Πανεπιστήμιο Πατρών

MR Programming Framework (MapReduce / Hadoop)

Γ.Γαροφαλάκης. Σ.Σιούτας

MapReduce

Big Data Processing

- Crawled web documents (at Google, Bing, Yahoo!)
  - inverted indices (which pages contain each word)
  - graph representation of the links between pages
- Monitoring
  - Web requests logs:
    what were the most popular queries today?
  - How did users click on ads in the last month?
    (who should pay for adwords traffic?)
- Information retrieval, machine learning, AI.
- Numerical mathematics
- Bioinformatics...

Big Data processing: characteristics

- Most of these computations are conceptually straightforward on a single machine
- But the volume of data is HUGE
  - Need to use many (1.000s) of computers together to get results in a reasonable amount of time
  - Management of parallelization, data distribution, failures handling, etc. => much more complex than the computation itself

MapReduce

- Simplifying model for large-scale data processing
  - Inspired by functional programming paradigm
    - LISP (**LIS**t **P**rocessing)
  - Adapted to embarrassingly parallel workloads
    - Lots of concurrent operations on separate parts of the data with little or no synchronization
  - Runtime support for parallelization, data distribution, failures handling, etc.

- Implementations
  - Google’s own C++ implementation
  - Hadoop Java open-source implementation
  - Many more in commercial and open-source products

Outline

- Some background on functional programming
- MapReduce as seen by the programmer
- Execution and runtime support
- Examples
- Some optimizations/extensions
- Hadoop

Functional Programming

- FP = computation as application of functions
  - Theoretical ground = lambda calculus

- How is it different from imperative programming?
  - Traditional notions of ‘data’ and ‘instructions’ are not applicable
    - Execution = evaluation of *functions*
  - *Functions* in the sense of mathematical functions
    - Referential transparency: no side effects in the function (such as updating shared state) -- unlike Java or C
    - Calling a function twice with the same arguments always returns the same value
  - Data flows are implicit in the program
    - Different orders of execution are possible

Some functional languages

- OCaml, Scala, ML, Haskell, Scheme, F# (in MS .NET), etc.

- Some languages are hybrids between imperative and functional styles
  - JavaScript, Lua, etc.

- In some aspects, a subset of SQL and Spreadsheets (Excel without VB macro) are forms of functional programming languages

- Let’s take the example of LISP

The example of LISP

- Lisp ≠ **L**ost **In** **S**illy **P**arentheses
  - Lists are a primitive data type

```
'(1 2 3 4 5)
'((a 1) (b 2) (c 3))
```

  - Functions written in prefix notation

```
(+ 1 2) → 3
(* 3 4) → 12
(sqrt (+ (* 3 3) (* 4 4))) → 5
(define x 3) → x
(* x 5) → 15
```

Functions

- Functions = lambda expression bound to variables

```
(define foo
  (lambda (x y)
    (sqrt (+ (* x x) (* y y)))))
```

- Syntactic sugar for defining functions
  - The expression above is equivalent to:

```
(define (foo x y)
  (sqrt (+ (* x x) (* y y))))
```

- Once defined, functions can be applied:

```
(foo 3 4) → 5
```

Other features

- In Lisp/Scheme, everything is an s-expression
  - No distinction between ‘data’ and ‘code’
  - Easy to write self-modifying code

- Higher-order functions
  - Functions that take other functions as arguments

```
(define (bar f x) (f (f x)))
```

Doesn’t matter what f is, just apply it twice.

```
(define (baz x) (* x x))
(bar baz 2) → 16
```

Recursion is your friend

- Simple factorial example

```
(define (factorial n)
  (if (= n 1)
      1
      (* n (factorial (- n 1)))))
(factorial 6) → 720
```

- Even iteration is written with recursive calls!

```
(define (factorial-iter n)
  (define (aux n top product)
    (if (= n top)
        (* n product)
        (aux (+ n 1) top (* n product))))
  (aux 1 n 1))
(factorial-iter 6) → 720
```

Lisp → MapReduce

- But what does this have to do with MapReduce?
  - After all, Lisp is about processing lists

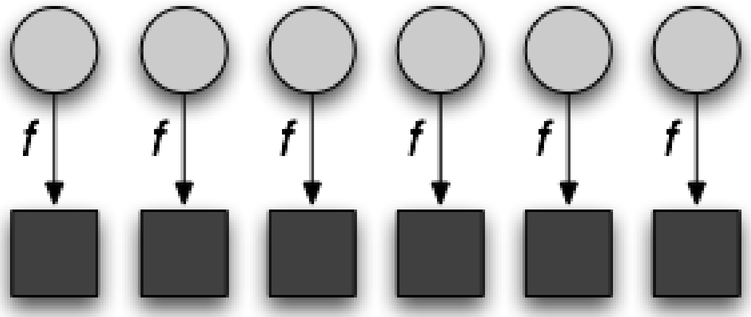
- Two important concepts (first class higher order functions) in functional programming
  - Map: do something to everything in a list
  - Fold: combine results of a list in some way

Map

- Map is a higher-order function

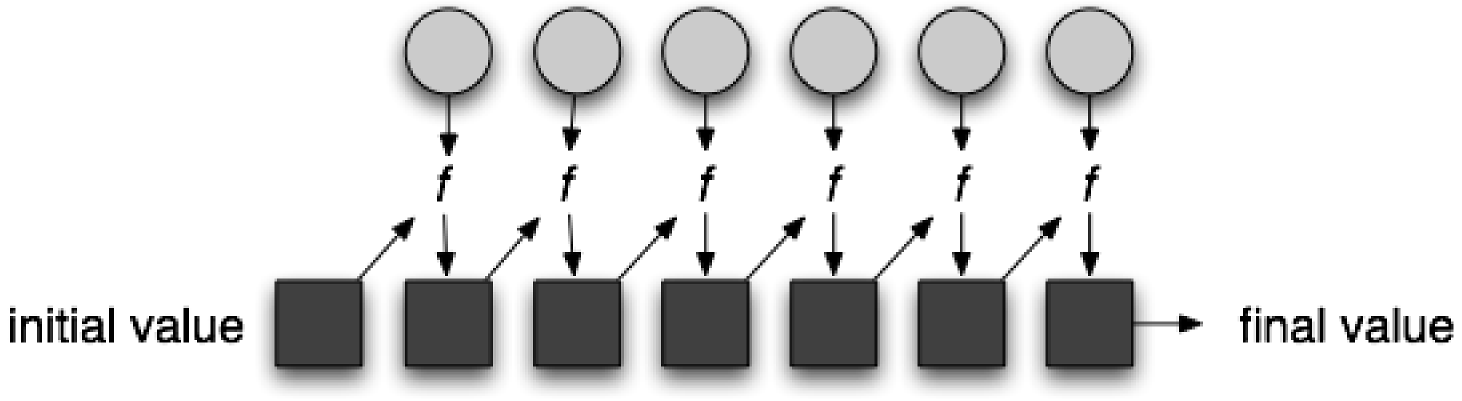
- How map works:
  - Function is applied to every element in a list
  - Result is a new list

- Note that each operation is independent and, due to referential transparency (no side effects of functions evaluation), applying *f* on one element and re-applying it again will always give the same result



Fold

- Fold is also a higher-order function
- How fold works:
  - Accumulator set to initial value
  - Function applied to list element and the accumulator
  - Result stored in the accumulator
  - Repeated for every item in the list
  - Result is the final value in the accumulator



Map/Fold in action

- Simple map example:

```
(map (lambda (x) (* x x))
      '(1 2 3 4 5))
 → '(1 4 9 16 25)
```

- Fold examples:

```
(fold + 0 '(1 2 3 4 5)) → 15
(fold * 1 '(1 2 3 4 5)) → 120
```

- Sum of squares:

```
(define (sum-of-squares v)
  (fold + 0 (map (lambda (x) (* x x)) v)))

(sum-of-squares    '(1 2 3 4 5)) → 55
```

Lisp → MapReduce

- Let’s assume a long list of records: imagine if...
  - We can parallelize map operations
  - We have a mechanism for bringing map results back together in the fold operation
- That’s MapReduce!
- Observations:
  - No limit to map parallelization since maps are independent
  - We can reorder folding if the fold function is commutative and associative

Typical processing

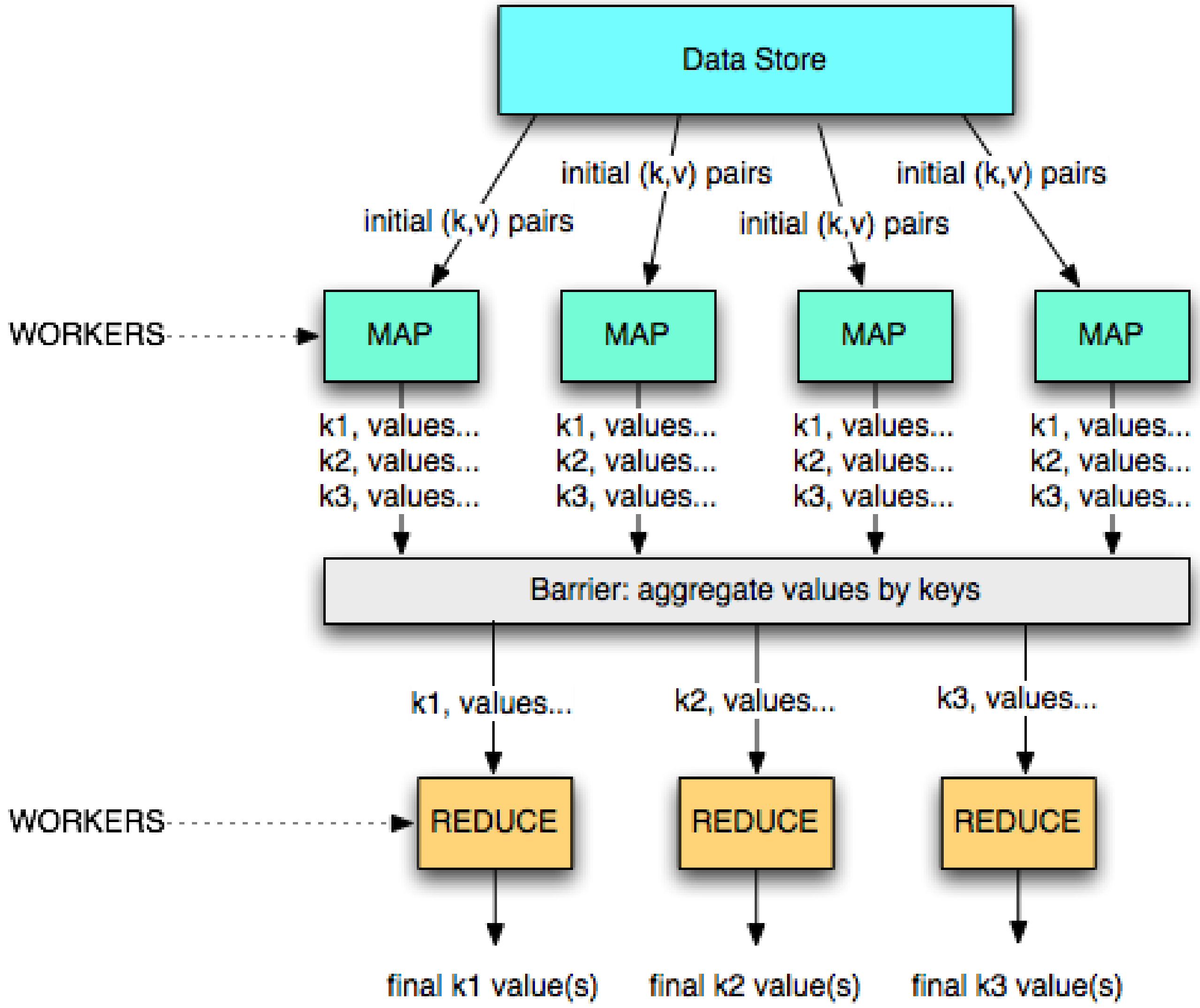
- Iterate over a large number of records
- Extract something of interest from each MAP
- Shuffle and sort intermediate results
- Aggregate intermediate results REDUCE
- Generate final output

- Key idea: provide an abstraction at the point of these two operations, make others implicit

MapReduce: Programmers' View

- Programmers specify two functions:
  - map (k, v) → <k', v'>*
  - reduce (k', v') → <k'', v''>*
    - All v' with the same k' are reduced together
- Usually, programmers also specify a **partition** function:
  - partition (k', number of partitions n) → partition for k'
  - Often a simple hash of the key, e.g., hash(k') mod n
  - Allows reduce operations for different keys in parallel
- MapReduce jobs are submitted to a scheduler that allocates the machines and deals with scheduling, fault tolerance, etc.

A divide and conquer approach



When is MapReduce relevant

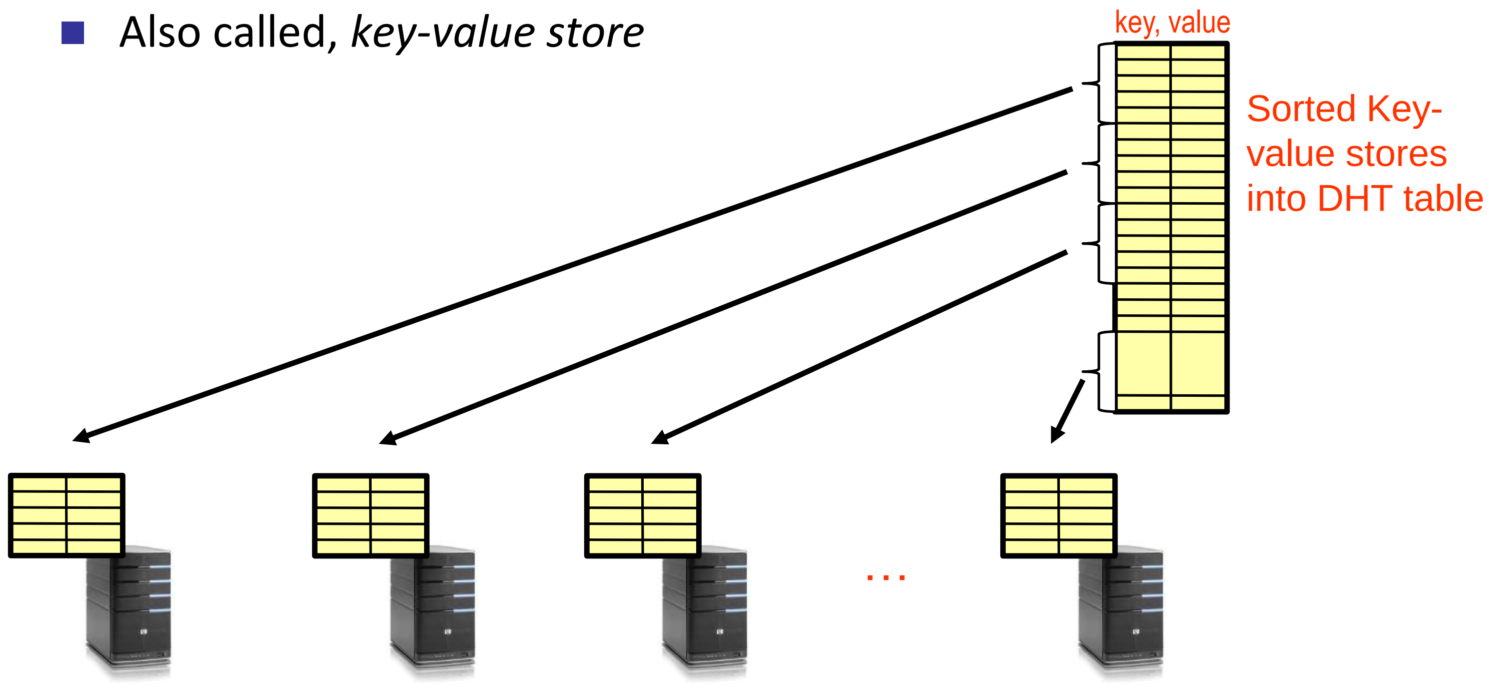
- Good choice for:
  - Log files indexing/analysis
  - Sorting huge data volumes
  - Image processing, etc.
- Bad choice for:
  - Computing the first 1,000,000 digits of π
  - Computing Fibonacci series
  - Replacing MySQL

Job Execution on a MapReduce cluster

- 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
- The communication is based on key/value pairs
  - e.g., a DHT or key-value store
  - Store the location of data in the DHT and use direct communication between workers
- Map jobs get the data from a storage layer
  - unstructured: file system (e.g., Google File System, HDFS, or local file system)
  - (semi-)structured: local database (e.g., MySQL)
    or distributed database (e.g., Google's BigTable)

DHTs

- Distribute (partition) a hash table data structure across a large number of servers
  - Also called, *key-value store*

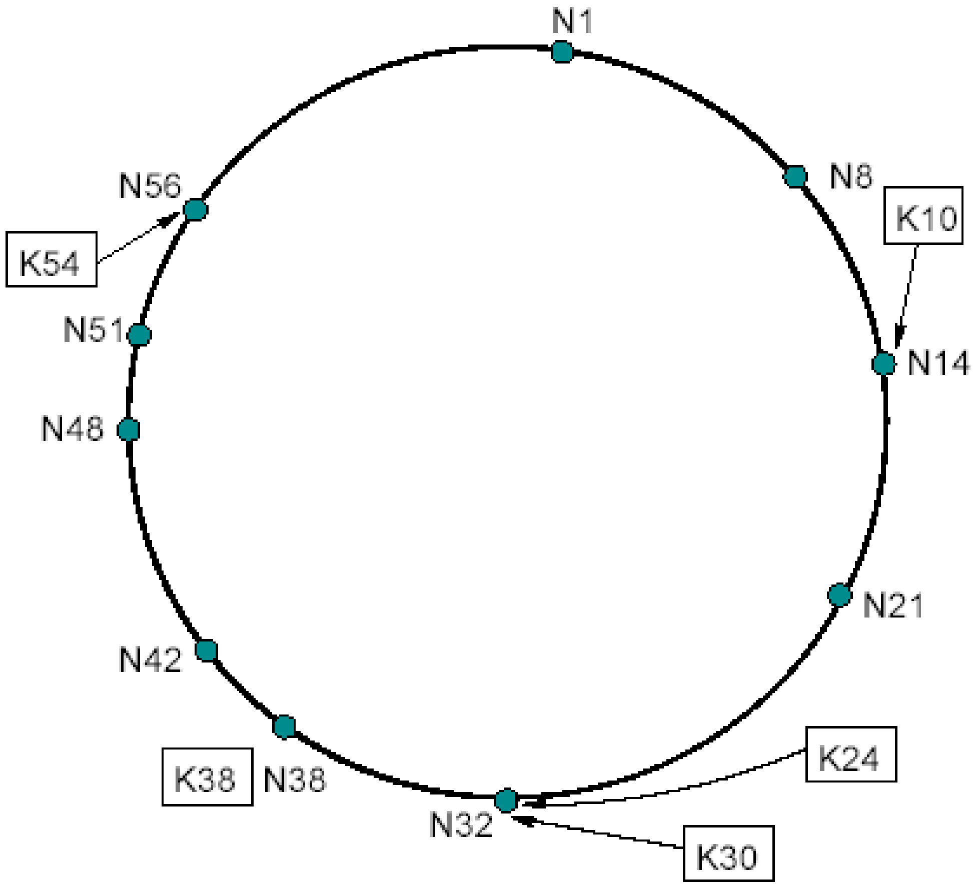


- **Key identifier = SHA-1(key), Node identifier = SHA-1(IP address)**
- **Each key_id is mapped to the node_id with the smallest node_id >= key_id**
- Two operations
  - **put**(key, data); // insert "data" identified by "key"
  - data = **get**(key); // get data associated to "key"

m-bit κλειδιά (Key_IDs και Node_IDs) πάνω στον δακτύλιο Chord

- **Key_ID = SHA-1(key) mod 2^m**
- **Node_ID = SHA-1(IP address) mod 2^m**

- **K10→ N14**
- **K24, K30→N32**
- **K38→ N38**
- **K54→N56**



Distributed Hash Tables (DHTs) (cont'd)

- Just need a lookup service, i.e., given a key (ID), map it to machine n

```
  n = lookup(key);
```

- Invoking **put()** and **get()** at node **m**

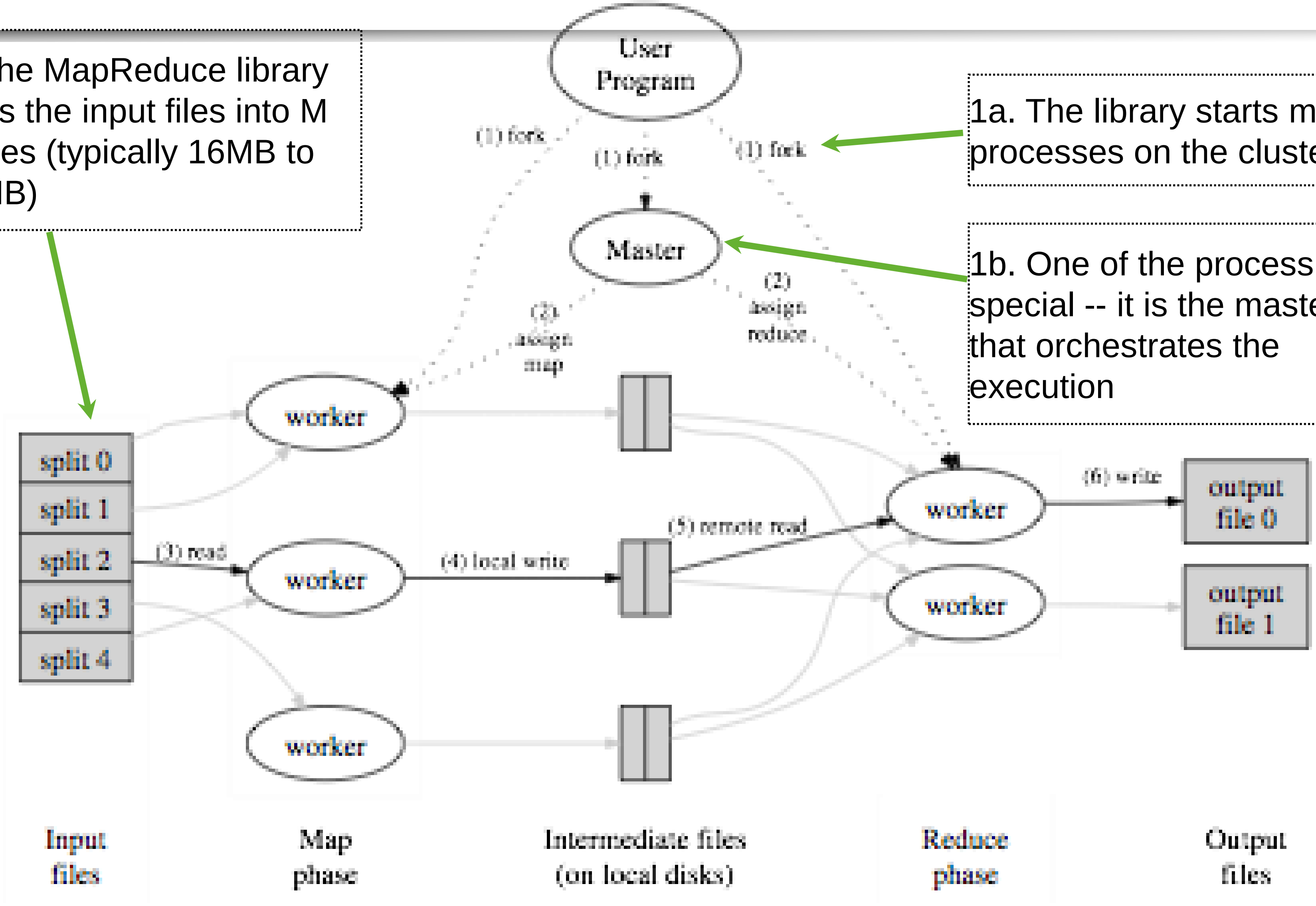
```
m.put(key, data)
{
  n= lookup(key); // get node "n" mapping "key"
  n.store(key, data); // store data at node "n"
}

data = m.get(key)
{
  n= lookup(key); // get node "n" storing data associated to "key"
      return n.retrieve(key); // get data stored at "n" associated to "key"
}
```

Execution overview

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

1a. The library starts many processes on the cluster

1b. One of the process is special -- it is the master that orchestrates the execution



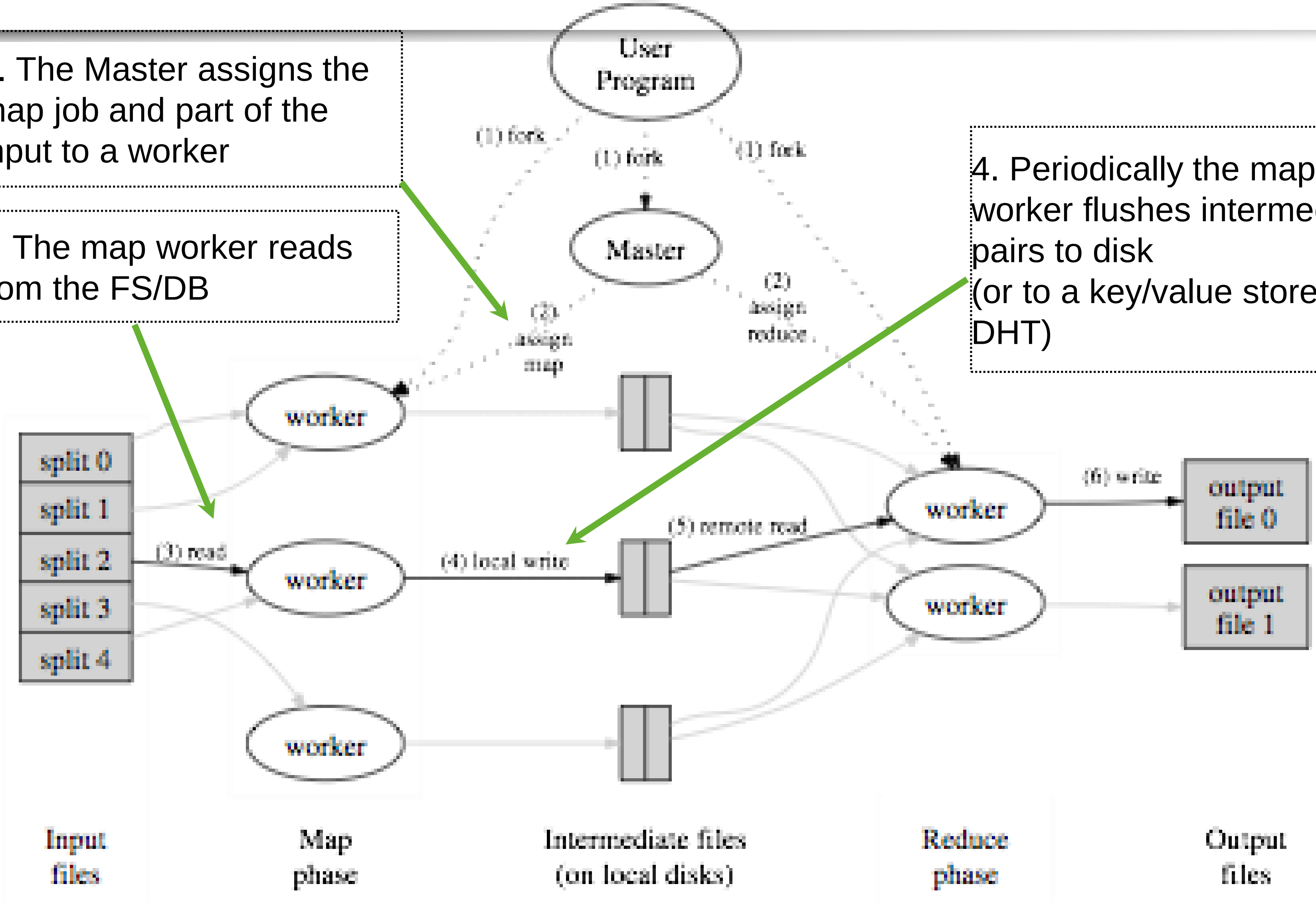
(from the original MapReduce paper)

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
(or to a key/value store or DHT)



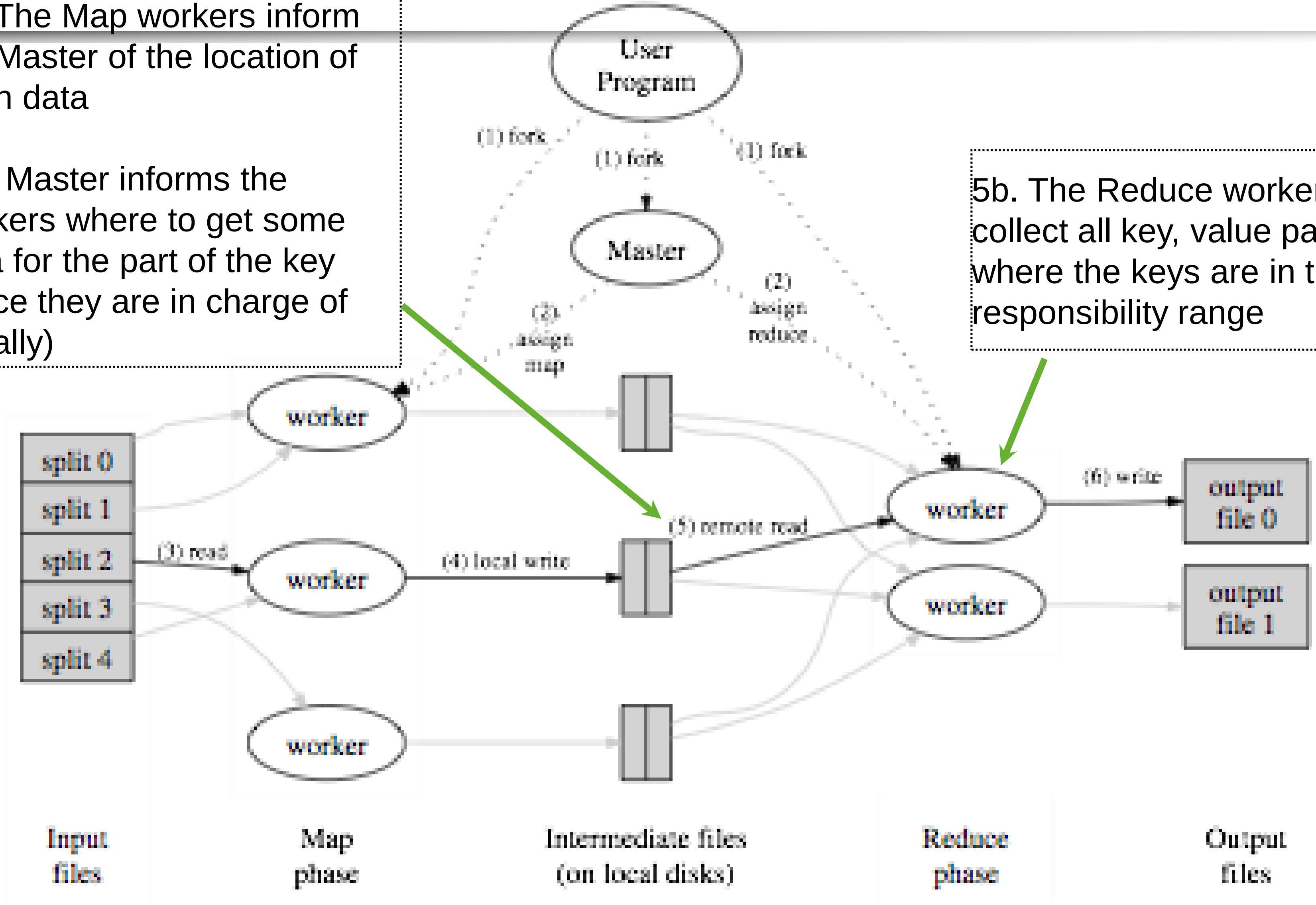
(from the original MapReduce paper)

Execution overview

5a. The Map workers inform the Master of the location of fresh data

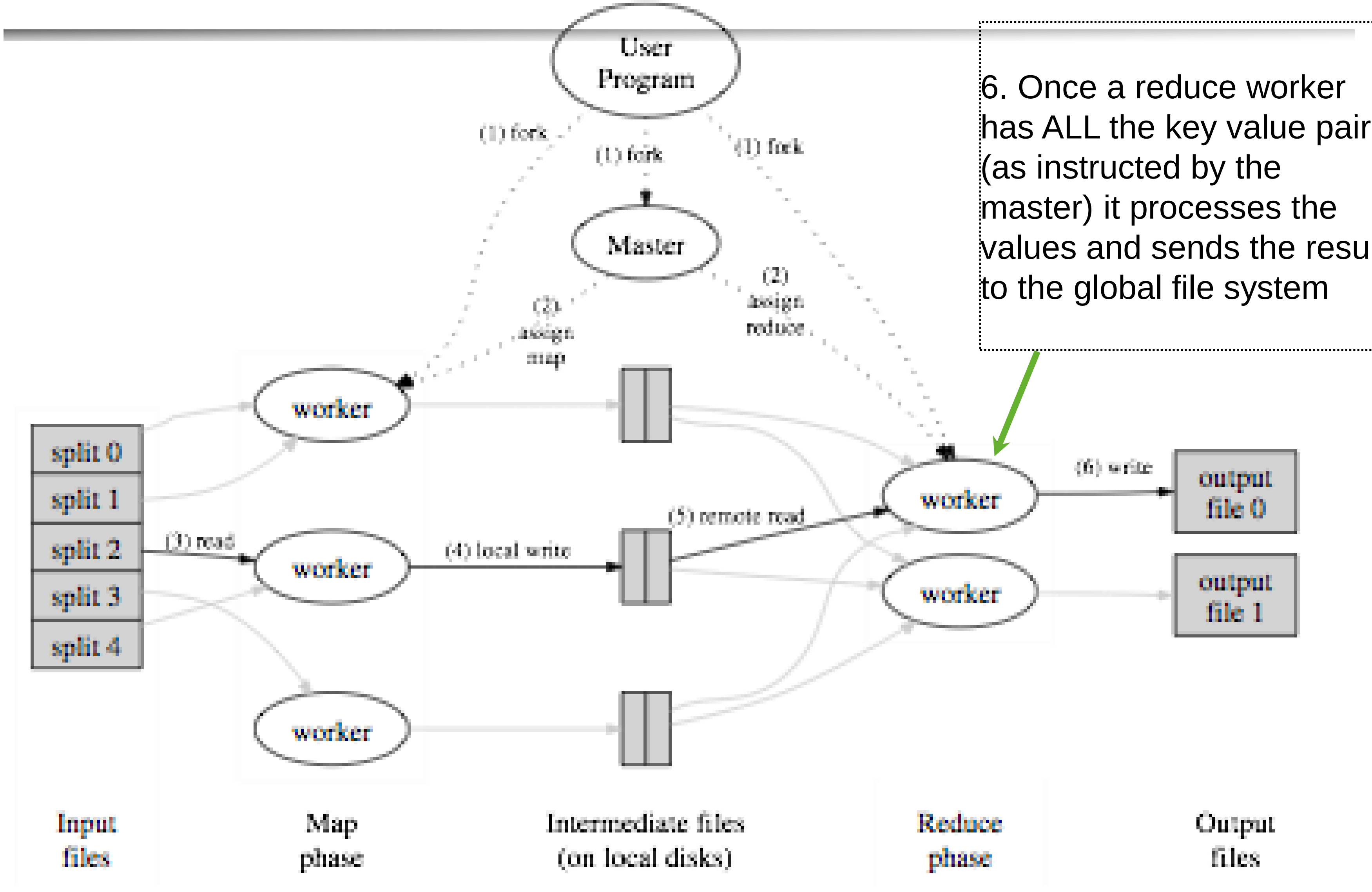
The Master informs the workers where to get some data for the part of the key space they are in charge of (locally)

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



(from the original MapReduce paper)

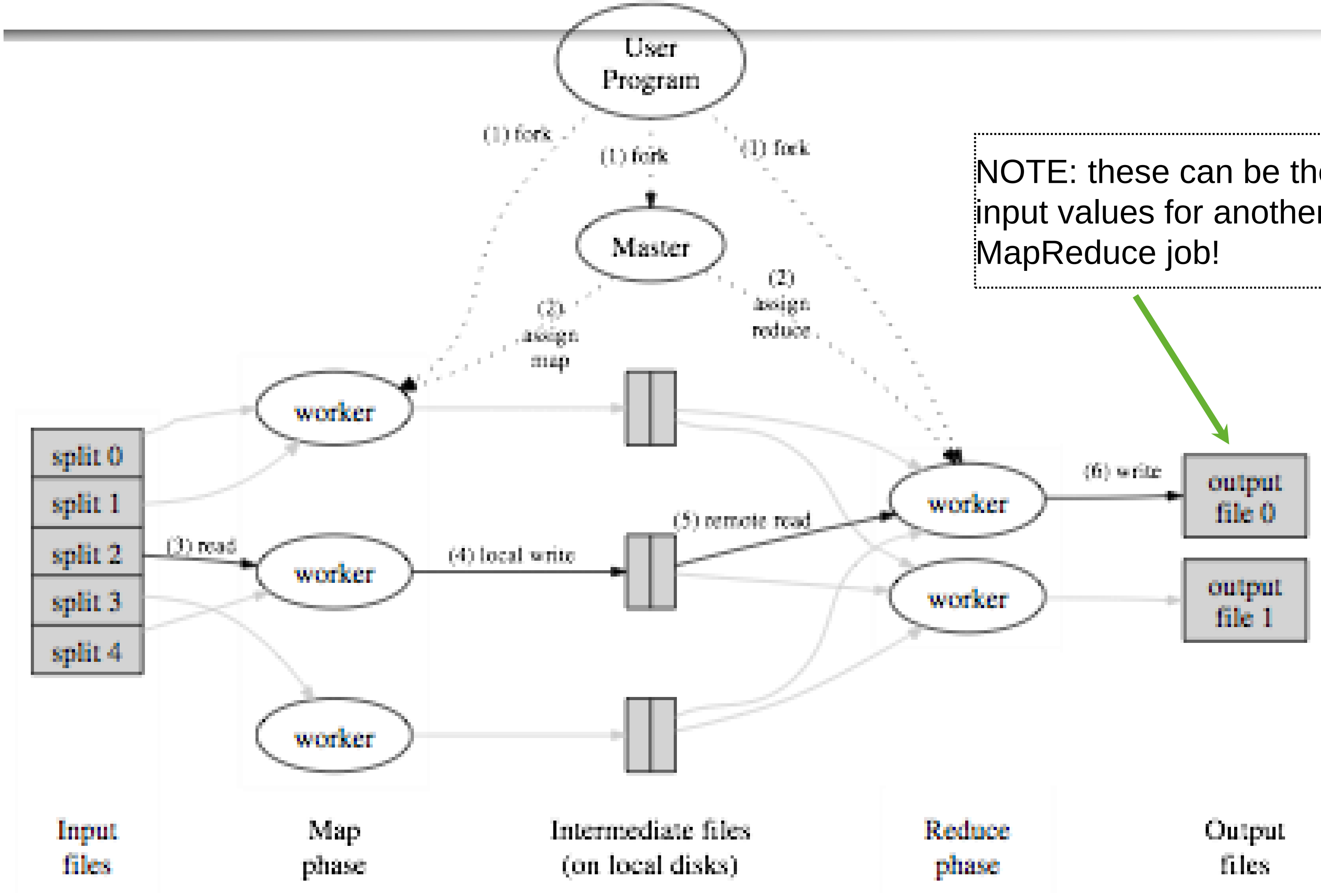
Execution overview



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

(from the original MapReduce paper)

Execution overview



NOTE: these can be the input values for another MapReduce job!

(from the original MapReduce paper)

MapReduce data flow

- Each **map job** gets part of the data to process
  - It outputs a set of **<key,value>** pairs to an intermediate storage
  - Each key is associated to a set of values

- The **reduce jobs** start once all map jobs have completed
  - (barrier)
  - It is often possible to aggregate partially the values before all maps are complete (e.g., on the same machine as the map job)
    - see the **combine** operation in a few slides

- Each reduce job gets the set of values associated to one key
  - Shuffling/sort operation aggregates the map results for each key
  - Each reduce job gets a set of keys and the associated values for each of the keys
    - Using an iterator provided by the MapReduce implementation
  - And outputs a final <key,value(s)> pair (or a boolean, a log, ...)

MapReduce Examples

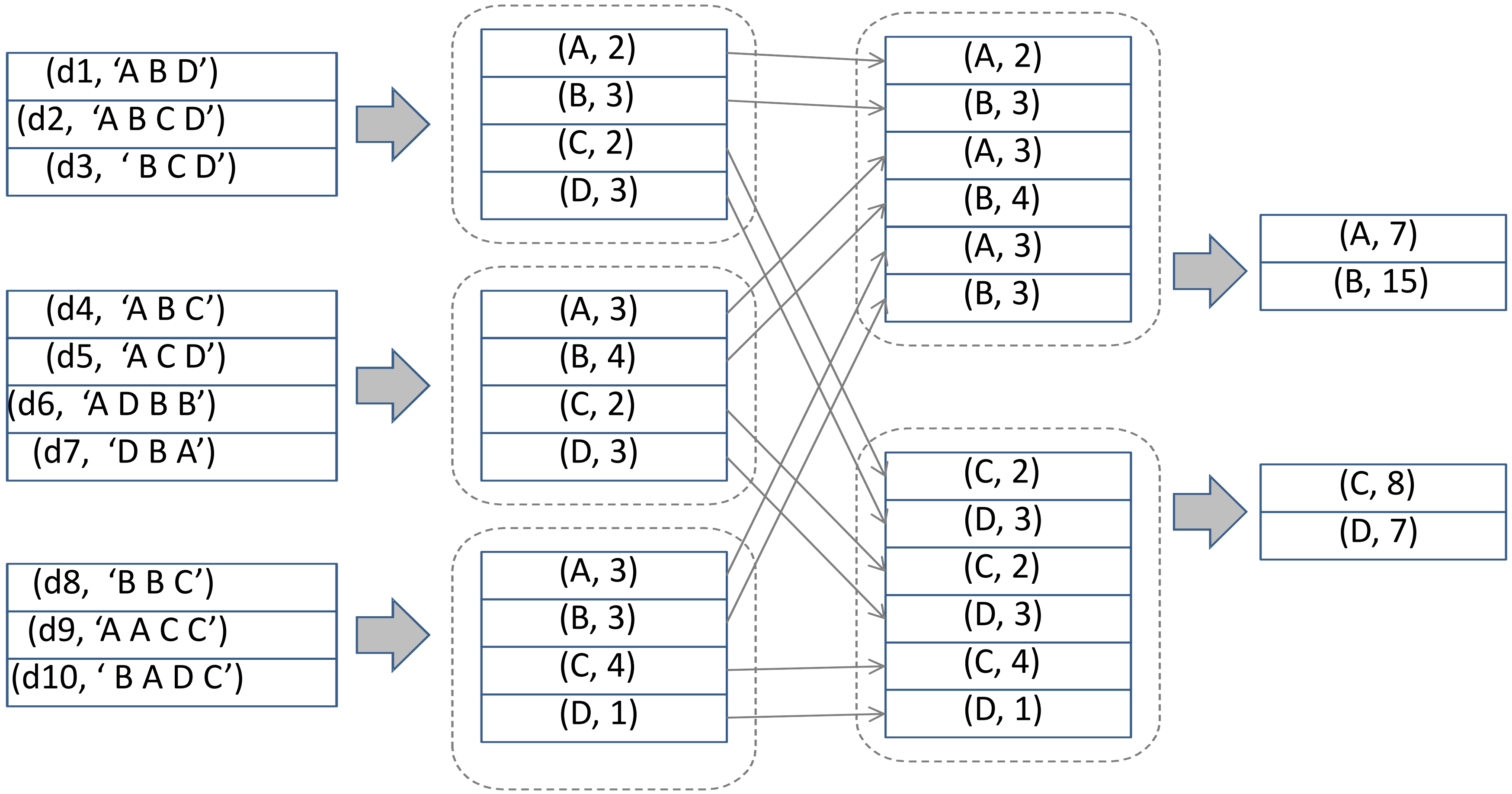
Example 1: word count

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

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

Reduce(String key, Iterator intermediate_values):
     // key: a word, same for input and output
     // intermediate_values: a list of counts
     int result = 0;
     for each v in intermediate_values:
          result += ParseInt(v);
     Emit(AsString(result));
```

(complete C code in the OSDI MapReduce paper)

Example 1: word count



M=3 mappers

R=2 reducers

MapReduce Programming Model

- Data type: key-value *records*

- Map function:

  $(K_{in}, V_{in}) \rightarrow list(K_{inter}, V_{inter})$

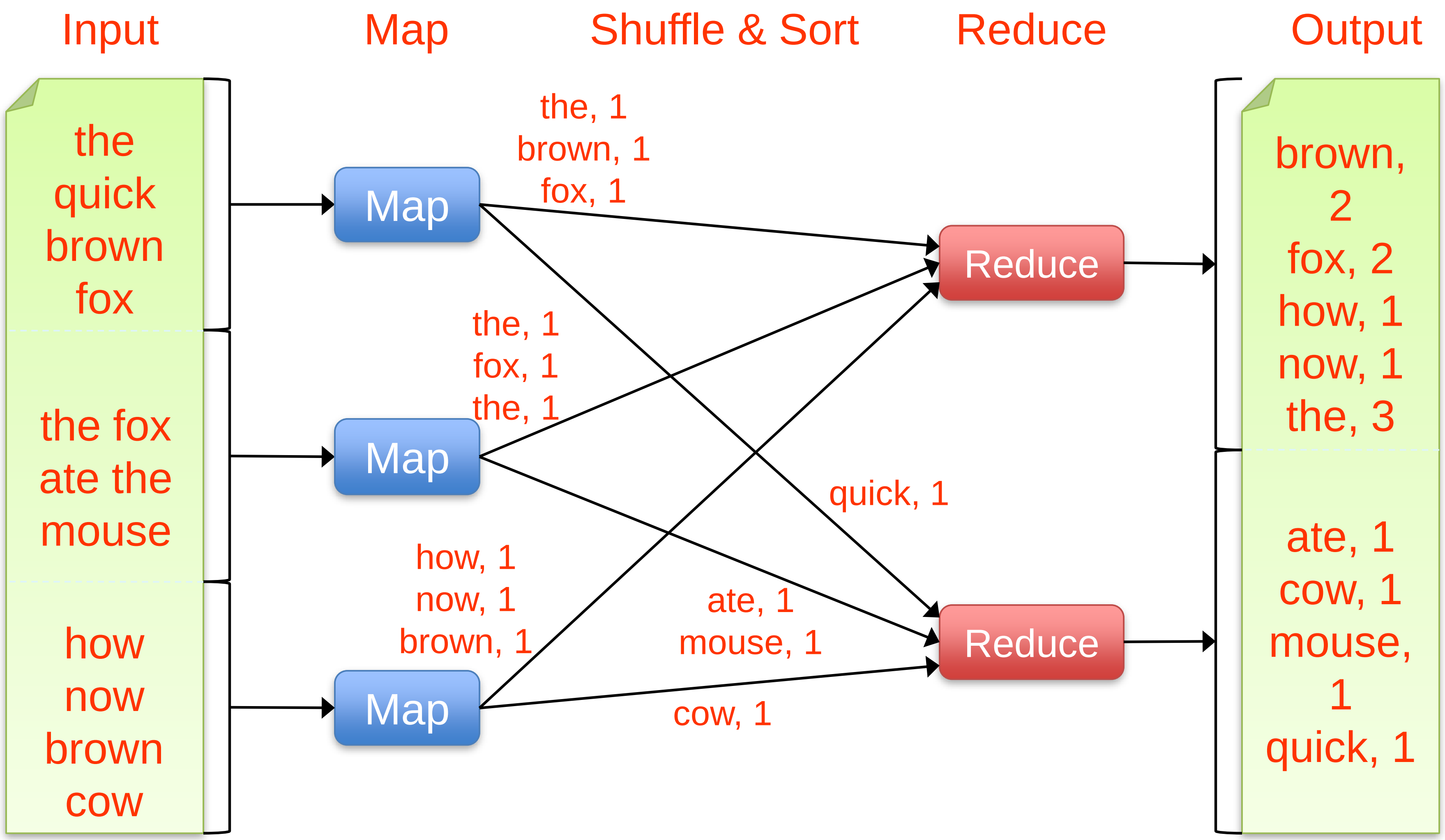
- Reduce function:

  $(K_{inter}, list(V_{inter})) \rightarrow list(K_{out}, V_{out})$

Example: Word Count

```
def mapper(line):
foreach word in line.split():
        output(word, 1)


def reducer(key, values):
    output(key, sum(values))
```

Word Count Execution



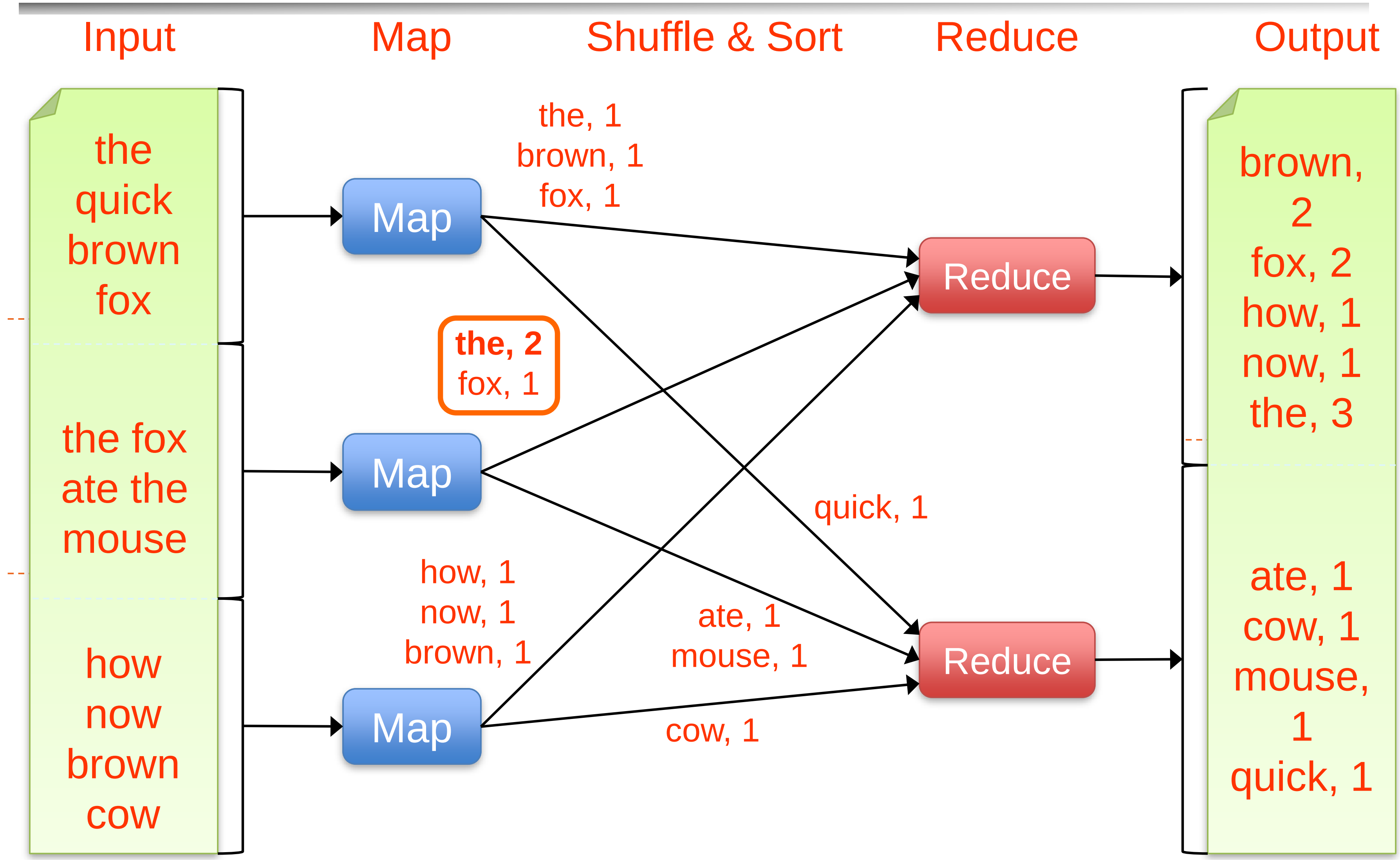
An Optimization: The Combiner

- Local reduce function for repeated keys produced by same map
- For associative ops. like sum, count, max
- Decreases amount of intermediate data

- Example: local counting for Word Count:

```
def combiner(key, values):
    output(key, sum(values))
```

Word Count with Combiner

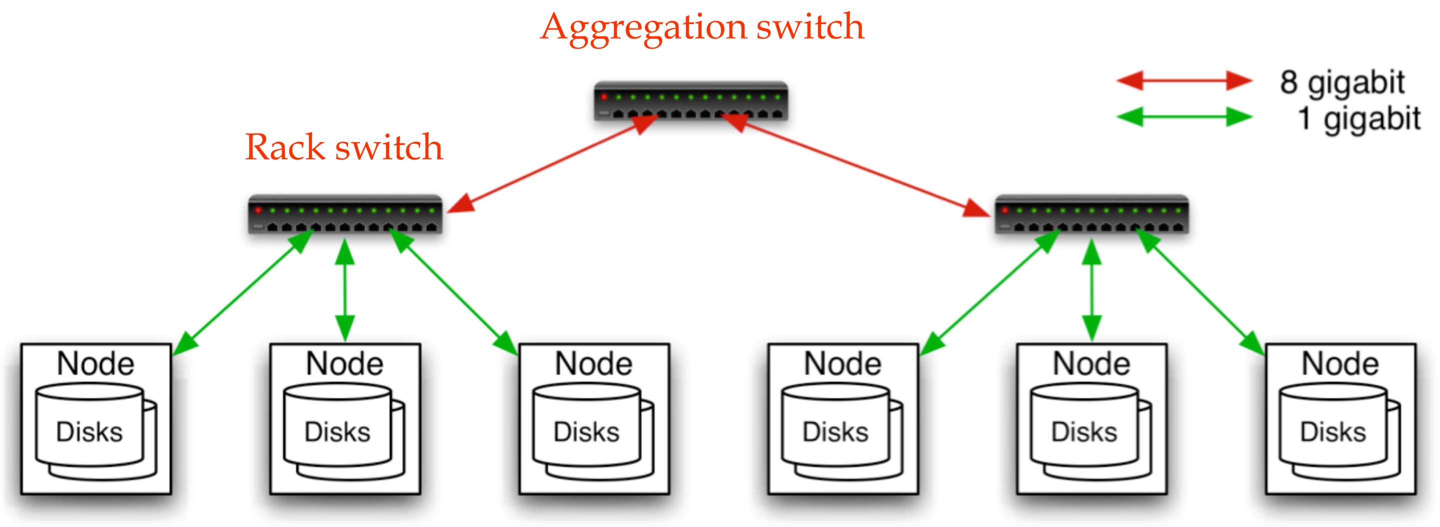


MapReduce Execution Details

- Mappers preferentially scheduled on same node or same rack as their input block
  - Minimize network use to improve performance
- Mappers save outputs to local disk before serving to reducers
  - Allows recovery if a reducer crashes
  - Allows running more reducers than # of nodes

Hadoop Cluster

Typical Hadoop Cluster



- 40 nodes/rack, 1000-4000 nodes in cluster
- 1 Gbps bandwidth in rack, 8 Gbps out of rack
- Node specs (Facebook):
  8-16 cores, 32-48 GB RAM, 10×2TB disks

Example 2: distributed grep

- Grep reads a file line by line, and if a line matches a pattern (e.g., regular expression), it outputs the line
- Map function
  - read a file or set of files
  - emit a line if it matches the pattern
    - key = original file (or unique key if origin file does not matter)
- Reduce function
  - identity (use intermediate results as final results)

Example 3: URL access frequency

- Input: log of web page requests (after a query)
- Output: how many times each URL is accessed
  - Variant: what are the top-k most-accessed URLs?
- Map function
  - Parse the log, output a <URL, 1> pair for each access
- Reduce function
  - For each key URL, a list of *n* "1" is associated (i.e., added)
  - Emit a final pair <URL, n>

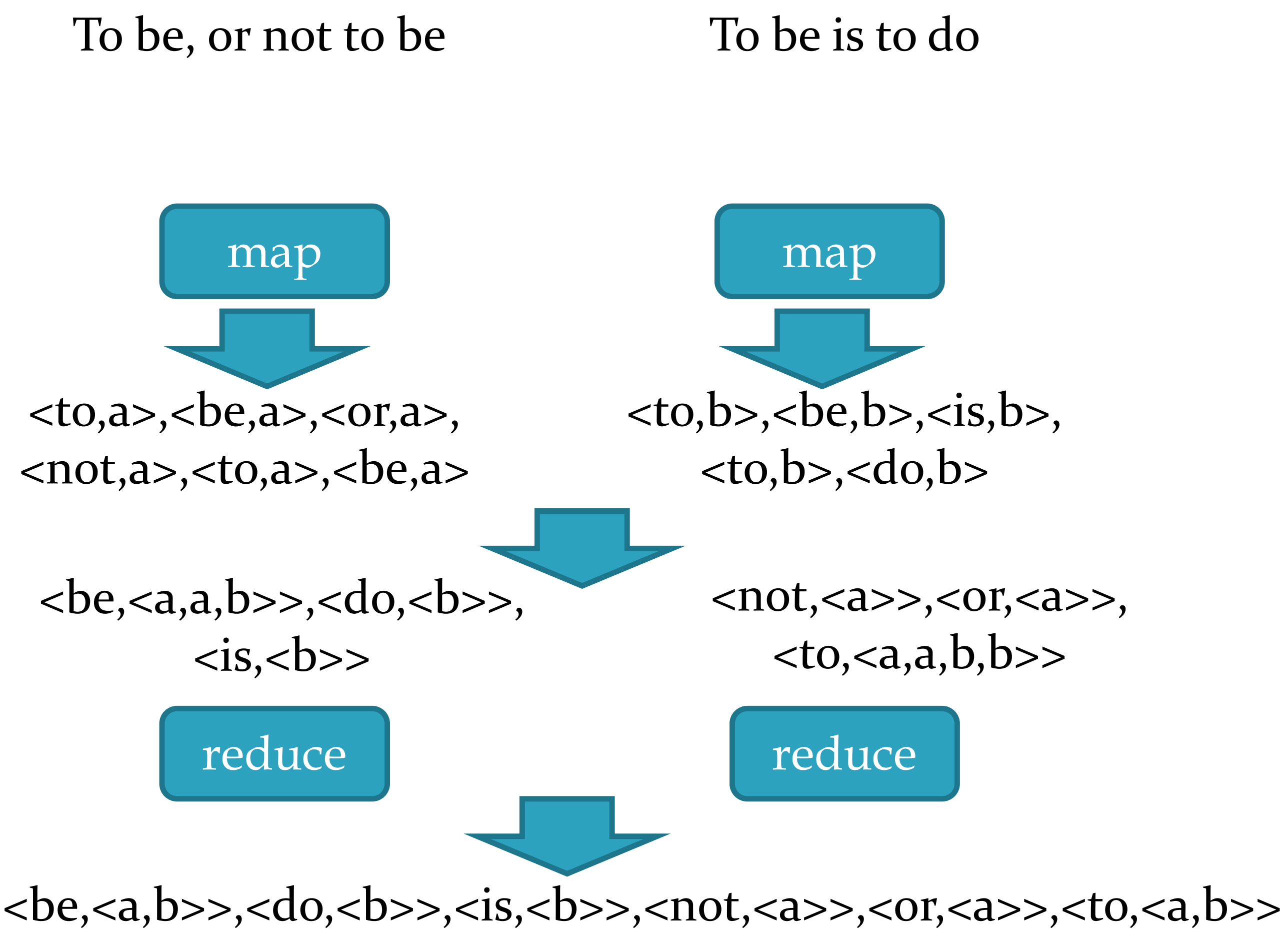
Example 4: Reverse Web-link graph

- Get all the links pointing to some page
  - This is the basis for the PageRank algorithm!
- Map function
  - 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)>

Example 5: Inverted index

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

Example 5: Inverted index



To be, or not to be

To be is to do

<to,a>,<be,a>,<or,a>,
<not,a>,<to,a>,<be,a>

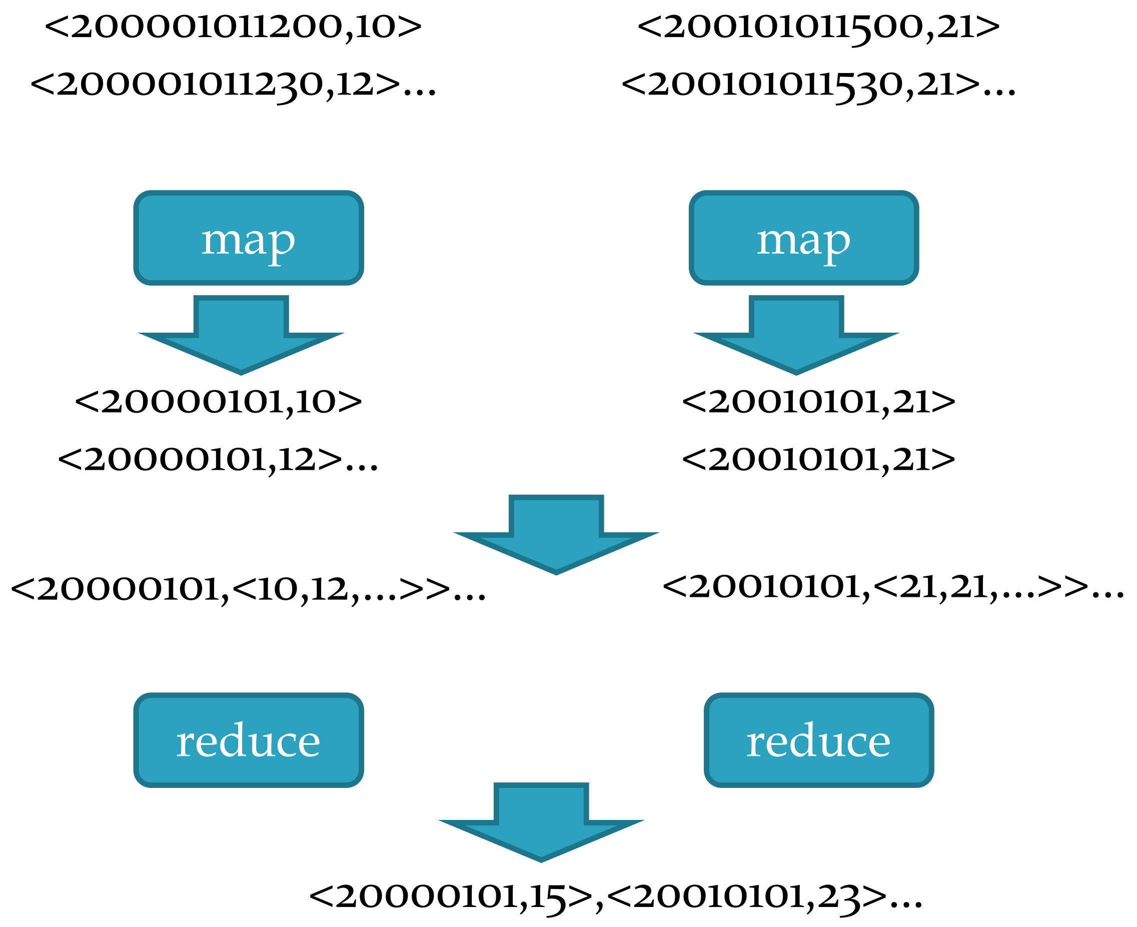
<to,b>,<be,b>,<is,b>,
<to,b>,<do,b>

<be,<a,a,b>>,<do,<b>>,
<is,<b>>

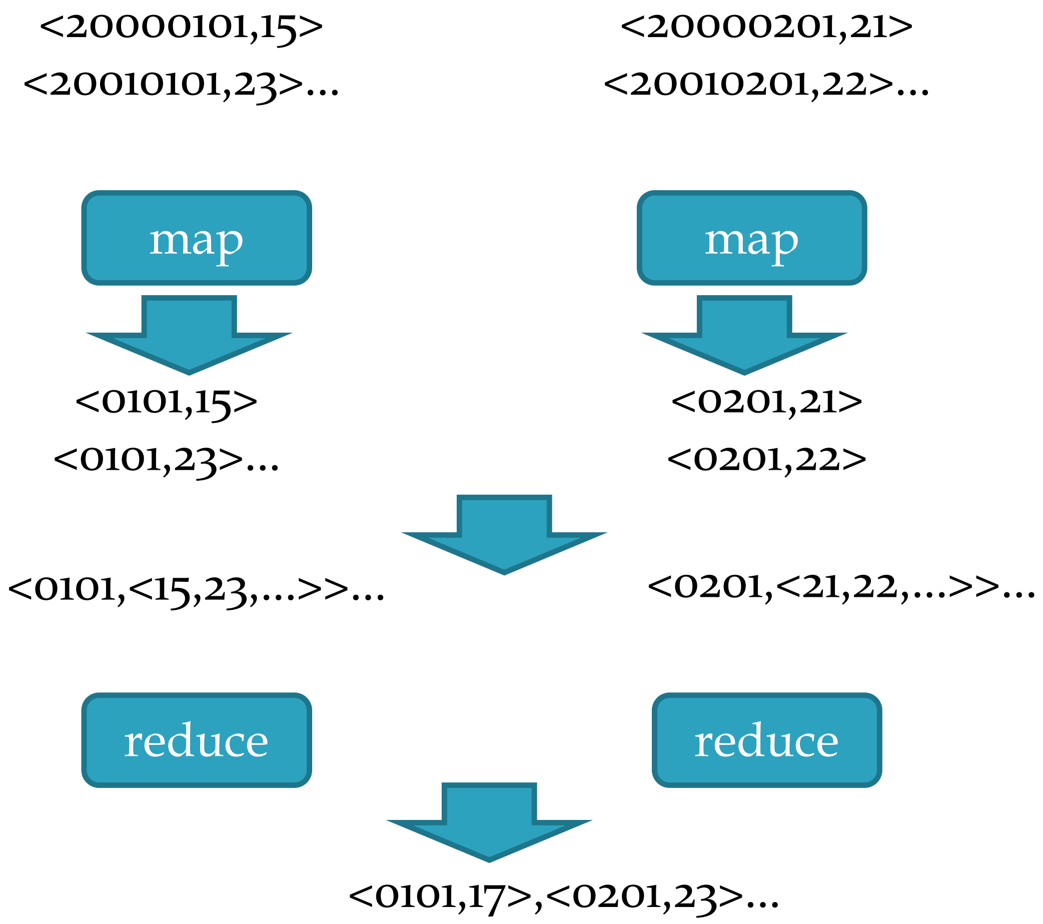
<not,<a>>,<or,<a>>,
<to,<a,a,b,b>>

<be,<a,b>>,<do,<b>>,<is,<b>>,<not,<a>>,<or,<a>>,<to,<a,b>>

Ex. 6: Avg. max temp per calendar day



Ex. 6: Avg. max temp per calendar day



MapReduce Challenges

MapReduce: (some) challenges

- Fault tolerance
- Overall performance and slow tasks
- Bandwidth costs
  - Data locality
  - Combiner functions: local pre-reduction

Fault tolerance

- Worker failure
  - The Master periodically pings each worker
  - If unresponsive, reassigns to another one
    - The map and reduce operations are stateless: restart is easy to implement
    - If transient failure of a Map or Reduce job, no problem of duplicates as the intermediate results are stored on local disk
    - Failure during the send to the global store: atomic commit protocol for results
- Master failure
  - Periodically checkpoint the state of the master to the global storage
  - If master fails, restart from last checkpoint
    - May lead to duplicate work for workers, but seldom happens

Slow tasks

- Some machines may be responsive but slow
  - e.g., if other jobs are scheduled by another MR master
  - Bottleneck for the entire job execution time

- When a complete MR is close to execution (either map or reduce phase)
  - Re-allocate the non-finished jobs on the fastest nodes
  - Let the first node that finishes commit its results to the reduce workers or to the global file system

Data locality

- The data is stored in a distributed file system
  - For Google: GFS
  - For Hadoop: HDFS (open source)
- The splits are GFS/HDFS blocks
- The master gets the location of a block from GFS and allocates the map job on the same node or on a close-by node

Bandwidth saving

- Moving the intermediate key,value pairs from map workers to reduce workers costs bandwidth
- Some reduce operations can be combined
  - e.g., counting
- Some others cannot
  - e.g., top-k computation
- If possible to combine, execute a combiner function on the same node as the map worker to pre-reduce the data
  - Often the same function as the final reduce job
  - Example: in the word-count MR job, apply the reduce job locally to all <word,1> pairs to send <word, n> pairs to the reduce job

Hadoop

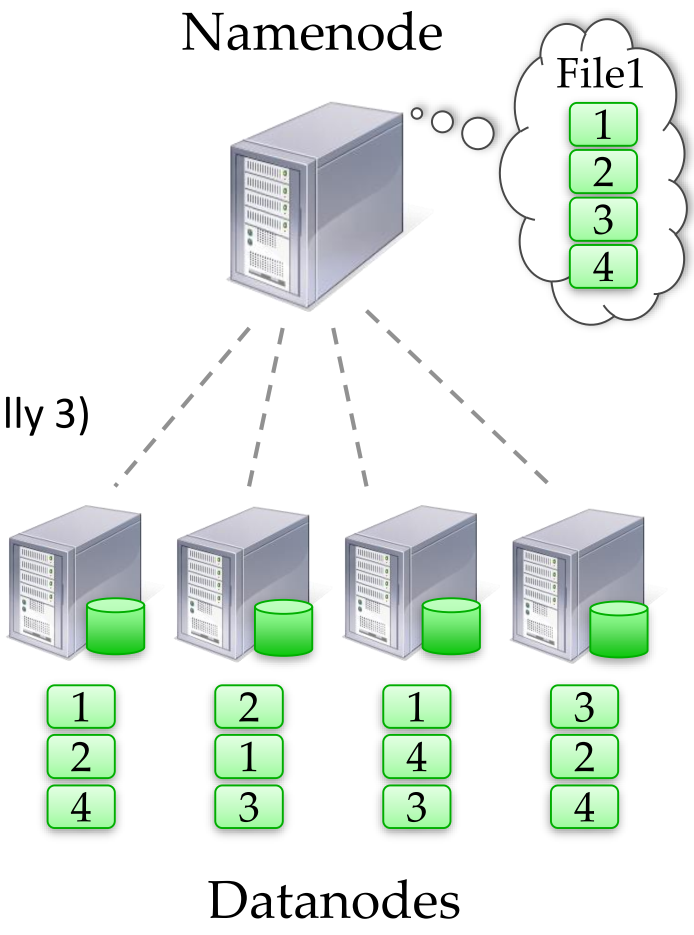
Hadoop

- Hadoop is the most known open-source MapReduce implementation
  - Lots of contributions by Yahoo!, now an Apache foundation project
  - Written in Java
  - Uses the **HDFS** file system (amongst others)
  - Many extensions and optimizations over the original Google paper

- A MapReduce implementation of choice when using Amazon's cloud services
  - EC2: rent computing power and temporary space
  - S3: rent long term storage space

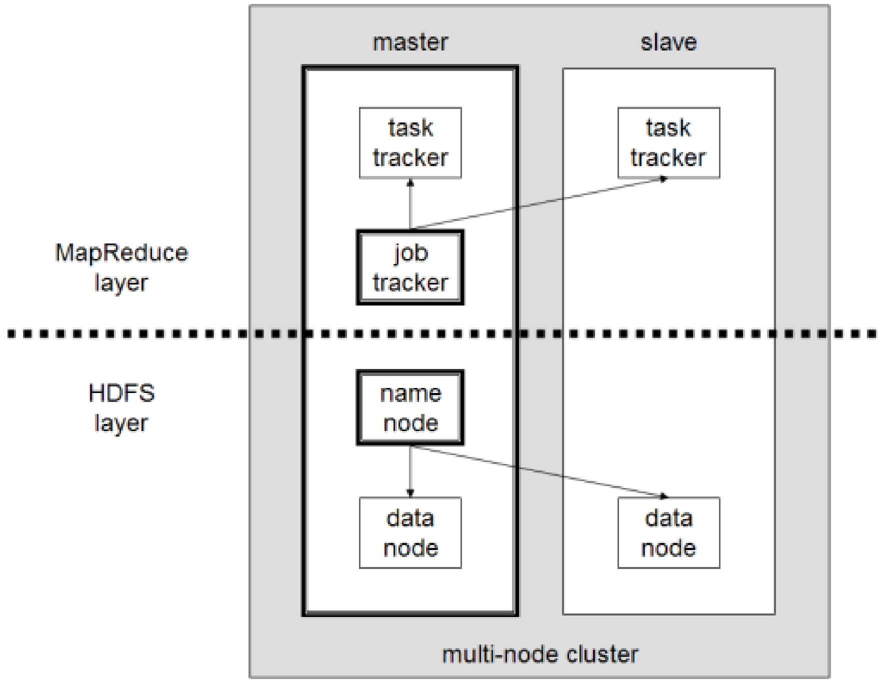
HDFS – Hadoop Distrib. File System

- A **distributed**, **scalable** file system for M-R applications
  - Distributed: Runs in a cluster
  - Scalable: 10K nodes, 100M files, 10PB storage
  - Closed-source optimizations
  - HDFS provides a single file system view to the whole cluster
- Files are split up in **blocks**
  - Typically 128MB
  - Each block is replicated on multiple **DataNodes** (typically 3)
  - Block placement is rack-aware

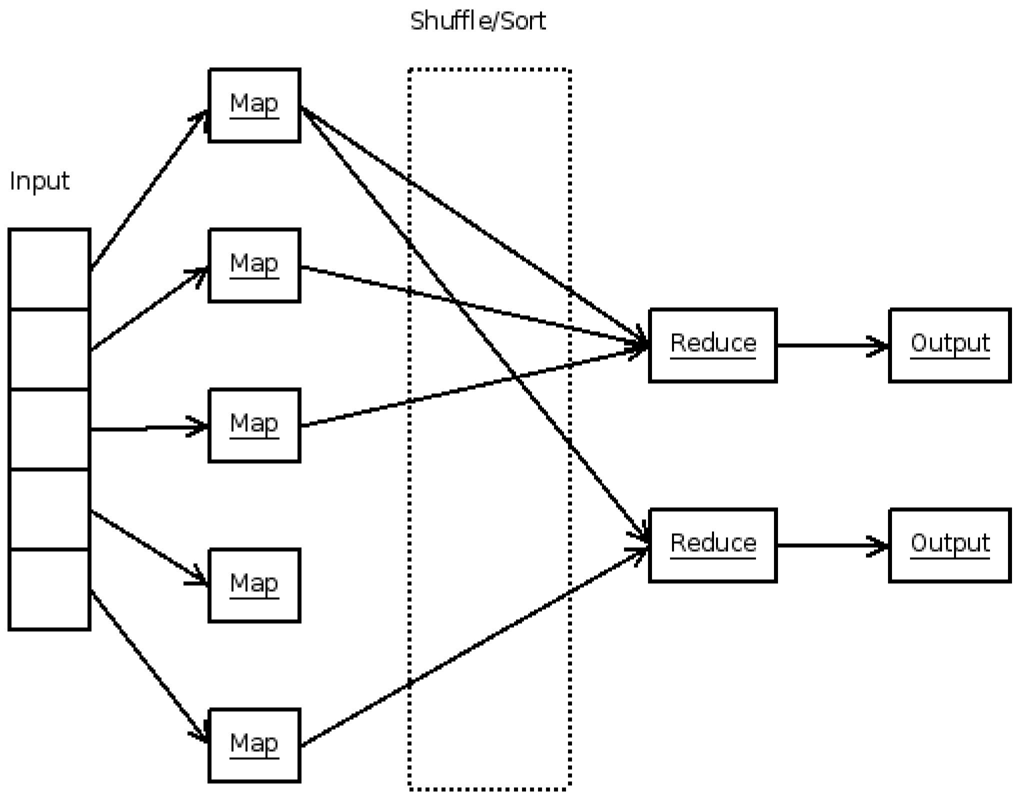


HDFS/MapReduce Architecture

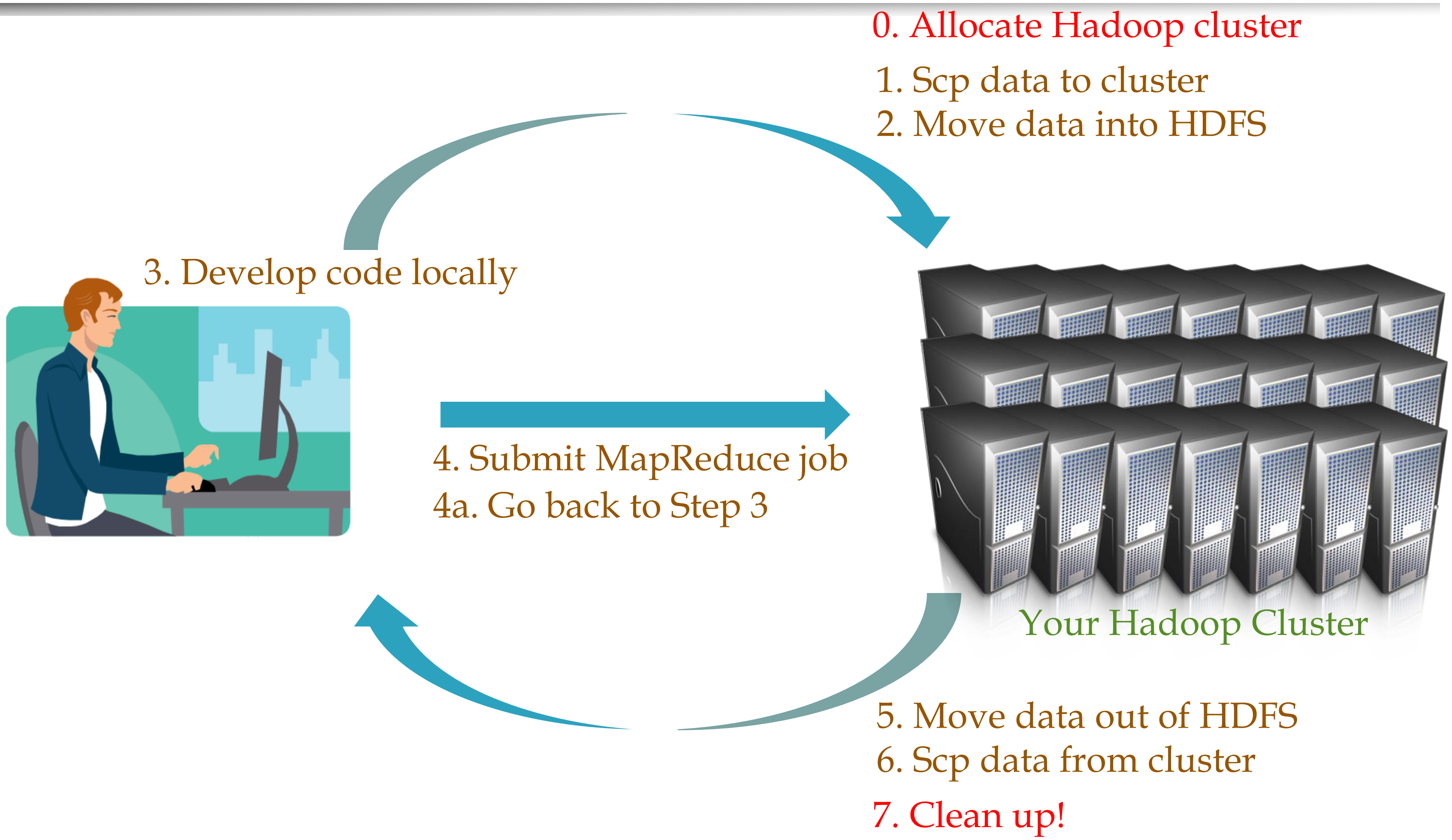
- Master/Slave Architecure
- HDFS
  - A centralized **NameNode** controls multiple **DataNodes**
  - **NameNode**: keeps track of which DataNode stores which block
  - **DataNodes**: “dumb” servers storing raw file chunks
- MapReduce
  - A centralized **JobTracker** controls multiple **TaskTrackers**
- Placement
  - **NameNode** and **JobTracker** run on the **master**
  - **DataNode** and **TaskTracker** run on **workers**
  - Data locality is exploited



MapReduce


Shuffle/Sort
Input
Map
Map
Map
Map
Map
Reduce
Reduce
Output
Output

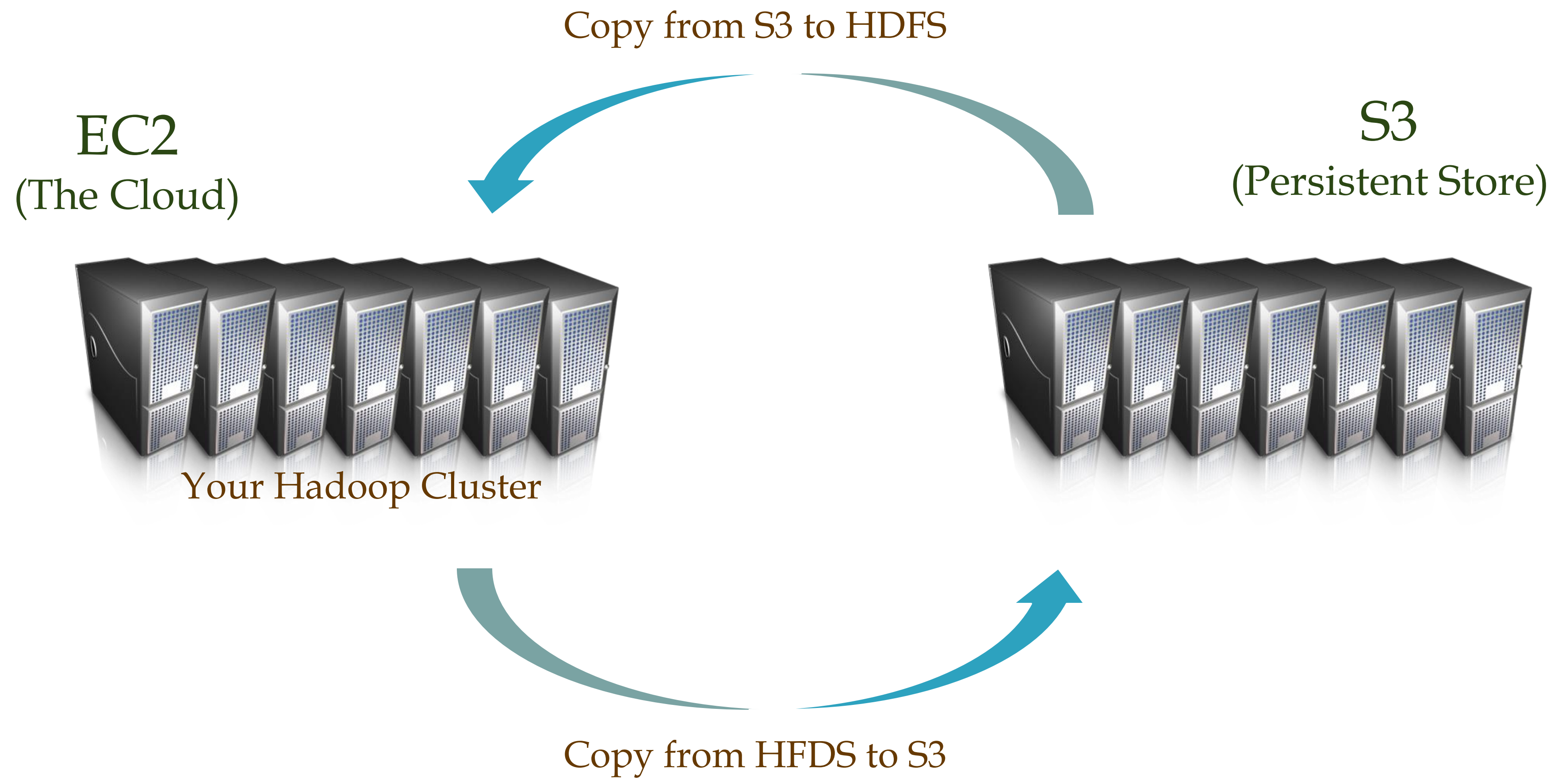
Hadoop: big picture



Uh oh. Where did the data go?

courtesy of Prof. Lin, university of Maryland, CC-BY-NC-SA (USA)

On Amazon: EC2 and S3



Hadoop Usecases

Use cases 1/3

- NY Times
  - Large Scale Image Conversions
  - 100 Amazon EC2 Instances, 4TB raw TIFF data
  - 11 Million PDF in 24 hours and 240$
- Facebook
  - Internal log processing
  - Reporting, analytics and machine learning
  - Cluster of 1110 machines, 8800 cores and 12PB raw storage
  - Open source contributors (Hive)
- Twitter
  - Store and process tweets, logs, etc
  - Open source contributors (Hadoop-lzo)

Use cases 2/3

- Yahoo
  - 100.000 CPUs in 25.000 computers
  - Content/Ads Optimization, Search index
  - Machine learning (e.g. spam filtering)
  - Open source contributors (Pig)
- Microsoft
  - Natural language search (through Powerset)
  - 400 nodes in EC2, storage in S3
  - Open source contributors (!) to HBase
- Amazon
  - ElasticMapReduce service
  - On demand elastic Hadoop clusters for the Cloud

Use cases 3/3

- AOL
  - ETL processing, statistics generation
  - Advanced algorithms for behavioral analysis and targeting
- LinkedIn
  - Used for discovering People you May Know, and for other apps
  - 3x30 node cluster, 16GB RAM and 8TB storage
- Baidu
  - Leading Chinese language search engine
  - Search log analysis, data mining
  - 300TB per week
  - 10 to 500 node clusters

Conclusion

Conclusion

- MapReduce is a powerful simplifying abstraction for programming large-scale data processing
  - Naturally suited to embarrassingly parallel jobs
    - But is not adapted to all types of jobs
      (e.g., jobs with data interdependencies)
- Master = single point of failure
- Extensions
  - Process streams of data
    (StreamMine project, StreamMapReduce)
    - Real-Time support and complex event processing
  - Decentralize the master and use a collaborative scheme
    - Build the master using a DHT and replication for fault tolerance
  - Automatic MapReduce-ization
    - Some work already on automatic MR code generation from SQL queries
      (Prof. W. Zwaenepoel @ EPFL - EuroSys 2011)