# MapReduce Framework

## MapReduce

MapReduce[2] is a two-step approach to distributing large computations over multiple servers, in which, on first step (Map), transformation of data occurs, and on second step (Reduce) data is combined to obtain the final answer.

### Map.

Map is defined as a function:

$$Map : K \times V \longrightarrow K' \times V'$$

Here,

- **Keys:** $K$ and $K$ are universes of keys, unique identifiers of the data being processed;
- **Values:** $V$ and $V'$ are universes of values — the data itself.

A more exact signature of a single execution of Map is

$$Map : (K \rightarrow V) \longrightarrow (K' \rightarrow V')$$

Map is a transformation function. It takes as input a dataset identified by keys from set $V$, and performs a transformation operation, by extracting from each tuple $(k, v)$, the new key value $v'$ from domain $V'$, and the new data value $k'$.

**Examples.** Here are some examples of Map.

**Word count.** Input: $(docID, Text)$ pairs, where $docID$ is a document id, and $Text$ is the text of the document.

The Map function works as follows:

```
map (String docId, String Text):
   for each w in Text   // for each word in text document
       emit(w, "1");
   end for
end
```

Map transforms a collection of documents into a stream of $(w, "1")$ pairs, where the new keyspace is the list of words found in the entire document collection.

**Aggregation.** Suppose our data is tuples of a form $(EmployeeId, Department, Salary)$ and our goal is to compute the total salary for each department.

The data is mapped as follows: $EmployeeId$ is $Key$, a pair $(Department, Salary)$ is $Value$. The Map function performs the equivalent of `GROUP BY` clause in the

```
SELECT Department, SUM(Salary)
FROM Employees
GROUP BY Department;
```

```
map (String Key, String Value):
  emit (Value.Department, Value.Salary);
end
```

## Reduce

Reduce is the *aggregation component* of the MapReduce framework. It's functional signature is:

$$Reduce : (K' \rightarrow (V')^*) \longrightarrow (V')^*$$

That is, given a mapping between keys from domain $K'$ and *lists* of values from domain $V'$, Reduce computes a list of values from the same domain $V'$.

In many applications, Reduce actually comuptes a single value:

$$Reduce : (K' \rightarrow (V')^*) \longrightarrow V'$$

**Examples.**

**Word count.** Input: $(word, List)$ pairs, where $word$ is a word in the document corpus, and $List$ is a collection of "1" strings emitted by the Map function above.

The Reduce function works as follows:

```
reduce (String word, List<String> Counts):
   count := 0;
```

```
    for each l in Counts  // for discovered occurrence of the word
        count := count+1;
    end for
    emit(toString(count));
end
```

Here is an alternative, functional way to describe Reduce:

```
reduce(w, []) --> toString(0);
reduce(w, [Head|Tail]) -->  toString(toInt(reduce(w, Tail)) + 1);
```

**Aggregation.** Recall that we want to compute the result of the query:

```
SELECT Department, SUM(Salary)
FROM Employees
GROUP BY Department;
```

Map groups salary information by department, so the input to the Reduce function is $(Department, SalaryList)$, where $SalaryList$ contains salary numbers for each employee of the department.

The Reduce function for this example is similar to the one above:

```
reduce(Department, []) --> toString(0);
reduce(w, [Head|Tail]) -->  toString(toInt(reduce(w, Tail)) + toInt(Head));
```

## MapReduce Implementation

**Hardware:** large cluster of commodity PCs, connected with switched Ethernet [2, 1]. Properties:

- local storage
- node failures are common
- distributed file system

**MapReduce run parameters.** MapReduce requires some setup parameters:

- **number of splits (M).** The number of chunks into which the input dataset is partitioned for the Map stage.
- **number of intermediate key partitions (R).** The number of chunks into which the intermediate key space is partitioned in Reduce.
- **partitioning function $h()$.** The (hash) function used to partition intermediate keys into $R$ partitions.

**Overall organization.**

- multiple machines.
- one *master* server (process): controls MapReduce flow, assigns tasks to other machines.
- multiple *worker* nodes: accept tasks from the *master*, perform them.

**Map overview.**

1. **Worker selection.** Master selects $M$ workers, assigns to each a Map task with a given split $D_i$ of the data $D = D_1 \cup \ldots D_M$.
2. **Worker operation.** Worker $w_i$ operates as follows:
   (a) processes contents of split $D_i$.
   (b) Runs Map on each (key, value) pair, produces (iKey, iValue) pair.
   (c) Buffers (iKey, iValue) pairs in main memory.
   (d) When buffer is filled, stores (iKey, iValue) pairs to disk.
   (e) When storing (iKey, iValue) pairs to disk, the partitioning function $h(iKey)$ is run to determine the partition of each pair. All data produced is split into $R$ partitions.
   (f) Reports to master the location (in the distributed file system) of the created partitions.

**Reduce overview.**

1. **Worker selection.** When master is notified by Map workers that data is ready for Reduce operations, it selects $R$ workers for Reduce. Each worker $wr_j$ is assigned partition $R_j$ of the intermediate data.
2. **Worker operation.** Worker $wr_j$ operates as follows.
   (a) Master notifies Reduce worker $wr_j$ of the data available from Map worker $w_i$.
   (b) $wr_j$ accesses the partition $j$ of intermediate data from $w_i$ (partition $R_{ij}$).
   (c) $wr_j$ sorts $R_{ij}$ on iKey values. This effectively turns a list of (iKey, iValue) pairs, into a list of (iKey, (iValue1,. . . ,iValueN)) pairs.
   (d) For each intermediate key value iKey, the pair (iKey, (iValue1,. . . ,iValueN)) is passed to Reduce function.
   (e) The results of running `Reduce` on $R_{ij}$ are combined with the results obtained from running it on data obtained from other Map workers.
   (f) When all Map processes stop, $wr_j$ finishes Reduce processing and passes the location of the output to the master.
3. **Final result assembly.** The master assembles the output from the information passed to it by workers $wr_1, \ldots wr_R$.

# References


[1] L. Barroso, J. Dean, U. Hölzle. Web Search for a Planet: the Google Cluster Architecture. *IEEE Micro*, 23(2), pp: 22–28, April 2003.

[2] J. Dean, S. Ghemawat, MapReduce: Simplified Data Processing on Large Clusters, Proceedings, Sixth Symposium on Operating System Design and Implementation (OSDI'04), San Francisco, CA, December 2004.