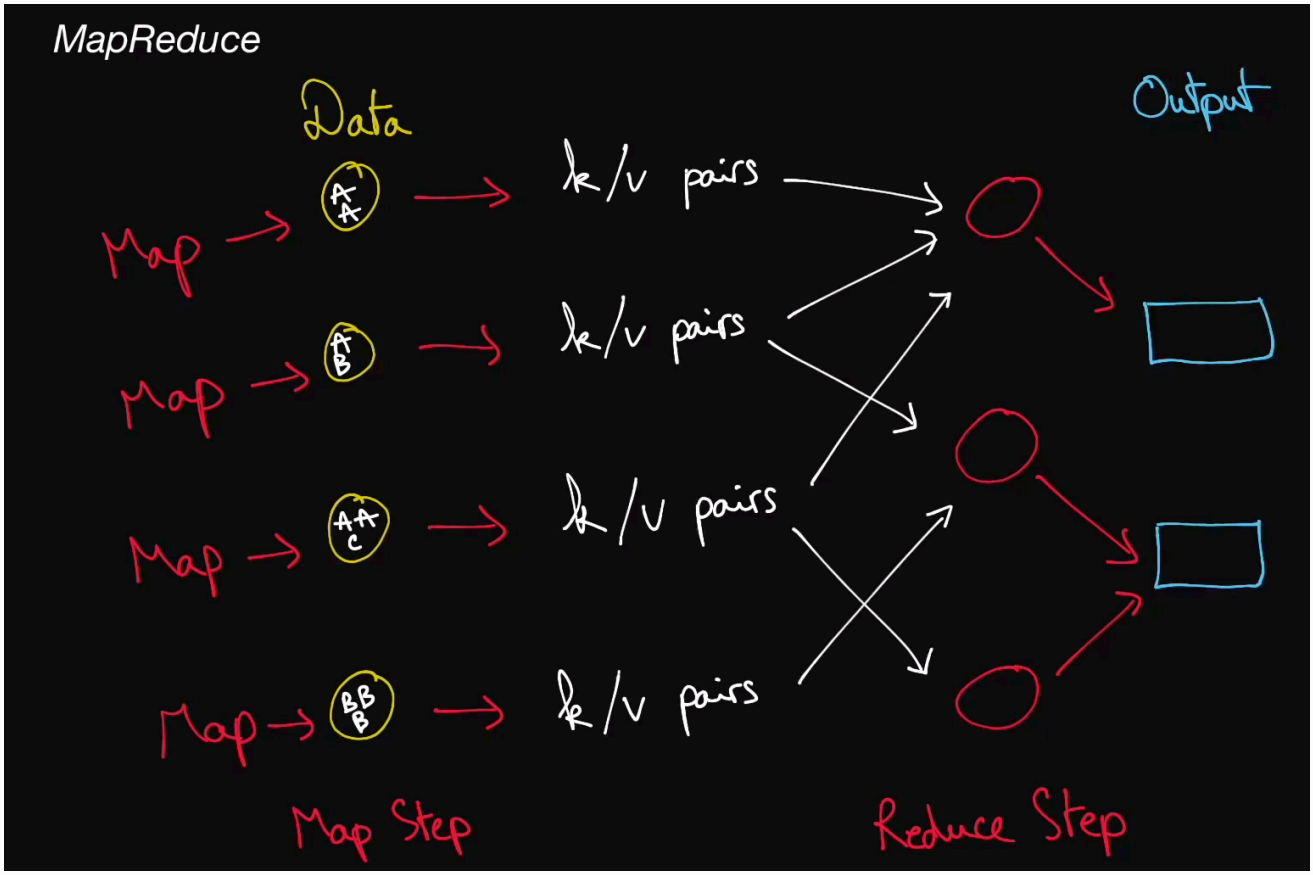
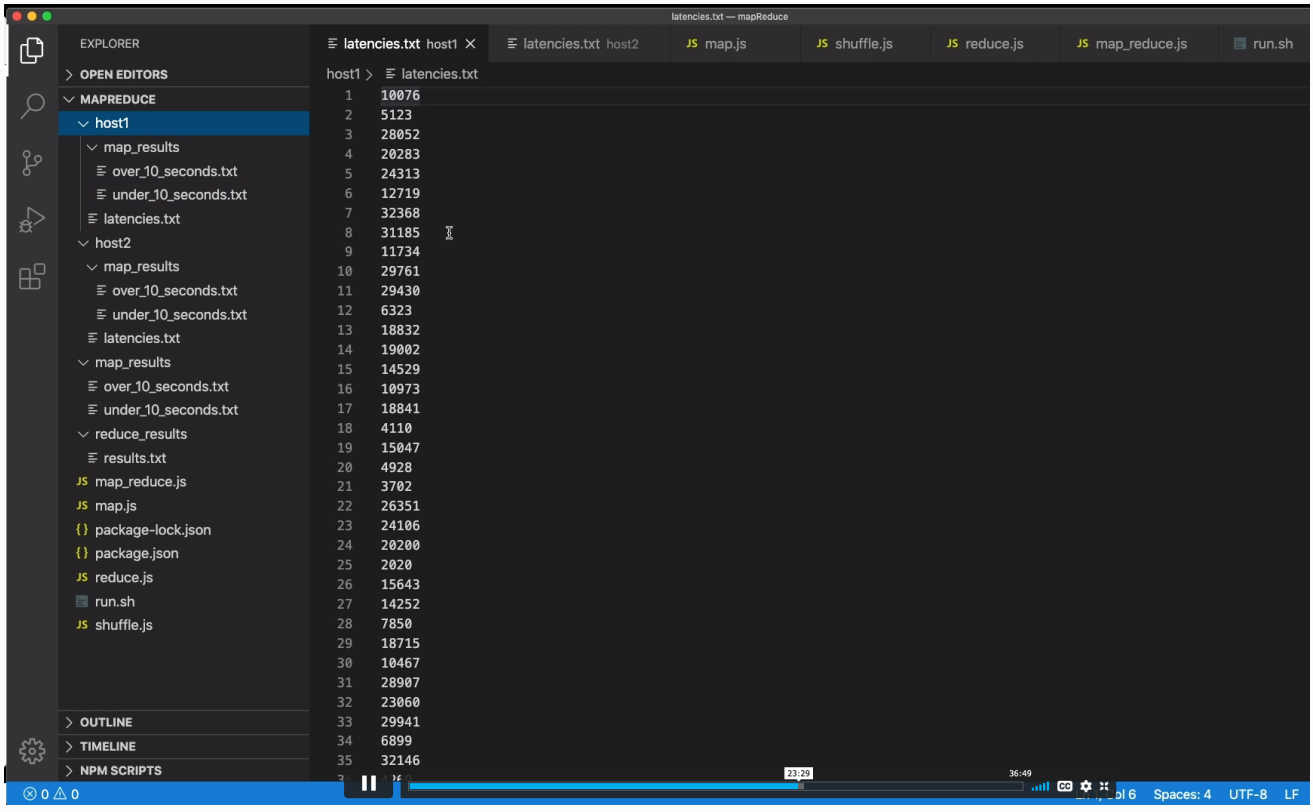
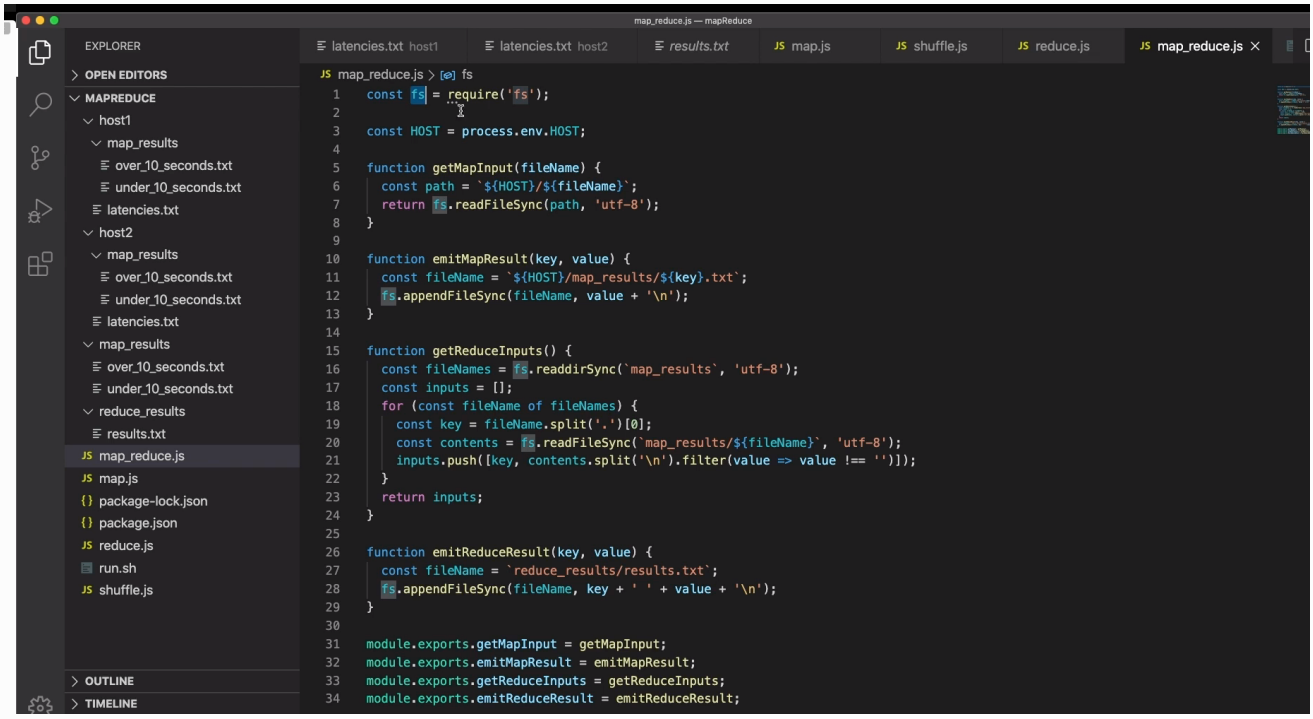
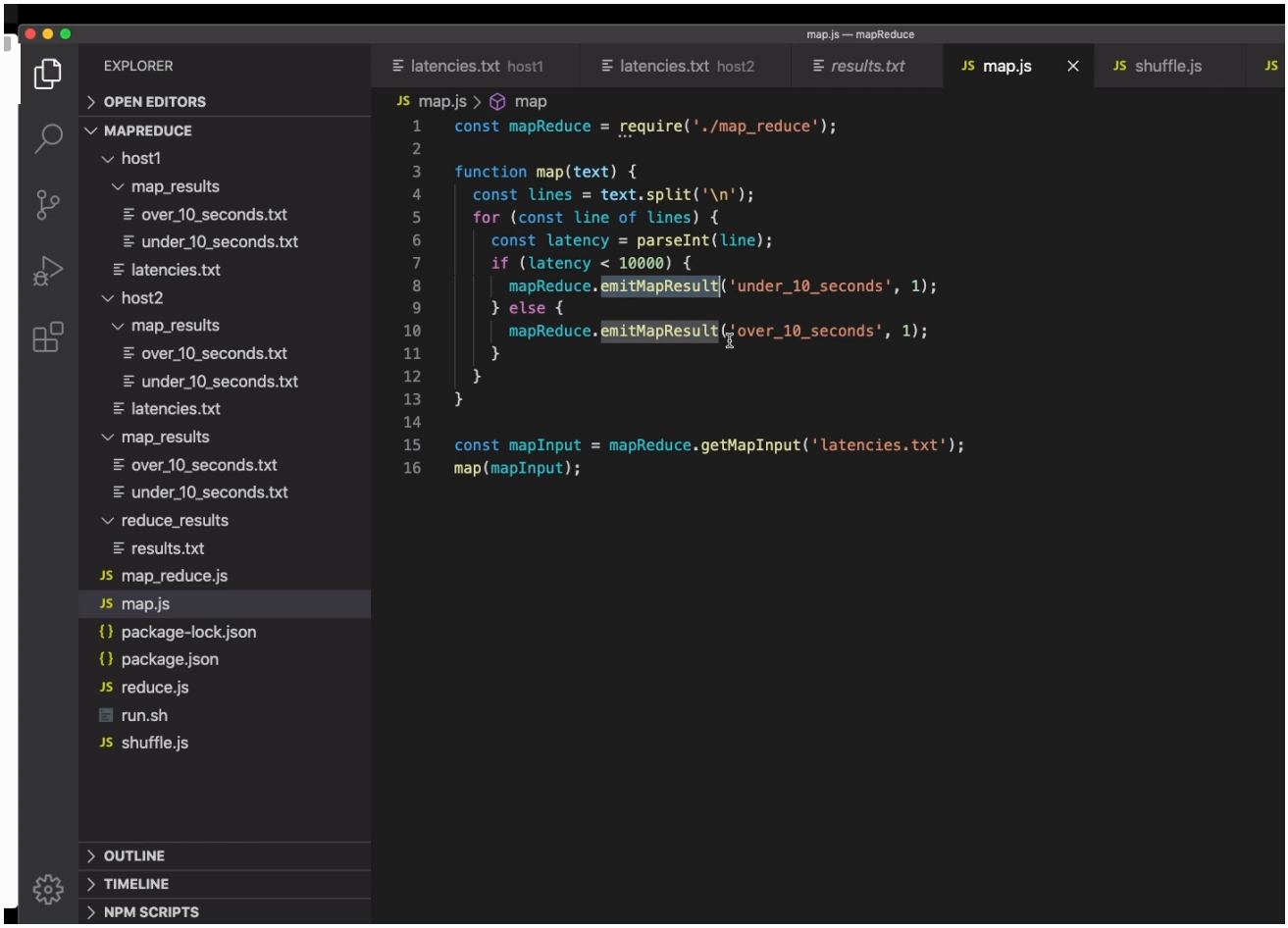
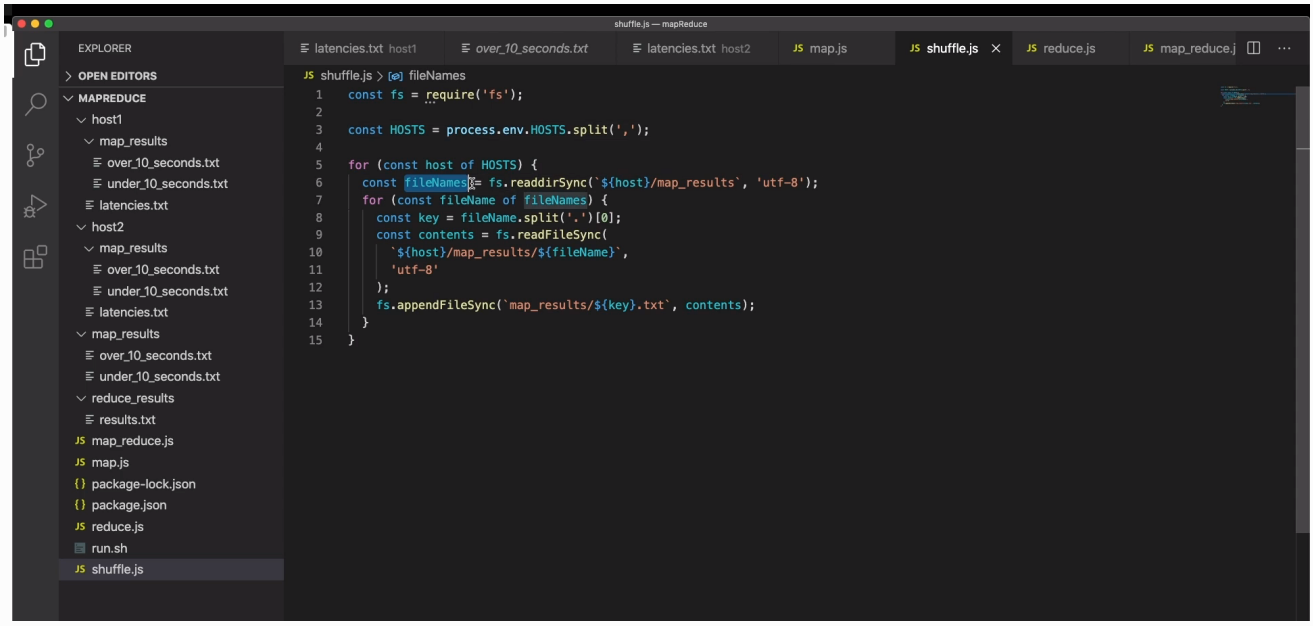
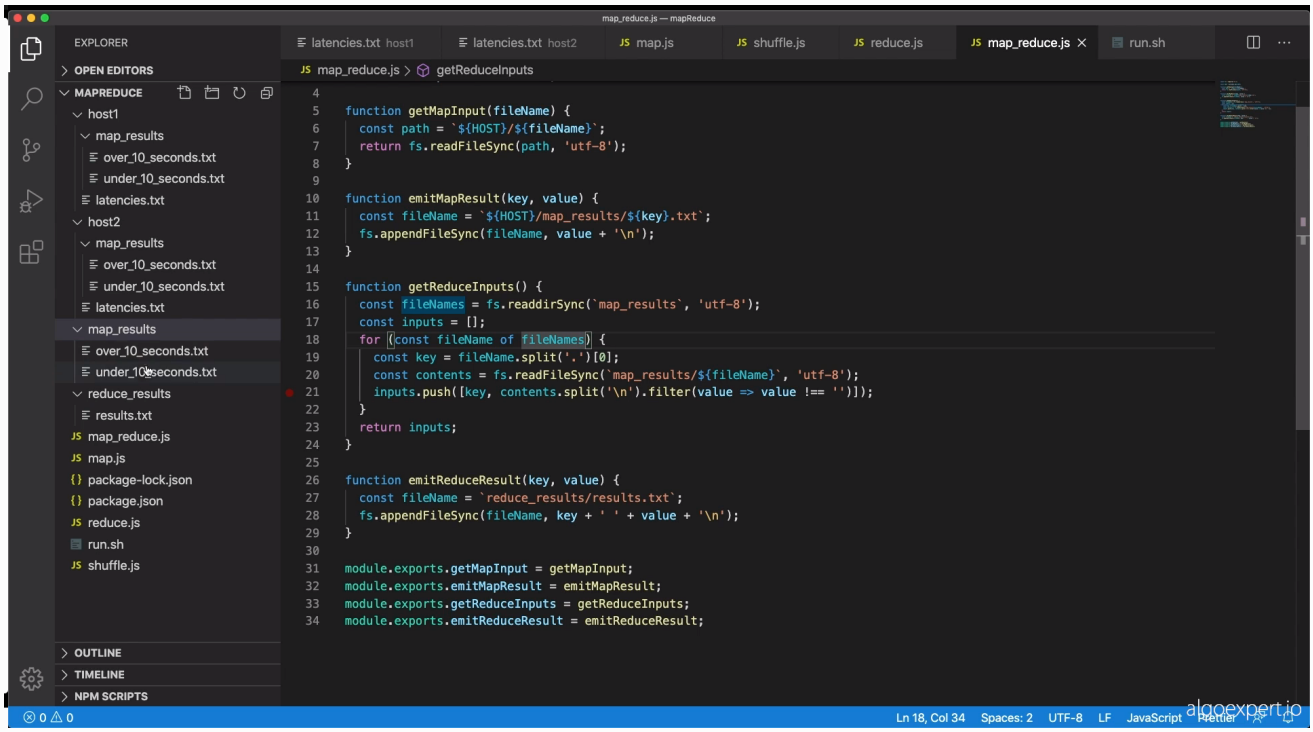

MapReduce
Data
Output
Map
Map
Map
Map
k/v pairs
k/v pairs
k/v pairs
k/v pairs
Map Step
Reduce Step

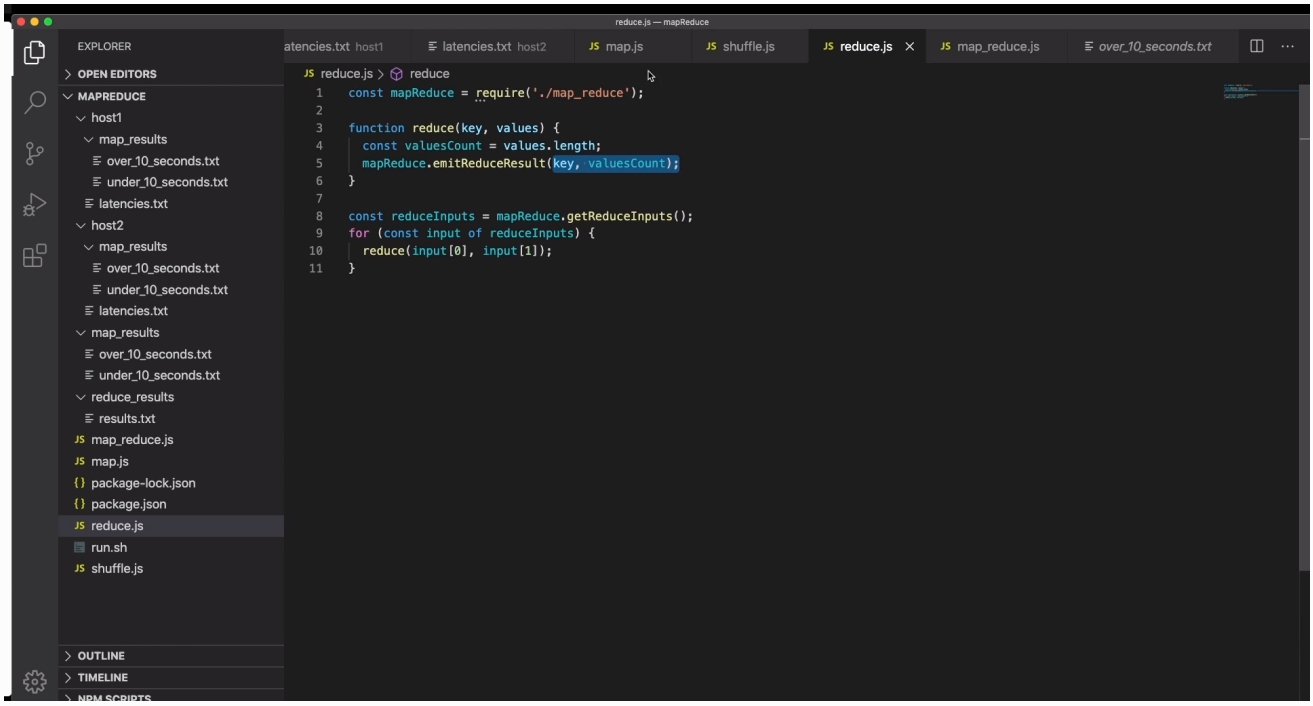

latencies.txt — mapReduce
EXPLORER
OPEN EDITORS
MAPREDUCE
host1
map_results
over_10_seconds.txt
under_10_seconds.txt
latencies.txt
host2
map_results
over_10_seconds.txt
under_10_seconds.txt
latencies.txt
map_results
over_10_seconds.txt
under_10_seconds.txt
reduce_results
results.txt
map_reduce.js
map.js
package-lock.json
package.json
reduce.js
run.sh
shuffle.js
OUTLINE
TIMELINE
NPM SCRIPTS
latencies.txt host1
latencies.txt host2
map.js
shuffle.js
reduce.js
map_reduce.js
run.sh
host1 > latencies.txt
10076
5123
28052
20283
24313
12719
32368
31185
11734
29761
29430
6323
18832
19002
14529
10973
18841
4110
15047
4928
3702
26351
24106
20200
2020
15643
14252
7850
18715
10467
28907
23060
29941
6899
32146
23:29
36:49
Spaces: 4
UTF-8
LF

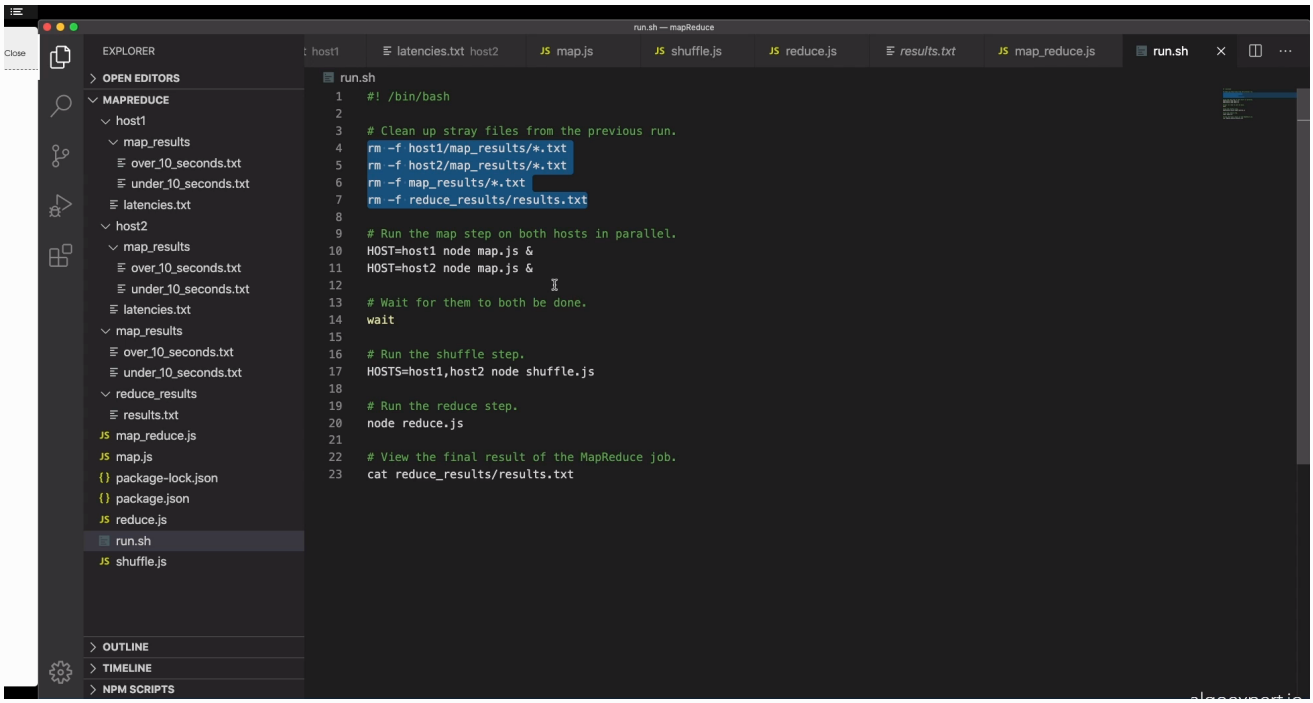

map_reduce.js — mapReduce
EXPLORER
latencies.txt host1
latencies.txt host2
results.txt
map.js
shuffle.js
reduce.js
map_reduce.js
OPEN EDITORS
MAPREDUCE
host1
map_results
over_10_seconds.txt
under_10_seconds.txt
latencies.txt
host2
map_results
over_10_seconds.txt
under_10_seconds.txt
latencies.txt
map_results
over_10_seconds.txt
under_10_seconds.txt
reduce_results
results.txt
map_reduce.js
map.js
package-lock.json
package.json
reduce.js
run.sh
shuffle.js
OUTLINE
TIMELINE
map_reduce.js > fs
const fs = require('fs');
const HOST = process.env.HOST;
function getMapInput(fileName) {
const path = `${HOST}/${fileName}`;
return fs.readFileSync(path, 'utf-8');
}
function emitMapResult(key, value) {
const fileName = `${HOST}/map_results/${key}.txt`;
fs.appendFileSync(fileName, value + '\n');
}
function getReduceInputs() {
const fileNames = fs.readdirSync(`map_results`, 'utf-8');
const inputs = [];
for (const fileName of fileNames) {
const key = fileName.split('.')[0];
const contents = fs.readFileSync(`map_results/${fileName}`, 'utf-8');
inputs.push([key, contents.split('\n').filter(value => value !== '')]);
}
return inputs;
}
function emitReduceResult(key, value) {
const fileName = `reduce_results/results.txt`;
fs.appendFileSync(fileName, key + ' ' + value + '\n');
}
module.exports.getMapInput = getMapInput;
module.exports.emitMapResult = emitMapResult;
module.exports.getReduceInputs = getReduceInputs;
module.exports.emitReduceResult = emitReduceResult;


map.js — mapReduce
EXPLORER
latencies.txt host1
latencies.txt host2
results.txt
map.js
shuffle.js
OPEN EDITORS
MAPREDUCE
host1
map_results
over_10_seconds.txt
under_10_seconds.txt
latencies.txt
host2
map_results
over_10_seconds.txt
under_10_seconds.txt
latencies.txt
map_results
over_10_seconds.txt
under_10_seconds.txt
reduce_results
results.txt
map_reduce.js
map.js
package-lock.json
package.json
reduce.js
run.sh
shuffle.js
OUTLINE
TIMELINE
NPM SCRIPTS
map.js > map
const mapReduce = require('./map_reduce');
function map(text) {
const lines = text.split('\n');
for (const line of lines) {
const latency = parseInt(line);
if (latency < 10000) {
mapReduce.emitMapResult('under_10_seconds', 1);
} else {
mapReduce.emitMapResult('over_10_seconds', 1);
}
}
}
const mapInput = mapReduce.getMapInput('latencies.txt');
map(mapInput);






reduce.js — mapReduce
EXPLORER
OPEN EDITORS
MAPREDUCE
host1
map_results
over_10_seconds.txt
under_10_seconds.txt
latencies.txt
host2
map_results
over_10_seconds.txt
under_10_seconds.txt
latencies.txt
map_results
over_10_seconds.txt
under_10_seconds.txt
reduce_results
results.txt
map_reduce.js
map.js
package-lock.json
package.json
reduce.js
run.sh
shuffle.js
OUTLINE
TIMELINE
NPM SCRIPTS
latencies.txt host1
latencies.txt host2
map.js
shuffle.js
reduce.js
map_reduce.js
over_10_seconds.txt
reduce.js > reduce
1 const mapReduce = require('./map_reduce');
2
3 function reduce(key, values) {
4 const valuesCount = values.length;
5 mapReduce.emitReduceResult(key, valuesCount);
6 }
7
8 const reduceInputs = mapReduce.getReduceInputs();
9 for (const input of reduceInputs) {
10 reduce(input[0], input[1]);
11 }


run.sh — mapReduce
Close
EXPLORER
OPEN EDITORS
MAPREDUCE
host1
map_results
over_10_seconds.txt
under_10_seconds.txt
latencies.txt
host2
map_results
over_10_seconds.txt
under_10_seconds.txt
latencies.txt
map_results
over_10_seconds.txt
under_10_seconds.txt
reduce_results
results.txt
map_reduce.js
map.js
package-lock.json
package.json
reduce.js
run.sh
shuffle.js
OUTLINE
TIMELINE
NPM SCRIPTS
host1
latencies.txt host2
map.js
shuffle.js
reduce.js
results.txt
map_reduce.js
run.sh
run.sh
1 #! /bin/bash
2
3 # Clean up stray files from the previous run.
4 rm -f host1/map_results/*.txt
5 rm -f host2/map_results/*.txt
6 rm -f map_results/*.txt
7 rm -f reduce_results/results.txt
8
9 # Run the map step on both hosts in parallel.
10 HOST=host1 node map.js &
11 HOST=host2 node map.js &
12
13 # Wait for them to both be done.
14 wait
15
16 # Run the shuffle step.
17 HOSTS=host1,host2 node shuffle.js
18
19 # Run the reduce step.
20 node reduce.js
21
22 # View the final result of the MapReduce job.
23 cat reduce_results/results.txt

2 Prerequisites

File System

An abstraction over a storage medium that defines how to manage data. While there exist many different types of file systems, most follow a hierarchical structure that consists of directories and files, like the **Unix file system**'s structure.

Idempotent Operation

An operation that has the same ultimate outcome regardless of how many times it's performed. If an operation can be performed multiple times without changing its overall effect, it's idempotent. Operations performed through a **Pub/Sub** messaging system typically have to be idempotent, since Pub/Sub systems tend to allow the same messages to be consumed multiple times.

For example, increasing an integer value in a database is *not* an idempotent operation, since repeating this operation will not have the same effect as if it had been performed only once. Conversely, setting a value to "COMPLETE" *is* an idempotent operation, since repeating this operation will always yield the same result: the value will be "COMPLETE".

3 Key Terms

MapReduce

A popular framework for processing very large datasets in a distributed setting efficiently, quickly, and in a fault-tolerant manner. A MapReduce job is comprised of 3 main steps:

- the **Map** step, which runs a **map function** on the various chunks of the dataset and transforms these chunks into intermediate **key-value pairs**.
- the **Shuffle** step, which reorganizes the intermediate **key-value pairs** such that pairs of the same key are routed to the same machine in the final step.
- the **Reduce** step, which runs a **reduce function** on the newly shuffled **key-value pairs** and transforms them into more meaningful data.

The canonical example of a MapReduce use case is counting the number of occurrences of words in a large text file.

When dealing with a MapReduce library, engineers and/or systems administrators only need to worry about the map and reduce functions, as well as their inputs and outputs. All other concerns, including the parallelization of tasks and the fault-tolerance of the MapReduce job, are abstracted away and taken care of by the MapReduce implementation.

Distributed File System

A Distributed File System is an abstraction over a (usually large) cluster of machines that allows them to act like one large file system. The two most popular implementations of a DFS are the **Google File System** (GFS) and the **Hadoop Distributed File System** (HDFS).

Typically, DFSs take care of the classic **availability** and **replication** guarantees that can be tricky to obtain in a distributed-system setting. The overarching idea is that files are split into chunks of a certain size (4MB or 64MB, for instance), and those chunks are sharded across a large cluster of machines. A central control plane is in charge of deciding where each chunk resides, routing reads to the right nodes, and handling communication between machines.

Different DFS implementations have slightly different APIs and semantics, but they achieve the same common goal: extremely large-scale persistent storage.

Hadoop 

A popular, open-source framework that supports MapReduce jobs and many other kinds of data-processing pipelines. Its central component is **HDFS** (Hadoop Distributed File System), on top of which other technologies have been developed.