MapReduce implementation in TypeScript
Scalable and Distributed Computing Project

Francesco Lorenzoni

September 2025

Contents

Chapter 1

Project Report

This is the report of the project developed for the "Scalable and Distributed Computing" course held at UniPi during the academic year 2024/25 by Prof. Dazzi. The developed project consists of an implementation in TypeScript of the MAPREDUCE framework, exploiting Kafka for message delivery and Docker for deployment. The purpose was *not* to develop a highly performant or resilient implementation of MAPREDUCE; we have discussed other implementations, such as Hadoop and Spark, which work well and certainly would not be outperformed by a small project developed by a university student ☺.

Instead, the intent was to define a problem whose resolution would lead to facing, from a practical point of view, some of the issues and concepts —highlighted during lectures— which arise when dealing with distributed applications.

This report will initially describe the main features and architecture of the implementation, and later discuss how it works under the hood and how it relates to the concepts discussed in the course.

The course topics this project touches are:

⬦ MAPREDUCE
The project is a simple implementation of the MAPREDUCE framework.

⬦ *Kafka and Distributed Messaging*
Kafka was chosen as the tool to share information among nodes, and to provide the messaging infrastructure. Kafka's design guided the messaging architecture, trying to exploit its *publish-subscribe* and features at their best.

⬦ *Synchronizing*
Clearly some synchronization has to occur in a MAPREDUCE framework. Making the nodes aware of when the pipeline of operations could proceed while keeping as little as possible **dependency** among them was one the main challenges of the development. Besides, to provide scalability, the synchronization mechanism could not rely on the number of active workers.

⬦ *BSP - Bulk Synchronous Parallel* model MAPREDUCE can be seen as an instance of the **BSP** model, which helped to understand where the synchronization phase had to occur, and, subsequently how to properly design the synchronization mechanism.

⬦ *Scalability*
The implementation had to be **scalable**, allowing to add and remove workers dynamically. Kafka's design naturally allows consumers to go online and offline, but some parts of the implementation had to be designed with this in mind. The key point was keeping the synchronization mechanism not explicitly depending on the number of workers, but rather on the number of **partitions** per topic.

⬦ *CAP Theorem*
The CAP theorem tells us that in a distributed data store, it is impossible to simultaneously guarantee **consistency**, **availability**, and **partition tolerance**. This is relevant to our implementation, as we have to make some trade-offs between these properties when designing the messaging and synchronization architecture.
More specifically, this led to have two slightly different implementations, trading off the properties in different ways.

Testing the implementation

```
1  git clone git@github.com:frenzis01/MapReduceTS.git
2  cd MapReduceTS
3  # to clean output folder and remove old containers
4  ./clean_output.sh && docker container prune -f && docker compose up
5  # CTRL+C to stop the execution
```

1.1   Abstract

The overall MapReduce word count process

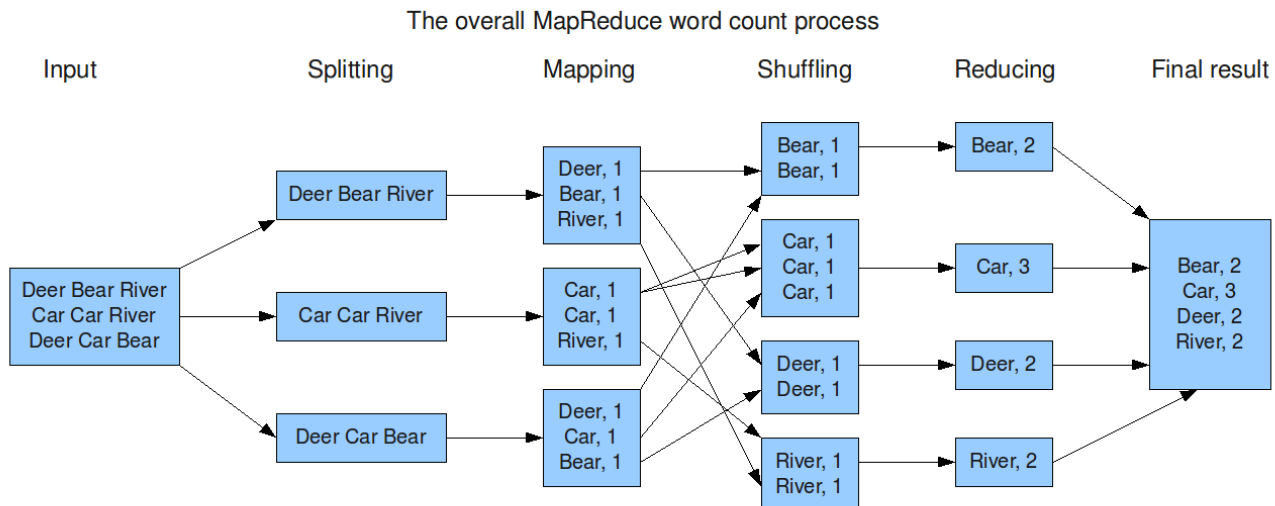| Input | Splitting | Mapping | Shuffling | Reducing | Final result |

Figure 1.1: MapReduce Word-Count schema

MapReduce, as we have discussed it, during the lectures, includes the steps depicted in Fig. 1.1, i.e.:

1. Input/Splitting
2. **Mapping** - `map(k1,v1) -> [(k2,v2),...]` is a function that takes a pair of data and returns a list of pairs of new keys paired with values, and is applied in parallel to all input data.[1]
3. **Shuffling** - Shuffling consists of flattening the list of values `[(k,v2), (k,v3), (k,v4), ...]` paired with a key `k` to get a pair like `(k,[v2,v3,v4,...])` , preparing it for the reduce phase. In other words, shuffling's purpose is to distribute the key-value pairs to reducer nodes, preferably evenly.
4. **Reducing** - `reduce(k2,[v2,...]) -> [(k3,v3),...]` is a function that takes a key and the related list of values and returns —typically one or none— pairs of keys and values.
5. Producing the result

The idea was to develop an implementation of MapReduce that allows the dynamic assignment of arbitrary *"pipelines"* to generic worker nodes. These pipelines consist of the `map` and `reduce` implementations for a given input data stream. This means that the actual `map` and `reduce` functions are not hardcoded in the workers, but are instead defined by the user in a `source` node, which is responsible for gathering input data and defining the related pipelines. Each worker simply knows its role in the architecture.

1.2   Architecture

1.2.1   Nodes

This led the application to consist of the following five types(/roles) of nodes:

1. `source` - responsible for gathering input data (from disk or elsewhere) and defining the related `map` and `reduce` functions (the *"pipeline"*, as it is referred to in the code), along with functions for selecting the key and the value from a data object, whose type and structure would be unknown to a worker.
   `source` nodes —mostly— ignore the underlying architecture of the MapReduce framework, they simply announce pipelines and send data records to a `dispatcher`.
   `source` nodes are the ones to be defined by the user, which have to obey some rules, but allow freedom regarding the actual MapReduce pipeline.
2. `dispatcher` - receives data records from the source and distributes them to mappers.
3. `worker` - this implements either a `mapper`, a `shuffler` or a `reducer`. The shuffling phase could also be performed as part of a `reducer` node, possibly improving performance and reducing message overhead, but it was kept separate for "separation of concerns" and to allow for possible more advanced load-balancing techniques for distributing records among reducer nodes.
4. `sink` - receives the results from the `reduce` and outputs them to disk, or elsewhere. As with `source` nodes, this may be customized to suit the user's needs.

---

[1]wikipedia.org/MapReduce

Clearly, the project includes basic implementations of `source` and `sink` nodes.

1.2.2   Message Passing

The exchange of messages between nodes is managed by a **Kafka broker**. Despite being the de-facto standard for distributed message passing, Kafka is *not* ideally suited for big data streams or high-throughput applications.
It is not perfect from a performance standpoint, partly because the broker acts as both a single point of failure and a bottleneck. Nevertheless, it was chosen because its concepts of topics and partitions fit well with some of the communication requirements (and potential issues to solve) of the framework, and because —based on how it was presented in the lectures— it seemed an interesting tool to explore and learn.

The initial, simpler and most intuitive Kafka topics architecture foresaw the following topics:
- ◇ `MAP_TOPIC`
- ◇ `SHUFFLE_TOPIC`
- ◇ `REDUCE_TOPIC`
- ◇ `OUTPUT_TOPIC`
- ◇ `PIPELINES_UPDATE_TOPIC` - Used for announcing new pipelines, allowing workers to get the `map` and `reduce` functions to execute.

The functioning of Kafka topics and groups naturally allows for multiple workers belonging to a single `'map-group'` to subscribe to `'MAP_TOPIC'` and simultaneously consume messages from it; clearly, the same applies for all the other topics. Hence, we can dynamically add workers to the pool to scale our application, and the Kafka broker will automatically handle the distribution of messages among them.

```
1   await producer.send({
2      topic: `${SHUFFLE_TOPIC}`,
3      messages: [{
4         key: resKey, // key resulting from the map operation
5         value: JSON.stringify(newMessageValue(resData, pipelineID)),
6         // partition: index
7         // 'partition' is not set, so that it is automatically computed based on resKey
8      }],
9   });
```

Listing 1.1: Sending MAP record to shuffle topic

Encoding the `pipelineID` in the `value` field allows us to distinguish the pipeline to which the data belongs; equivalently, we could also add a prefix in the `key` field of a Kafka message, something like `key: `${pipelineID}__map-record__{resKey}``. In this way, multiple data streams may be processed simultaneously with different `map` and `reduce` functions for each of them.

1.3   Dynamic Topic Creation architecture

1.3.1   Kafka message retransmission

Kafka consumers have an automatic mechanism for handling lost messages. Messages that have not been **committed** are **retransmitted** by the broker after some time. Hence, if a worker fails, another worker will resume the work it had left undone, based on the latest `offset` available for a given partition, which is incremented when *committing* messages (automatically performed by Kafka consumers).

1.3.2   Issues for lost messages

Each pipeline (a tuple ⟨`PipelineID`, `map`, `reduce`, `keySelector`, `dataSelector`⟩) defined in a `source` node is encoded in a `PipelineConfig` object, serialized as a string and later parsed in the workers.
Since pipeline updates are produced by `source` nodes and consumed by `worker`s in a dedicated topic (`PIPELINES_UPDATE_TOPIC`), it may happen that the `PipelineConfig` object for a pipeline `P1` is not yet available when a *data message* (to be processed by either a `mapper` or a `reducer`) related to `P1` is received, forcing the worker to add the message to a queue of "not-yet-ready-to-be-processed" messages.
This is necessary because the worker needs to know the `map` and `reduce` functions in order to process a data record, which are defined in the `PipelineConfig` object; thus, in case the latter is not available, the worker cannot process the message and has to wait for it to be received.

If in this scenario the worker crashes for some reason, its messages pending to be processed are lost with it. They cannot simply get retransmitted by the Kafka Broker, because in the meantime some other messages related to "ready"

pipelines $P_i \neq P_1$ would make the offset of the `MAP_TOPIC` or `REDUCE_TOPIC` progress. So even manually handling the offset is not a viable solution.

Instinctively it may seem appropriate to halt the worker until the `PipelineConfig` object is available, to allow Kafka's retransmission system to work, but —in case the `PipelineConfig` is never received, or it is received after a long time— this would lead to a performance drop, since the worker would not be able to process any other messages while waiting for the `PipelineConfig` object.

This led to the development of an alternative and more intricate architecture for topics, involving two **dynamically created topics** (`MAP/REDUCE`) for each pipeline `Pi`. Having a topic for each data stream allows for separate message committing, hence avoiding the issue of lost messages.

1.3.3 Dynamic topics

This approach is implemented in the `multi-topic` branch of the repo.

This implementation is error prone. Error messages are displayed due to rebalancing, which is expected, but something else happens under the hood which I couldn't figure out, leading to message duplication and other issues.
Even though the code looked correct from a theoretical point of view, in practice something doesn't work out, possibly due to the approach attempting to stretch out too much Kafka's natural behavior. Nevertheless, I still thought that a brief discussion on this implementation could be interesting.

The `dispatcher` node, upon receiving a new pipeline from a source, creates two topics specific for it:

```
1    [  topic: `${MAP_TOPIC}---${pipelineID}`,
2       topic: `${REDUCE_TOPIC}---${pipelineID}`  ]
```

For what concerns **shuffler**s instead, since they do not need to compute either `map` or `reduce`, we can use a fixed unique `${SHUFFLE_TOPIC}`, and encode the `pipelineID` in the `value` field of a message as displayed in 1.1.
Recall that **shuffler**s only need the `key` and the `pipelineID` to perform their job, and these two are encoded in the `key` and `value` fields of a Kafka message, respectively.

mappers and **reducer**s have to subscribe to the topics dynamically created by the `dispatcher` while consuming other messages.
This is in general *not* trivial for Kafka.
`KafkaJS` does *not* allow a consumer to subscribe to a topic while running and consuming messages.

The only way to make consumers implement this mechanism appears to be:
1. `stop()` the consumer
2. `disconnect()` it from the broker
3. `connect()` again to the broker
4. `subscribe({newTopic})` to new topics
5. *start* (`run(eachMessage: ...)`) consuming messages again

Skipping disconnecting and reconnecting caused issues with duplicated messages and other weird behaviours.

Clearly, every time a worker performs these operations, a *rebalance* is triggered, since the broker has to recompute the partition assignment to consumers of both the newly created topic and the ones to which consumers had already subscribed, resulting in the application essentially halting for a few seconds.

The **simpler approach** depicted earlier at 1.2.2 **avoids this** and follows the more natural way of working for Kafka, where a consumer does not need to decide at runtime to which topics it should be interested in, or at least, decides "only once"; however, this comes at the cost of losing the benefits of dedicating a topic for each data stream, such as handling lost messages.

CAP theorem reminder

The CAP theorem states that we can simultaneously have only two properties among *Consistency, Availability*, and *Network Partitioning*. In our network-partitioned scenario, enforcing consistency leads to periods of unavailability, while guaranteeing (better) availability implies the possibility of ending up in an inconsistent state with some information lost.

1.4 Partitioning and Scalability
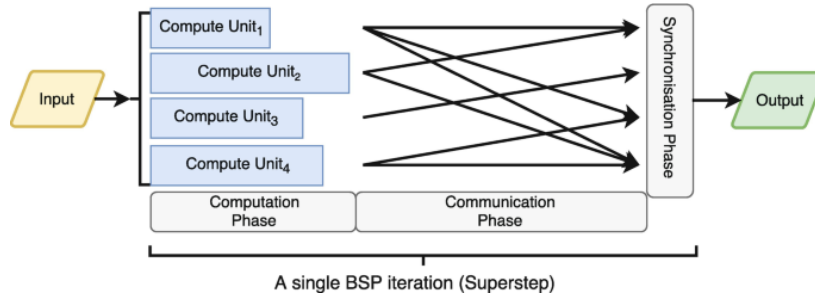
1.4.1 BSP model

Figure 1.2: BSP Model schema

BSP model is a way of thinking about distributed applications, which consists in dividing the application into *supersteps*, where each superstep consists of a sequence of operations which are performed in parallel by the workers, followed by a synchronization phase where all the workers wait for each other to finish their superstep before proceeding to the next one.

In our case, synchronization occurs at the end of the computation *map* records, and after they have been "communicated" to `shuffler`s. The keys for a given pipeline can be sent to the `reduce` only after all the *map*s for that pipeline have been completed.

Shuffling

Since the shuffling process consists in putting map records into buckets, each bucket being a key, this happens as the map records are being consumed.

Note that thanks to Kafka, records with the same key are always sent to the same partition, hence they will be consumed by the same `shuffler` node, which will then be able to group them together.

The shuffling phase can be considered over when there are no more *map* records to be consumed. At that point, the `shuffler` nodes can start sending the *shuffle* records to the `reduce` topic, which will be consumed by the `reducer` nodes.

Reducing

The `reducer` nodes can start consuming messages from the `reduce` topic as soon as they are available, and process them right away. This is because the `reduce` operation needs only the list of values for a given key, which is all packed in a single *shuffle* record, and does not depend on the completion of other *reduce* operations. Besides, given that the `shuffler`s wait for all `map` records to be consumed before sending the *shuffle* records, we can be sure that all the *map*s for a given key have been completed before the *shuffle* record is sent to the `reduce` topic, so if a shuffle record is received, then we are *sure* that all the *map*s for that key have been completed and are in such *shuffle* record.

Note also that if a `reducer` receives the *shuffle* record for a key k_1, it is guaranteed by Kafka that it is the only worker handling k_1, so no duplicated processing will happen.

The `sink` nodes instead, can simply start consuming messages from the `OUTPUT_TOPIC` as soon as they are available, without waiting for all the `reduce` operations to finish.

1.4.2 Implementation Key Points

There are some key points concerning our implementation which we have to highlight:

◇ There must be a special message to signal the end of *source* records, so that the *dispatcher* node knows when the data stream has ended.

◇ The *dispatcher* has to propagate the end of the source data stream to all the *map* nodes, so that they know when to propagate the end-of-stream message to the *shuffle* nodes.

◇ There must be special messages to signal the end of *map* records, so that the *shuffle* nodes know when they can

start feeding the reduce nodes.

◇ The special messages will have a special `key` and a special `value`.

◇ Topics are divided into partitions, which are the basic unit of parallelism in Kafka. Each partition is consumed by a single worker, but a single worker can consume multiple partitions.

◇ The partition to which a message is sent is determined by the `key` of the message, which is hashed to determine the partition. This means that messages with the same key will always be sent to the same partition, and hence will be consumed by the same worker.

◇ Partitions indicate the degree of parallelism of the application, i.e. how many workers can consume messages from a topic at the same time.

1.4.3   Ending the Stream

The end of the stream is signaled by a special message with a special key and value, which is sent by the `source` node to the `dispatcher`.

Listing 1.2: Sending STREAM_ENDED message from source to dispatcher

```
1    await producer.send({
2       topic: DISPATCHER_TOPIC,
3       messages: [{
4          key: `${pipelineID}__${STREAM_ENDED_KEY}`,
5          value: JSON.stringify(newStreamEndedMessage(pipelineID, data.length)),
6       }],
7    });
```

The dispatcher will then forward this message to all the *map* nodes, which will then propagate it to the *shuffle* nodes, which will then propagate it to the *reduce* nodes, and finally to the `sink` nodes.

However, there is a major problem which must be addressed and discussed, before talking about the solution. A message with this key `${pipelineID}__${STREAM_ENDED_KEY}`, will only be received by *exactly one* map consumer, so there must be some way to either propagate or distribute it among the consumers.

1.4.3.1   Partitions for message propagation

It is crucial for a proper distributed and scalable architecture to allow adding and removing workers, so the propagation *cannot* rely on workers knowing each other or making assumptions about the cluster topology.

To solve this we can use a **fixed** explicit number of partitions per topic, such as 10. This will represent the maximum number of workers that can consume messages from a topic at the same time, so having more workers than partitions will not lead to performance improvements. This will be one of the few data structures shared among nodes.

The number of partitions can be adjusted by the user to fit the needs of the application.

Listing 1.3: BUCKET_SIZE represents the number of partitions per topic

```
1     for (let i = 0; i < BUCKET_SIZE; i++) {
2        await producer.send({
3           topic: `${MAP_TOPIC}---${pipelineID}`,
4           messages: [{
5              key: `${pipelineID}__${STREAM_ENDED_KEY}`,
6              value: JSON.stringify(newStreamEndedMessage(pipelineID, null)),
7              partition: i
8           }],
9        });
10    }
```

We can send the message to all the partitions of the topic as depicted in Lst. 1.3, so that it is guaranteed that all consumers will receive it. If there are less workers than partitions, some of them will receive the message multiple times, but we will address this later on.

1.4.3.2   Message ordering

It is important to note that Kafka guarantees message ordering only within a partition, not across partitions. Here lies the reason why we can't have a `mapper` receiving the *end of stream* message to propagate it to other mappers polling other partitions. It may happen that the propagated end message is received by a `mapper` before other remaining source records, which would lead to the `mapper` not processing them, and hence losing some data.

Hence, we must send a `STREAM_ENDED` for each `MAP_TOPIC` partition from the dispatcher after it has sent all the source records. In this way, we are sure that the in each partition, the `STREAM_ENDED` message is received after all the source records.

1.4.4   Wrap Up

Below we have a scheme wrapping up the overall messaging architecture of the application, to display how the messages flow among the nodes.
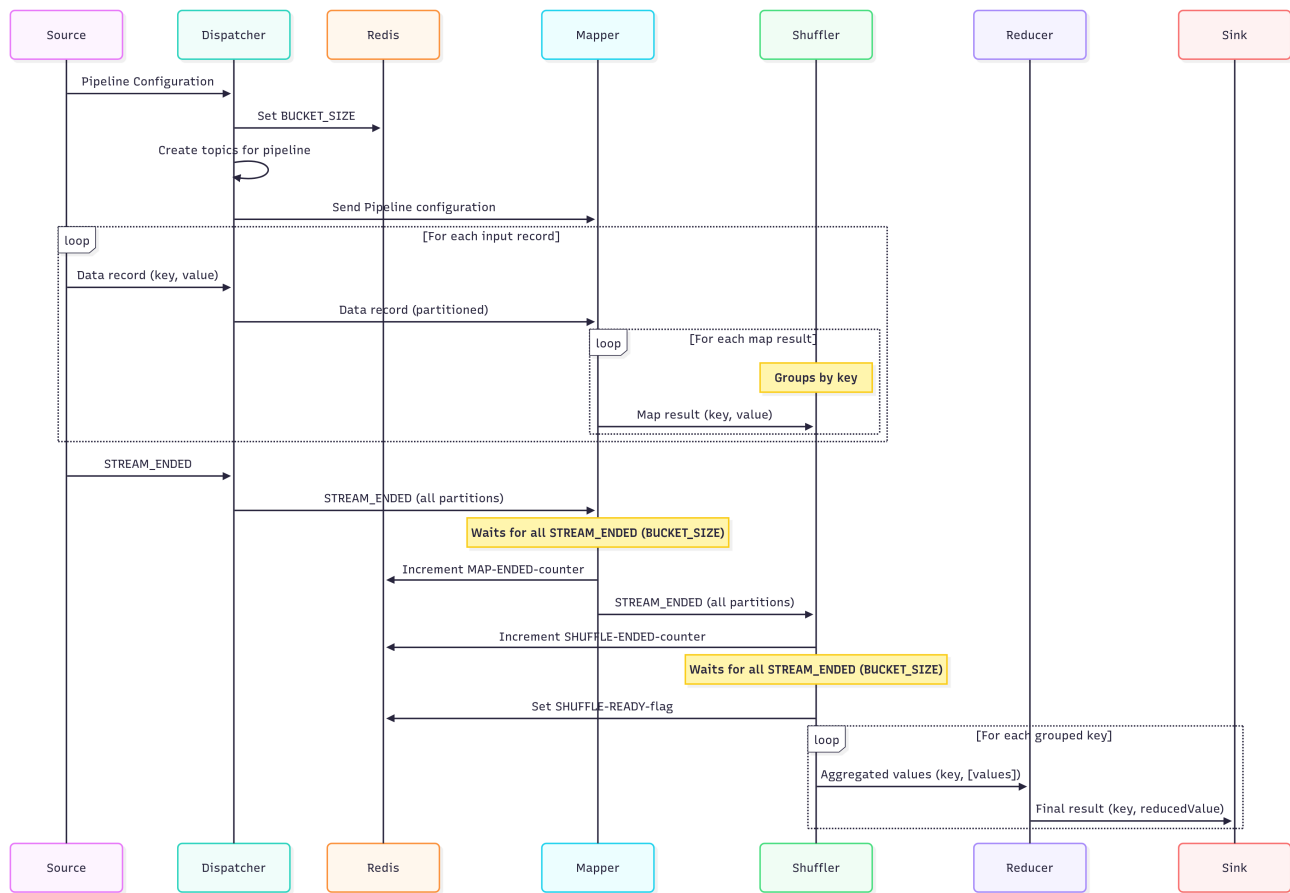

Figure 1.3: Message passing scheme

1.4.5   Deeper into the stream ending

1.4.5.1   Mappers

A mapper receiving its `STREAM_ENDED` message cannot directly propagate it to the shuffle topic, because other mappers may still be processing source records. The only way I could find to guarantee that all mappers have finished processing is to use a **redis shared counter**. When a mapper receives the `STREAM_ENDED` message, it increments the counter in **redis**. If the counter equals the number of mappers, then the mapper can propagate the `STREAM_ENDED` message to the shuffle topic. This is shown in Listing 1.4.

Listing 1.4: Handling the STREAM_ENDED message in a mapper

```
// Increment the counter for the number of ended messages
await redis.incr(`${pipelineID}-MAP-ENDED-counter`);

// Dispatcher sends one STREAM_ENDED message for each partition, i.e. BUCKET_SIZE times
// We need to wait for all the STREAM_ENDED messages to arrive before starting to send to shuffle
const counter = await redis.get(`${pipelineID}-MAP-ENDED-counter`);
if (!counter || Number(counter) !== BUCKET_SIZE) {
   // In case we have not yet received all the STREAM_ENDED messages, we simply return,
   // as we are not ready to send to shuffle yet, and we have already incremented the counter
   console.log(`[MAP/${WORKER_ID}] Received stream ended message. Got ${counter}/${BUCKET_SIZE}
       messages... for ${pipelineID}`);
   return;
}
else {
   // One of the mappers will receive the last STREAM_ENDED message from the dispatcher
   // and enter this else branch. Here, we propagate the STREAM_ENDED message to the shuffle,
   // One for each partition, i.e. BUCKET_SIZE times
   console.log(`[MAP/${WORKER_ID}] Received last STREAM_ENDED message. `);

   // Send to shuffle consumer special value to start feeding the reduce
   // Send onto all partitions
```

```
21        for (let i = 0; i < BUCKET_SIZE; i++) {
22            await producer.send({
23                topic: `${SHUFFLE_TOPIC}`,
24                messages: [{
25                    key: `${pipelineID}__${STREAM_ENDED_KEY}`,
26                    value: JSON.stringify(newStreamEndedMessage(val.pipelineID, null)),
27                    partition: i,
28                }],
29            });
30        }
31        console.log(`[MAP/${WORKER_ID}] Propagated stream ended message to shuffle...`);
```

1.4.5.2 Shufflers

The shuffler nodes use similar logic for `STREAM_ENDED` messages. They handle standard map records by grouping them by key, but when a `STREAM_ENDED` message is received, they increment a counter in `redis`. If it's still below the `BUCKET_SIZE` threshold, they do nothing and return. The shuffler receiving the last `STREAM_ENDED` message sets a `redis` flag, signaling that the shuffling phase is over, and sends another `STREAM_ENDED` message to all *shuffle* partitions to "wake up" the shufflers and have them start sending the *shuffle* records to the *reduce* topic.
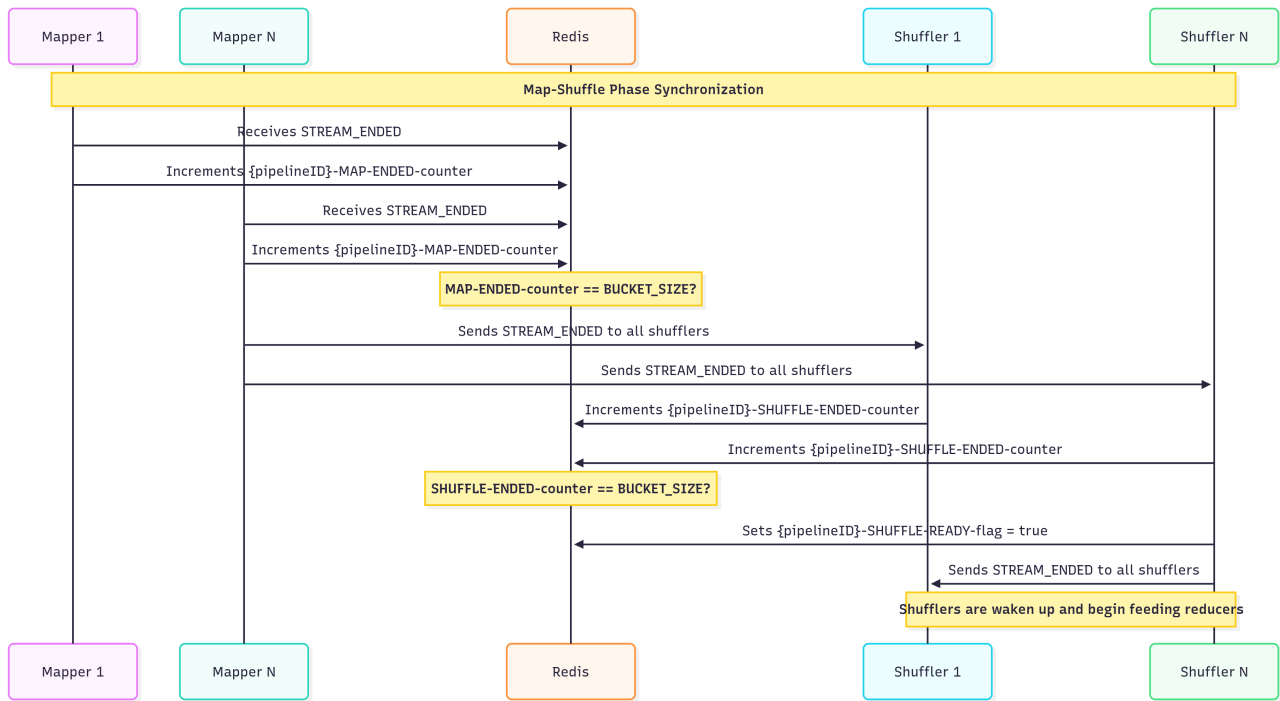
Listing 1.5: Handling the STREAM_ENDED message in a shuffler

```
1    // Increment the counter for the number of ended messages
2    if (!await redis.get(`${pipelineID}-SHUFFLE-READY-flag`)) {
3        await redis.incr(`${pipelineID}-SHUFFLE-ENDED-counter`);
4        const streamEndedCounter = await redis.get(`${pipelineID}-SHUFFLE-ENDED-counter`);
5        console.log(`[SHUFFLE/${WORKER_ID}] Got ${streamEndedCounter} STREAM_ENDED messages...`);
6    }
7    // If we have not yet received all the STREAM_ENDED messages, we simply return
8    const streamEndedCounter = await redis.get(`${pipelineID}-SHUFFLE-ENDED-counter`);
9    if (!streamEndedCounter || Number(streamEndedCounter) < BUCKET_SIZE) {
10        return;
11    }
12    // if reached the number of STREAM_ENDED messages, wake everyone with a stream Ended message
13    // having a flag set to make them recognize it as a dummy message
14    if (!(await redis.get(`${pipelineID}-SHUFFLE-READY-flag`))) {
15        await redis.set(`${pipelineID}-SHUFFLE-READY-flag`, `true`);
16        for (let i = 0; i < BUCKET_SIZE; i++) {
17            await producer.send({
18                topic: `${SHUFFLE_TOPIC}`,
19                messages: [{
20                    key: `${pipelineID}__${STREAM_ENDED_KEY}`,
21                    value: JSON.stringify(newStreamEndedMessage(val.pipelineID, val.data)),
22                    partition: i
23                }],
24            });
25        }
26    }
27
28    // At this point we are ready to send the shuffle records to the reduce topic, all STREAM_ENDED
           messages have been received
29    if (await redis.get(`${pipelineID}-SHUFFLE-READY-flag`)) {
30        // Send to REDUCE_TOPIC the shuffle records
```

Figure 1.4: Scheme wrapping up the redis logic for synchronizing the end of the stream

| Mapper 1 | Mapper N | Redis | Shuffler 1 | Shuffler N |

Map-Shuffle Phase Synchronization

Receives STREAM_ENDED

Increments {pipelineID}-MAP-ENDED-counter

Receives STREAM_ENDED

Increments {pipelineID}-MAP-ENDED-counter

MAP-ENDED-counter == BUCKET_SIZE?

Sends STREAM_ENDED to all shufflers

Sends STREAM_ENDED to all shufflers

Increments {pipelineID}-SHUFFLE-ENDED-counter

Increments {pipelineID}-SHUFFLE-ENDED-counter

SHUFFLE-ENDED-counter == BUCKET_SIZE?

Sets {pipelineID}-SHUFFLE-READY-flag = true

Sends STREAM_ENDED to all shufflers

Shufflers are waken up and begin feeding reducers

| Mapper 1 | Mapper N | Redis | Shuffler 1 | Shuffler N |

1.4.5.3 Reducers

The reducer nodes will simply perform the reduce operation on the *shuffle* records they receive and forward the results to the *sink* topic. No `STREAM_ENDED` messages are needed anymore.

> **Making the sinks know when the stream has ended**
>
> Making sink nodes know when the stream has ended would require the same "messy" logic as above. Given that the sink nodes are not meant to be used for further processing, but only for outputting the results, and that ideally they should be "user-defined", i preferred to let things be simpler.

1.5 Conclusions

Throughout the development, it has been interesting to deal with the issues discussed earlier, especially the **synchronization** among nodes. Kafka's design naturally avoided the need to handle access to critical code sections or shared data, but some synchronization mechanism over the MAPREDUCE phases was still needed, as the overall process essentially follows the BSP model.

The major challenge that may arise with this type of system appears to be message passing and synchronization handling while maintaining scalability and isolation among nodes.

An enhancement in the computational performance of a single operation may be negligible or ineffective if not supported by an efficient and robust way to communicate the result of such operation. In this basic implementation, the Kafka broker acts as the main bottleneck, since it must handle all messages. For very basic operations (such as word counting), the overhead introduced by Kafka's message passing, even if batching is used, may be more significant than the processing itself.

A posteriori, an architecture similar to the adopted one may better suit node clusters that do not have to work as "sequentially" as in the MAPREDUCE framework, where the information exchanged among them is less intensive and does not enforce strict process ordering, allowing for better exploitation of Kafka's design while avoiding synchronization delays.

The aim of the project was to pose challenges whose resolution has led me to gain a deeper understanding of Kafka, MapReduce, TypeScript —which I had never used before—, and other concepts such as Synchronization and Scalability.