Πανεπιστήμιο Πατρών

Συναρτησιακός Προγραμματισμός για Επεξεργασία Δεδομένων Μεγάλης Κλίμακας με χρήση Map/Reduce Functions

MapReduce

Γ.Γαροφαλάκης. Σ.Σιούτας

MapReduce

Big Data Processing

- Web Crawling (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 web pages today?
  - How many users clicked on ads in the last month?
- Sorting of 100 Pbytes numbers…
- MIN, MAX, SUM, AVG of 100 Pbytes numbers…
- Range Queries of 100 Pbytes numbers…

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

The example of LISP

- Lisp ≠ **L**ost **I**n **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
```

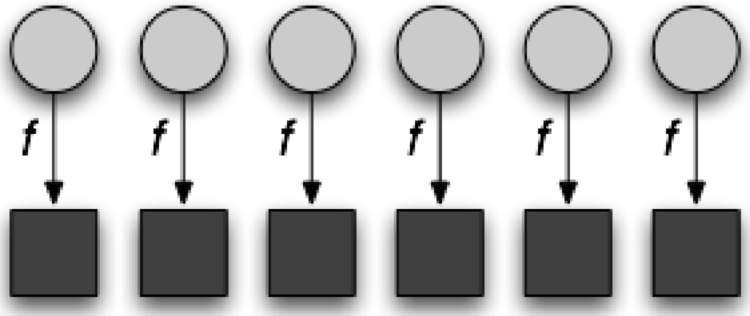
Lisp → MapReduce

- 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

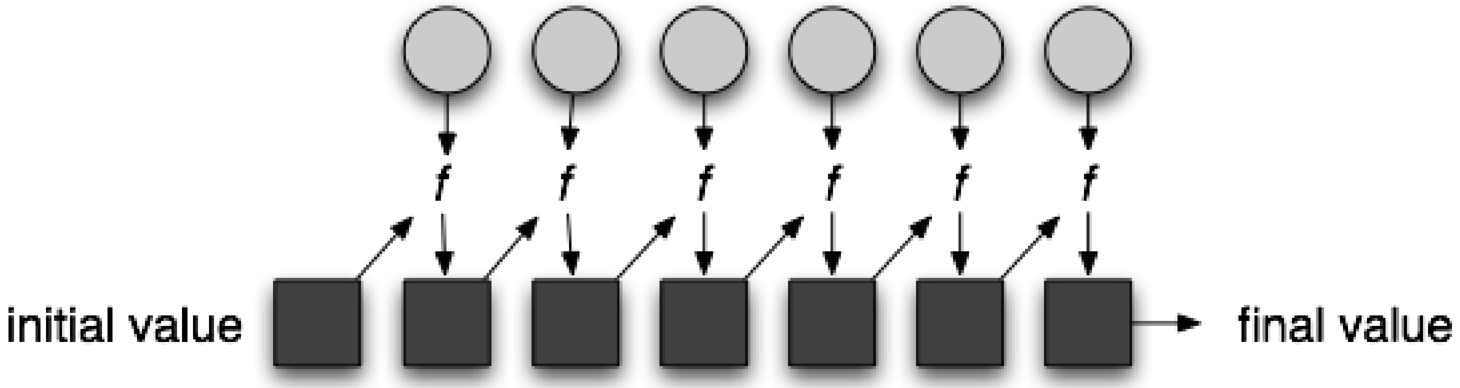
- Map is a higher-order function
- How map works:
  - Function is applied to every element in a list
  - Result is a new list
- Note that each operation is independent and, due to referential transparency (no side effects of functions evaluation), applying *f* on one element and re-applying it again 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

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.

MapReduce Programming Model

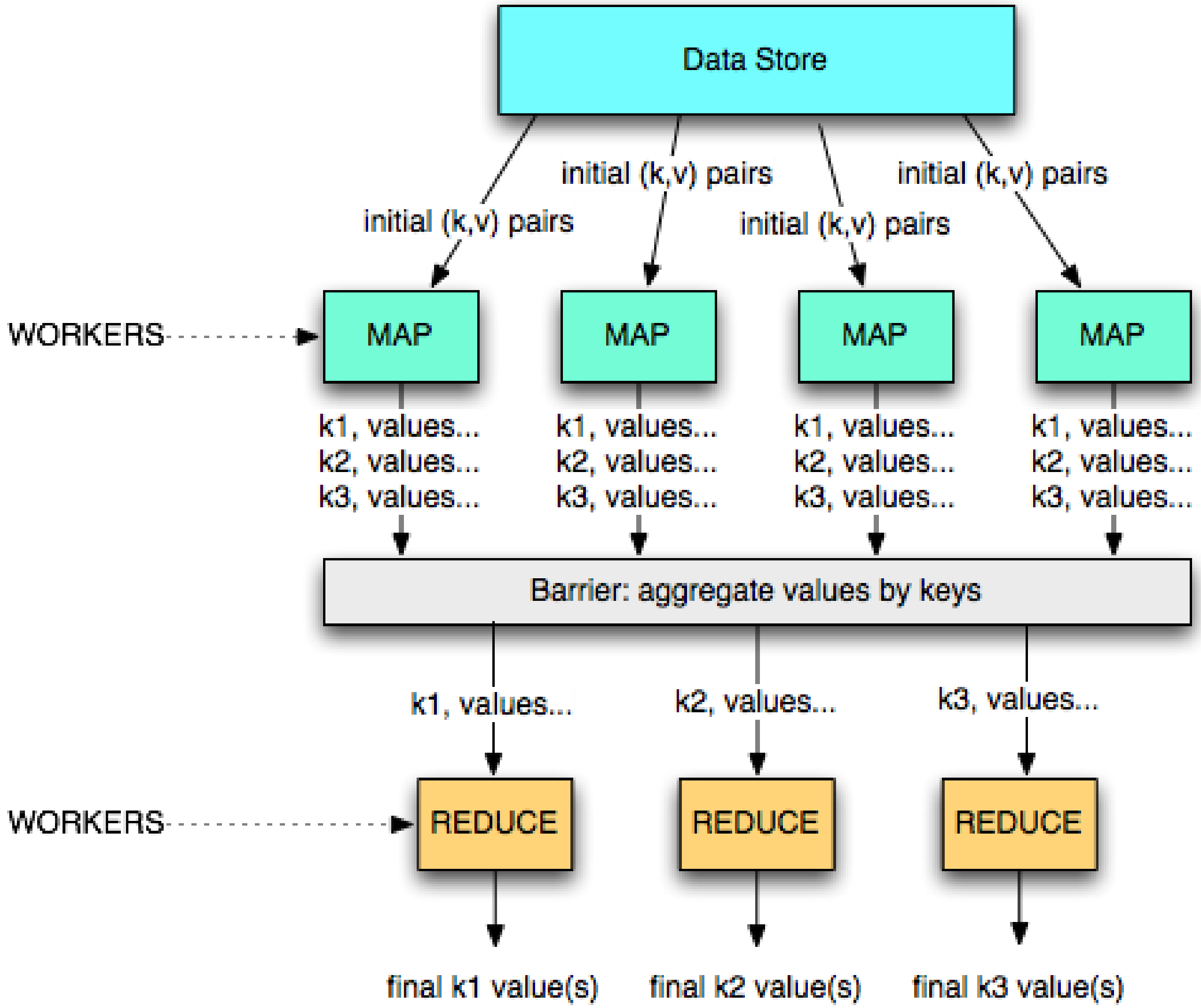
- Data type: key-value *records*
- Map function:

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

- Reduce function:

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

A divide and conquer approach



MapReduce Examples

Example 1: Word Count

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


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

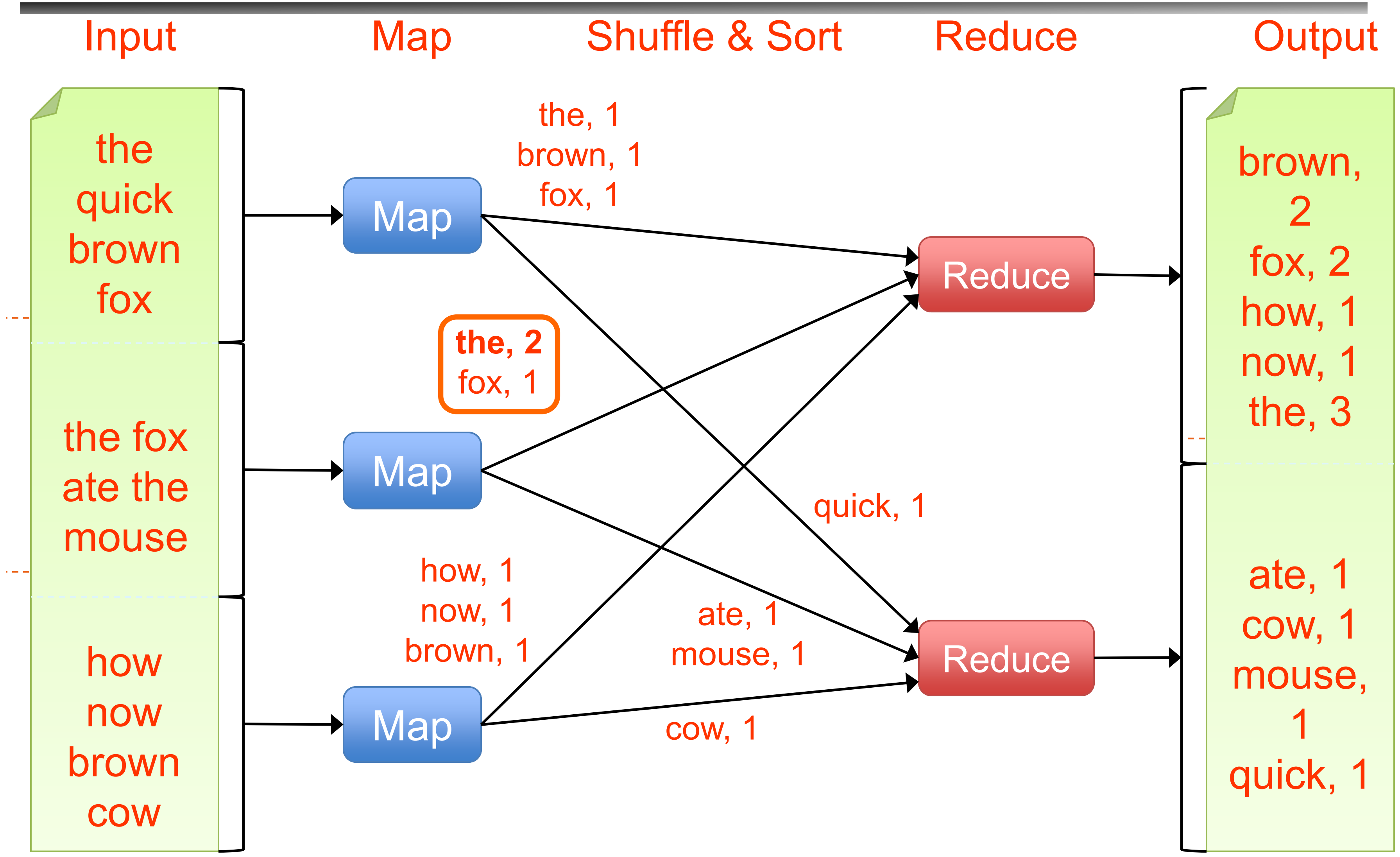
An Optimization: The Combiner

- 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



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)
    - (file_id, line_number)

- Reduce function
  - identity (use intermediate results as final results)
  - (file_id, list (line_number))

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)>

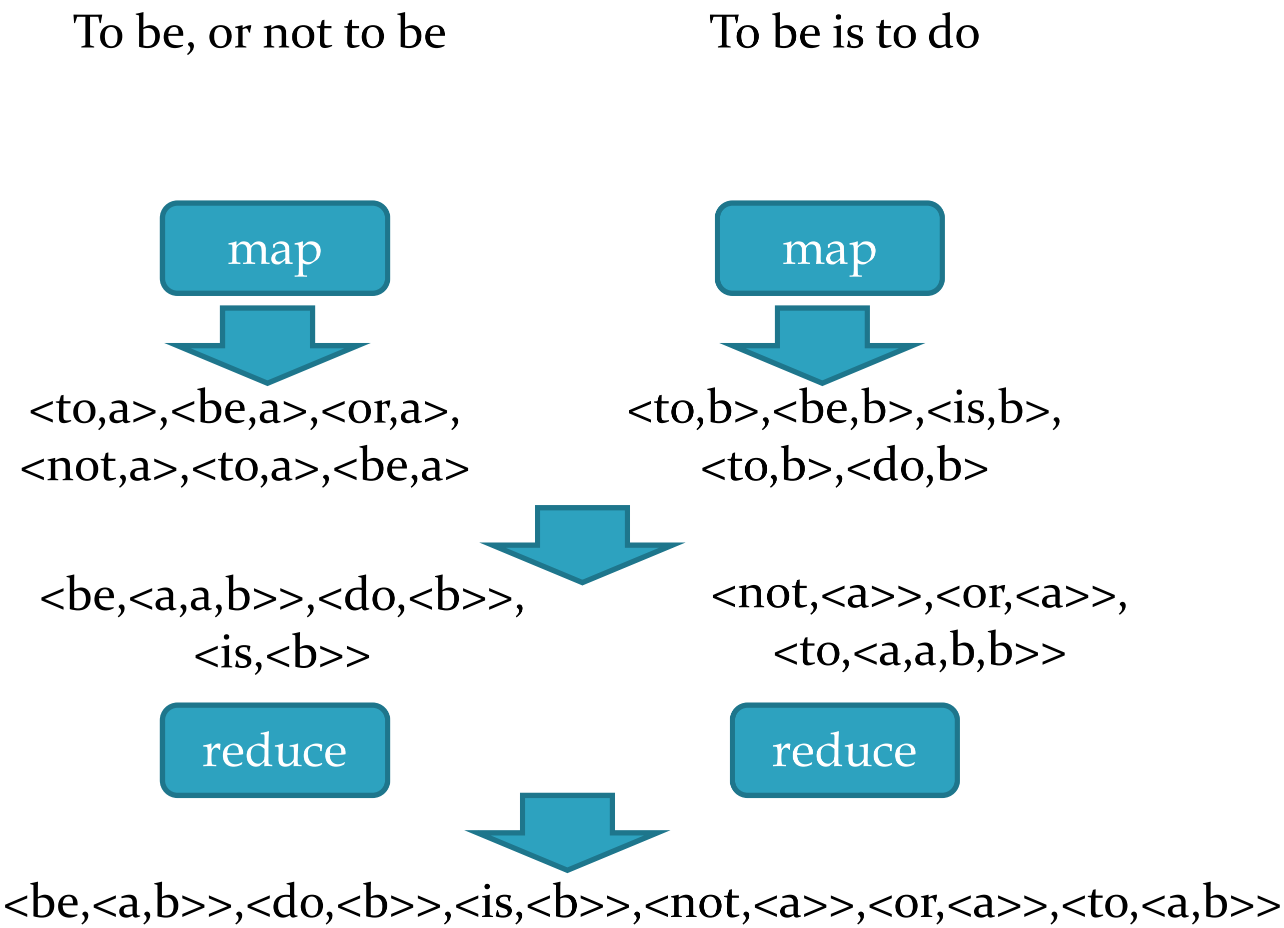
Example 5: Inverted index

- Get all documents containing some particular keyword
  - Used by the search mechanisms of Google, Yahoo!, etc.
  - Second input for PageRank

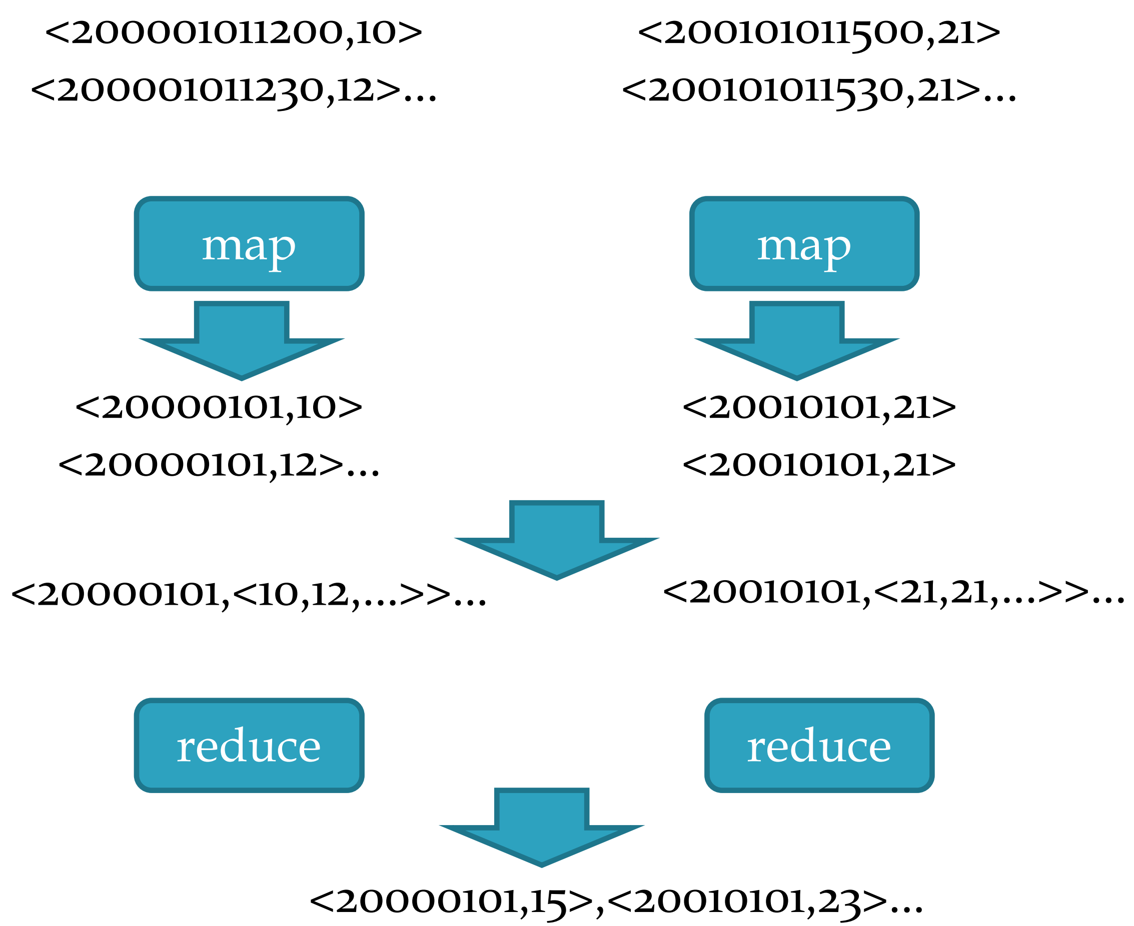
- Map function
  - Parse each document and emit a set of pairs <word, documentID>

- Reduce function
  - Take all pairs for a given word
  - Sort the document IDs
  - Emit a final <word,list(document IDs)> pair

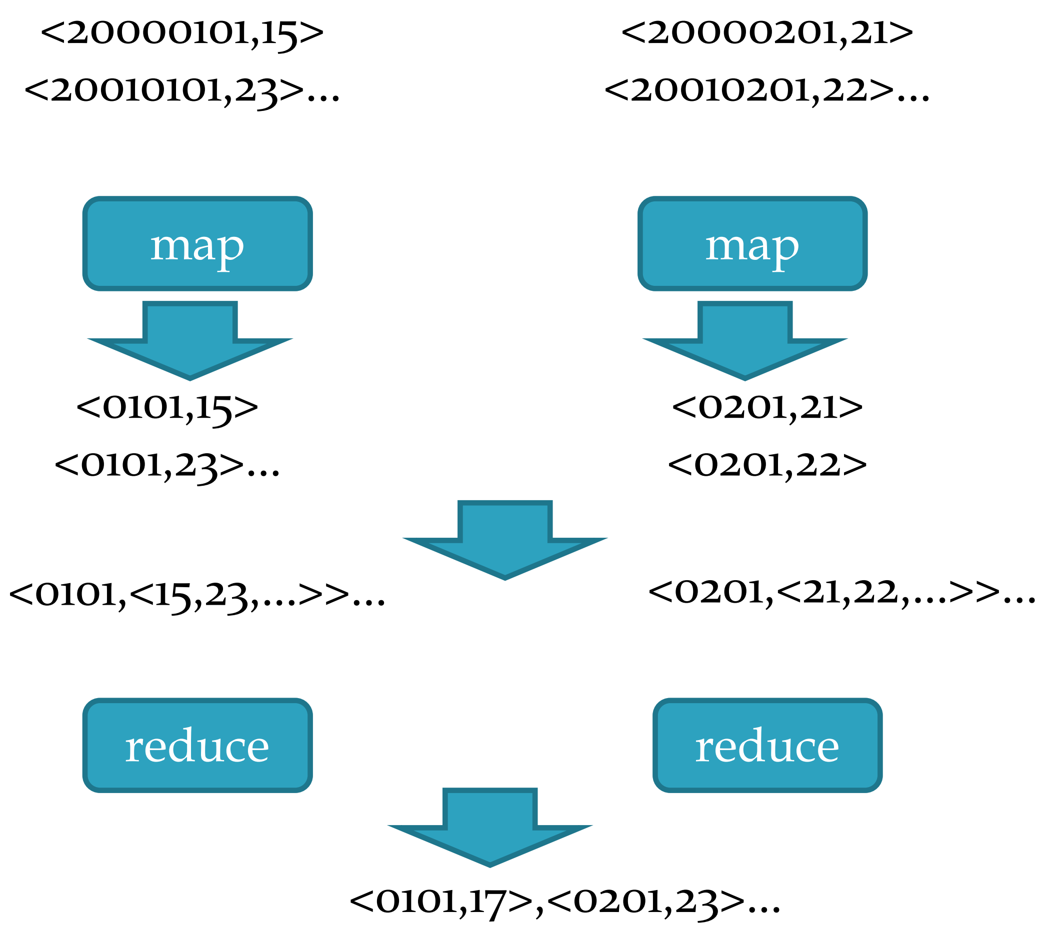
Example 5: Inverted index



Ex. 6: Avg. max temp per calendar day



Ex. 6: Avg. max temp per calendar day



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

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