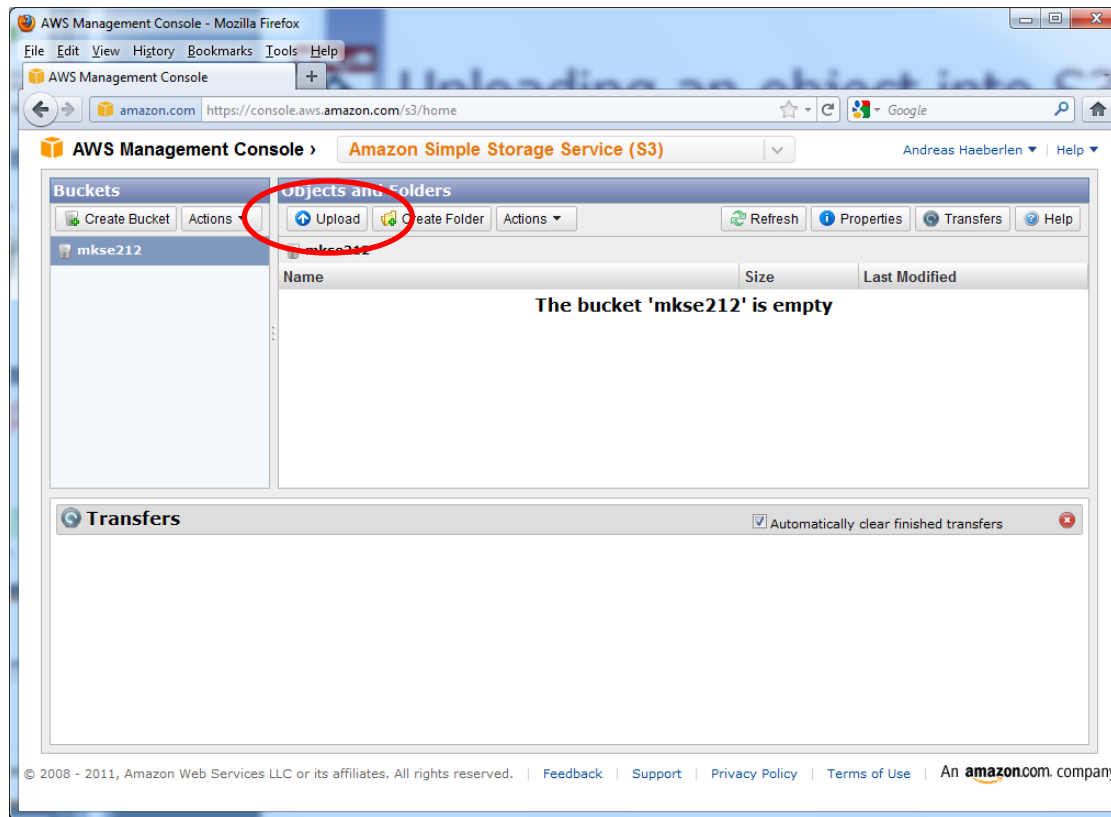
- ▶ Objects in S3 can be accessed
 - ▶ ... via REST or SOAP
 - ▶ ... via BitTorrent
 - ▶ ... over the web: <http://s3.amazonaws.com/bucket/key>
 - ▶ Web Services use HTTP (the Web browser protocol over sockets) and XML to send requests and data
 - ▶ AWS Console also enables configuration
- ▶ We'll mostly be using Java(script) libraries to interact with S3
 - ▶ You'll just call them as normal functions, but they will open and close sockets as necessary
 - ▶ <http://bitbucket.org/jmurty/jets3t/wiki/Home>
 - ▶ <http://aws.amazon.com/sdkfornodejs/>



S3: Access permissions

- ▶ Permissions are assigned through **Access Control Lists (ACLs)**
 - ▶ Essentially, a list of users/groups → permissions
 - ▶ Bucket permissions are inherited by objects unless overridden at the object level
- ▶ What can you control?
 - ▶ Can be at the level of buckets or individual objects
 - ▶ Available rights: Read, write, read ACL, write ACL
 - ▶ Possible grantees: Everyone, authenticated users, specific users (by AWS account email address)

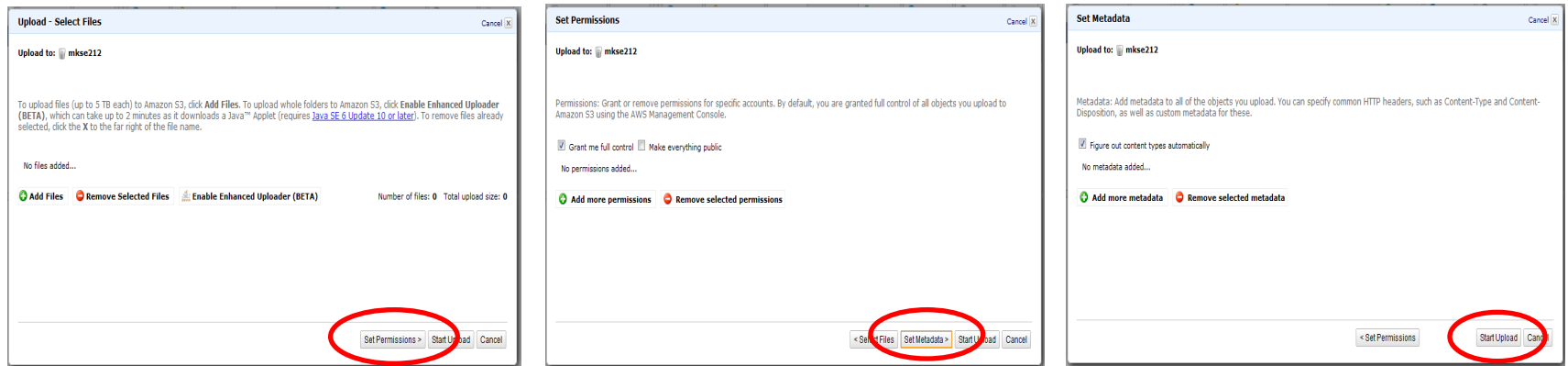
▶ S3: Uploading an object



- ▶ Step 1: Hit 'upload' in management console



S3: Uploading an object



- ▶ Step 2: Select files
- ▶ Step 3: Set metadata (or accept default)
- ▶ Step 4: Set permissions (or make public)

▶ S3: Current pricing and usage

Region: US Standard	Standard Storage	Reduced Redundancy Storage	Glacier Storage
First 1 TB / month	\$0.095 per GB	\$0.076 per GB	\$0.010 per GB
Next 49 TB / month	\$0.080 per GB	\$0.064 per GB	\$0.010 per GB
Next 450 TB / month	\$0.070 per GB	\$0.056 per GB	\$0.010 per GB
Next 500 TB / month	\$0.065 per GB	\$0.052 per GB	\$0.010 per GB
Next 4000 TB / month	\$0.060 per GB	\$0.048 per GB	\$0.010 per GB
Over 5000 TB / month	\$0.055 per GB	\$0.037 per GB	\$0.010 per GB

Request Pricing

Region: US Standard	Pricing
PUT, COPY, POST, or LIST Requests	\$0.005 per 1,000 requests
Glacier Archive and Restore Requests	\$0.05 per 1,000 requests
Delete Requests	Free †
GET and all other Requests	\$0.004 per 10,000 requests
Glacier Data Restores	Free ††
† No charge for delete requests of Standard or RRS objects. For objects that are archived to Glacier, there is a pro-rated charge of \$0.03 per gigabyte for objects deleted prior to 90 days. Learn more.	
†† Glacier is designed with the expectation that restores are infrequent and unusual, and data will be stored for extended periods of time. You can restore up to 5% of your average monthly Glacier storage (pro-rated daily) for free each month. If you choose to restore more than this amount of data in a month, you are charged a restore fee starting at \$0.01 per gigabyte. Learn more.	

Data Transfer Pricing

The pricing below is based on data transferred "in" to and "out" of Amazon S3.

Region: US Standard	Pricing
Data Transfer IN To Amazon S3	
All data transfer in	\$0.000 per GB
Data Transfer OUT From Amazon S3 To	
Amazon EC2 in the Northern Virginia Region	\$0.000 per GB
Another AWS Region or Amazon CloudFront	\$0.020 per GB
Data Transfer OUT From Amazon S3 To Internet	
First 1 GB / month	\$0.000 per GB
Up to 10 TB / month	\$0.120 per GB
Next 40 TB / month	\$0.090 per GB
Next 100 TB / month	\$0.070 per GB
Next 350 TB / month	\$0.050 per GB

http://aws.amazon.com/s3/ (9/19/2013)



S3: Bucket operations

- ▶ Create bucket
(optionally versioned;
see later)
- ▶ Delete bucket

Create a Bucket - Select a Bucket Name and Region Cancel

A bucket is a container for objects stored in Amazon S3. When creating a bucket, you can choose a Region to optimize for latency, minimize costs, or address regulatory requirements. For more information regarding bucket naming conventions, please visit the [Amazon S3 documentation](#).

Bucket Name:

Region:

Set Up Logging > Create Cancel

Source: Amazon S3 User's Guide

- ▶ List all keys in bucket (may not be 100% up to date)
- ▶ Modify bucket permissions

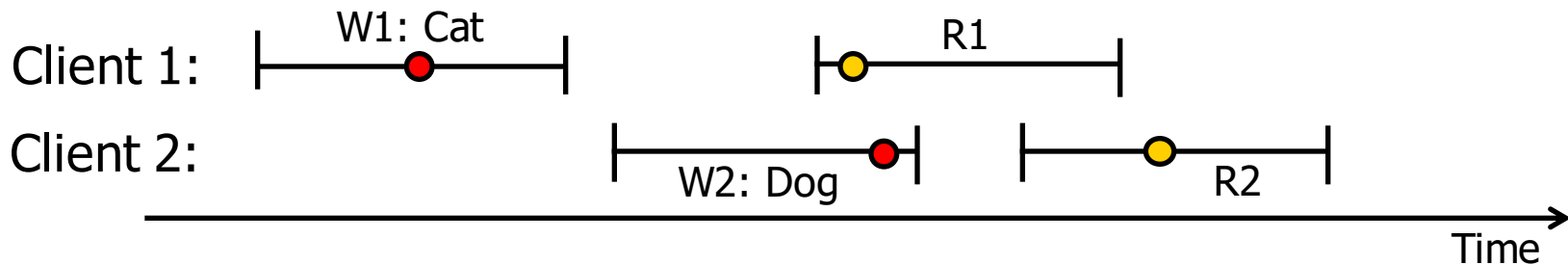


S3: Object operations

- ▶ PUT object in bucket
 - ▶ GET object from bucket
 - ▶ DELETE object from bucket
 - ▶ Modify object permissions
-
- ▶ The key issue: How do we manage concurrent updates?
 - ▶ Will I see objects you delete? the latest version? etc.

▶ S3: Consistency models

- ▶ Consistency model depends on the region
 - ▶ US West, EU, Asia Pacific, S. America: **read-after-write** consistency for PUTs of new objects and **eventual consistency** for overwrite PUTs and DELETES
 - ▶ S3 buckets in the US Standard Region: **eventual consistency**



- ▶ **Read-after-write consistency:**
 - ▶ Each read or write operation becomes effective at some point between its start time and its completion time
 - ▶ Reads return the value of the last effective write



S3: Versioning

- ▶ S3 handles consistency through **versioning** rather than locking
 - ▶ The idea: every bucket + key maps to a list of versions
 - ▶ [bucket+key] → [object v1] [object v2] [object v3] ...
 - ▶ Each time we PUT an object, it gets a new version
 - ▶ The last-received PUT overwrites any previous ones!
 - ▶ When we GET:
 - ▶ An unversioned request likely receives the last version – but this is not guaranteed depending on propagation delays
 - ▶ A request for bucket + key + version uniquely maps to a single object!
- ▶ Versioning can be enabled for each bucket
 - ▶ Why would you (not) want versioning?



Recap: Amazon S3

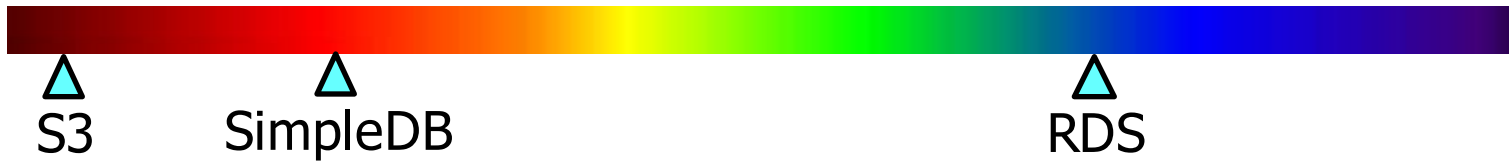
- ▶ A key-value store for large objects
 - ▶ Buckets, keys, objects, folders
 - ▶ Various ways to access objects, e.g., HTTP and BitTorrent
- ▶ Provides eventual consistency
 - ▶ +/- a few details that depend on the region
- ▶ Supports versioning and access control
 - ▶ Access control is based on ACLs

▶ Plan for today

- ▶ Key-value stores (KVS) ✓
 - ▶ Basic concept; operations ✓
 - ▶ Examples of KVS ✓
 - ▶ KVS and concurrency ✓
 - ▶ Key-multi-value stores; cursors ✓
- ▶ Key-value stores in the Cloud ✓
 - ▶ Challenges ✓
 - ▶ Specialized KVS ✓
- ▶ Two implementations ✓
 - ▶ Amazon S3 ✓
 - ▶ Amazon SimpleDB 

▶ What is Amazon SimpleDB?

- ▶ A highly scalable, non-relational data store
 - ▶ Despite its name, not really a database
 - ▶ Stronger consistency guarantees than S3
 - ▶ Highly scalable; built-in replication; automatic indexing
 - ▶ No 'real' transactions, just a conditional put/delete
 - ▶ No 'real' relations, just a fairly basic select





SimpleDB: Data model

Name (key)

Attributes (key-multivalued)

Items

Customer ID	First name	Last name	Street address	City	State	Zip
123	Bob	Smith	123 Main St	Springfield	MO	65801
456	James	Johnson	456 Front St	Seattle	WA	98104

- ▶ Somewhat analogous to a spreadsheet:
 - ▶ Domains: Entire 'tables'; like buckets
 - ▶ Items: Names with attribute-multivalued sets
 - ▶ For example, an item could have more than one street address
- ▶ It is possible to add attributes later
 - ▶ No pre-defined schema

▶ SimpleDB: Basic operations

- ▶ ListDomains
- ▶ CreateDomain, DeleteDomain
- ▶ DomainMetadata
- ▶ PutAttributes
 - ▶ Also atomic BatchPutAttributes – all must succeed
- ▶ DeleteAttributes
- ▶ GetAttributes
- ▶ Select (like an SQL query)



SimpleDB: PUT and GET

- ▶ **PutAttributes** has a very simple model:
 - ▶ Specify the domain and the item name
 - ▶ [key] → [list of name/value pairs], where we list Attribute.1.Name, Attribute.1.Value, etc.
 - ▶ Each Attribute.X has an optional Replace flag (Replace = 0 means add another value)

- ▶ **GetAttributes**
 - ▶ Specify the domain and the item name + optionally attribute
 - ▶ Can choose whether the read should be consistent or not
 - ▶ What are the advantages of each choice?



SimpleDB: Conditional Put

- ▶ SimpleDB also supports a **conditional put**
 - ▶ Item is updated only if the existing value of an attribute matches the value you specify; otherwise update is rejected
- ▶ Can we use this to guarantee consistency?
 - ▶ Idea: implement a version number, e.g., like this:

```
do {  
    List<Attributes> attrs = kvs.getAttributesFor(key);  
    ... update the attribute values as we like ...  
    retCode = kvs.conditionalPut(key, attrs,  
        ("version", attrs.get("version")));  
} while (retcode == ErrorCode.ConditionalCheckFailed);
```



SimpleDB: Select

- ▶ A very simple “query” interface based on SQL syntax
 - ▶ `SELECT output_list FROM domain_name WHERE expression [sort expression] [limit spec]`
 - ▶ Example: "select * from books where author like 'Tan%' and price <= 55.90 and year is not null order by title desc limit 50"
 - ▶ Can choose whether or not read should be consistent
 - ▶ Supports a cursor



Alternatives to SimpleDB

- ▶ There is a similar service to SimpleDB underneath most major “cloud” companies’ infrastructure
 - ▶ Google calls theirs BigTable
 - ▶ Yahoo’s is called PNUTS
 - ▶ See reading list at the end
- ▶ All consist of items with a variable set of attribute-value pairs
 - ▶ More flexible than a relational DBMS table
 - ▶ But don’t support full-fledged transactions



Recap: Amazon SimpleDB

- ▶ A scalable, non-relational data store
 - ▶ Domains, items, keys, values
 - ▶ Stronger consistency than S3
 - ▶ No pre-defined schema

▶ Plan for today

- ▶ Key-value stores (KVS) ✓
 - ▶ Basic concept; operations ✓
 - ▶ Examples of KVS ✓
 - ▶ KVS and concurrency ✓
 - ▶ Key-multi-value stores; cursors ✓
- ▶ Key-value stores in the Cloud ✓
 - ▶ Challenges ✓
 - ▶ Specialized KVS ✓
- ▶ Two implementations ✓
 - ▶ Amazon S3 ✓
 - ▶ Amazon SimpleDB ✓

▶ Where could we go beyond this?

- ▶ KVSs present one of the simplest data representations: key + one or more objects/properties
- ▶ Some alternatives:
 - ▶ Relational databases represent data as interlinked tables (in essence, a limited form of a graph)
 - ▶ Hierarchical storage systems represent data as nested entities
 - ▶ More general graph storage might represent entire graph structures with links
- ▶ All are implementable over a KVS
 - ▶ But all allow **higher level requests** (e.g., paths), and might optimize for this
 - ▶ Example: I know that the customer always asks for images related to patients' records, so maybe we should put the two in the same place

▶ Summary: Cloud Key/Value Stores

- ▶ Attempt to provide very high **durability**, **availability** in a persistent, geographically distributed storage system
- ▶ Need to choose **compromises** due to limitations of communications, hardware, software
 - ▶ Large, seldom-changing objects – **eventual consistency** and versioned model in S3
 - ▶ Small, more frequently changing objects – lower-latency response, **conditional updates** in SimpleDB
- ▶ Both are useful in different situations



Further reading

- ▶ A. Rowstron and P. Druschel: "Storage management and caching in **PAST**, a large-scale, persistent peer-to-peer storage utility" (SOSP'01)
 - ▶ <http://www.research.microsoft.com/~antr/PAST/past-sosp.pdf>
 - ▶ F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber: "**Bigtable**: A Distributed Storage System for Structured Data" (OSDI'06)
 - ▶ labs.google.com/papers/bigtable-osdi06.pdf
 - ▶ G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels: "**Dynamo**: Amazon's Highly Available Key-Value Store" (SOSP'07)
 - ▶ <http://dl.acm.org/citation.cfm?id=1294281>
 - ▶ B. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H. Jacobsen, N. Puz, D. Weaver, and R. Yerneni: "**PNUTS**: Yahoo!'s Hosted Data Serving Platform" (PVLDB'08)
 - ▶ <http://infolab.stanford.edu/~usriv/papers/pnuts.pdf>
 - ▶ H. Lim, B. Fan, D. Andersen, and M. Kaminsky: "**SILT**: A Memory-Efficient, High-Performance Key-Value Store" (SOSP'11)
 - ▶ <http://www.cs.cmu.edu/~dga/papers/silt-sosp2011.pdf>
-



Slides adapted (under permission) from
Andreas Haeberlen
(NETS 212: Scalable and Cloud Computing)
CIS Department, Penn-State University

Class projects

Big Data in ???

▶ Big Data in ???

- ▶ Pick a subject in groups of 2 (online at the class website starting from tomorrow):
 - ▶ Astronomy
 - ▶ Energy
 - ▶ Bioinformatics
 - ▶ Cities
 - ▶ Healthcare
- ▶ How to pick:
 - ▶ Send an e-mail to: Dan.Vodislav@u-cergy.fr
 - ▶ FIFO
 - ▶ Send 3 choices ordered
 - ▶ By Wednesday 04/02 noon

What is expected?

- ▶ A report of around 5 to 6 pages
- ▶ A presentation of 15 min + time for questions
- ▶ Report and Presentation Language can be English or French

- ▶ Make sure that you understood the problem correctly
- ▶ Don't focus on technical details; add references if needed
- ▶ Try to collect/add additional information



Report structure

Big Data and Healthcare

- ▶ Introduction
 - ▶ Set the context
 - ▶ Describe the problem
 - ▶ Identify sources of big data
- ▶ Big Data in Healthcare
 - ▶ Types of data
 - ▶ Properties of data (size, velocity, variety, etc.)
 - ▶ Open Data
 - ▶ Linked Data (models, ontologies, etc.)



Report structure

- ▶ **Methods Used to Process Big Data**
 - ▶ Methods to collect data
 - ▶ Mining methods
 - ▶ Processing methods
 - ▶ Analytics
 - ▶ Privacy/Trust concerns
- ▶ **Infrastructure to process Big Data**
 - ▶ Centralized / distributed
 - ▶ Cloud computing
 - ▶ Public / private processing
- ▶ **Conclusions**