Hadoop & MapReduce

NYC Data Science Academy

Outline

1. **An Overview Of Big Data And Hadoop**
2. **HDFS**
    2.1. HDFS Concepts
    2.2. HDFS CLI Operations
3. **YARN**
4. **MapReduce**
    4.1. MR Streaming Interface
    4.2. mrJob

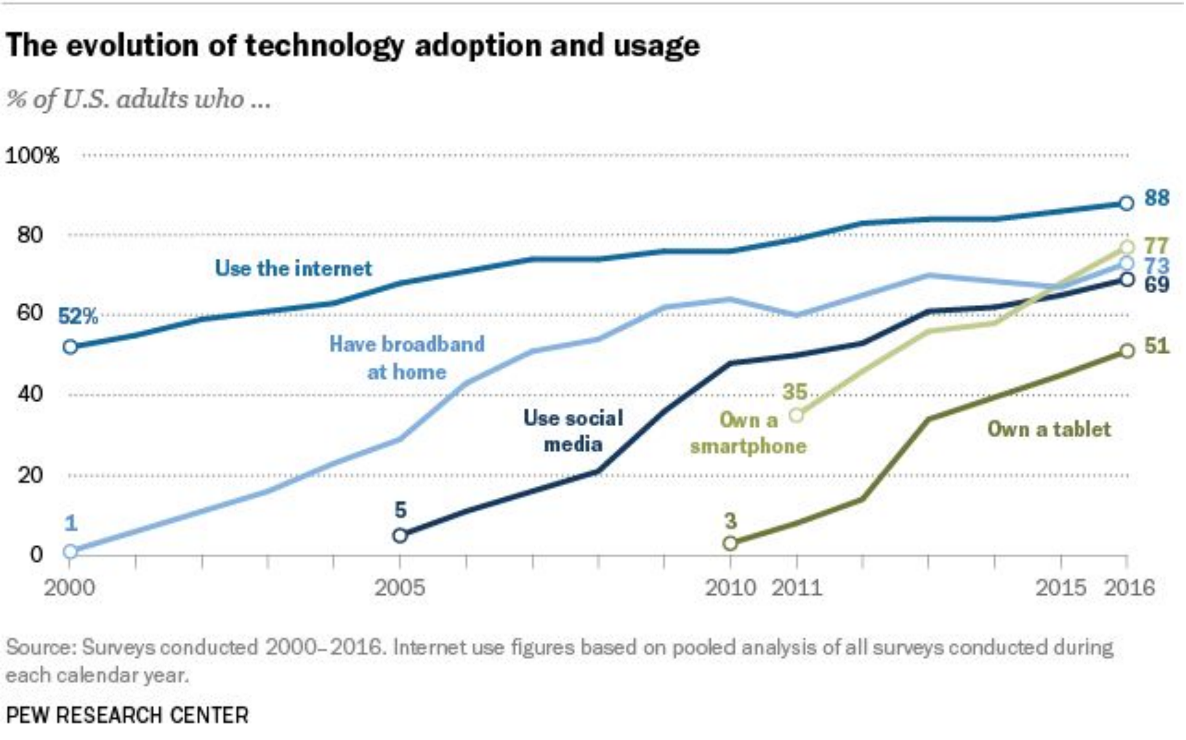
Appendix: MR Java Interface

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)

Big Data: What's the problem?

- Every day, we create 2.5 quintillion (10^{18}) bytes of data. To put that into perspective, 90% of the data in the world today has been created in the last two years. - 10 Key Marketing Trends for 2017



Traditional Large-Scale Computation

- The good solution: use powerful computers
  - Faster CPU, bigger memory

- The better solution: distributed systems
  - Use multiple computers

The IBM Blue Gene/P supercomputer "Intrepid" at Argonne National Laboratory runs 164,000 processor cores.[wikipedia]

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
  - Usage: Larger data sets and faster computation
  - Challenges: Programming complexity and machine failure

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

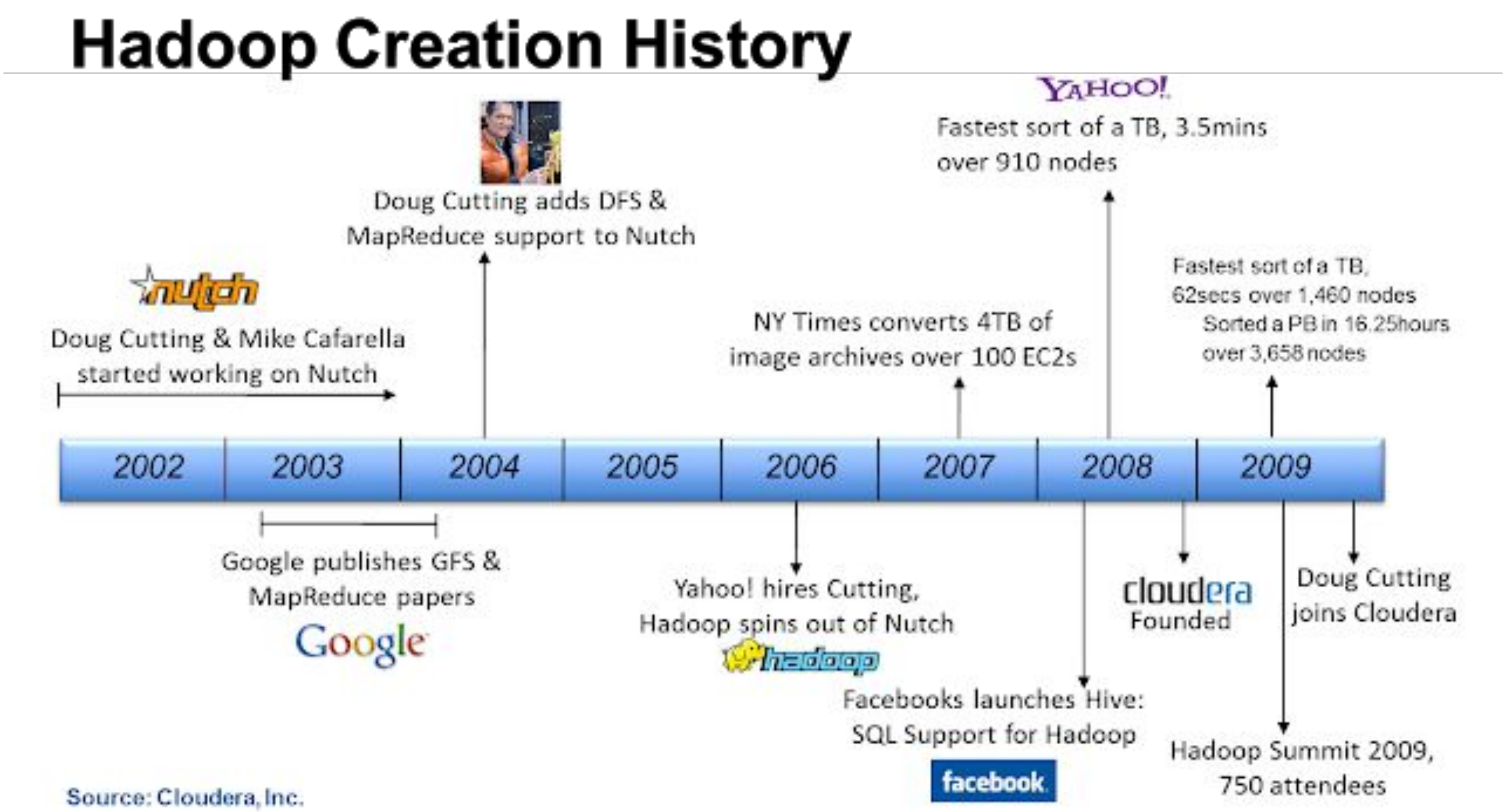
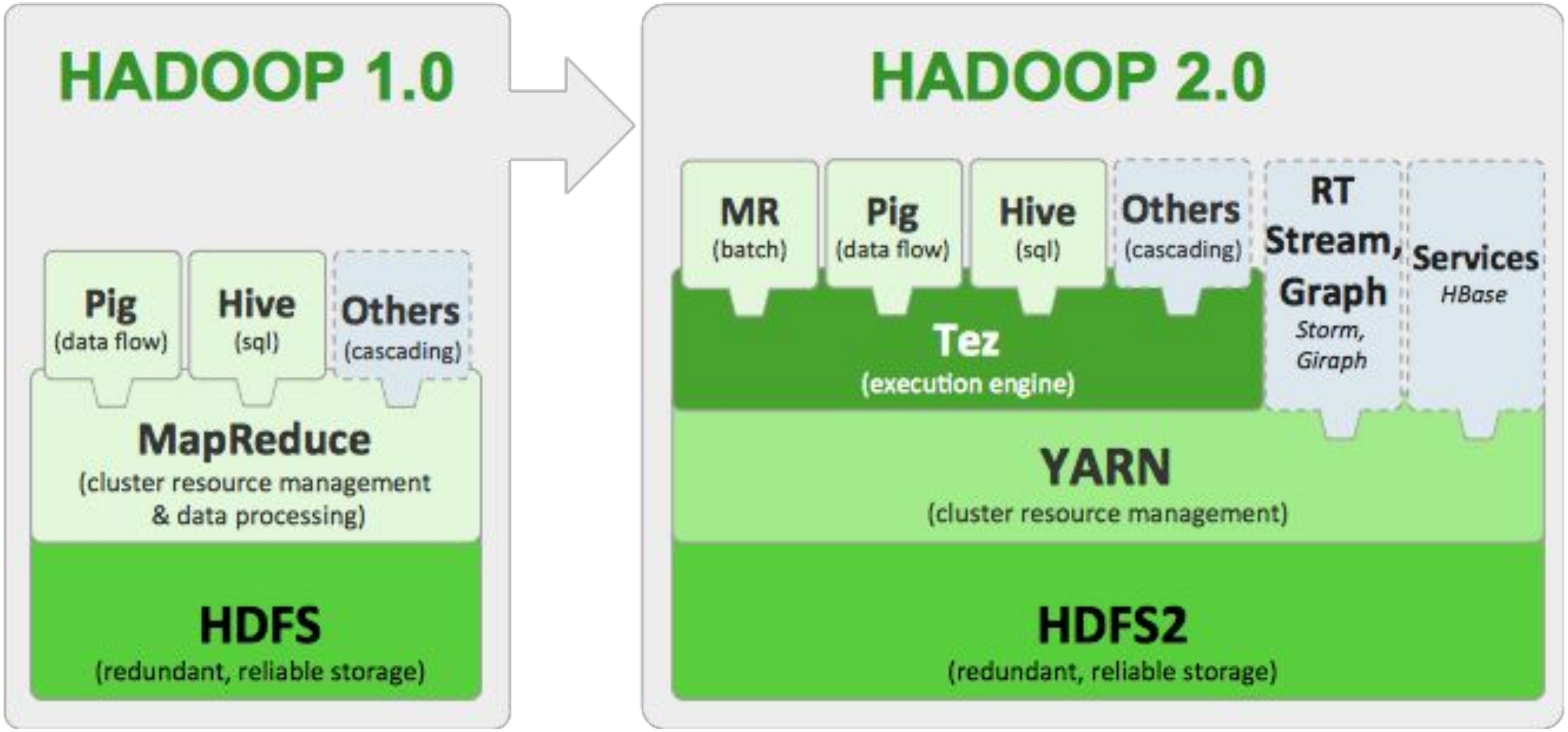


Image Link

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



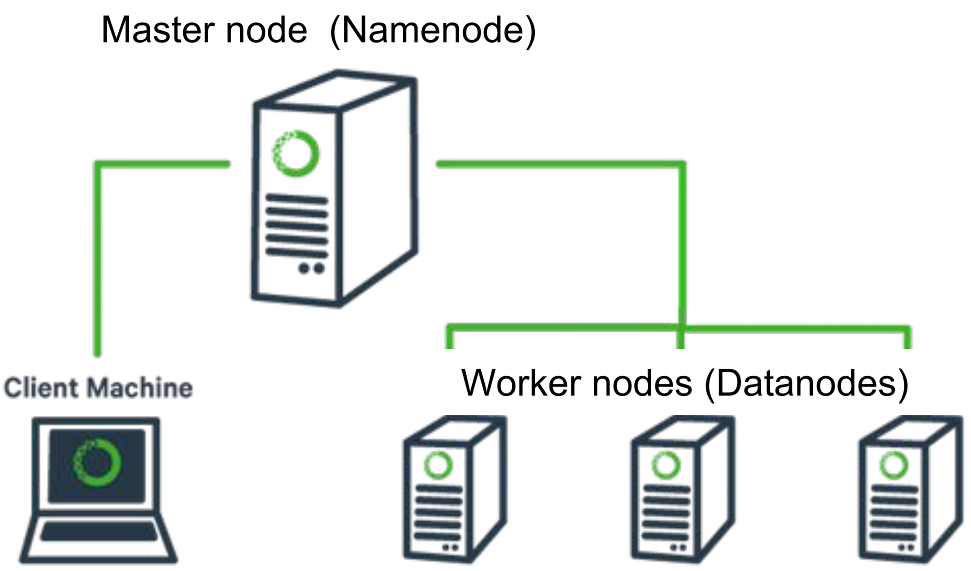
[image link]

Architecture of Hadoop Cluster

- A **cluster** is a group of computer working together
  - Provides data storage, data processing, and resource management
- A **node** is an individual computer in the cluster
  - **Master** nodes manage distribution of work and data
  - **Worker** nodes save the data and execute the work
- A **daemon** is a program running on a node
  - ResourceManager - runs on master nodes
  - NodeManager - runs on worker nodes
  - ApplicationMaster - manages application lifecycle and task scheduling

Architecture of Hadoop Cluster

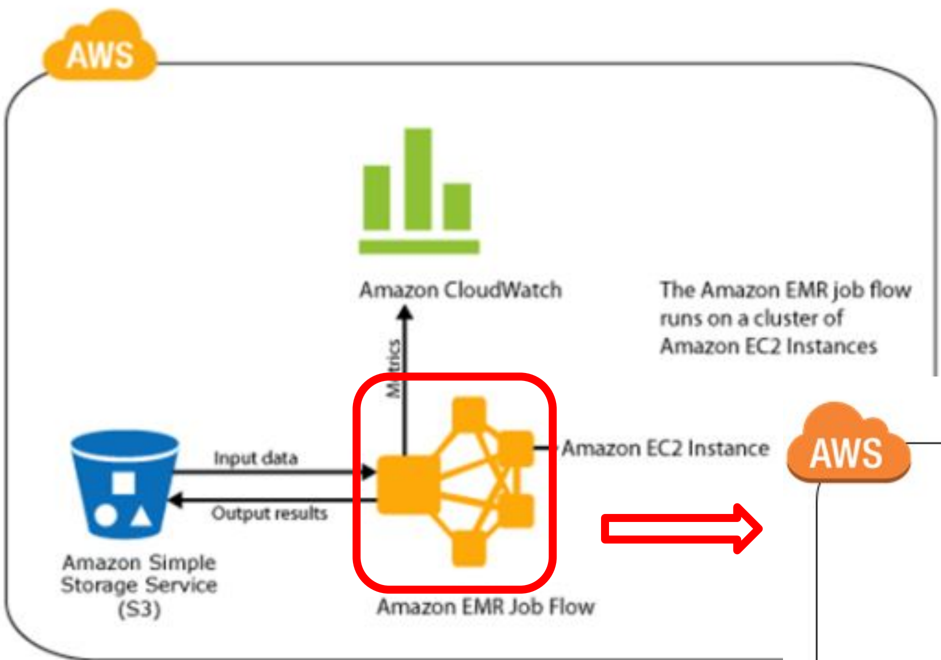
❖ Hadoop Architecture

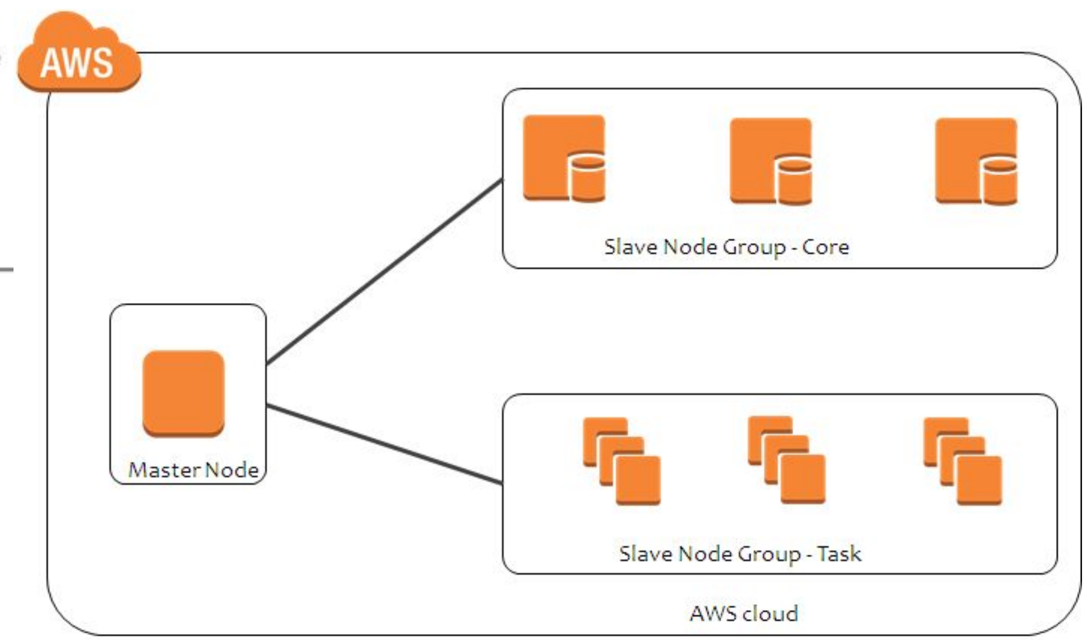


Master Nodes and Worker Nodes

- Master nodes (Namenodes) manage the work
  - Master nodes are essential
  - Daemons running on master nodes ensure that the entire cluster works
  - A failed daemon could cause the entire cluster to become unusable
- Worker/Slave nodes (Datanodes) do the work
  - Worker nodes are expendable
  - Daemons running on worker nodes handle data processing
  - A failed worker node will not bring down the entire cluster

Hadoop On AWS (EMR)





Top Image
Right Image

Hadoop On GCP (DataProc)

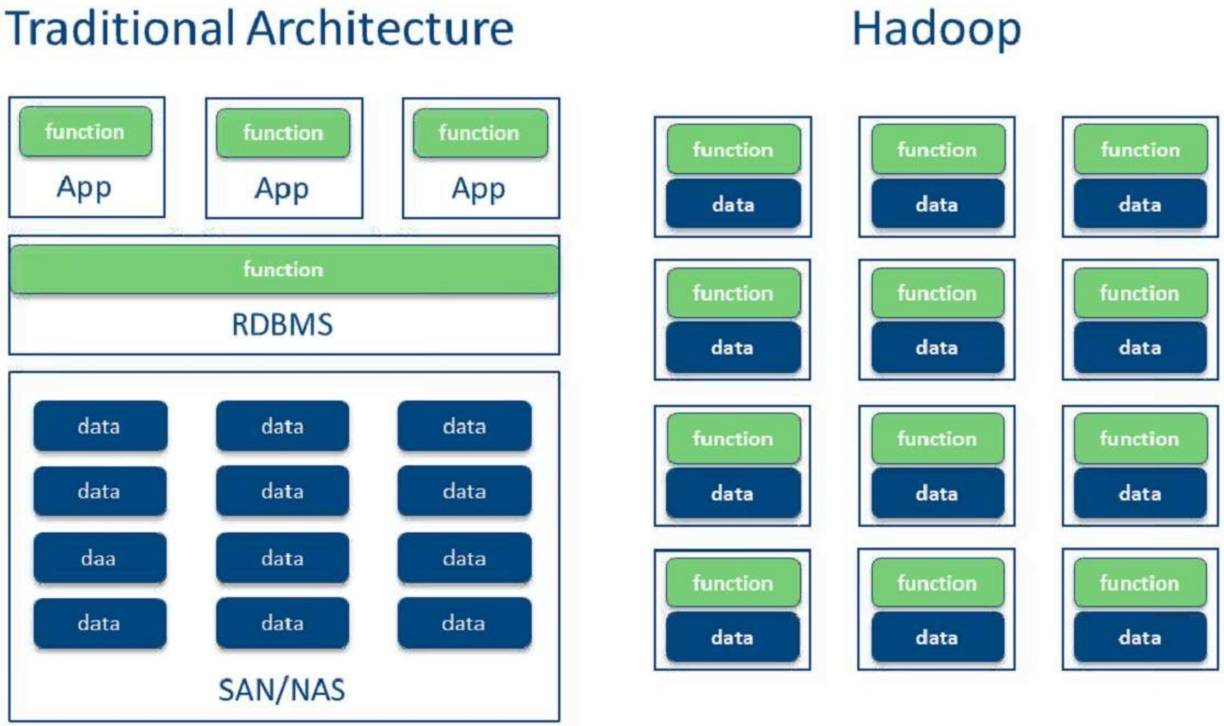
Google Cloud's Hadoop service (Dataproc) integrates Google Cloud Platform products, including Google BigQuery and Google Cloud Bigtable, etc.

Image Source

Outline

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.



Assumption And 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.

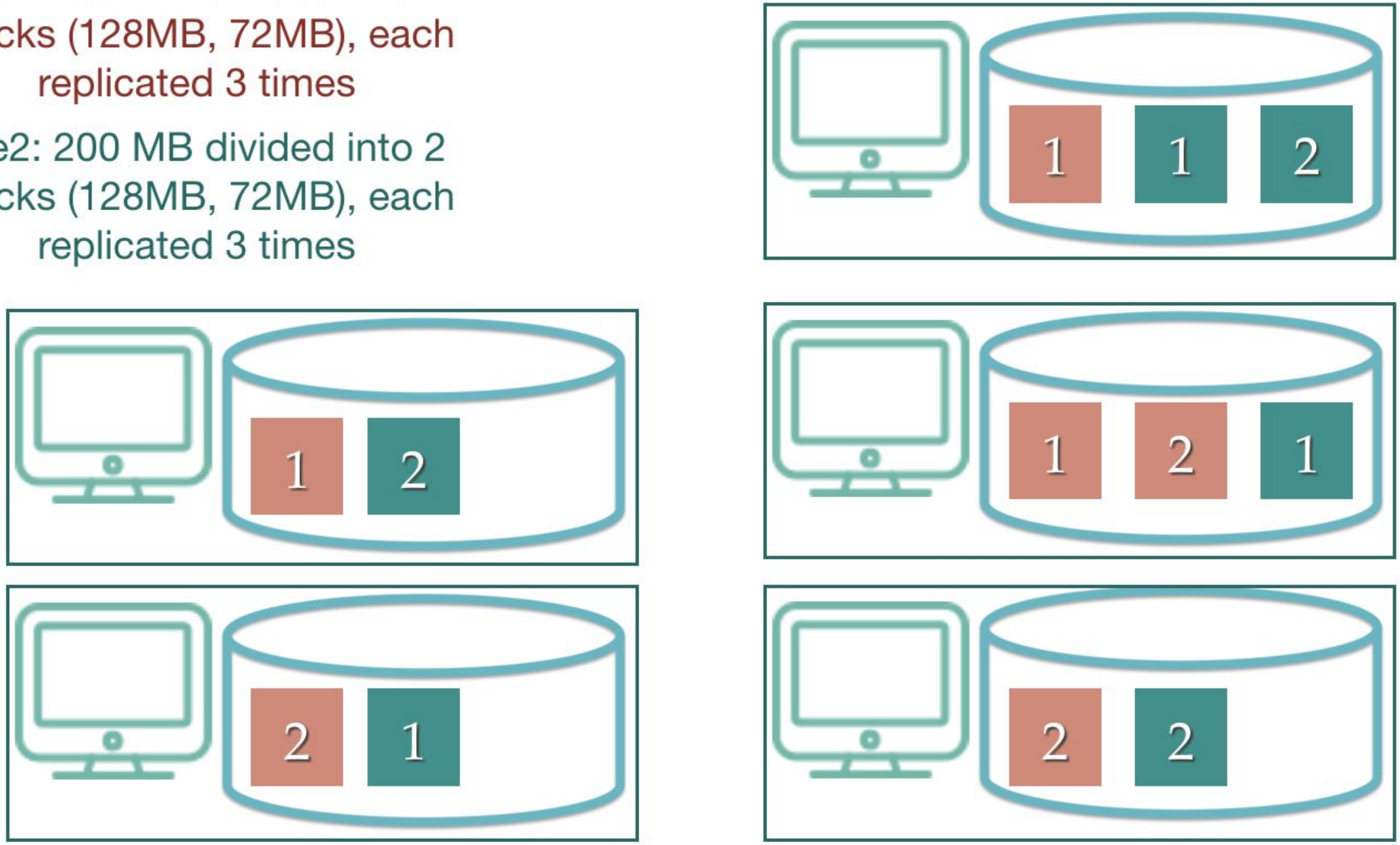
Properties of HDFS

- *Data is accessed sequentially*
  - HDFS is tuned for high-speed throughput of entire files, *not* random access.
- *Data is read-only*
  - Appends are permitted, no random writes are allowed.
- *Partitioning for reliability and efficiency*: Files are broken into replicated **blocks** of 128MB or 64MB (depending upon the Hadoop configuration), for reliability and parallel processing.

How HDFS Stores Data

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

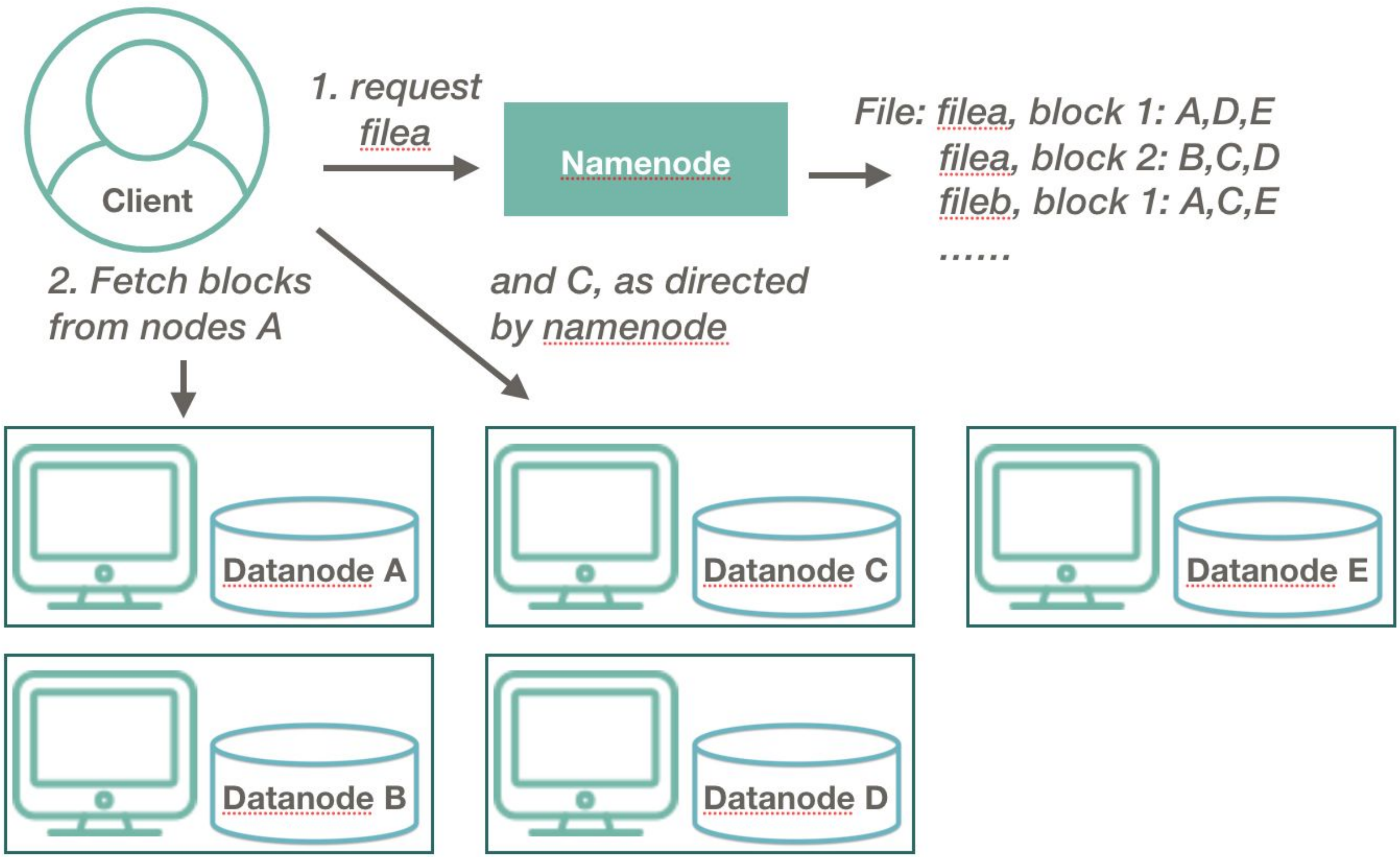
File2: 200 MB divided into 2 blocks (128MB, 72MB), each replicated 3 times



How HDFS Stores Data

- 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.

How HDFS Stores Data



Outline

1. **An Overview Of Big Data And Hadoop**
2. **HDFS**
   2.1. HDFS Concepts
   2.2. HDFS CLI Operations
3. **YARN**
4. **MapReduce**
   4.1. MR Streaming Interface
   4.2. mrJob

Appendix: MR Java Interface

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 HDFS 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
```

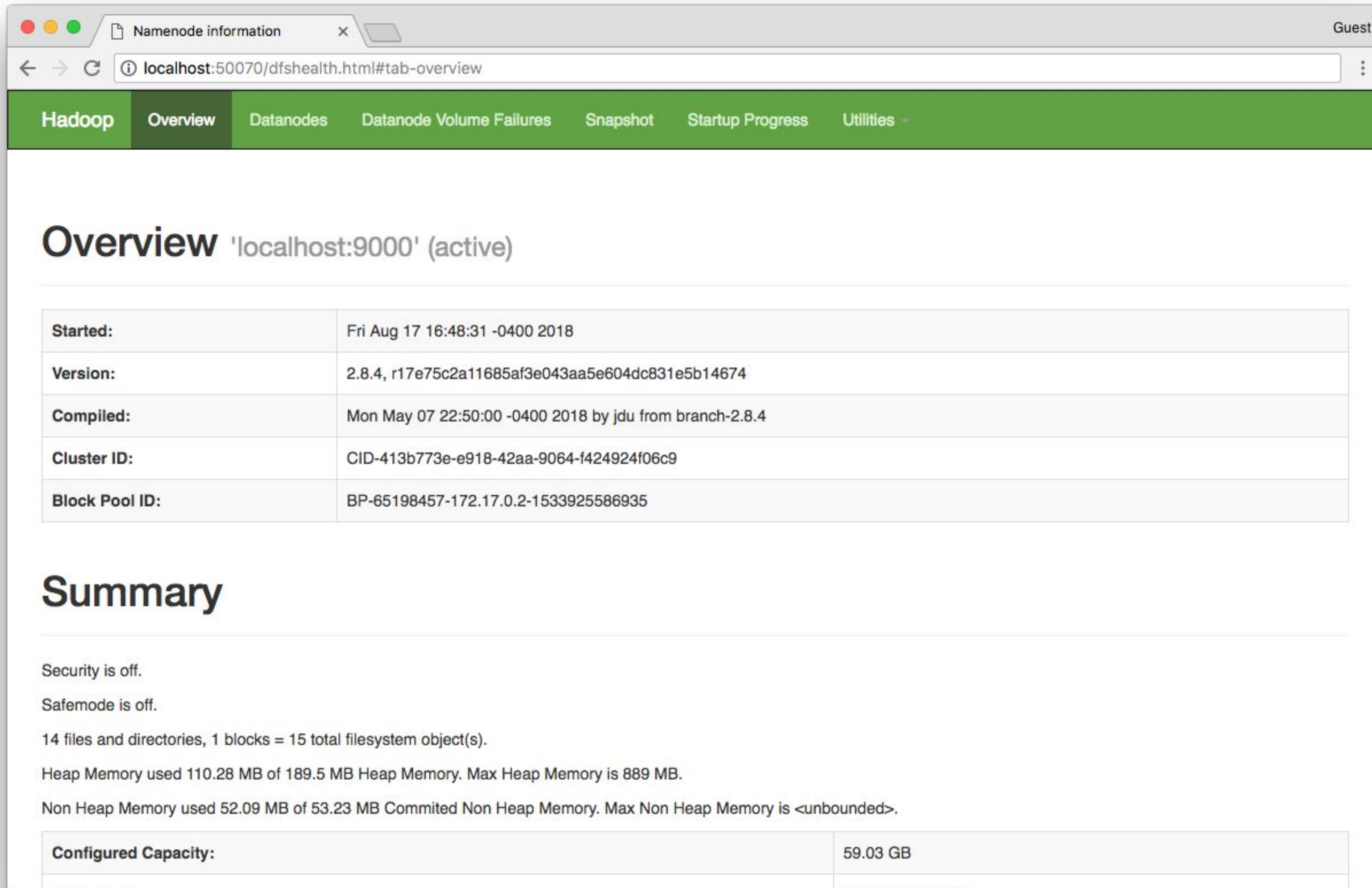
8. Remove the two files:

```
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



Outline

1. **An Overview Of Big Data And Hadoop**
2. **HDFS**
    2.1. HDFS Concepts
    2.2. HDFS CLI Operations
3. **YARN**
4. **MapReduce**
    4.1. MR Streaming Interface
    4.2. mrJob

Appendix: MR Java Interface

YARN

- YARN stands for "Yet Another Resource Negotiator", which is a resource management layer of Hadoop.
- Core components:
  - **Resource manager**: managing the use of resources across the cluster (one per cluster)
  - **Node managers**: running on all nodes to launch and monitor containers

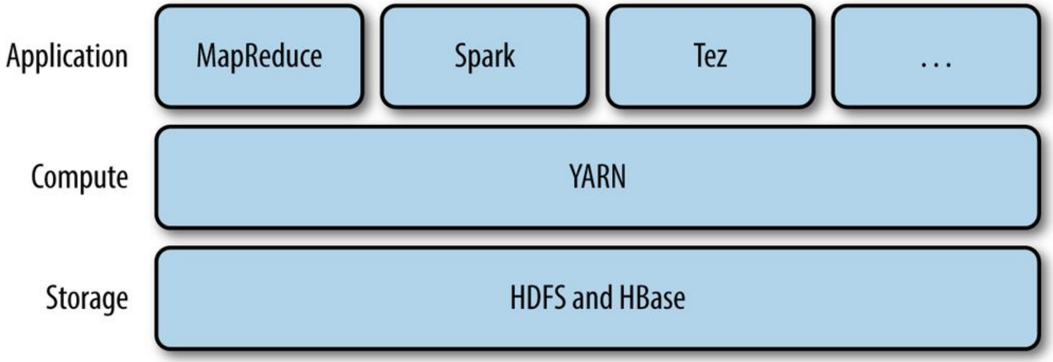


Image Source

How YARN Runs An Application

Resource manager send requests to node manager to launches **application master** process, one for each MR job.

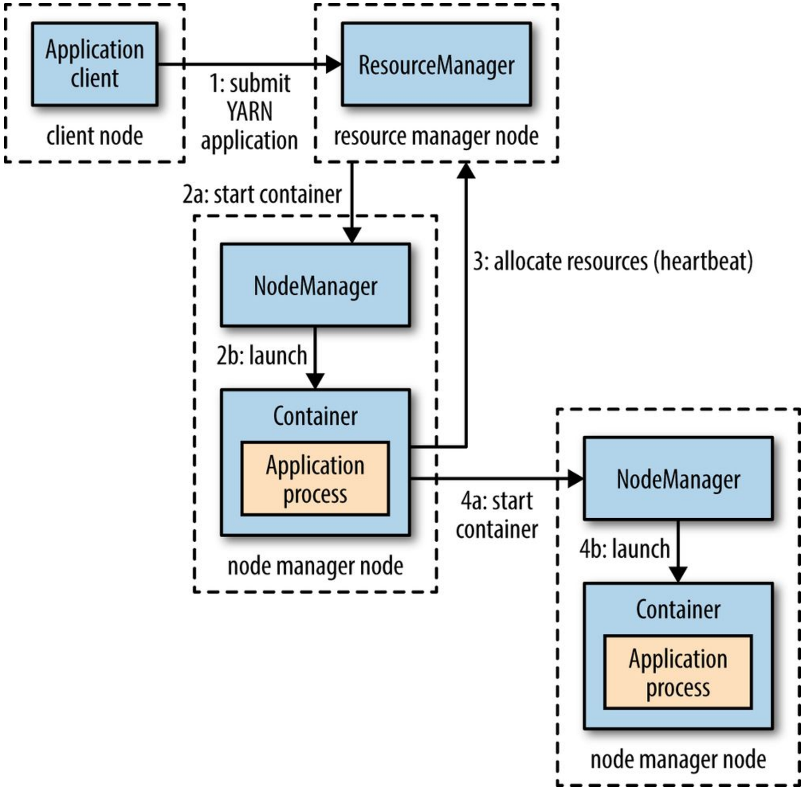


Image Source

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. **An Overview Of Big Data And Hadoop**
2. **HDFS**
    2.1. HDFS Concepts
    2.2. HDFS CLI Operations
3. **YARN**
4. **MapReduce**
    4.1. MR Streaming Interface
    4.2. mrJob

Appendix: MR Java Interface

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.

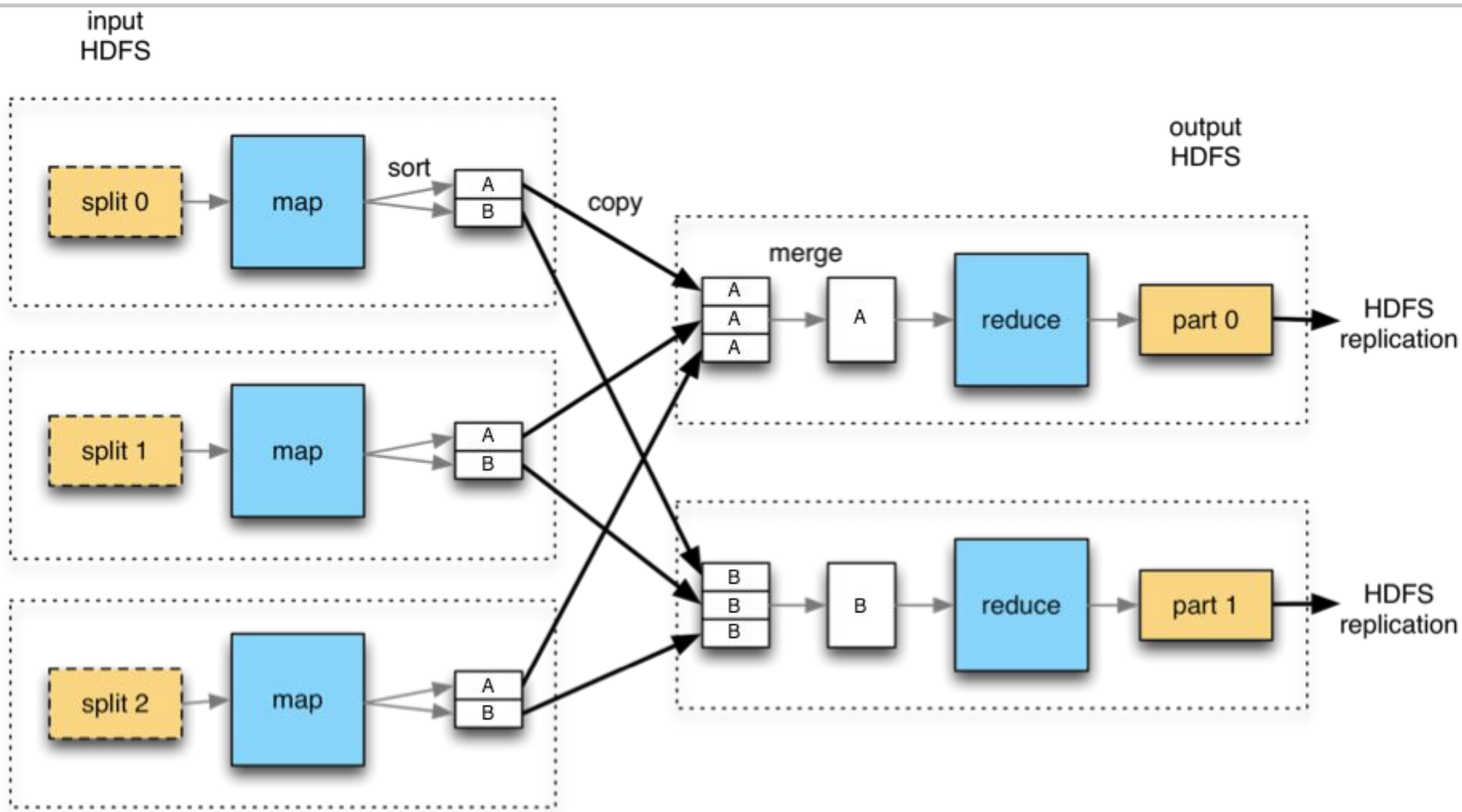
Computing with MapReduce

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/

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.

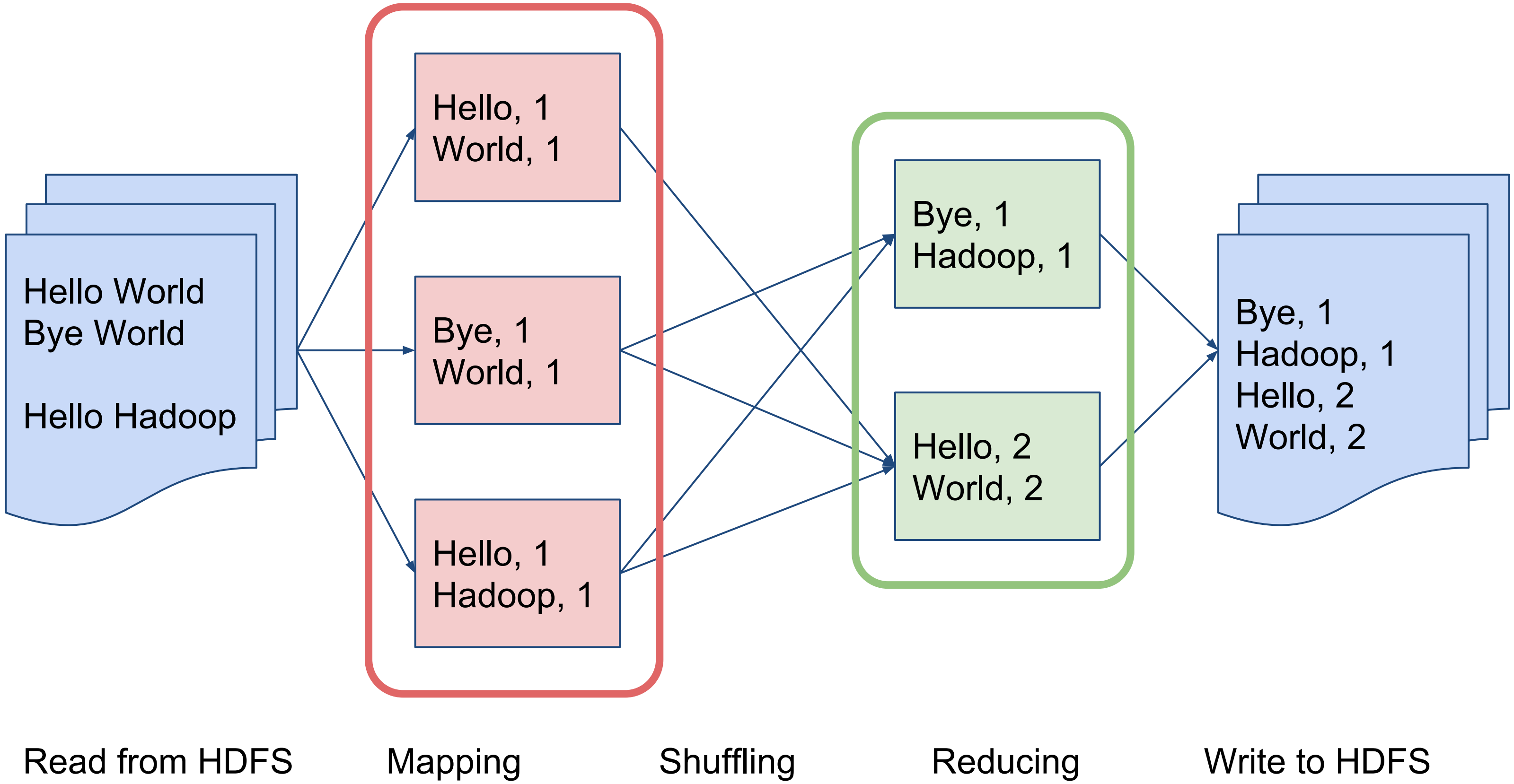
Developing MapReduce Applications

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+.

MR Example: WordCount

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



MR 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**.

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.

Outline

1. **An Overview Of Big Data And Hadoop**
2. **HDFS**
    2.1. HDFS Concepts
    2.2. HDFS CLI Operations
3. **YARN**
4. **MapReduce**
    4.1. MR Streaming Interface
    4.2. mrJob

Appendix: MR Java Interface

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.

Launching Cluster With Mounted Volume

1. Download or `git clone` the repository to your local:

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

2. 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
```

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

Run MapReduce Streaming Jobs

1. Change your working directory to `examples/streaming/wordCount/`

2. Test `mapper.py`/`reducer.py` locally with the following command:

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

3. Submit a streaming job by executing the bash script `run-jobs.sh`.

4. Check the output files in HDFS when the job completes.

5. (Optional) Modify your `mapper.py` to remove punctuation and convert words to lowercases, and then run the job again.

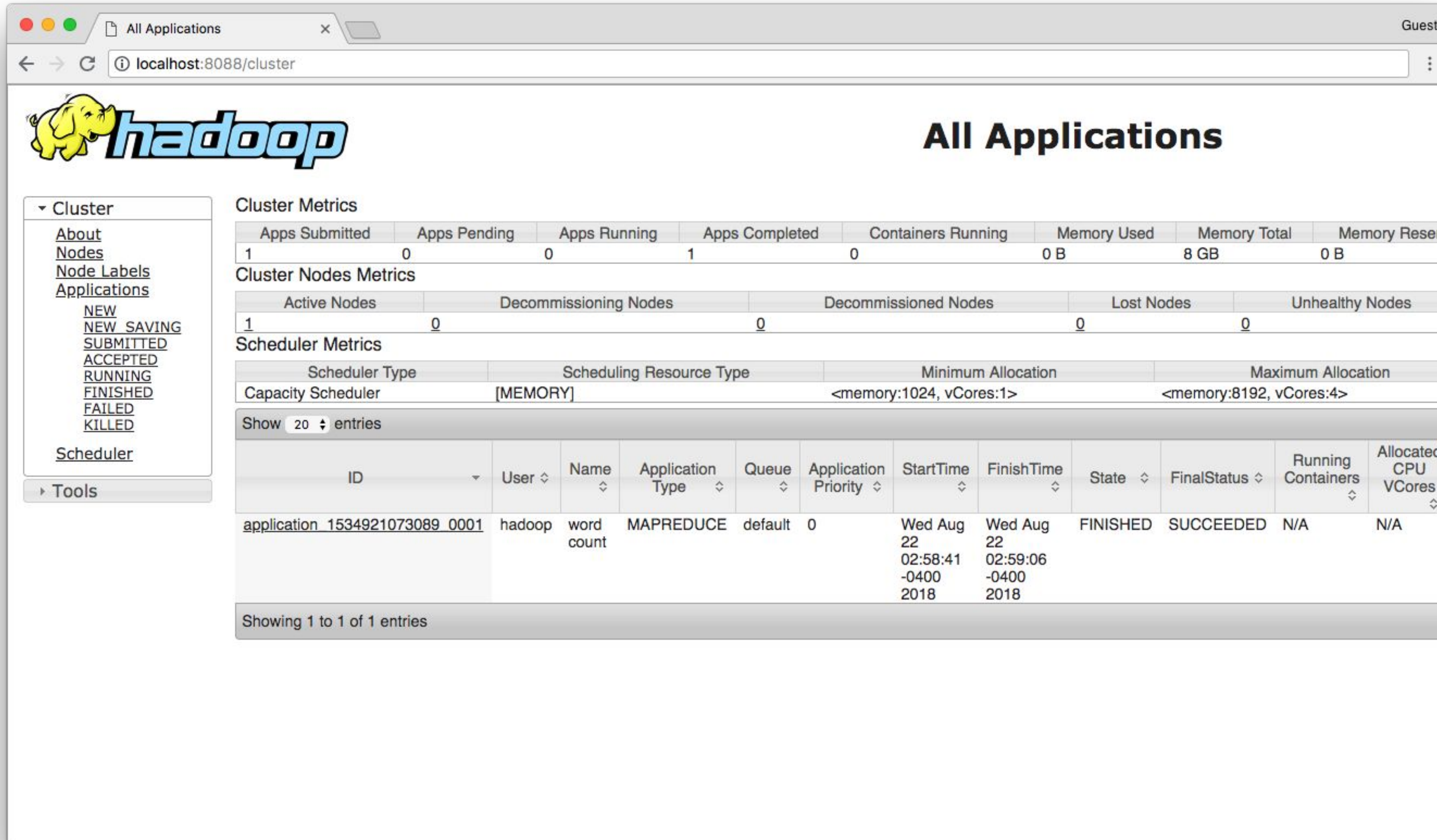
ResourceManager Web UI

- **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



Outline

1. **An Overview Of Big Data And Hadoop**
2. **HDFS**
    2.1. HDFS Concepts
    2.2. HDFS CLI Operations
3. **YARN**
4. **MapReduce**
    4.1. MR Streaming Interface
    4.2. mrJob

Appendix: MR Java Interface

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.

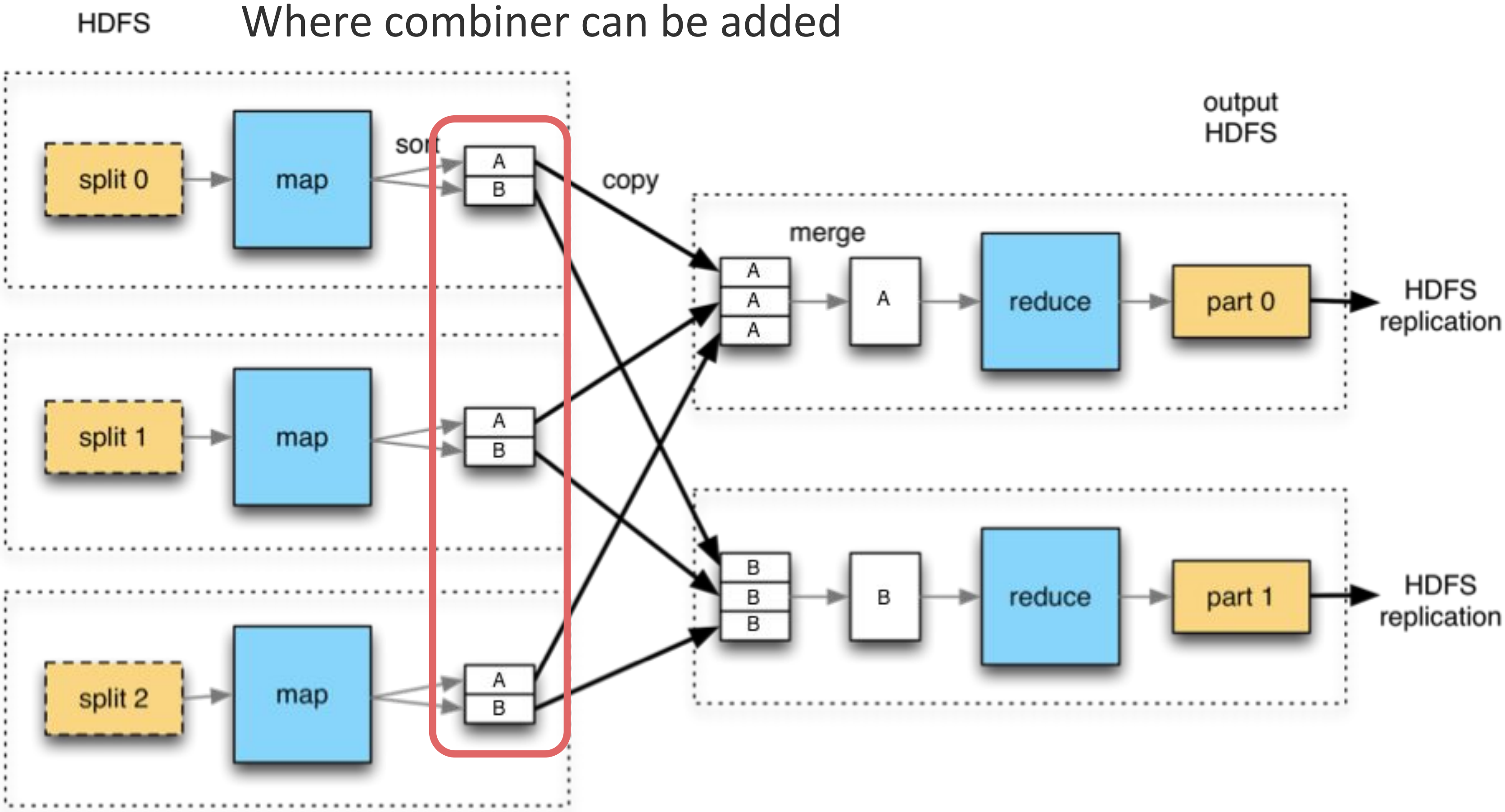
Combiners

Hadoop Combiner is an optional class in the MapReduce framework:

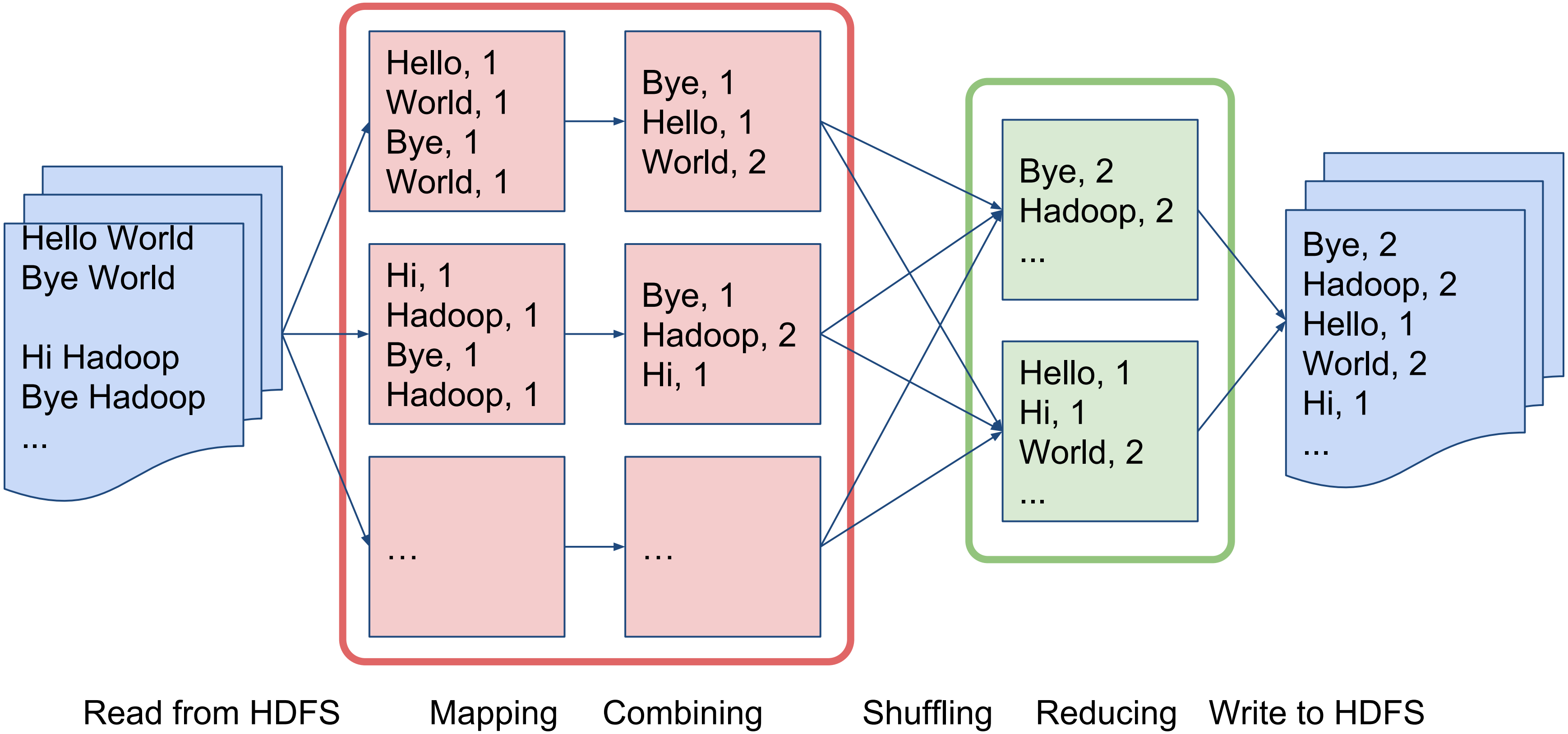
- added in between the mapper and the reducer,
- used to reduce the amount of data received by reducer by combining the data output from mapper.
- The main function of a Combiner is to summarize the output from mapper so that the stress of data processing from reducer can be managed and network congestion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

```
...

class MRWordCount(MRJob):
    def mapper(self, _, line):
        ...

    def reducer(self, key, values):
        ...

    def combiner(self, key, values):
        ...

if __name__ == '__main__':
    ...
```

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.

Outline

1. **An Overview Of Big Data And Hadoop**
2. **HDFS**
   - 2.1. HDFS Concepts
   - 2.2. HDFS CLI Operations
3. **YARN**
4. **MapReduce**
   - 4.1. MR Streaming Interface
   - 4.2. mrJob

Appendix: MR Java Interface

Appendix: Running MapReduce 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`

Appendix: Running MapReduce 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/*
```