CLOUD COMPUTING

MapReduce

Lorenzo Leonini
lorenzo.leonini@unine.ch

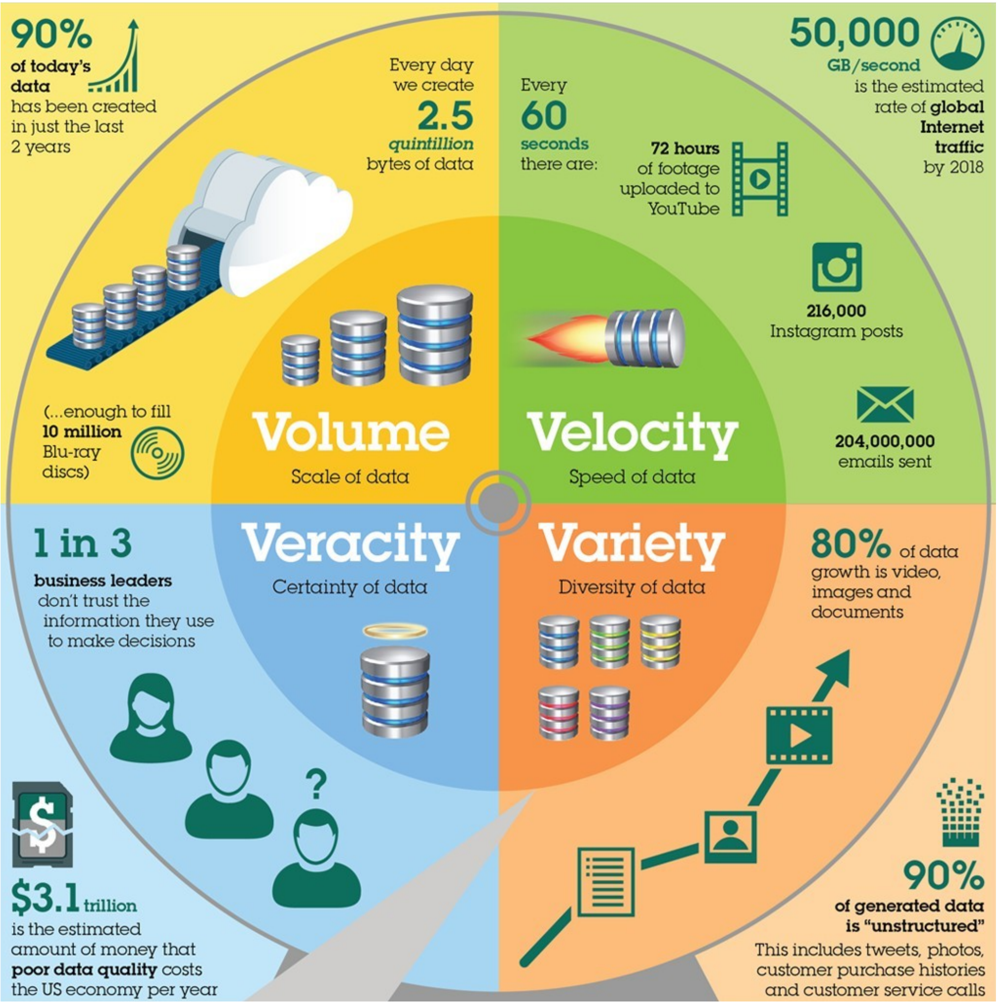
Objectives

- Introduce the need for large-scale data processing in cloud environments
- Present the Map/Reduce principle and supporting frameworks
- Detail some representative applications

Introduction

- Cloud computing allows larger-scale applications
- Large-scale applications attract more users
  - Users generate more data
  - Application generate more data (e.g. logs)
- How can we leverage this data to make the application better?
- "Big Data" phenomenon

The 4 "V" of Big Data



https://www.ibmbigdatahub.com/infographic/extracting-business-value-4-vs-big-data

Data-supported services

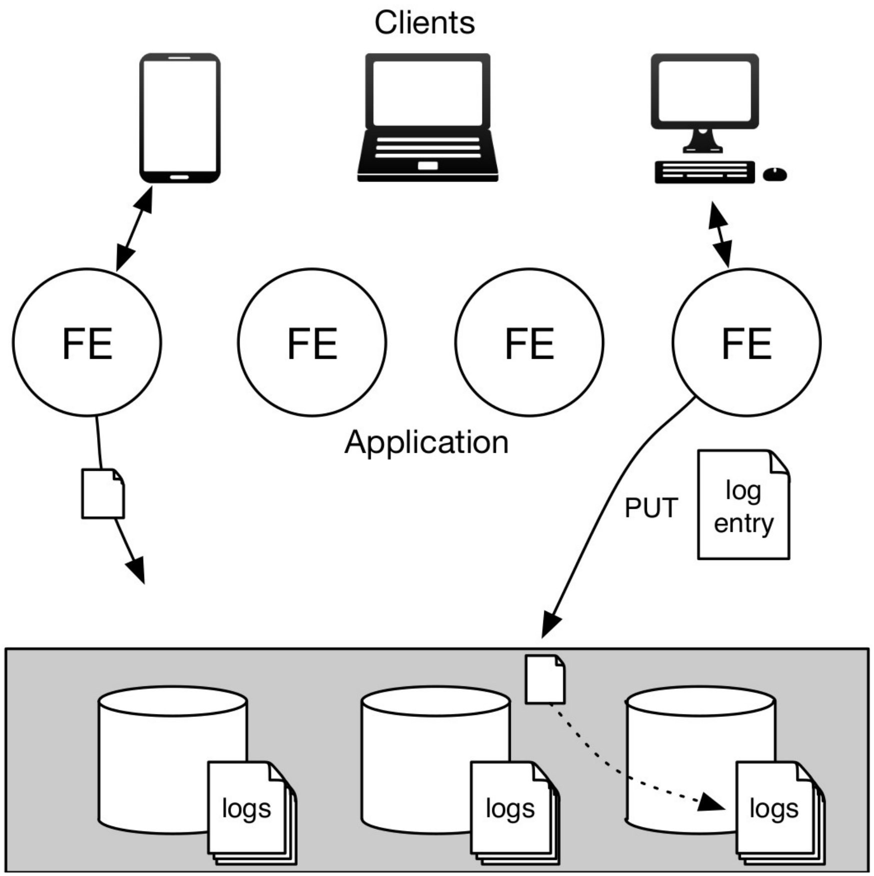
- Applications using large volumes of public data
  - Web search and indexing (initially)
- Applications using user-generated data
  - Social networks
  - Recommendation engines (Netflix, Spotify, etc.)
  - Taxi hailing predictive models (where will clients be? Where will they want to go?)
- And using both
  - Web search and indexing (now)
  - Cambridge Analytica...

Dealing with the two first “V”s

- Need specific tools and programming models
  - (Very) large amounts of data
  - Complex environment (replication, distribution)
  - Faults and slow machines
- How to handle **V**olume & **V**elocity
  - Map/Reduce framework for large static data
  - Stream processing frameworks for dynamic data
- We will not cover Variety (data curation) and Veracity (data authenticity)

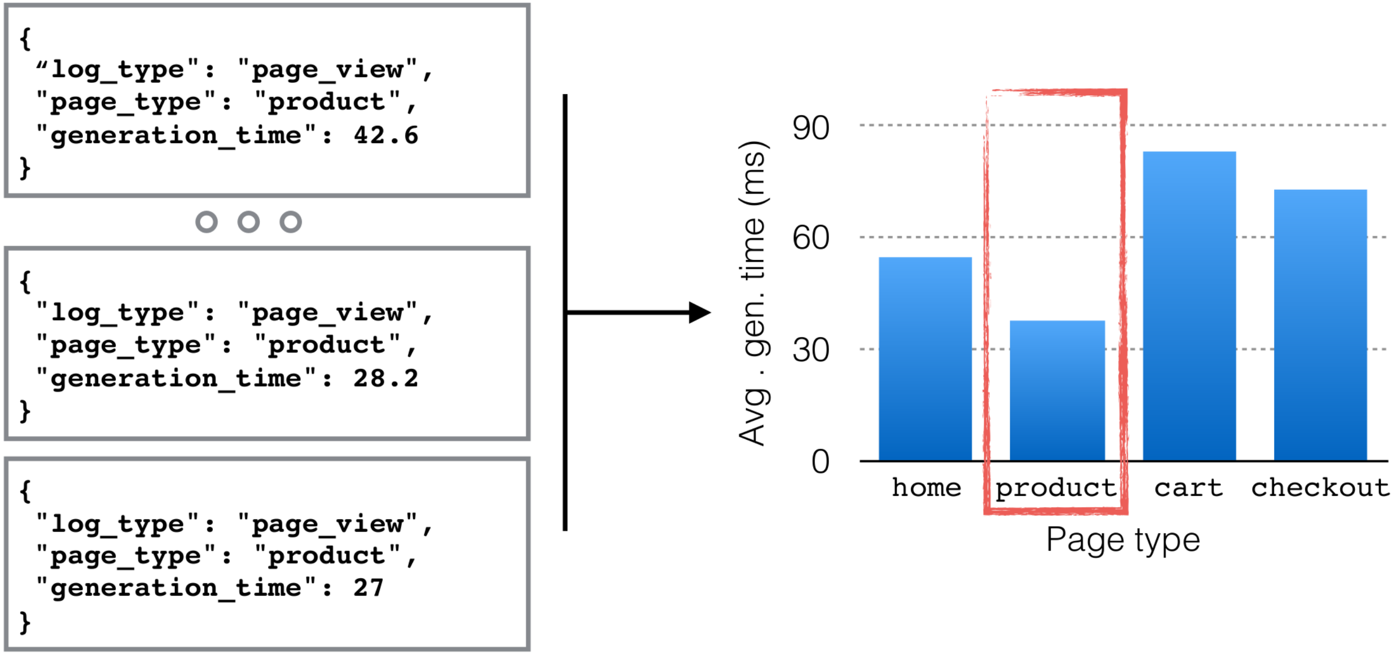
Motivating example: Logging

- Application front-end components generate logs
  - Client information
  - Errors
  - Page generation time
  - Items accessed/purchased
- Log entries stored as JSON documents in a distributed database



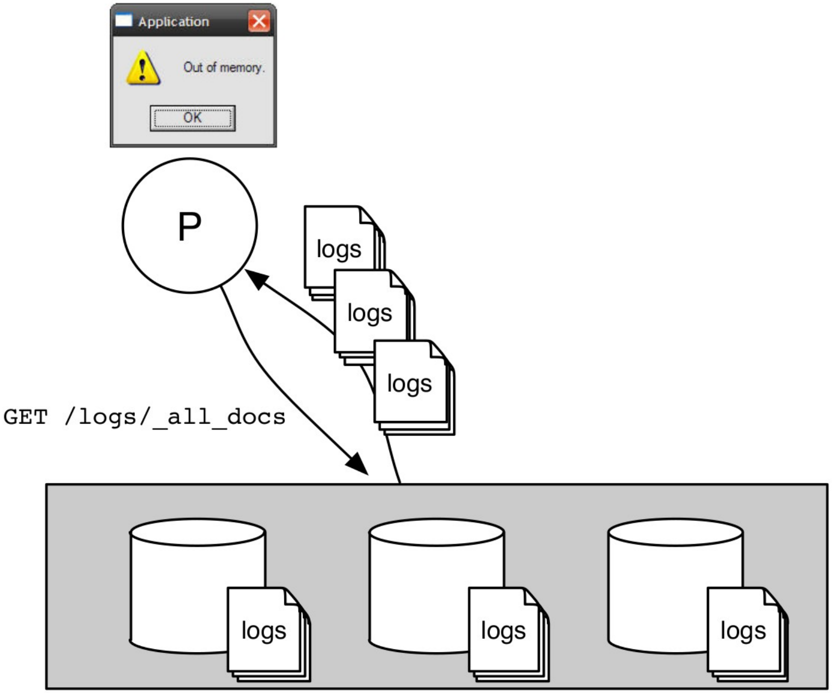
Processing logs

- We would like to know the *average generation time* for each type of page
  - **Types**: home, product, cart, checkout
- This information is available by *processing the logs*

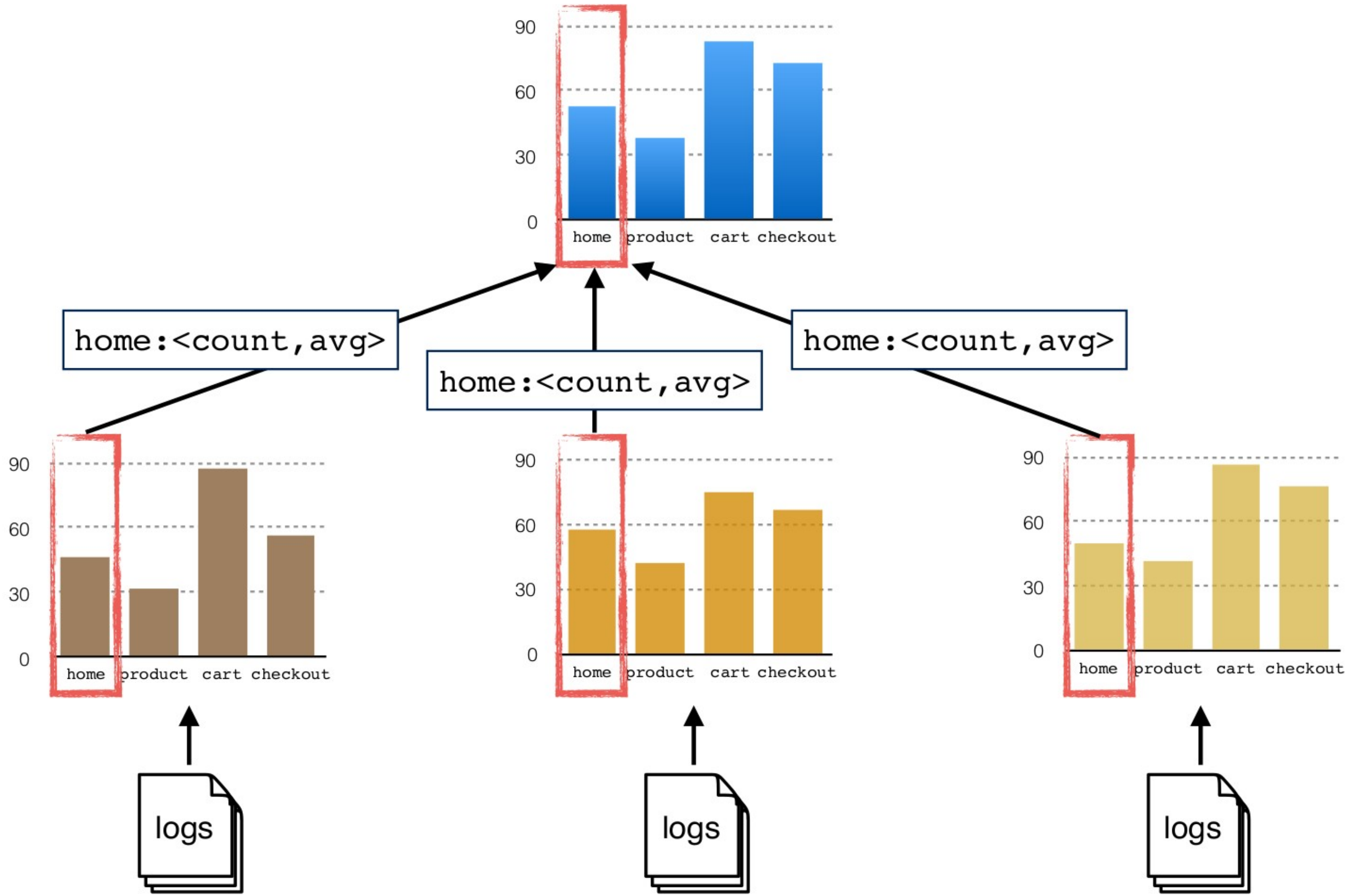


Centralized processing?

- Should we collect all log documents and process them in a *dedicated single process*?
  - Amount of logs can be very large
    - May not fit in memory at all!
  - Not all logs entries are useful for our operation: wasted bandwidth
  - Very slow
- Need *parallel processing*



Processing logs in parallel



Handling Volume

- Split the dataset in multiple partitions, process each partition independently
  - Later, merge outputs for final result
- Difficult to do "manually"
  - Deploy worker processes close to the data
  - Coordinate all the workers
  - Handle faults at the workers
  - Collect all outputs and process them
  - Start again ...

Map/Reduce

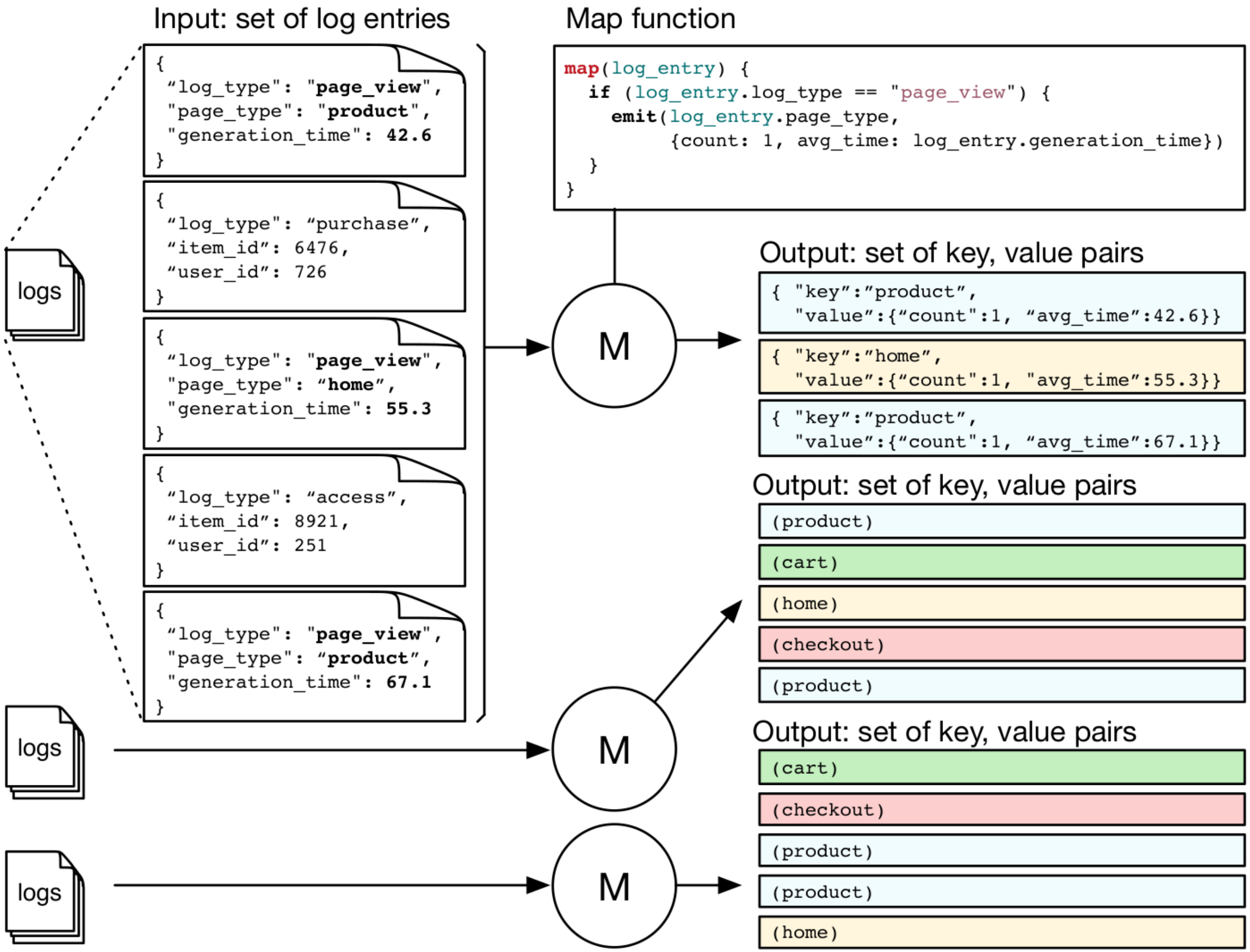
- Big Data processing often follows a similar pattern:
  - PARTITION - Iterate in parallel over a large number of records
  - MAP - Extract information of interest from the records
  - SHUFFLE - Regroup this information in sets, one for each category
  - REDUCE - Aggregate each of the sets to obtain the final result(s)
- Map/Reduce is a programming model that allows expressing such queries and running them in parallel on many machines
- *Key idea:* provide an abstraction at the point of the two operations MAP and REDUCE, make others implicit

Programming model

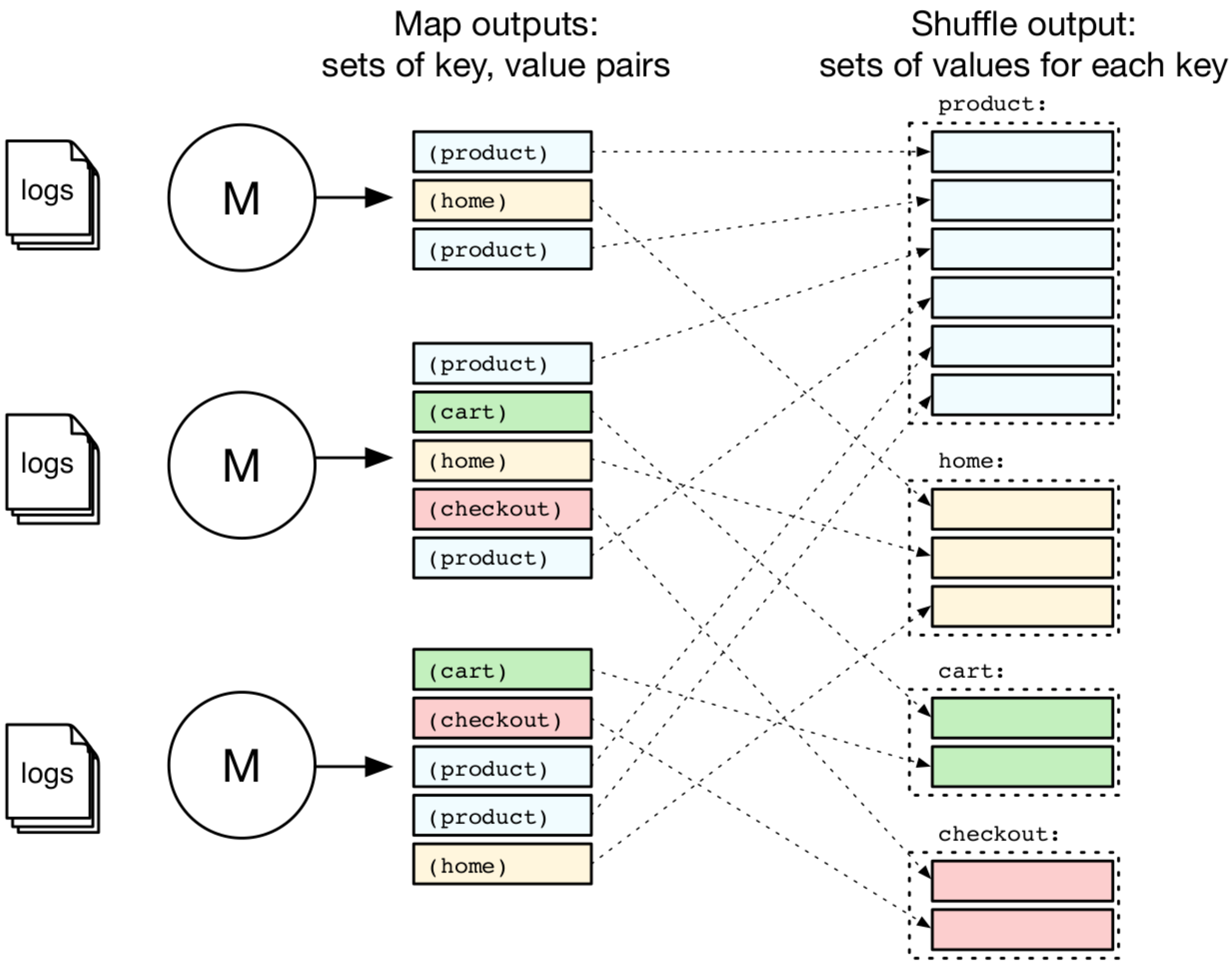
- Programmer specifies two functions
  - `map(record)`
    - get records from a partition of the source data
    - can generate `<key, value>` pairs with the emit call
  - `reduce(key, [values])`
    - receives all values for the same key
    - generate aggregate results, also as `<key, value>` pairs

- PARTITION and SHUFFLE are handled transparently

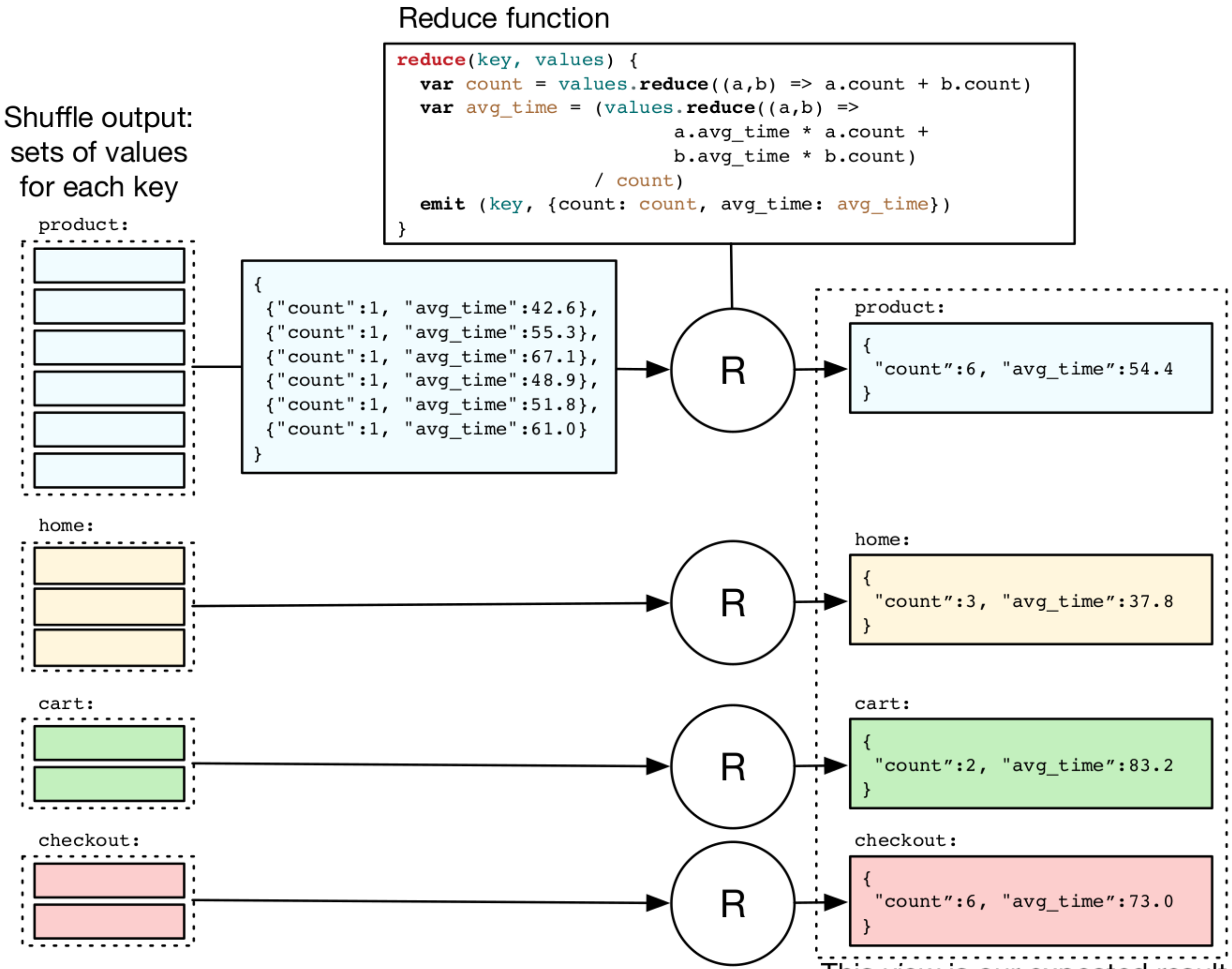
MAP phase


Input: set of log entries
Map function
logs
{ "log_type": "page_view", "page_type": "product", "generation_time": 42.6 }
{ "log_type": "purchase", "item_id": 6476, "user_id": 726 }
{ "log_type": "page_view", "page_type": "home", "generation_time": 55.3 }
{ "log_type": "access", "item_id": 8921, "user_id": 251 }
{ "log_type": "page_view", "page_type": "product", "generation_time": 67.1 }
map(log_entry) {
if (log_entry.log_type == "page_view") {
emit(log_entry.page_type,
{count: 1, avg_time: log_entry.generation_time})
}
}
M
Output: set of key, value pairs
{ "key":"product", "value":{"count":1, "avg_time":42.6}}
{ "key":"home", "value":{"count":1, "avg_time":55.3}}
{ "key":"product", "value":{"count":1, "avg_time":67.1}}
Output: set of key, value pairs
(product)
(cart)
(home)
(checkout)
(product)
Output: set of key, value pairs
(cart)
(checkout)
(product)
(product)
(home)

SHUFFLE phase

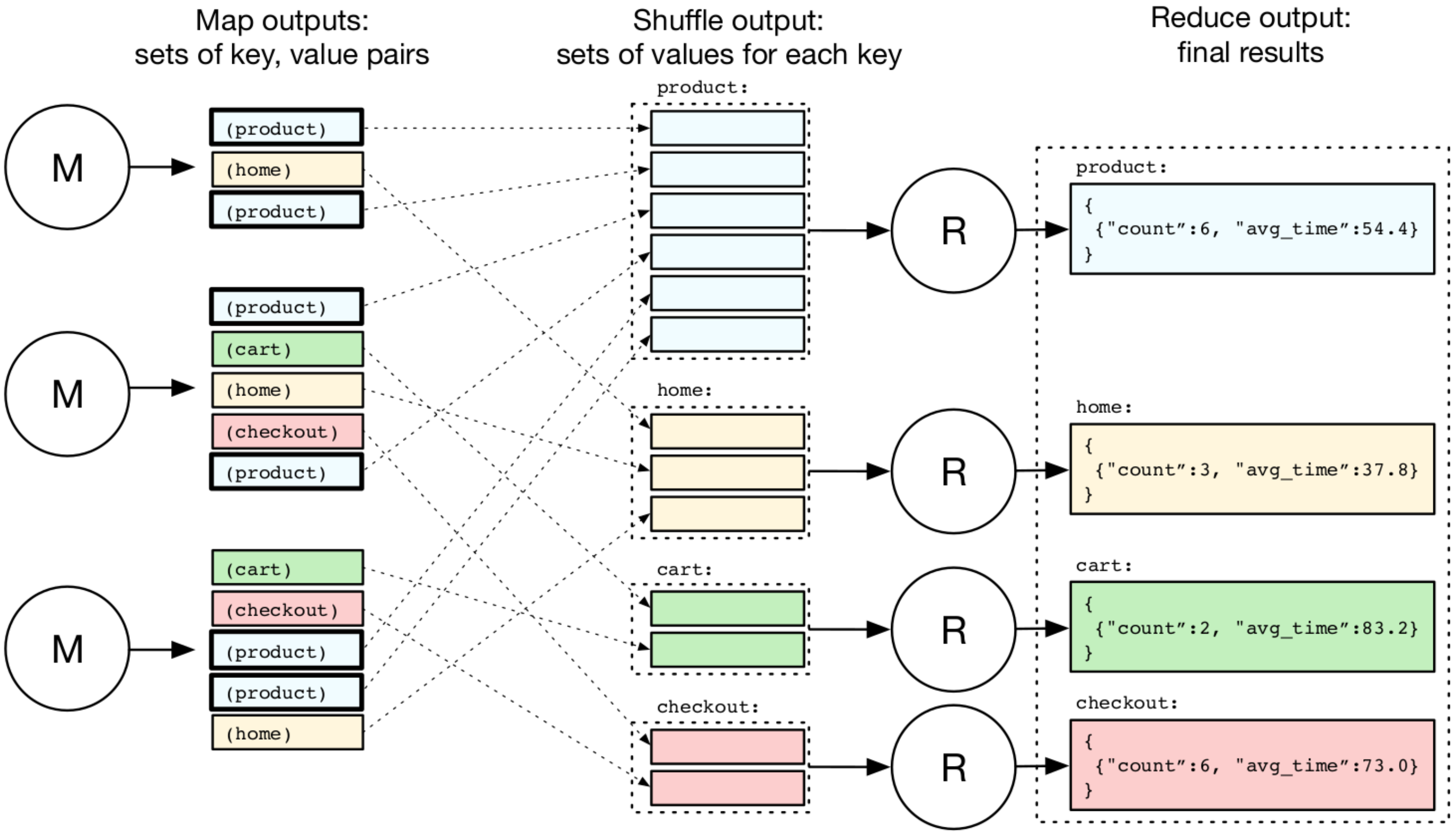


REDUCE phase



This *view* is our expected result

Bandwith inefficiency



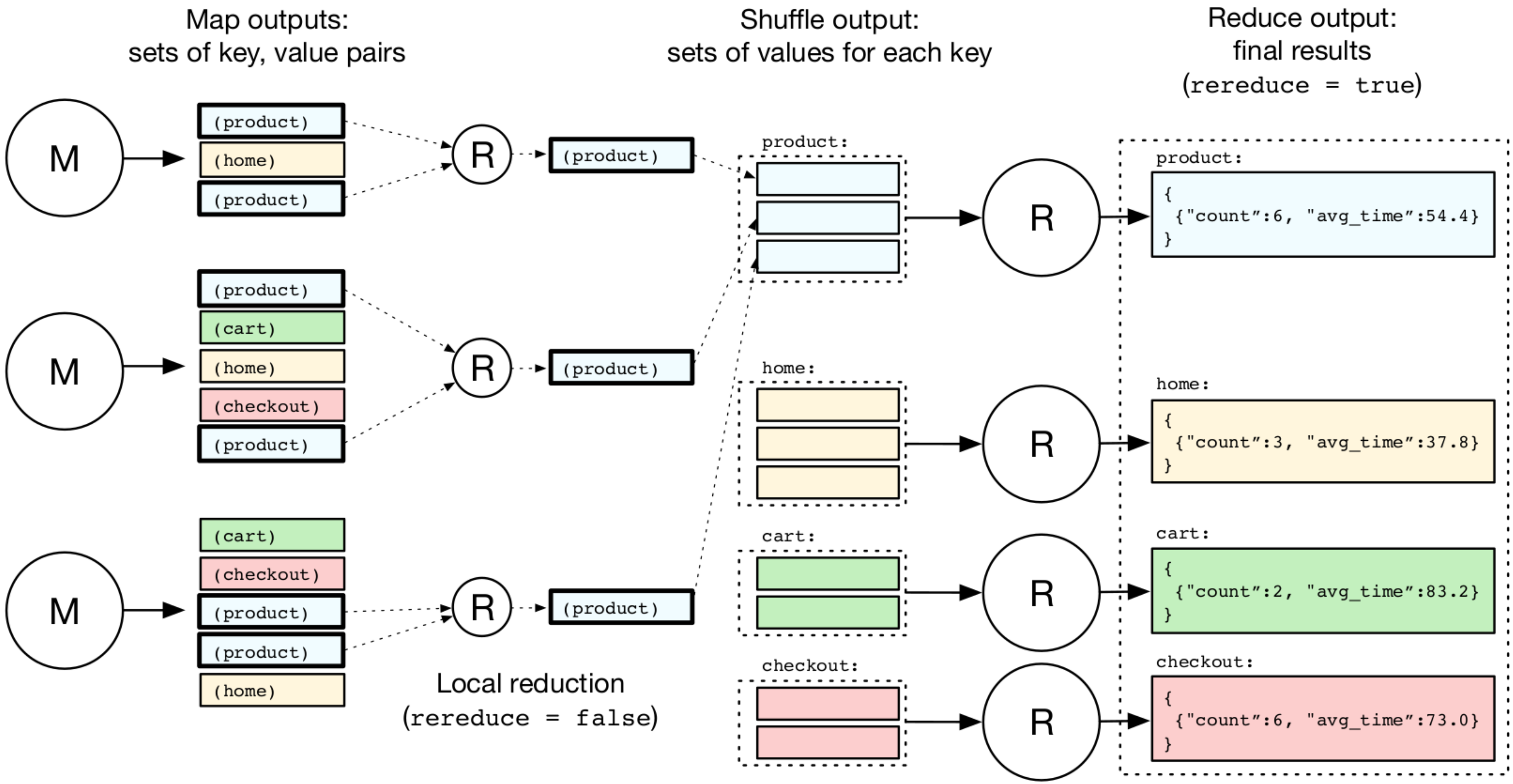
- Several `<key, value>` pairs for the same key are generate by each mapper and shuffled independently to the corresponding reducer!

Local reduction

- Could we locally aggregate all `<key, value>` pairs for the same key at the Mapper process, before sending them to the SHUFFLE phase?
- The output of reduce() functions can be used in its input list
  - Local MAP output can be locally reduced before the SHUFFLE phase
  - This property holds for many aggregations: max, min, average,...
- Sometimes, we need to know if we are reducing local MAP results or the result of the SHUFFLE phase
  - Optional "rereduce" parameter to the reduce() call tells us if this is a local reduction or the final one:

    `reduce(key, [values], **rereduce**)`

Local reduction applied



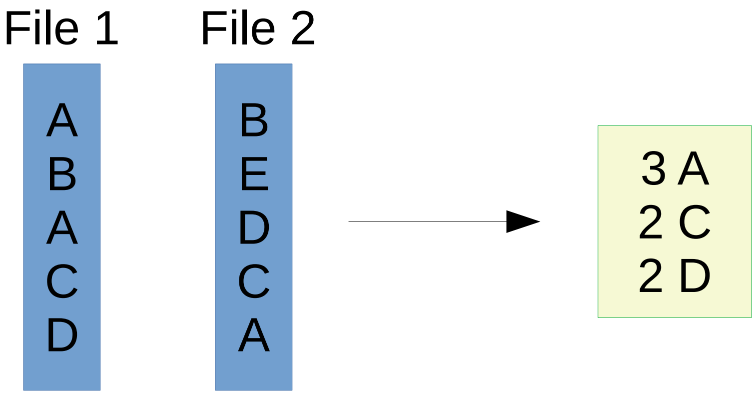
Link with functional programming

- Higher-order functions take a function as argument
- Map: function applied to every element in a list
  - Result is a new list
  - Each operation is independent
- Fold: accumulator set to initial value, function applied to list element and the accumulator, result stored in accumulator
  - Repeated for every item in the list
  - Result is the final value in the accumulator
- Map/Reduce similar in principle but multiple output lists for Map and parallel Fold operations

Examples of application

Distributed grep

- Count lines in all files that match <regex>, show in descending order
  - `grep -Eh <regex> dir/* | sort | uniq -c | sort -nr`
- Example
  - `grep -Eh 'A|C|D' File* | sort | uniq -c | sort -nr`



Distributed grep

- Map function
  - Input: files <line>
  - Emit: if line matches pattern <line, 1>
- Reduce function
  - Input: <line, [1,1, …]>
  - Emit: <line, n> where n is the total of 1s

Distributed grep

File 1

A
B
A
C
D

File 2

B
E
D
C
A

```
MAP
(A) → (A, 1)
(B) →
(A) → (A, 1)
(C) → (C, 1)
(D) → (D, 1)
(B) →
(E) →
(D) → (D, 1)
(C) → (C, 1)
(A) → (A, 1)
```

```
REDUCE
(A, [1, 1, 1]) → (A, 3)
(C, [1, 1]) → (C, 2)
(D, [1, 1]) → (D, 2)
```

3 A
2 C
2 D

Wordcount

- Count how many times each word appears in a text corpus

```
map(text) {
  foreach (word: text.split(' ')) {
    emit(word, {count: 1})
  }
}
```

```
reduce(key, values) {
  var count = values.reduce((a,b) => a.count + b.count)
  emit (key, {count: count})
}
```

Local reducers

- Local reducers helps save bandwidth by pre-reducing the results of a map locally and send fewer `<key, value>` intermediate pairs
- In the previous examples, the same code can be used as local reducer
  - Aggregate `<regex, 1>` counts for lines matching the pattern(s) under a single `<regex, n>` pair
  - Aggregate `<word,1>` counts for different words under a single `<word, n>` pair
  - Not always the case!

Top-k page frequency

- Input: log of web page requests
- Output: the top-*k* accessed webpages
- Map function
  - Parse log, output `<URL, 1>` pairs
  - Similar to word count but goes to a **single key**
- Reduce function (for single key)
  - Aggregate `<URL, 1>` pairs for individual URLs
  - Sort by decreasing frequency
  - Output top-*k* `<URL, n>` pairs

Top-k page frequency

- Going to a single key is vastly inefficient
  - A single worker will receive all individual `<URL,1>` pairs
  - Only to aggregate them and get the top-k URLs
- Use local reducer?
  - Here we cannot use the same code as in previous examples
    - It includes the selection of the top-k URLs
    - The union of the local top-k is not the global top-k
    - Sometimes OK to do local trimming for approximate solution

Reverse Web-link graph

- Get all the links pointing to some page
  - This is the basis for the PageRank algorithm!
- Map function
  - Input: web pages (fetched by a *web crawler*)
  - Output a `<target, source>` pair for each link to *target* URL in a page named source
- Reduce function
  - Concatenate the list of all source URLs associated with a given target URL and emits the pair:

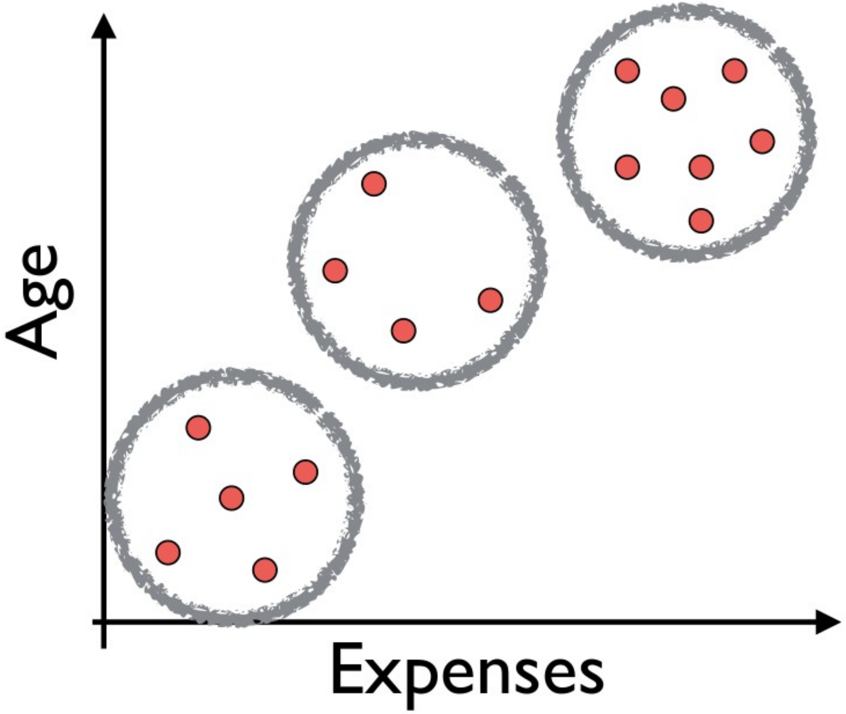
    `<target, list(sources)>`

Inverted index

- Get all documents containing some particular keyword
  - Used by the search mechanisms of Google, Yahoo!, etc.
  - Second input for PageRank
- Map function
  - Input: (e.g.) web pages (fetched by a *web crawler*)
  - Parse each document and emit a set of pairs `<word, documentID>`
- Reduce function
  - Take all pairs for a given word
  - Sort the documents IDs
  - Emit a final `<word, list(document IDs)>` pair

K-Means clustering

- Typical data mining / database problem
- Group items into k clusters
  - Clusters must minimize distance of contained points (relative to the center)
- NP hard problem
  - Heuristic algorithm



K-Means algorithm

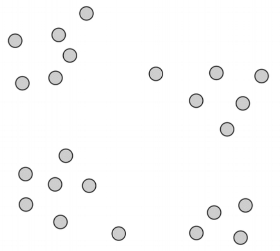
- Goal: obtain m_1, m_2, …, m_k as representative points of the k clusters
  - These will be the *centroids* of the clusters
- Initialize m_1, m_2, …, m_k to random values
- Iterate (t = iteration)
  - Assign each point to the clostest centroid

$$S_i^{(t)} = \left\{ \mathbf{x}_j : \left\| \mathbf{x}_j - \mathbf{m}_i^{(t)} \right\| \le \left\| \mathbf{x}_j - \mathbf{m}_{i^*}^{(t)} \right\| \, \forall \, i^* = 1, \ldots, k \right\}$$

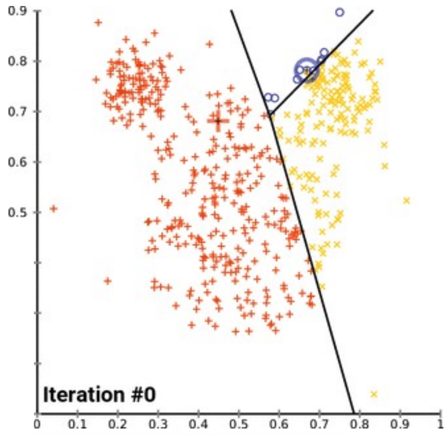
  - m_i becomes the new centroid for its points

$$m_i^{(t+1)} = \frac{1}{\left| S_i^{(t)} \right|} \sum_{x_j \in S_i^{(t)}} x_j$$

K-Means iterations



0a. Données d'entrée



Source: wikipedia

K-Means in Map/Reduce

- Initialize
  - Choose centroids randomly
- Classify
  - Map eachpoint to the closest centroid

$$S_i^{(t)} = \left\{ \mathbf{x}_j : \left\| \mathbf{x}_j - \mathbf{m}_i^{(t)} \right\| \leq \left\| \mathbf{x}_j - \mathbf{m}_{i^*}^{(t)} \right\| \forall\, i^* = 1, \ldots, k \right\}$$

- Recenter
  - m_i becomes new centroid for its points

$$m_i^{(t+1)} = \frac{1}{\left|S_i^{(t)}\right|} \sum_{x_j \in S_i^{(t)}} x_j$$

- Repeat until no change
  - Centroids have converged

Classification step as Map

- Initialize
  - Set `global var centroids` from file
    - Initially *k* random points
- `map(point)`
  - Access to `global var centroids`
    - Can also be provided as data directly inside the map function definition to keep the functionnal paradigm
  - Compute nearest centroid based on centroids:
    - `emit(nearest centroid, point)`

Recenter step as Reduce

- Initialize
  - Set `global var centroids = []`
- `reduce(centroid`$_t$`, [points])`
  - Recompute centroid from points in it
    - `emit(centroid`$_{t+1}$`)`
- Cleanup (after all calls to reduce are made):
  - Save new centroids to file
  - Check if change since last iteration (if yes → re-iterate)
- Does not completely "fit" Map/Reduce model that assume no global shared state
  - But widely used in practice

Origin and implementation

Origin

- Map/Reduce originally proposed by Google in 2004
  - Necessary due to their unprecedented scale
  - One (if not the) most cited paper in computer science
  - Computation of the PageRank for webpages
    - Also, extraction of keywords, sanitation, etc.
    - Many other uses: log parsing, network monitoring, etc.
- Proprietary implementation
- Huge and immediate success
  - Simple programming model
  - Distributed computation complexity left to framework
  - Usable by non-CS majors

Original Map/Reduce

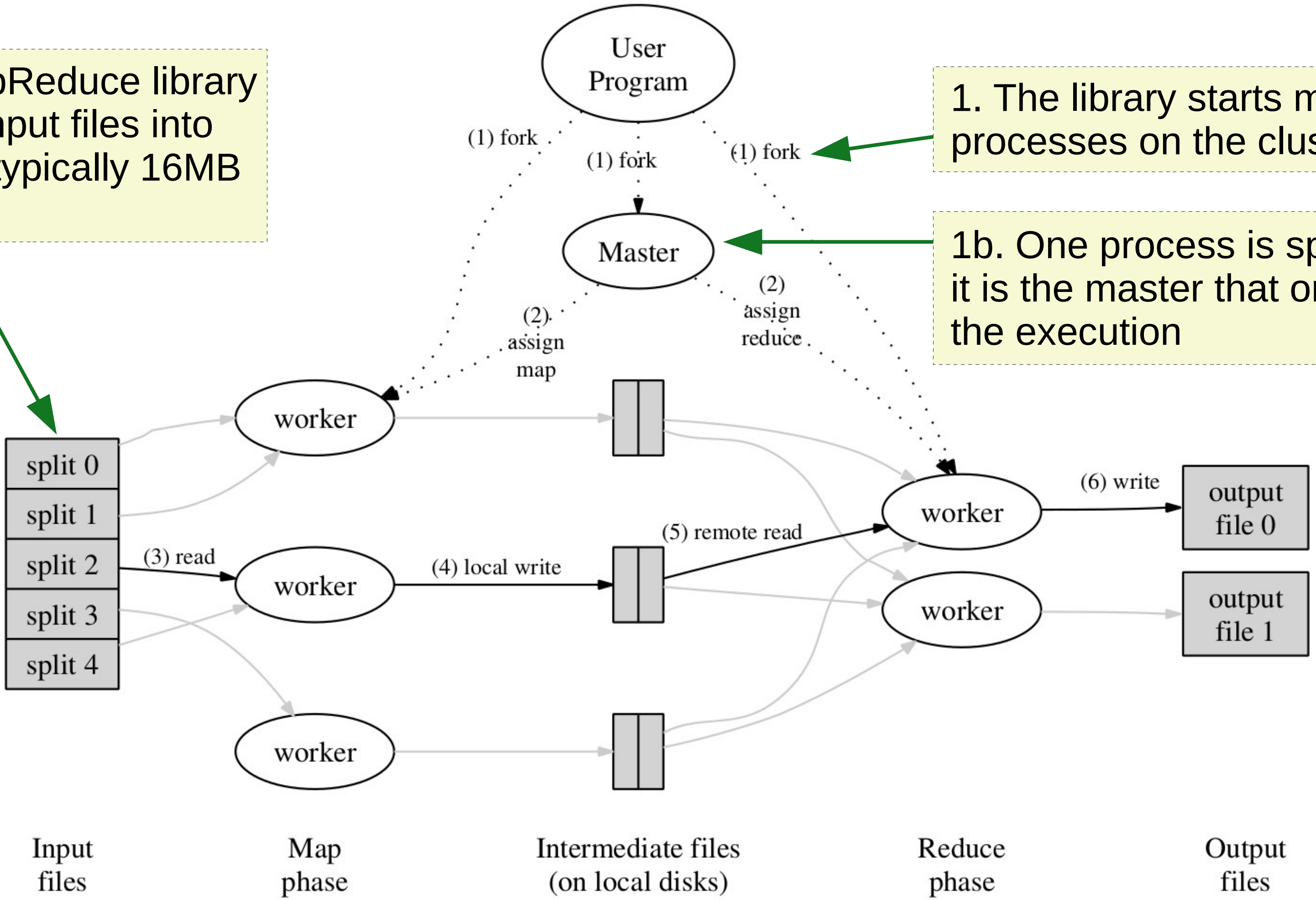
- The job is submitted to a master process
- The master orchestrates its execution
- Each node supports one or more workers
- Each worker can handle a map or reduce job when instructed by the master
- Map jobs get the data from file system
  - Google File System: optimized for storing large append-only files
  - Also possible from NoSQL database: Google BigTable
- Communication is based on key/values pairs
- Output written to file system

Execution overview

0. The MapReduce library splits the input files into M pieces (typically 16MB to 64MB)

1. The library starts many processes on the cluster

1b. One process is special: it is the master that orchstrates the execution

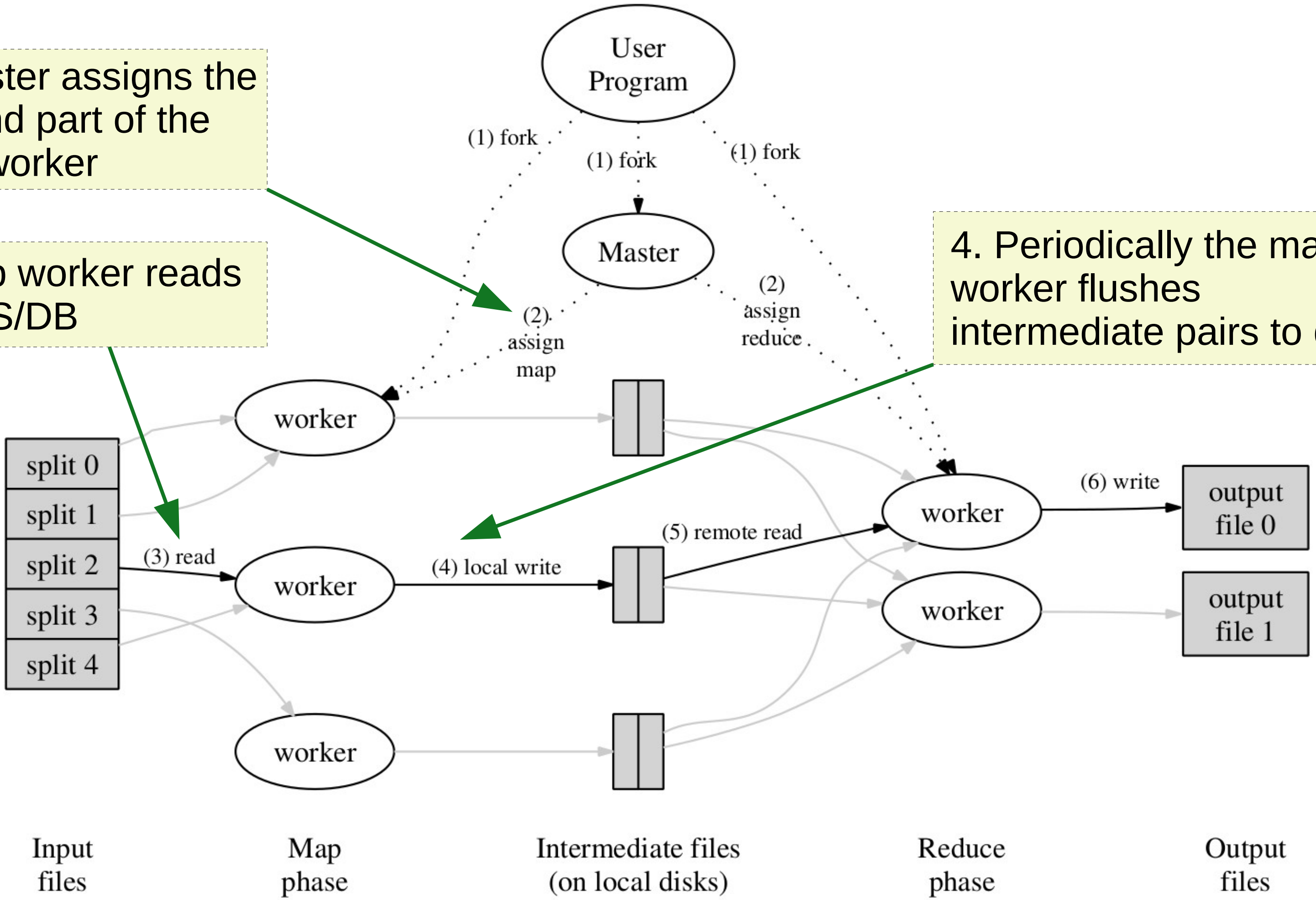


Execution overview

2. The Master assigns the map job and part of the input to a worker

3. The map worker reads from the FS/DB

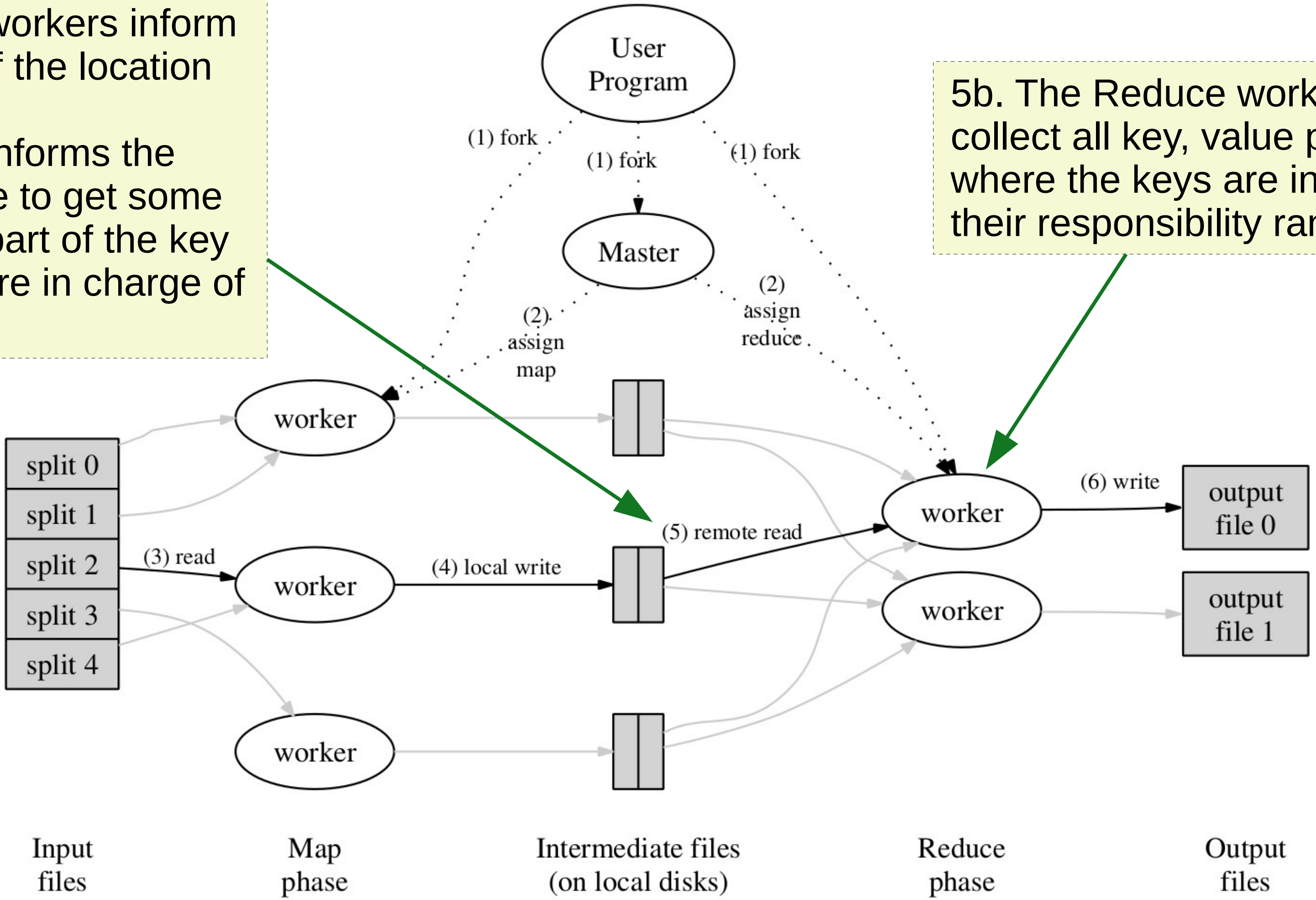
4. Periodically the map worker flushes intermediate pairs to disk



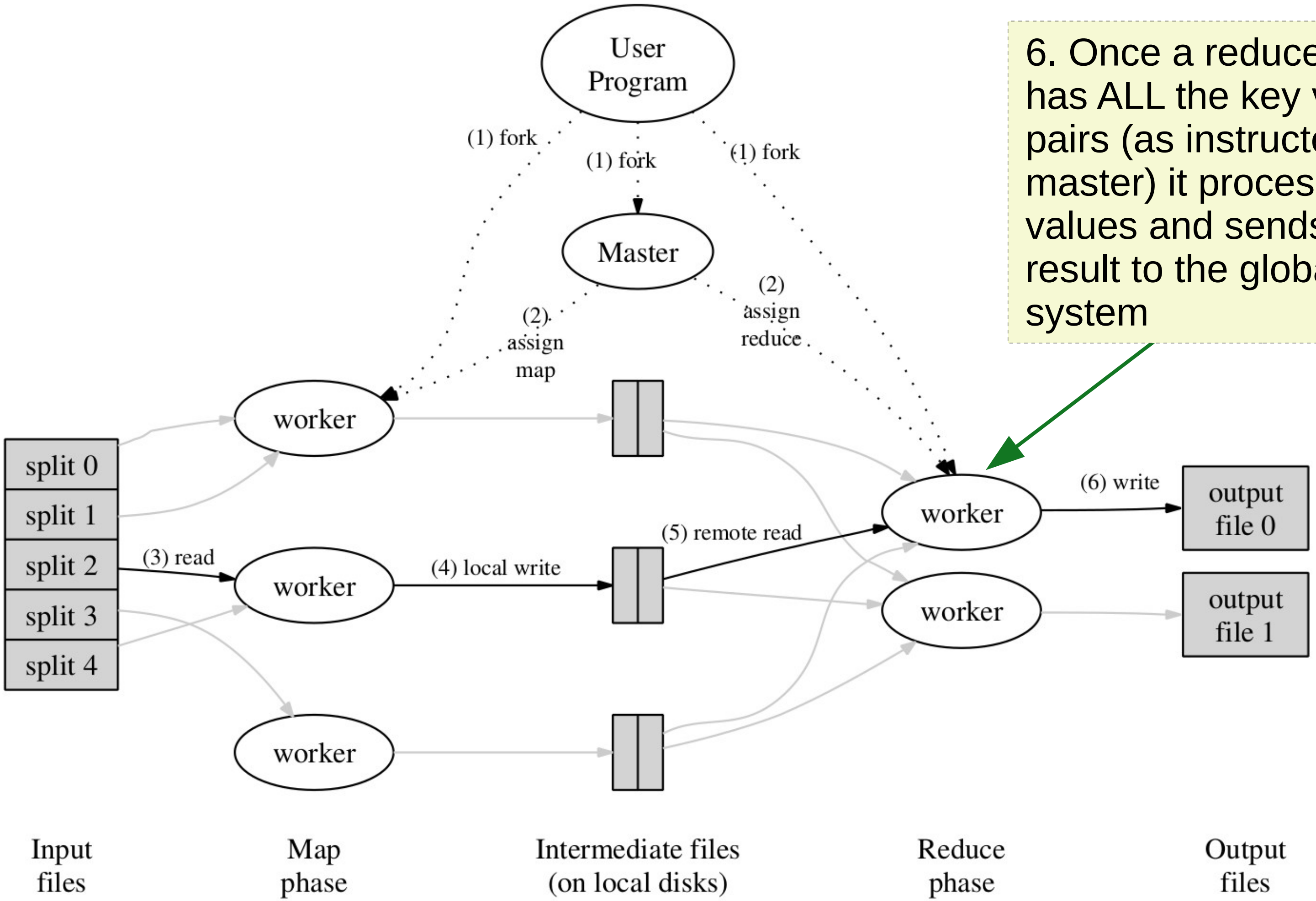
Execution overview

5. The Map workers inform the Master of the location of fresh data
The Master informs the worker where to get some data for the part of the key space they are in charge of (locally)

5b. The Reduce workers collect all key, value pairs where the keys are in their responsibility range

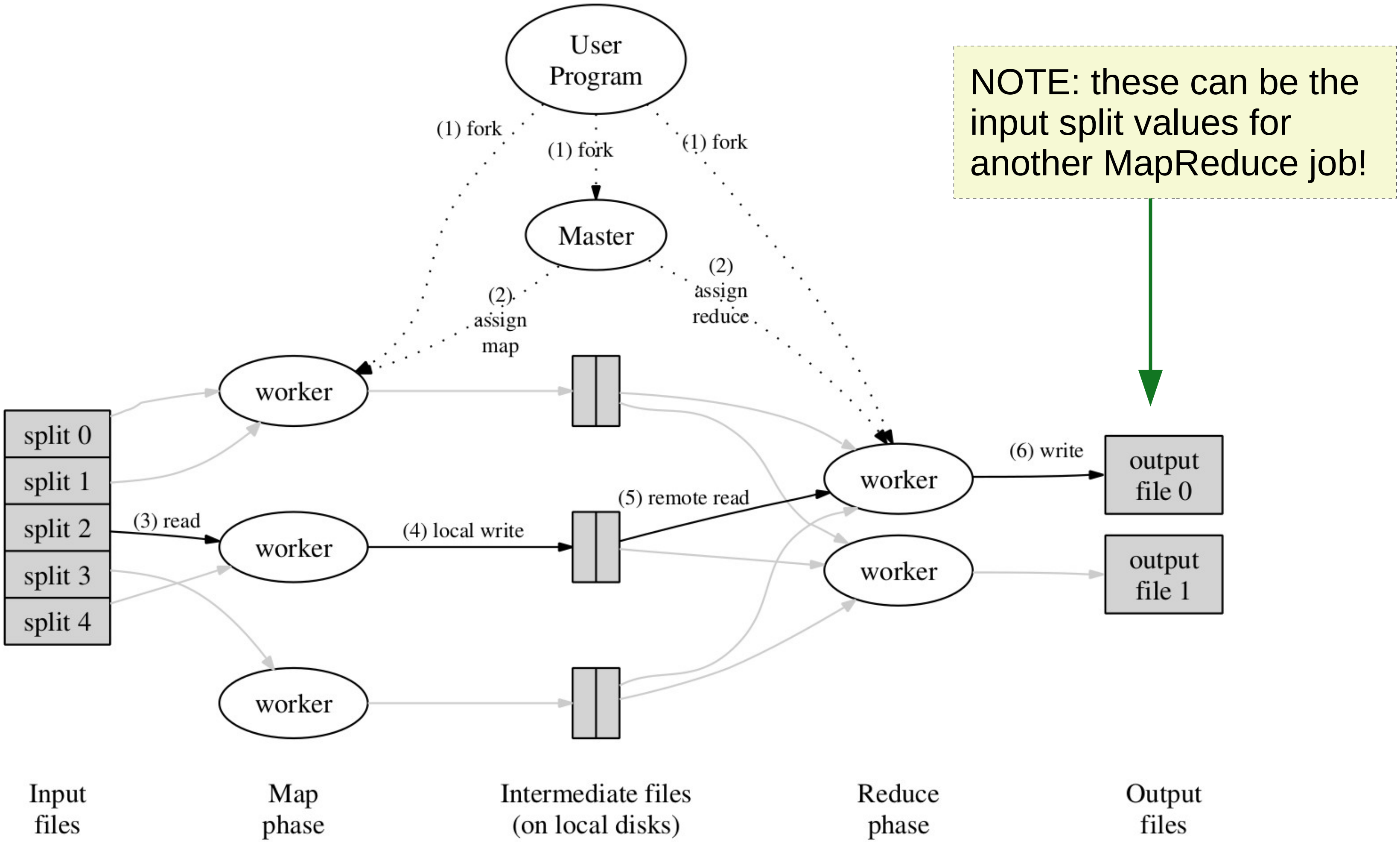


Execution overview



6. Once a reduce worker has ALL the key value pairs (as instructed by the master) it processes the values and sends the result to the global file system

Execution overview



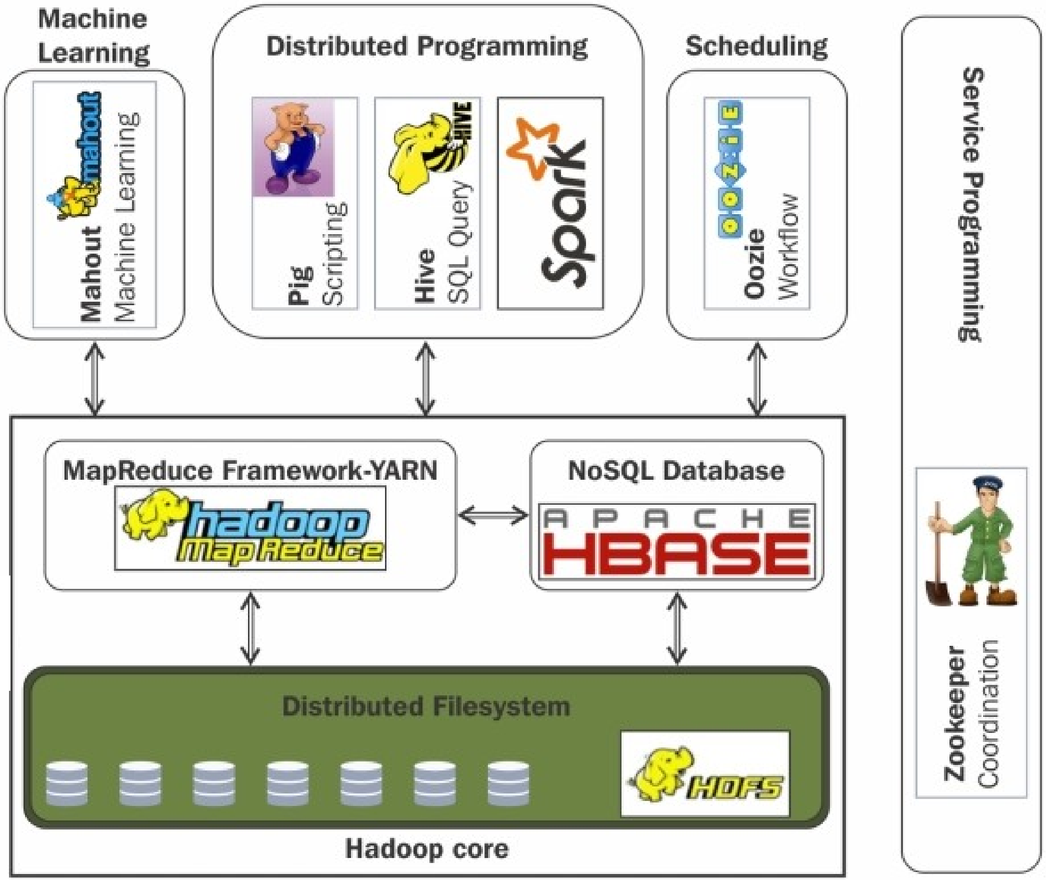
Implementation challenges

- Failing workers
  - Must detect and re-assign to another worker
  - Partitioning helps: partially processed data simply discarded
- Slow workers
  - Alive but slow worker can slow down entire job execution
  - Monitor work, assign fast workers work done by slow ones
    - Redundant computations, keep only first finisher
- Failing master
  - Original Map/Reduce: snapshot and retry
  - Today: use a coordination kernel!

Map/Reduce frameworks

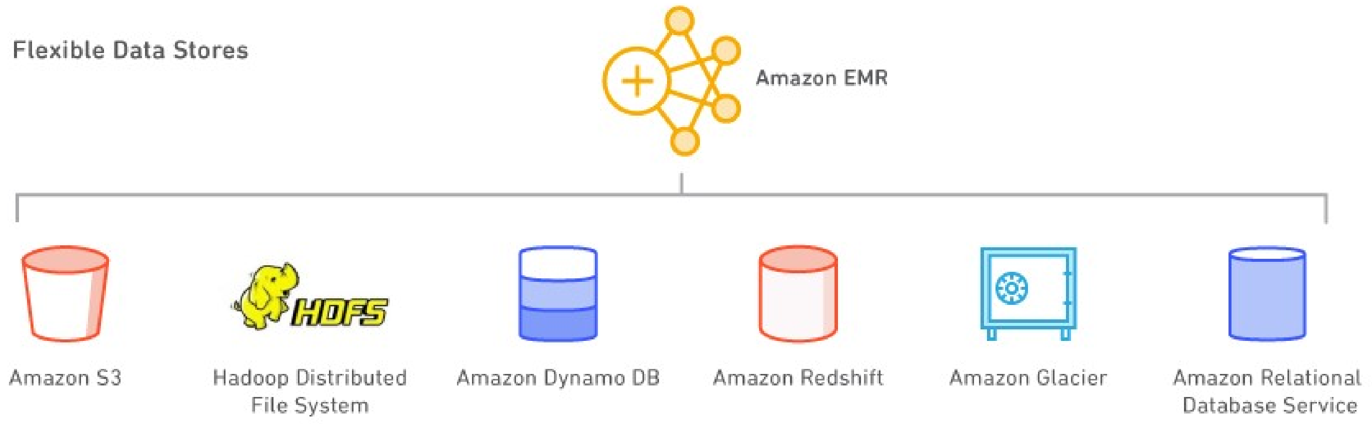
Map/Reduce implementations

- Original Google Map/ Reduce proprietary
- Hadoop: open source implementation of MapReduce and associated tools
  - File systems, workflow management



Map/Reduce as a service

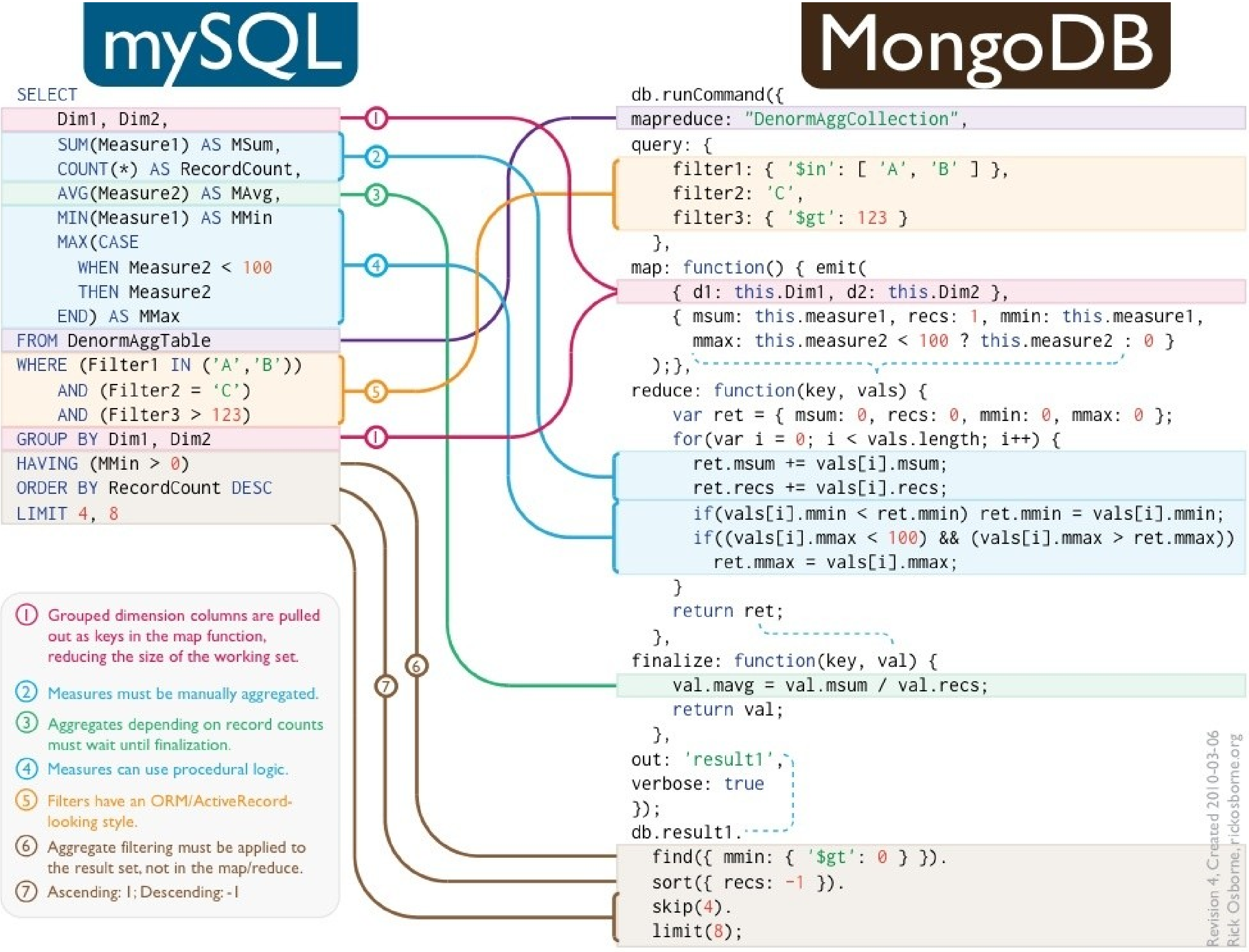
- Amazon Elastic Map Reduce: pre-configured versions of BigData frameworks including Hadoop, etc.
  - Can feed in data from S3, Dynamo DB, etc.



- Similar offers at other providers

Map/Reduce in NoSQL databases

- Hadoop is a heavy machinery
  - Useful for processing vast amounts of unstructured data
  - But Cloud application data is stored in NoSQL databases
- Map/Reduce calls supported by most NoSQL databases
  - Computations performed directly on top of the data



https://rickosborne.org/download/SQL-to-MongoDB.pdf

MongoDB - Reduce function

- The reduce function should not affect the outside system
- MongoDB will not call the reduce function for a key that has only a single value
- MongoDB can invoke the reduce function more than once for the same key
  - In this case, the previous output from the reduce function for that key will become one of the input values to the next reduce function invocation for that key
  - The type of the return object must be identical to the type of the value emitted by the map function
  - The reduce function must be **associative**
    - reduce(key, [C, reduce(key, [A, B])]) == reduce(key, [C, A, B])
  - The reduce function must be **idempotent**
    - reduce(key, [reduce(key, valuesArray)]) == reduce(key, valuesArray)
  - The reduce function must be **commutative**
    - reduce(key, [A, B]) == reduce(key, [B, A])

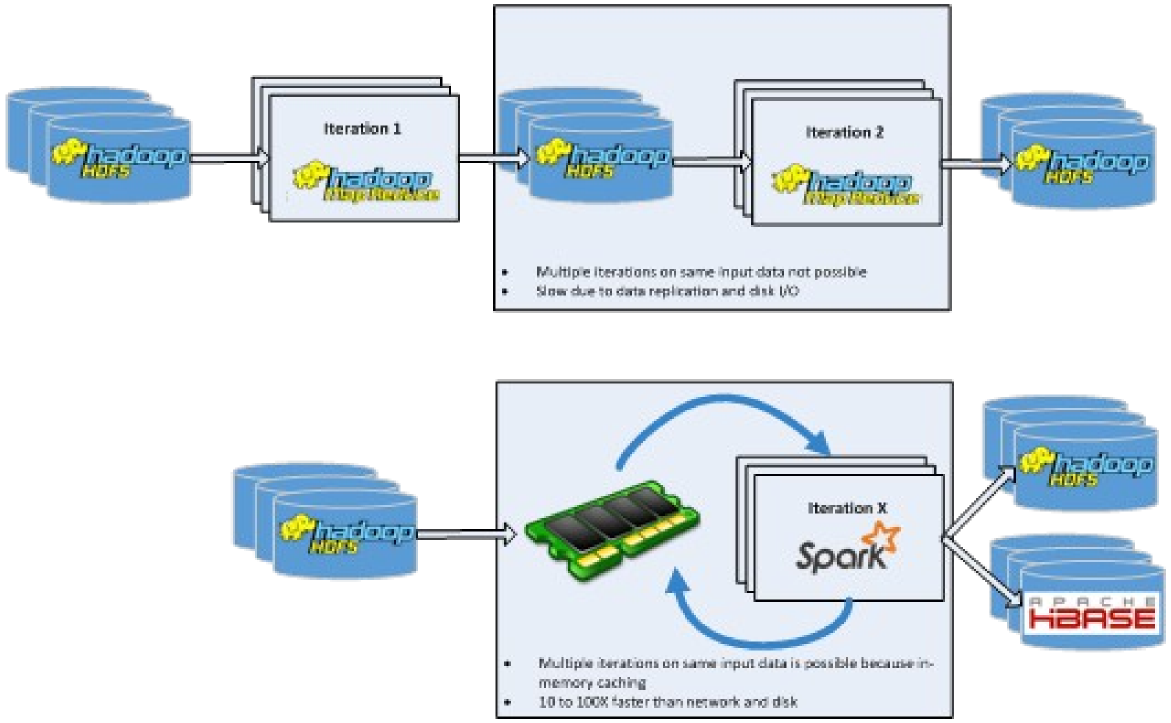
Source: MongoDB Documentation

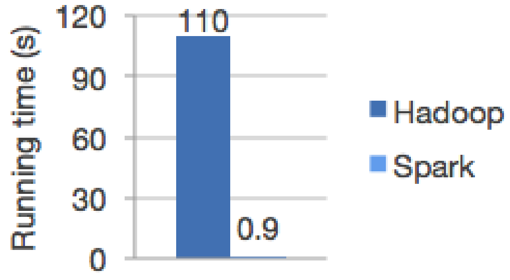
Beyond Map/Reduce

- Many algorithms require to iterate until stopping condition
  - K-Means: no point changes to new cluster
  - PageRank: ranking of webpage based on incoming links
  - Many *machine learning* algorithms
- And new usages
  - Algorithms applied on data streams
  - Interactive queries
- Map/Reduce intermediate storage (on distributed FS) is not well suited for these use cases
  - New abstrations have emerged to keep intermediary results in memory

- Writing to disk is slow and costly
  - Especially if we need to re-read immediately!
- Can we keep data in-memory instead?
- Apache Spark introduces Resilient Distributed Datasets
  - Shared memory abstraction
  - Programming model remains the same as Map/Reduce
  - Avoids re-reading from disk systematically
  - Builds different querying models on top of core engine
  - Can use pipeline of queries under different models

Spark VS Map/Reduce





Logistic regression in Hadoop and Spark

Conclusion

- Cloud-based web applications collect vast amounts of information about their clients, servers, infrastructure
  - Big Data processing
    - Necessary for Cloud scale
    - Enables new usages and applications
  - Process it in the background to derive information useful for business
- Map/Reduce: game changer for large-scale Big Data processing
  - Pioneered the use of a functional approach
  - Handles large-scale orchestration and fault tolerance
  - Specific (and simple) programming model
- Apache Spark improves/generalizes Map/Reduce principle
  - Support for iterative and interaction computations

References

- MapReduce: Simplified Data Processing on Large Clusters
  - Jeffrey Dean and Sanjay Ghemawat
    - OSDI'04: Sixth Symposium on Operating System Design and Implementation
- The Anatomy of a Large-Scale Hypertextual Web Search Engine
  - Sergey Brin, Lawrence Page
    - Computer Networks 30 - 1998
- Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing
  - Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, Ion Stoica
    - NSDI'12
- Scaling Spark in the Real World: Performance and Usability
  - Michael Armbrust, Tathagata Das, Aaron Davidson, Ali Ghodsi, Andrew Or, Josh Rosen, Ion Stoica, Patrick Wendell, Reynold Xin, Matei Zaharia
    - Proceedings of the VLDB Endowment - 2015