MapReduce: Evaluation

Let's see how well the MapReduce design meets its requirements.

We'll cover the following

Evaluation

The use cases of our system span various problems, ranging from processing raw data to gaining insights from the derived data, but we can evaluate our design to fulfill the requirements for general use.

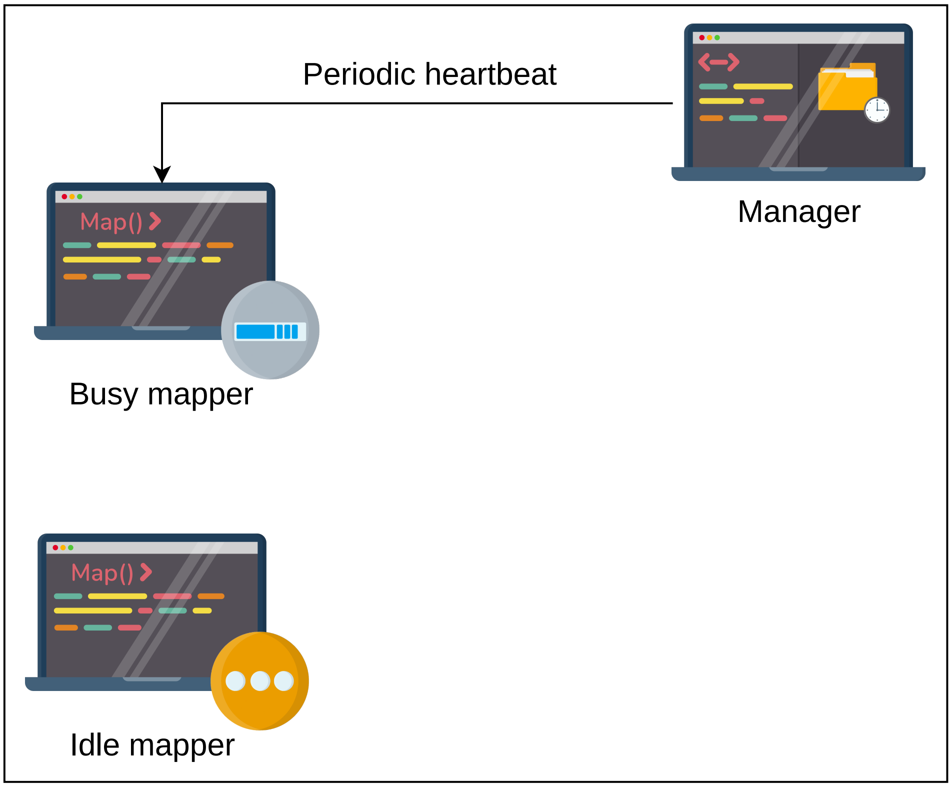
The `MapReduce` library inherently handles the mechanisms of automatic parallelization, data splitting, load balancing, and fault tolerance. Let’s see how our design fulfills additional non-functional requirements.

Fault tolerance

Since we deal with large datasets, fault tolerance is critical. Faults can happen at any stage or component. Let's see each one in detail.

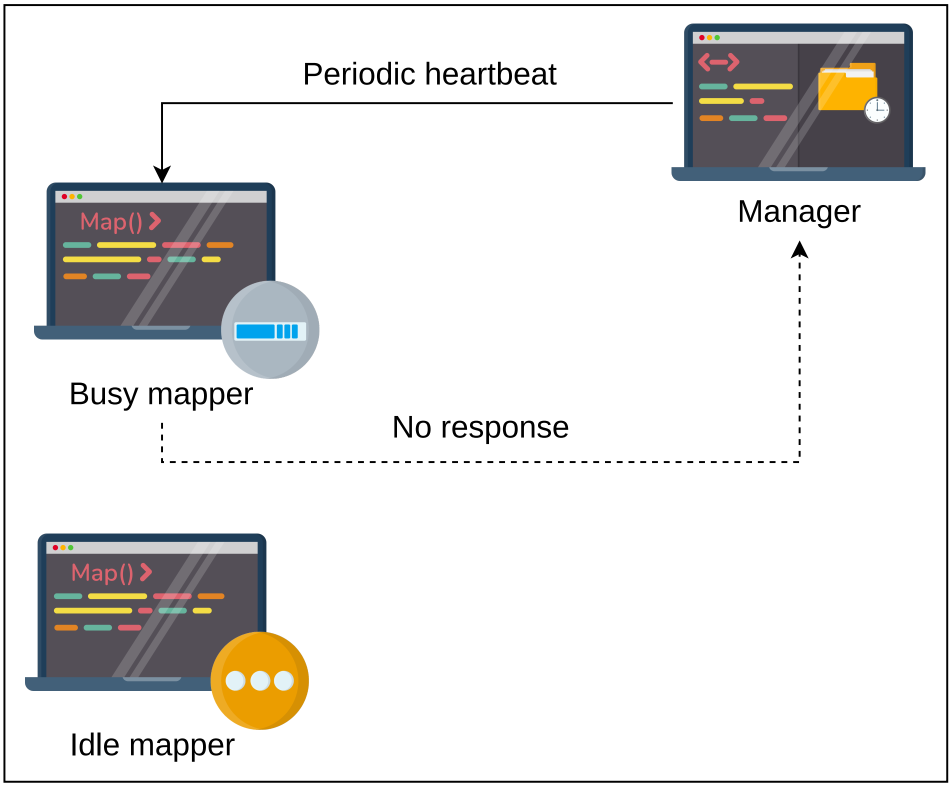
1. **Worker failure:** Let's consider the scenario of a failed worker. The manager identifies any worker not responding to the periodic calls as a failure. Once the manager declares a worker as a failure, it reschedules all of its completed and in-progress tasks to another available worker (if the actual disk or server fails and is unreachable, and `Reduce` tasks haven't fetched out completed map outputs yet, then the manager will need to get them rescheduled. If only the worker process fails, the manager will only need to reschedule the in-progress work). When the reassigned work is completed, the manager notifies all the reducers working on the processed data of the failed mapper and assigns them the new mapper address to fetch data. The manager also wipes down all the processed data by the failed mapper from its local disk to free up space.

   In case of a reducer failure, the manager reassigns its failed tasks to another reducer and provides the new reducer with all the required information for data fetching.



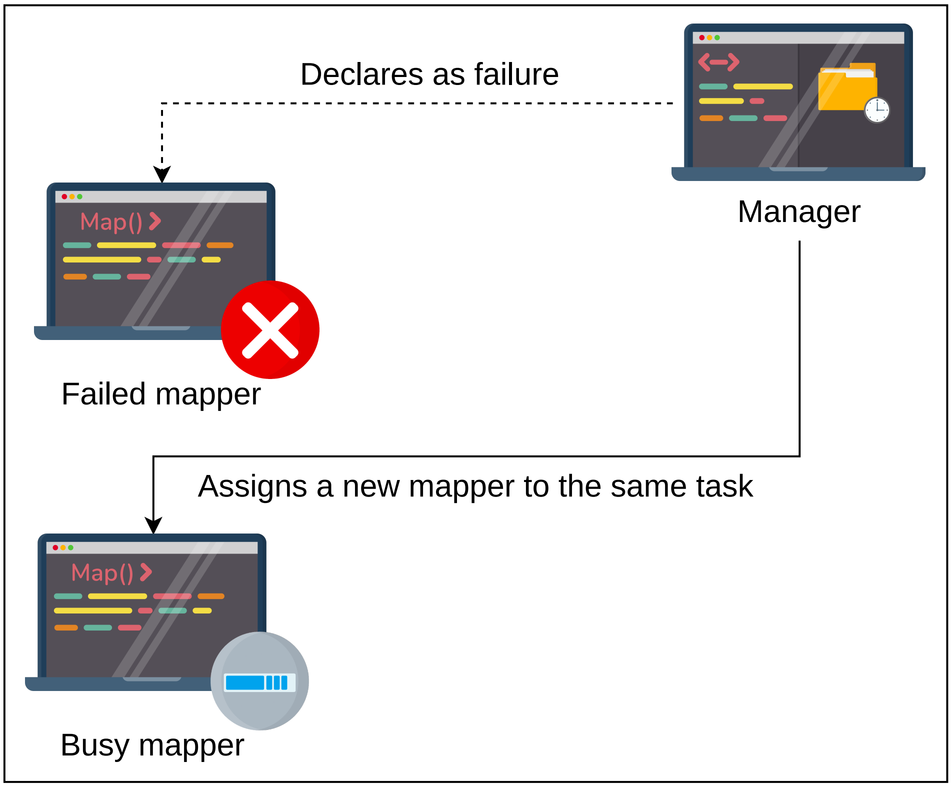
Manager periodically pings a busy worker (doing map or reduce tasks)

1 of 3



If there is no response after few retries, the manager declares it as a failed attempt

2 of 3



The manager assigns the same task to a new mapper

3 of 3

2. **Manager failure:** Remember that when a MapReduce job starts, one worker takes on the role of a manager. Each user-spawned job will have its own manager. Therefore, the failure of a manager will only have a limited impact (on a specific user job). We should note that many MapReduce jobs are prolonged (for example, processing a crawl of the WWW), and manager failure can impact them badly. The `MapReduce` library does not deal with the manager's failure and leaves it to the end users.

   Our current implementation stops the MapReduce job once the manager fails. However, we can implement a fail-safe option by making the manager save its snapshots periodically and revert to the latest one in case of a failure (thereby reducing the impact of such failures).

3. **Bad records:** As discussed earlier, our design handles bad records by skipping them from the re-executions.

By handling all these situations, our system ensures fault tolerance. Let’s go through the semantics in case of failures.

Semantics in case of failures

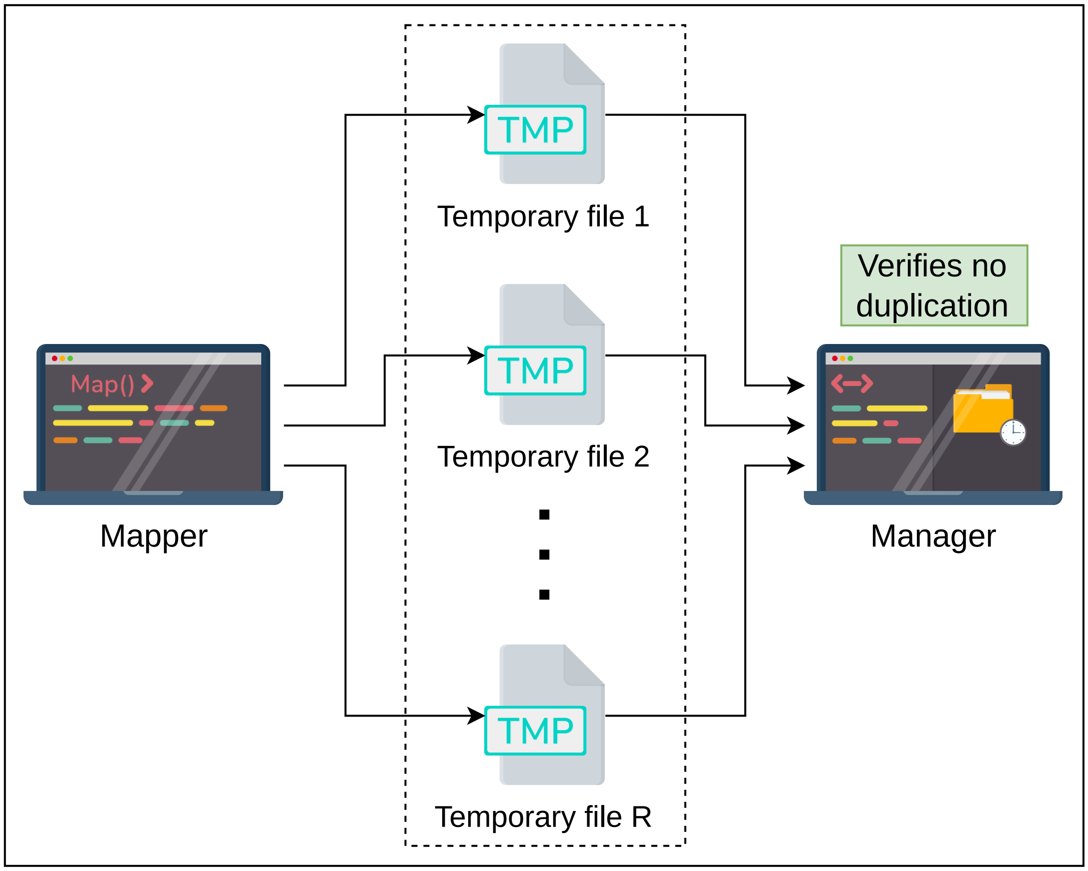
We can further divide these semantics based on the deterministic or non-deterministic nature of the `Map` or `Reduce` functions.

Deterministic functions

In the case of the deterministic nature of the user-defined `Map` and `Reduce` functions, the output of our distributed implementation is similar to their non-faulting sequential execution. It provides a strong baseline for the users to predict and understand the behavior of their program.

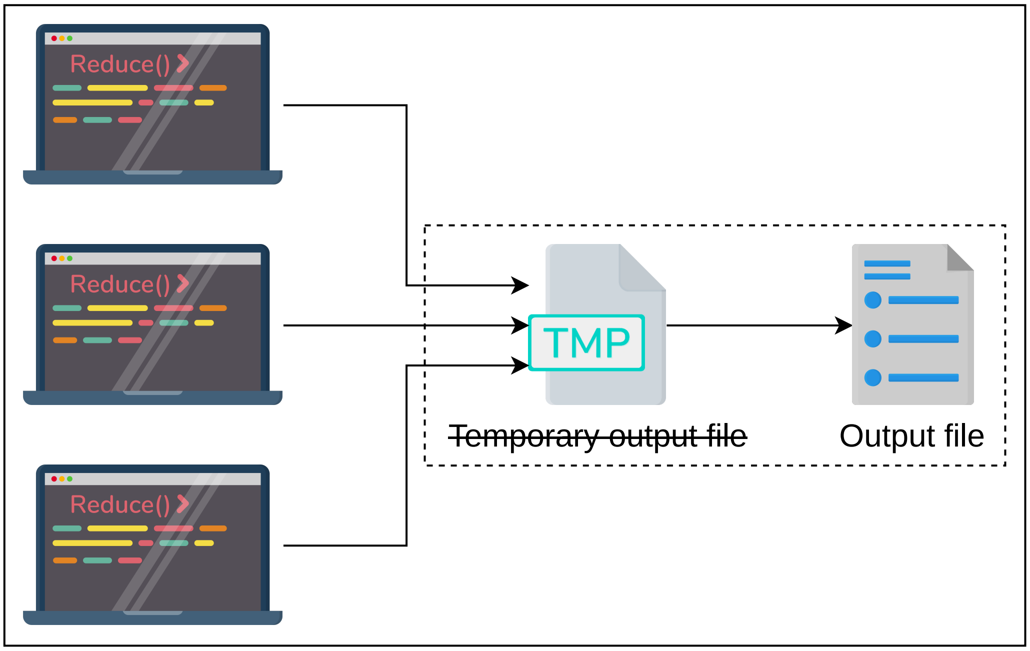
The program achieves this property by relying on the atomic nature of the commits made by the `Map` or `Reduce` tasks. Both these tasks write their outputs to temporary files and GFS and change the names once entirely generated.

1. A `Map` task writes its output to R temporary files, which is equivalent to the number of `Reduce` tasks. Once completed, the `Map` task sends the names of these R temporary files to the manager. After verifying that there is no duplication in these files, the manager confirms the completion of the task and saves the names of these R files into a manager data structure.



A Map() task on completion. A mapper only sends the local file paths to the manager

2. Unlike the `Map` task, a `Reduce` task yields one output file. Similar to the `Map` task, it writes its output to a temporary file in GFS and atomically renames it to an output file after the task's completion. Multiple workers performing the same `Reduce` task results in multiple rename calls for the same output file. GFS's atomic rename operation ensures that the final file system contains the output file of just one execution of the `Reduce` task, excluding the possibility of repetition. Here, we'll use GFS's metadata atomicity.

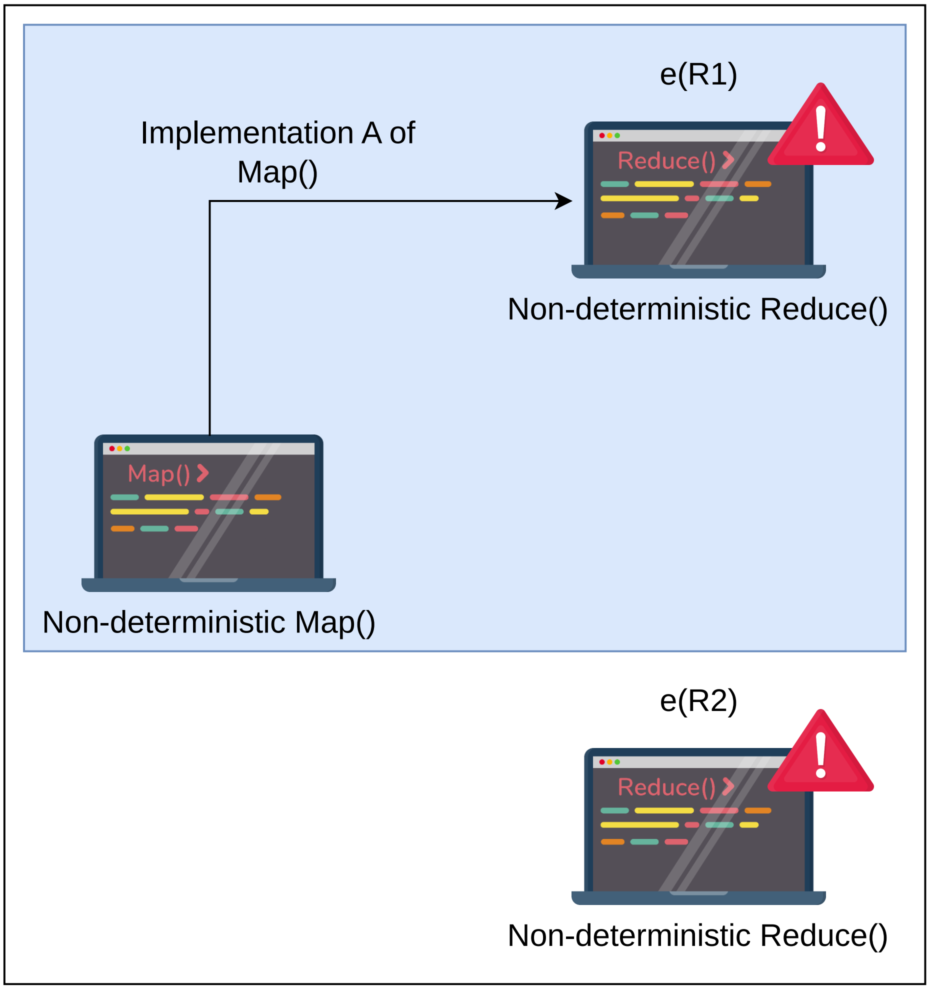


A Reduce() task on completion. The files are on the GFS

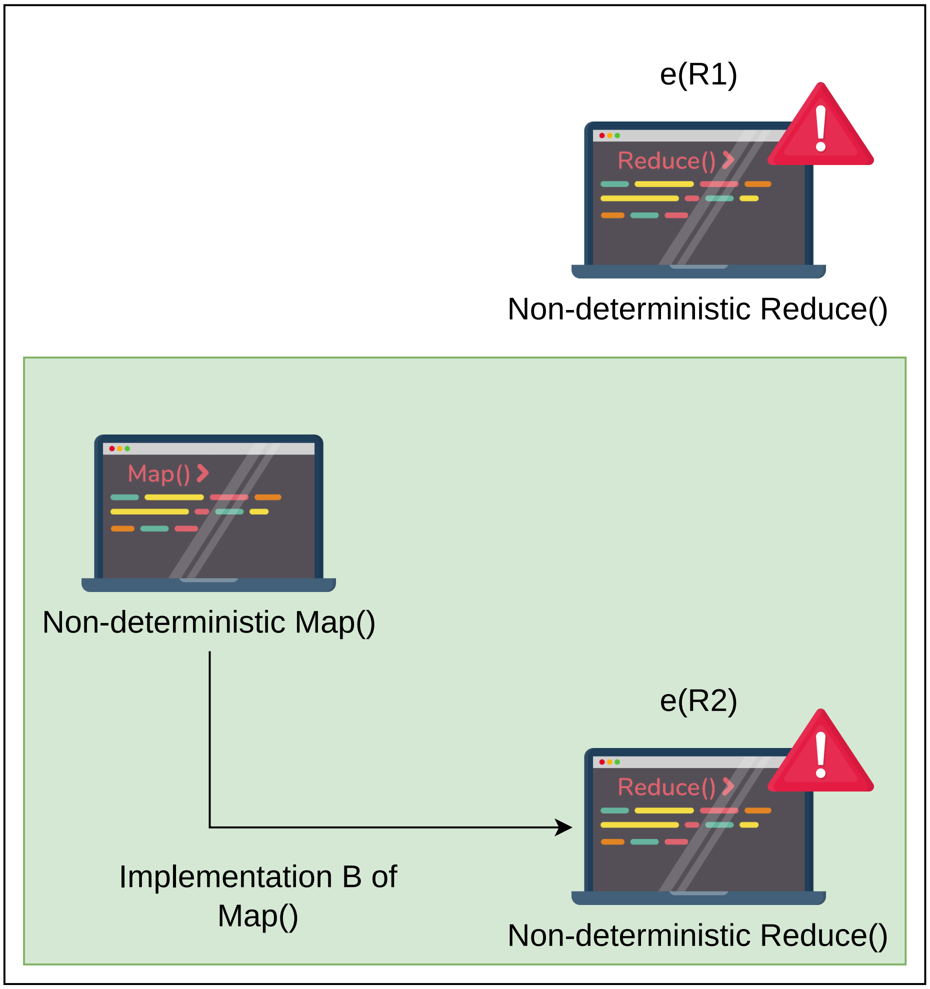
Non-deterministic functions

In the case of the non-deterministic `Map` and `Reduce` functions, there is no direct correlation between a distributed execution and a single sequential execution, which results in weaker semantics for failures.

Consider a `Map` task M and two `Reduce` tasks $R1$ and $R2$. Let $e(R_i)$ be the execution of a `Reduce` task that is committed. The weaker semantics arise because $e(R_1)$ might have fetched the output of a different sequential implementation of M than $e(R_2)$. The programmers will need to deal with these cases.


e(R1)
Implementation A of Map()
Reduce()
Non-deterministic Reduce()
Map()
Non-deterministic Map()
e(R2)
Reduce()
Non-deterministic Reduce()

e(R1) fetching data from an implementation of M


e(R1)
Reduce()
Non-deterministic Reduce()
Map()
Non-deterministic Map()
e(R2)
Reduce()
Implementation B of Map()
Non-deterministic Reduce()

e(R2) fetching data from a different implementation of M

2 of 2

Throughput

Our system ensures high throughput by engaging the maximum number of available workers and dynamic load balancing among them.

Latency

Our system uses GFS for read/write operations, which performs replication during the writing tasks and can increase latency. However, even with the GFS, we're using its locality feature, and many reads are local to the server. MapReduce is primarily a high throughput system, and it can trade off latency for that.

Scalability

The MapReduce system is highly scalable; we can successfully add or remove new workers without hassle. The manager automatically registers newly added workers and utilizes them on a need basis. For example, if we add more resources during the `Map` task phase, the manager can easily utilize them to assign the pending tasks out of M tasks. The same happens if we add more resources during the `Reduce` task phase; the manager uses them to assign the pending **R** tasks. Therefore, our system is horizontally scalable for the workers.

Availability and reliability

The system utilizes idle workers in case of worker failures and relies on GFS for safe data storage to ensure availability and reliability. MapReduce's use of local disks and replicated GFS enables it to provide a simple and robust reliability mechanism. However, this simple reliability comes at the cost of higher latency in terms of IO operations.

Lesser code complexity

The user achieves the results without programming the distribution and parallelization codes. Moreover, the system allows users to fine-tune the related internal parameters without in-depth programming knowledge.

> **Note:** The simple MapReduce model, which mimics serial programming in many ways at the user level, is a major reason for its broad acceptability across the spectrum of programmers.

Performance analysis

Let's assume a sorting job where the program has to sort 10^{10} 100-byte records, approximately one terabyte of data. With 64 MB splits of the input data, we have the values of M and R as 15,000 and 4000, respectively. We'll deploy around 1800 machines for this MapReduce operation, making each mapper perform approximately 9 `Map` tasks and each reducer approximately 3 `Reduce` tasks.

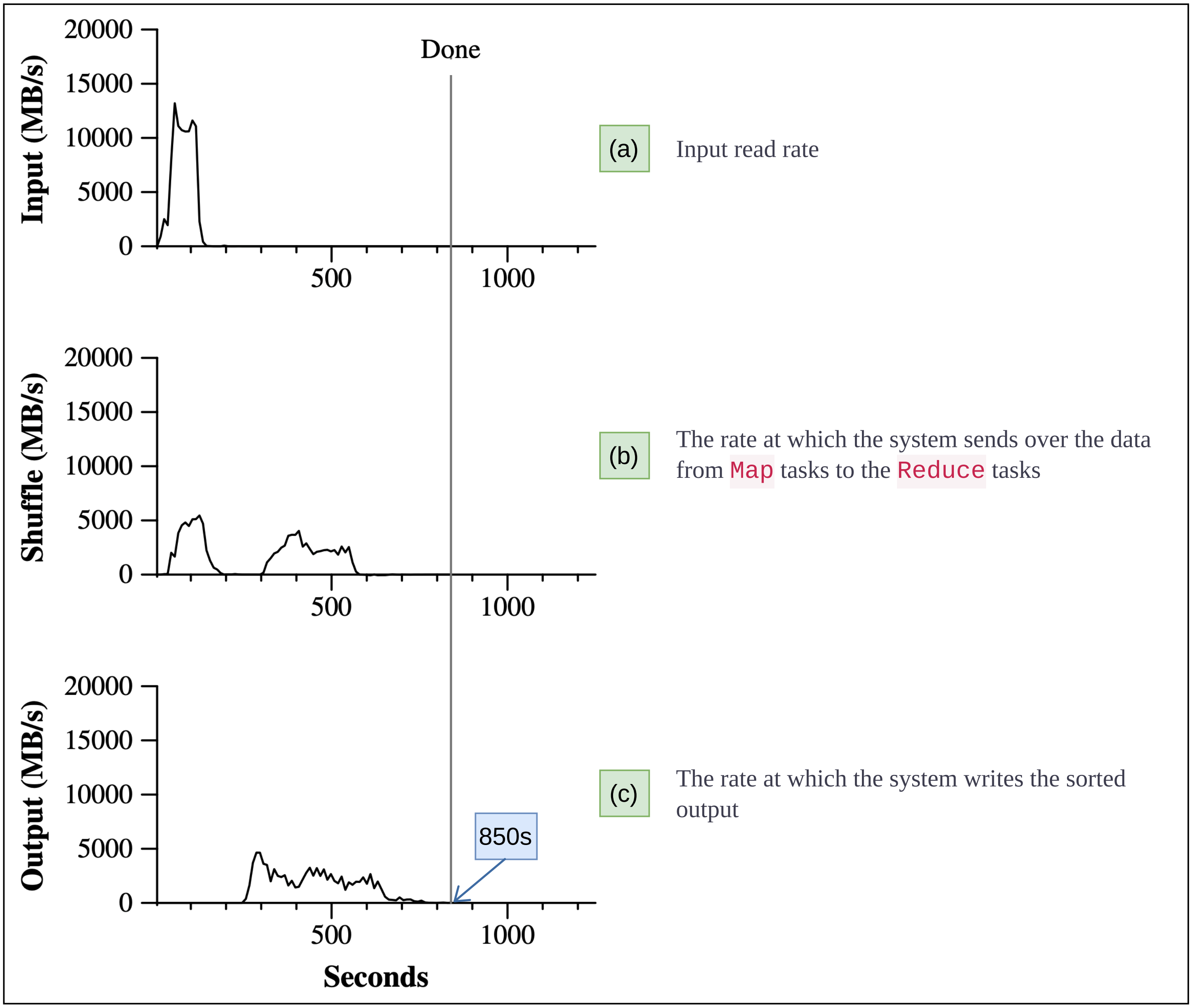
There are less than 50 lines of code in the user program. It contains a `Map` function (three lines) and a `Reduce` function (a built-in `Identity` function). The `Map` function extracts a sorting key (10-byte) from a line of text and outputs it with the original text line as the intermediate key-value pair. The `Reduce` function passes the intermediate key-value pair unchanged as the output key-value pair. The program writes the final sorted output to a set of two-way replicated GFS files, which is 2 terabytes of output.

We can analyze the performance of our system on three different configurations for this sorting program:

1. Normal execution
2. No backup tasks
3. Failures/terminated tasks

Normal execution

Let’s show the behavior of our program for this configuration with the help of a graph.



(a) Input read rate

(b) The rate at which the system sends over the data from `Map` tasks to the `Reduce` tasks

(c) The rate at which the system writes the sorted output

Normal execution (Source: Dean, Jeffrey, and Sanjay Ghemawat. “MapReduce: Simplified Data Processing on Large Clusters.” Communications of the ACM 51, no. 1 (2008): 107–13. https://doi.org/10.1145/1327452.1327492.)

- Part (a) of the graph shows the input read rate. The rate peaks after the initialization and dies out quickly as all the `Map` tasks finish.
- Part (b) of the graph shows the rate at which the system sends over the data from `Map` tasks to the `Reduce` tasks. The shuffling starts after the system completes its first batch of `Map` tasks. The two humps in the graph represent two batches of the shuffled data sent over the network, followed by one another. The shuffling for the second batch starts right after a reducer completes its assigned task.
- Part (c) shows the rate at which the system writes the sorted, processed data into the output files. The workers sort the intermediate data between the end

of the first shuffling period and the start of the output period, hence the delay between them.

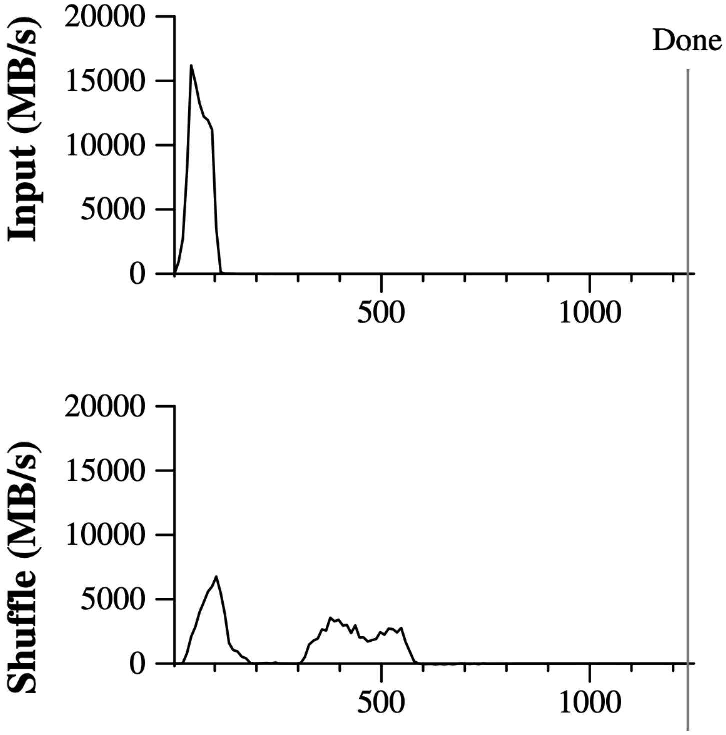
Deductions

We can deduce some interesting observations by comparing the above three parts of the graph.

1. The input rate is higher than the shuffling and output rates because of our locality optimization for the `Map` tasks.
2. The shuffling rate is greater than the writing rate as GFS replicates the output to ensure availability and reliability.

No backup tasks

Our normal execution includes implementing the backup tasks for stragglers. Let’s see a graph for the case where we disable this functionality.



(a) Input read rate

(b) The rate at which the system sends over the data from `Map` tasks to the `Reduce` tasks