MapReduce

MapReduce?

Reason	Explanation
1. Scalability	Efficiently processes terabytes/petabytes by distributing the workload.
2. Parallelism	Exploits data parallelism with minimal developer effort.
3. Fault Tolerance	Built-in recovery of failed tasks via task trackers and job schedulers.
4. Simplicity	Developers focus only on writing map() and reduce() logic.
5. Cost Efficiency	Runs on commodity hardware using open-source tools like Hadoop.
6. Data Locality	Moves computation to the data, not vice versa, saving bandwidth.
7. Ecosystem Integration	Integrates well with tools like Hive, Pig, Spark, HBase, etc.

MapReduce Framework

What is MapReduce?

- **Programming model + implementation**
- Developed by Google in 2008

> *Google*:
> A simple and powerful interface that enables **automatic parallelization and distribution of large-scale computations**, combined with an implementation of this interface that achieves high performance on **large clusters of commodity PCs**.

History and Motivation

Google PageRank problem (2003)

- How to rank tens of billions of web pages by their importance
  - … efficiently in a reasonable amount of time
  - … when data is scattered across thousands of computers
  - … data files can be enormous (terabytes or more)
  - … data files are updated only occasionally (just appended)
  - … sending the data between compute nodes is expensive
  - … hardware failures are rule rather than exception
- Centralized index structure was no longer sufficient
- Solution
  - **Google File System** – a distributed file system
  - **MapReduce** – a programming model

MapReduce Framework

MapReduce **programming model**

- Cluster of commodity personal computers (nodes)
  - Each running a host operating system, mutually interconnected within a network, communication based on IP addresses, …
- **Data is distributed among the nodes**
- **Tasks executed in parallel across the nodes**

Classification

- Process interaction: **message passing**
- Problem decomposition: **data parallelism**

Basic Idea

Divide-and-conquer paradigm

- Breaks down a given problem into simpler sub-problems
- Solutions of the sub-problems are then combined together

Two core functions

- Map **function**
  - Generates a set of so-called **intermediate key-value pairs**
- Reduce **function**
  - Reduces values associated with a given intermediate key

And that's all!

Basic Idea

And that's really all!
It means...

- We only need to **implement *Map* and *Reduce* functions**
- **Everything else** such as
  - input data distribution,
  - scheduling of execution tasks,
  - monitoring of computation progress,
  - inter-machine communication,
  - handling of machine failures,
  - ...

  is managed automatically by the framework!

Model Description

Map function

- *Input*: **input key-value pair** = input record
- *Output*: **list of intermediate key-value pairs**
  - Usually from a different domain
  - Keys do not have to be unique
  - Duplicate pairs are permitted
- $(key, value) \rightarrow$ list of $(key, value)$

Reduce function

- *Input*: **intermediate key + list of (all) values** for this key
- *Output*: **possibly smaller list of values** for this key
  - Usually from the same domain
- $(key,$ list of $values) \rightarrow (key,$ list of $values)$

Example: Word Frequency

```
/**
 * Map function
 * @param key   Document identifier
 * @param value Document contents
 */
map(String key, String value) {
  foreach word w in value: emit(w, 1);
}
```

where,

- *value* is a line of text from the input (e.g., a sentence).
- *for each word w in value*: this loops through each word in the sentence.
- *emit(w, 1)*: for every word w, output a key-value pair where:
  - the key is the word itself,
  - the value is 1, representing one occurrence of that word.

Example:
Input line (value):
"cat dog cat“
Map function output:

$\left.\begin{array}{l}(\text{"cat"}, 1)\\(\text{"dog"}, 1)\\(\text{"cat"}, 1)\end{array}\right\}$ Intermediate key value

These key-value pairs are then sent to the Reduce function, which adds up all the counts for each word.

After the map() step emits these pairs, the framework groups all values by key and sends them to reduce().

Example: Word Frequency

```
/**
 * Reduce function
 * @param key     Particular word
 * @param values List of count values generated for this word
 */
reduce(String key, Iterator values) {
  int result = 0;
  foreach v in values: result += v;
  emit(key, result);
}
```

values: A list/collection of integers associated with a specific key (from the Map step).

foreach v in values: Loop through each value.

result += v: Add each value to result.

emit(key, result): final output of that key.

For example:

reduce("cat", [1, 1]) → ("cat", 2)

reduce("dog", [1]) → ("dog", 1)

E.g., a word "cat" was emitted like this from the Map step for another sentence containing cat 3 times:

("cat", 1)

("cat", 1)

("cat", 1)

The framework groups these by key and sends key "cat" and a list of values [1, 1, 1] to Reducer.

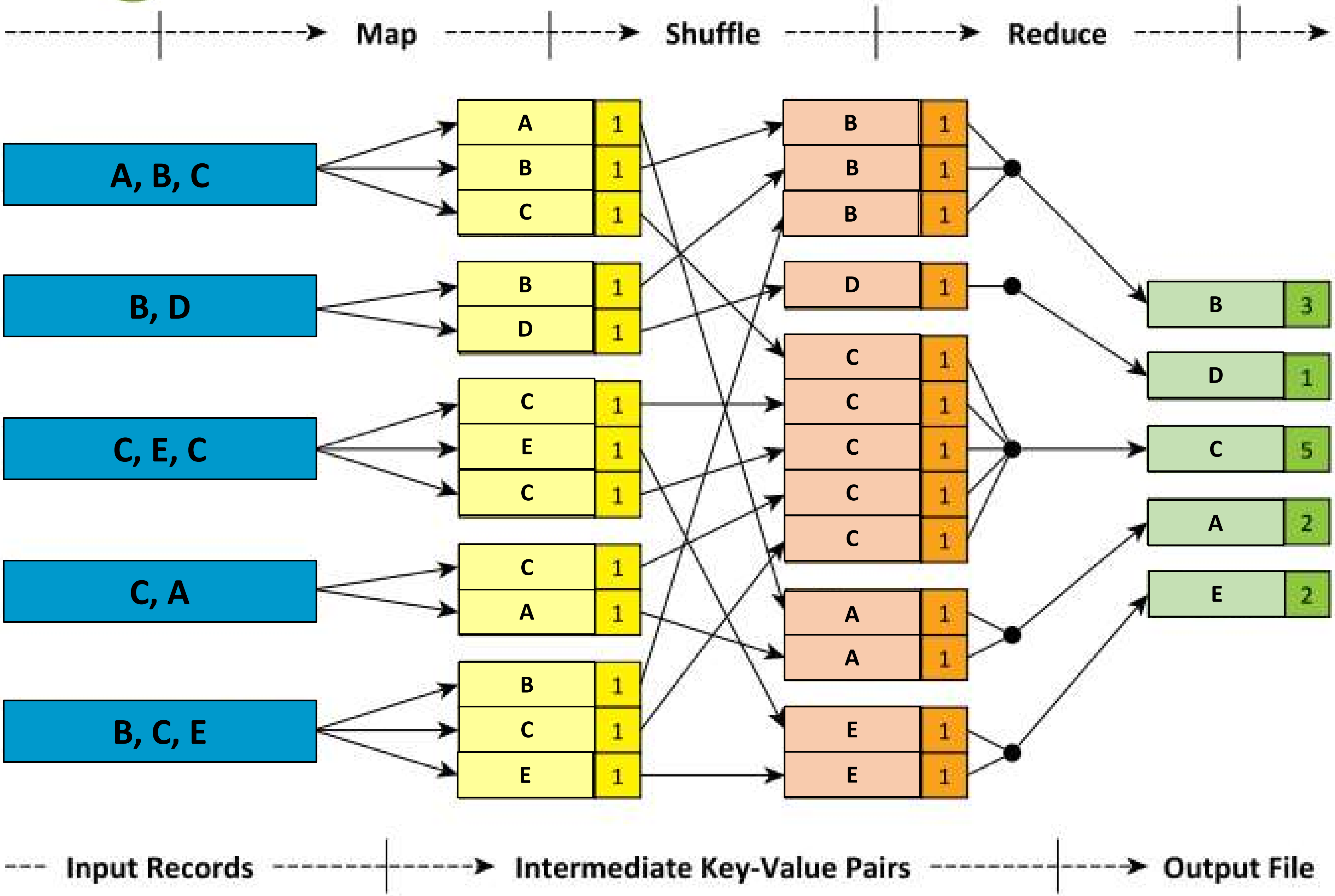
Reducer Code:

int result = 0;

foreach v in [1, 1, 1]: result += v;

emit("cat", result); **// Output: ("cat", 3)**

Logical Phases



Logical Phases

Mapping phase

- **Map function** is executed **for each input record**
- Intermediate key-value pairs are emitted

Shuffling phase

- Intermediate key-value pairs are **grouped and sorted** according to the keys

Reducing phase

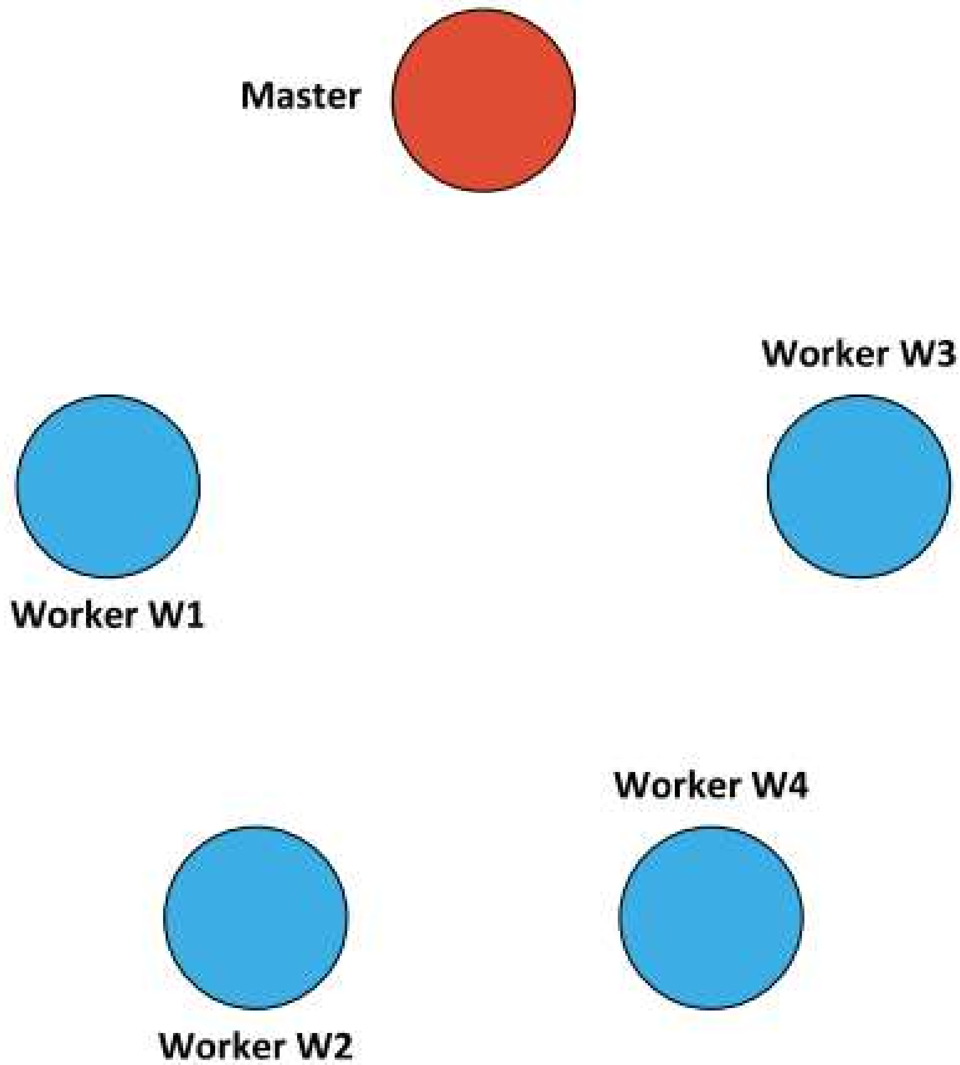
- **Reduce function** is executed **for each intermediate key**
- Output key-value pairs are generated

Cluster Architecture

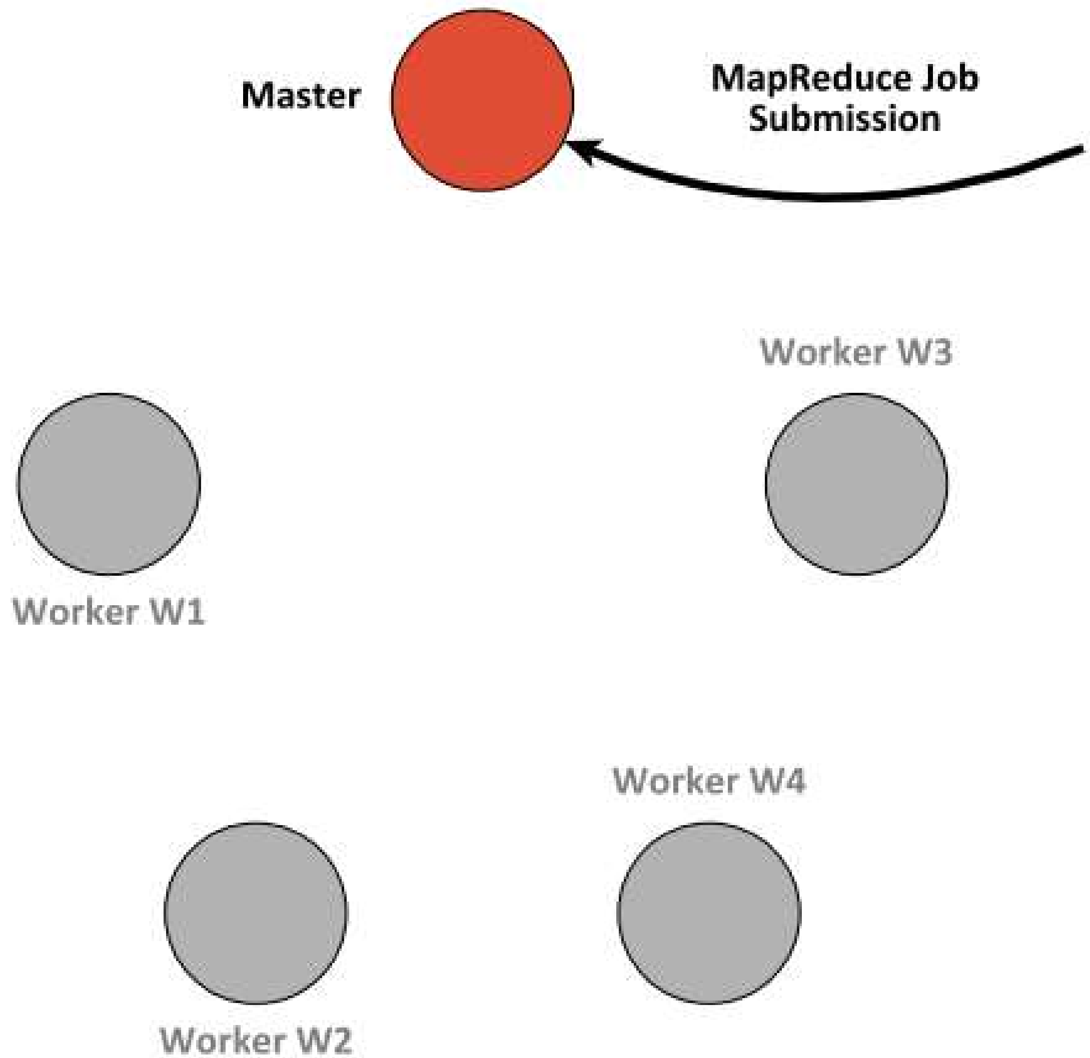
Master-slave architecture

- Two types of nodes, each with two basic roles
- **Master**
  - **Manages the execution of MapReduce jobs**
    - Schedules individual Map / Reduce tasks to idle workers
    - …
  - **Maintains metadata about input / output files**
    - These are stored in the underlying distributed file system
- **Slaves** (workers)
  - **Physically store the actual data contents of files**
    - Files are divided into smaller parts called splits
    - Each split is stored by one / or even more particular workers
  - **Accept and execute assigned Map / Reduce tasks**

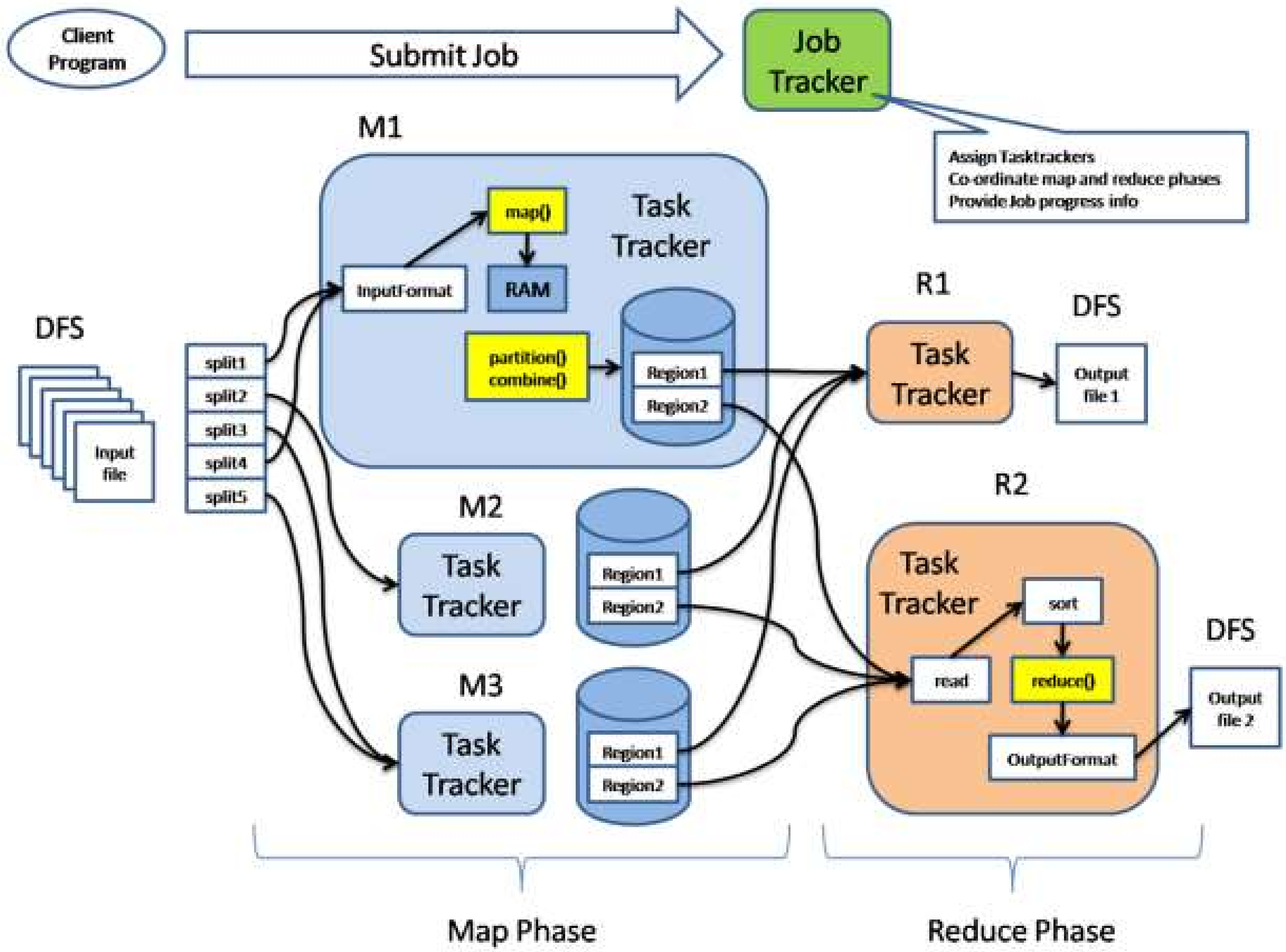
Cluster Architecture



MapReduce Job Submission



Execution Schema



MapReduce Job Submission

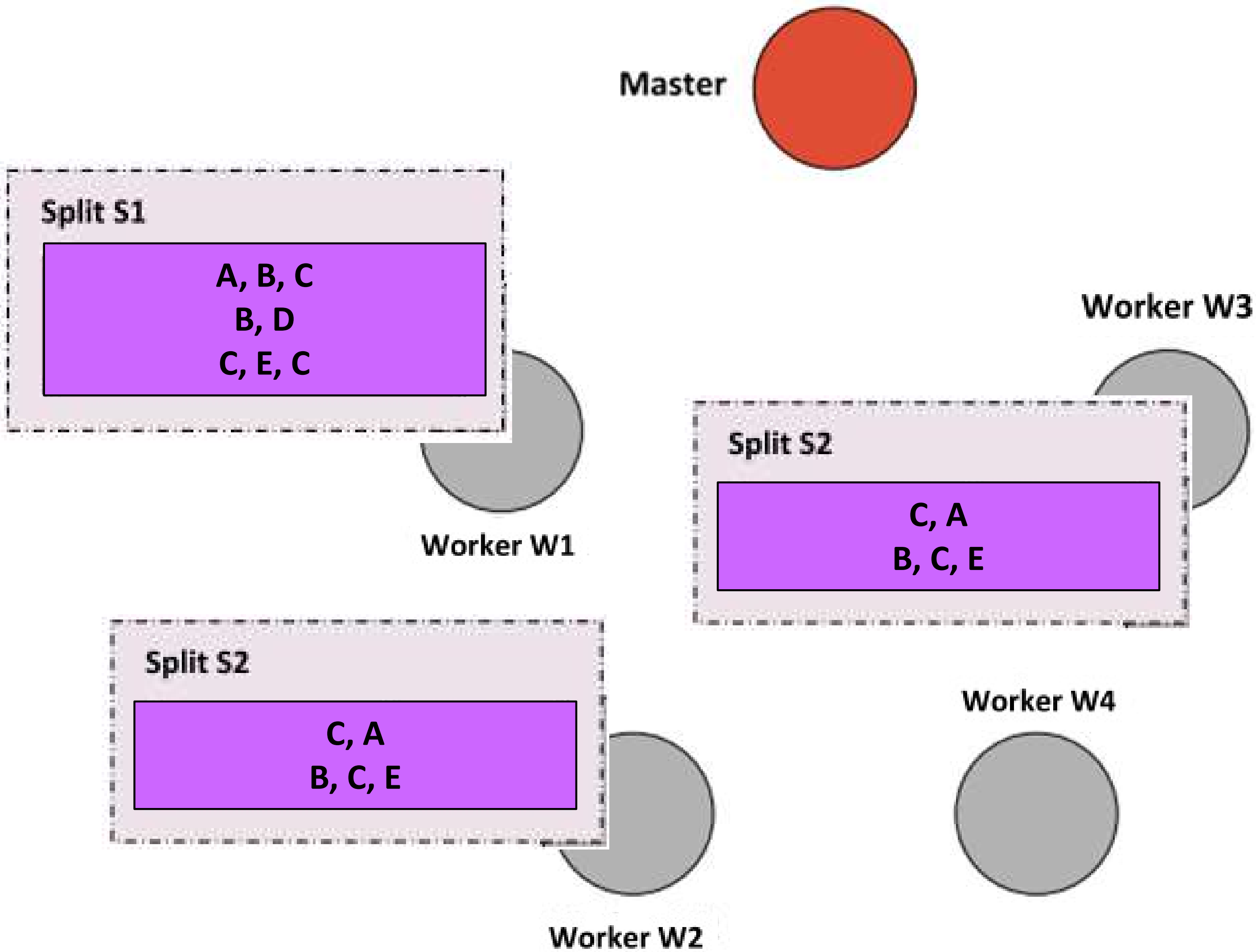
Submission of MapReduce jobs

- Jobs can only be submitted to the master node
- Client provides the following:
  - **Implementation** of (not only) **Map and Reduce functions**
  - Description of **input file** (or even files)
  - Description of **output directory**

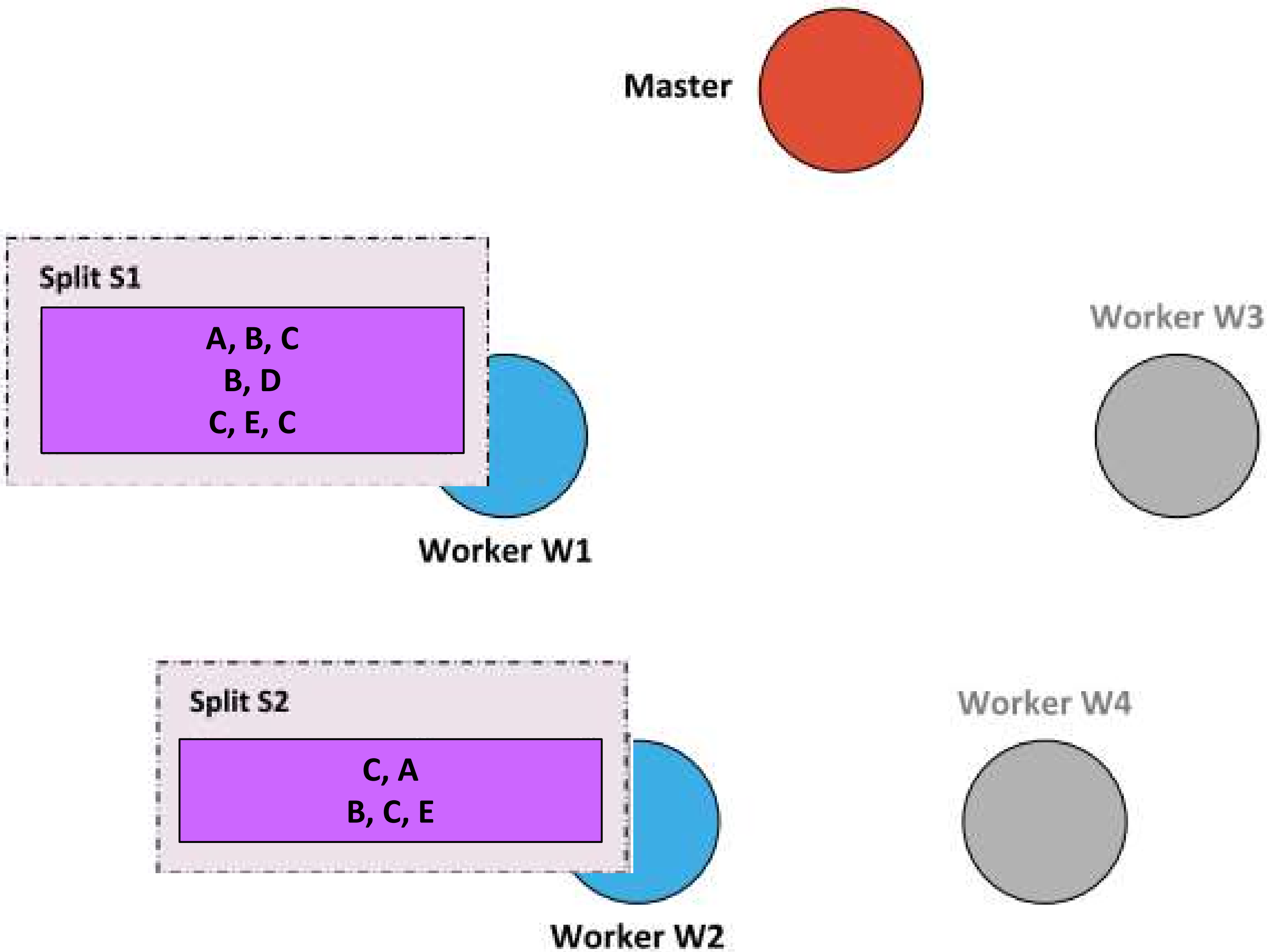
Localization of input files

- Master determines **locations of all involved splits**
  - I.e. workers containing these splits are resolved

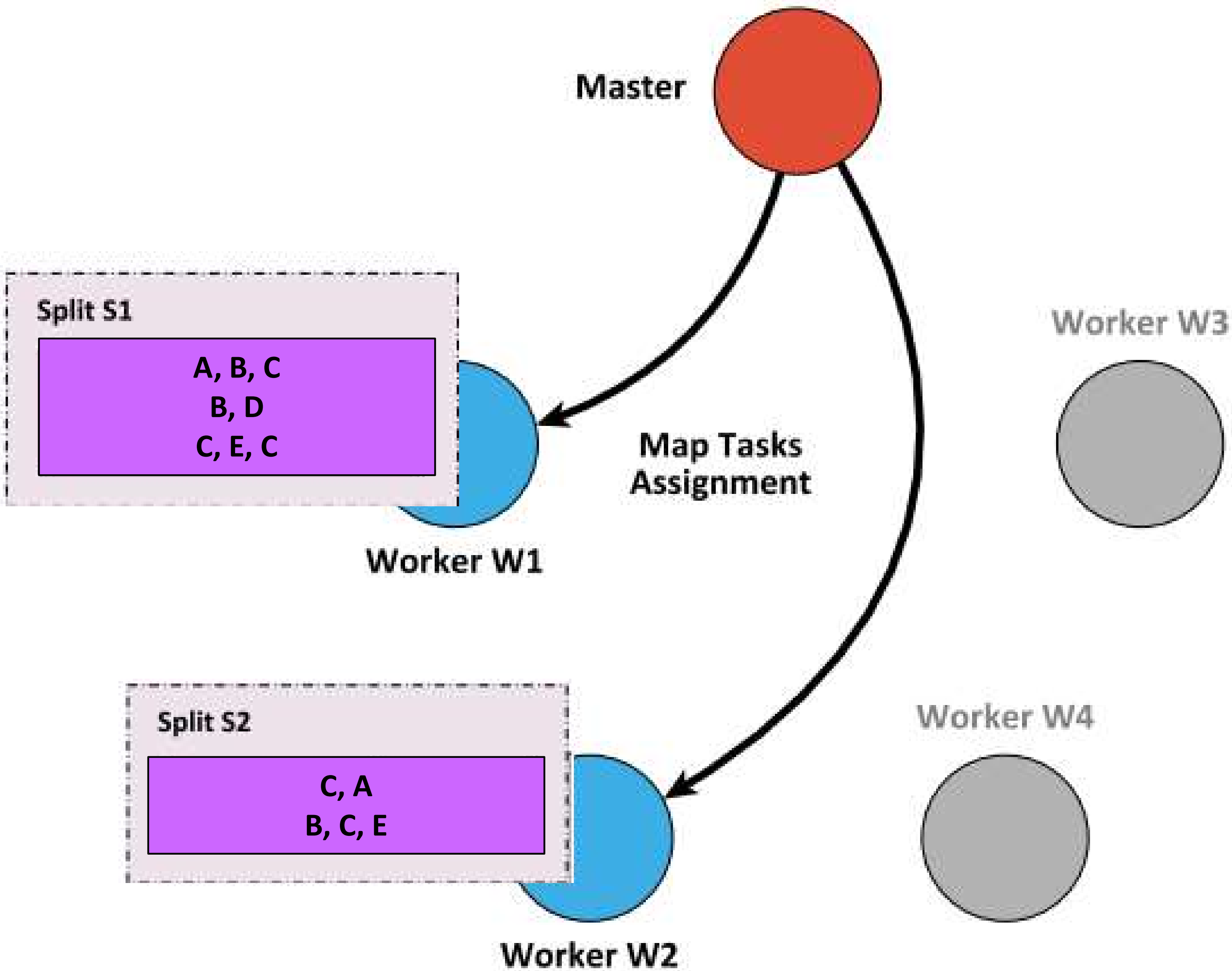
Input Splits Localization



Input Splits Localization



Map Task Assignment



Map Task Execution

Map Task = **processing of 1 split by 1 worker**

- Assigned by the master to an idle worker that is (preferably) already containing (physically storing) a given split

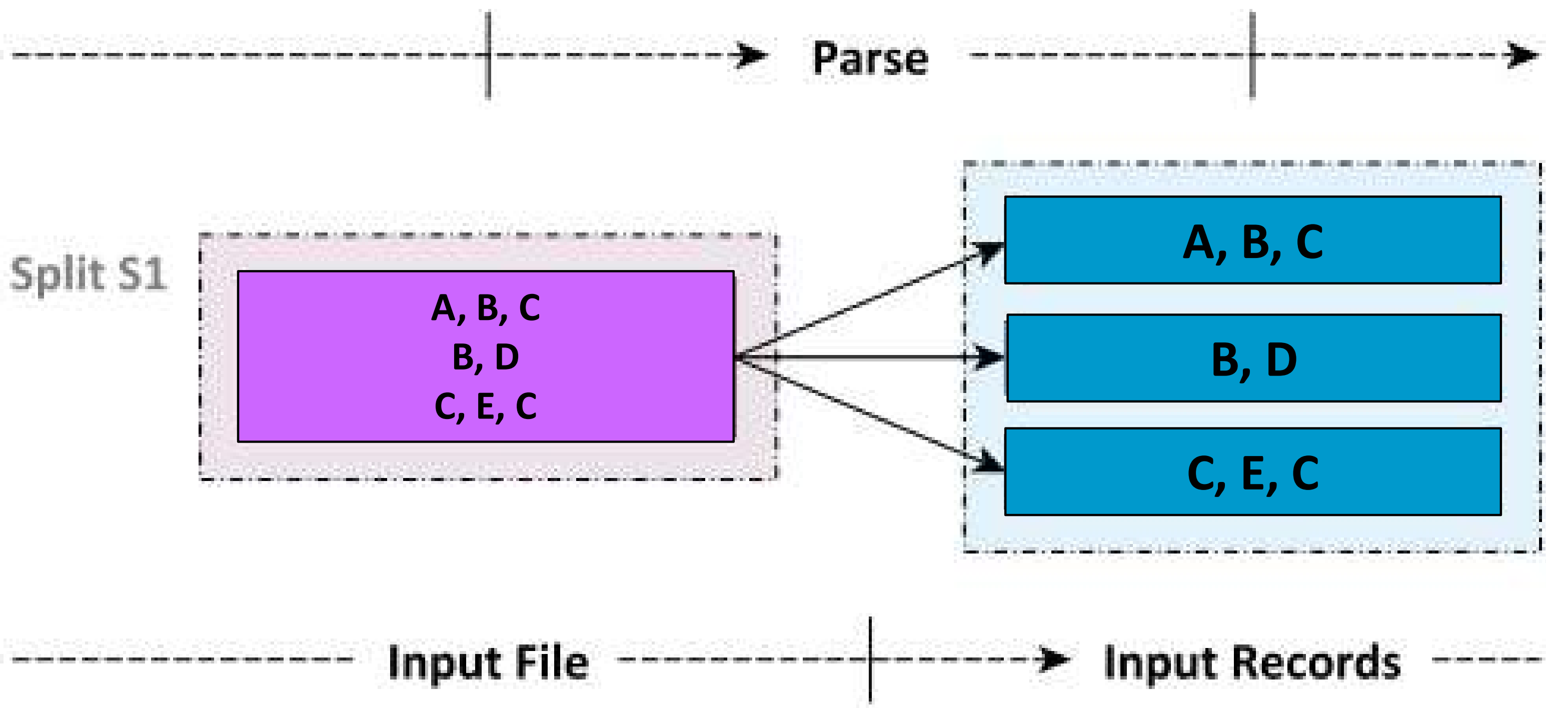
Individual steps…

- Input reader is used to **parse contents of the split**
  - I.e. **input records are generated**
- **Map function is applied on each input record**
  - Intermediate key-value pairs are emitted
- These pairs are **stored <u>locally</u> and organized into** regions
  - Either in the system memory,
    or flushed to a local hard drive when necessary
  - **Partition function** is used to determine the intended region
    - Intermediate <u>keys</u> (not values) are used for this purpose
    - E.g. hash of the key modulo the overall number of reducers

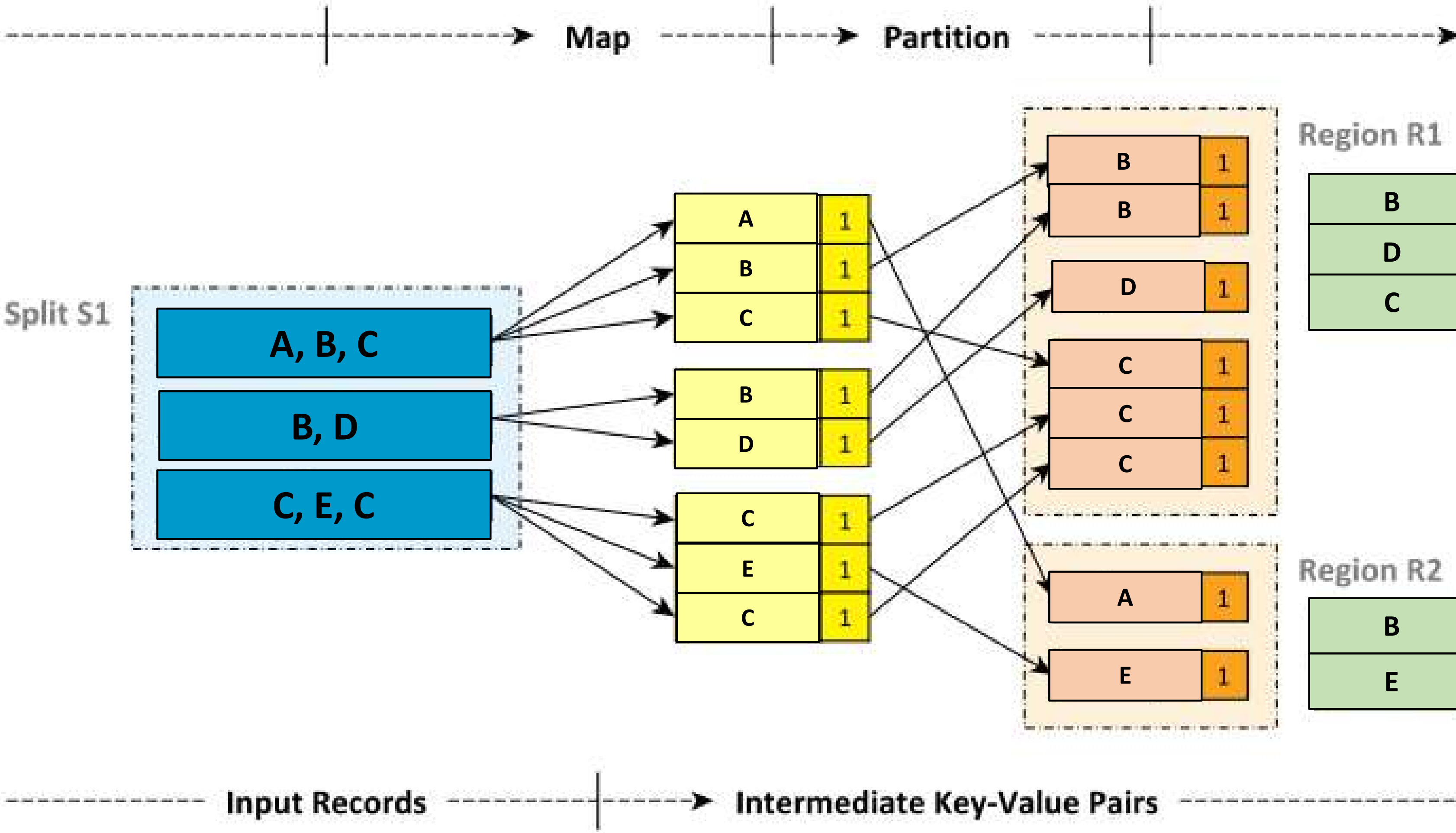
Input Parsing

Parsing phase

- **Each split is parsed** so that **input records are retrieved** (i.e. input key-value pairs are obtained)

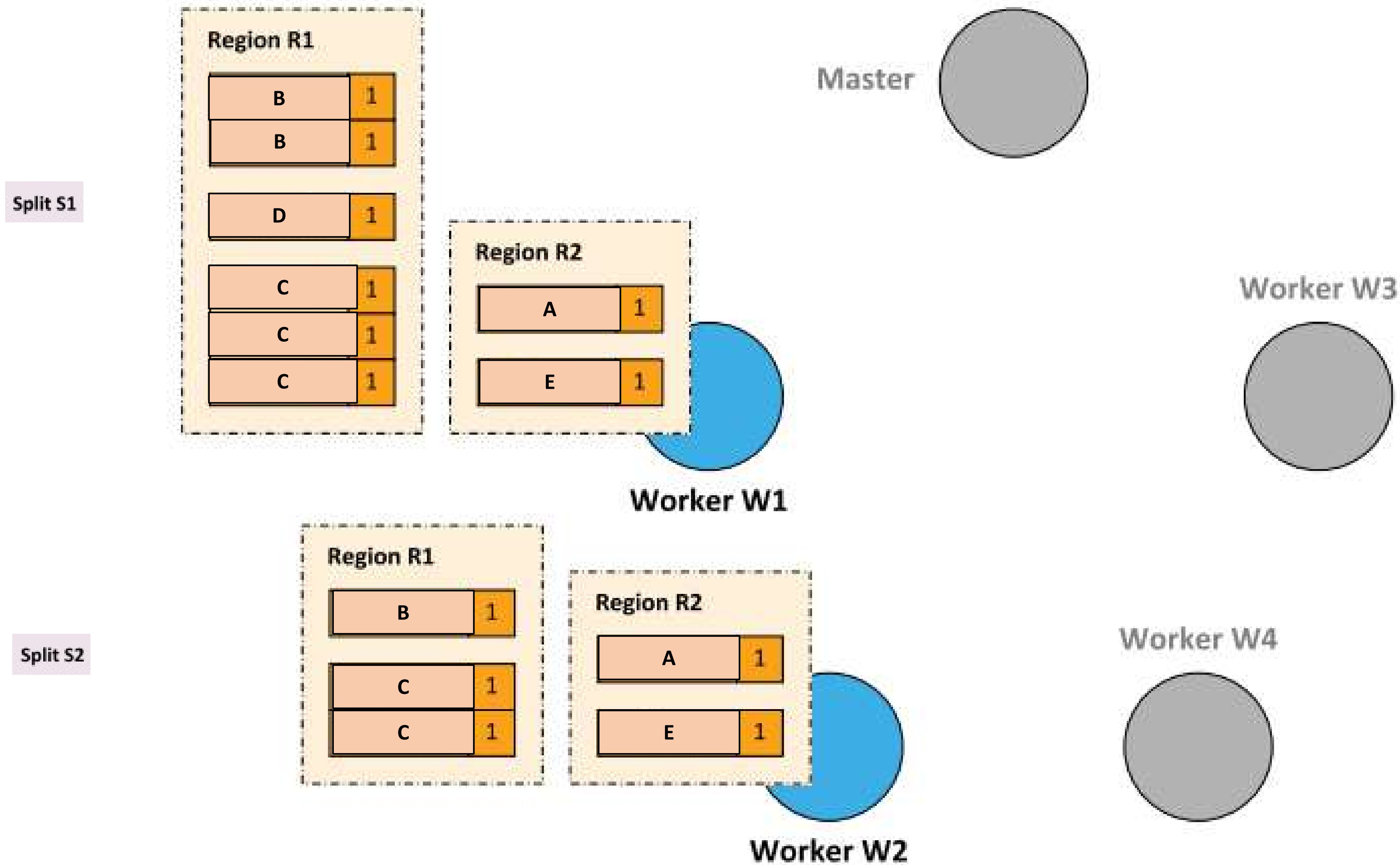


Map Phase



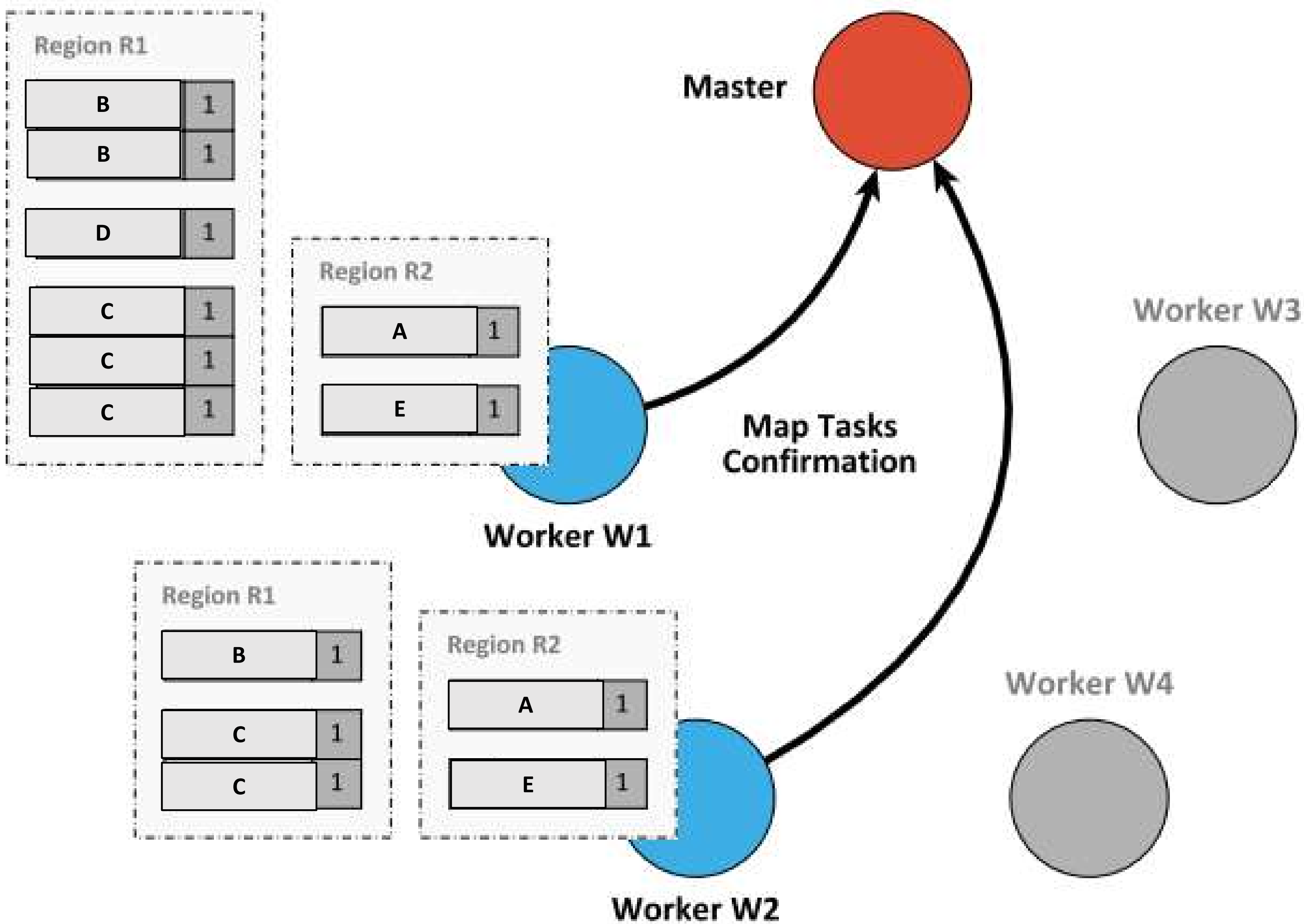
Map
Partition
Split S1
A, B, C
B, D
C, E, C
A 1
B 1
C 1
B 1
D 1
C 1
E 1
C 1
B 1
B 1
D 1
C 1
C 1
C 1
A 1
E 1
Region R1
B
D
C
Region R2
B
E
Input Records
Intermediate Key-Value Pairs

Map Phase

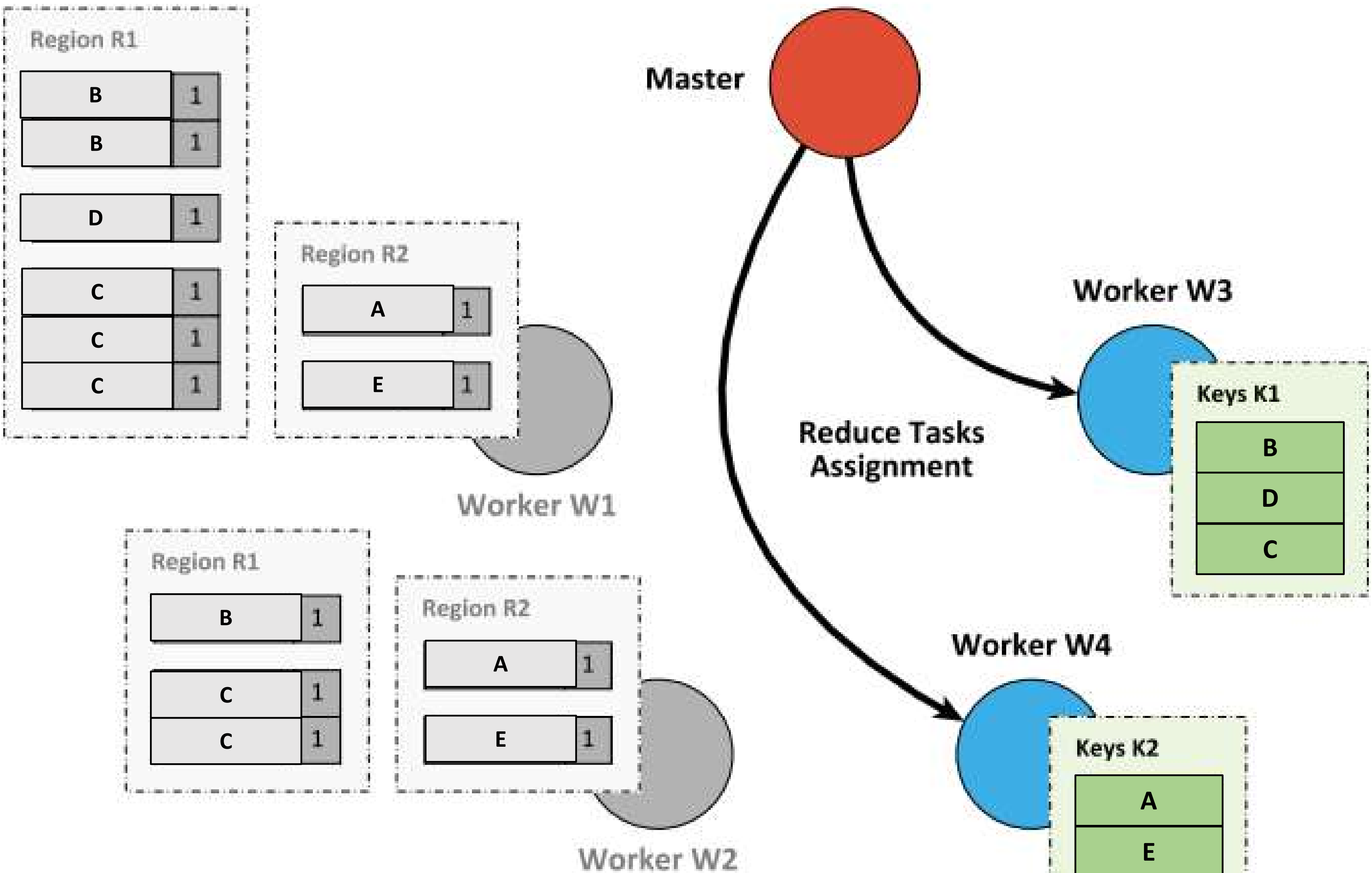


Split S1
Region R1
B 1
B 1
D 1
C 1
C 1
C 1
Region R2
A 1
E 1
Worker W1
Master
Worker W3
Split S2
Region R1
B 1
C 1
C 1
Region R2
A 1
E 1
Worker W2
Worker W4

Map Task Confirmation



Reduce Task Assignment



Reduce Task Execution

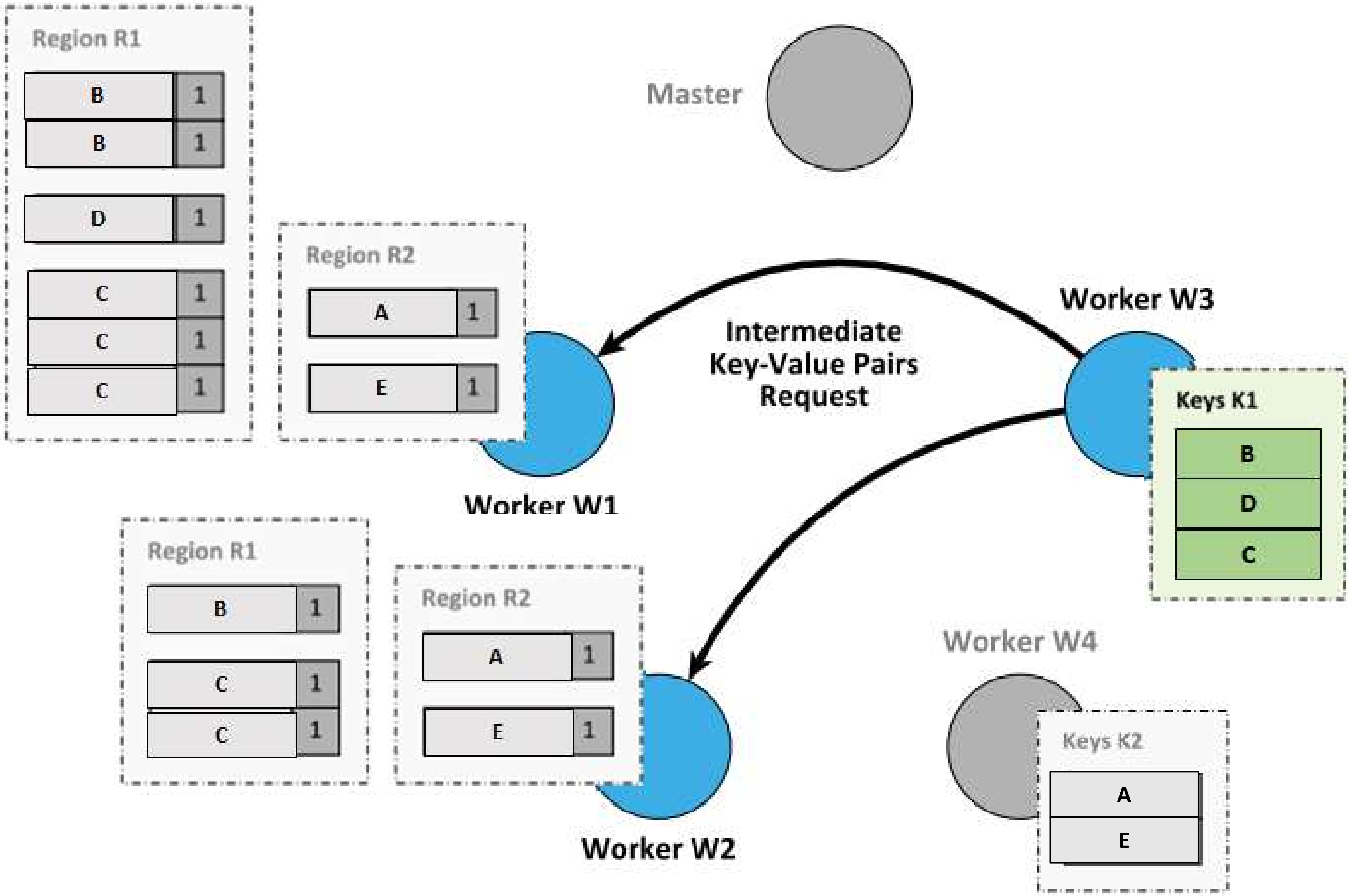
Reduce Task = reduction of selected key-value pairs by 1 worker

- Goal: processing of all emitted **intermediate key-value pairs belonging to a particular region**

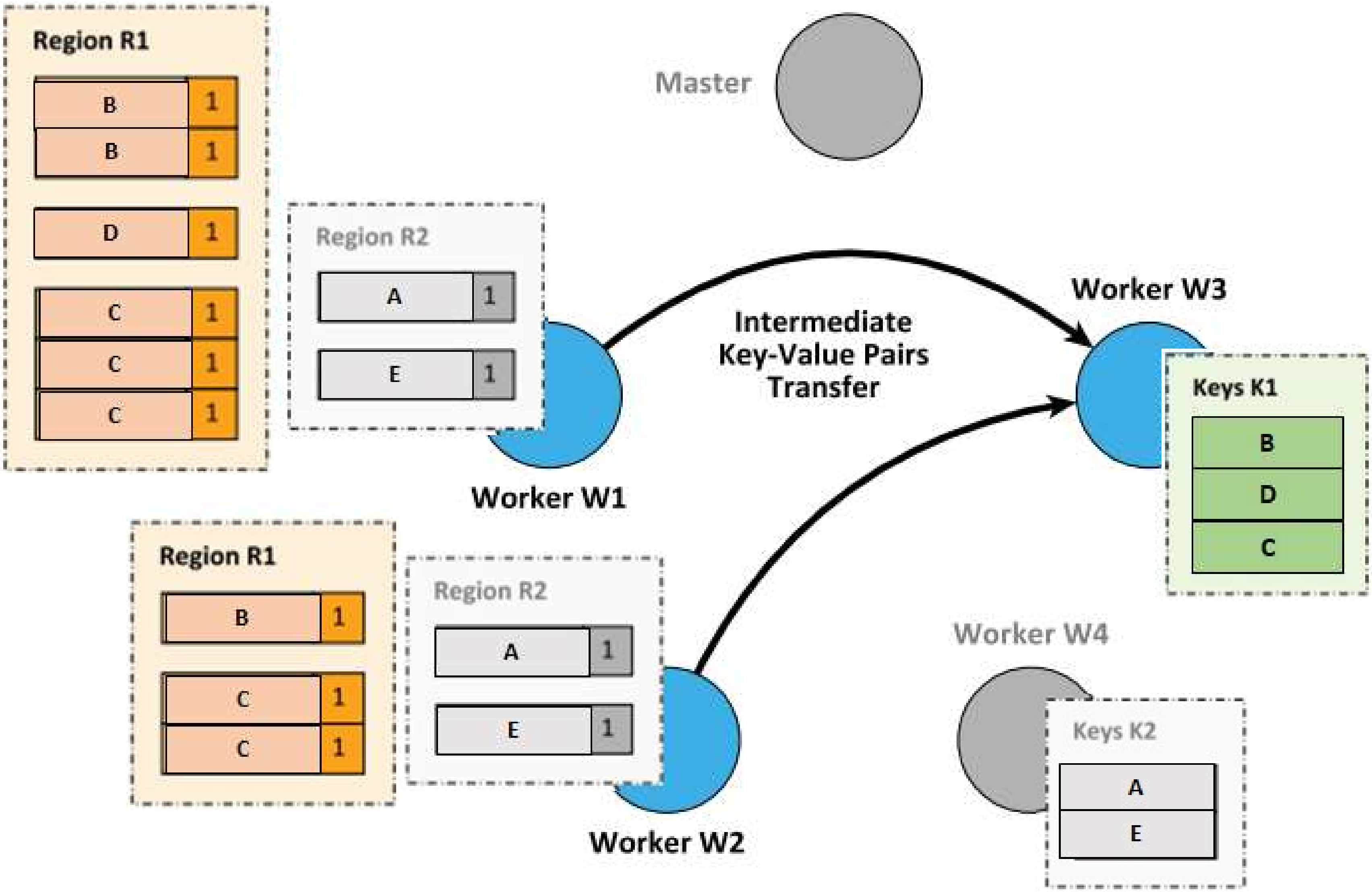
Individual steps…

- **Intermediate key-value pairs are first acquired**
  - All relevant mapping workers are addressed
  - Data of corresponding **regions are transfered** (remote read)
- Once downloaded, they are **locally merged**
  - I.e. sorted and grouped based on keys
- **Reduce function** is applied on each intermediate key
- **Output key-value pairs** are emitted and stored (output writer)
  - Note that each worker produces its own separate output file

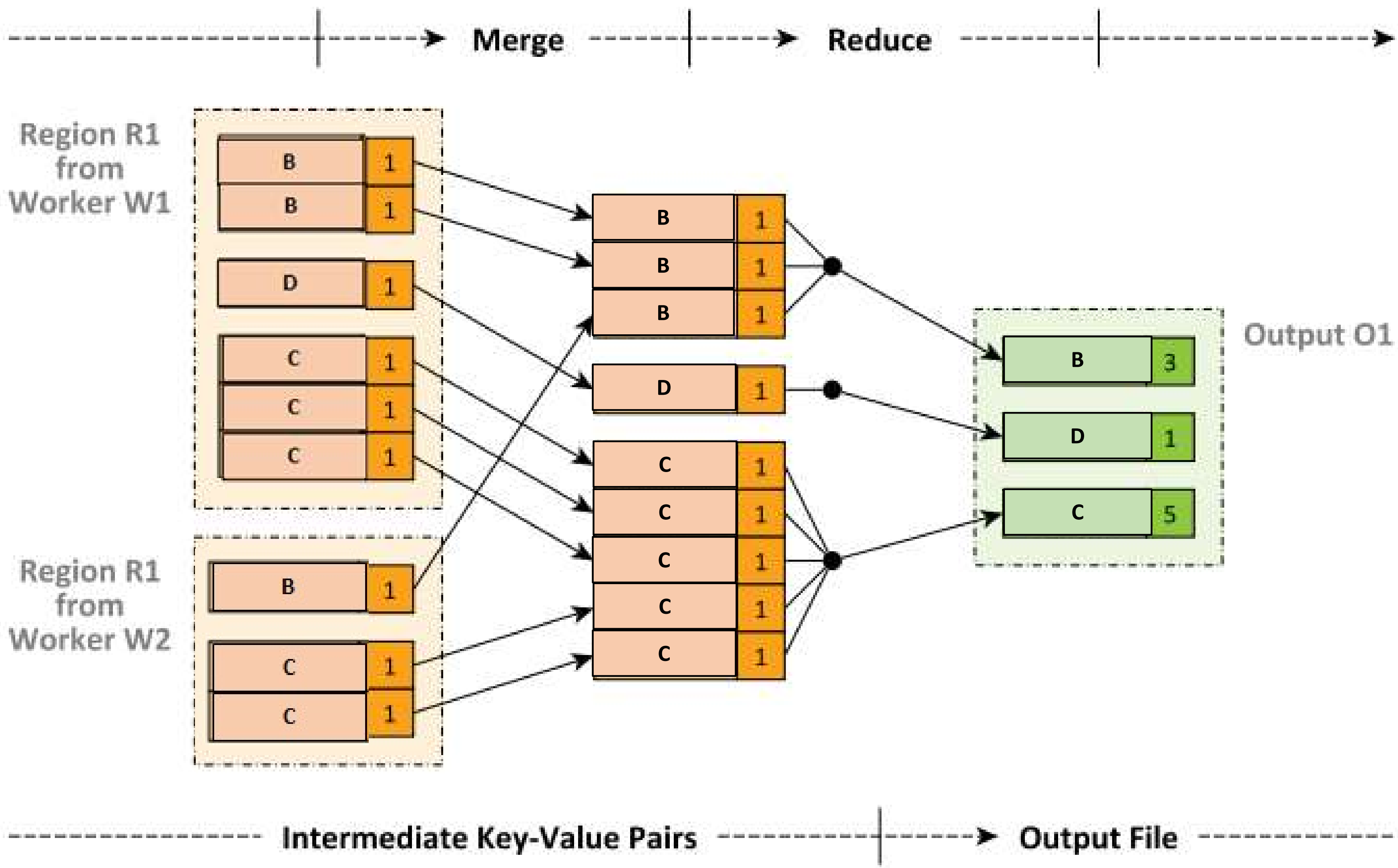
Region Data Retrieval



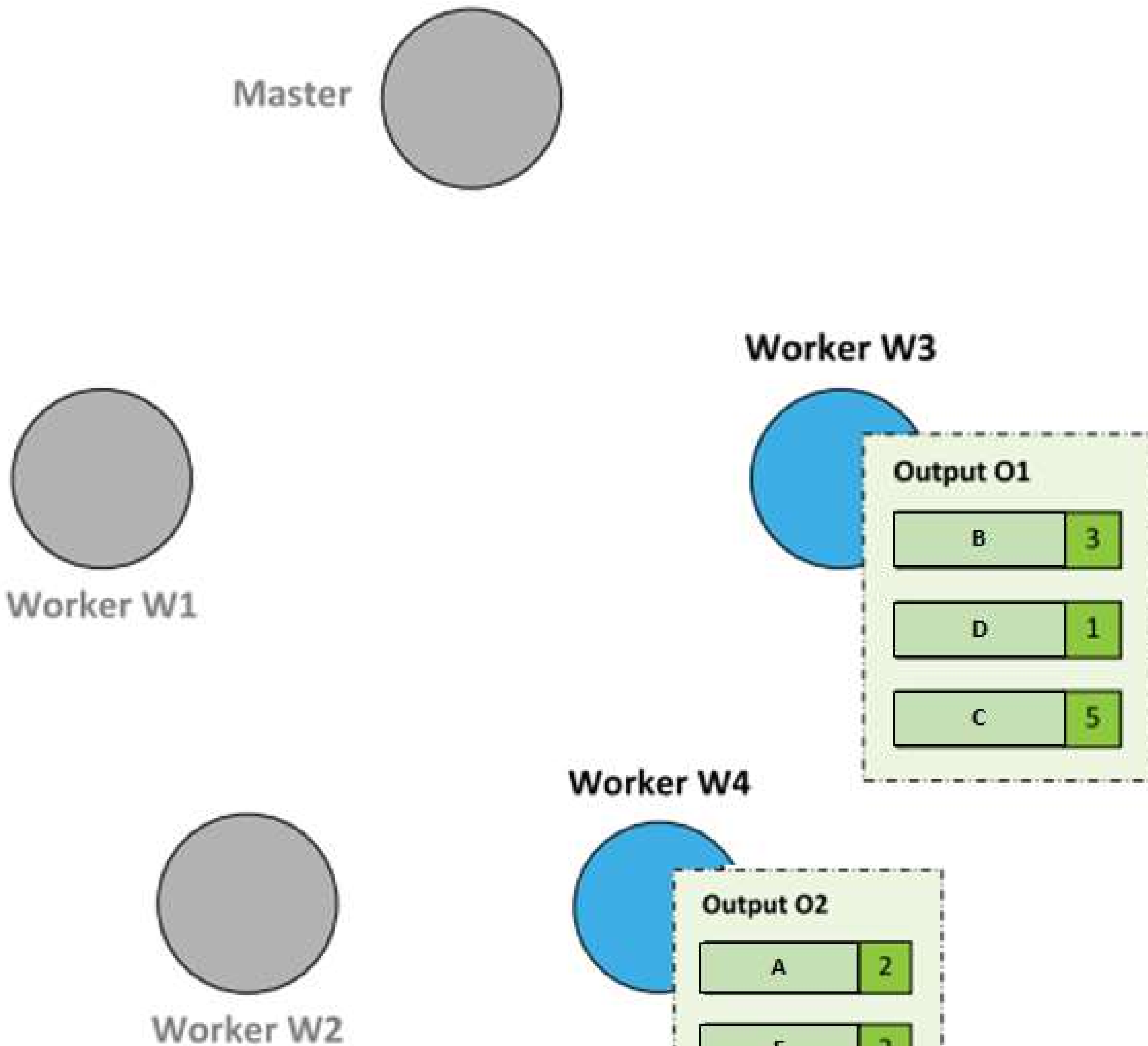
Region Data Retrieval



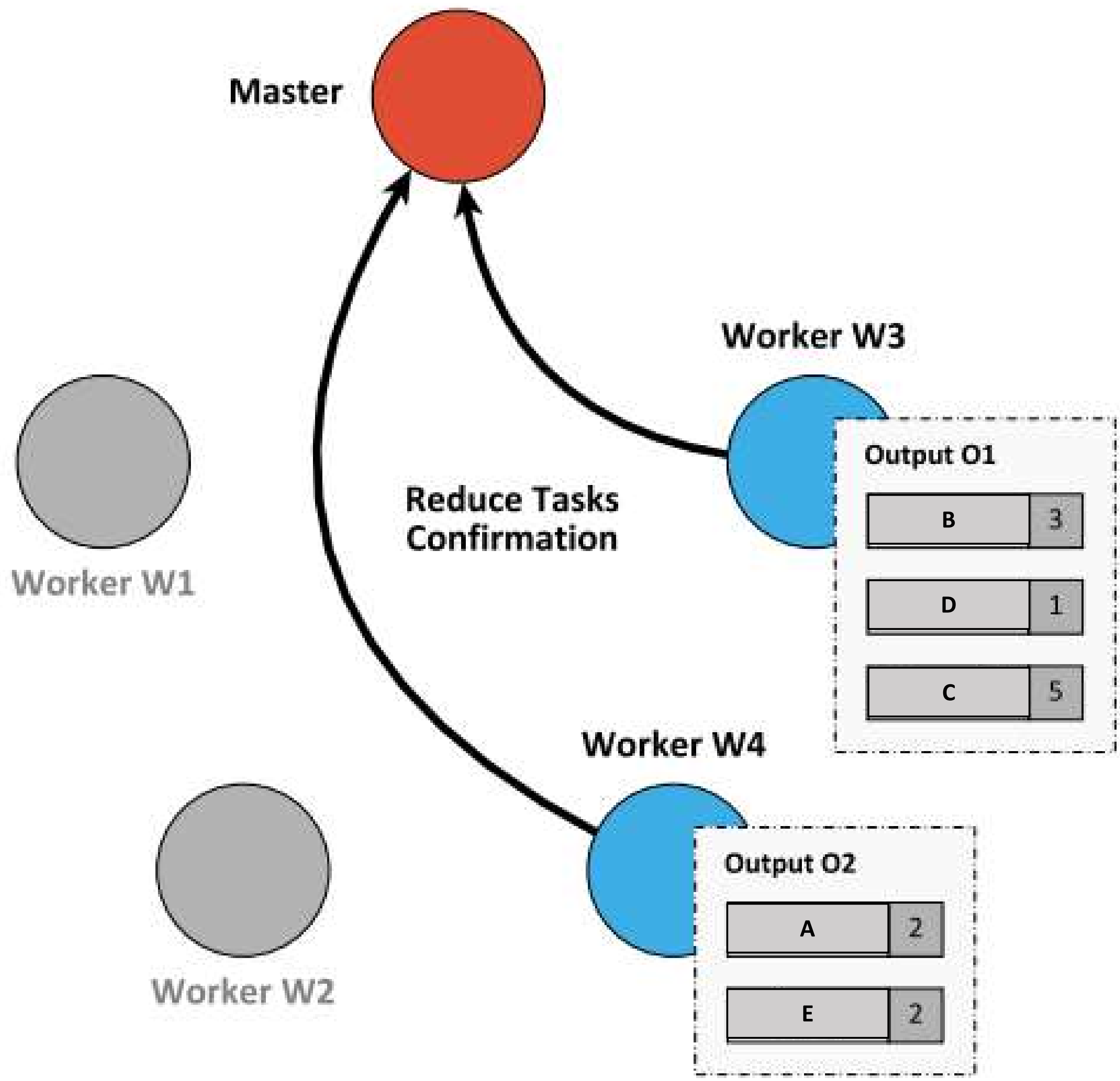
Reduce Phase



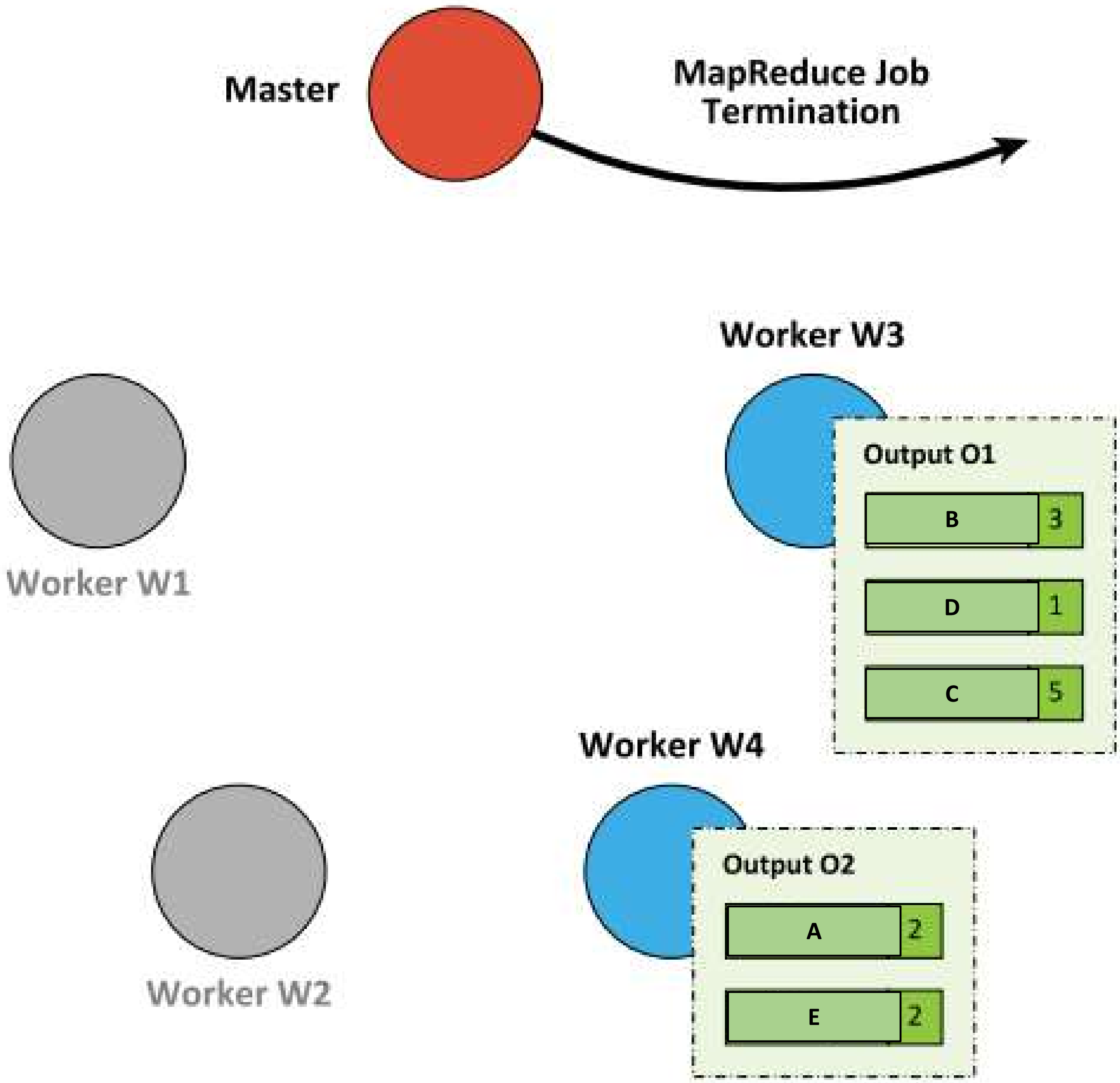
Reduce Phase



Reduce Task Confirmation



MapReduce Job Termination



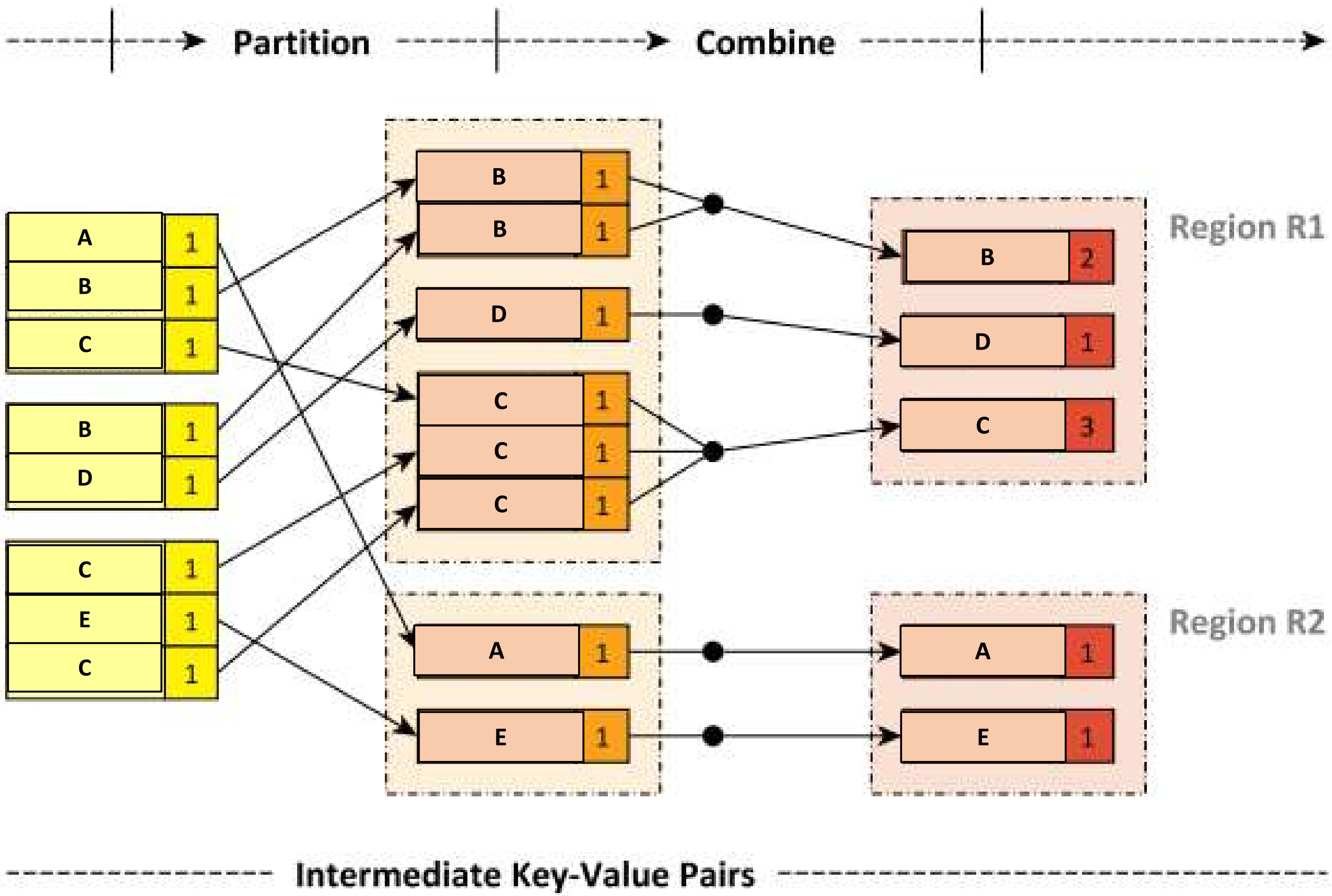
Combine Function

Optional Combine **function**

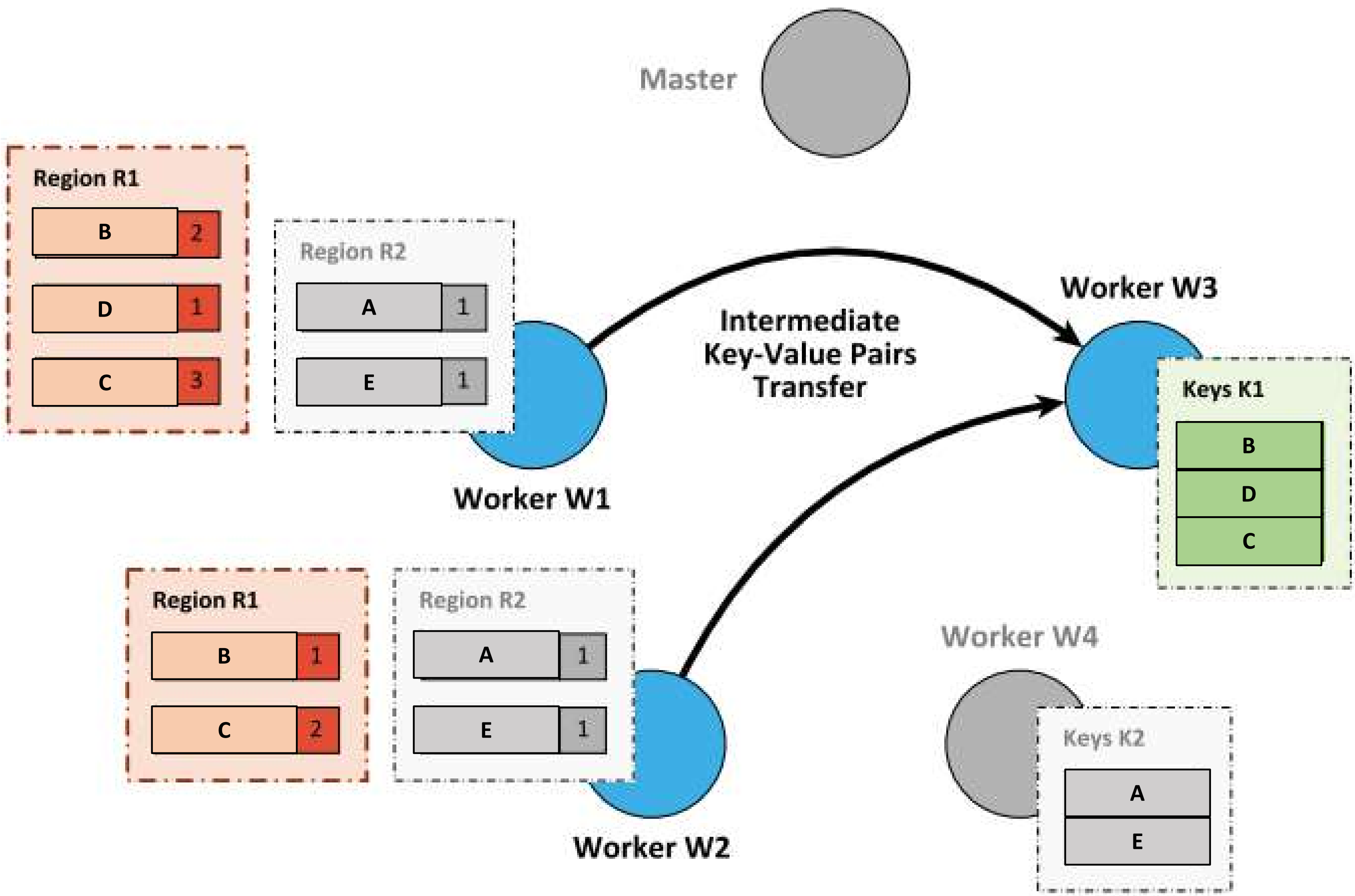
- Objective
  - **Decrease the amount of intermediate data**
    i.e. decrease the amount of data that is needed to be transferred from Mappers to Reducers
- Analogous purpose and implementation to **Reduce function**

- **Executed locally by Mappers**
- However, only applicable when the reduction is…
  - **Commutative**
  - **Associative**
  - **Idempotent**: $f(f(x)) = f(x)$

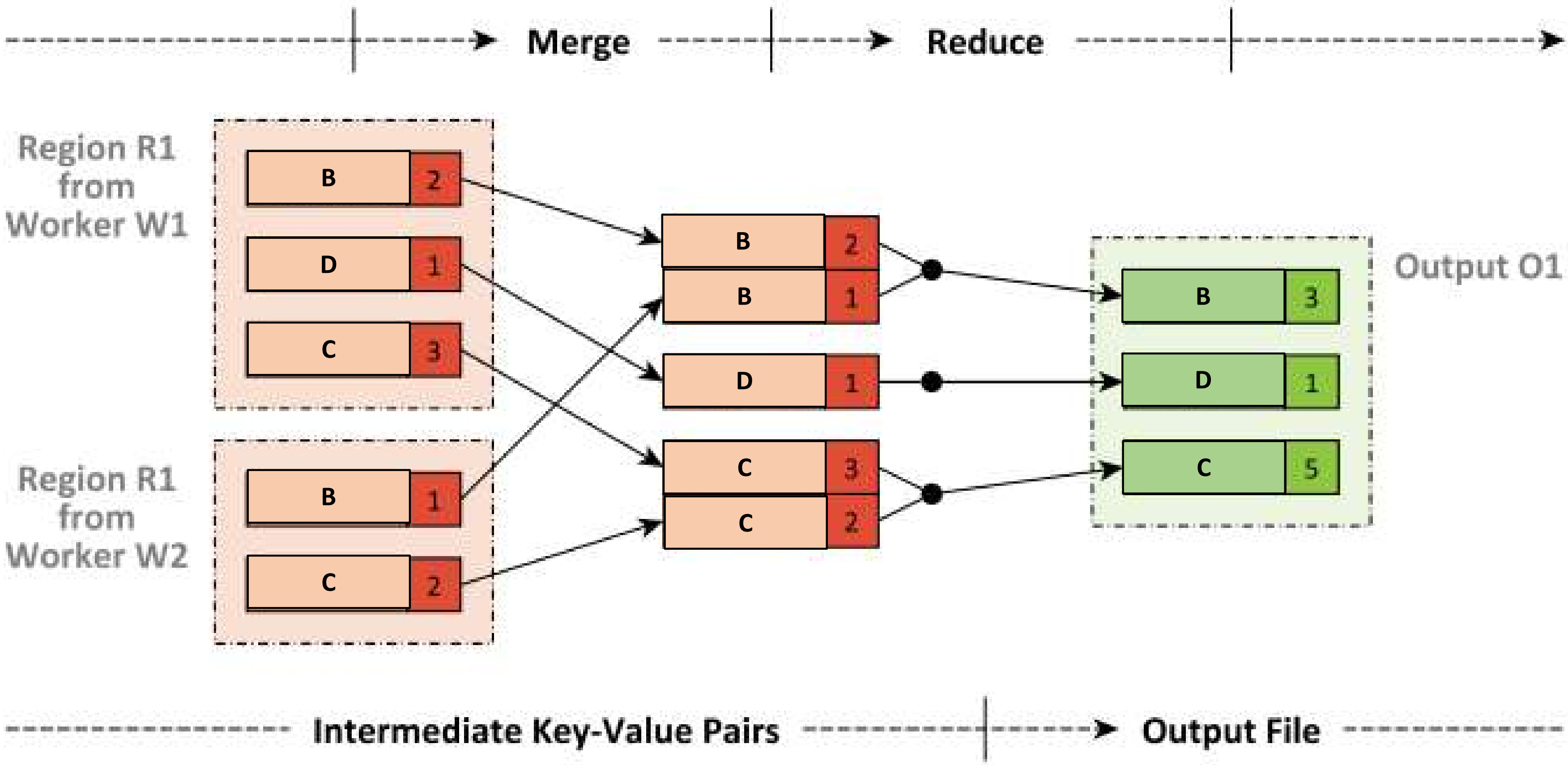
Improved Map Phase



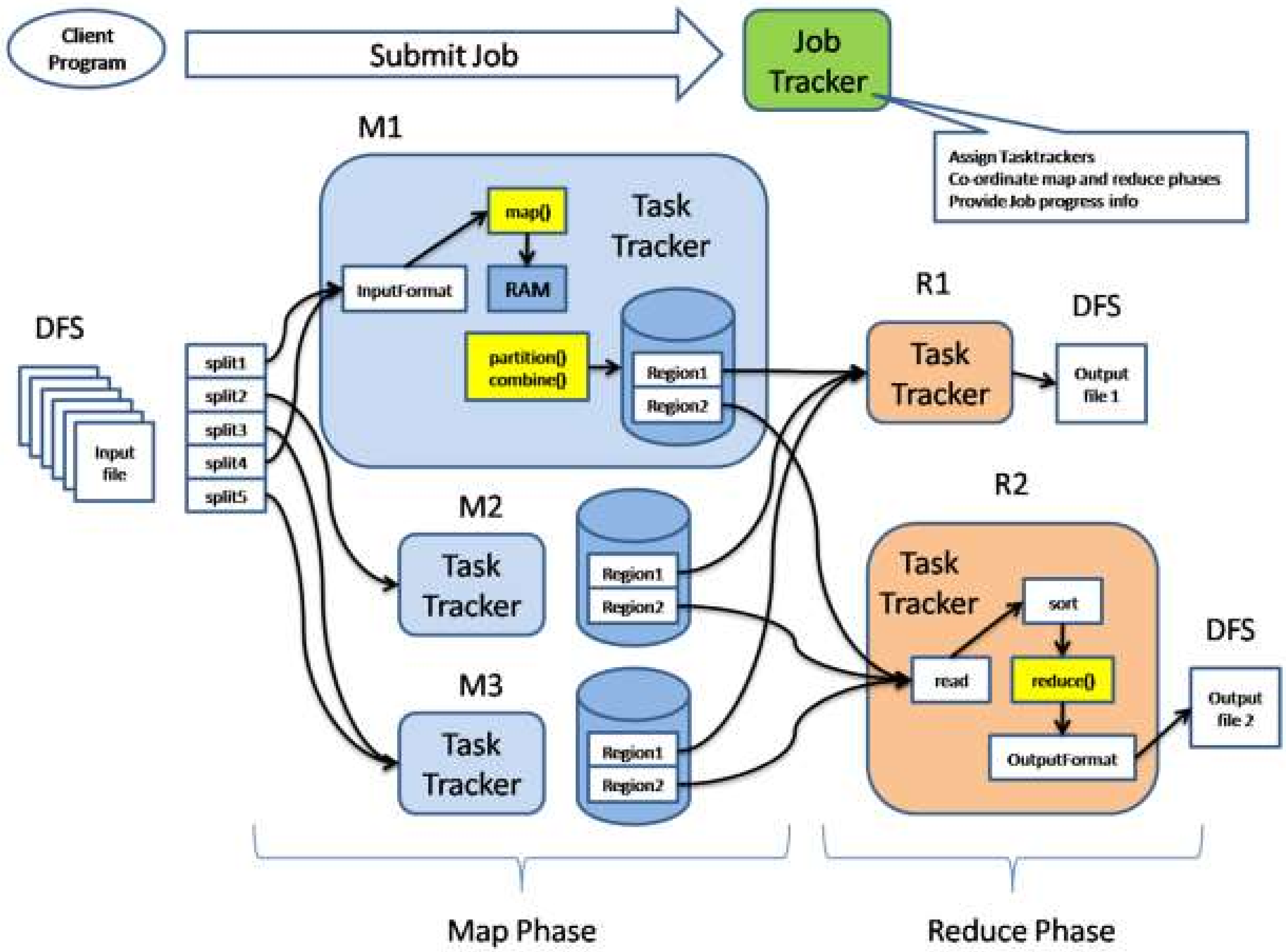
Improved Map Phase



Improved Map Phase



Execution Schema



Functions Overview

Input reader

- Parses a given input split and **prepares input records**

Map function

Partition function

- **Determines a particular Reducer** for a given intermediate key

Compare function

- Mutually **compares two intermediate keys**

Combine function

Reduce function

Output writer

- **Writes the output** of a given Reducer

Java Interface

Mapper class

- Implementation of the **map function**
- Template parameters
  - `KEYIN`, `VALUEIN` – types of input key-value pairs
  - `KEYOUT`, `VALUEOUT` – types of intermediate key-value pairs
- Intermediate pairs are emitted via `context.write(k, v)`

```
class MyMapper extends Mapper<KEYIN, VALUEIN, KEYOUT, VALUEOUT> {
  @Override
  public void map(KEYIN key, VALUEIN value, Context context)
    throws IOException, InterruptedException
  {
    // Implementation
  }
}
```

Java Interface

Reducer class

- Implementation of the **reduce function**
- Template parameters
  - `KEYIN`, `VALUEIN` – types of intermediate key-value pairs
  - `KEYOUT`, `VALUEOUT` – types of output key-value pairs
- Output pairs are emitted via `context.write(k, v)`

```
class MyReducer extends Reducer<KEYIN, VALUEIN, KEYOUT, VALUEOUT> {
  @Override
  public void reduce(KEYIN key, Iterable<VALUEIN> values, Context context)
    throws IOException, InterruptedException
  {
    // Implementation
  }
}
```

Example

Word Frequency

- *Input*: Documents with words
  - Files located at `/home/input` HDFS directory
- *Map*: parses a document, emits (word, 1) pairs
- *Reduce*: computes and emits the sum of the associated values
- *Output*: overall number of occurrences for each word
  - Output will be written to `/home/output`

MapReduce **job execution**

```
hadoop jar wc.jar WordCount /home/input /home/output
```

Example: Mapper Class

```
public class WordCount {
  …
  public static class MyMapper
    extends Mapper<Object, Text, Text, IntWritable>
  {
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();
    @Override
    public void map(Object key, Text value, Context context)
      throws IOException, InterruptedException
    {
      StringTokenizer itr = new StringTokenizer(value.toString());
      while (itr.hasMoreTokens()) {
        word.set(itr.nextToken());
        context.write(word, one);
      }
    }
  }
  …
}
```

Example: Reducer Class

```
public class WordCount {
  …
  public static class MyReducer
    extends Reducer<Text, IntWritable, Text, IntWritable>
  {
    private IntWritable result = new IntWritable();
    @Override
    public void reduce(Text key, Iterable<IntWritable> values,
      Context context) throws IOException, InterruptedException
    {
      int sum = 0;
      for (IntWritable val : values) {
        sum += val.get();
      }
      result.set(sum);
      context.write(key, result);
    }
  }
  …
}
```

Advanced Aspects

Counters

- Allow to track the progress of a MapReduce job in real time
  - **Predefined counters**
    - E.g. numbers of launched / finished Map / Reduce tasks, parsed input key-value pairs, …
  - **Custom counters** (user-defined)
    - Can be associated with any action that a Map or Reduce function does

Fault tolerance

- When a large number of nodes process a large number of data
  $\Rightarrow$ **fault tolerance is necessary**

Advanced Aspects

Worker failure

- Master periodically pings every worker; if no response is received in a certain amount of time, master marks the worker as failed
- **All its tasks are reset back to their initial idle state and become eligible for rescheduling on other workers**

Master failure

- Strategy A – periodic checkpoints are created; if master fails, a new copy can then be started
- Strategy B – master failure is considered to be highly unlikely; users simply resubmit unsuccessful jobs

Advanced Aspects

Stragglers

- Straggler = **node that takes unusually long time to complete a task it was assigned**

- Solution
  - When a MapReduce job is close to completion, the master schedules backup executions of the remaining in-progress tasks
  - A given task is considered to be completed whenever either the primary or the backup execution completes

Additional Examples

URL access frequency

- *Input*: HTTP server access logs
- *Map*: parses a log, emits (accessed URL, 1) pairs
- *Reduce*: computes and emits the sum of the associated values
- *Output*: overall number of accesses to a given URL

Inverted index

- *Input*: text documents containing words
- *Map*: parses a document, emits (word, document ID) pairs
- *Reduce*: emits all the associated document IDs sorted
- *Output*: list of documents containing a given word

Additional Examples

Distributed sort

- *Input*: records to be sorted according to a specific criterion
- *Map*: extracts the sorting key, emits (key, record) pairs
- *Reduce*: emits the associated records unchanged

Reverse web-link graph

- *Input*: web pages with `<a href="…">…</a>` tags
- *Map*: emits (target URL, current document URL) pairs
- *Reduce*: emits the associated source URLs unchanged
- *Output*: list of URLs of web pages targeting a given one

Additional Examples

Reverse web-link graph

```
/**
 * Map function
 * @param key   Source web page URL
 * @param value HTML contents of this web page
 */
map(String key, String value) {
  foreach <a> tag t in value: emit(t.href, key);
}
```

```
/**
 * Reduce function
 * @param key    URL of a particular web page
 * @param values List of URLs of web pages targeting this one
 */
reduce(String key, Iterator values) {
  emit(key, values);
}
```

Use Cases: General Patterns

Counting, summing, aggregation

- When the overall number of occurrences of certain items or a different aggregate function should be calculated

Collating, grouping

- When all items belonging to a certain group should be found, collected together or processed in another way

Filtering, querying, parsing, validation

- When all items satisfying a certain condition should be found, transformed or processed in another way

Sorting

- When items should be processed in a particular order with respect to a certain ordering criterion

Use Cases: Real-World Problems

Just a few **real-world examples**…

- Risk modeling, customer churn
- Recommendation engine, customer preferences
- Advertisement targeting, trade surveillance
- Fraudulent activity threats, security breaches detection
- Hardware or sensor network failure prediction
- Search quality analysis
- …

Real world Problems

Use Case	Description
1. Word Count / Log Analysis	Counting frequency of words or log entries from large-scale documents/logs.
2. Indexing Web Pages	Creating inverted indexes (like in search engines) from web crawled data.
3. Recommendation Engines	Used in collaborative filtering for e-commerce/movie platforms.
4. Sentiment Analysis	Extracting and aggregating sentiments from large text datasets (e.g., tweets, reviews).
5. Network Traffic Monitoring	Analyzing logs from distributed network devices for trends or threats.
6. Genomics / DNA Processing	Processing and comparing millions of DNA sequences efficiently.
7. Financial Risk Modeling	Large-scale simulation and aggregation of financial datasets.

MapReduce criticism

- MapReduce **is a step backwards**
  - Does not use database schema
  - Does not use index structures
  - Does not support advanced query languages
  - Does not support transactions, integrity constraints, views, …
  - Does not support data mining, business intelligence, …
- MapReduce **is not novel**
  - Ideas more than 20 years old and overcome
  - Message Passing Interface (MPI), Reduce-Scatter

The end of MapReduce?