Industrial Machine Learning on Hadoop and Spark

Maks Nakhodnov, Bremen 2025

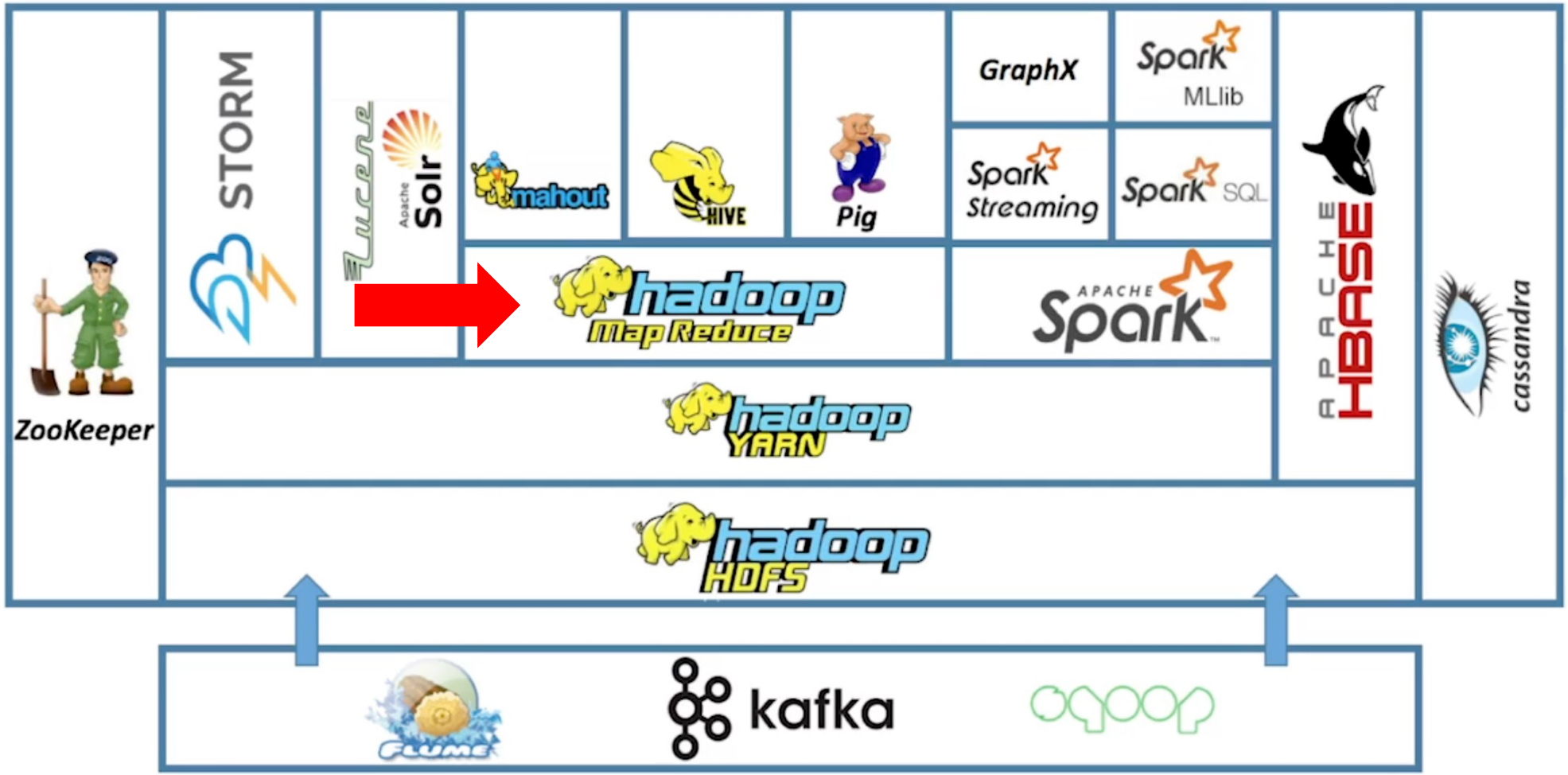
Distributed computing. MapReduce

Plan:

How to process big data?

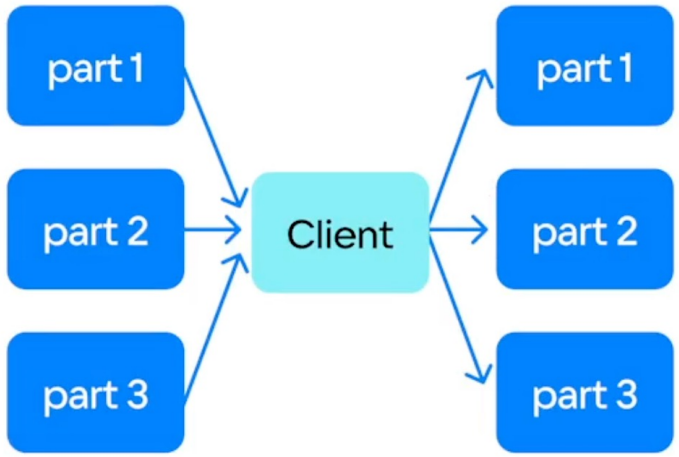
- MapReduce algorithm. Basic components.
- Hadoop MapReduce
- Hadoop Streaming

Hadoop Ecosystem

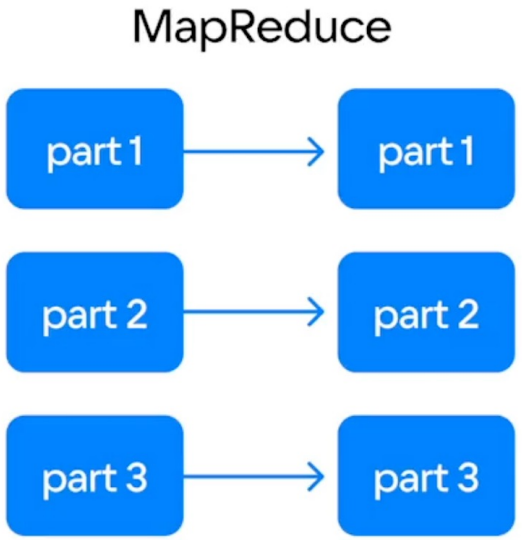


MapReduce. How to process data efficiently?

- Data locality
- Lack of efficient random access



hdfs dfs
-cp



hdfs distcp

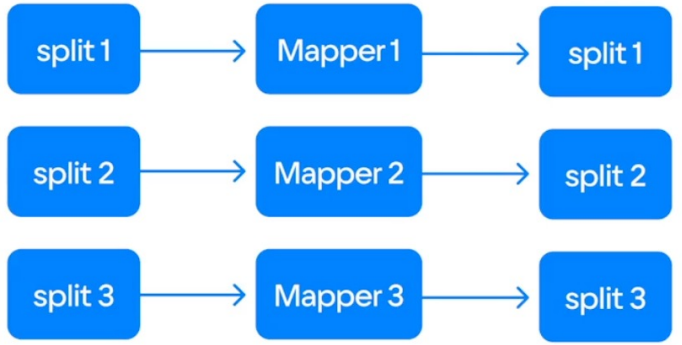
MapReduce. Massive data parallelism

- Most of the processing tasks can be separated into individual subtasks

I. Example: Remove punctuation (no reduction)

$$D = \{d_1, \ldots, d_n\} \Longrightarrow \hat{D} = \{\hat{d}_1, \ldots, \hat{d}_n\}$$

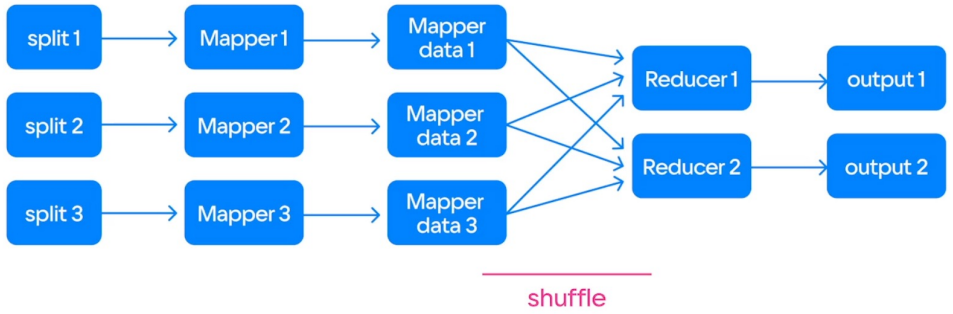
1. Process each document independently



II. Example: Build an index for the data corpus

$$D = \{d_1, \ldots, d_n\} \overset{\text{to index}}{\Longrightarrow} \{w_i \rightarrow [d_{i_1}, \ldots, d_{i_p}]\}$$

1. Process each document independently
2. Combine the results



MapReduce. Simple functional primitives

Stateless functional primitives that operate on immutable data:

- Allows reusability and **reruns**
- No shared memory
- Deterministic

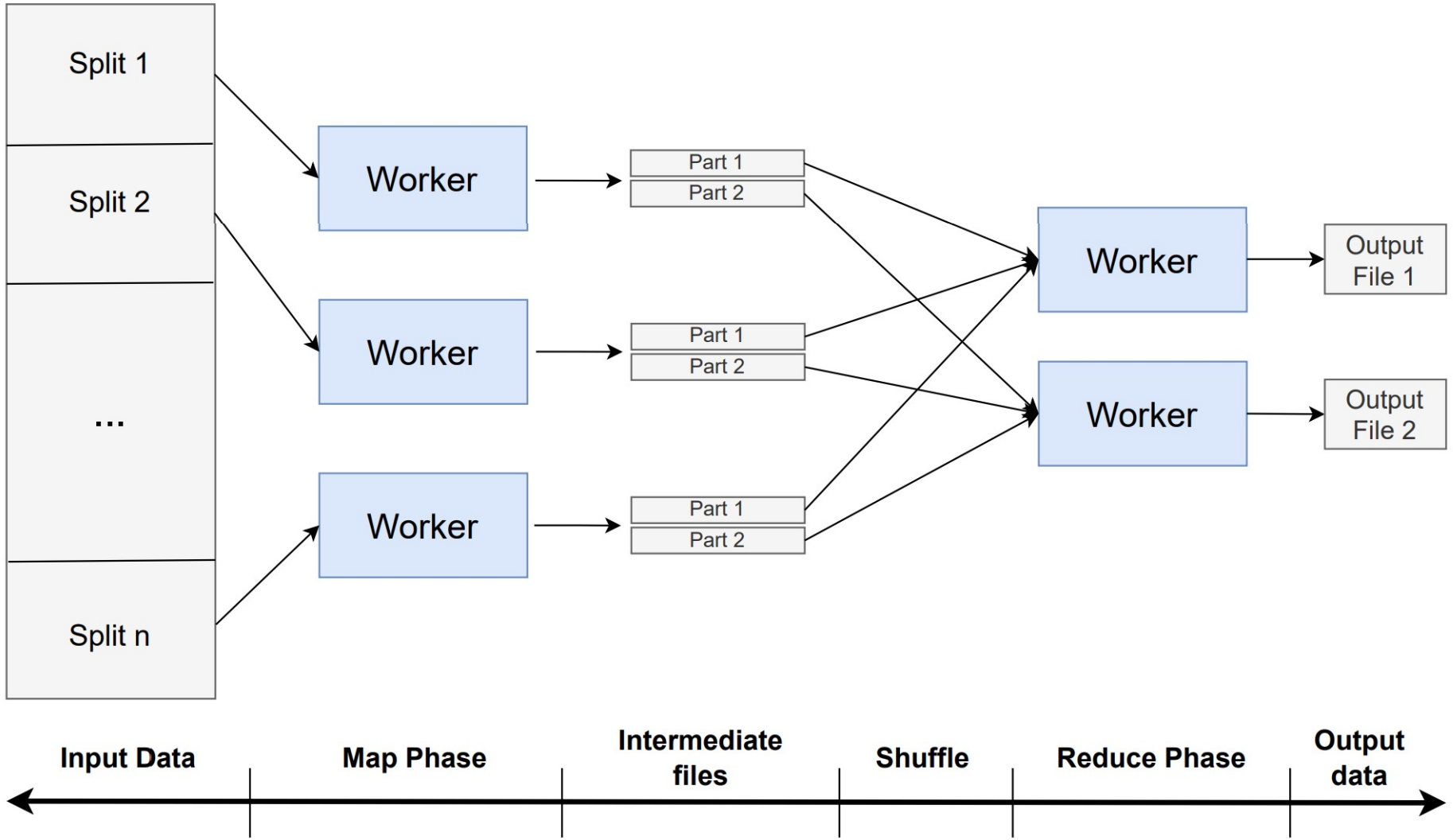
List one-to-one transformation:

$$map : f, [v_1, \ldots, v_n] \rightarrow [f(v_1), \ldots, f(v_n)]$$

List reduction:

$$reduce : g, [v_1, \ldots, v_n] \rightarrow g(v_1, g([v_2, \ldots, v_n]))$$

MapReduce Flow



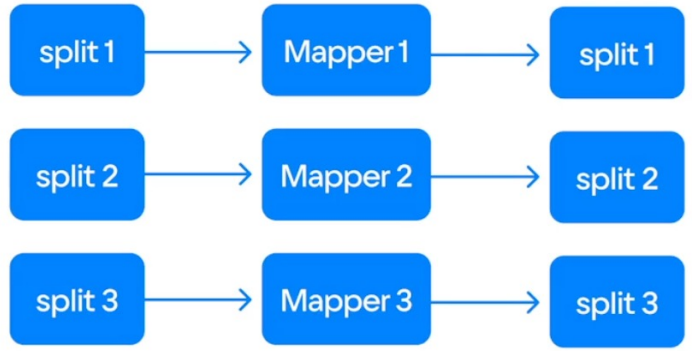
MapReduce Flow. Massive data parallelism

- Most of the processing tasks can be separated into individual subtasks

I. Example: Remove punctuation (no reduction)

$$D = \{d_1, \ldots, d_n\} \Longrightarrow \hat{D} = \{\hat{d}_1, \ldots, \hat{d}_n\}$$

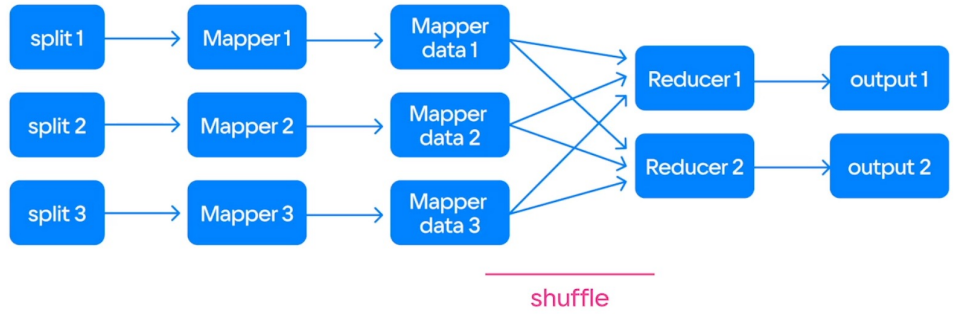
1. Process each document independently



II. Example: Build an index for the data corpus

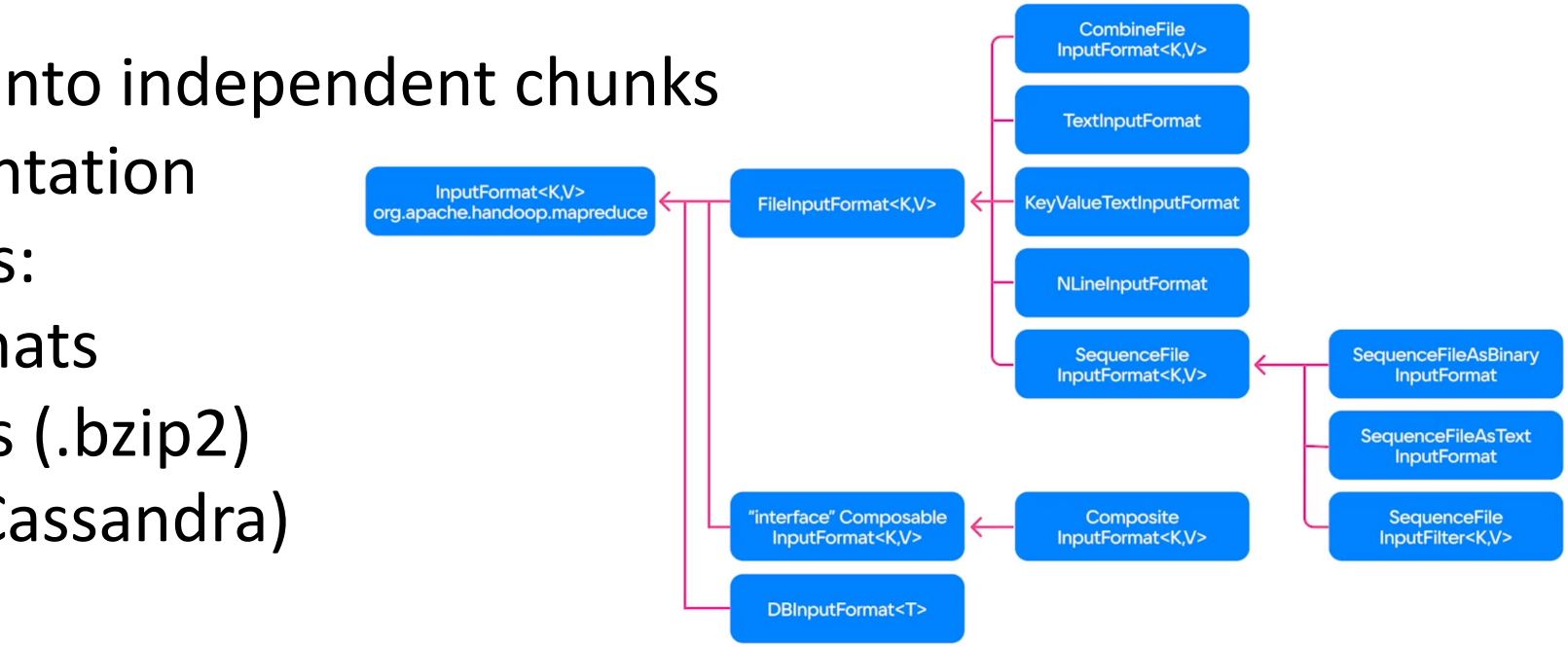
$$D = \{d_1, \ldots, d_n\} \overset{\text{to index}}{\Longrightarrow} \{w_i \rightarrow [d_{i_1}, \ldots, d_{i_p}]\}$$

1. Process each document independently
2. Combine the results



MapReduce Flow. Input Data

- A single file or a group of files
- Located in HDFS
- Allows it to be split into independent chunks
- A key-value representation
- Various data sources:
  - Various text formats
  - Splitable archives (.bzip2)
  - SQL databases (Cassandra)

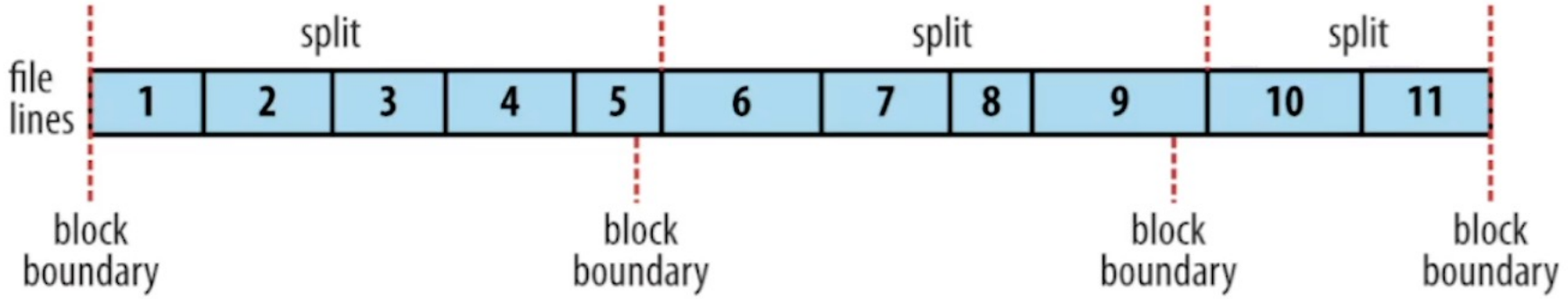


MapReduce Flow. Input Data

Split size: max(minSize, min(maxSize, blockSize))

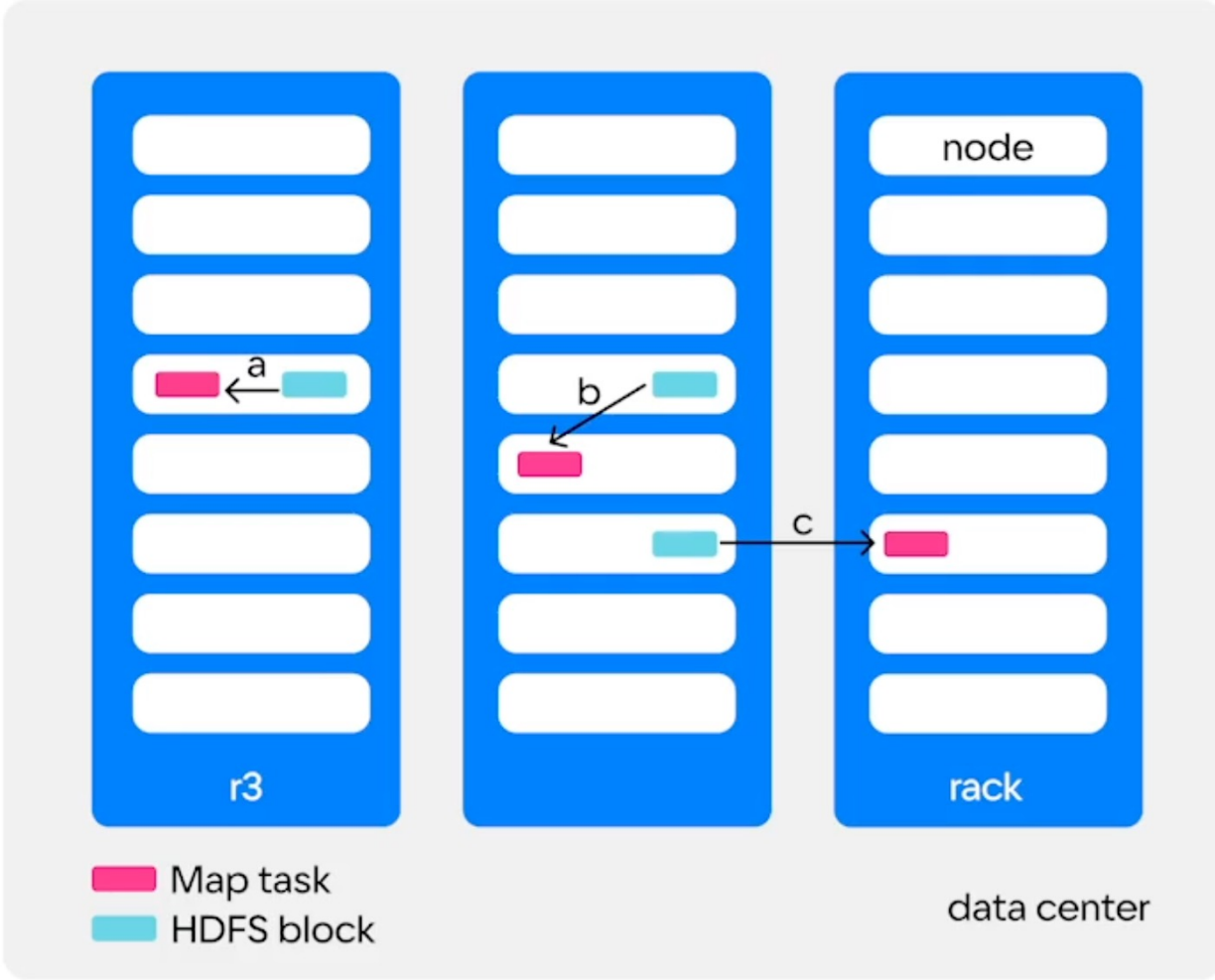
maxSize —> mapred.max.split.size
minSize —> mapred.min.split.size
blockSize —> HDFS block size

- Lower split size —> better data locality
- Lower split size —> better fault tolerance
- Larger split size —> lower computational overhead



MapReduce Flow. Map Phase

- Independent workers
- Transforms the sequence of KV pairs from the split
- Number of mappers equal to the number of splits
- Tries to have data locality, but not as important nowadays



MapReduce Flow. Intermediate data

- Mapper output is partitioned by the key — *one* file for each reducer (Partitioner)
  - *P(K) = hash(K) % #Reduces*
  - *Spill files are created before the local merge (*multi-way merge sort*)*
  - Salting: key —> (key, salt)

- Each file is sorted by key (SortComparator)
  - Distributed sorting
  - Allows for secondary sort

- Usually stored in the local FC (instead of HDFS)
  - Reduces replication overhead
  - Cause recomputations in case of failure

MapReduce Flow. Combiner

- The Optional Combiner Phase can perform reduction locally to decrease the amount of data transferred through the network
- Pure optimization execution: has to be an associative $(a \oplus b) \oplus c = a \oplus (b \oplus c)$ and commutative $a \oplus b = b \oplus a$ function
- Has to be compatible with Mapper output and Reducer input formats

MapReduce Flow. Shuffle

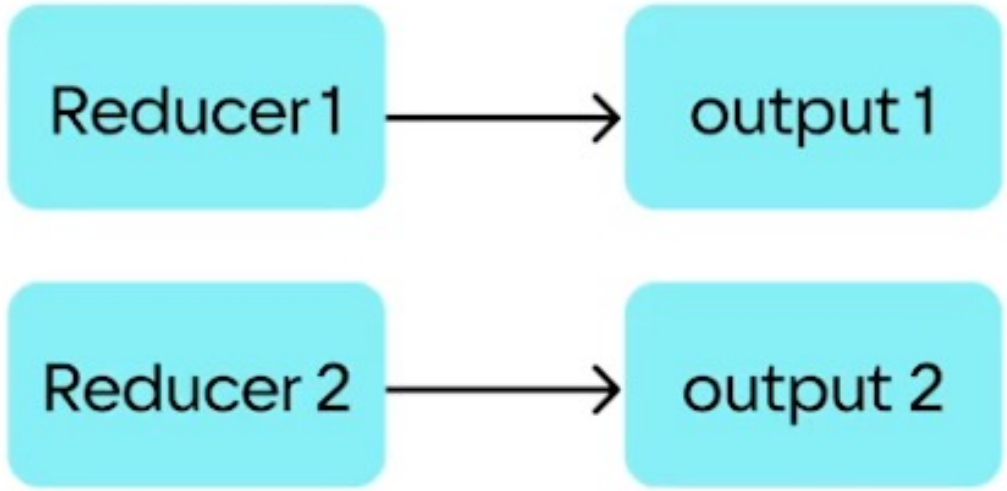
- The most network-heavy part of the framework
  - Key optimization factor

- Second part of the distributed sorting
  - Merge the sorted lists
  - Additional Combiner calls are possible

- The sorted list of keys is separated into groups of values with the “same” key (GroupingComparator)

MapReduce Flow. Reduce Phase

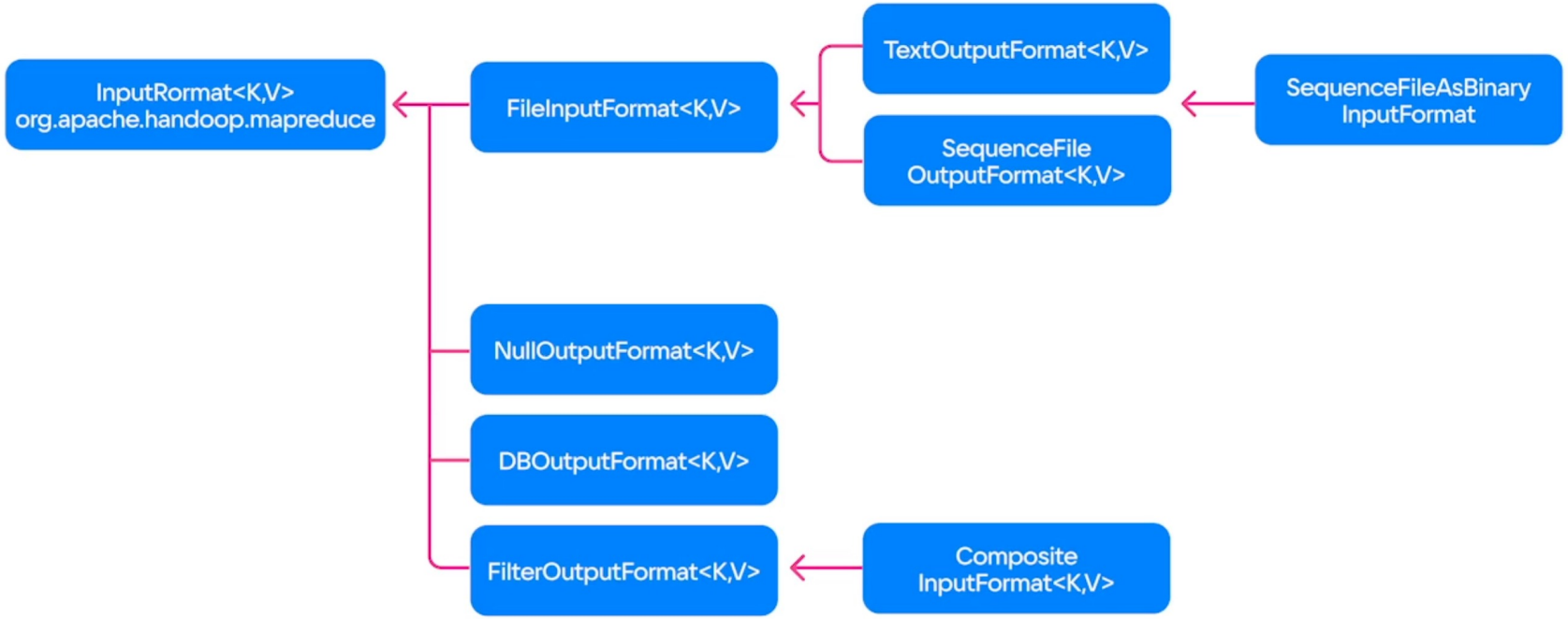
- The sequence of groups (K, [V1, …, Vp]) is passed to the reducer to be aggregated
- The resulting sequence of KV pairs is written into HDFS as it is



MapReduce Flow. Output Data

- Stored in HDFS

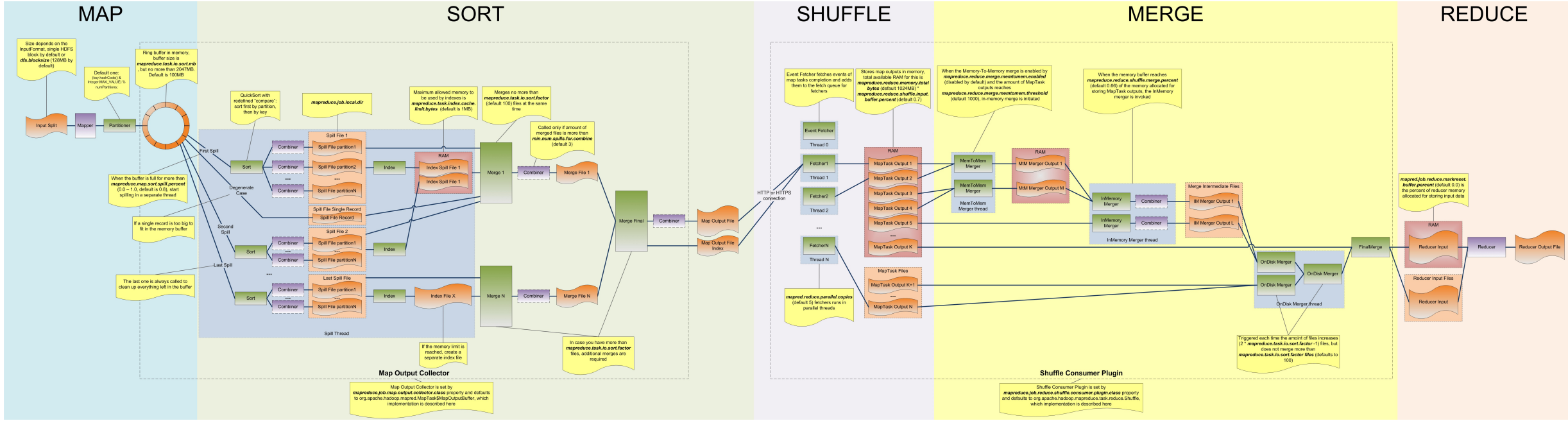
- The number of output files is equal to the number of reducers



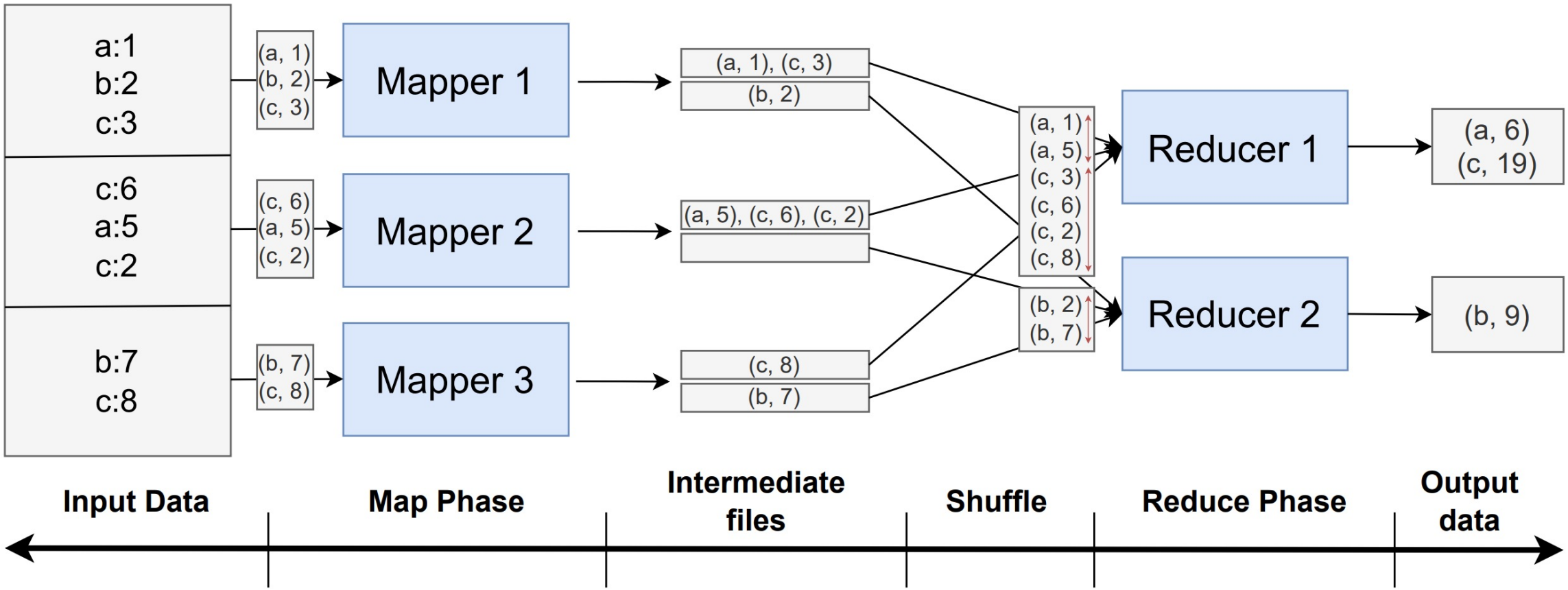
MapReduce Flow. What can you change?

1. Input data location
2. Input data KV representation
   and size of data splits
3. Mapper function
4. Partitioner
5. Primary Sort Comparator
6. Combiner
7. Secondary Sort Comparator
   (Grouping Comparator)
8. Reducer
9. Output data location

MapReduce Flow



MapReduce Flow. Example



MapReduce Flow. Fault tolerance

- Input and output data are stored in HDFS, which ensures their availability
- If any intermediate data is lost, just recompute it from scratch — **stateless, deterministic Map/Reduce is required**
- If the Map Phase takes too long, it might be beneficial to separate the MapReduce task into the Map-only and Reduce-only tasks to store intermediate results in HDFS

MapReduce Flow. Speculative execution

- Tasks (map/reduce) can be executed multiple times if the process stalls on one of the nodes

- The first result to be computed is accepted; others are discarded

- **Stateless, deterministic Map/Reduce is required**

- mapreduce.map.speculative
- mapreduce.reduce.speculative

- Helps against straggler-tasks and the “long tail”

MapReduce Flow. Reducer input imbalance

- The user has to ensure that all reducers are utilized equally

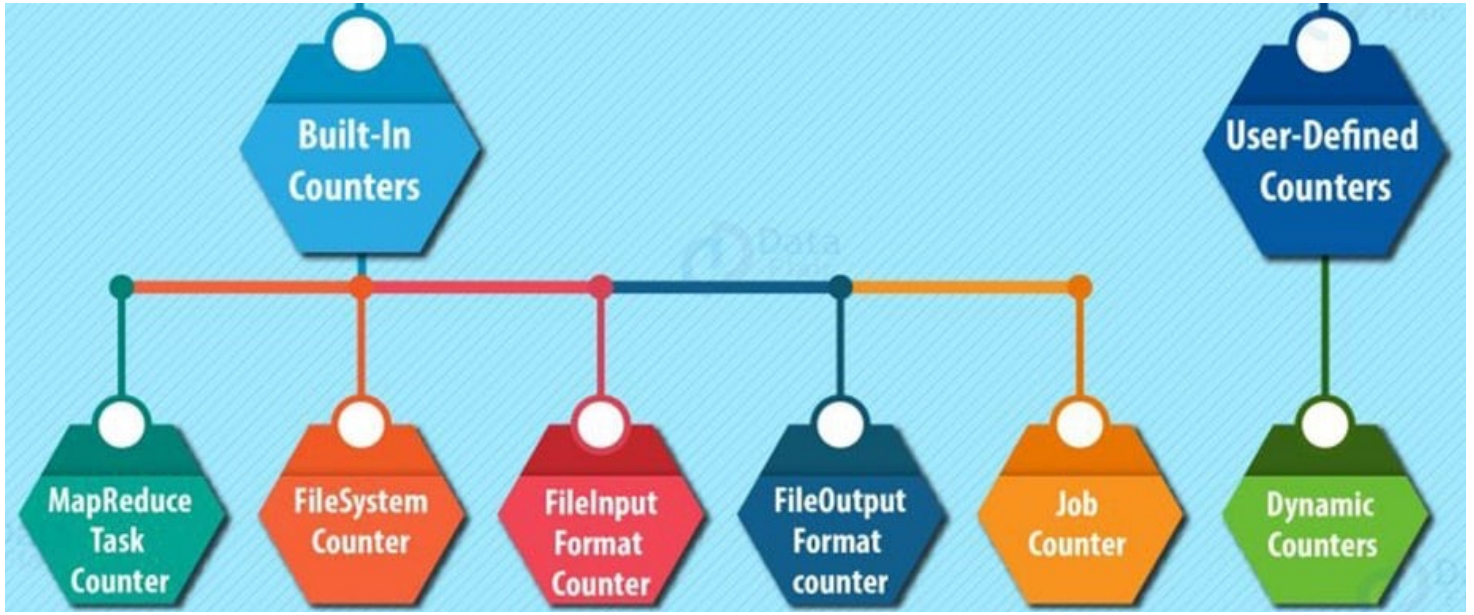
- The load imbalance can be solved using a custom partitioner

Example. Websites Processing: (URL, Content) —> Statistics

- URL can be distributed unequally
- i.e., google.com is much more frequent
- Naïve hash partitioning will create load imbalance

MapReduce. Counters

- Mechanism for monitoring and collecting non-important results
- Expensive to update
- Can only be a 64-bit integer
- Non-deterministic by design (i.e., task reruns)
- Available from UI to monitor execution
- Either built-in or user-defined:



MapReduce. Hadoop Streaming

- Hadoop MapReduce has native implementations for:
  - Java, C++, Scala
- No Python API
- Hadoop Streaming — a wrapper for the Java API for Python code
- Downside: purely textual data transfer

MapReduce. Hadoop Streaming

- Mapper, Reducer, … receive the input from *stdin* and write to *stdout*
- Mapper processes input line-by-line and writes the output KV pair, separating using <TAB>
- Reducer receives sorted by key KV pairs (instead of the values list)
- Roughly equal to:

  $cat input | python map.py | sort | python reduce.py

- All of the 9 parts of the MapReduce workflow can be adjusted using command-line keys
- https://hadoop.apache.org/docs/r1.2.1/streaming.html

MapReduce. Word Count I

Count the number of occurrences of each word in the text corpus: $D = \{d_1, \ldots, d_n\} \Longrightarrow \{(w_1, c_1), \ldots, (w_p, c_p)\}$

The simplest implementation:

The problem is that too much data is being transferred between the map and reduce phases.

```
def map(doc_id, doc):
    for word in doc:
        Emit(word, 1)

def reduce(word, [c1, c1, ...]):
    count = 0
    for c in [c1, c2, ...]:
        count += c
    Emit(word, count)
```

MapReduce. Word Count II. Combiner

Count the number of occurrences of each word in the text corpus: $D = \{d_1, \ldots, d_n\} \Longrightarrow \{(w_1, c_1), \ldots, (w_p, c_p)\}$

The Combiner function is optional and can be used to optimize the process by reducing the amount of data transferred between the map and reduce phases:

```
def combiner(word, [c1, c2, ...]):
    count = 0
    for c in [c1, c2, ...]:
        count += c
    Emit(word, count)
```

MapReduce. Word Count III. In-mapper combiner

Count the number of occurrences of each word in the text corpus: $D = \{d_1, \ldots, d_n\} \Longrightarrow \{(w_1, c_1), \ldots, (w_p, c_p)\}$

The next problem is that the map function is emitting too many intermediate key-value pairs.
We can optimize the map function by using a local aggregation technique:

```
def map(doc_id, doc):
    H = defaultdict(int)
    for word in doc:
        H[word] += 1
    for word, count in H.items():
        Emit(word, count)
```

MapReduce. Word Count III. In-mapper combiner

Count the number of occurrences of each word in the text corpus: $D = \{d_1, \ldots, d_n\} \Longrightarrow \{(w_1, c_1), \ldots, (w_p, c_p)\}$

- The number of intermediate key-value pairs emitted by the map function (+ the amount of data transferred between the map and reduce phases) is reduced
- The local aggregation should not use too much memory
- If a document is too large, we may need to flush the local aggregation to the output periodically

MapReduce. Word Count IV. In-mapper combiner V2

Count the number of occurrences of each word in the text corpus: $D = \{d_1, \ldots, d_n\} \Longrightarrow \{(w_1, c_1), \ldots, (w_p, c_p)\}$

If the number of distinct words is small, we can reuse the same dictionary for different documents:

We have to flush the dictionary periodically to avoid OOM

The dictionary can be initialized from local FC to implement tricky behavior (i.e., Map-side join), or be broadcast using --files CLI flag

```
# Initialize the dictionary outside the
map function.
H = defaultdict(int)
# The map function now reuses the same
dictionary for different documents.
def map(doc_id, doc):
    for word in doc:
        H[word] += 1
# Close the map function by emitting the
contents of the dictionary.
for word, count in H.items():
    Emit(word, count)
```