Unit 5: Big Data Processing Concepts

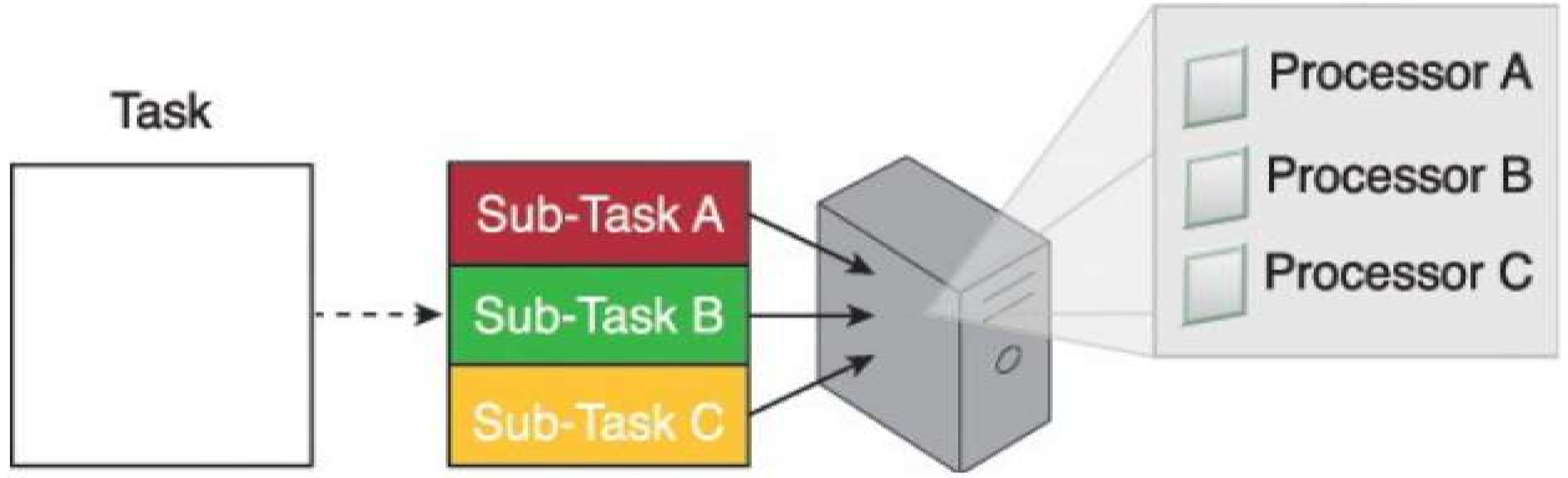
Outline

- Parallel Data Processing
- Distributed Data Processing
- Hadoop
- Processing Workloads
- Cluster
- Processing in Batch Mode
- Processing in Real-time Mode

Parallel Data Processing

- Parallel data processing involves the simultaneous execution of multiple sub-tasks that collectively comprise a larger task.
- The goal is to reduce the execution time by dividing a single larger task into multiple smaller tasks that run concurrently.
- Although parallel data processing can be achieved through multiple networked machines, it is more typically achieved within the confines of a single machine with multiple processors or cores, as shown in the following Figure.

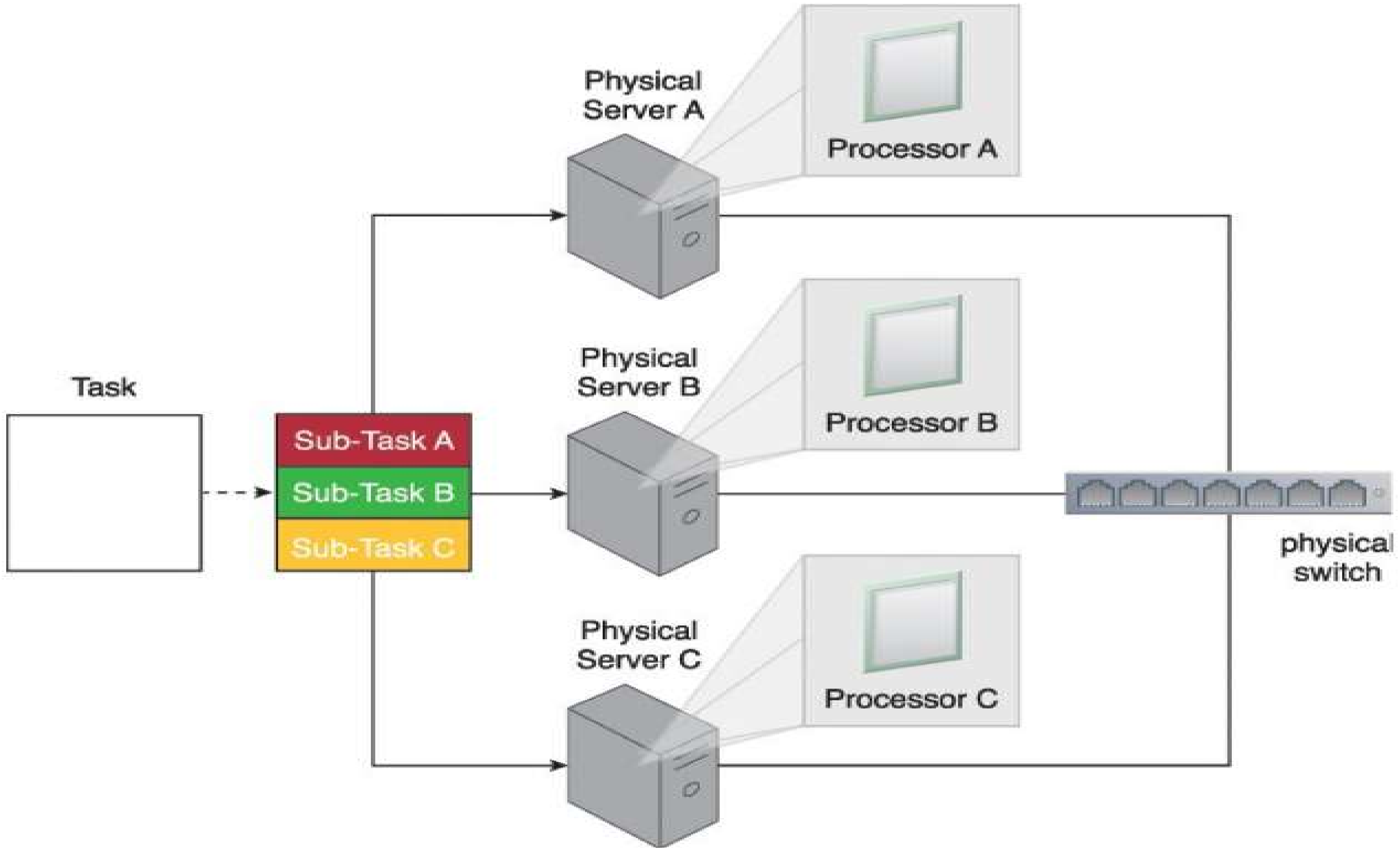
- A task can be divided into three sub-tasks that are executed in parallel on three different processors within the same machine



Distributed Data Processing

- Distributed data processing is closely related to parallel data processing in that the same principle of “divide-and-conquer” is applied.
- However, distributed data processing is always achieved through physically separate machines that are networked together as a cluster.

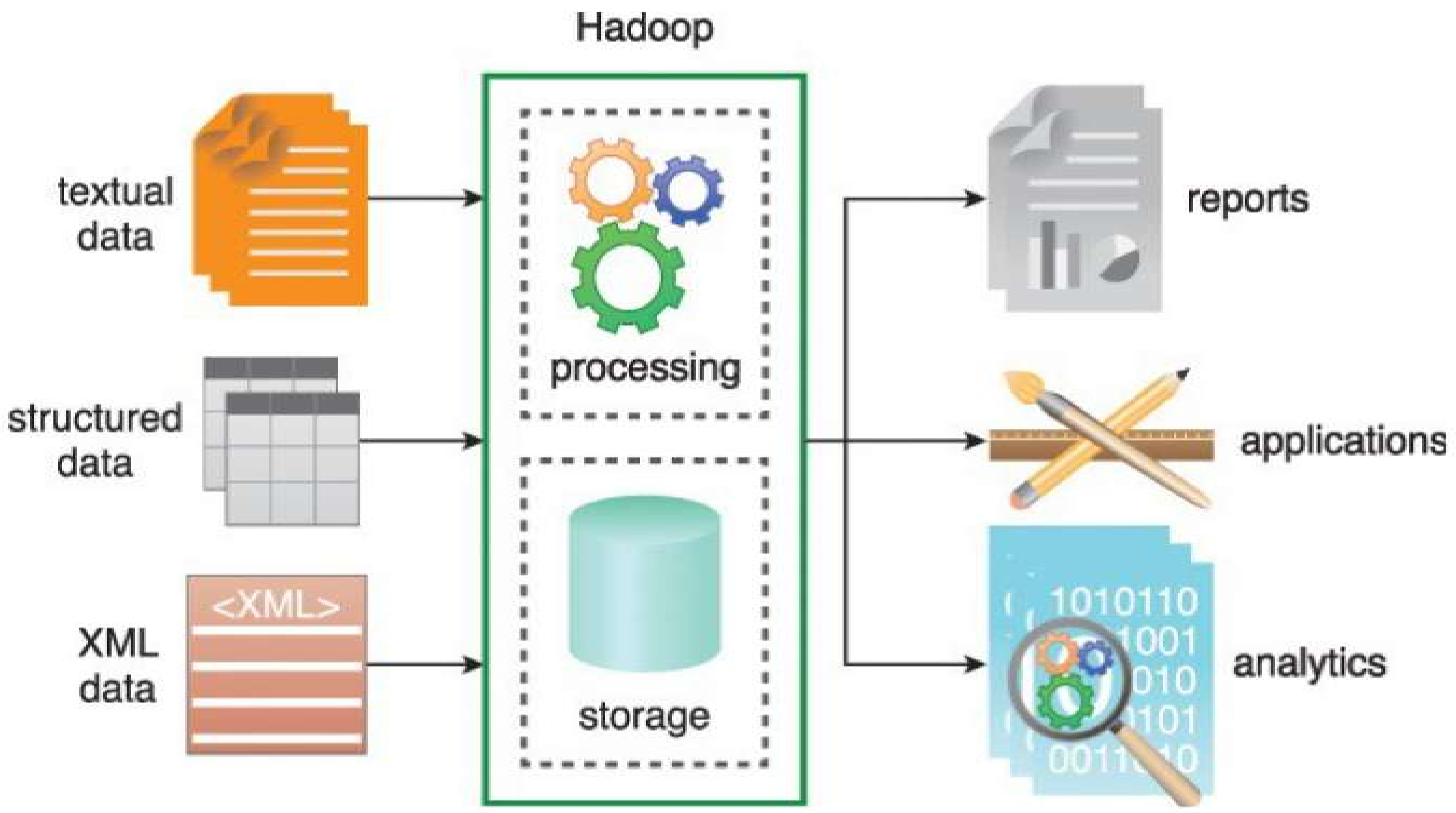
An example of distributed data processing



Hadoop

- Hadoop is an open-source framework for large-scale data storage and data processing that is compatible with commodity hardware.
- The Hadoop framework has established itself as a de facto industry platform for contemporary Big Data solutions.
- It can be used as an ETL engine or as an analytics engine for processing large amounts of structured, semistructured and unstructured data.
- From an analysis perspective, Hadoop implements the MapReduce processing framework.

Hadoop is a versatile framework that provides both processing and storage capabilities



Processing Workloads

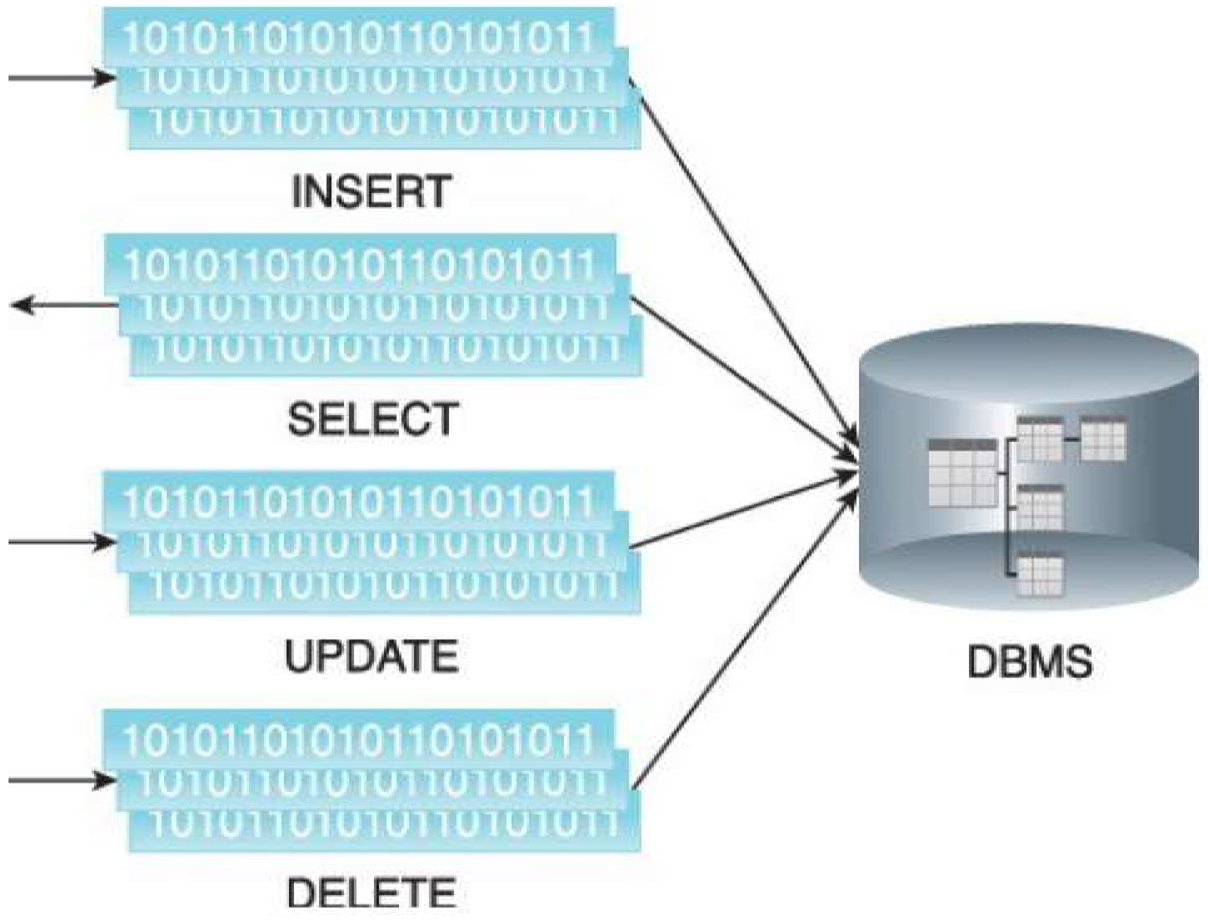
- A processing workload in Big Data is defined as the amount and nature of data that is processed within a certain amount of time. Workloads are usually divided into two types:
- batch
- transactional

Batch

- Batch processing, also known as offline processing, involves processing data in batches
- and usually imposes delays, which in turn results in high-latency responses. Batch
- workloads typically involve large quantities of data with sequential read/writes and
- comprise of groups of read or write queries.

- Queries can be complex and involve multiple joins. OLAP systems commonly process workloads in batches.
- Strategic BI and analytics are batch-oriented as they are highly read-intensive tasks involving large volumes of data.

A batch workload can include grouped read/writes to INSERT, SELECT, UPDATE and DELETE

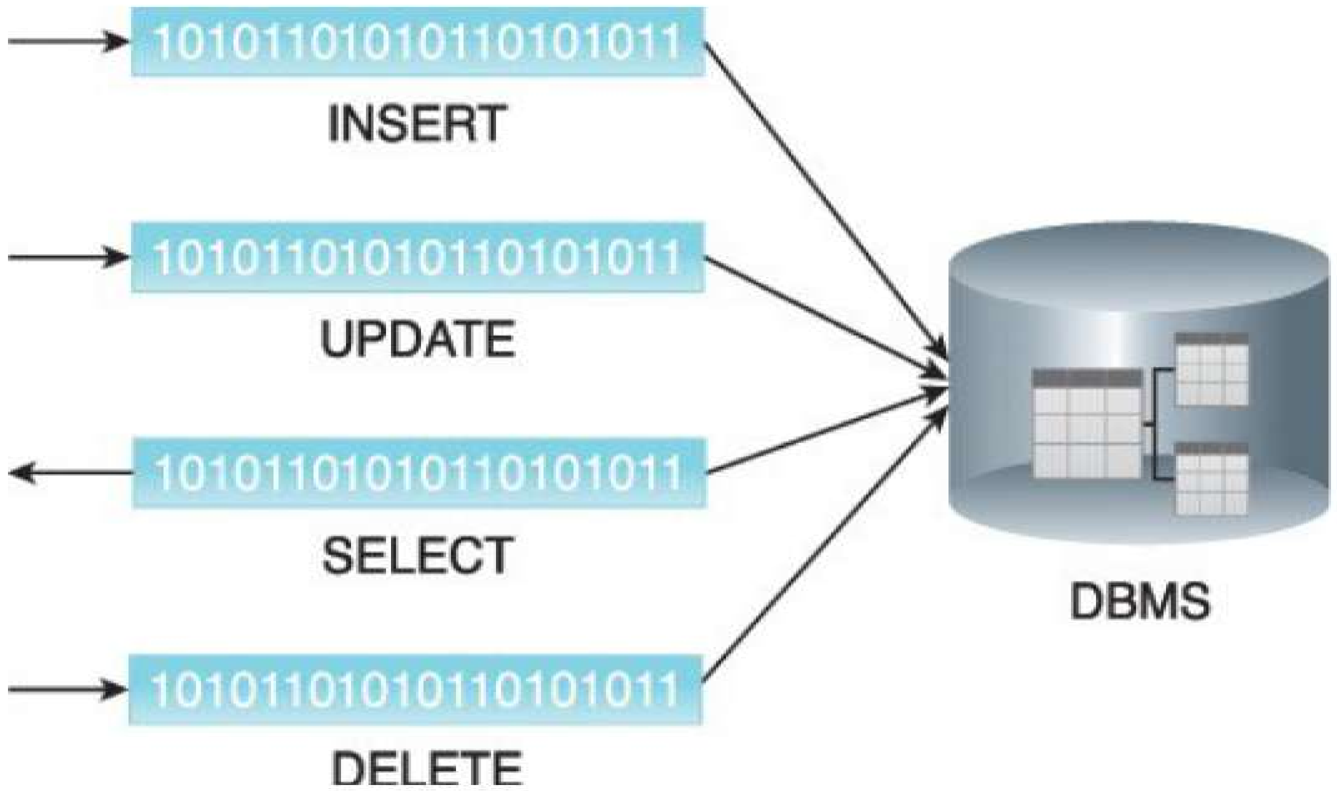


Transactional

- Transactional processing is also known as online processing. Transactional workload processing follows an approach whereby data is processed interactively without delay, resulting in low-latency responses.
- Transaction workloads involve small amounts of data with random reads and writes.

- OLTP and operational systems, which are generally write-intensive, fall within this category.
- Although these workloads contain a mix of read/write queries, they are generally more write-intensive than read-intensive.
- Transactional workloads comprise random reads/writes that involve fewer joins than business intelligence and reporting workloads.
- Given their online nature and operational significance to the enterprise, they require low-latency responses with a smaller data footprint

Transactional workloads have few joins and lower latency responses than batch workloads

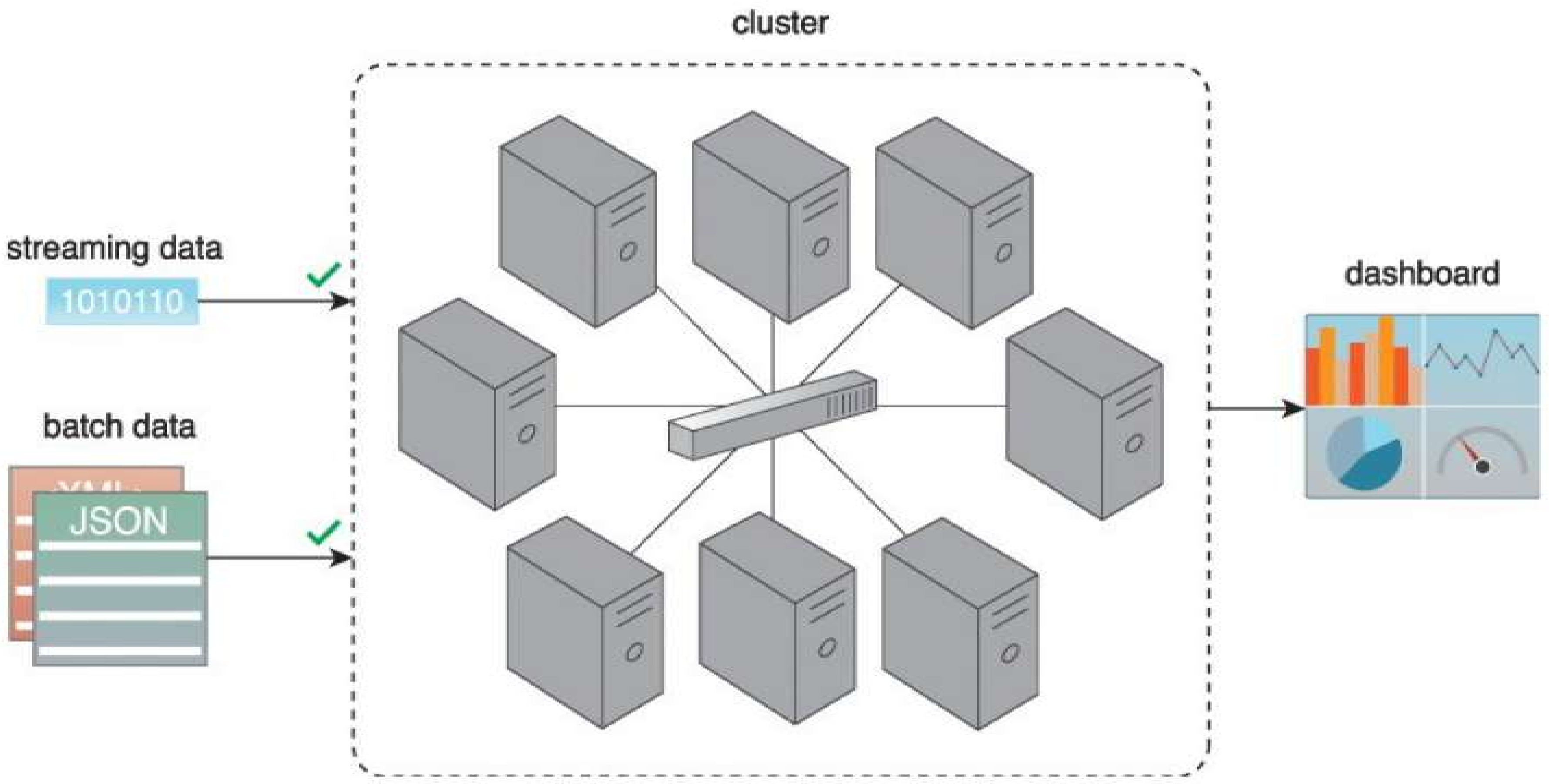


Cluster

- In the same manner that clusters provide necessary support to create horizontally scalable storage solutions, clusters also provides the mechanism to enable distributed data processing with linear scalability.
- Since clusters are highly scalable, they provide an ideal environment for Big Data processing as large datasets can be divided into smaller datasets and then processed in parallel in a distributed manner.

- An additional benefit of clusters is that they provide inherent redundancy and fault tolerance, as they consist of physically separate nodes.
- Redundancy and fault tolerance allow resilient processing and analysis to occur if a network or node failure occurs.
- Due to fluctuations in the processing demands placed upon a Big Data environment, leveraging cloud-host infrastructure services, or ready-made analytical environments as the backbone of a cluster, is sensible due to their elasticity and pay-for-use model of utility-based computing.

A cluster can be utilized to support batch processing of bulk data and realtime processing of streaming data



Processing in Batch Mode

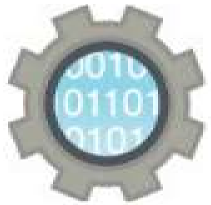
- In batch mode, data is processed offline in batches and the response time could vary from minutes to hours. As well, data must be persisted to the disk before it can be processed.
- Batch mode generally involves processing a range of large datasets, either on their own or joined together, essentially addressing the volume and variety characteristics of Big Data datasets.

- The majority of Big Data processing occurs in batch mode. It is relatively simple, easy to set up and low in cost compared to realtime mode.
- Strategic BI, predictive and prescriptive analytics and ETL operations are commonly batch-oriented.

Batch Processing with MapReduce

- MapReduce is a widely used implementation of a batch processing framework.
- It is highly scalable and reliable and is based on the principle of divide-and-conquer, which provides built-in fault tolerance and redundancy.
- It divides a big problem into a collection of smaller problems that can each be solved quickly.
- MapReduce has roots in both distributed and parallel computing.
- MapReduce is a batch-oriented processing engine used to process large datasets using parallel processing deployed over clusters of commodity hardware.

The symbol used to represent a processing engine



- MapReduce does not require that the input data conform to any particular data model.
- Therefore, it can be used to process schema-less datasets. A dataset is broken down into multiple smaller parts, and operations are performed on each part independently and in parallel.
- The results from all operations are then summarized to arrive at the answer.
- Because of the coordination overhead involved in managing a job, the MapReduce processing engine generally only supports batch workloads as this work is not expected to have low latency.
- MapReduce is based on Google’s research paper on the subject, published in early 2000.

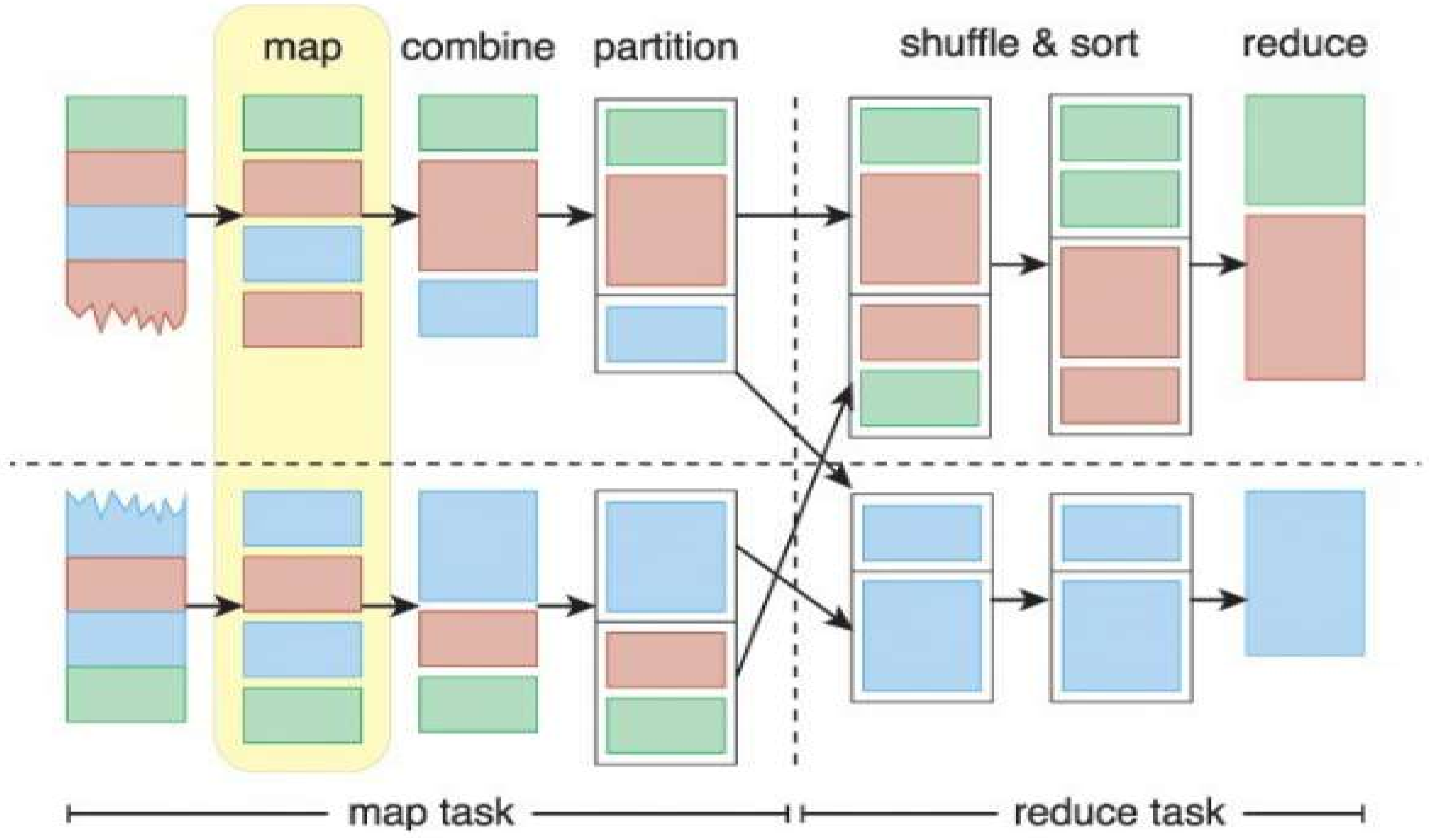
- The MapReduce processing engine works differently compared to the traditional data processing paradigm.
- Traditionally, data processing requires moving data from the storage node to the processing node that runs the data processing algorithm.
- This approach works fine for smaller datasets; however, with large datasets, moving data can incur more overhead than the actual processing of the data.

- With MapReduce, the data processing algorithm is instead moved to the nodes that store the data.
- The data processing algorithm executes in parallel on these nodes, thereby eliminating the need to move the data first.
- This not only saves network bandwidth but it also results in a large reduction in processing time for large datasets, since processing smaller chunks of data in parallel is much faster.

Map and Reduce Tasks

- A single processing run of the MapReduce processing engine is known as a MapReduce job.
- Each MapReduce job is composed of a map task and a reduce task and each task consists of multiple stages.
- Map tasks
    - map
    - combine (optional)
    - partition
- Reduce tasks
    - shuffle and sort
    - reduce

An illustration of a MapReduce job with the map stage highlighted

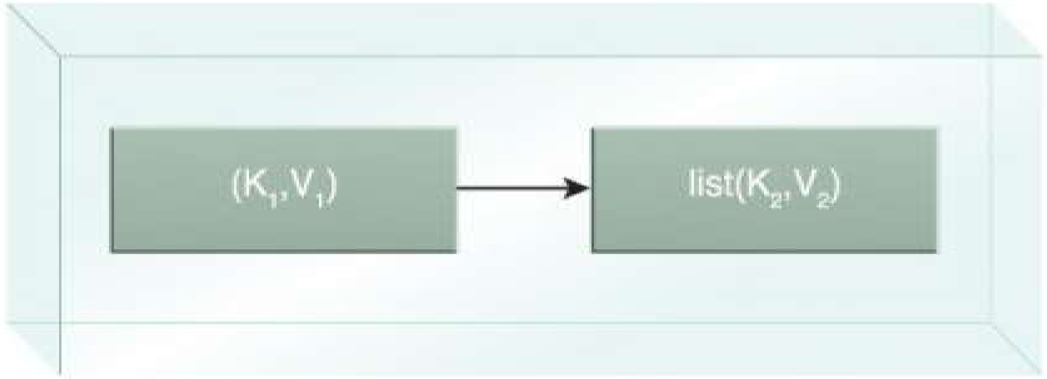


Map

- The first stage of MapReduce is known as map, during which the dataset file is divided into multiple smaller splits.
- Each split is parsed into its constituent records as a key-value pair. The key is usually the ordinal position of the record, and the value is the actual record.
- The parsed key-value pairs for each split are then sent to a map function or mapper, with one mapper function per split. The map function executes user-defined logic.
- Each split generally contains multiple key-value pairs, and the mapper is run once for each key-value pair in the split.

- The mapper processes each key-value pair as per the user-defined logic and further generates a key-value pair as its output.
- The output key can either be the same as the input key or a substring value from the input value, or another serializable user-defined object.
- Similarly, the output value can either be the same as the input value or a substring value from the input value, or another serializable user-defined object.

- When all records of the split have been processed, the output is a list of key-value pairs where multiple key-value pairs can exist for the same key.
- It should be noted that for an input key-value pair, a mapper may not produce any output key-value pair (filtering) or can generate multiple key-value pairs (demultiplexing.)

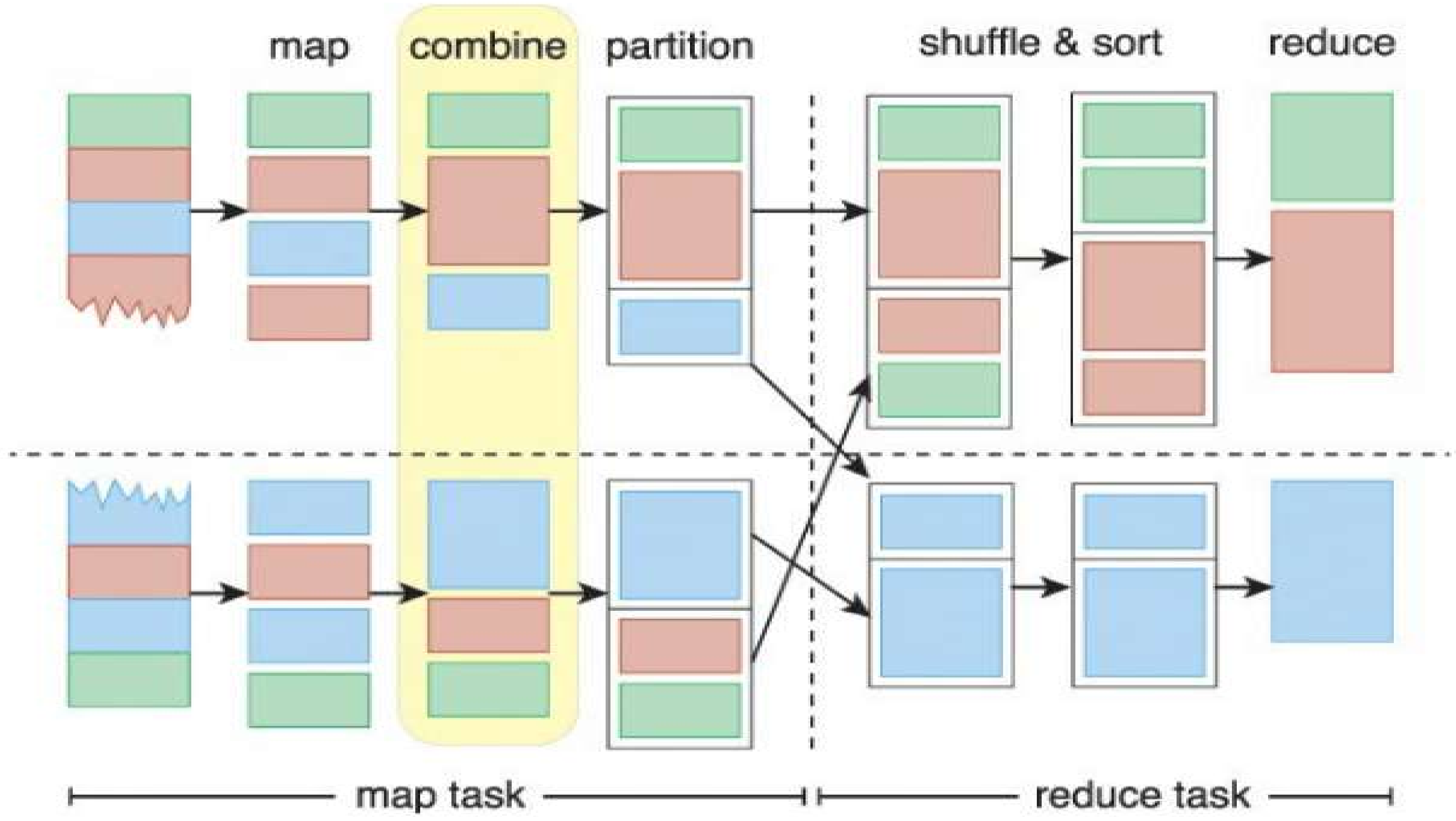


Combine

- Generally, the output of the map function is handled directly by the reduce function.
- However, map tasks and reduce tasks are mostly run over different nodes. This requires moving data between mappers and reducers. This data movement can consume a lot of valuable bandwidth and directly contributes to processing latency.

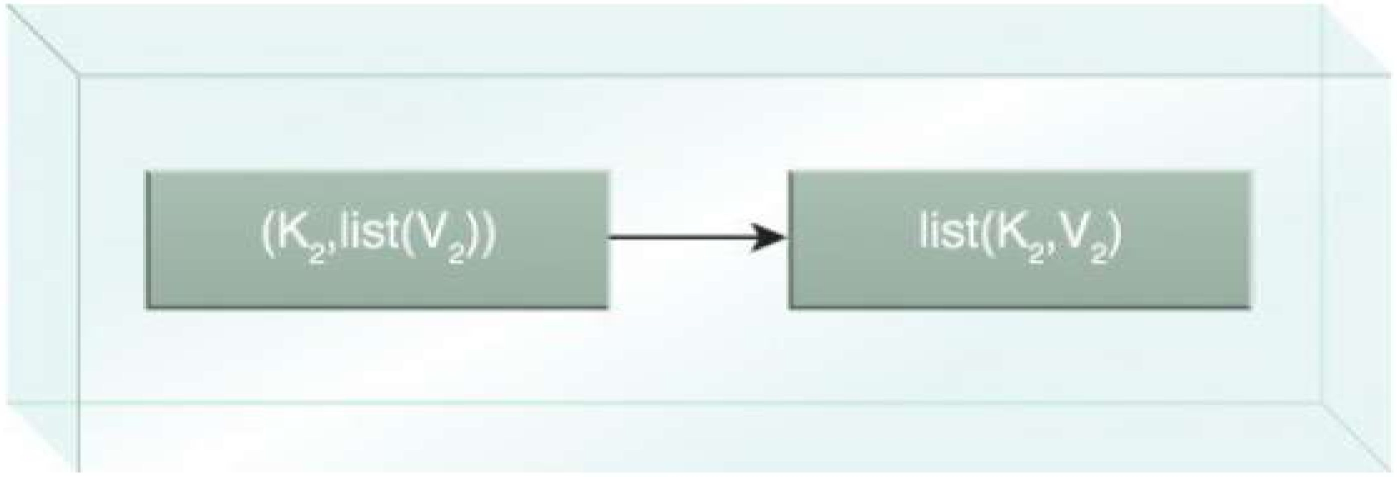
- With larger datasets, the time taken to move the data between map and reduce stages can exceed the actual processing undertaken by the map and reduce tasks.
- For this reason, the MapReduce engine provides an optional combine function (combiner) that summarizes a mapper’s output before it gets processed by the reducer.

The combine stage groups the output from the map stage



- A combiner is essentially a reducer function that locally groups a mapper’s output on the same node as the mapper.
- A reducer function can be used as a combiner function, or a custom user-defined function can be used.
- The MapReduce engine combines all values for a given key from the mapper output, creating multiple key-value pairs as input to the combiner where the key is not repeated and the value exists as a list of all corresponding values for that key.
- The combiner stage is only an optimization stage, and may therefore not even be called by the MapReduce engine.

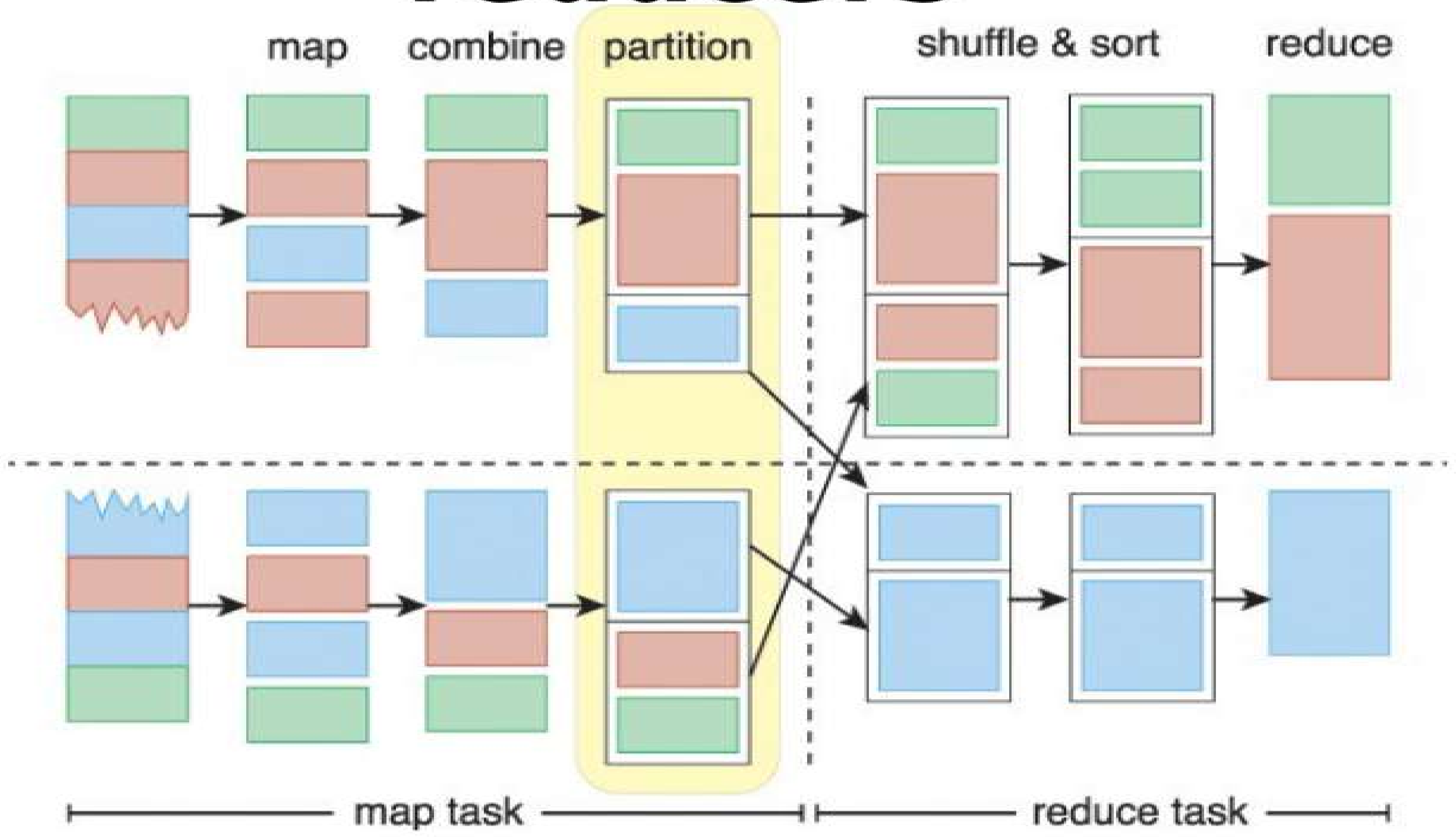
- For example, a combiner function will work for finding the largest or the smallest number, but will not work for finding the average of all numbers since it only works with a subset of the data.



Partition

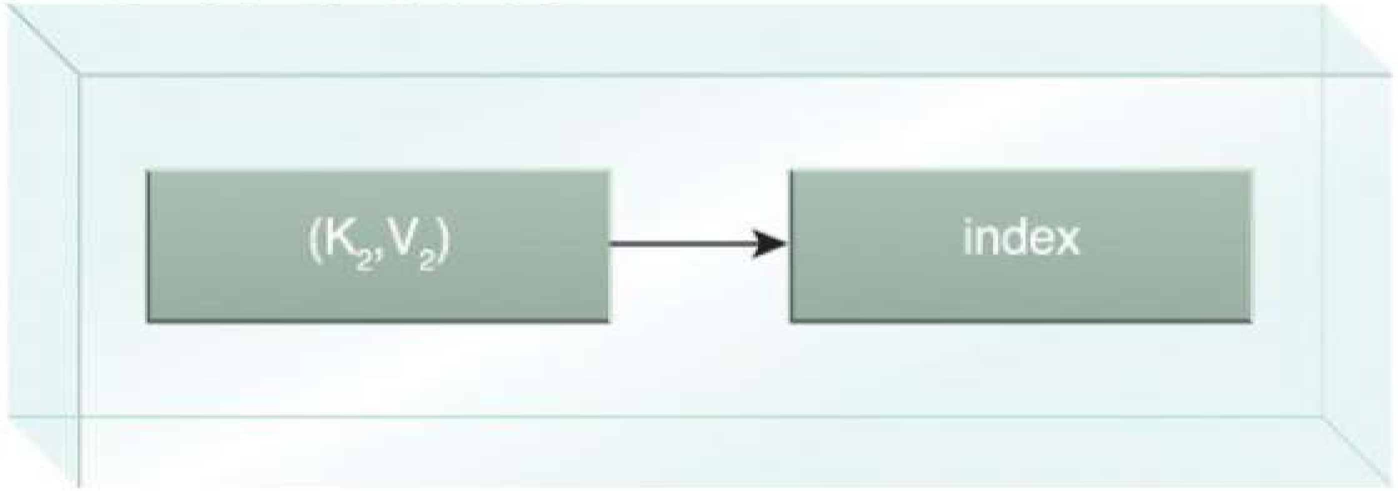
- During the partition stage, if more than one reducer is involved, a partitioner divides the output from the mapper or combiner (if specified and called by the MapReduce engine) into partitions between reducer instances. The number of partitions will equal the number of reducers.

The partition stage assigns output from the map task to reducers



- Although each partition contains multiple key-value pairs, all records for a particular key are assigned to the same partition.
- The MapReduce engine guarantees a random and fair distribution between reducers while making sure that all of the same keys across multiple mappers end up with the same reducer instance.
- Depending on the nature of the job, certain reducers can sometimes receive a large number of key-value pairs compared to others. As a result of this uneven workload, some reducers will finish earlier than others.
- Overall, this is less efficient and leads to longer job execution times than if the work was evenly split across reducers. This can be rectified by customizing the partitioning logic in order to guarantee a fair distribution of key-value pairs.

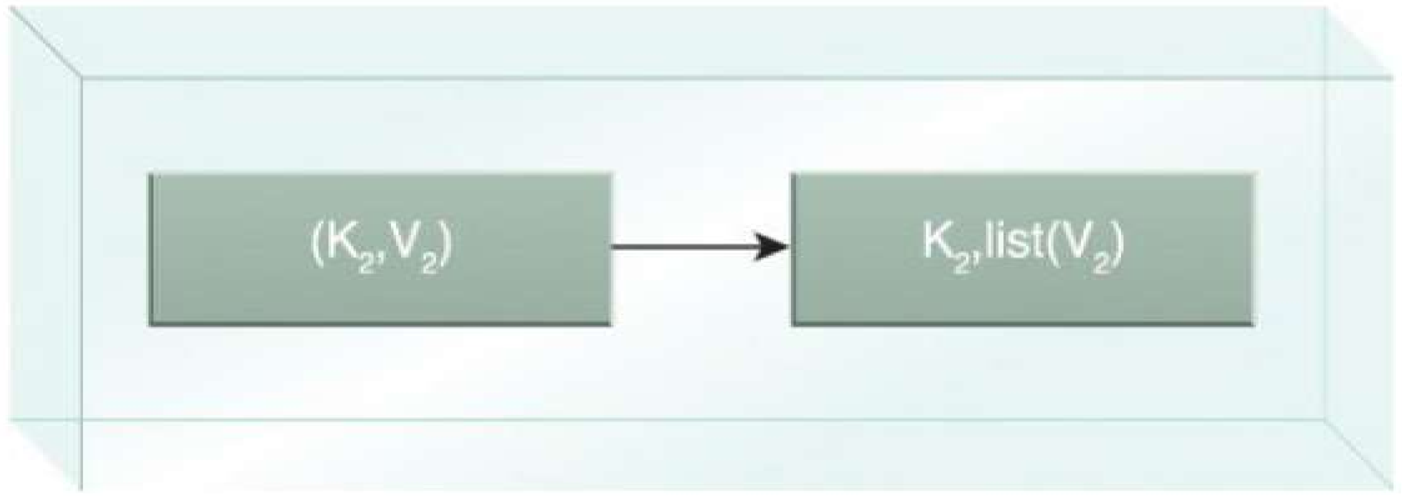
- The partition function is the last stage of the map task. It returns the index of the reducer to which a particular partition should be sent.



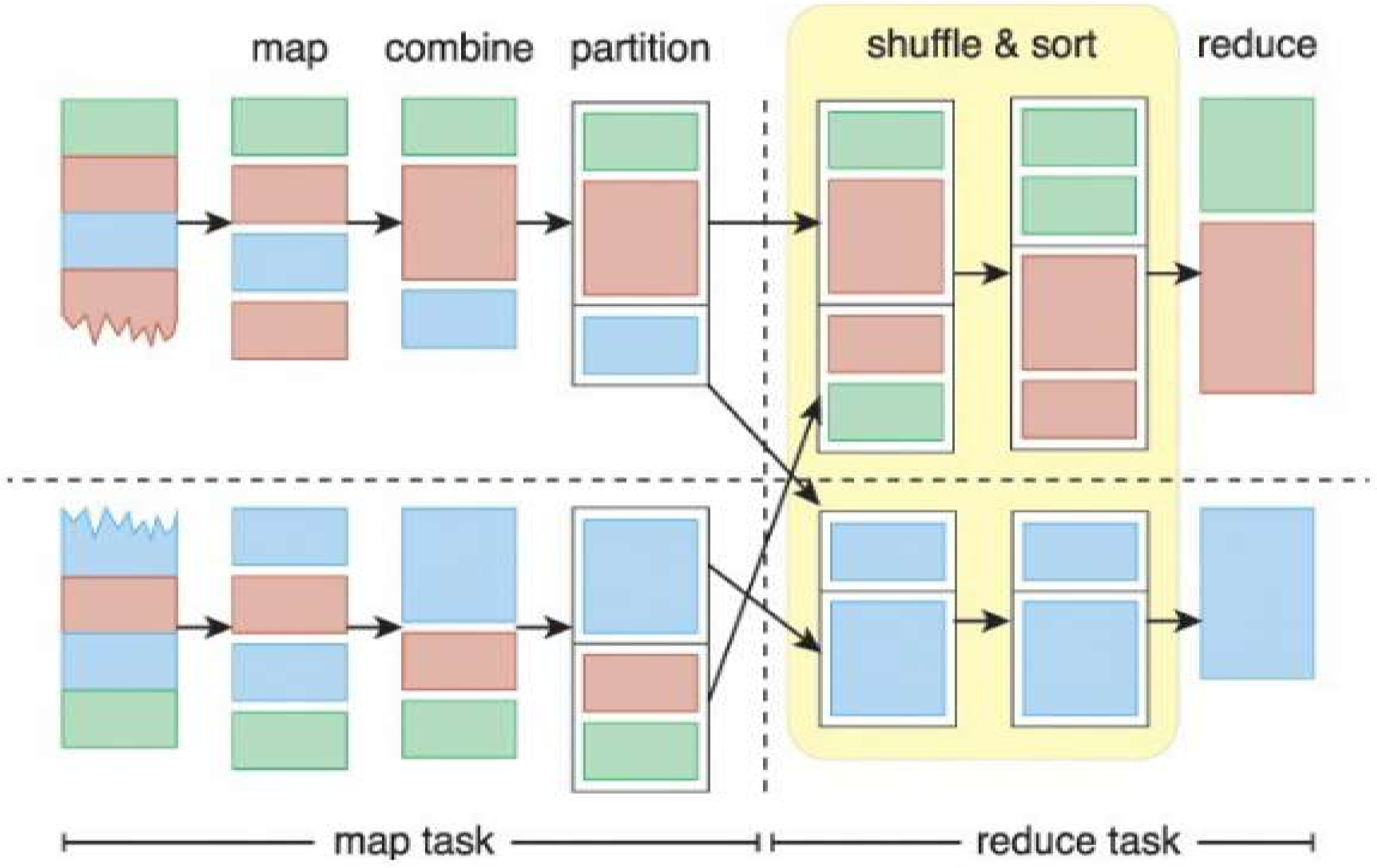
Shuffle and Sort

- During the first stage of the reduce task, output from all partitioners is copied across the network to the nodes running the reduce task. This is known as shuffling. The list based key-value output from each partitioner can contain the same key multiple times.
- Next, the MapReduce engine automatically groups and sorts the key-value pairs according to the keys so that the output contains a sorted list of all input keys and their values with the same keys appearing together. The way in which keys are grouped and sorted can be customized.

- This merge creates a single key-value pair per group, where key is the group key and the value is the list of all group values.



During the shuffle and sort stage, data is copied across the network to the reducer nodes and sorted by key

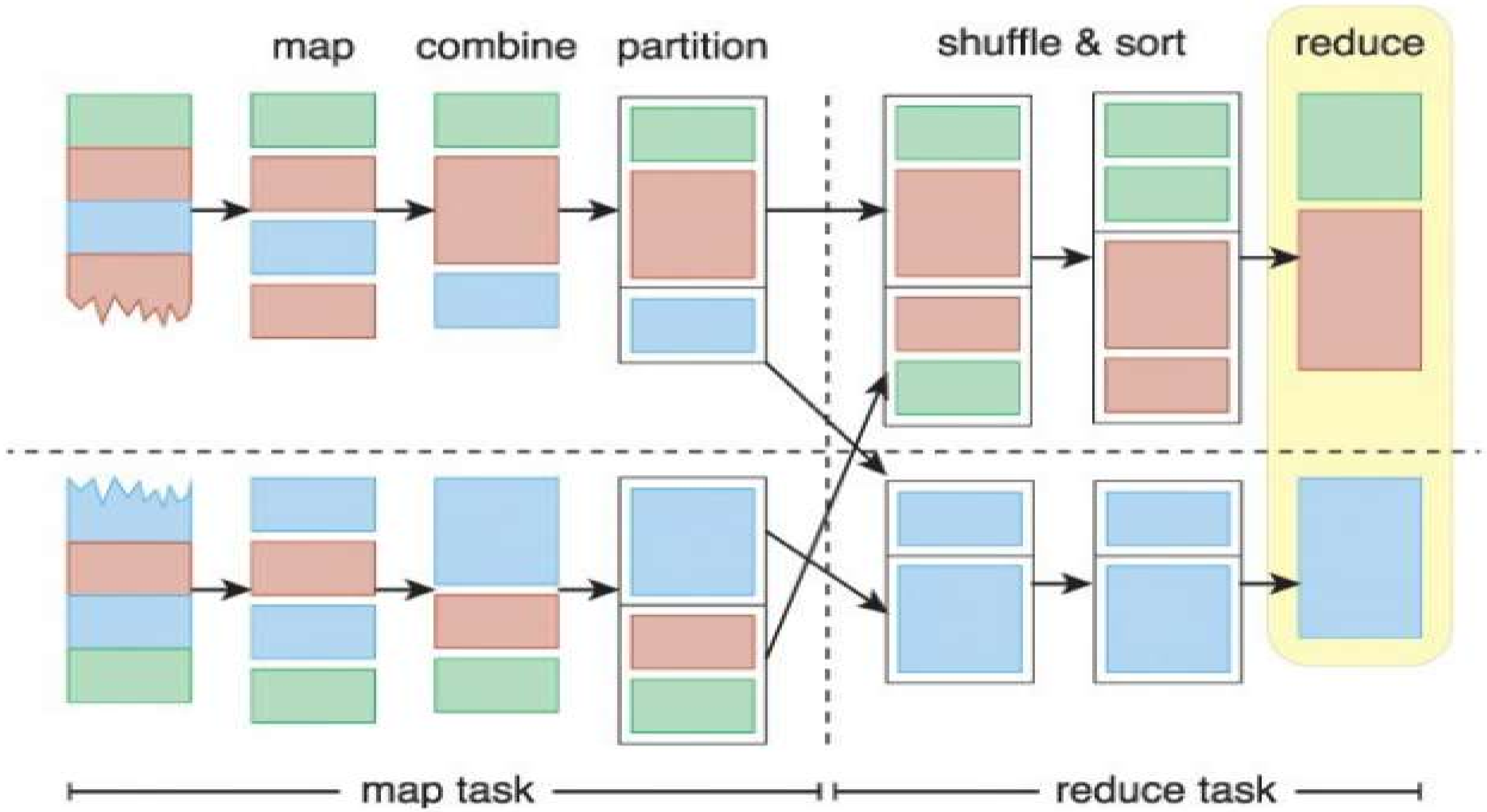


Reduce

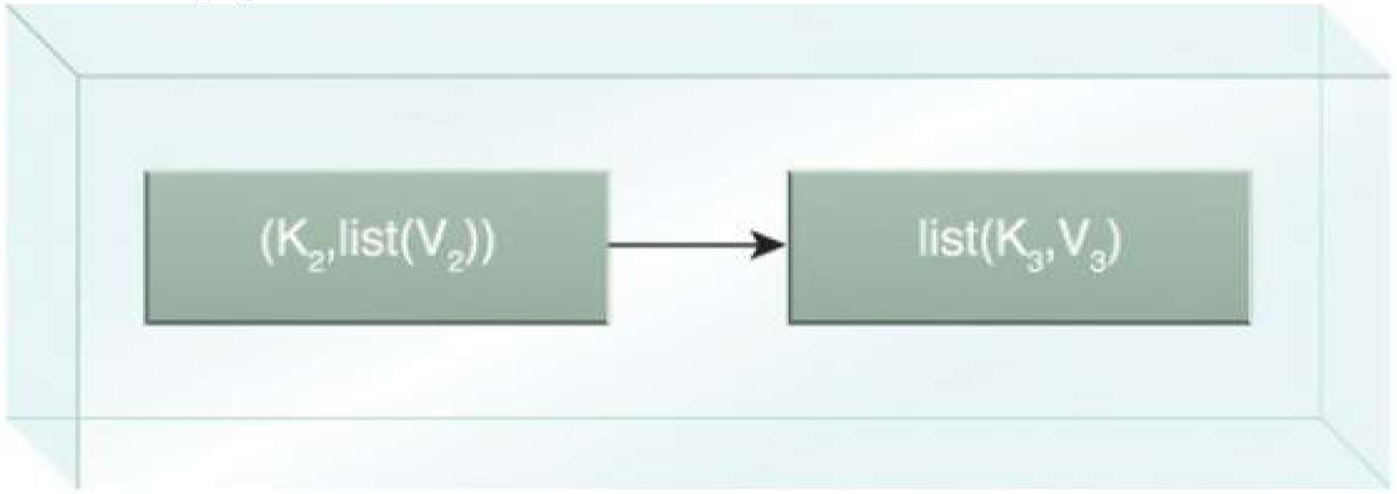
- Reduce is the final stage of the reduce task. Depending on the user-defined logic specified in the reduce function (reducer), the reducer will either further summarize its input or will emit the output without making any changes.
- In either case, for each key-value pair that a reducer receives, the list of values stored in the value part of the pair is processed and another key-value pair is written out.

- The output key can either be the same as the input key or a substring value from the input value, or another serializable user-defined object.
- The output value can either be the same as the input value or a substring value from the input value, or another serializable userdefined object.
- Note that just like the mapper, for the input key-value pair, a reducer may not produce any
- output key-value pair (filtering) or can generate multiple key-value pairs (demultiplexing).
- The output of the reducer, that is the key-value pairs, is then written out as a separate file —one file per reducer.

The reduce stage is the last stage of the reduce task



- The number of reducers can be customized. It is also possible to have a MapReduce job without a reducer, for example when performing filtering.
- Note that the output signature (key-value types) of the map function should match that of the input signature (key-value types) of the reduce/combine function.



A Simple MapReduce Example

- MapReduce steps are following

1. The input (sales.txt) is divided into two splits.

2. Two map tasks running on two different nodes, Node A and Node B, extract product and quantity from the respective split’s records in parallel. The output from each map function is a key-value pair where product is the key while quantity is the value.

3. The combiner then performs local summation of product quantities.

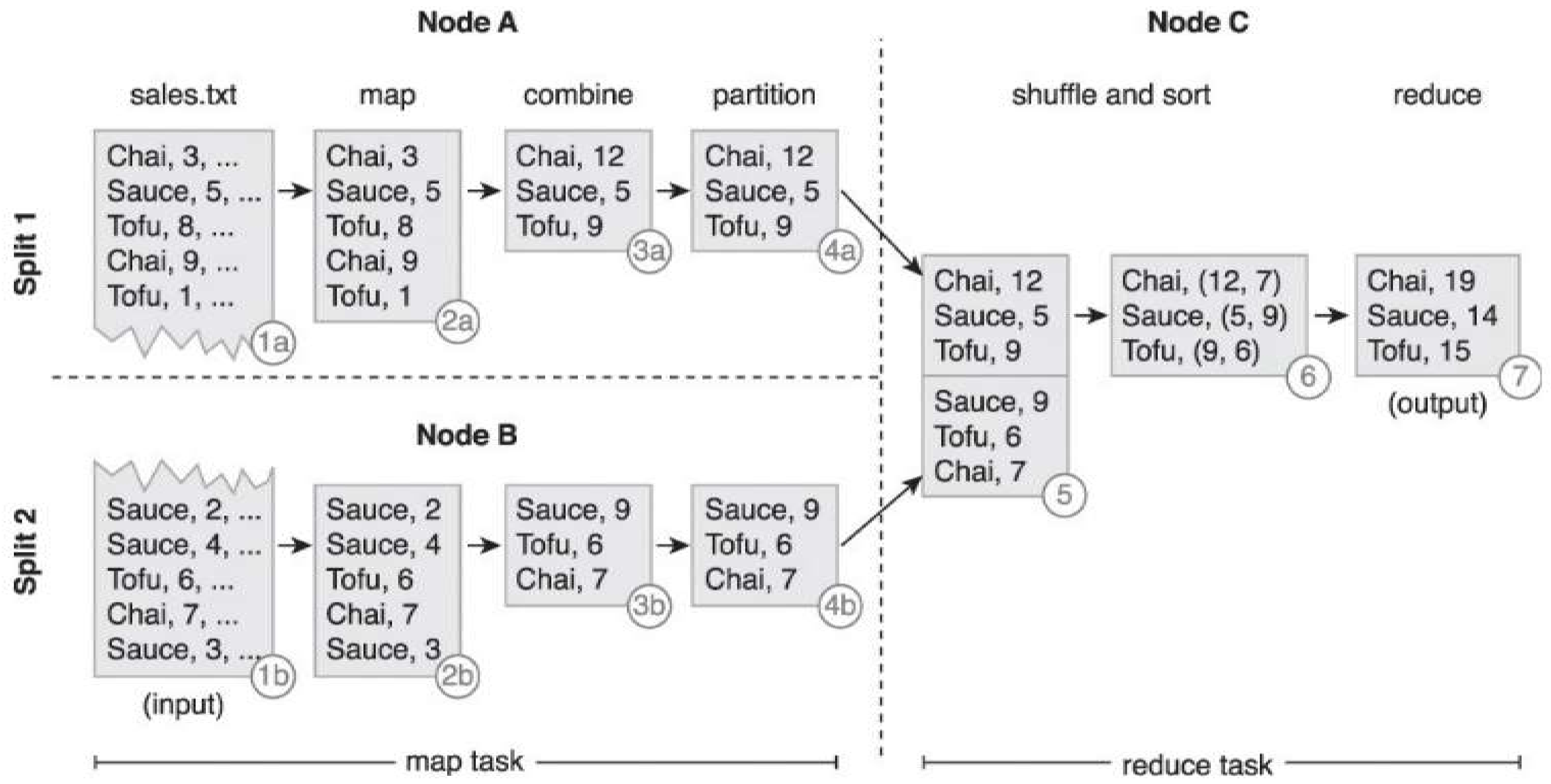
4. As there is only one reduce task, no partitioning is performed.

5. The output from the two map tasks is then copied to a third node, Node C, that runs the shuffle stage as part of the reduce task.

6. The sort stage then groups all quantities of the same product together as a list.

7. Like the combiner, the reduce function then sums up the quantities of each unique product in order to create the output.

An example of MapReduce in action



Node A
Node C
sales.txt
map
combine
partition
shuffle and sort
reduce
Split 1
Chai, 3, ...
Sauce, 5, ...
Tofu, 8, ...
Chai, 9, ...
Tofu, 1, ...
1a
Chai, 3
Sauce, 5
Tofu, 8
Chai, 9
Tofu, 1
2a
Chai, 12
Sauce, 5
Tofu, 9
3a
Chai, 12
Sauce, 5
Tofu, 9
4a
Chai, 12
Sauce, 5
Tofu, 9
Sauce, 9
Tofu, 6
Chai, 7
5
Chai, (12, 7)
Sauce, (5, 9)
Tofu, (9, 6)
6
Chai, 19
Sauce, 14
Tofu, 15
7
(output)
Node B
Split 2
Sauce, 2, ...
Sauce, 4, ...
Tofu, 6, ...
Chai, 7, ...
Sauce, 3, ...
1b
(input)
Sauce, 2
Sauce, 4
Tofu, 6
Chai, 7
Sauce, 3
2b
Sauce, 9
Tofu, 6
Chai, 7
3b
Sauce, 9
Tofu, 6
Chai, 7
4b
map task
reduce task