# MapReduce - TS
# Scalable and Distributed Computing Project

Francesco Lorenzoni

January 2025

# Chapter 1

# Introduction

This is the report of the project developed for the "Scalable and Distributed Computing" held at UniPi during the academic year 2024/25 by Prof. Dazzi. The project developed consists of a implementation in TypeScript of the MapReduce framework exploiting Kafka for message delivery and Docker for the deployment. The purpose was *not* to develop a highly-performant resilient implementation of MapReduce, we have discussed other implementations such as Hadoop and Spark which work fine and surely won't be outperformed by a small project developed by a university student ☺. The intent was instead to define a problem whose resolution would led to face, 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, to later discuss how it works under the hood and how it relates to the concepts discussed in the course.

The course topics this projects touches are:

- MapReduce
- Kafka and Distributed Messaging
- Synchronization
- Scalability
- BSP - Bulk Synchronous Parallel model
- CAP Theorem

## 1.1 Abstract

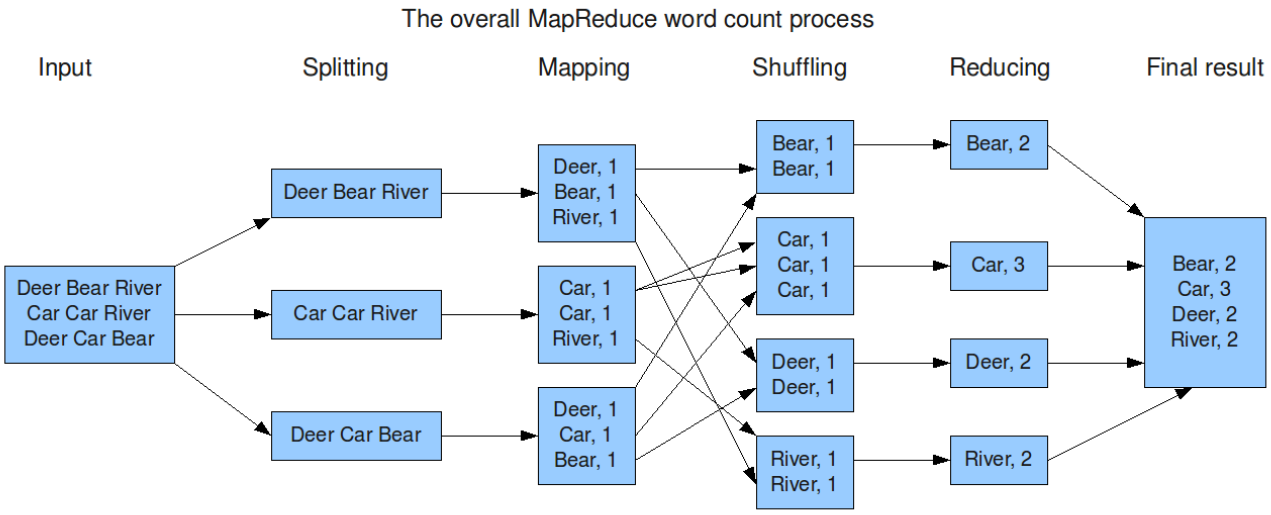


Figure 1.1: MapReduce Word-Count schema

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

1. Input/Splitting
2. **Mapping** - `map(k1,v1)-> [(k2,v2),...]` is a function taking a pair of data and returning a list of pairs of new keys paired with values, and is applied in parallel to all input data. [1]
3. **Shuffling** - Shuffling consists in 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. In other words, shuffling's purpose is to distributed the key,value pairs towards reducers nodes, possibly evenly.
4. **Reducing** - `reduce(k2,[v2,...])-> [(k3,v3),...]` is a function taking 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 allowing to dynamically assign to some generic worker nodes arbitrary "pipelines", intended as the `map` and `reduce` implementations for a given input data stream.
This means 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. Every 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 whatever) 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 for a worker.
   `source` nodes 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 to some rules, but allow for freedom for what concerns the actual MapReduce pipeline
2. `dispatcher` - receives data records from the source and distributes them towards 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 by also improving performance and reducing message overhead, but it was kept separated for "separation of concerns" and allow to eventually establish more advanced load-balancing techniques for distributing records amongst reducer nodes.
4. `sink` - receives the results by the `reduce` and outputs them to disk, or whatever. As `source` nodes, this may be customized to suit the user's needs.

[1] wikipedia.org/MapReduce

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

### 1.2.2 Message Passing

The exchange of messages between the nodes is managed by a **kafka broker**. Despite being the de-facto standard for distributed message passing, Kafka is *not* properly suited for big data streams or high-throughput applications. It is not perfect from a performance point of view, also because the broker acts as both a single point-of-failure and a bottleneck, but nevertheless it was chosen because its concepts of topics and partitions nicely fit 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 delve into and to 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 subcribe 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,
5        value: JSON.stringify(newMessageValue(resData, pipelineID)),
6        // partition: index
7     }],
8  });
```

Listing 1.1: Sending MAP record to shuffle topic

Encoding the `pipelineID` in the `value` field allows 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 way of handling lost messages. Messages which have not been committed, after some time, are retransmitted by the broker. Hence, in case a worker fails, someone else will resume the work it had left undone, based on the latest `offSet` available for a given partition, which is increased when committing messages (automatically perfomed by Kafka consumers).

### 1.3.2 Issues for lost messages

Each pipeline 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 `workers` 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; so, 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 `Pi != P1` would make the offset progress.

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 intricated architecture for topics, involving two **dynamically created topics** (`MAP/REDUCE`) for each pipeline `Pi`.

### 1.3.3 Dynamic topics

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

The `dispatcher` node, upon the receival of 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 need to compute neither `map` or `reduce`, we can use a fixed unique `` `${SHUFFLE_TOPIC}` ``, and encode the `pipelineID` in the value 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.

`mapper`s 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 functioning way to make consumers implement this mechanism seems to be to:

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 recompute the partitions assignment to consumer 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 a runtime to what 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 recalls**

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

## 1.4 Partitioning and Scalability

### 1.4.1 BSP model

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, the synchronization happens at the end of *map* records. 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, without waiting for all the *reduce*s to finish. This is because the reduce 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*s. Besides, given that the `shuffler`s waits 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 the *shuffle* record.
Note also that if a `reducer` receives the *shuffle* record for a key $K1$, it is guaranteed by Kafka that it is the only worker handling $K1$, so no duplicated processing will happen.

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

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 stream to all the *map* nodes, so that they know when to propagate the end of the 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.

### 1.4.2 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`.

```
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.

#### 1.4.2.1 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.
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.

```
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 **??**, 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.

#### 1.4.2.2 Message ordering

It is important to note that Kafka guarantees message ordering only within a partition, not across partitions. This here lies the reason why we can't have a mapper receiving the *end of stream* message to propagate it to other mappers. 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 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.

### 1.4.3 Wrap Up

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

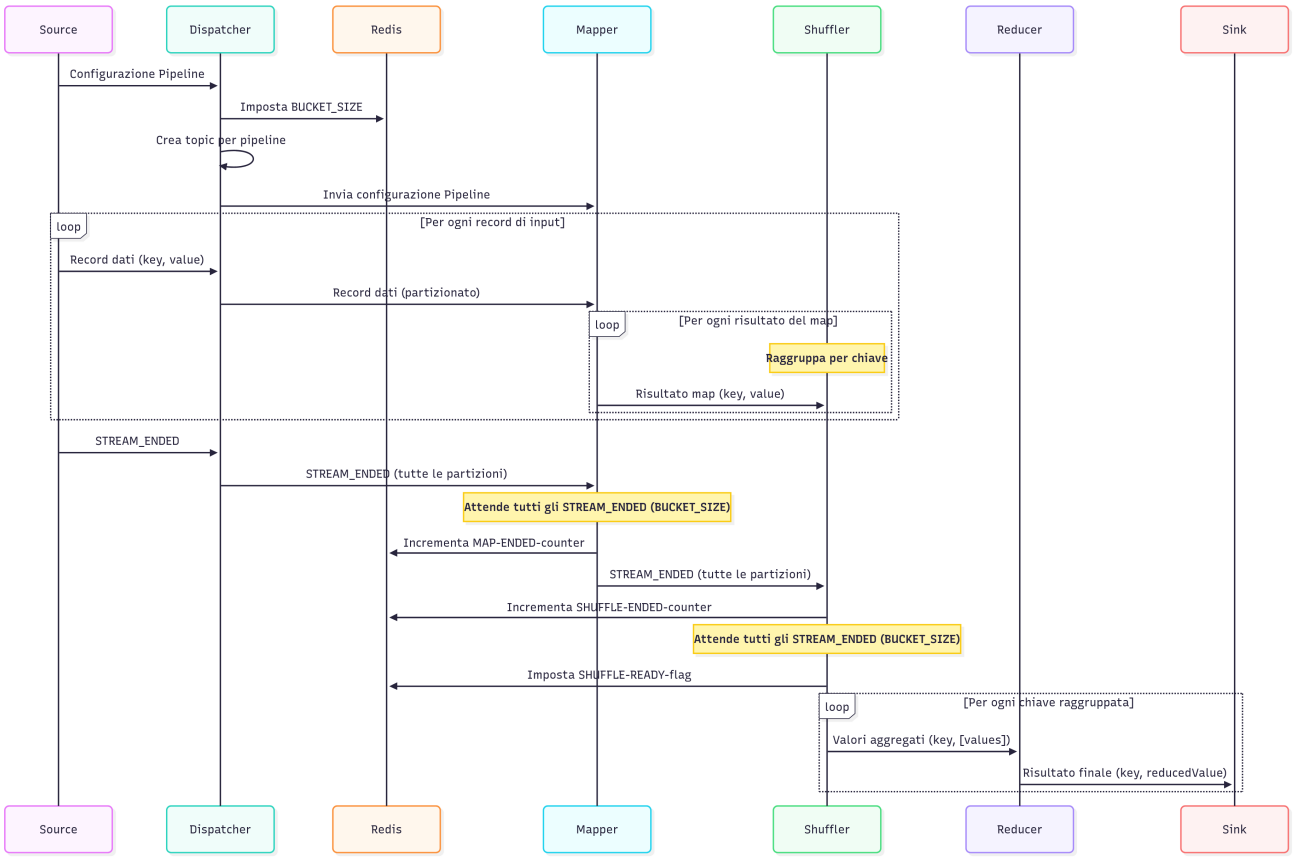


Figure 1.2: Message passing scheme

### 1.4.4 Deeper into the the stream ending

#### 1.4.4.1 Mappers

A mapper receiving his `STREAM_ENDED` message cannot propagate directly propagate it to the shuffle topic, because it may happen that other mappers are still 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. In case the counter is equal to the number of mappers, then the mapper can propagate the `STREAM_ENDED` message to the shuffle topic. This is displayed in Lst.1.2.

```
1   // Increment the counter for the number of ended messages
2   await redis.incr(`${pipelineID}-MAP-ENDED-counter`);
3
4   // Dispatcher sends one STREAM_ENDED message for each partition, i.e. BUCKET_SIZE times
5   // We need to wait for all the STREAM_ENDED messages to arrive before starting to send to shuffle
6   const counter = await redis.get(`${pipelineID}-MAP-ENDED-counter`);
7   if (!counter || Number(counter) !== BUCKET_SIZE) {
8      // In case we have not yet received all the STREAM_ENDED messages, we simply return,
9      // as we are not ready to send to shuffle yet, and we have already incremented the counter
10     console.log(`[MAP/${WORKER_ID}] Received stream ended message. Got ${counter}/${BUCKET_SIZE}
           messages... for ${pipelineID}`);
11     return;
12  }
13  else {
14     // One of the mappers will receive the last STREAM_ENDED message from the dispatcher
15     // and enter this else branch. Here, we propagate the STREAM_ENDED message to the shuffle,
16     // One for each partition, i.e. BUCKET_SIZE times
17     console.log(`[MAP/${WORKER_ID}] Received last STREAM_ENDED message. `);
18
19     // Send to shuffle consumer special value to start feeding the reduce
20     // Send onto all partitions
21     for (let i = 0; i < BUCKET_SIZE; i++) {
22        await producer.send({
23           topic: `${SHUFFLE_TOPIC}`,
```

```
            messages: [{
               key: `${pipelineID}__${STREAM_ENDED_KEY}`,
               value: JSON.stringify(newStreamEndedMessage(val.pipelineID, null)),
               partition: i,
            }],
         });
      }
      console.log(`[MAP/${WORKER_ID}] Propagated stream ended message to shuffle...`);
```

Listing 1.2: Handling the STREAM_ENDED message in a mapper

#### 1.4.4.2 Shufflers

The shufflers nodes exploit a similar logic for `STREAM_ENDED` messages. They handle standard map records by grouping them by key, but in case a `STREAM_ENDED` message is received, they increment a counter in redis, and if it's still below `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" the shufflers, and have them start sending the *shuffle* records to the *reduce* topic.

```
   // Increment the counter for the number of ended messages
   if (!await redis.get(`${pipelineID}-SHUFFLE-READY-flag`)) {
      await redis.incr(`${pipelineID}-SHUFFLE-ENDED-counter`);
      const streamEndedCounter = await redis.get(`${pipelineID}-SHUFFLE-ENDED-counter`);
      console.log(`[SHUFFLE/${WORKER_ID}] Got ${streamEndedCounter} STREAM_ENDED messages...`);
   }
   // If we have not yet received all the STREAM_ENDED messages, we simply return
   const streamEndedCounter = await redis.get(`${pipelineID}-SHUFFLE-ENDED-counter`);
   if (!streamEndedCounter || Number(streamEndedCounter) < BUCKET_SIZE) {
      return;
   }
   // if reached the number of STREAM_ENDED messages, wake everyone with a stream Ended message
   // having a flag set to make them recognize it as a dummy message
   if (!(await redis.get(`${pipelineID}-SHUFFLE-READY-flag`))) {
      await redis.set(`${pipelineID}-SHUFFLE-READY-flag`, `true`);
      for (let i = 0; i < BUCKET_SIZE; i++) {
         await producer.send({
            topic: `${SHUFFLE_TOPIC}`,
            messages: [{
               key: `${pipelineID}__${STREAM_ENDED_KEY}`,
               value: JSON.stringify(newStreamEndedMessage(val.pipelineID, val.data)),
               partition: i
            }],
         });
      }
   }

   // At this point we are ready to send the shuffle records to the reduce topic, all STREAM_ENDED
       messages have been received
   if (await redis.get(`${pipelineID}-SHUFFLE-READY-flag`)) {
      // Send to REDUCE_TOPIC the shuffle records
```

#### 1.4.4.3 Reducers

The reducer nodes, will simply perform the reduce operation on the *shuffle* records they receive, and forward the the results to the *sink* topic. No `STREAM_ENDED` is needed no more.

> **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.

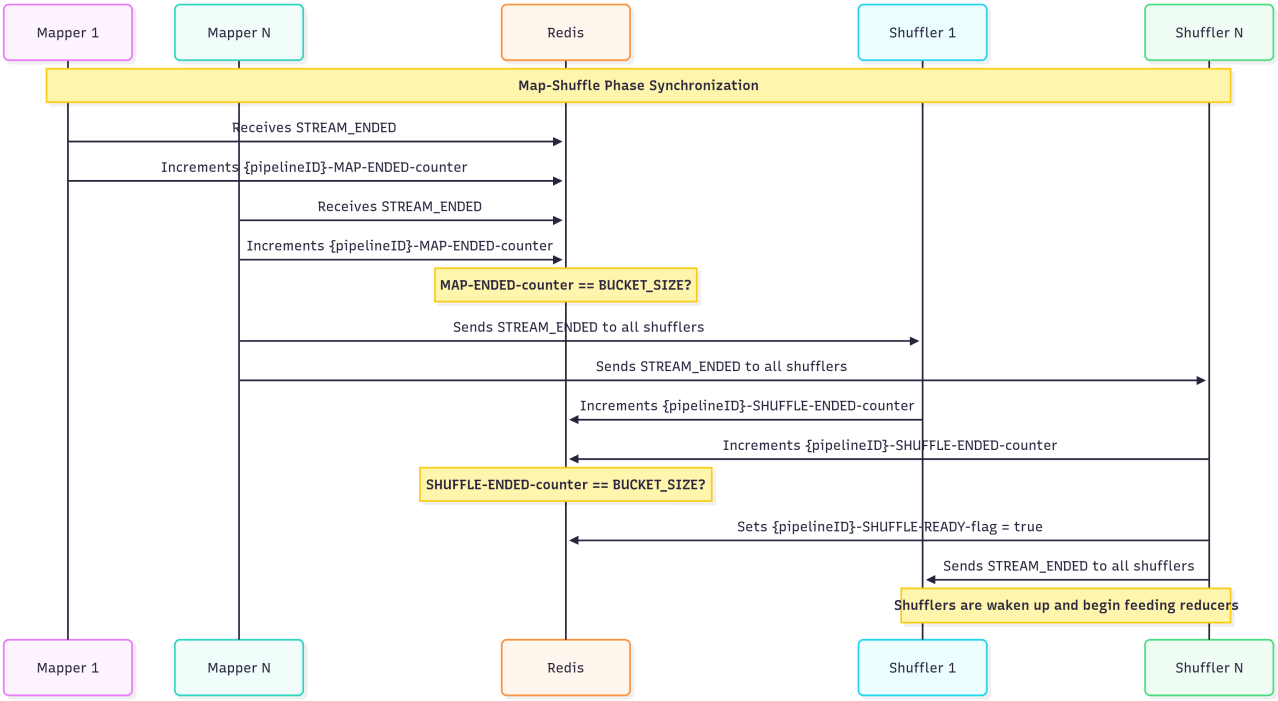


Figure 1.3: Scheme wrapping up the redis logic for synchronizing the end of the stream