MapReduce
Simplified Data Processing on Large Clusters

Amir H. Payberah
amir@sics.se

Amirkabir University of Technology
(Tehran Polytechnic)
What do we do when there is too much data to process?
Scale Up vs. Scale Out (1/2)

- Scale up or scale vertically: adding resources to a single node in a system.
- Scale out or scale horizontally: adding more nodes to a system.
Scale up: more expensive than scaling out.

Scale out: more challenging for fault tolerance and software development.
Taxonomy of Parallel Architectures

Taxonomy of Parallel Architectures

A shared nothing architecture for processing large data sets with a parallel/distributed algorithm on clusters of commodity hardware.
Challenges

- How to distribute computation?
- How can we make it easy to write distributed programs?
- Machines failure.
Idea

- **Issue:**
  - Copying data over a network takes time.
Idea

- **Issue:**
  - Copying data over a network takes time.

- **Idea:**
  - Bring computation close to the data.
  - Store files multiple times for reliability.
Don’t worry about parallelization, fault tolerance, data distribution, and load balancing (MapReduce takes care of these).

Hide system-level details from programmers.
A programming model: to batch process large data sets (inspired by functional programming).
MapReduce Definition

- A **programming model**: to **batch** process large data sets (inspired by **functional programming**).

- An **execution framework**: to run parallel algorithms on **clusters of commodity hardware**.
Programming Model
Warm-up Task (1/2)

- We have a **huge text document**.

- **Count** the number of times each **distinct word** appears in the file

- **Application**: analyze web server logs to find popular URLs.
Warm-up Task (2/2)

- File too large for memory, but all \( \langle \text{word}, \text{count} \rangle \) pairs fit in memory.
Warm-up Task (2/2)

- File too large for memory, but all \(\langle \text{word}, \text{count} \rangle\) pairs fit in memory.

- \texttt{words(doc.txt) | sort | uniq -c}
  - where \texttt{words} takes a file and outputs the words in it, one per a line
Warm-up Task (2/2)

- File too large for memory, but all \(\langle \text{word}, \text{count} \rangle\) pairs fit in memory.

- \texttt{words(doc.txt) | sort | uniq -c}
  - where \texttt{words} takes a file and outputs the words in it, one per a line

- It captures the essence of \texttt{MapReduce}: great thing is that it is naturally \texttt{parallelizable}. 
MapReduce Overview

- words(doc.txt) | sort | uniq -c

- Sequentially read a lot of data.
- Map: extract something you care about.
- Group by key: sort and shuffle.
- Reduce: aggregate, summarize, filter or transform.
- Write the result.
MapReduce Overview

- `words(doc.txt) | sort | uniq -c`

- Sequentially **read** a lot of data.
MapReduce Overview

- `words(doc.txt) | sort | uniq -c`
- Sequentially *read* a lot of data.
- *Map:* extract something you care about.

![Diagram showing MapReduce process]

- Map
- Shuffle
- Reduce
MapReduce Overview

- `words(doc.txt) | sort | uniq -c`

- Sequentially read a lot of data.

- **Map:** extract something you care about.

- **Group by key:** sort and shuffle.
MapReduce Overview

- words(doc.txt) | sort | uniq -c

- Sequentially read a lot of data.

- Map: extract something you care about.

- Group by key: sort and shuffle.

- Reduce: aggregate, summarize, filter or transform.
MapReduce Overview

- `words(doc.txt) | sort | uniq -c`

- Sequentially read a lot of data.

- **Map**: extract something you care about.

- **Group by key**: sort and shuffle.

- **Reduce**: aggregate, summarize, filter or transform.

- Write the result.
MapReduce Dataflow

- **map** function: processes data and generates a set of intermediate key/value pairs.

- **reduce** function: merges all intermediate values associated with the same intermediate key.
Consider doing a word count of the following file using MapReduce:

Hello World Bye World
Hello Hadoop Goodbye Hadoop
The map function reads in words one a time and outputs \((\text{word}, 1)\) for each parsed input word.

The map function output is:

\[
\begin{align*}
\text{(Hello, 1)} \\
\text{(World, 1)} \\
\text{(Bye, 1)} \\
\text{(World, 1)} \\
\text{(Hello, 1)} \\
\text{(Hadoop, 1)} \\
\text{(Goodbye, 1)} \\
\text{(Hadoop, 1)}
\end{align*}
\]
The **shuffle** phase between **map** and **reduce** phase creates a list of values associated with each key.

The **reduce** function **input** is:

- (Bye, (1))
- (Goodbye, (1))
- (Hadoop, (1, 1))
- (Hello, (1, 1))
- (World, (1, 1))
Example: Word Count - reduce

- The **reduce** function sums the numbers in the list for each key and outputs \((\text{word}, \text{count})\) pairs.

- The output of the reduce function is the output of the MapReduce job:

  - (Bye, 1)
  - (Goodbye, 1)
  - (Hadoop, 2)
  - (Hello, 2)
  - (World, 2)
In some cases, there is significant repetition in the intermediate keys produced by each map task, and the reduce function is commutative and associative.

Machine 1:
(Hello, 1)
(World, 1)
(Bye, 1)
(World, 1)

Machine 2:
(Hello, 1)
(Hadoop, 1)
(Goodbye, 1)
(Hadoop, 1)
Users can specify an optional combiner function to merge partially data before it is sent over the network to the reduce function.

Typically the same code is used to implement both the combiner and the reduce function.

**Machine 1:**
- (Hello, 1)
- (World, 2)
- (Bye, 1)

**Machine 2:**
- (Hello, 1)
- (Hadoop, 2)
- (Goodbye, 1)
public static class MyMap extends Mapper<...> {
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(LongWritable key, Text value, Context context)
        throws IOException, InterruptedException {
        String line = value.toString();
        StringTokenizer tokenizer = new StringTokenizer(line);
        while (tokenizer.hasMoreTokens()) {
            word.set(tokenizer.nextToken());
            context.write(word, one);
        }
    }
}
public static class MyReduce extends Reducer<...> {
    public void reduce(Text key, Iterator<...> values, Context context) throws IOException, InterruptedException {
        int sum = 0;

        while (values.hasNext())
            sum += values.next().get();

        context.write(key, new IntWritable(sum));
    }
}
```java
public static void main(String[] args) throws Exception {
    Configuration conf = new Configuration();
    Job job = new Job(conf, "wordcount");
    
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    
    job.setMapperClass(MyMap.class);
    job.setCombinerClass(MyReduce.class);
    job.setReducerClass(MyReduce.class);
    
    job.setInputFormatClass(TextInputFormat.class);
    job.setOutputFormatClass(TextOutputFormat.class);
    
    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));
    
    job.waitForCompletion(true);
}
```
Example: Word Count - Compile and Run (1/2)

```bash
# start hdfs
> hadoop-daemon.sh start namenode
> hadoop-daemon.sh start datanode

# make the input folder in hdfs
> hdfs dfs -mkdir -p input

# copy input files from local filesystem into hdfs
> hdfs dfs -put file0 input/file0
> hdfs dfs -put file1 input/file1

> hdfs dfs -ls input/
input/file0
input/file1

> hdfs dfs -cat input/file0
Hello World Bye World

> hdfs dfs -cat input/file1
Hello Hadoop Goodbye Hadoop
```
Example: Word Count - Compile and Run (2/2)

```shell
> mkdir wordcount_classes

> javac -classpath
$HADOOP_HOME/share/hadoop/common/hadoop-common-2.2.0.jar:
$HADOOP_HOME/share/hadoop/mapreduce/hadoop-mapreduce-client-core-2.2.0.jar:
$HADOOP_HOME/share/hadoop/common/lib/commons-cli-1.2.jar
-d wordcount_classes sics/WordCount.java

> jar -cvf wordcount.jar -C wordcount_classes/ .

> hadoop jar wordcount.jar sics.WordCount input output

> hdfs dfs -ls output
output/part-00000

> hdfs dfs -cat output/part-00000
Bye 1
Goodbye 1
Hadoop 2
Hello 2
World 2
```
Execution Engine
The user program divides the input files into M splits.
- A typical size of a split is the size of a HDFS block (64 MB).
- Converts them to key/value pairs.

It starts up many copies of the program on a cluster of machines.

One of the copies of the program is master, and the rest are workers.

The master assigns works to the workers.

- It picks idle workers and assigns each one a map task or a reduce task.

A map worker reads the contents of the corresponding input splits.

It parses key/value pairs out of the input data and passes each pair to the user defined map function.

The intermediate key/value pairs produced by the map function are buffered in memory.

The buffered pairs are periodically written to local disk.
- They are partitioned into $R$ regions ($\text{hash(key)} \mod R$).

The locations of the buffered pairs on the local disk are passed back to the master.

The master forwards these locations to the reduce workers.

A reduce worker reads the buffered data from the local disks of the map workers.

When a reduce worker has read all intermediate data, it sorts it by the intermediate keys.

The reduce worker iterates over the **intermediate data**.

For each **unique intermediate key**, it passes the key and the corresponding set of intermediate values to the **user defined reduce function**.

The output of the reduce function is appended to a **final output file** for this reduce partition.
When all map tasks and reduce tasks have been completed, the master wakes up the user program.

Fault Tolerance - Worker

- Detect failure via **periodic heartbeats**.

- Re-execute **in-progress map** and **reduce** tasks.

- Re-execute **completed map** tasks: their output is stored on the local disk of the failed machine and is therefore inaccessible.

- **Completed reduce** tasks do not need to be re-executed since their output is stored in a global filesystem.
State is periodically checkpointed: a new copy of master starts from the last checkpoint state.
Summary

- Programming model: Map and Reduce
- Execution framework
- Batch processing
- Shared nothing architecture
References:

Questions?