NYC DATA SCIENCE ACADEMY

Hadoop & MapReduce

NYC Data Science Academy

Outline

1. **Big Data Overview**
2. **Hadoop Fundamentals**
   - 2.1. The Hadoop Distributed Filesystem (HDFS)
   - 2.2. YARN
   - 2.3. MapReduce
3. **Hands-On Exercise: Basic HDFS Operations**
4. **Hands-On Exercise: Developing MapReduce Jobs**
   - 4.1. Example: WordCount
   - 4.2. Combiner

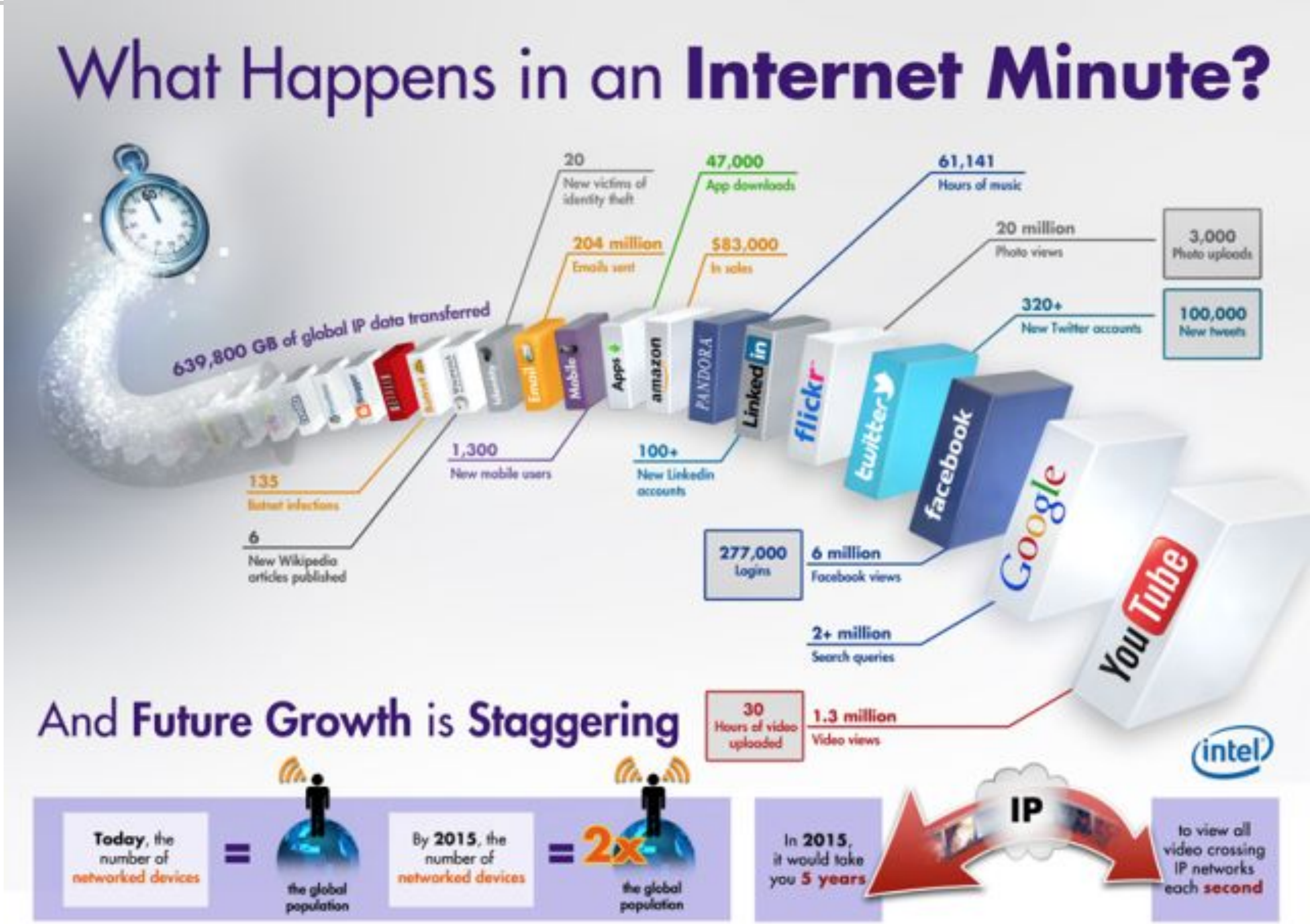
1. Big Data Overview

- Traditional Applications:
  - Relatively small amounts of data
  - Significant amount of complex computation performed on that data
- When a workload reaches capacity limits:
  - Faster processor
  - more RAM,
  - larger hard drive, etc.

What Is Big Data?

- Volume:
  - The amount of data generated
- Velocity
  - The frequency at which the data is generated or captured
- Variety
  - The actual content of the dataset (structured / unstructured data, images, video, etc)

The Data Explosion



[image link]

Big Data Use Cases

- Retail (Amazon, eBay, Walmart)
    - Production recommendations, supply chain optimization, marketing and promotions
- Social Media (Facebook, Instagram, LinkedIn)
    - User data mining, brand perception
- Advertising
    - Real-time ads bidding, ads targeting
- Travel (Uber, Lyft, Google maps)
    - Dynamic pricing, routes & schedules optimization

Scale Up v.s. Scale Out

- Scale Up (shared memory processing): Use large systems with enough resources to analyze the data
  - Easy to manage
  - Expensive
- Scale Out (distributed processing): Distribute the load to many separate computers, with each analyze a portion of the data
  - Larger data sets
  - Faster computation

What Is Hadoop?

- Hadoop: A scalable data storage and batch processing framework
  - Designed for storing very large files (petabytes)
  - runs on clusters of commodity hardware
  - Provides fault tolerance through software
  - Ingests, processes aggregates data
  - Easy to scale-out / scale-in

The History of Hadoop

- Google published the idea of Google Filesystem (GFS) and MapReduce in 2003-2004.
- Doug Cutting was working on an open source web crawler called Nutch, and was dealing with the same big data problems as Google. He realized that Google's idea was the best idea available at that time.
- Yahoo hired Doug Cutting and he began working on an open-source version of GFS and MapReduce.
- Originally part of the Nutch project, Hadoop eventually split off as a top-level Apache project.
- Since then, numerous components have been added to Hadoop, many as other Apache projects.

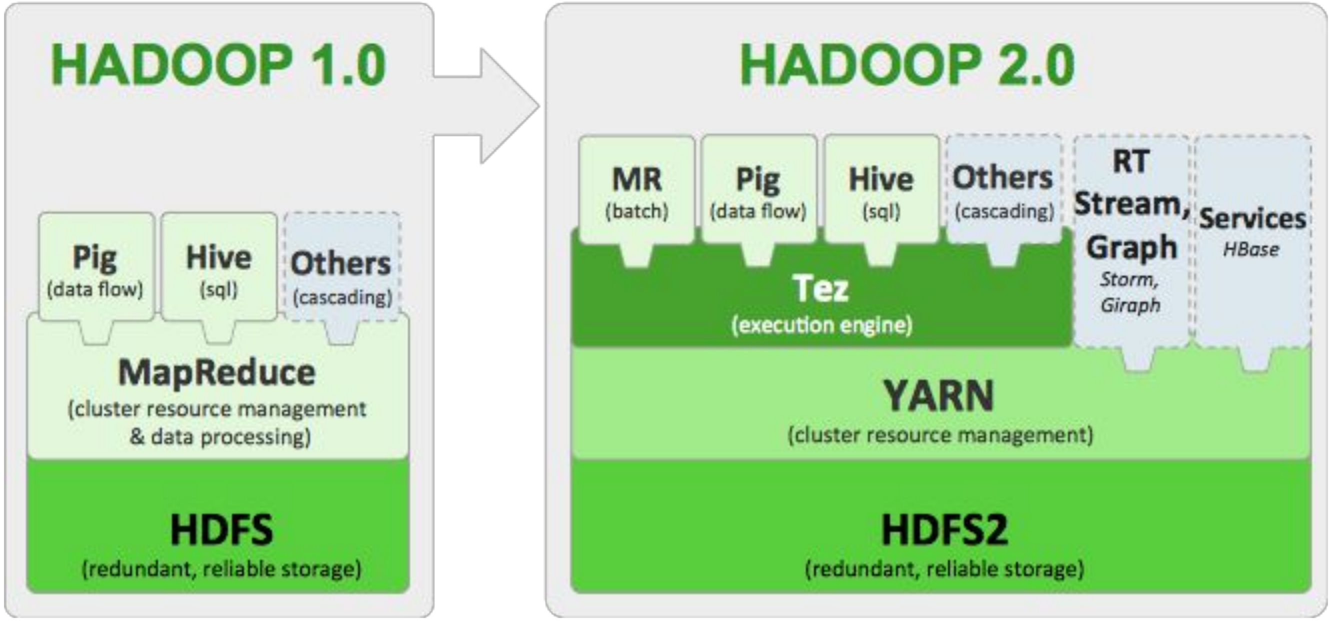
The History of Hadoop

- Become the standard open source software for cloud-computing.
- More than ten active top-level sub-projects at Apache, and numerous other projects.
- Able to run on thousands of nodes, breaking the world record of data size and time of sorting.

2. Hadoop Fundamentals

- Hadoop 2.x Core Components:
  - Hadoop Common - pre-defined set of utilities and libraries that can be used by other modules within the Hadoop ecosystem
  - Hadoop Distributed File System (HDFS) - data storage layer for Apache Hadoop
  - Hadoop Yarn - cluster resource management
  - MapReduce - a java-based distributed data processing framework

Hadoop Architecture



[image link]

Outline

1. **Big Data Overview**
2. **Hadoop Fundamentals**
    - **2.1. The Hadoop Distributed Filesystem (HDFS)**
    - 2.2. YARN
    - 2.3. MapReduce
3. **Hands-On Exercise: Basic HDFS Operations**
4. **Hands-On Exercise: Developing MapReduce Jobs**
    - 4.1. Example: WordCount
    - 4.2. Combiner

The Hadoop Distributed FileSystem (HDFS)

- Hadoop Distributed File System (HDFS): Java-based file system for scalable and reliable data storage, designed to span large clusters of commodity servers.

The Goals of HDFS

- HDFS is highly fault-tolerant: Hardware failure is the rule rather than the exception. An HDFS instance may consist of hundreds or thousands of server machines, each storing part of the file system's data. Therefore, detection of faults and quick, automatic recovery from them is a core architectural goal of HDFS.
- *Moving Computation is Cheaper than Moving Data*: A computation requested by an application is much more efficient if it is executed near the data it operates on. HDFS provides interfaces for applications to move themselves closer to where the data is located.
- *Portability Across Heterogeneous Hardware and Software Platforms*: HDFS has been designed to be easily portable from one platform to another.

Properties about HDFS

- *Data is accessed sequentially*: HDFS is tuned for high-speed throughput of entire files, *not* random access.
- *Simple Coherency Model*: HDFS applications need a write-once-read-many access model for files. A file once created, written, and closed need not be changed. Appends are permitted.
- ***Partitioning for reliability and efficiency***: Files are broken into replicated blocks of 128MB, for reliability and parallel processing. *This is always done, regardless of how files get into HDFS.*

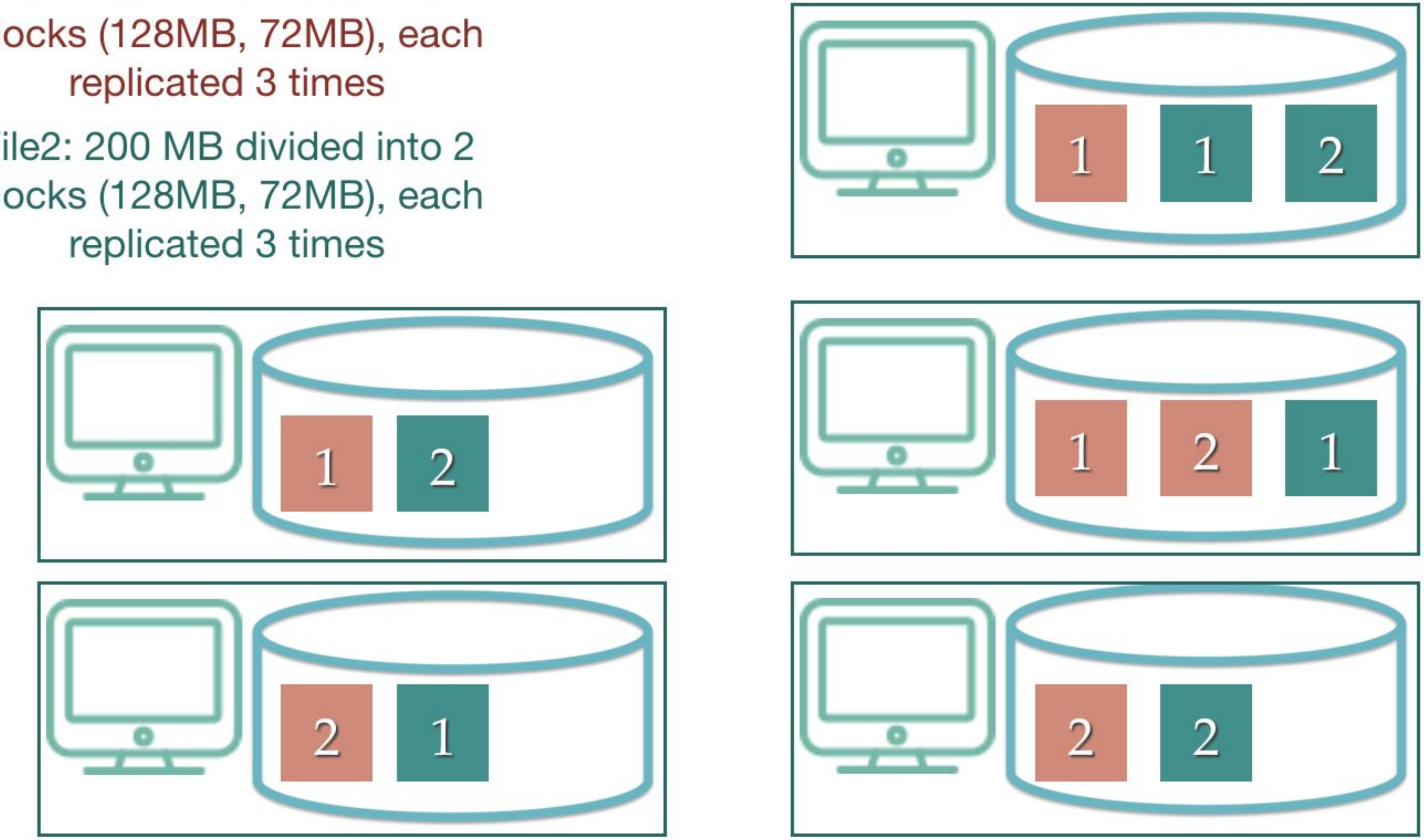
Properties about HDFS

- Why is a block in HDFS so large?
- If the block is large enough, the time it takes to transfer the data from the disk can be significantly longer than the time to seek the start of the block. Thus transferring a large file made of multiple blocks operates at the disk transfer rate.
- If the seek time is around 10 ms and the transfer rate is 100MB/s, to make the seek time 1% of the transfer time, we need to make the block size around 100MB.

Properties about HDFS

File1: 200 MB divided into 2 blocks (128MB, 72MB), each replicated 3 times

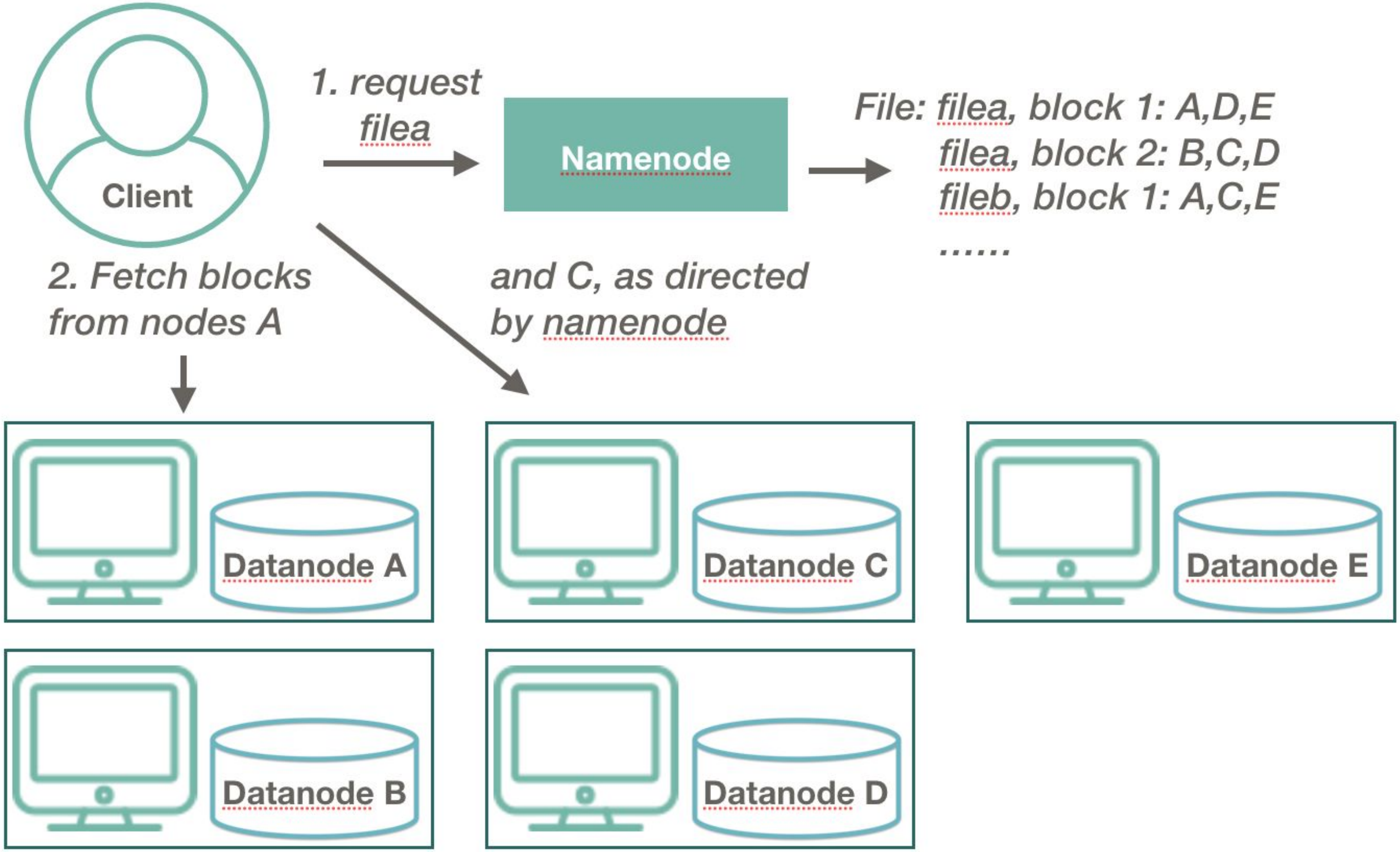
File2: 200 MB divided into 2 blocks (128MB, 72MB), each replicated 3 times



HDFS Internal Structure

- Master node called **namenode**.
- Worker nodes called **datanodes**.
- When files are copied to HDFS, they are divided into **blocks** stored on datanodes. Blocks are replicated (usually by 3) and distributed among datanodes.
- The namenode retains all metadata about files – which blocks are stored on which datanodes.

HDFS Internal Structure



Outline

1. **Big Data Overview**
2. **Hadoop Fundamentals**
   2.1. The Hadoop Distributed Filesystem (HDFS)
   2.2. YARN
   2.3. MapReduce
3. **Hands-On Exercise: Basic HDFS Operations**
4. **Hands-On Exercise: Developing MapReduce Jobs**
   4.1. Example: WordCount
   4.2. Combiner

YARN

- YARN stands for “Yet Another Resource Negotiator”, which is a resource management layer of Hadoop. The core components of YARN are the **ResourceManager** and the **ApplicationMaster**. The ResourceManager allocates resources to various applications, and the ApplicationMaster monitors the execution of the process.

YARN - ResourceManager

- The ResourceManager consists of two components: **Scheduler** and **ApplicationManager**
  - The Scheduler is responsible for allocating resources (CPU and memory) to the various running applications. Note that the scheduler performs no monitoring.
  - The ApplicationManager accepts job-submissions, negotiates the first container for executing ApplicationMaster and provides the service for restarting the ApplicationMaster container on failure.

YARN - ApplicationMaster

- The **per-application** ApplicationMaster has the responsibility of negotiating appropriate resource containers from the ResourceManager, tracking their status and monitoring for progress.

YARN - Benefit

- YARN is the main difference between Hadoop 1.x and 2.x. The main benefits are:
  - Hadoop 1.x supports only MapReduce. Hadoop 2.x supports multiple programming models with YARN.
  - Hadoop 2.x has overcome that limitations of scalability with the new architecture. Hadoop 1.x supports maximum 4,000 nodes per cluster where Hadoop 2.x supports more than 10,000 nodes per cluster.
  - Instead of having a fixed number of map and reduce slots, YARN's **NodeManager** has a number of dynamically created resource containers. Resource utilization is more efficient with YARN.

Outline

1. **Big Data Overview**
2. **Hadoop Fundamentals**
   - 2.1. The Hadoop Distributed Filesystem (HDFS)
   - 2.2. YARN
   - **2.3. MapReduce**
3. **Hands-On Exercise: Basic HDFS Operations**
4. **Hands-On Exercise: Developing MapReduce Jobs**
   - 4.1. Example: WordCount
   - 4.2. Combiner

MapReduce

- Calculations on large data sets often have this form: Start by aggregating the data (possibly in a different order from the "natural order"), then perform a summarizing calculation on the aggregated groups.
- The idea of MapReduce: If your calculation is explicitly structured like this, it can be *automatically* parallelized.

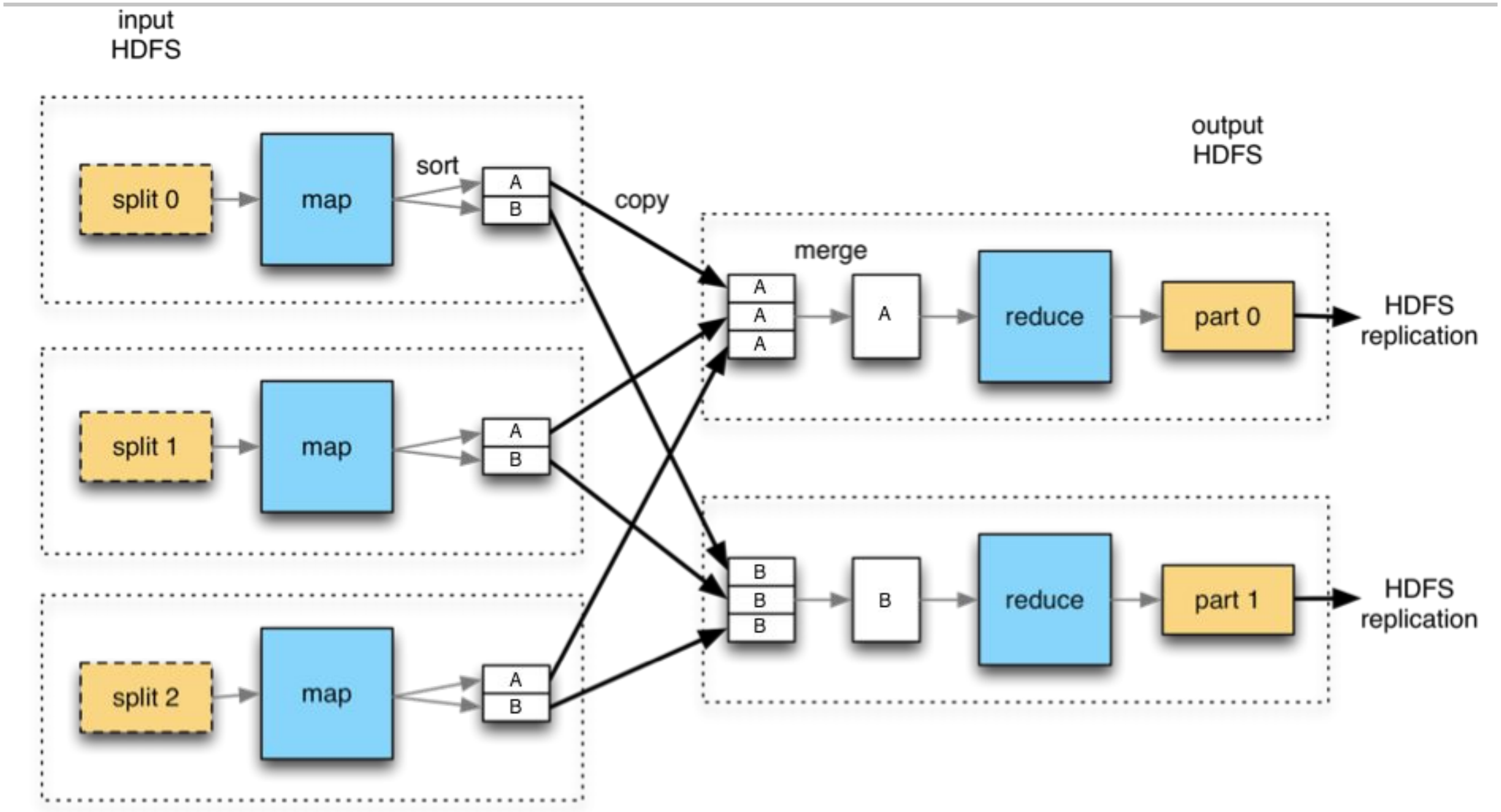
Computing with MapReduce

A MapReduce computation has three stages:

1. **Map**: A function called *map* is applied to each record in your input. It produces zero or more records as output, each with a key and value. Keys may be repeated.
2. **Shuffle**: The output from step 1 is sorted and combined: All records with the same key are combined into one.
3. **Reduce**: A function called *reduce* is applied to each record (key + values) from step 2 to produce the final output.

As the programmer, you only write map *and* reduce.

MapReduce DataFlow



Edited from: https://blog.cloudera.com/blog/2014/03/the-truth-about-mapreduce-performance-on-ssds/

Outline

1. **Big Data Overview**
2. **Hadoop Fundamentals**
   2.1. The Hadoop Distributed Filesystem (HDFS)
   2.2. YARN
   2.3. MapReduce
3. **Hands-On Exercise: Basic HDFS Operations**
4. **Hands-On Exercise: Developing MapReduce Jobs**
   4.1. Example: WordCount
   4.2. Combiner

3. Hands-On Exercise: Basic HDFS Operations

- HDFS can be accessed from applications in many different ways. The **command line** (CLI) is one of the simplest and the most familiar to many developers.
- In this exercise you will run Hadoop cluster in pseudo-distributed mode inside a docker container and manipulate files in HDFS via command line.
- The replication has been set to 1 because HDFS can’t replicate blocks to three datanodes when running with a single datanode.

Setting Up Hadoop Cluster

- A single node Hadoop Cluster with all the core components has been installed and configure.
- Follow this link to start the hadoop cluster. The docker image we will be using is `hadoop-core`:

```
docker run -it -p 50070:50070 syan83/hadoop-core
```

- When the Hadoop cluster is up, run the following command to find the hadoop version:

```
hadoop@master-node:~$ hadoop version
Hadoop 2.8.4
...
```

Basic Filesystem Operations

- The filesystem is now ready to be used, and we can do all the basic filesystem operations, such as reading files, creating directories, coping/deleting files, etc., via simple **hdfs** commands.
- The **hdfs** commands are very similar to linux file system shell commands, except that they need to be invoked by `hadoop fs`, or `hdfs dfs`, which is a synonym.
- To get detailed help, you can run:

```
hadoop@master-node:~$ hadoop fs -help
```

- For a complete list of available command and the usage, please go to:

  https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-common/FileSystemShell.html

Exercise: Exploring HDFS

`-ls` command list information about the files as in Unix:

- for a file ls returns stat on the file,
- for a directory it returns list of its direct children.

In the terminal window, enter the following commands and check the outputs:

```
1.  hadoop fs -ls
2.  hadoop fs -ls /
3.  hadoop fs -ls -R /
4.  hadoop fs -ls -R /user
5.  hdfs dfs -ls -R /user
```

Uploading And Downloading Files

HDFS and **Linux FS** are two separated file systems, which means you will not have direct access to the files on your namenode through HDFS, and vice versa.

The commands that allow you to copy files between Linux FS and HDFS are:

- Upload to hdfs: `hadoop fs -put <localsrc> <dst>`
- Download from hdfs: `hadoop fs -get <src> <localdst>`

Also you can use (restricted to local file reference):

- Upload to hdfs: `hadoop fs -copyFromLocal <localsrc> <dst>`
- Append to hdfs: `hadoop fs -appendToFile <localsrc> <dst>`
- Download from hdfs: `hadoop fs -copyToLocal <dst> <localdst>`

Viewing and Manipulating Files

HDFS is designed to be “write once read many”, which means editing/modify files is not permitted. The permitted operations are creation, appending, viewing, and deletion of files.

The usage of those commands are very similar to Unix commands:

- Display content via stdout: `hadoop fs -cat URI`
- Display last kb of file to stdout: `hadoop fs -tail <files>`
- Create directories within HDFS: `hadoop fs -mkdir <paths>`
- Copy files from `src` to `dst` in HDFS: `hadoop fs -cp <src> <dst>`
- Delete files in HDFS: `hadoop fs -rm <files>`

Exercise: Using HDFS

1. Download the hadoop wiki page and name it **hadoop.html**:

   ```
   wget -O hadoop.html https://wiki.apache.org/hadoop
   ```

2. Copy file to HDFS:

   ```
   hadoop fs -put hadoop.html
   ```

3. List file in HDFS:

   ```
   hadoop fs -ls ./
   ```

   ```
   hadoop fs -ls -R ../
   ```

4. Print the full content to stdout and display with `less`:

   ```
   hadoop fs -cat hadoop.html | less
   ```

Exercise: Using HDFS

5. Make a copy in hdfs and then check:

```
hadoop fs -cp hadoop.html hadoop-wiki.html
```

```
hadoop fs -ls ./
```

6. Now display the last few lines of the copied file:

```
hadoop fs -tail hadoop-wiki.html
```

7. Download the copy to local:

```
hadoop fs -get hadoop-wiki.html
```

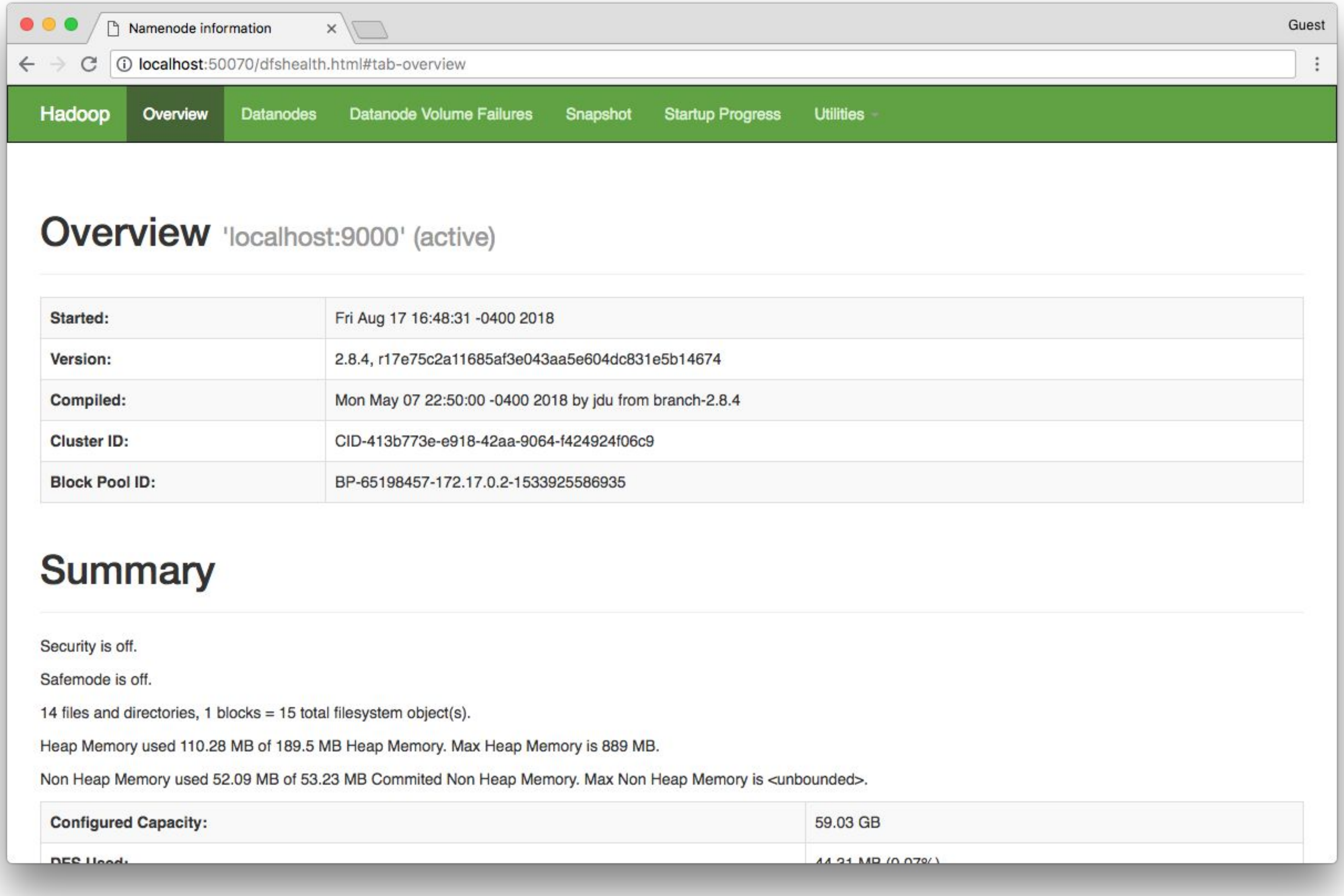
8. Remove the two files:

```
hadoop fs -rm hadoop*.html
```

HDFS Web UI

- **HDFS** comes with a web UI for viewing information about datanode/namenode and monitoring the health.
- By default the address of the web UI is:
  - NameNode WebUI: http://<namenode-ip>:50070/
  - DataNode WebUI: http://<datanode-ip>:50075/
- If you're running your hadoop cluster locally via docker, then you can open the NameNode UI at http://localhost:50070/

NameNode Web UI



Overview 'localhost:9000' (active)

Started:	Fri Aug 17 16:48:31 -0400 2018
Version:	2.8.4, r17e75c2a11685af3e043aa5e604dc831e5b14674
Compiled:	Mon May 07 22:50:00 -0400 2018 by jdu from branch-2.8.4
Cluster ID:	CID-413b773e-e918-42aa-9064-f424924f06c9
Block Pool ID:	BP-65198457-172.17.0.2-1533925586935

Summary

Security is off.

Safemode is off.

14 files and directories, 1 blocks = 15 total filesystem object(s).

Heap Memory used 110.28 MB of 189.5 MB Heap Memory. Max Heap Memory is 889 MB.

Non Heap Memory used 52.09 MB of 53.23 MB Commited Non Heap Memory. Max Non Heap Memory is <unbounded>.

Configured Capacity:	59.03 GB
DFS Used:	44.31 MB (0.07%)

Outline

1. **Big Data Overview**
2. **Hadoop Fundamentals**
    - 2.1. The Hadoop Distributed Filesystem (HDFS)
    - 2.2. YARN
    - 2.3. MapReduce
3. **Hands-On Exercise: Basic HDFS Operations**
4. **Hands-On Exercise: Developing MapReduce Jobs**
    - 4.1. Example: WordCount
    - 4.2. Combiner

4. Hands-On Exercise: Developing MapReduce Applications

- The MapReduce framework operates on <key, value> pairs, that is, the framework views the input to the job as a set of <key, value> pairs and produces a set of <key, value> pairs as the output of the job.
- MapReduce works by breaking the processing into two phases: the **map** phase and the **reduce** phase. Each phase has <key, value> pairs as input and output.

MapReduce Applications

The Hadoop framework is implemented in Java. You can develop MapReduce applications in Java or any JVM-based language, or use one of the following interfaces:

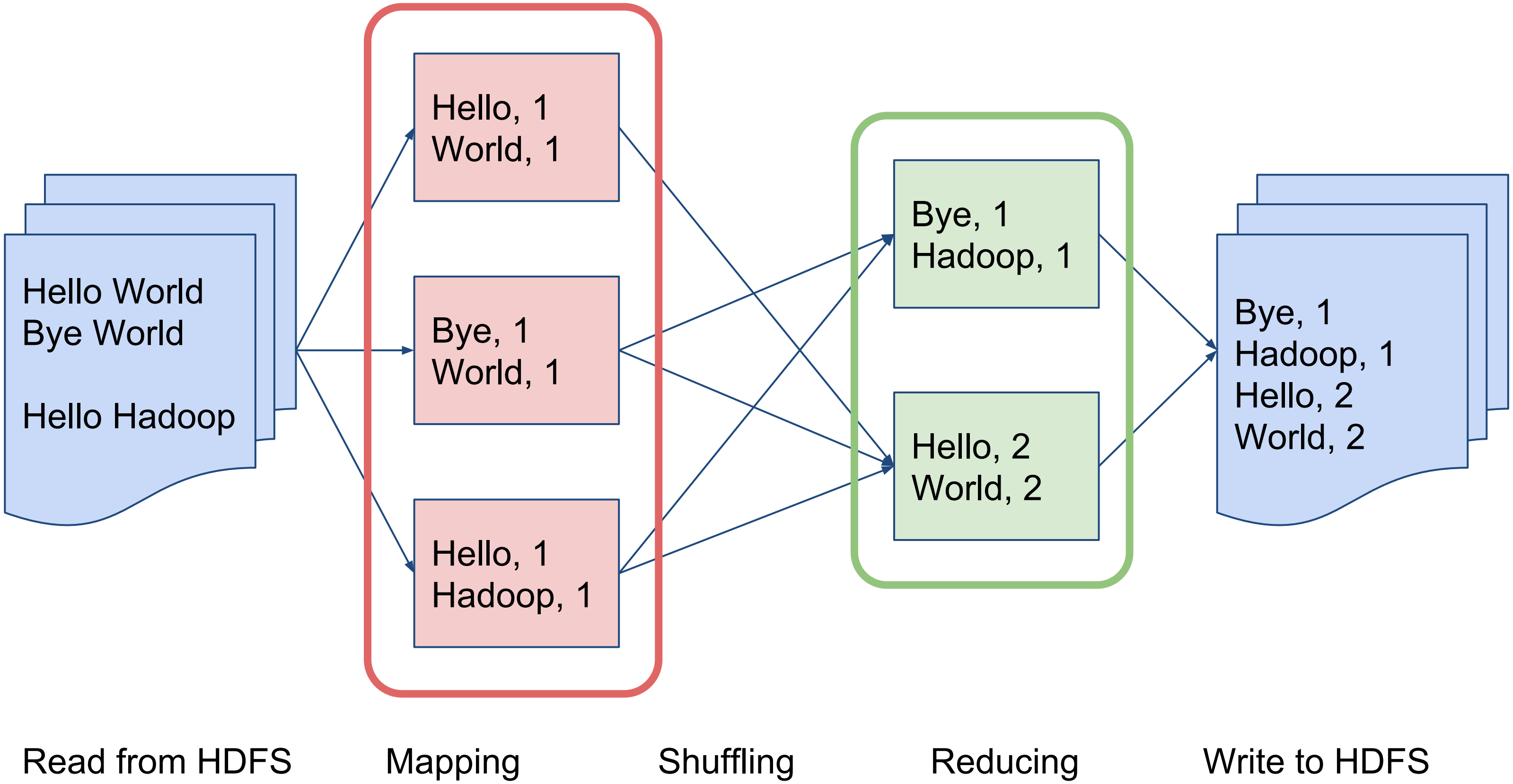
- **Hadoop Streaming** - A utility allows you to create and run MapReduce jobs with any executable or script, for example, Python, Ruby or Bash, as the mapper and/or the reducer.
- **mrJob** - A Python library which let you write MapReduce jobs and run them on a Hadoop cluster. Currently support Python 2.7/3.4+.

Outline

1. **Big Data Overview**
2. **Hadoop Fundamentals**
    2.1. The Hadoop Distributed Filesystem (HDFS)
    2.2. YARN
    2.3. MapReduce
3. **Hands-On Exercise: Basic HDFS Operations**
4. **Hands-On Exercise: Developing MapReduce Jobs**
    4.1. Example: WordCount
    4.2. Combiner

Example: WordCount

- WordCount: counts the number of occurrences of each word in a given input set.

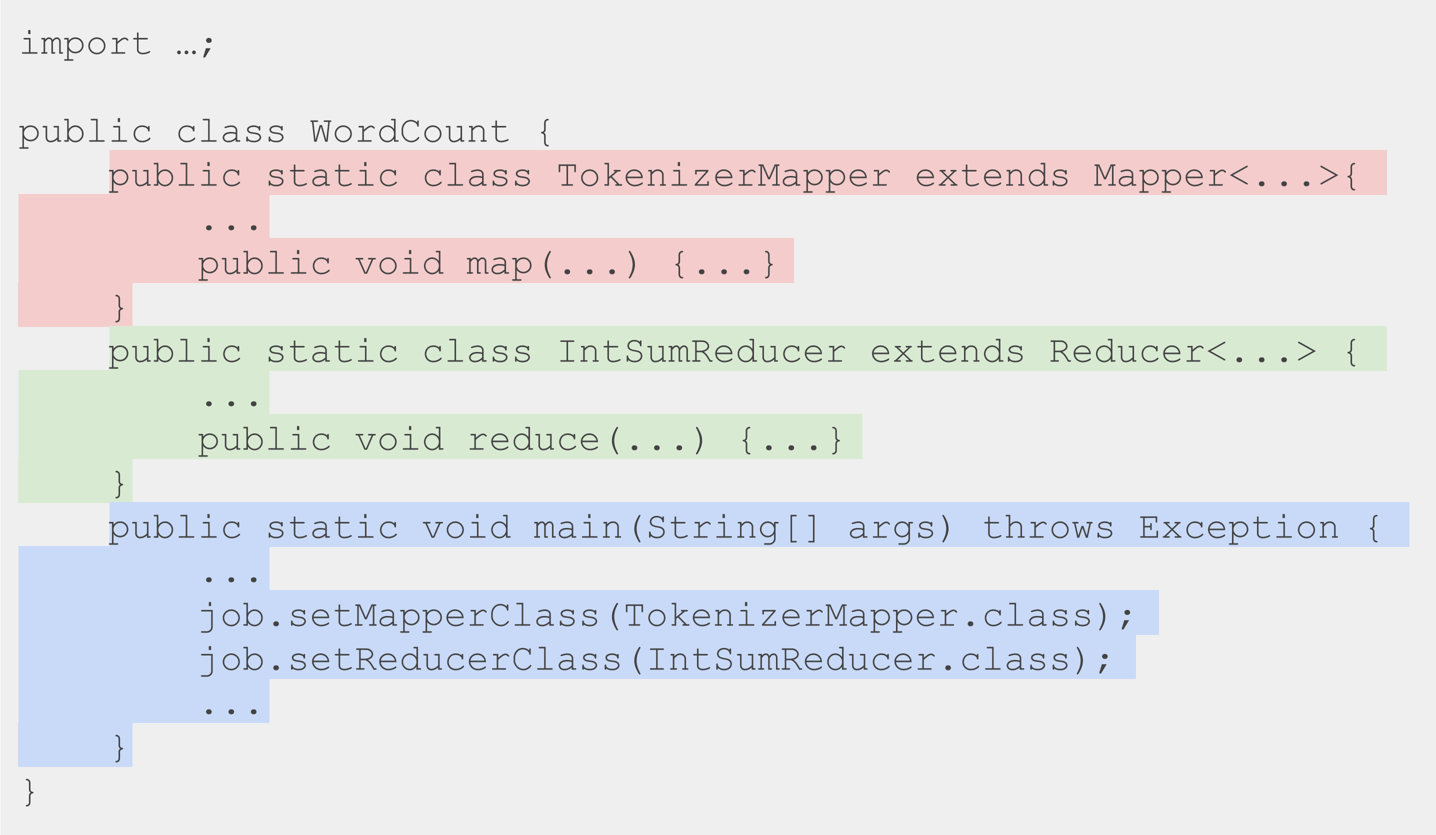


Example: WordCount

The next step is to express the framework in code. We need three things: a **mapper** function, a **reducer** function, and some code to run the job.

- We will first show the source code and job submission in **Java** without detailed explanation.
- We will then explain the implementation using both **streaming** API and **mrJob**.

Example: WordCount in Java



```
import …;

public class WordCount {
    public static class TokenizerMapper extends Mapper<...>{
        ...
        public void map(...) {...}
    }
    public static class IntSumReducer extends Reducer<...> {
        ...
        public void reduce(...) {...}
    }
    public static void main(String[] args) throws Exception {
        ...
        job.setMapperClass(TokenizerMapper.class);
        job.setReducerClass(IntSumReducer.class);
        ...
    }
}
```

The java source code can be found at:

`examples/mapReduce/wordCount/wordCount.java`

Example: WordCount in Java

To run the java application, we need to:

1. Add input data into HDFS if necessary,
2. Remove the output directory if exists to prevent error.
3. Compile the java source code and create a jar file,
4. Run the compiled jar file with `hadoop jar` command.

After running the application, you should be able to see the output file being generated in HDFS.

For details, please refer to the bash script `run-jobs.sh` in the same directory.

Launching Cluster With Mounted Volume

- Download or `git clone` the repository to your local:

  https://github.com/casunlight/hadoop-tutorial

- Move your current working directory into the examples directory and start the cluster again with the following command:

```
docker run -it \
 -p 8088:8088 \
 -p 19888:19888 \
 -p 50070:50070 \
 --name hadoop-core \
 --mount type=bind,source="$(pwd)",target=/home/hadoop/examples \
  syan83/hadoop-core
```

- When the Hadoop cluster is up, check to see if you can find the `examples/` directory inside the cluster.

Run MapReduce Application in Java

- Inside the cluster, change your directory into the wordCount directory and run the mapReduce job:

```
cd examples/mapReduce/wordCount
./run-jobs.sh
```

- The application takes about 30s to finish. Once it’s done, check your hdfs directory:

```
hadoop fs -ls -R wordcount
hadoop fs -cat wordcount/output/*
```

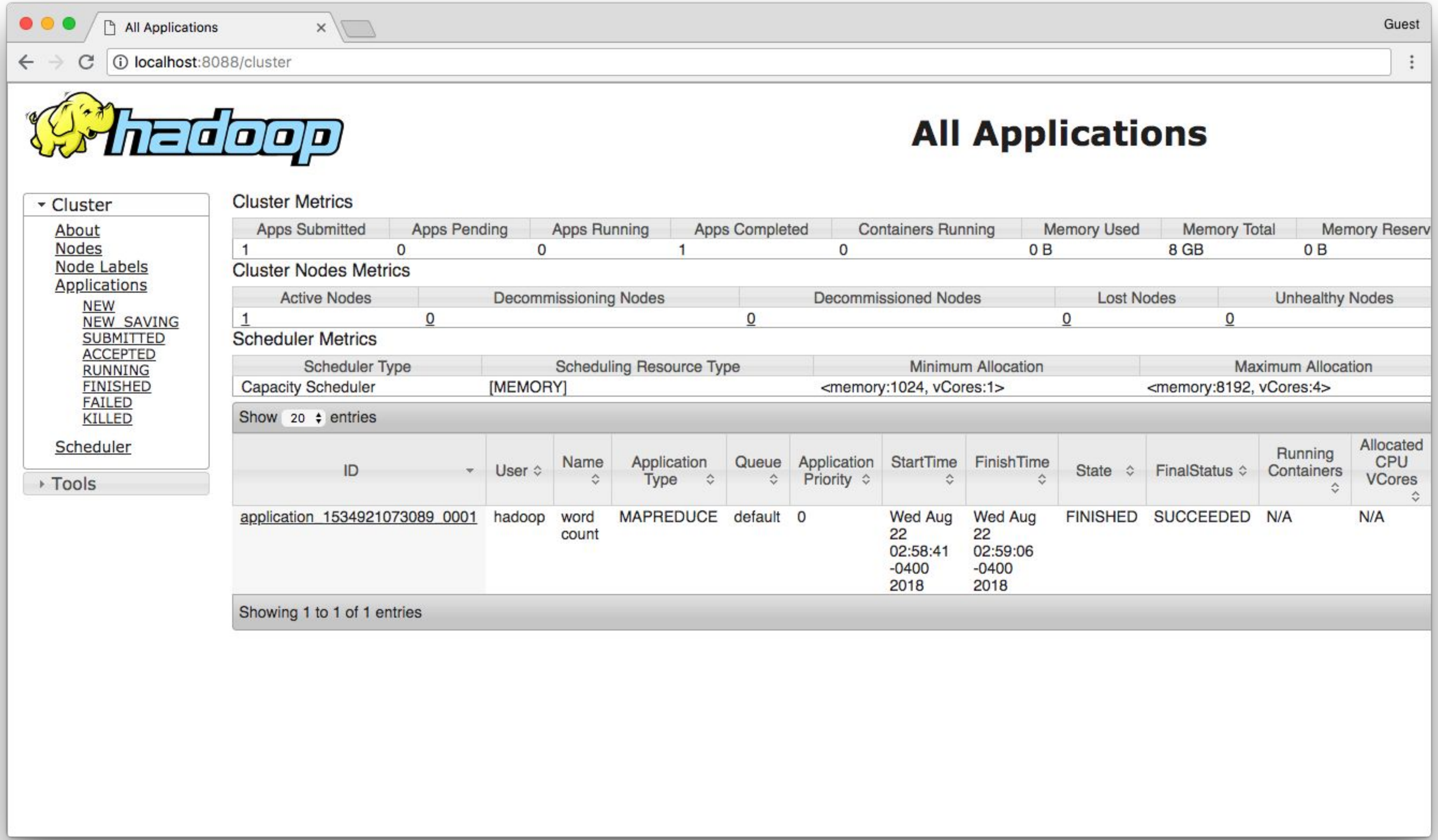
ResourceManager Web UI

- **Yarn** (ResourceManager) comes with a web UI for monitoring all applications running on the cluster.
- By default the address of the web UI is:

  http://<namenode-ip>:8088/

- If you're running your hadoop cluster locally via docker, then you can open the NameNode UI at http://localhost:8088/

ResourceManager Web UI



All Applications

Cluster Metrics

Apps Submitted	Apps Pending	Apps Running	Apps Completed	Containers Running	Memory Used	Memory Total	Memory Reserv
1	0	0	1	0	0 B	8 GB	0 B

Cluster Nodes Metrics

Active Nodes	Decommissioning Nodes	Decommissioned Nodes	Lost Nodes	Unhealthy Nodes
1	0	0	0	0

Scheduler Metrics

Scheduler Type	Scheduling Resource Type	Minimum Allocation	Maximum Allocation
Capacity Scheduler	[MEMORY]	<memory:1024, vCores:1>	<memory:8192, vCores:4>

Show 20 entries

ID	User	Name	Application Type	Queue	Application Priority	StartTime	FinishTime	State	FinalStatus	Running Containers	Allocated CPU VCores
application_1534921073089_0001	hadoop	word count	MAPREDUCE	default	0	Wed Aug 22 02:58:41 -0400 2018	Wed Aug 22 02:59:06 -0400 2018	FINISHED	SUCCEEDED	N/A	N/A

Showing 1 to 1 of 1 entries

Using the Hadoop Streaming Utility

- Hadoop Streaming is a utility that comes with Hadoop that enables you to develop MapReduce executables in languages other than Java.
- To use Hadoop's Streaming utility, the process is:
  1. Write **mapper** and **reducer** executable in the programming language of your choice. Here we will use Python.
  2. Test executables locally with Linux Piping.
  3. Use Hadoop streaming interface to run your application.

WordCount With Hadoop Streaming: Mapper

The following code sample is a mapper executable written in Python:

```
#!/usr/bin/python
from __future__ import print_function
import sys

def mapper():
    for line in sys.stdin:
        line = line.strip()
        words = line.split()
        for word in words:
            print('{}\t{}'.format(word, 1))

if __name__ == '__main__':
    mapper()
```

Note: in hadoop streaming, by default the prefix of a line up to the first tab character (`\t`) is the **key** and the rest (excluding the tab character) is the **value**.

WordCount With Hadoop Streaming: Reducer

The following code sample is a reducer executable written in Python

```
def reducer():
    current_word = None
    current_count = 0
    word = None
    for line in sys.stdin:
        word, count = line.strip().split('\t', 1)
        try:
            count = int(count)
        except ValueError:
            continue
        if current_word == word:
            current_count += count
        else:
            if current_word:
                print('{}\t{}'.format(current_word, 
current_count))
            current_count = count
            current_word = word
    if current_word == word:
        print('{}\t{}'.format(current_word, current_count))
```

Testing Hadoop Streaming Application Locally

- It’s recommended to test your `mapper.py` and `reducer.py` locally before deploying to the cluster using piping in Linux:

```
cat <local_input> | ./mapper.py | sort | ./reducer.py
```

- Note: make sure to include a shebang at the beginning of your scripts and make them self-executable:
  - Shebang for python: `#!/usr/bin/python`
  - Adding execution permission to user: `chmod u+x <script_path>`

Submitting a Hadoop Streaming Job

- To submit a mapReduce streaming job, run:

```
hadoop jar <path_to_streaming_jar>/hadoop-streaming.jar \
    -files mapper.py,reducer,py
    -input <inputDirs> \
    -output <outputDir> \
    -mapper mapper.py \
    -reducer reducer.py
```

- The utility will then create a Map/Reduce job, submit the job to the cluster, and monitor the progress until it completes.

Run MapReduce Streaming Jobs

- Change your working directory to `examples/streaming/wordCount/`
- Test `mapper.py`/`reducer.py` locally with the following command:

```
cat file01 | ./mapper.py | sort | ./reducer.py
```

- Submit a streaming job by executing the bash script `run-jobs.sh`.
- Check the output files in HDFS when the job completes.
- Monitor the jobs via ResourceMonitor WebUI.
- (Optional) Modify your `mapper.py` to remove punctuation and convert words to lowercases, and then run the job again.

Yelp’s mrJob

mrJob is a open-source Python library developed at Yelp aiming to help developers to write multi-step Hadoop Streaming jobs and to run/deploy them on several platforms. You can:

- Write multi-step MapReduce jobs in pure Python
- Test on your local machine
- Run on many different platforms including: Hadoop cluster, Amazon Elastic MapReduce (EMR), Google Cloud Dataproc (Dataproc), etc.

WordCount With mrJob

The source code below shows the wordCount example in mrJob:

```
from mrjob.job import MRJob


class MRWordCount(MRJob):

    def mapper(self,  , line):
        Words = line.strip().split()
        for word in words:
                yield word, 1

    def reducer(self, key, values):
        yield key, sum(values)


if   name   == '  main__':
    MRWordCount.run()
```

Running Your Job with mrJob

mrJob allows you to run your job in different ways by passing different parameters along with the job submission:

- Run your job in local mode:

```
python my_job.py <input_data>
```

- Run your job in Hadoop cluster:

```
python wordCountMRJob.py -r hadoop <input_data>
```

Note:

1. The input path could be either a local directory or a HDFS directory.
2. If no `--output-dir` is specified, by default the output will be directed to stdout.

Run MapReduce Jobs With mrJob

- Change your working directory to `examples/mrJob/wordCount/`
- Test locally with the following command:

```
python3 wordCountMRJob.py file01
```

- Test on the cluster with option `-r hadoop`:

```
python3 wordCountMRJob.py -r hadoop file01
```

- Submit a streaming job by executing the bash script `run-jobs.sh`.
- Check the output files in HDFS when the job completes.
- Monitor the jobs via ResourceMonitor WebUI.
- (Optional) Modify your `MRWordCount` class to remove punctuation and convert words to lowercases, and then run the job again.

Outline

1. **Big Data Overview**
2. **Hadoop Fundamentals**
    - 2.1. The Hadoop Distributed Filesystem (HDFS)
    - 2.2. YARN
    - 2.3. MapReduce
3. **Hands-On Exercise: Basic HDFS Operations**
4. **Hands-On Exercise: Developing MapReduce Jobs**
    - 4.1. Example: WordCount
    - **4.2. Combiner**

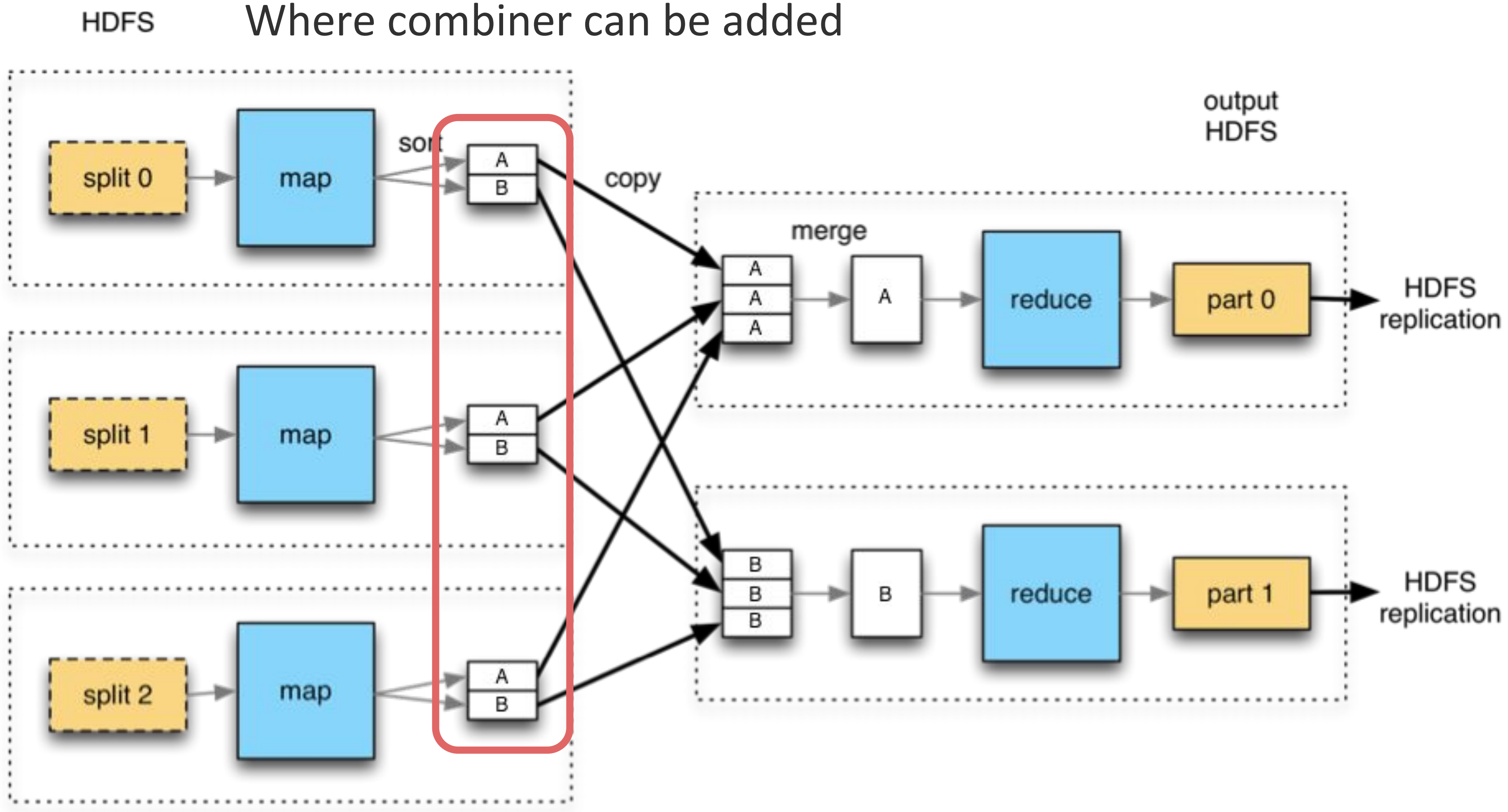
Combiners

Hadoop Combiner is an optional class in the MapReduce framework:

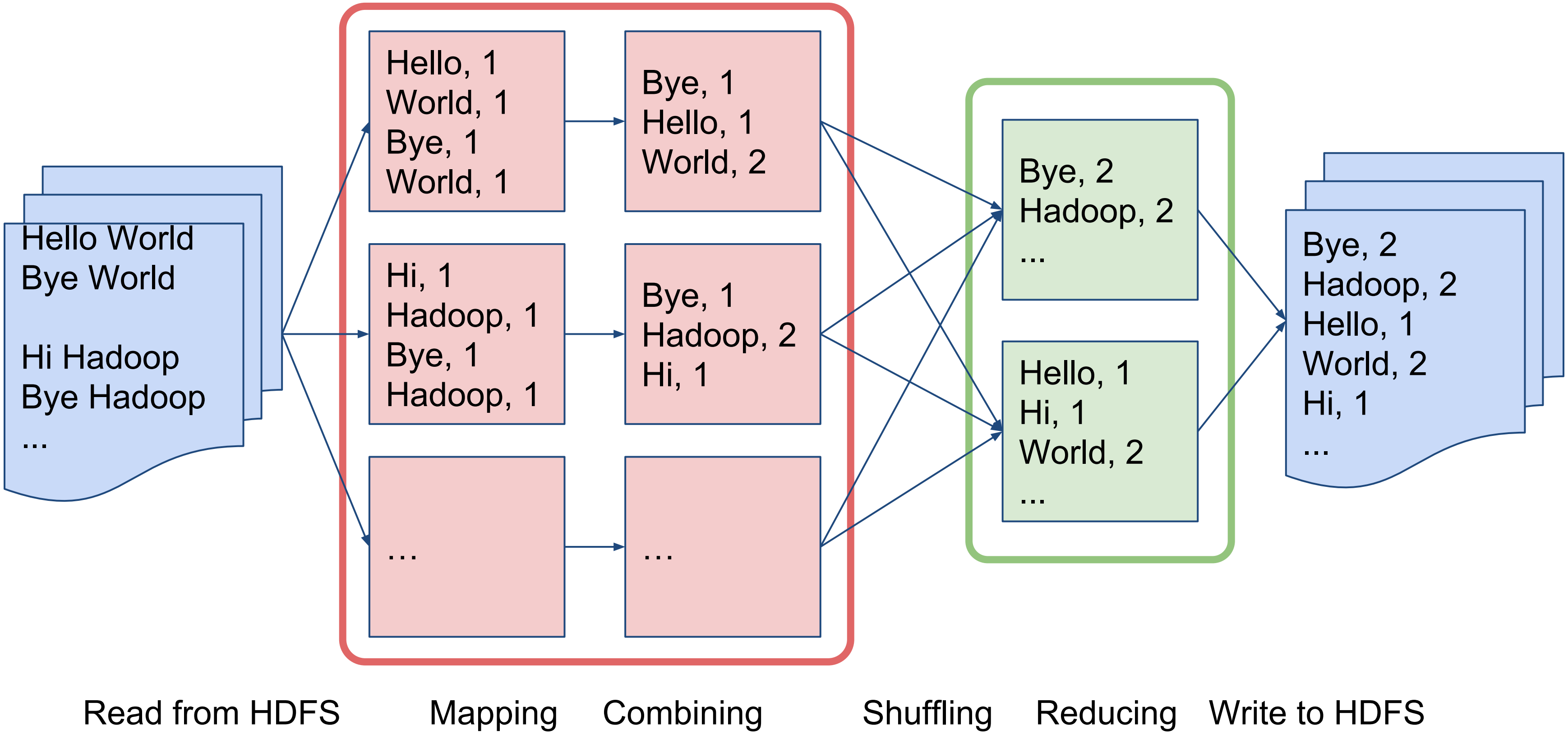
- added in between the mapper and the reducer,
- used to reduce the amount of data received by reducer by combining the data output from mapper.
- The main function of a Combiner is to summarize the output from mapper so that the stress of data processing from reducer can be managed and network congestion can be handled.

MapReduce With a Combiner

Where combiner can be added



WordCount With Combiner



Hadoop Streaming Job With Combiner Option

- To add a combiner to a mapReduce streaming job, run:

```
hadoop jar <path_to_streaming_jar>/hadoop-streaming.jar \
    -files mapper.py,reducer,py,combiner.py
    -input <inputDirs> \
    -output <outputDir> \
    -mapper mapper.py \
    -reducer reducer.py \
    -combiner combiner.py \
```

- Usually combiner and reducer share the same code. If we can reuse the reducer code (which is true for the WordCount example), then we just need to make the following change:

```
-combiner reducer.py
```

mrJob With Combiner

- If you define a combiner function in your MRJob class, then a combiner step will be automatically added after the mapper.
- Again, in the WordCount example, the combiner function should be defined in the same way as reducer function.

Running WordCount With Combiner

1. Move to `examples/streaming/wordCount/` directory and execute the bash script `run-jobs-2.sh`, compare the log output (particularly `Combine input records` and `Combine output records`)of the job execution with the previous streaming job.
2. Move to `examples/mrJob/wordCount/` directory and add a combiner function to your `MRWordCount` class, then rerun the `run-jobs.sh` and check the log output.