Università degli Studi di Messina

Data Science

Big Data Acquisition MapReduce

Prof. Daniele Ravi

Honorary Associate Professor
University College London (UCL) – UK
Department of Computer Science

dravi@unime.it

Recap from Last Lecture

- Large-scale data analysis poses new challenges on traditional single-node architecture
  - Cluster computing architecture (scaling out)
- Need for novel frameworks supporting clustered architectures:
  - Reliability
  - Network communication
  - Distributed programming model

MapReduce

- A **programming model** (and an associated implementation) for processing big data sets with parallel, distributed algorithms on a cluster
- It addresses the **3** main **challenges** of cluster architecture
  - Stores data **redundantly** on multiple nodes to ensure data/computation availability
  - Moves computation **close to data** to minimize network data transfers
  - Provides a simple computational model to **hide all the complexities** of the distributed environment

MapReduce: Distributed File System

- Redundant storage infrastructure
- Provides **global file namespace** and availability across nodes in a cluster
- Well-known implementations:
  - Google GFS
  - Hadoop HDFS
- Usage pattern:
  - Large files (100 GB / 10 TB)
  - Many "read" operations vs. few "updates" (append)

MapReduce: Distributed File System

- **3** main **components**:
  - Chunk Servers
  - Master Nodes
  - Client API

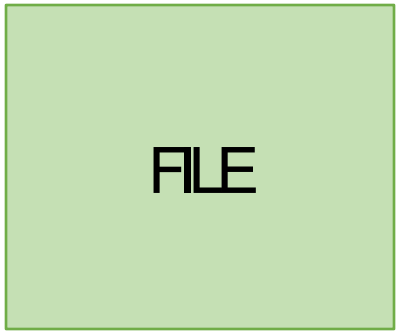
MapReduce: Distributed File System

- 3 main components:
  - **Chunk Servers**
  - Master Nodes
  - Client API

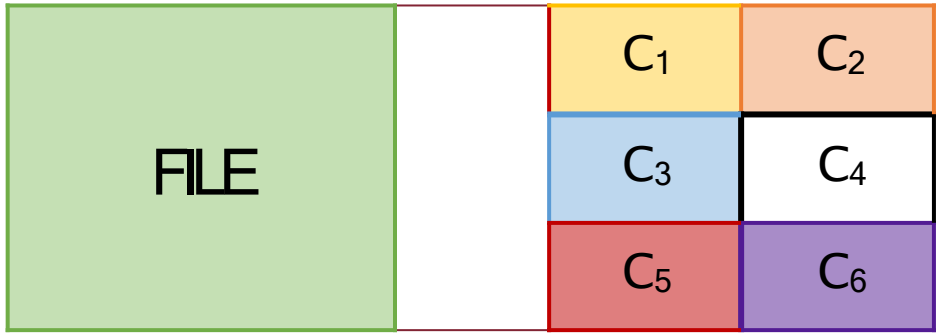
Distributed File System: Chunk Servers

- Large data files are split into contiguous "**chunks**" of fixed size
  - e.g., 16÷64 MB
- Each chunk is **replicated** across multiple nodes (chunk servers)
  - 2 or 3 replicas per chunk
  - Each replica on a different node
  - At least, one replica on a different rack
- Chunk servers act also as **computational servers**
  - move computation to data

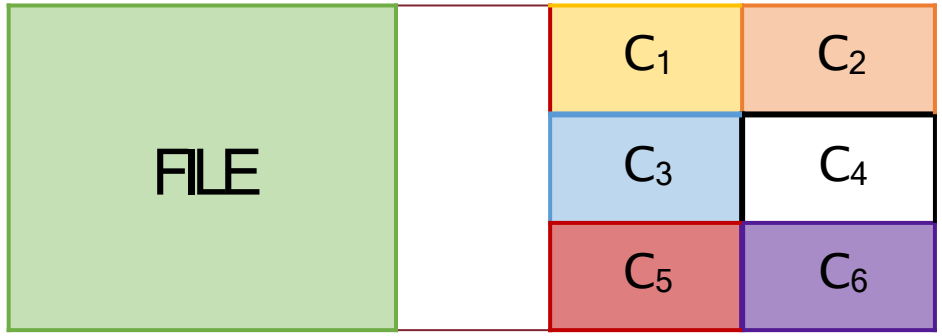
Distributed File System: Chunk Servers

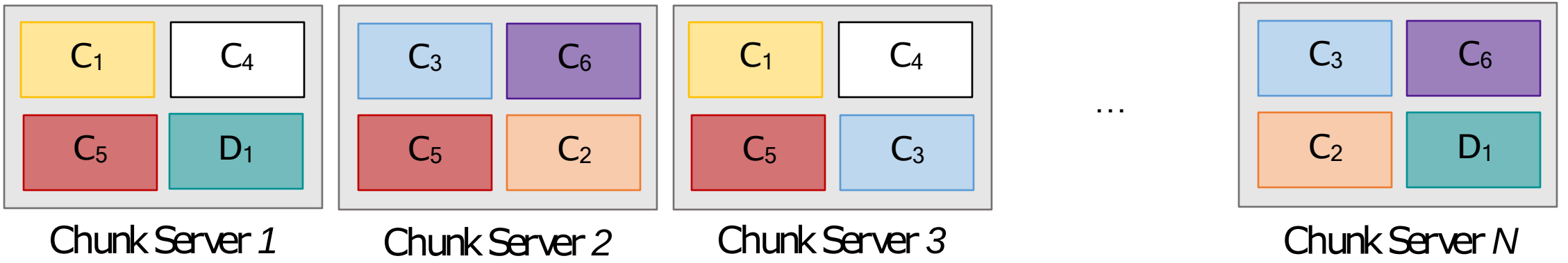


Distributed File System: Chunk Servers



Distributed File System: Chunk Servers





MapReduce: Distributed File System

- 3 main components:
  - Chunk Servers
  - **Master Nodes**
  - Client API

Distributed File System: Master Node

- Stores **metadata** about files in the distributed filesystem
  - How many chunks each file is split into
  - Where each of those chunks are located
- Possibly **replicated** to avoid single-point of failure

MapReduce: Distributed File System

- 3 main components:
  - Chunk Servers
  - Master Nodes
  - **Client API**

Distributed File System: Client API

- Allows clients to **access data** stored on chunk servers

- Client asks the Master Node through the API where a particular chunk is located
- The Master Node replies with the information needed

- Afterwards, any communication between the client and the chunk server storing the data happens directly (i.e., without the Master Node)

MapReduce: Programming Model

- MapReduce is a **style of programming** designed for:
  - Easy parallel programming
  - Invisible management of hardware and software failures
  - Easy management of very-large-scale data

- It has **several implementations**, including
  - Hadoop, Spark (used in this class), Flink, and the original Google implementation just called "MapReduce"

MapReduce: Intuition through an Example

- Suppose you are given avery large text document (e.g., 10TB)
  - The text document clearly does not fit into main memory!

- **Word Counting Task**: compute how many times each individual word appears in the document

- Possible applications:
  - Analysis of web/query logs
  - Statistical language modeling

MapReduce: Intuition through an Example

- The result of the task will be a list of (word, count) pairs

- 2 possible scenarios:
  - The total number of (word, count) pairs fit into main memory
  - The total number of (word, count) pairs **does not** fit into main memory

Word Counting: Result Fits into Main Memory

Contrary to popular belief, Lorem Ipsum is not simply random text.
It has roots in a piece of classical Latin literature from 45 BC, making it over 2000 years old. Richard McClintock, a Latin professor at Hampden-Sydney College in Virginia, looked up one of the more obscure Latin words, consectetur, from a Lorem Ipsum passage, and going through the cites of the word in classical literature, discovered the undoubtable source. Lorem Ipsum comes from sections 1.10.32 and 1.10.33 of "de Finibus Bonorum et Malorum" (The Extremes of Good and Evil) by Cicero, written in 45 BC.
This book is a treatise on the theory of ethics, very popular during the Renaissance. The first line of Lorem Ipsum, "Lorem ipsum dolor sit amet..", comes from a line in section 1.10.32.

doc.txt

Word Counting: Result Fits into Main Memory

Contrary to popular belief, Lorem Ipsum is not simply random text.
It has roots in a piece of classical Latin literature from 45 BC, making it over 2000 years old. Richard McClintock, a Latin professor at Hampden-Sydney College in Virginia, looked up one of the more obscure Latin words, consectetur, from a Lorem Ipsum passage, and going through the cites of the word in classical literature, discovered the undoubtable source. Lorem Ipsum comes from sections 1.10.32 and 1.10.33 of "de Finibus Bonorum et Malorum" (The Extremes of Good and Evil) by Cicero, written in 45 BC.
This book is a treatise on the theory of ethics, very popular during the Renaissance. The first line of Lorem Ipsum, "Lorem ipsum dolor sit amet..", comes from a line in section 1.10.32.

`doc.txt`

Initialize an empty hash map/table

word	count

Word Counting: Result Fits into Main Memory

Contrary to popular belief, Lorem Ipsum is not simply random text.

It has roots in a piece of classical Latin literature from 45 BC, making it over 2000 years old. Richard McClintock, a Latin professor at Hampden-Sydney College in Virginia, looked up one of the more obscure Latin words, consectetur, from a Lorem Ipsum passage, and going through the cites of the word in classical literature, discovered the undoubtable source.

Lorem Ipsum comes from sections 1.10.32 and 1.10.33 of "de Finibus Bonorum et Malorum" (The Extremes of Good and Evil) by Cicero, written in 45 BC.

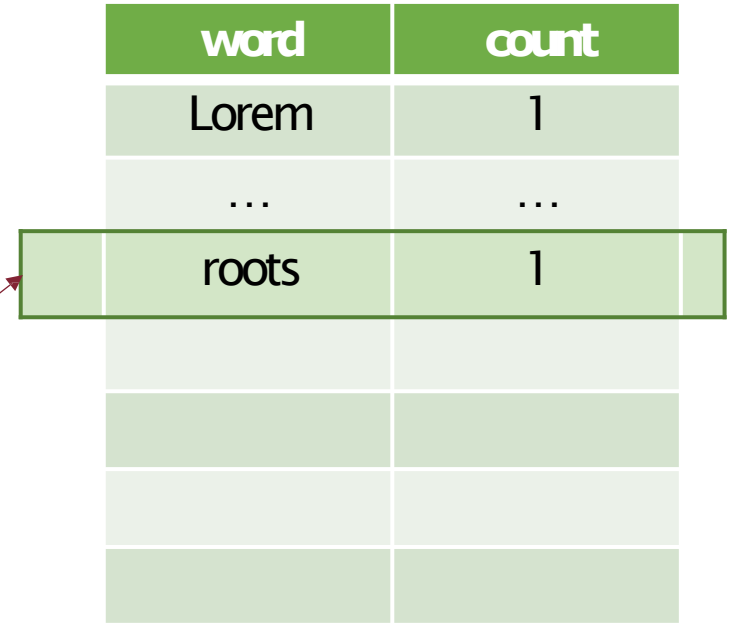
This book is a treatise on the theory of ethics, very popular during the Renaissance. The first line of Lorem Ipsum, "Lorem ipsum dolor sit amet..", comes from a line in section 1.10.32.

Process one line at a time

word	count
Lorem	1
…	…

Word Counting: Result Fits into Main Memory

It has roots in a piece of classical Latin literature from 45 BC, making it over 2000 years old. Richard McClintock, a Latin professor at Hampden–Sydney College in Virginia, looked up one of the more obscure Latin words, consectetur, from a Lorem Ipsum passage, and going through the cites of the word in classical literature, discovered the undoubtable source.



add new entry

Case 1: this is the first time we see the current word

Word Counting: Result Fits into Main Memory

It has roots in a piece of classical Latin literature from 45 BC, making it over 2000 years old. Richard McClintock, a Latin professor at Hampden-Sydney College in Virginia, looked up one of the more obscure Latin words, consectetur, from a Lorem Ipsum passage, and going through the cites of the word in classical literature, discovered the undoubtable source.

update existing entry

word	count
Lorem	2
…	…
roots	1

Case 2: we have already seen it the current word

Word Counting: Result Does Not Fit into Main Memory

- Use a mixture of simple scripting and UNIX command line tools

```
> print_words(doc.txt) | sort | uniq -c
```

`print_words` is a simple script which just prints each word of **`doc.txt`** to **`stdout`**, one per line

Word Counting: Result Does Not Fit into Main Memory

- Use a mixture of simple scripting and UNIX command line tools

```
> print_words(doc.txt) | sort | uniq -c
```

`print_words` is a simple script which just prints each word of `doc.txt` to `stdout`, one per line

- This solution nicely fits the MapReduce philosophy! We'll see how

Word Counting: Result Does Not Fit into Main Memory

- Use a mixture of simple scripting and UNIX command line tools

```
> print_words(doc.txt) | sort | uniq -c
```

`print_words` is a simple script which just prints each word of `doc.txt` to `stdout`, one per line

- This solution nicely fits the MapReduce philosophy! We'll see how

Note:
UNIX `sort` utility uses an external merge sorting algorithm
and therefore it doesn't require the data to be sorted to fit entirely in main memory

MapReduce: Steps

- **Input:** a set of (key, value) pairs
- **Output:** another set of (key, value) pairs
- Programmer defines **2 methods**:
  - `map`
  - `reduce`
- An intermediate `shuffle` step is implicitly provided by the framework

MapReduce: Map Phase

- The input data is split into smaller chunks and distributed across different nodes in the cluster.
- Each node runs a **Map function**, which processes its assigned chunk of data.
- The **Map function** takes in a key-value pair and outputs an intermediate set of key-value pairs. This intermediate output is typically organized by key.
- For example:
  - If you're counting the frequency of words in documents, the **Map function** could take a **line of text** and output key-value pairs where the key is a word and the value is 1 (i.e., ("word", 1)).

MapReduce: **Shuffle & Sort**

- After the Map phase, the intermediate key-value pairs are shuffled and grouped by key.
- This means all the values corresponding to a particular key (from different nodes) are **collected together**.

MapReduce: Reduce Phase

- The **Reduce function** is then applied to each group of intermediate results (i.e., each key and its associated list of values).

- The **Reduce function** aggregates these values and outputs the final result.

- For example:
- In our example, the **Reduce function** would take all the occurrences of a word (e.g., ("word", [1, 1, 1])) and sum them to get the total count for that word (e.g., ("word", 3)).

MapReduce: Steps (More Formally)

- Input key-value pairs: $\{(k_1, v_1), (k_2, v_2), \ldots, (k_M, v_M)\}$
- For instance, if you're processing documents, k_i might be an index of a line, v_i could be the content of that line.
- **map**$(k_i, v_i) \rightarrow \{(k_i', v_i')\}^*$
  - Takes an input key-value pair and outputs a set of 0 or more new, intermediate key-value pairs
  - One **map** function call for each input key-value pair (k_i, v_i)
    - map task -> multiple map calls executed in parallel on a subset of the input key-value pairs
- **reduce**$(k_i', \{v_i'\}^*) \rightarrow \{(k_i', v_i'')\}^*$
  - All values v_i' associated with the same key k_i' are reduced together
  - One **reduce** function call for each unique key k_i'

Word Counting: Map (`print_words`)

```
> print_words(doc.txt)
```

- Resembles the role of **`map`** function in MapReduce paradigm

Word Counting: Map (`print_words`)

```
> print_words(doc.txt)
```

- Resembles the role of **`map`** function in MapReduce paradigm
- A **`map`** function:
  - takes as input the original data (e.g., a chunk of the whole **`doc.txt`** file)
  - produces as output something out of the data called **intermediate keys** (e.g., a word for each line in the chunk)

Word Counting: Shuffle (`sort`)

```
> print_words(doc.txt) | sort
```

- The intermediate keys generated by the map function are sorted and shuffled

Word Counting: Shuffle (`sort`)

```
> print_words(doc.txt) | sort
```

- The intermediate keys generated by the map function are sorted and shuffled
- Note that intermediate keys are not unique!

Word Counting: Shuffle (`sort`)

```
> print_words(doc.txt) | sort
```

- The intermediate keys generated by the map function are sorted and shuffled
- Note that intermediate keys are not unique!
- For example, **`print_words`** may print out the same word multiple times

Word Counting: Reduce (`uniq -c`)

```
> print_words(doc.txt) | sort | uniq -c
```

- Resembles the role of `reduce` function in MapReduce paradigm

Word Counting: Reduce (`uniq -c`)

```
> print_words(doc.txt) | sort | uniq -c
```

- Resembles the role of `reduce` function in MapReduce paradigm
- A `reduce` function:
  - takes as input the groups of intermediate keys
  - computes an aggregating/filtering/transforming function over those keys
  - persists out the result

MapReduce: Input Key-Value Pairs

Contrary to popular belief, Lorem Ipsum is not simply random text.
It has roots in a piece of classical Latin literature from 45 BC, making it over 2000 years old. Richard McClintock, a Latin professor at Hampden-Sydney College in Virginia, looked up one of the more obscure Latin words, consectetur, from a Lorem Ipsum passage, and going through the cites of the word in classical literature, discovered the undoubtable source. Lorem Ipsum comes from sections 1.10.32 and 1.10.33 of "de Finibus Bonorum et Malorum" (The Extremes of Good and Evil) by Cicero, written in 45 BC.

…

This book is a treatise on the theory of ethics, very popular during the Renaissance. The first line of Lorem Ipsum, "Lorem ipsum dolor sit amet..", comes from a line in section 1.10.32.

record ID 1

Contrary to popular belief, Lorem Ipsum is not simply random text.

MapReduce: Input Key-Value Pairs

Contrary to popular belief, Lorem Ipsum is not simply random text.
It has roots in a piece of classical Latin literature from 45 BC, making it over 2000 years old. Richard McClintock, a Latin professor at Hampden-Sydney College in Virginia, looked up one of the more obscure Latin words, consectetur, from a Lorem Ipsum passage, and going through the cites of the word in classical literature, discovered the undoubtable source.
Lorem Ipsum comes from sections 1.10.32 and 1.10.33 of "de Finibus Bonorum et Malorum" (The Extremes of Good and Evil) by Cicero, written in 45 BC.

…

This book is a treatise on the theory of ethics, very popular during the Renaissance. The first line of Lorem Ipsum, "Lorem ipsum dolor sit amet..", comes from a line in section 1.10.32.

record ID 1

Contrary to popular belief, Lorem Ipsum is not simply random text.

record ID 2

It has roots in a piece of classical Latin literature from 45 BC, making it over 2000 years old. Richard McClintock, a Latin professor at Hampden-Sydney College in Virginia, looked up one of the more obscure Latin words, consectetur, from a Lorem Ipsum passage, and going through the cites of the word in classical literature, discovered the undoubtable source.

MapReduce: Input Key-Value Pairs

Contrary to popular belief, Lorem Ipsum is not simply random text.
It has roots in a piece of classical Latin literature from 45 BC, making it over 2000 years old. Richard McClintock, a Latin professor at Hampden-Sydney College in Virginia, looked up one of the more obscure Latin words, consectetur, from a Lorem Ipsum passage, and going through the cites of the word in classical literature, discovered the undoubtable source. Lorem Ipsum comes from sections 1.10.32 and 1.10.33 of "de Finibus Bonorum et Malorum" (The Extremes of Good and Evil) by Cicero, written in 45 BC.

…

This book is a treatise on the theory of ethics, very popular during the Renaissance. The first line of Lorem Ipsum, "Lorem ipsum dolor sit amet..", comes from a line in section 1.10.32.

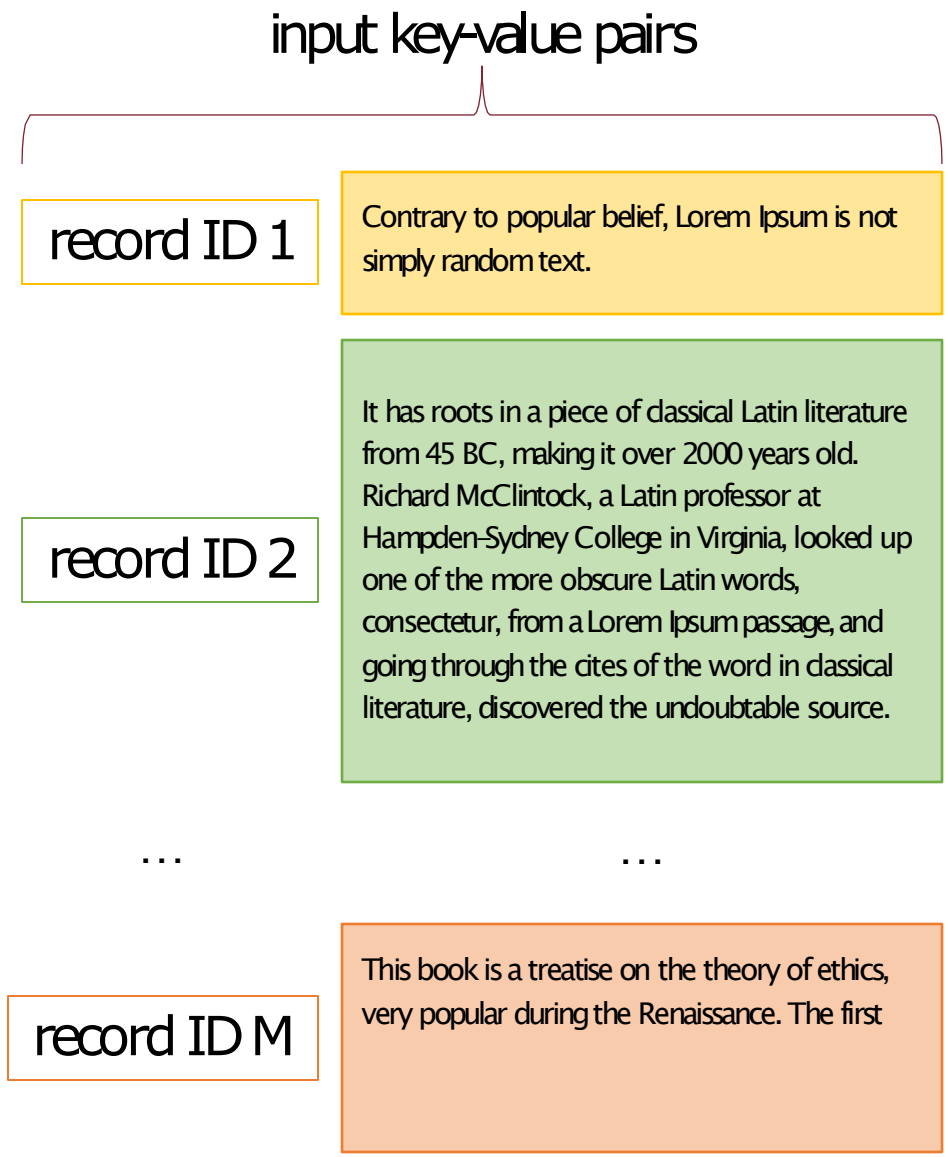
record ID 1	Contrary to popular belief, Lorem Ipsum is not simply random text.
record ID 2	It has roots in a piece of classical Latin literature from 45 BC, making it over 2000 years old. Richard McClintock, a Latin professor at Hampden-Sydney College in Virginia, looked up one of the more obscure Latin words, consectetur, from a Lorem Ipsum passage, and going through the cites of the word in classical literature, discovered the undoubtable source.
…	…
record ID M	This book is a treatise on the theory of ethics, very popular during the Renaissance. The first line of Lorem Ipsum, "Lorem ipsum dolor sit amet..", comes from a line in section 1.10.32.

MapReduce: Input Key-Value Pairs

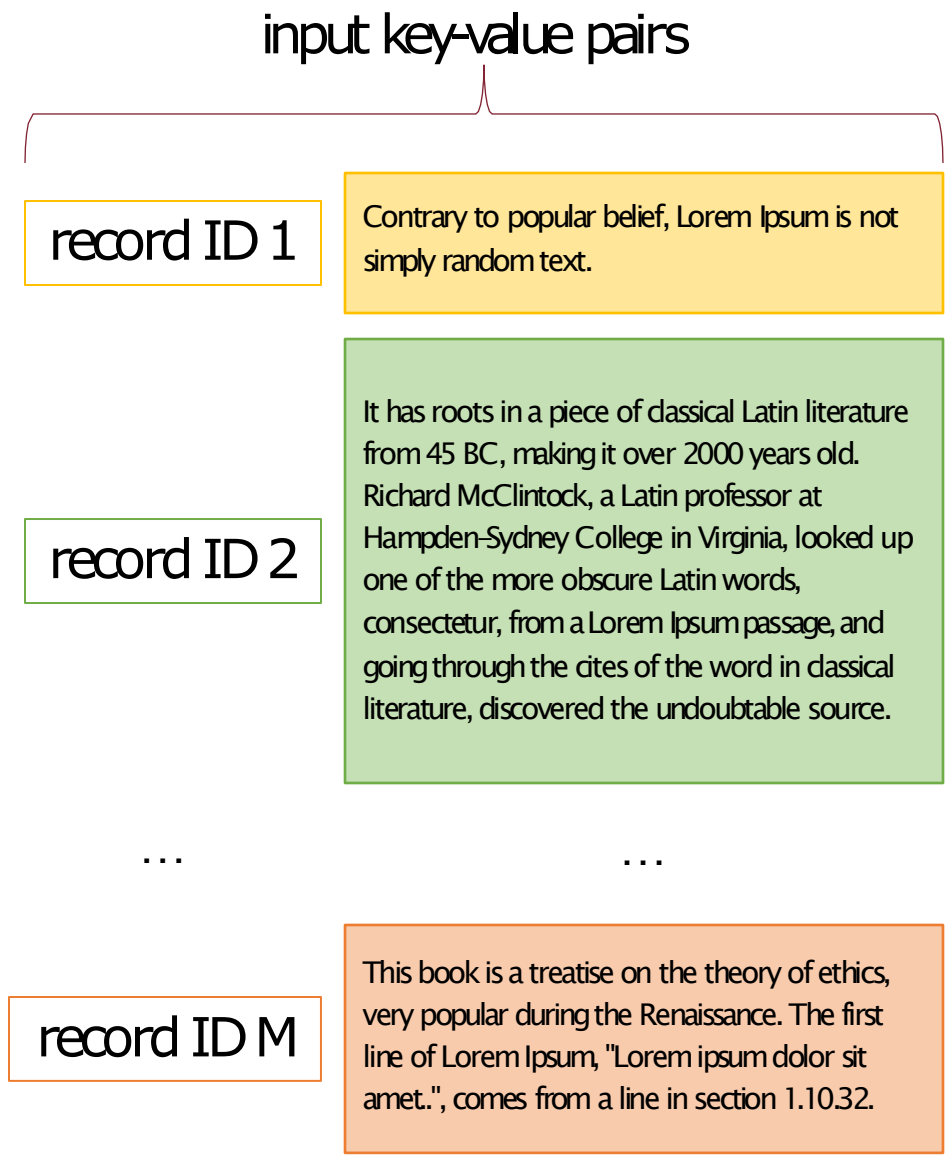
Contrary to popular belief, Lorem Ipsum is not simply random text.
It has roots in a piece of classical Latin literature from 45 BC, making it over 2000 years old. Richard McClintock, a Latin professor at Hampden-Sydney College in Virginia, looked up one of the more obscure Latin words, consectetur, from a Lorem Ipsum passage, and going through the cites of the word in classical literature, discovered the undoubtable source. Lorem Ipsum comes from sections 1.10.32 and 1.10.33 of "de Finibus Bonorum et Malorum" (The Extremes of Good and Evil) by Cicero, written in 45 BC.

…

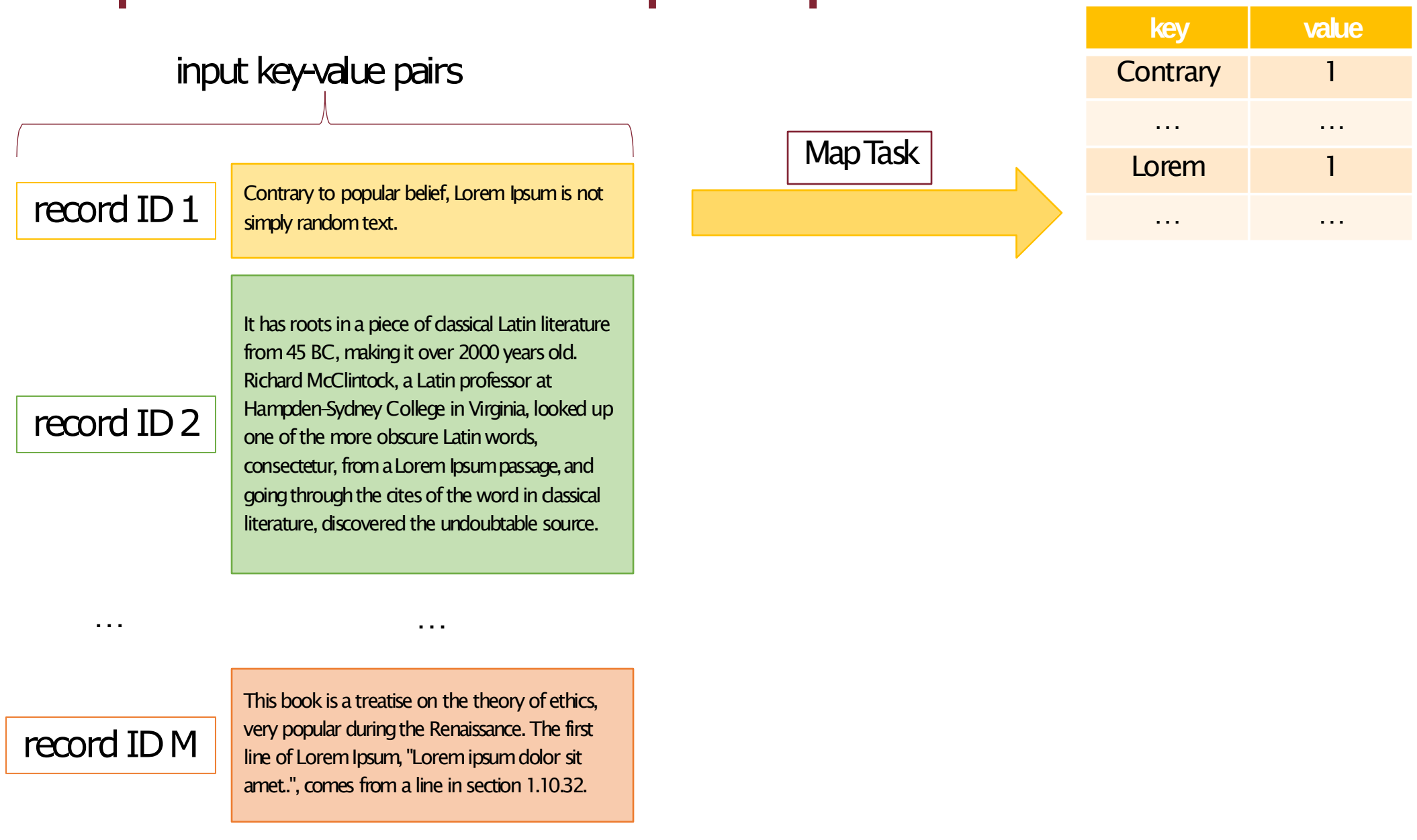
This book is a treatise on the theory of ethics, very popular during the Renaissance. The first line of Lorem Ipsum, "Lorem ipsum dolor sit amet..", comes from a line in section 1.10.32.



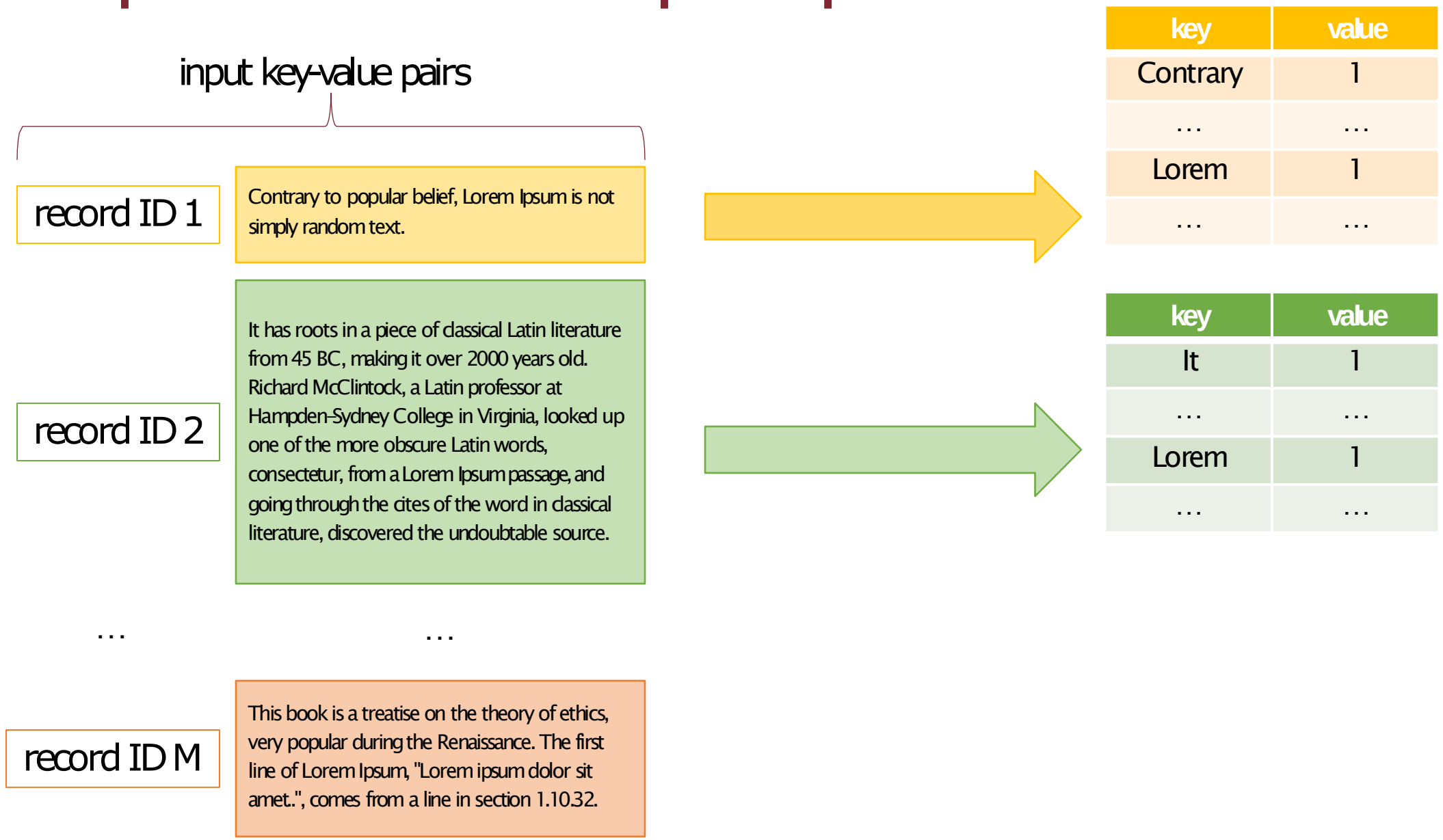
MapReduce: The Map Step



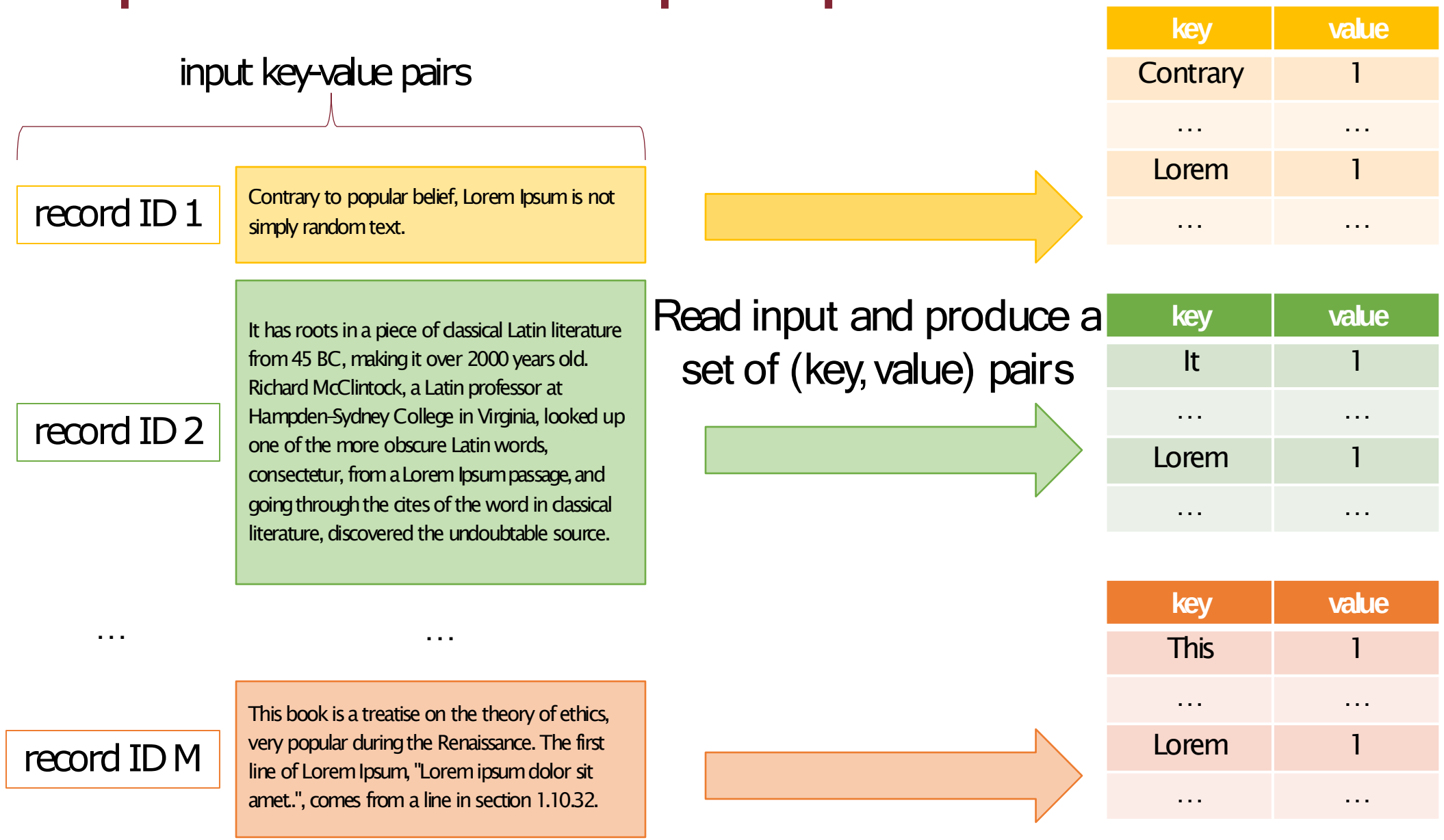
MapReduce: The Map Step



MapReduce: The Map Step



MapReduce: The Map Step



MapReduce: The Shuffle Step

key	value
Contrary	1
…	…
Lorem	1
…	…

key	value
It	1
…	…
Lorem	1
…	…

key	value
This	1
…	…
Lorem	1
…	…

MapReduce: The Shuffle Step

key	value
Contrary	1
…	…
Lorem	1
…	…

key	value
It	1
…	…
Lorem	1
…	…

key	value
This	1
…	…
Lorem	1
…	…

Collect (i.e., group) all pairs with the same key

key	value
A	1
A	1
…	…
Lorem	1
Lorem	1
Lorem	1

key	value
the	1
the	1
…	…
Ipsum	1
Ipsum	1
Ipsum	1

MapReduce: The Reduce Step

key	value
A	1
A	1
…	…
Lorem	1
Lorem	1
Lorem	1

key	value
the	1
the	1
…	…
Ipsum	1
Ipsum	1
Ipsum	1

Process all values belonging to a given key and output the result

key	value
A	2
…	…
Ipsum	3
…	…
Lorem	3
…	…
the	2
…	…
undoubtable	1

MapReduce: Word Counting Pseudocode

```
map(key, value):
# key: docID; value: text
    For each word in value:
        emit(word, 1)
```

```
reduce(key, values):
# key: word; values: iterator
    result = 0
    for each v in values:
        result += v
    emit(key, result)
```

Note:

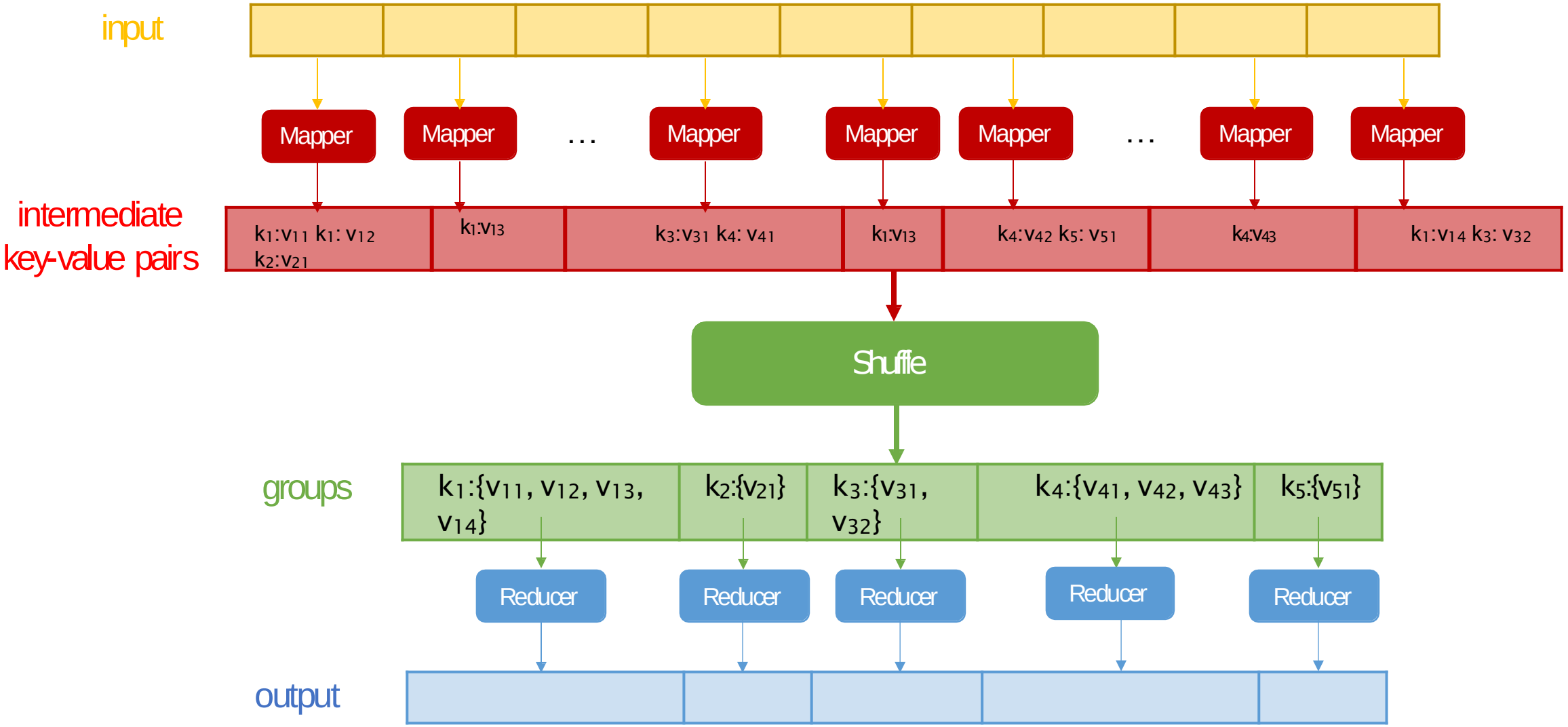
input (key,value) can be just a single pair as the actual split of the input is done transparently by the framework

MapReduce: PROs and CONs

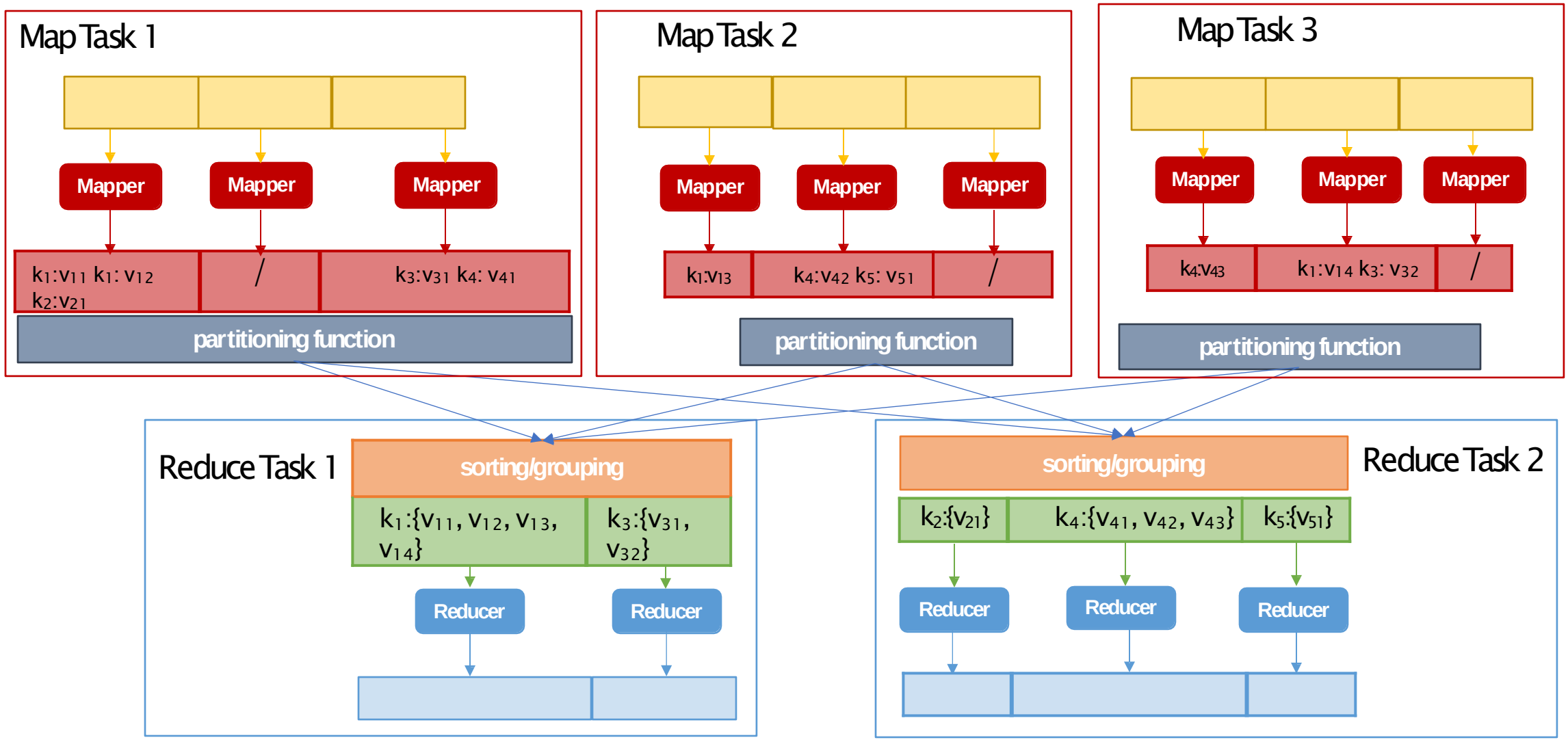
- MapReduce is **great** for:
  - Problems that require many sequential data access (from disk)
  - Large batch jobs (i.e., not interactive nor real time)

- MapReduce is **not suitable** for:
  - Problems that require random access to data
  - Working with graphs
  - Interdependent data

MapReduce on a Single-Node



MapReduce on a Cluster



MapReduce: The Infrastructure

- Remember! Programmer needs only to specify `map` and `reduce` functions
- Everything else is managed by the infrastructure
  - Input data partitioning (physical = chunk/block or logical = split)
  - Scheduling tasks across nodes of the cluster
  - Shuffling/group by of intermediate keys output by mappers
  - Handling node failures
  - Managing inter-node communications

Data Flow

- Both input and output are stored on the distributed file system (DFS)
  - MapReduce scheduler tries to allocate map tasks "close" to data
  - Each map task running on a node will be using the chunks of data that are stored on that node (chunk server)

- Intermediate results of map/reduce tasks are stored on local filesystem of each node
  - This is to avoid copies/replica of useless files across the cluster (DFS)

The Master Node

- Takes care of node coordination/orchestration
  - Status associated with each task (either map or reduce):
    - idle,
    - in-progress
    - completed
  - Idle tasks are eligible to be executed assoon asaworker node becomes available
  - When a map task completes it sends notification of that to the master node who propagates that information to the reducers
  - The master node periodically pings mappers/reducers to detect failures

Failure Detection

- Map worker node fails
  - All the map tasks completed or in-progress at the (failed) worker node are reset to idle
  - Idle map tasks will be eventually rescheduled later on other worker node(s)

- Reduce worker node fails
  - Only in-progress tasks are reset to idle (completed ones have already output to the DFS)
  - Idle reduce tasks will be eventually rescheduled later on other worker node(s)

- Master node fails -> The whole MapReduce job is aborted

How Many Map/Reduce Tasks?

- N = # nodes of the cluster; M = # map tasks; R = # reduce tasks
- Again, mostly transparent to the programmer
- Rule of thumb:
  - M >> N (in fact, one map task per DFS chunk is pretty common)
  - Having M >> N speeds up recovery from node failures
  - R < M (convenient to have the output spread across a limited number of nodes)
  - The number of reduce tasks is usually fewer than the number of map tasks because it's more convenient to have the final output spread across fewer nodes. Too many reduce tasks could result in unnecessary overhead and make it harder to collect and manage the final results.

Partition Function

- Controls how intermediate key-value pairs produced by mappers are distributed across (i.e., sent over) reducers
- Assuming R reducer nodes, default partition function is as simple as

```
hash(key) mod R
```

- Sometimes may be useful to override the default partition function with a custom one

Partition Function

- **Intermediate Key-Value Pairs**: After the mappers process input data, they output key-value pairs. These pairs are intermediate results that need to be grouped and processed by reducers.
- **Partitioning**: The process of sending these key-value pairs to different reducer nodes is called **partitioning**. Each reducer processes a subset of the intermediate data, usually grouped by keys.

- **Default Partition Function**: By default, MapReduce uses a **partition function** that distributes the data based on the key. The most common default function is:
    - hash(key)mod R
    - H**ash(key)**: Hash value for the key is a number.
    - **mod R**: Is the remainder when the hash value is divided by the number of reducer nodes, R. This ensures that the key-value pairs are distributed relatively evenly across the reducers.

Implementations

- **Google MapReduce**
  - Uses Google File System (GFS) for redundant storage
  - Not available outside Google
- **Hadoop**
  - Apache's open-source implementation of MapReduce
  - Uses Hadoop Distributed File System (HDFS)
  - Terminology: Master -> NameNode, Chunk Server -> DataNode
  - Hive/Pig -> SQL-like abstractions on top of Hadoop MapReduce

MapReduce as a Service

- Allows to rent computing by the hour along with other services like persistent storage

- Amazon's "Elastic Computing Cloud" (EC2) provides:
  - Stable Storage (S3)
  - Elastic MapReduce (EMR)

MapReduce: Criticisms

- **3 major limitations** of MapReduce:
  - Hard to program directly
    - Many problems are not easily described as map-reduce
  - Relies heavily on disk I/O communication which can become a bottleneck for performance.
    - **Latency**: MapReduce is optimized for batch processing and may not be suitable for real-time applications.
    - Persistence to disk slower than in-memory computation
  - MapReduce is not suitable for more complex operations composed of several map-reduce steps or iterative algorithms (e.g., machine learning),

Take-Home Message of Today

- MapReduce is a framework for distributed computing
- Typical implementations come with a suite of tools/services for reliably storing and processing large volumes of data
- Useful in all those situations where data need to be accessed sequentially
- May be hard to program and does not support well multiple map-reduce rounds