CSx55: Distributed Systems [HDFS]

Circumventing The Perils of Doing Too Much

To sidestep the curse of overreach
 the namenode needs gumption and guardrails alike
What's not an option?
 Playing it by ear

With data volumes on an upward trajectory
 avoid the bottleneck strain
A way out? This ain't much of a mystery
 separate data from the control plane

Shrideep Pallickara
Computer Science
Colorado State University

Frequently asked questions from the previous class survey

- How are splits/blocks/chunks assigned to different machines in a way that is commensurate with their capabilities?
- Where is the combiner?
  - Right next to the mapper!

Topics covered in today's lecture

- Hadoop MapReduce (wrap-up)
- HDFS

Matrix Cup
Final Standings

Position	Student	Time (seconds)
1	Brenner Lattin	1.66
2	Tyler Malone	2.04
3	Adam Nasla	4.09
4	Kushal Reddy Alimineti	13.03
5	Job Roloff	13.45
6	Trevor Chartier	18.54
7	Henry Gates	19.46
8	Ayden Garza	20.76
9	Tommy McRoskey	21.62
10	Zacharie Guida	22.46

Hadoop MapReduce [wrap-up]

In Hadoop a Reduce task has 4 phases

- Shuffle
- Sort
- Reducer
- Output format

Reduce task phases: **Shuffle and sort**

- Shuffle
  - Takes the output files written by all of the partitioners and downloads them to the local machine in which the reducer is running
- Sort
  - Individual data pieces are then **sorted by key** into one larger data list
  - **Groups equivalent keys together** so that their values can be iterated over easily in the reduce task
- Shuffle brings it all home; sort lines it up for the reducer

Reduce task phases: **Shuffle and sort**

- This phase is **not customizable** and the framework handles everything automatically
- The only control a developer has is how the keys are sorted and grouped by specifying a custom `Comparator` object
  - A `Comparator` in Java defines how two objects are compared
    - It is the rulebook that tells the sorting algorithm what "*comes before*" what

Reduce task phases: **Reducer**

- Takes the grouped data as input and runs a reduce function **once per key grouping**
- The function is passed the key and an iterator/iterable over all of the values associated with that key
  - A wide range of processing can happen in this function: data can be aggregated, filtered, and combined etc.
- Once the reduce function is done, it sends zero or more key/value pairs to the final step, the output format
- N.B.: map & reduce functions will change from job to job

Reduce task phases: **Output format**

- Translates the final key/value pair from the reduce function and writes it out to a file using a record writer
- By default:
  - Separate the key and value with a tab
  - Separates records with a newline character
- Can typically be customized to provide richer output formats
  - But in the end, the data is written out to HDFS, regardless of format

Combiner Functions

Combiner functions

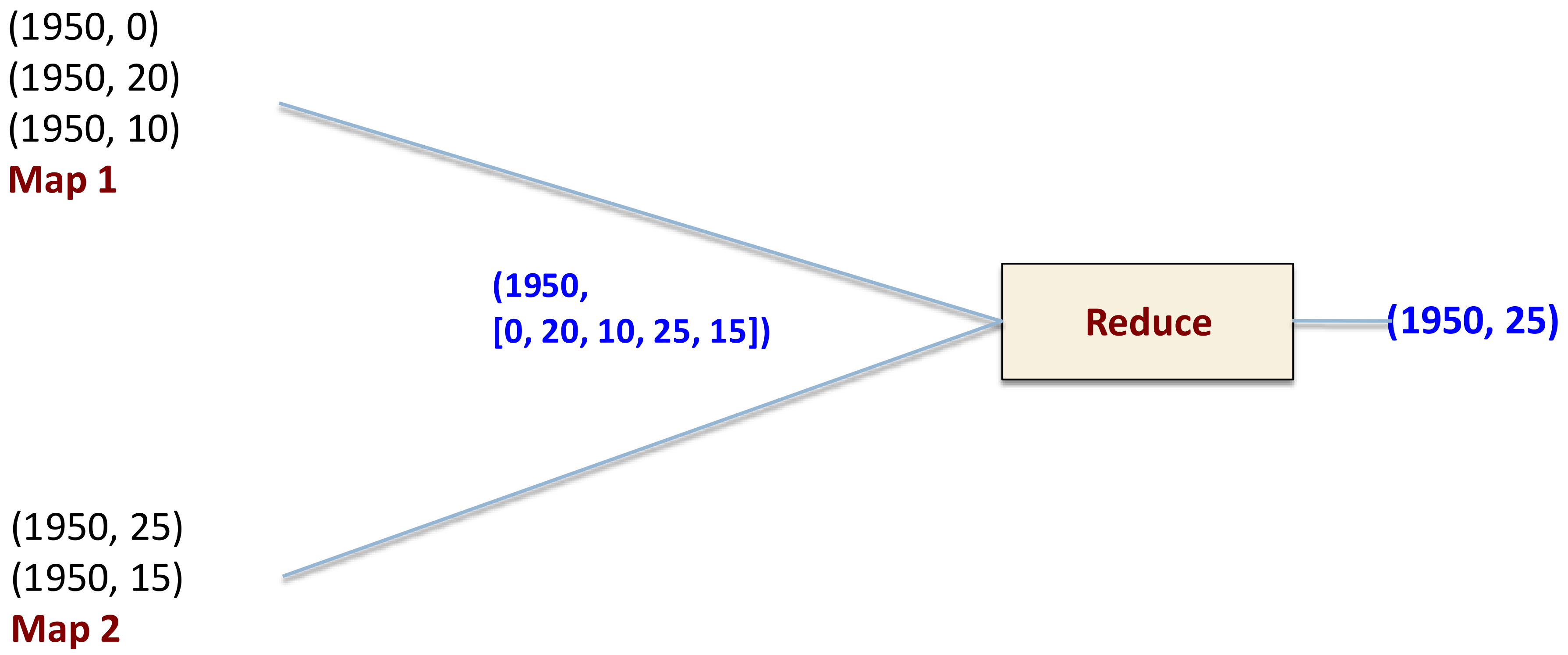
- Many MapReduce jobs are limited by the available network bandwidth
  - Framework has mechanisms to *minimize the data transferred* between map and reduce tasks

- A **combiner** function is run <u>on the map output</u>
  - Combiner output fed to the reduce task

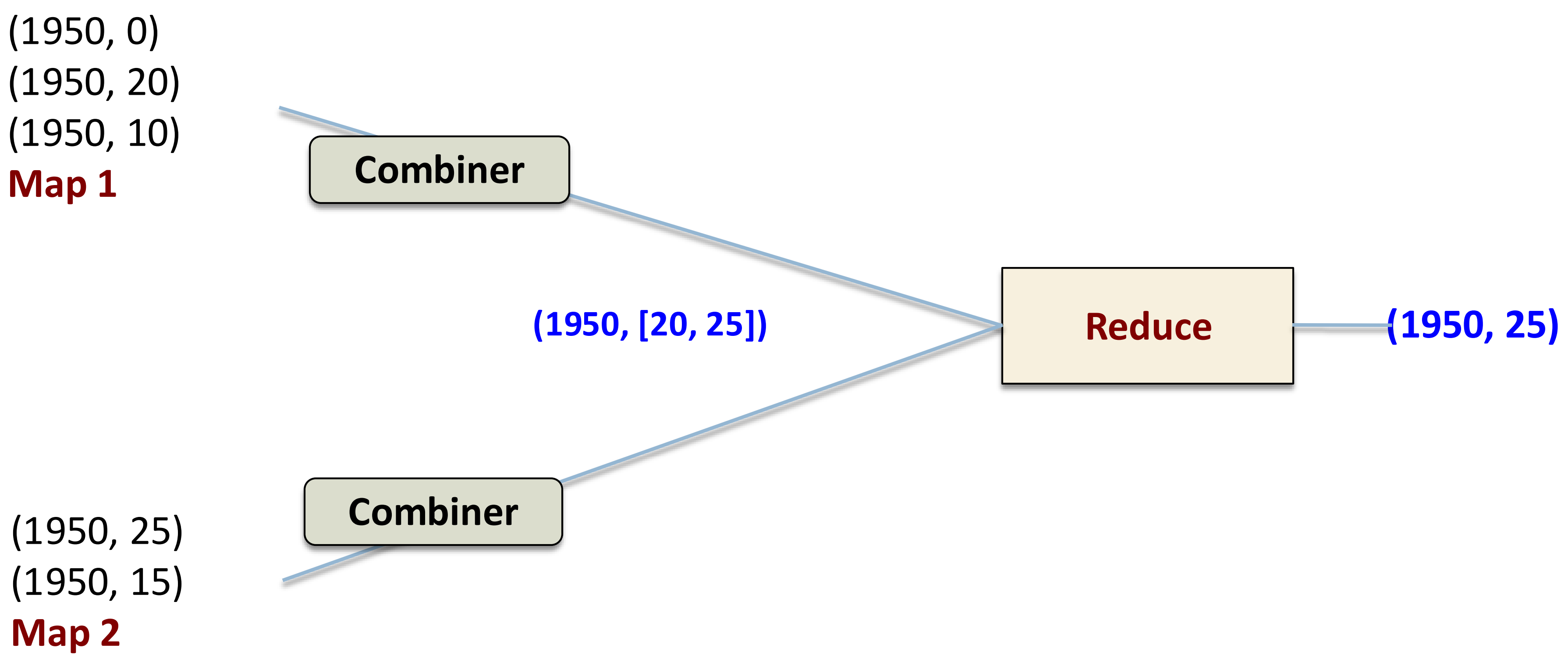
Combiner function

- No guarantees on *how many times* Hadoop will call this on a map output record
  - The combiner should, however, result in the same output from the reducer
- **Contract** for the combiner **constrains the type of function** that can be used

Combiner function: Let's look at the maximum temperature example [1/2]



Combiner function: Let's look at the maximum temperature example [2/2]



A closer look at the function calls

- max(0, 20, 10, 25, 15) =

  max (max(0, 20, 10), max(25, 15)) =

  max (20, 25) = 25

- Functions with this property are called **commutative** *and* **associative**
  - Commutative: Order of operands (5+2) = (2 + 5)
    - Division and subtraction are not commutative
  - Associative: Order of operators 5 x (5x3) = (5x5)x3
    - Vector cross products are not; also note that division/subtraction aren't for e.g.,

      $$(a \div b) \div c \neq a \div (b \div c)$$

Not all functions posses the commutative and associative properties

- What if we were computing the mean temperatures?
- We can cannot use mean as our combiner function

mean(0, 20, 10, 25, 15) = 14

BUT

mean(mean(0, 20, 10), mean(25, 15)) =
mean(10, 20) = 15

Combiner: Summary

- The combiner **does not replace** the reduce function
  - Reduce *is still needed* to process records from different maps

- But it is useful for **cutting down traffic** from maps to the reducer

Specifying a combiner function

```
public class MaxTemperatureWithCombiner {

  public static main(String[] args) throws Exception {
    Job job = Job.getInstance();
    job.setJarByClass(MaxTemperature.class);
    job.setJobName("Max temperature");

    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));

    job.setMapperClass(MaxTemperatureMapper.class);
    job.setCombinerClass(MaxTemperatureReducer.class);
    job.setReducerClass(MaxTemperatureReducer.class);

    job.setOutputKey(Text.class);
    job.setOutputValueClass(IntWritable.class);

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

  }
}
```

Another example (Combiner)

Another example with StackOverflow [1/2]

- Given a list of user's comment determine the average comment length per-hour

- To calculate the average, we need two things:
  - Sum values that we want to average
  - Number of values that went into the sum

Another example with StackOverflow [2/2]

- Reducer can do this very easily by iterating through each value in the set and adding to a running sum while keeping count
- But if you do this you cannot use the reducer as your combiner!
  - Calculating an average is not an associative operation
    - You cannot change the order of the operators
    - mean(0, 20, 10, 25, 15) = 14 BUT ..
    - mean(mean(0, 20, 10), mean(25, 15)) = mean(10, 20) = 15

Approach to ensuring code reuse at the combiner

- Mapper will output two columns of data
  - Count and average
- Reducer will multiply "count" field by the "average" field to add to a running count *and* add "count" to the running count
  - Then divide the running sum with running count
    - Output the count with the calculated average

Mapper code

```
public static class AverageMapper extends
    Mapper < Object, Text, IntWritable, CountAverageTuple > {

   private CountAverageTuple outCountAverage = new CountAverageTuple();
   public void map( Object key, Text value, Context context)
      throws IOException, InterruptedException {
     Map < String, String > parsed =
            MRDPUtils.transformXmlToMap( value.toString());
     String strDate = parsed.get(" CreationDate");
     String text = parsed.get(" Text");
     // get the hour this comment was posted in
     Date creationDate = frmt.parse( strDate);
    outHour.set( creationDate.getHours());

    outCountAverage.setCount( 1);
    outCountAverage.setAverage( text.length());

     // write out the hour with the comment length
     context.write( outHour, outCountAverage);
   }
}
```

Reducer code

```
public class AverageReducer extends Reducer < IntWritable,
CountAverageTuple, IntWritable, CountAverageTuple > {
    private CountAverageTuple result = new CountAverageTuple();

    public void
    reduce(IntWritable key, Iterable < CountAverageTuple > values,
        Context context) throws IOException, InterruptedException {
       float sum = 0; float count = 0;

       // Iterate through all input values for this key
       for (CountAverageTuple val : values) {
          sum +=  val.getCount() * val.getAverage();
          count += val.getCount();
       }
       result.setCount( count);
       result.setAverage( sum / count);
       context.write( key, result);
     }
}
```

Data flow for the average example

	Input key	Input Value	
	Hour	**Count**	**Average**
Group 1	4	1	10
	4	1	8
	4	1	21
Group 2	3	1	1
	3	1	19
	9	1	7
	9	1	12

<u>Setting</u>:
Combiner executes over Groups 1 and 2
DOES NOT execute on the last two rows

Combiner Output/ Reducer Input

Output key	Output Value	
Hour	**Count**	**Average**
3	2	10
4	3	13
9	1	7
9	1	12

Hadoop Distributed File System

Block

- Filesystems for a single disk deal with data in blocks
  - Integral number of the HDD block size

- Block sizes
  - Filesystem blocks are a few KB
  - Disk blocks are normally 512 bytes

HDFS Blocks

- Have a much larger size: **256 MB** [default was 64 till version 1.x]
- Files are **broken** into block-sized *chunks*
  - Each block is stored as an independent unit
- If the last chunk is less than the HDFS block size?
  - No space is wasted because the blocks are themselves stored as files

Why is the block-size so big?

- **Time to transfer** data from disk can be made significantly larger than the time to *seek* first block
- If the seek time is 10 ms and transfer rate is 100 MB/sec?
  - To make seek time 1% of the transfer time, block size should be 100 MB
- Must be careful not to overdo block size increase
  - Since tasks operate on blocks, the number of tasks could reduce

Benefits of the block abstraction in distributed file systems

- File can be **larger than any single disk** in the cluster
- Simplifies the storage subsystem
  - File metadata (including permissions) handled by another subsystem and not stored with the block

Blocks and replication

- Each block is replicated on a small number of **physically separate** machines

- If a block becomes unavailable?
  1. Copy *read from another location* transparently
  2. That block is also *replicated from its alternative locations* to other live machines
     - Bring replication factor back to the desired level

HDFS’ fsck command

- List blocks that make up each file in the filesystem

```
% hadoop fsck / -files -blocks
```

Nodes in the HDFS

- Namenode {master}
- Datanode {worker}

Namenode

- Manages filesystem **namespace**
- Maintains filesystem tree and metadata
  - For all files and directories in the tree
- Information stored persistently on local disk in two files
  - **Namespace image** and the **edit log**

Tracking location of blocks comprising files

- Namenode knows about datanodes on which all blocks of a file are located

- The locations of the blocks are not stored persistently
  - Information **reconstructed** from datanodes during start up

Interacting with HDFS

- HDFS presents a **POSIX-like** file system interface
- Client code does not need to know about the namenode and datanode to function

Datanodes

- Store and retrieve blocks
  - Initiated by the client or the namenode

- **Periodically reports** back to the namenode with the *list of blocks* that they store

Failure of the namenode

- Decimates the filesystem
- **All files on the filesystem are lost**
    - No way of knowing how to reconstitute the files from the blocks

Guarding against namenode failures

- **Backup** files comprising the persistent state of the **filesystem metadata**
  - Hadoop can be configured so that the namenode writes its persistent state to multiple filesystems
    - Writes are synchronous and atomic
- Run a **secondary** namenode
  - Does not act as a namenode
  - Periodically merges namespace image with edit log

Secondary namenode

- Runs on a separate physical machine
  - Requires as much memory as the namenode to perform the merge operation
- Keeps a copy of the merged namespace image
  - Can be used if the namenode fails
- However, the secondary namenode **lags** the primary
  - Data loss is almost certain

Though my soul may set in darkness, it will rise in perfect light;
I have loved the stars too fondly to be fearful of the night.

Poem: *The Old Astronomer*; Sarah Williams (1837-1868)

Scaling the NameNodes

Too many files, not enough memory: Enter federation

- On large clusters with many files, memory is a limiting factor for scaling
- HDFS federation allows scaling with the addition of namenodes
  - Each manages a portion of the filesystem namespace
    - For e.g., one namenode for `/user` and another for `/share`

HDFS Federation [1/2]

- Each namenode manages a namespace volume
  - Metadata for the namespace and block pool
- Namespace volumes are **independent** of each other
  - No communications between namenodes
  - Failure of one namenode does not affect availability of another

HDFS Federation [2/2]

- Block pool storage is **not partitioned**

- Datanodes register with each namenode in the cluster
  - Store blocks from multiple blockpools

Recovering from a failed namenode [1/2]

- Admin starts a new primary namenode
  - With one of the filesystem metadata replicas
  - Configure datanodes and clients to use this namenode

- New namenode unable to serve requests until:
  1. Namespace image is **loaded** into memory
  2. **Replay** of edit log is complete
  3. Received enough **block reports** from datanodes to leave safe mode

Recovering from a failed namenode [2/2]

- Recovery can be really long
  - On large clusters with many files and blocks this can be about 30 minutes
- This also impacts routine maintenance

HDFS High Availability has features to cope with this

- Pair of namenodes in **active standby** configuration
- During failure of active namenode, standby takes over the servicing of client requests
  - In 10s of seconds

HDFS High-Availability: Additional items to get things to work

- Namenodes use a highly-available **shared storage** to store the *edit log*
- Datanodes must send block reports to **both** namenodes
  - Block mappings stored in memory not disk
- Clients must be configured to handle namenode failover

HDFS HA: Dealing with ungraceful failovers

- Slow network or a network partition can trigger failover transition
  - Previously active namenode thinks it is ***still*** the active namenode
- The HDFS HA tries to avoid this situation using **fencing**
  - Previously active namenode should be prevented from causing corruptions

Fencing mechanisms: To shutdown previously active namenode

- Kill the namenode's process
- Revoking access to the shared storage directory
- Disabling namenode's network port
  - Using the remote management command
- STONITH
  - Use specialized power distribution unit to forcibly power down the host machine

Basic Filesystem Operations

- Type **hadoop fs –help** to get detailed help on commands
  - We are invoking Hadoop's filesystem shell command **fs** which supports other subcommands
- Start copying a file from the local filesystem to HDFS

```
% hadoop fs –copyFromLocal input/docs/quangle.txt
        /user/tom/quangle.txt
```

Basic Filesystem Operations

- Copy file back to the local filesystem

```
%hadoop fs –copyToLocal /user/tom/quangle.txt
    input/docs/quangle.copy.txt
```

- Verify if the movement of the files have changed the files in any way

```
% openssl md5 quangle.txt quangle.copy.txt
```

Basic Filesystem Operations

```
% hadoop fs -mkdir books
% hadoop fs -ls .
Found 2 items
drwxr-xr-x - tom supergroup 0 2019-04-02 22:41 /user/tom/books
-rw-r--r-- 1 tom supergroup 118 2019-04-02 22:29 /user/tom/quangle.txt
```

- Directories are treated as metadata and **stored by the namenode** not the datanodes

Hadoop File Systems

Hadoop filesystems

- Hadoop has an abstract notion of filesystem
- HDFS is just one implementation
  - Others include HAR, KFS (Cloud Store), S3 (native and block-based)
- Uses URI scheme to pick correct filesystem instance to communicate with

  % hadoop fs –ls file:// to communicate with local file system

Interacting with the filesystem

- Hadoop has a `FileSystem` class
- HDFS implementation is accessible through the `DistributedFileSystem`
  - Write your code against the `FileSystem` class for maximum portability

Reading data from a Hadoop URL

```
InputStream in = null;
try {
    in = new URL("hdfs://host/path").openStream();
    // process in
} finally {
    IOUtils.closeStream(in);
}
```

Make Java recognize Hadoop's URL scheme

- Call `setURLStreamHandlerFactory()` on URL with an instance of `FsURLStreamHandlerFactory`
- Can only be called once per JVM, so it is typically executed in a static block

Displaying files from a Hadoop filesystem

```
public class URLCat {
  static {
       URL.setURLStreamHandlerFactory(
                      new FsUrlStreamHandlerFactory());
  }

 public static void main(String[] args) throws Exception {
    InputStream in = null;
    try {
       in = new URL(args[0]).openStream();
       IOUtils.copyBytes(in, System.out, 4096, false);
    } finally {
      IOUtils.closeStream(in);
    }
 }
}
```

Buffer size used for copying

Close streams after copying is complete?

A sample run of the `URLCat`

```
% hadoop URLCat hdfs://localhost/user/tom/quangle.txt

On the top of the Crumpetty Tree
The Quangle Wangle sat,
But his face you could not see,
On account of his Beaver Hat.
```

The contents of this slide set are based on the following references

- *Tom White. Hadoop: The Definitive Guide. 3rd Edition. Early Access Release. O'Reilly Press. ISBN: 978-1-449-31152-0.* Chapters [2 and 3].