CES-27 Processamento Distribuído

MapReduce

Prof Juliana Bezerra
Prof Celso Hirata
Prof Vitor Curtis

Outline

- What is MapReduce?
- MapReduce Execution Overview
- Curiosities
  - Failure Tolerance
  - Task Granularity
  - Combiner Function
- Patterns and applications

MapReduce

- Reference paper: J. Dean and S. Ghemawat, “MapReduce: Simplified Data Processing on Large Clusters”, Google, 2004

- MapReduce is a **programming model** and an associated **implementation** for processing and generating **large data sets**

  - Inspired by the map and reduce primitives present in Lisp and many other functional languages

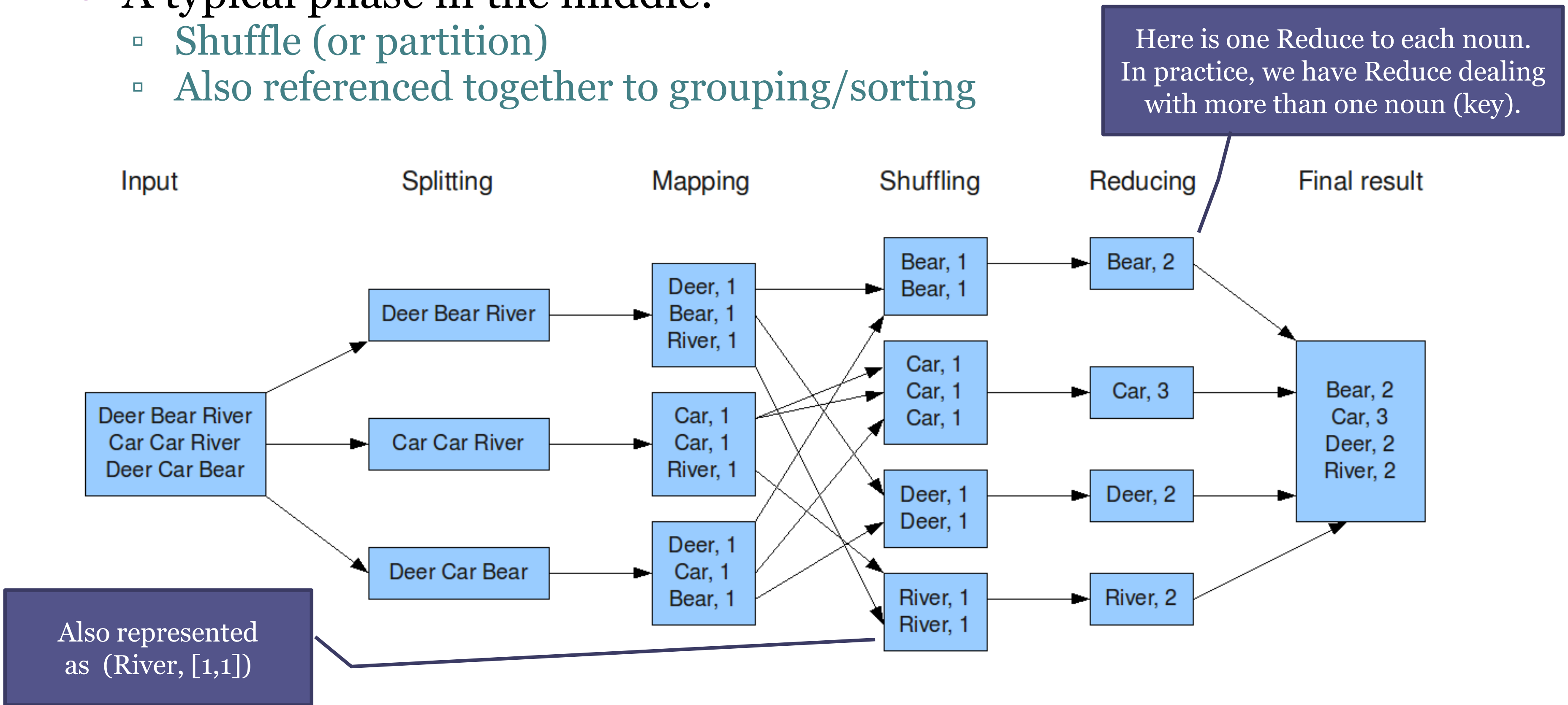
  - Map: (input shard) → intermediate(key/value pairs)
    - A **map** operation to each logical “record” in our input in order to compute a set of intermediate **key/value pairs**

  - Reduce: intermediate(key/value pairs) → result files
    - A **reduce** operation to all the values that shared the **same key**, in order to **combine the derived data** appropriately

  - Focus: high performance
    - … but hides the details of parallelization, fault-tolerance, data distribution and load balancing in a library

Word Count - A Typical Example

- Input: Large number of text documents (or a big one document)
- Operation: Compute word count across all the documents
- Solution
  - Mapper: For every word in a document, output (word, "1")
  - Reducer: Sum all occurrences of words and output (word, totalCount)
- A typical phase in the middle:
  - Shuffle (or partition)
  - Also referenced together to grouping/sorting

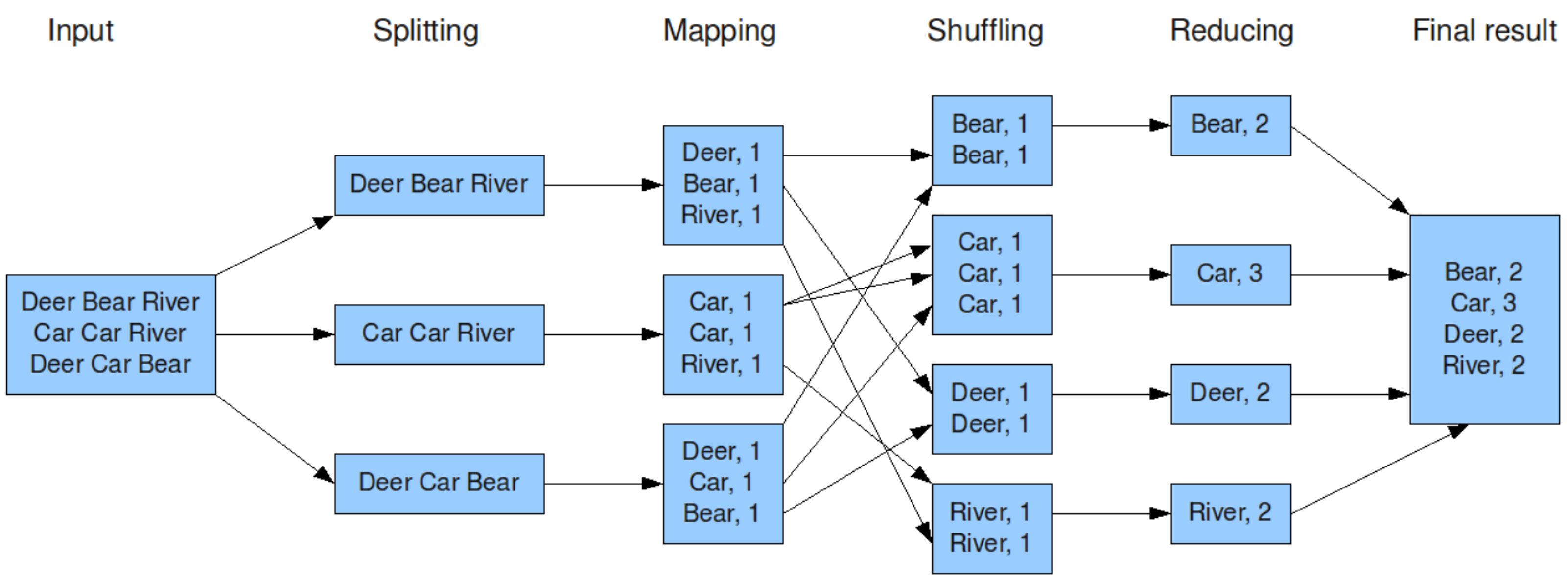


Word Count - A Typical Example

```
map(String input_key, String input_value):
  // input_key: document name
  // input_value: document contents
  for each word w in input_value:
    EmitIntermediate(w, "1");
```

```
reduce(String output_key, Iterator intermediate_values):
  // output_key: a word
  // intermediate values: list of counts
  int result = 0;
  for each v in intermediate_values:
    result += ParseInt(v);
  Emit(AsString(result));
```

Only v of the given output_key



Who has MapReduce?

- Google
  - Original proprietary implementation

- Apache Hadoop MapReduce
  - Most common (open-source) implementation
  - Built based on specs defined by Google
  - Proprietary solutions:
    - Amazon Elastic MapReduce (run on Amazon EC2)
    - IBM Hadoop

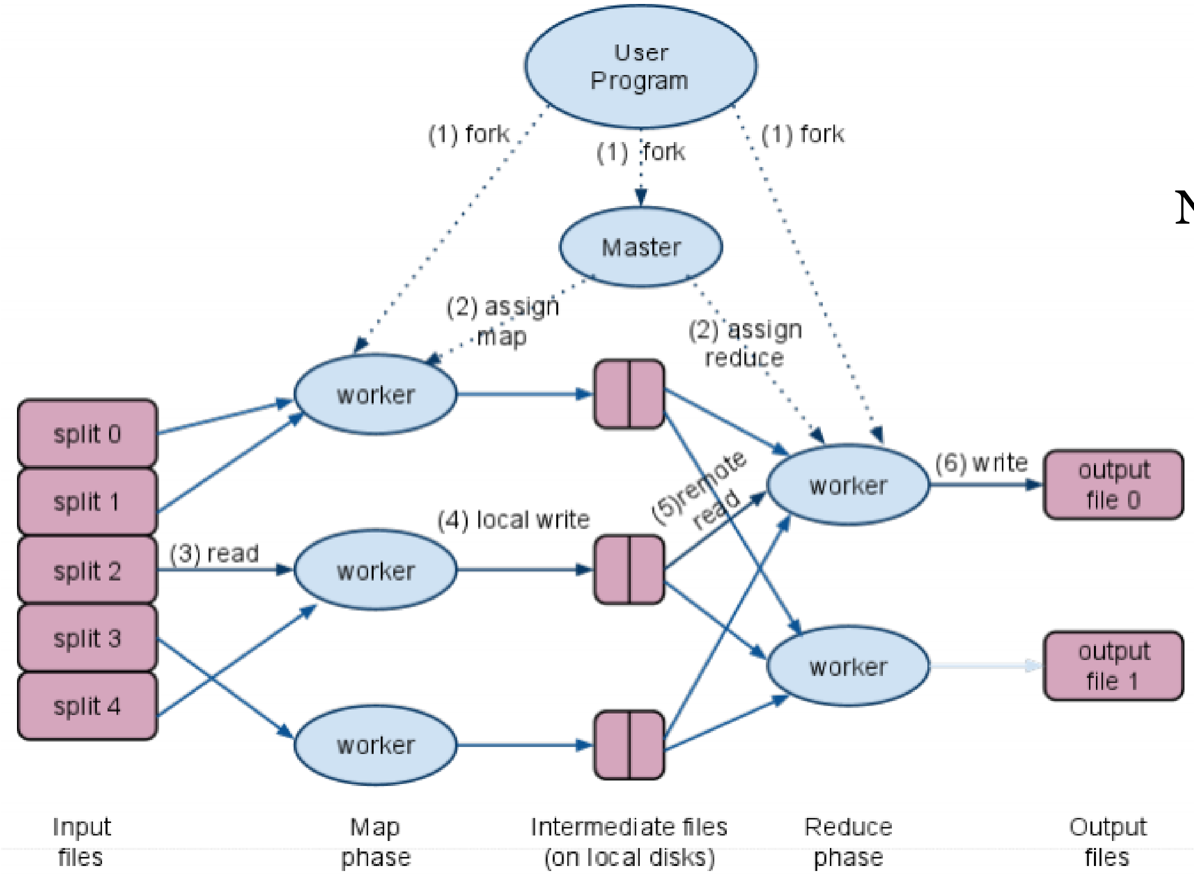
- Apache Spark
  - A fast and general open-source engine for large-scale data processing
  - It has MapReduce and other things
    - SPARK SQL, SPARK Streaming, MLlib (Machine Learning) and GraphX (graph processing)
  - Spark is capable to run programs up to 100x faster than Hadoop MapReduce <u>in memory</u>, or 10x faster <u>on disk</u>

Outline

- What is MapReduce?
- MapReduce Execution Overview
- Curiosities
  - Failure Tolerance
  - Task Granularity
  - Combiner Function
- Patterns and applications

MapReduce Execution Overview

- Keep in mind:
  - map task ≠ map worker ≠ user-defined *Map* function
  - reduce task ≠ reduce worker ≠ user-defined *Reduce* function

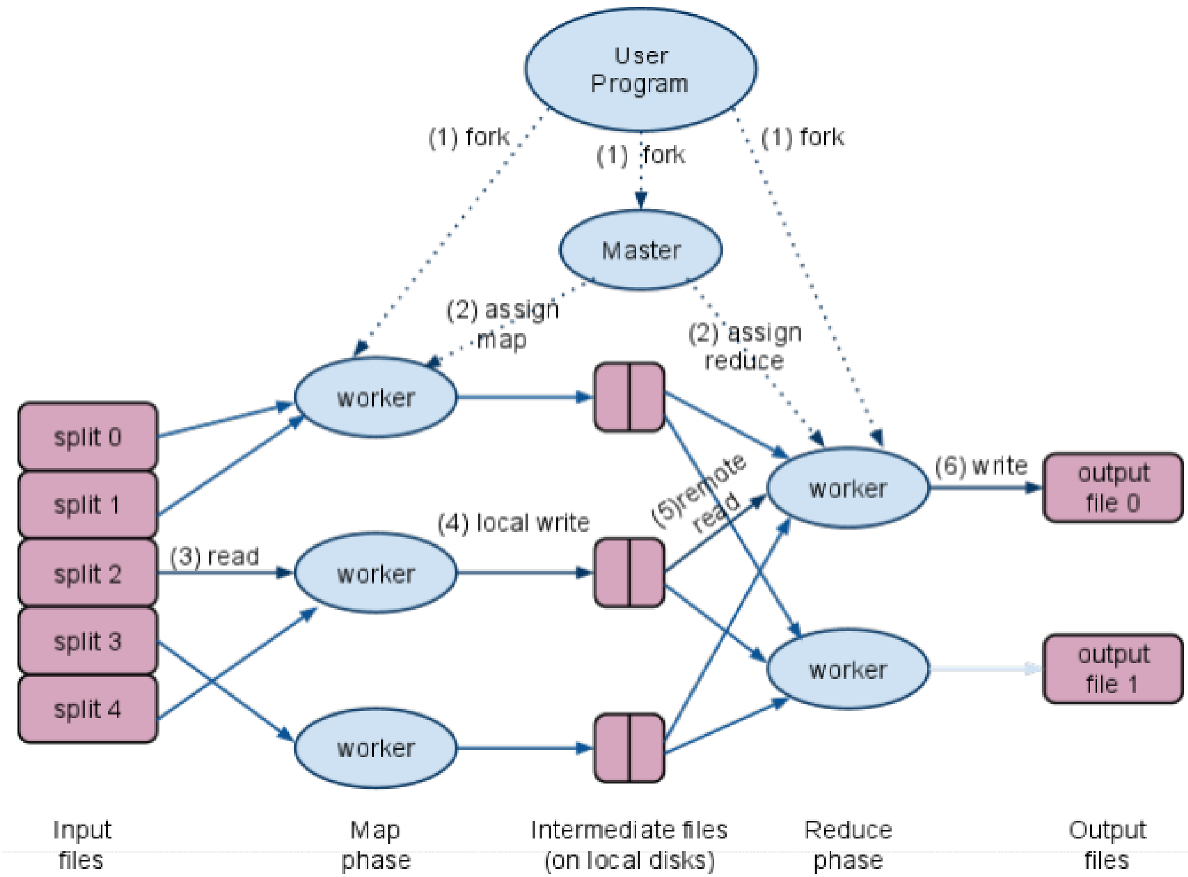


Note: task == job

MapReduce Execution Overview – Step 1

- The MapReduce library in the user program first **splits the input files into M pieces** of typically 16 megabytes to 64 megabytes (MB) per piece (controllable by the user via an optional parameter).

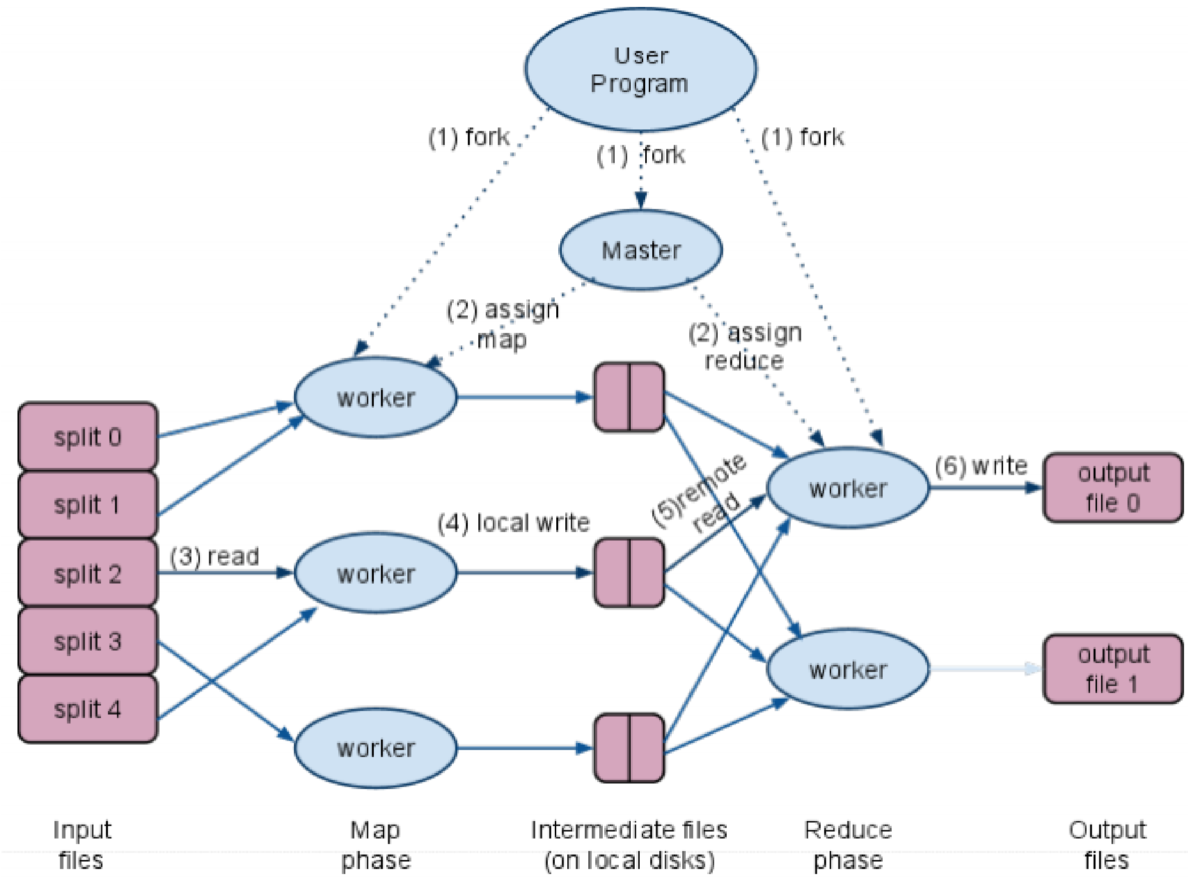
- It then starts up many copies of the program on a cluster of machines ⇨ **fork processes**



Note: We say split or shard

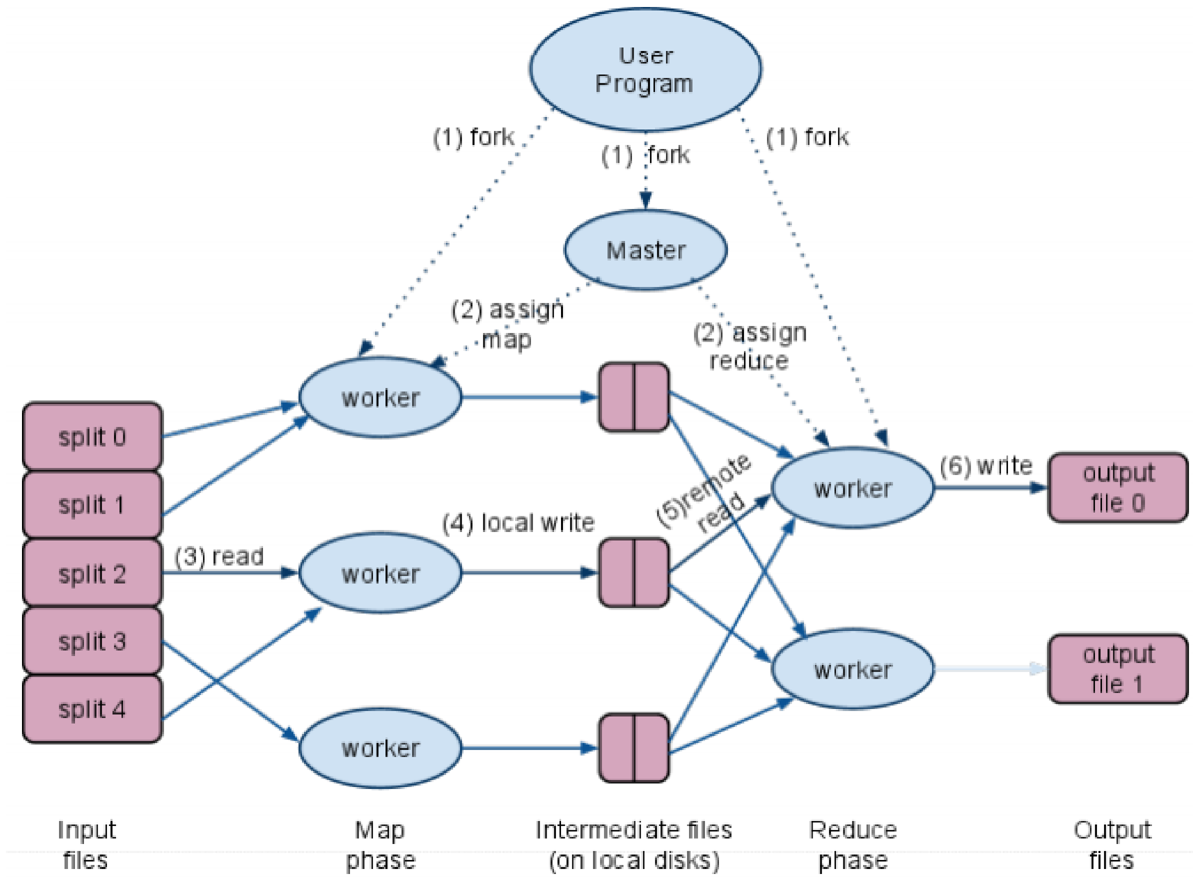
MapReduce Execution Overview – Step 2

- One of the copies of the program is special – the **master** (leader).
- The rest are **workers** that are assigned work by the master.
- There are **M map tasks** and **R reduce tasks** to assign.
- The **master** picks idle workers and **assigns** each one a map task or a reduce task.



MapReduce Execution Overview – Step 3

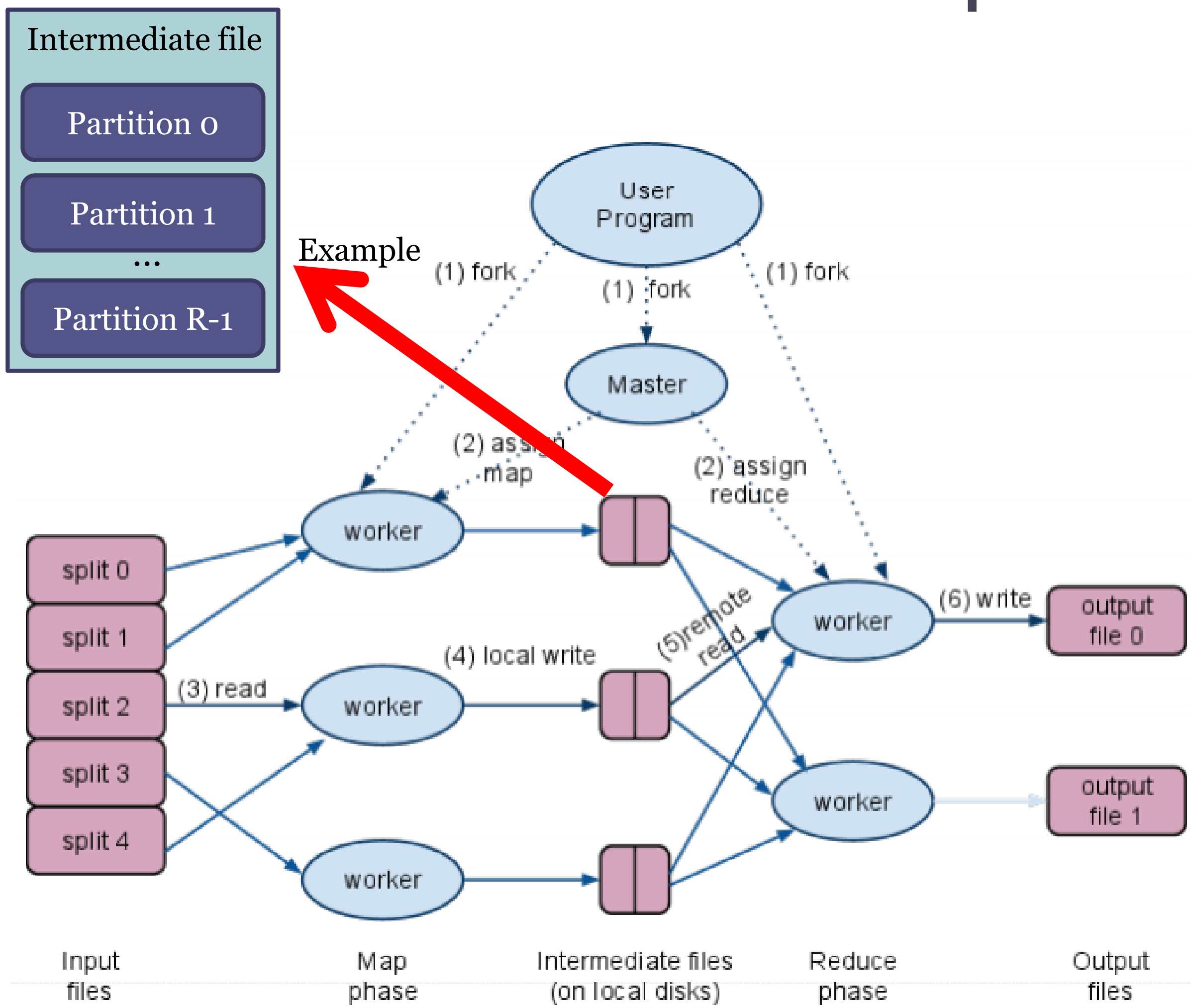
- A worker who is assigned a **map task** reads the contents of the corresponding input split.
- It calls user-defined ***Map* function**.
- The **intermediate key/value pairs** produced by the *Map* function are buffered in **memory**.



MapReduce Execution Overview – Step 4

- Periodically, the buffered pairs are written to **local disk**, partitioned into **R regions** by the **partitioning function**.
  - Default function: hash(key) mod R
  - “Map worker” partitions the data by keys
  - It also decides which of R reduce workers will work on which key

- The locations of these buffered pairs on the local disk are passed back to the **master**, who is responsible for forwarding these locations to the reduce workers.

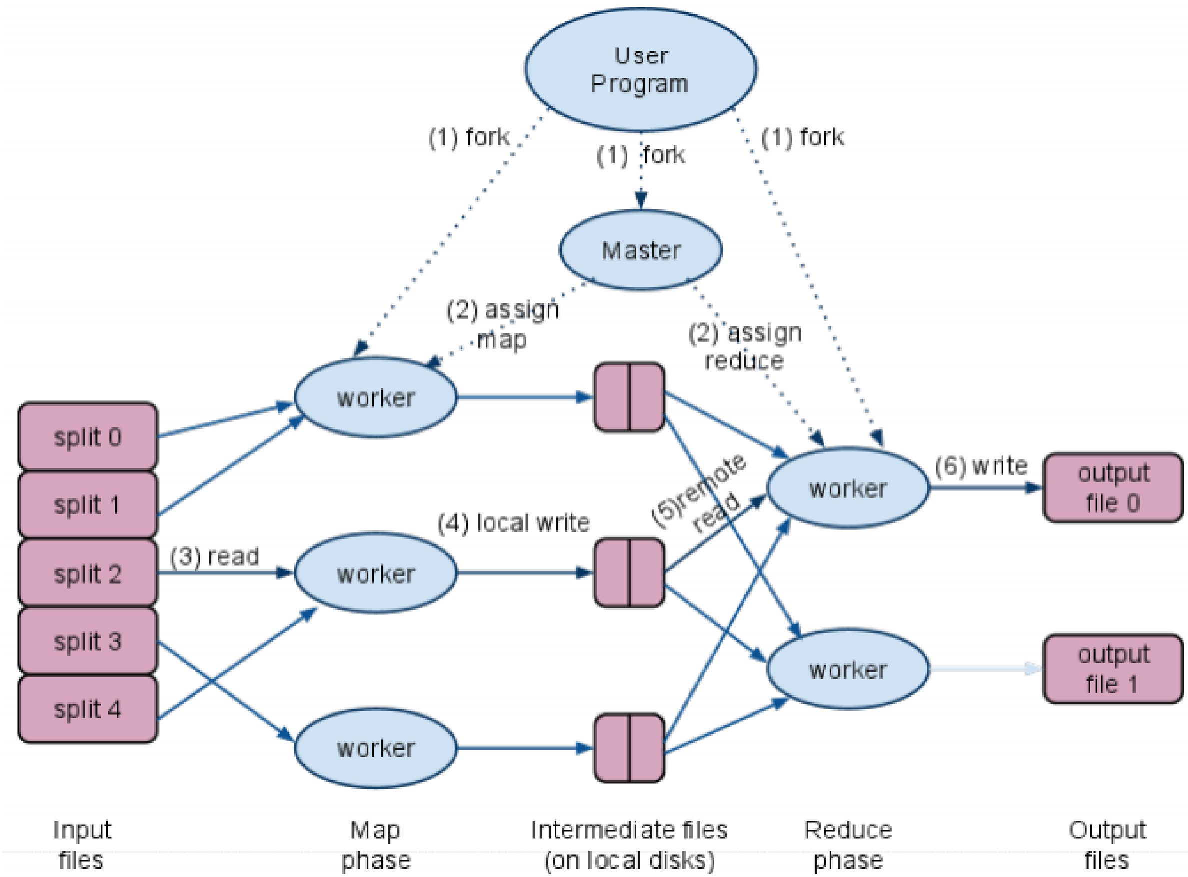


MapReduce Execution Overview – Step 5

- When a **reduce worker** is notified by the master about these locations, it uses **remote procedure calls** to **read** the buffered data from the **local disks** of the map workers.

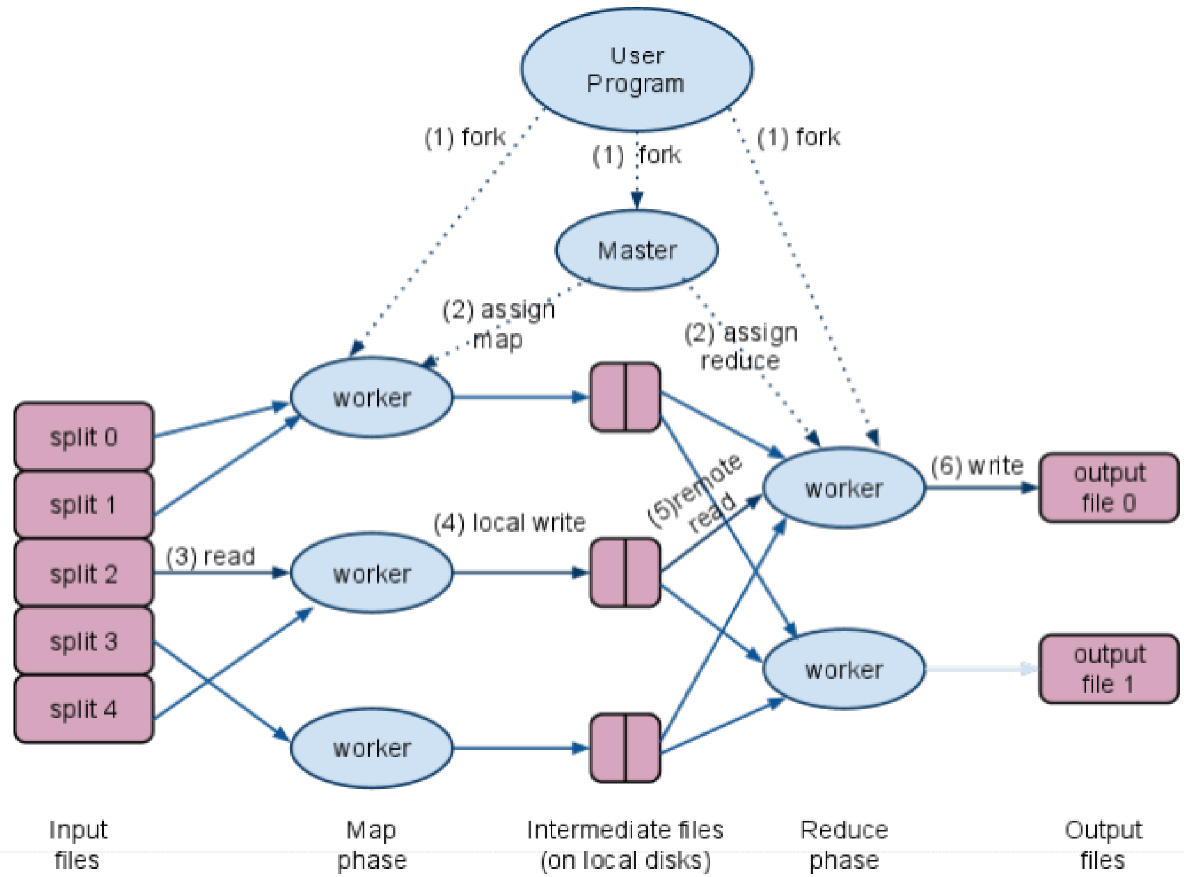
- When a reduce worker has read all intermediate data
  - It **sorts it by the intermediate keys**
  - All occurrences of the **same key are grouped** together.

- The sorting is needed because typically many **different keys map to the same reduce task**.
  - If the amount of intermediate data is too large to fit in memory, an external sort is used.



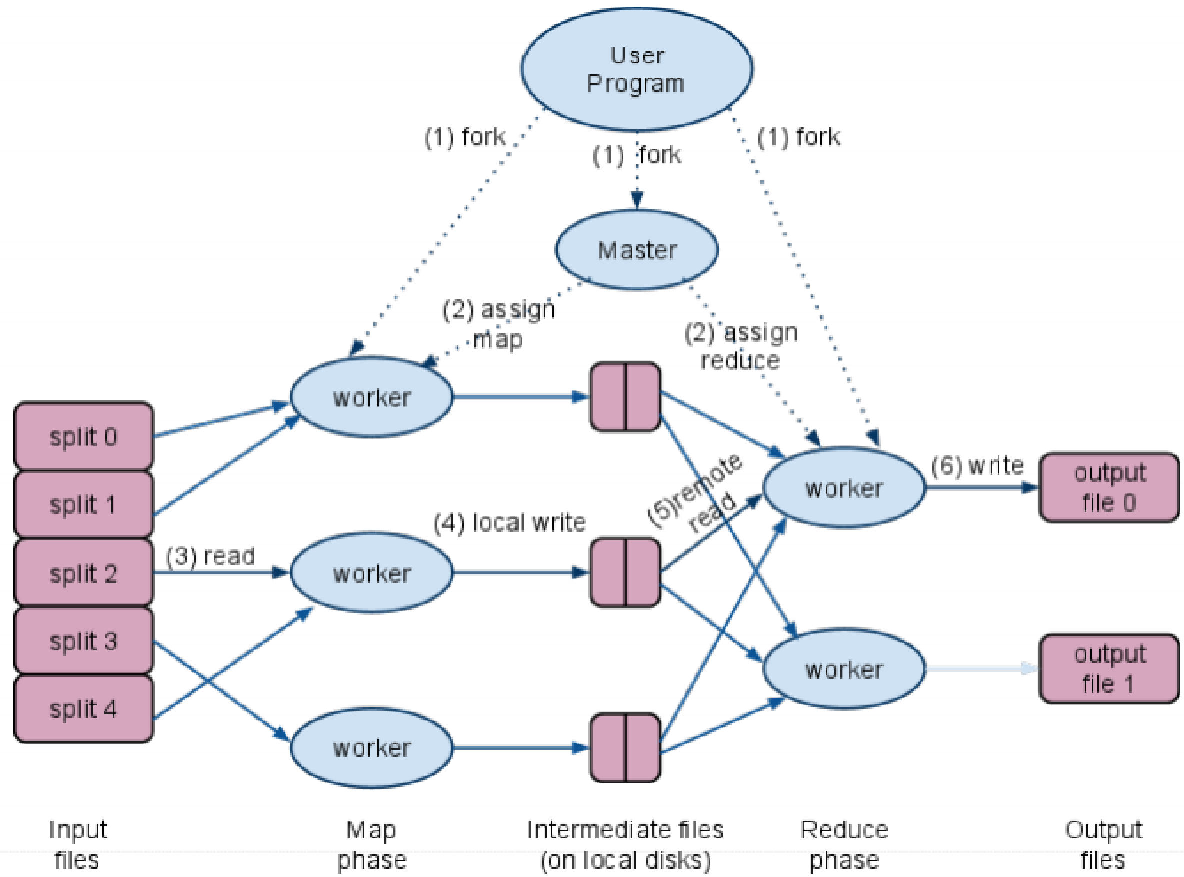
MapReduce Execution Overview – Step 6

- The reduce worker iterates over the sorted intermediate data and, for each unique intermediate key encountered,…
  - It passes the **key and the corresponding set of intermediate values** to the **user's *Reduce* function**.
  - < key, (value1, value2, value3, value4, …) >
- The output of the *Reduce* function is appended to a final **output file** for this reduce partition



MapReduce Execution Overview – Step 7

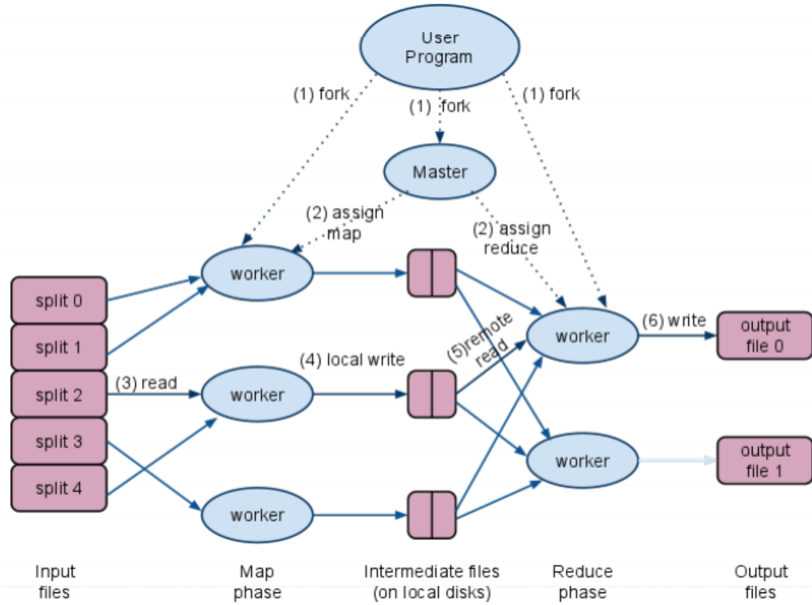
- When all map tasks and reduce tasks have been completed, the **master wakes up the user program**.
  - At this point, the MapReduce call in the user program returns back to the user code.
- The output of the MapReduce execution is available in the **R output files**
  - Typically, **users do not need to combine these R output files into one file**
  - They often pass these files as input to another MapReduce call, or
  - They use them from another distributed application that is able to deal with input that is partitioned into multiple files



Outline

- What is MapReduce?
- MapReduce Execution Overview
- Curiosities
  - Failure Tolerance
  - Task Granularity
  - Combiner Function
- Patterns and applications

Failure Tolerance

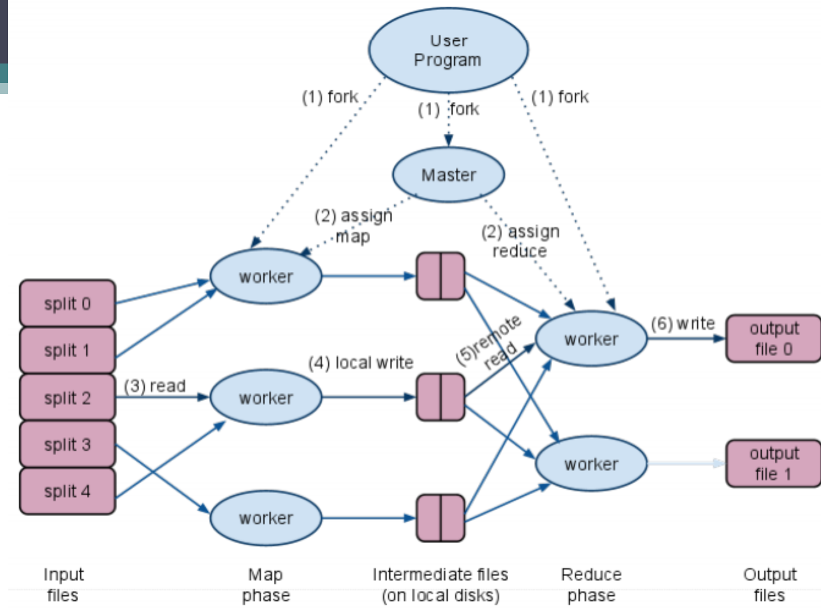


- Master Data Structures
  - **State** (*idle*, *in-progress*, or *completed*) of each **map task or reduce task**
  - **Identity** of the **worker** machine (for non-idle tasks)
  - **Locations** and **sizes** of the **R intermediate file** regions produced by the map task

- Master Failure
  - One approach
    - Master writes periodic **checkpoints** of the **master data structures**
    - If the master task dies, a **new copy can be started** from the last **checkpointed** state.

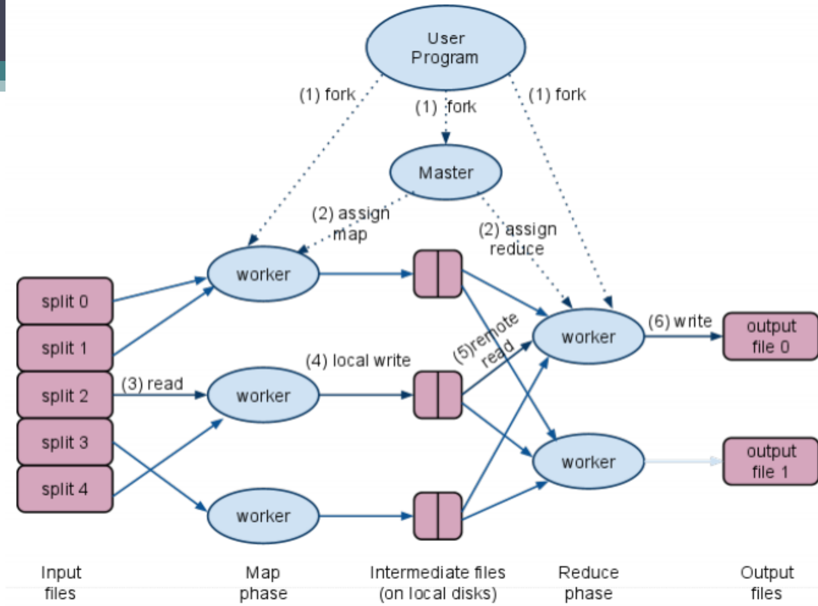
  - Current approach (with one master)
    - To **abort** the MapReduce computation if the master fails.
    - **Clients** can check for this condition and **retry** the MapReduce operation if they desire.

Failure Tolerance



- Worker Failure
  - The master **pings** every **worker** periodically.
  - If **no response** in a certain amount of time, the master marks the **worker as failed**.
  - **Any map task or reduce task <u>in progress</u>** on a failed worker is also **reset to idle** and becomes eligible for **rescheduling**.
  - Any **map tasks <u>completed</u>** by the worker are **reset** back to their initial **idle** state, and therefore become **eligible for scheduling** on other workers.
    - Because the output of map tasks is stored on the local disk(s) of the failed machine and is therefore inaccessible.
    - **<u>Completed</u> reduce tasks** do not need to be re-executed since their output is stored in a global file system
  - When a map task is executed first by worker A and then later executed by worker B (because A has failed), all workers executing reduce tasks are notified of the re-execution. Any reduce task that has not already read the data from worker A will read the data from worker B.

Task Granularity

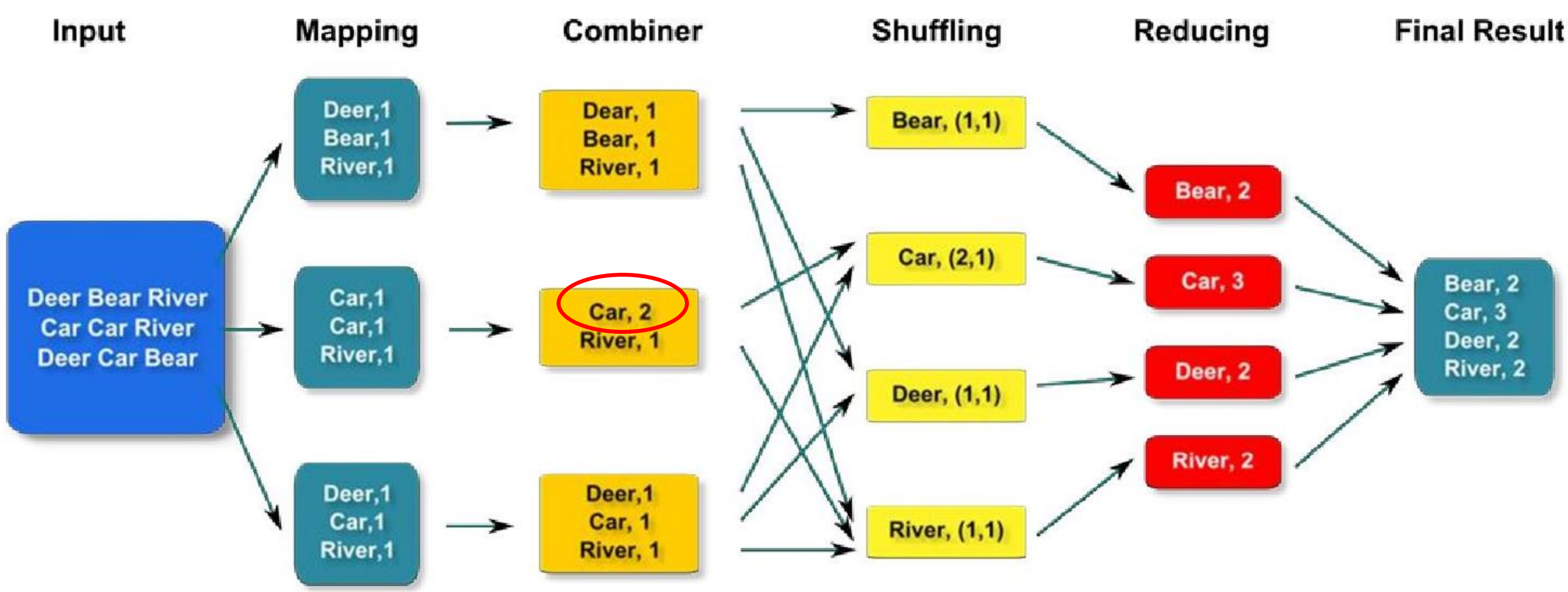


- MapReduce subdivide:
  - map phase into M pieces
  - reduce phase into R pieces
- Master must:
  - Make O(M + R) scheduling decisions
  - Keep O(M ∗ R) state in memory
    - However, it is **small**!
    - Approximately **one byte** of data per “map task / reduce task” pair.
- In practice:
  - Choose M so that each individual task is roughly 16 MB to 64 MB of input data
  - Make R a small multiple of the number of worker machines we expect to use.
  - M = 200.000 and R = 5.000, using 2.000 worker machines

Combiner Function

- In some cases, there is a significant **repetition in the intermediate keys** produced by each map task
  - In word count example: <the, 1>

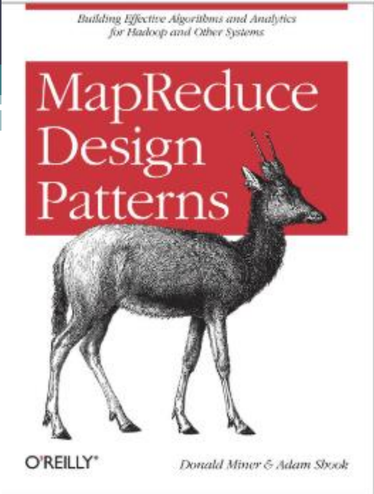
- The **Combiner function** is executed on each machine that performs a map task
  - Typically the **same code** is used to implement both the **combiner** and the **reduce** functions



Outline

- What is MapReduce?
- MapReduce Execution Overview
- Curiosities
  - Failure Tolerance
  - Task Granularity
  - Combiner Function
- Patterns and applications

Patterns and applications



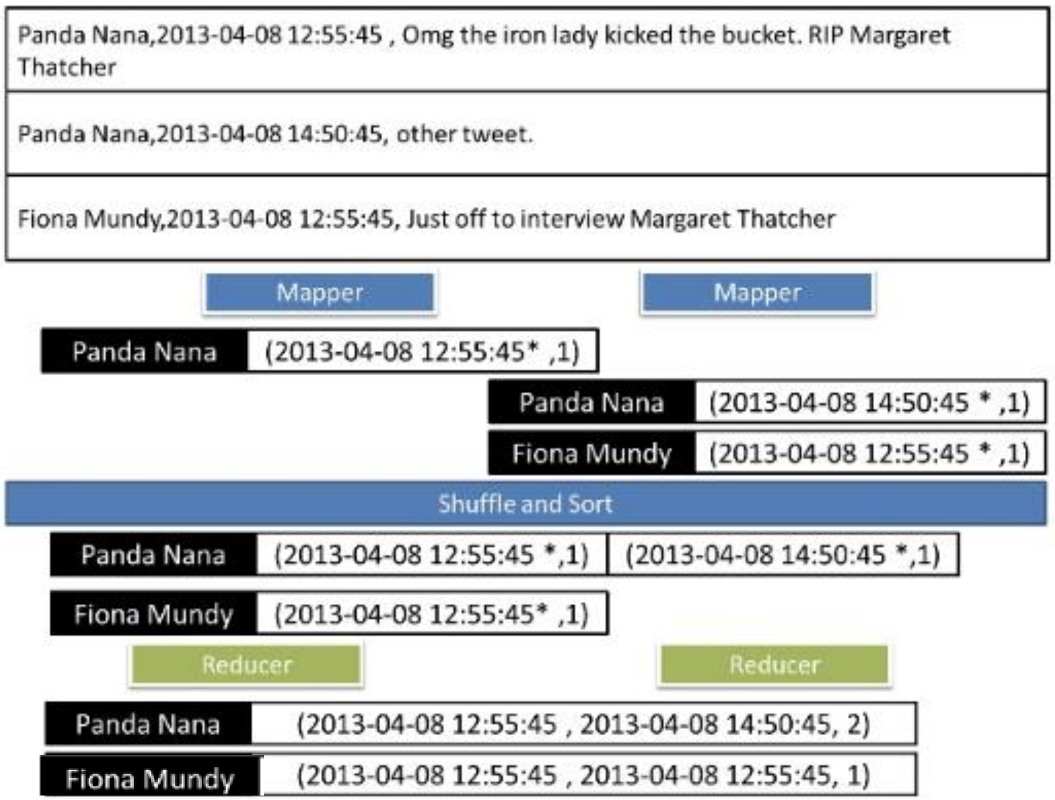
- 23 patterns grouped into six categories
  - Summarization (*)
    - Top-down summaries to get a top-level view
  - Filtering (*)
    - Extract interesting subsets of the data
  - Data Organization
    - Reorganize and restructure data to work with other systems or to make MapReduce analysis easier
  - Joins
    - Bringing and analyze different data sets together to discover interesting relationships
  - Metapatterns
    - Piece together several patterns to solve a complex problem or to perform several analytics in the same job
  - Input and output
    - Custom the way to use Hadoop to input and output data.

Numerical Summarization

- A general pattern for calculating aggregated statistical values over your data
  - To deal with numerical data or counting
  - To group data by specific fields

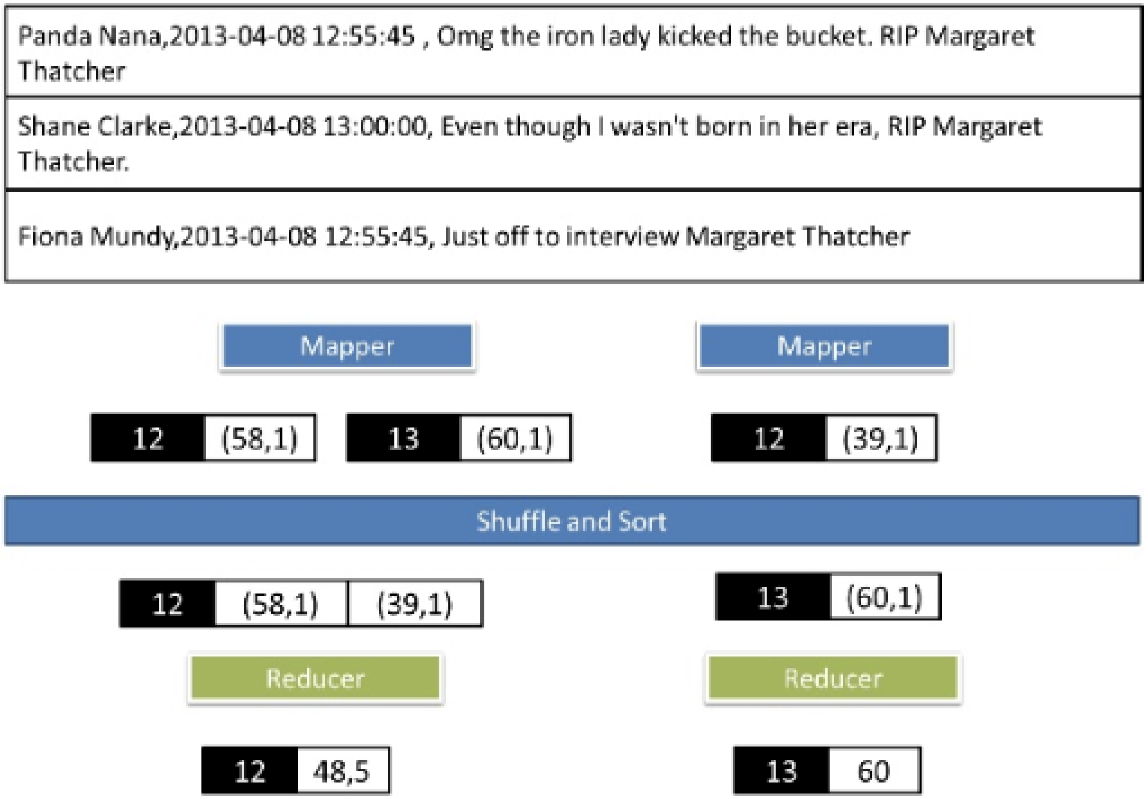
Numerical Summarization

- Word count, Record count
- **Min, max, count** of a particular event
  - E.g. Given a list of tweets (username, date, text), determine first and last time an user commented and the number of times



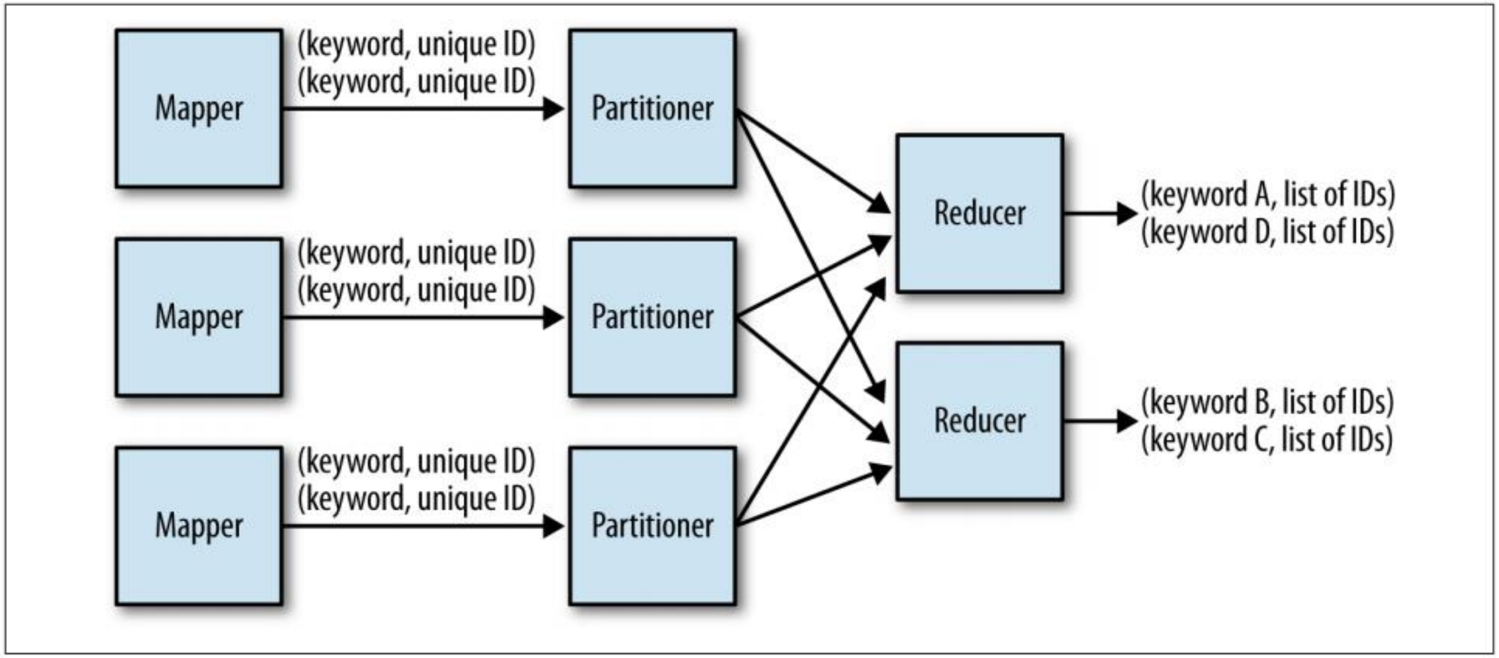
Numerical Summarization

- **Average**, median, standard deviation
  - E.g. Given a list of tweets (username, date, text), determine the average comment length per hour of day



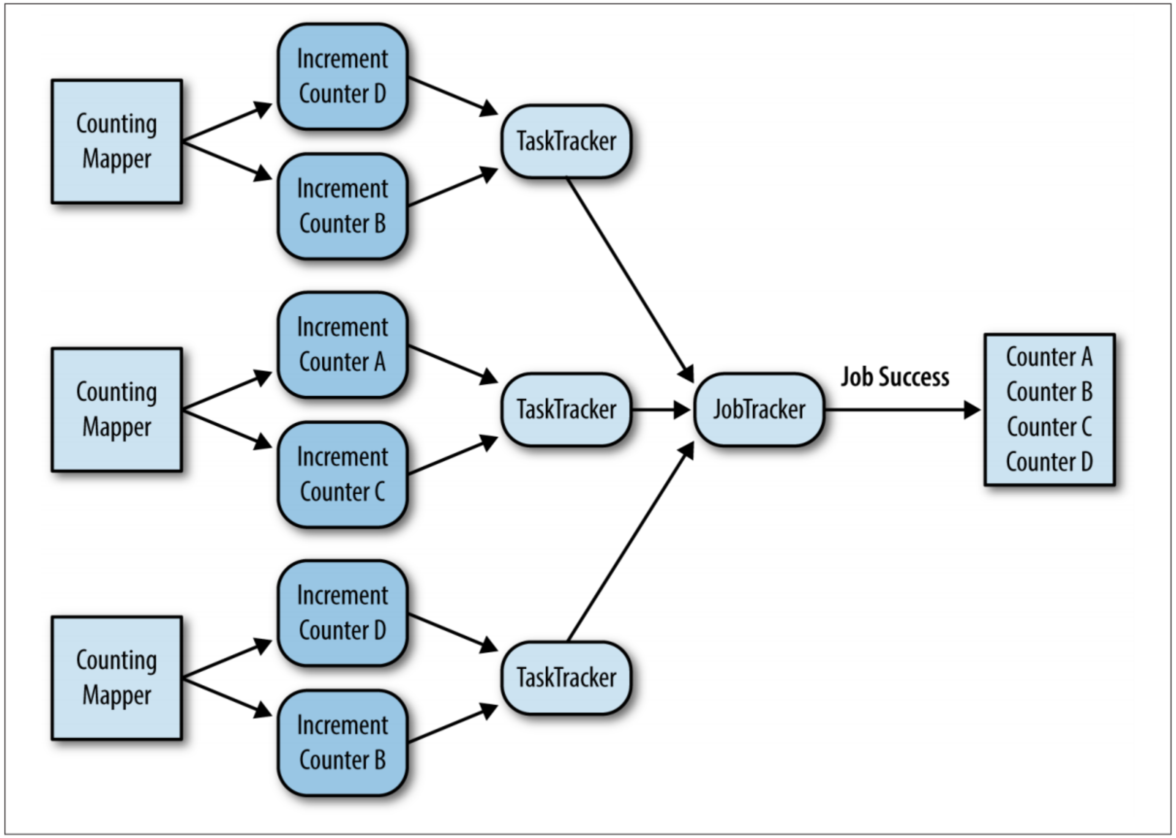
Numerical Summarization

- **Inverted Index**
  - General case: we want to build a map of some term to a list of identifiers
  - E.g. We want to add StackOverflow links to each Wikipedia page that is referenced in a StackOverflow comment.
    - So, Given a set of user's comments, build an inverted index of Wikipedia URLs to a set of answer post IDs



Numerical Summarization

- **Counting with Counters**
  - To calculate a global sum entirely on the map side without producing any output. It is map-only job!
  - E.g. Count the number of users from each state using Hadoop custom counters

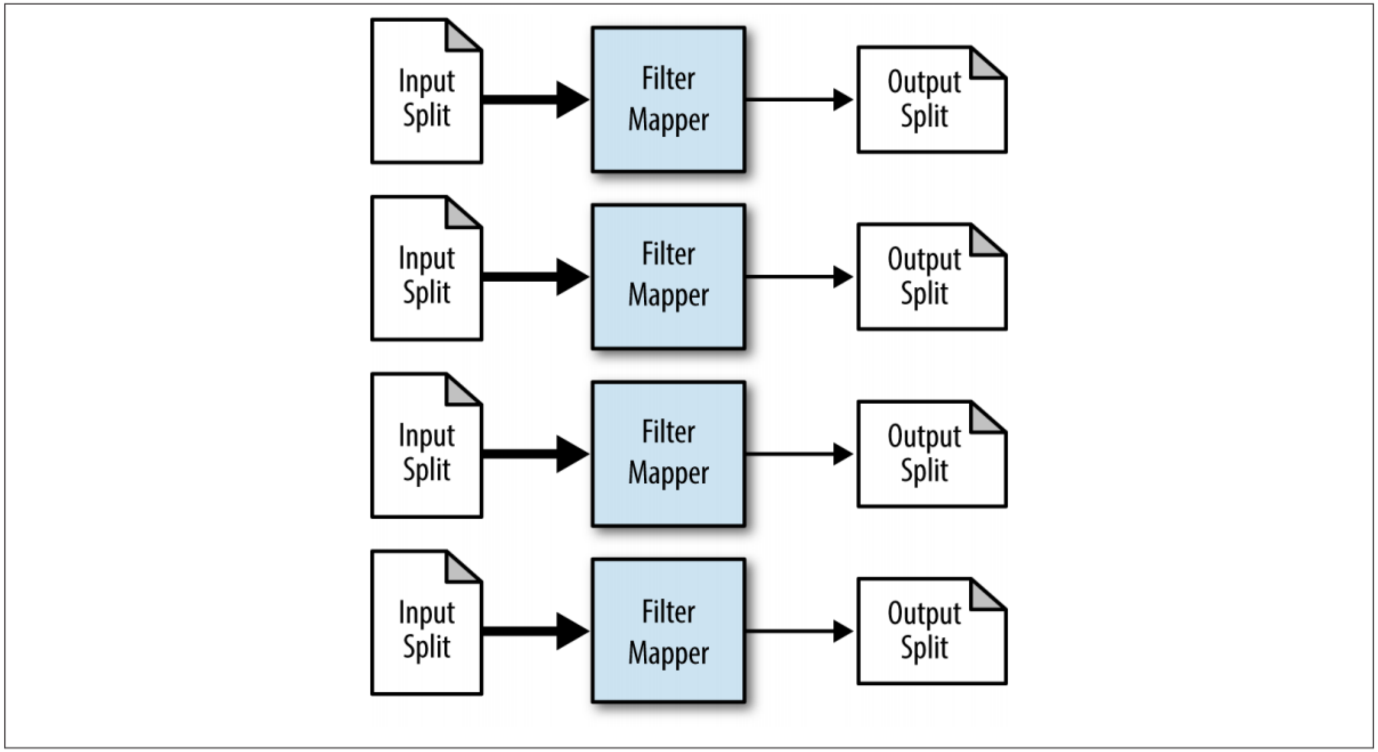


Filtering

- It evaluates each record separately and decides, based on some condition, whether it should stay or go
  - To collate/group data

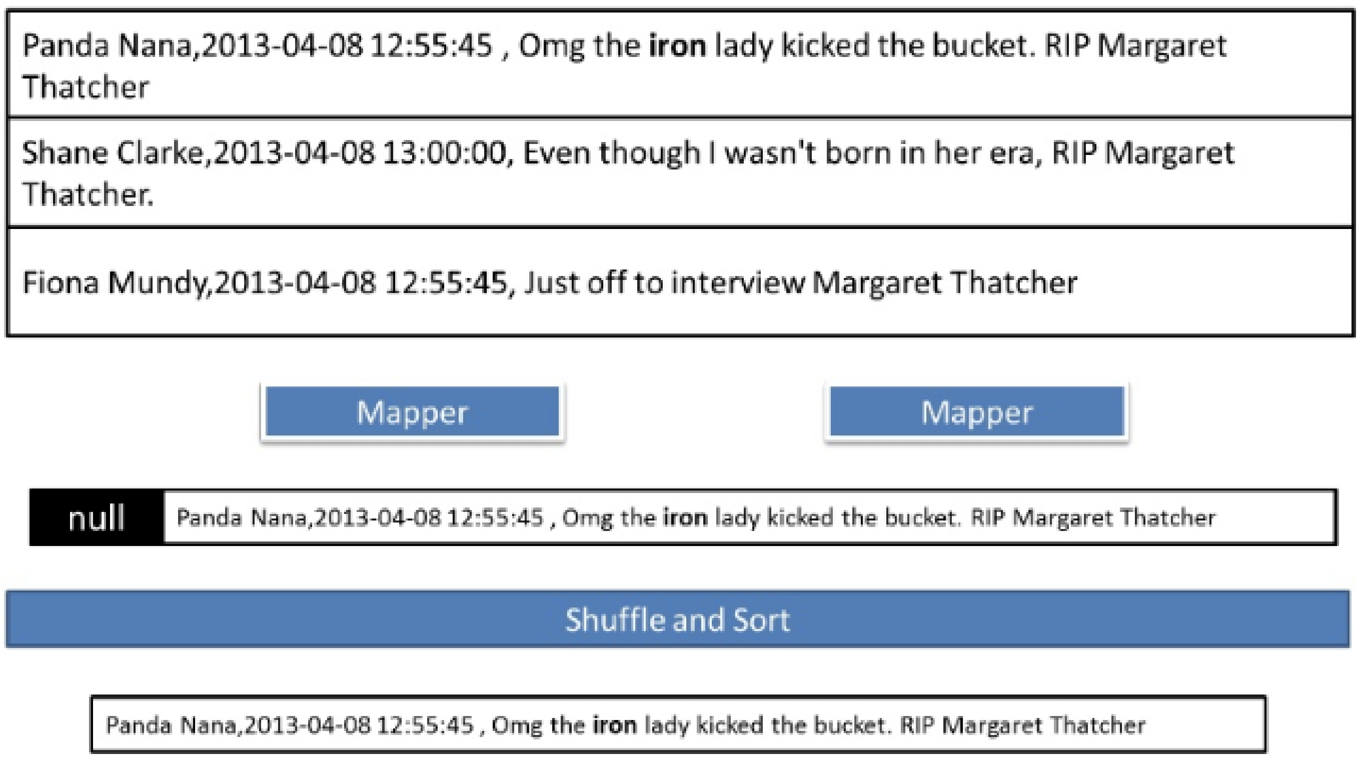
Filtering

- Filtering
  - It simply evaluates each record separately and decides, based on some condition, whether it should stay or go.



Filtering

- Filtering (Distributed Grep)
  - E.g. Given a list of tweets (username, date, text), determine the tweets that contain a *word*

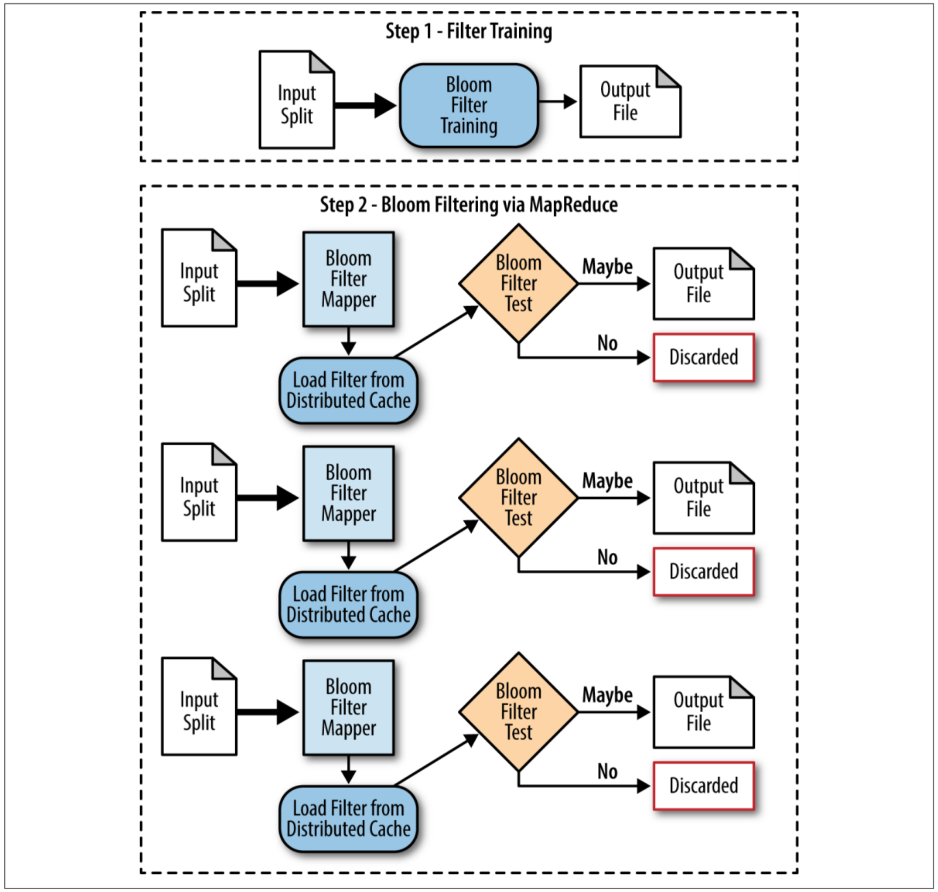


Filtering

- Filtering - Tracking a thread of events
  - E.g. By filtering for that user's IP address, you are able to get a good view of that particular user's activities.
- Filtering - Data cleansing
  - E.g. To validate that each record is well-formed and remove any junk that does occur.
- Filtering - Simple random sampling
  - E.g. If you want a simple random sampling of your data set, you can use filtering where the evaluation function randomly returns true or false.
- Removing low scoring data
  - E.g. If you can score your data with some sort of scalar value, you can filter out records that don't meet a certain threshold.

Filtering

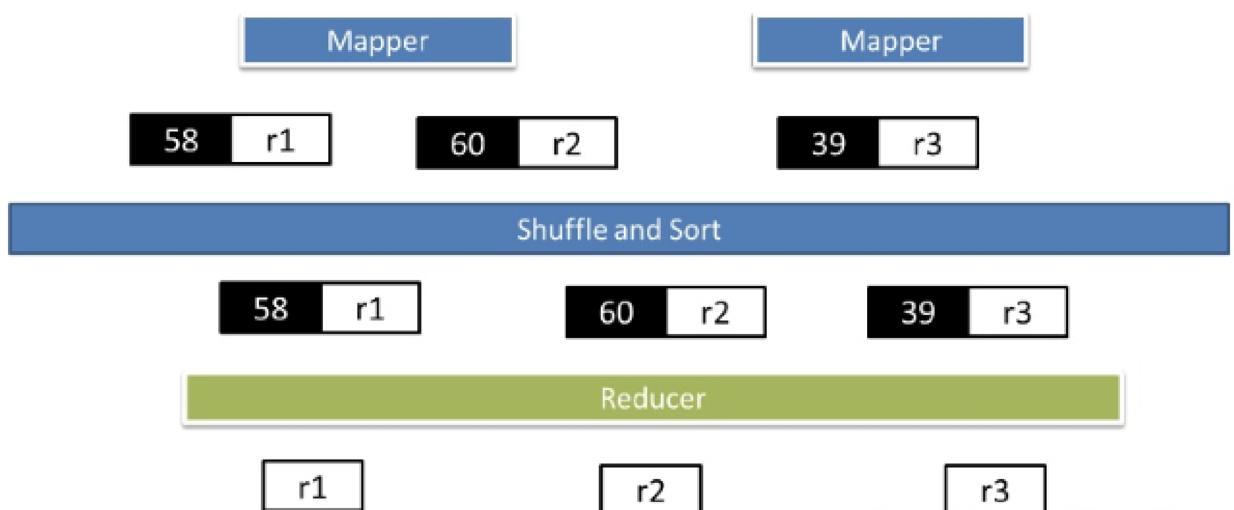
- **Bloom filtering**
  - It does the same thing as filtering pattern, but it has a unique evaluation function applied to each record.
  - E.g. Hot list



Filtering

- Top N
  - Retrieve a relatively small number of top N records, according to a ranking scheme in your data set, no matter how large the data.
  - E.g. Given a list a list of tweets (username, date, text), determine the 5 users that wrote longer tweets

r1	Panda Nana,2013-04-08 12:55:45 , Omg the iron lady kicked the bucket. RIP Margaret Thatcher
r2	Shane Clarke,2013-04-08 13:00:00, Even though I wasn't born in her era, RIP Margaret Thatcher.
r3	Fiona Mundy,2013-04-08 12:55:45, Just off to interview Margaret Thatcher



Filting

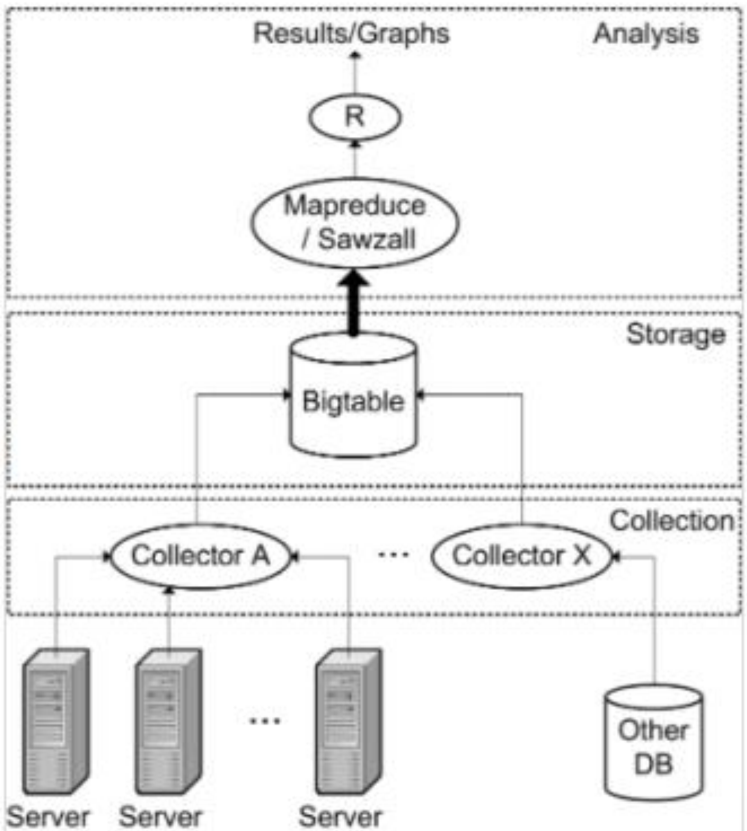
- **Distinct**
  - You have data that contains similar records and you want to find a unique set of values.
  - E.g. Given a list of user's comments, determine the distinct set of user IDs.

```
map(key, record):
    emit record,null

reduce(key, records):
    emit key
```

System Health Monitoring in Google

- Monitoring service talks to every server frequently
- Collect: health signals, activity information, configuration data
- Store time-series data forever
- Parallel analysis of repository data MapReduce
- E.g. DRAM errors observed in a new Gmail cluster



Geographical Data in Google

- Problems that Google Maps has used MapReduce to solve
  - Locating roads connected to a given intersection
  - Rendering of map tiles
  - Finding nearest feature to a given address or location
- Example:
  - Input: List of roads and intersections
  - Map: Creates pairs of connected points (road, intersection) or (road, road)
  - Sort: Sort by key
  - Reduce: Get list of pairs with same key
  - Output: List of all points that connect to a particular road