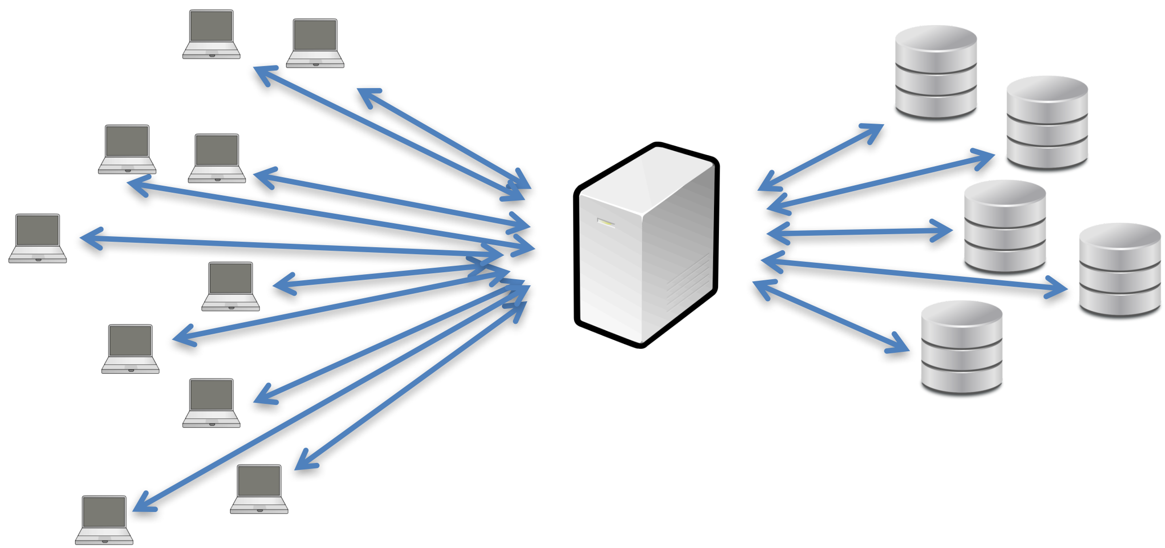
COMP207
Database Development

Lecture 19

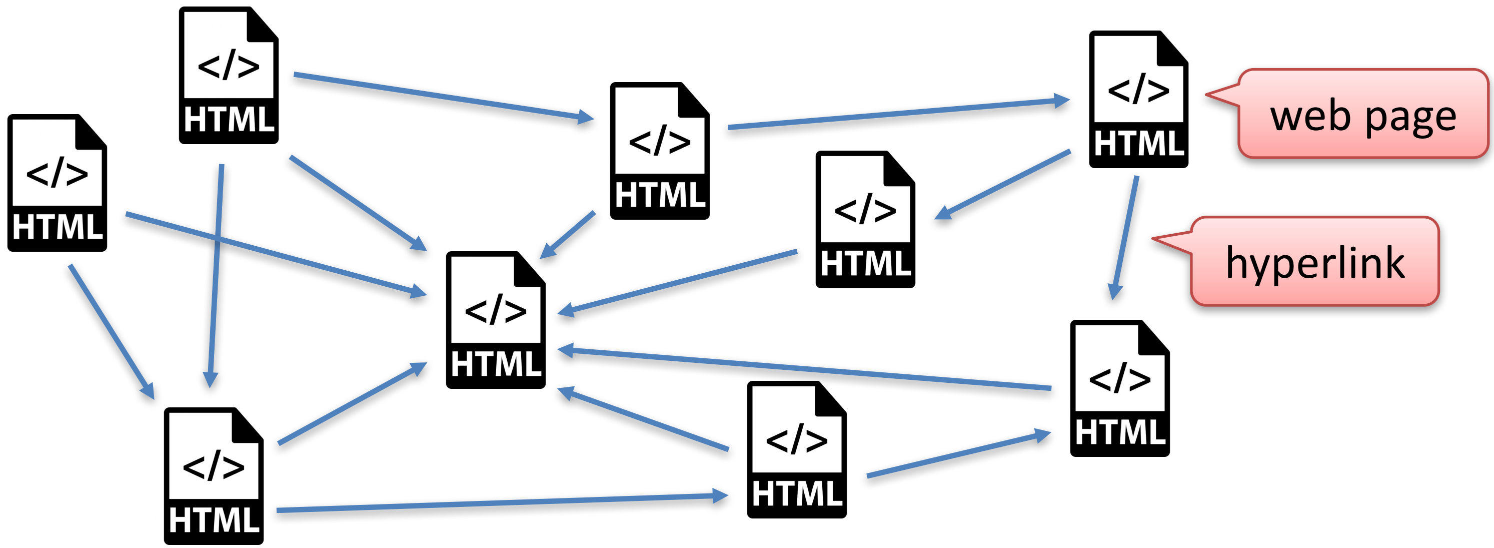
Digression: The MapReduce Framework

Relational Databases are Powerful



- Allow us to store, manage, and analyse large datasets
- Some data analysis applications go beyond what such systems can handle…

Importance of Web Pages



- Graph with all relevant HTML pages on the web
- Task: rank pages by "importance"
  - Basis of Google Search
  - ≈ an iterated matrix product (number of rows = number of pages)

Problem: millions to billions of web pages

Finding Patterns in Large Datasets

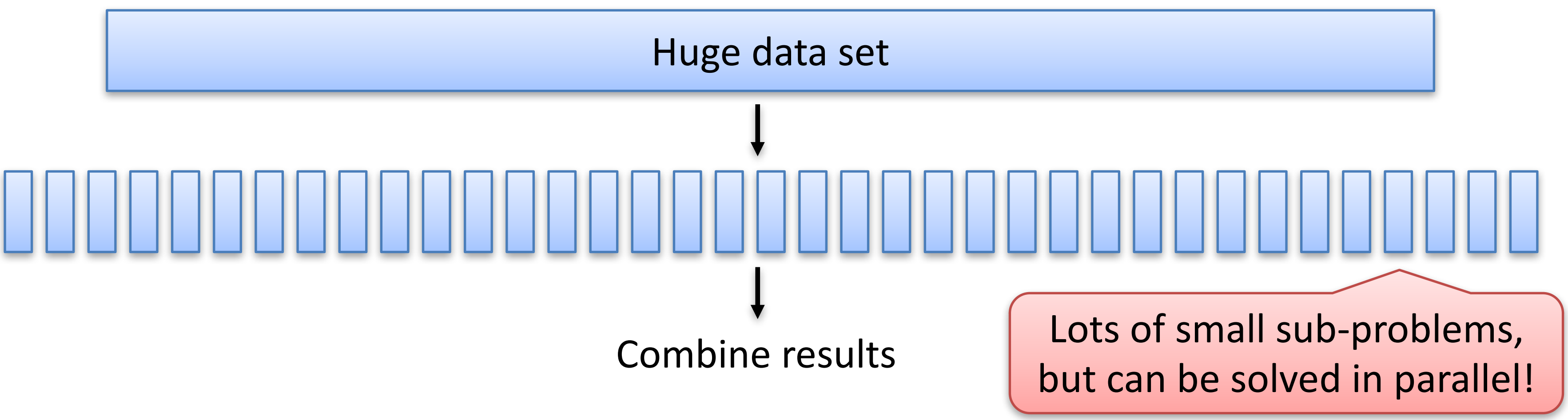
"Big Data"

- Predictive maintenance, e.g.
  - Which parts of a system are likely to fail or cause certain effects?
  - Learn from sensor data, infrastructure data, etc.
- Detect interesting patterns, e.g.
  - Interesting phenomena in space, on earth, under water, in stock markets, etc.
  - Recognise plants with certain diseases
  - Estimate the likelihood of archeological artifacts at a certain site based on archeological data, geological parameters, etc.
- Understand the structure/dynamics of large networks, e.g.
  - Train networks
  - Social networks

In these applications, the data is potentially huge.

Divide and Conquer

- Often such problems can be solved as follows:
  - Divide data into smaller chunks
  - Perform computations on smaller chunks
  - Combine results



- Idea behind MapReduce!

MapReduce

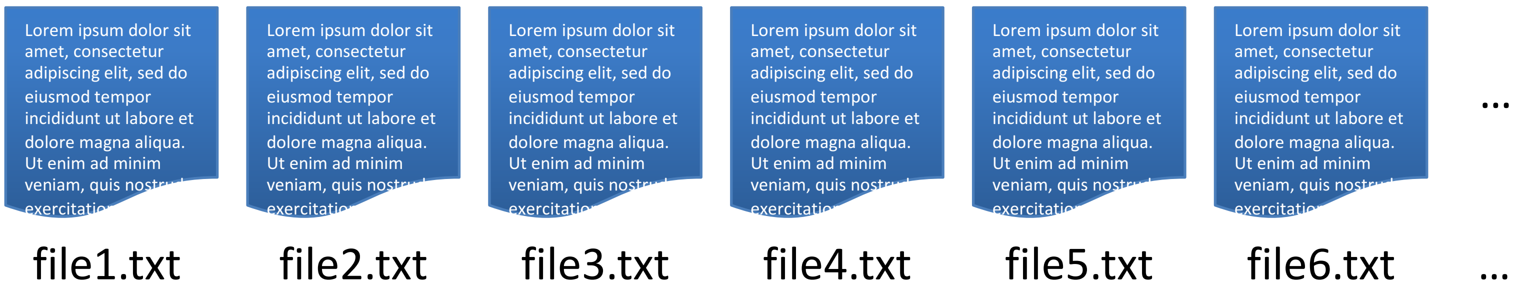
- Programming framework
  - Provides simple means to create programs that can scale to terabytes of data
- Users only implement two methods
  - **Map**: the computation on the smaller chunks of data
  - **Reduce**: how the results are combined to the final result
  - MapReduces manages the rest!
- Implementations:
  - MapReduce: Google’s original implementation
  - Apache Hadoop: open-source

Counting Words in Documents

or: Hello World!

The Task

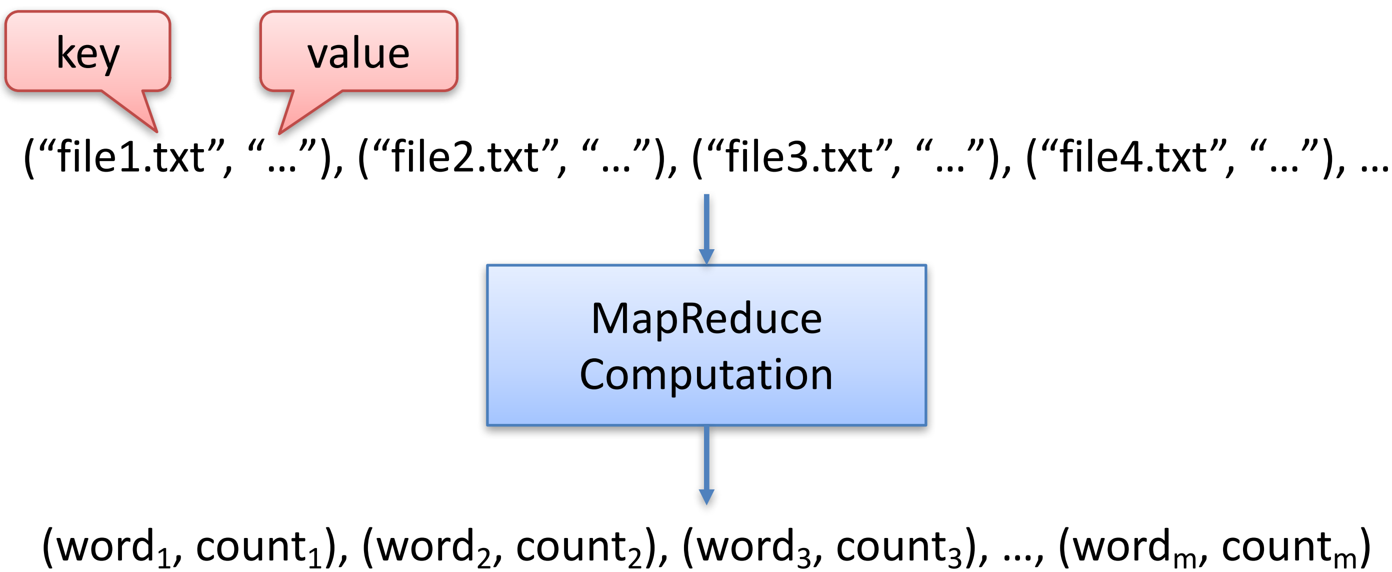
- Given: a collection of “not too large” text files



- Task: for each word **w**, count how often **w** occurs in these files

How To Do This In MapReduce?

- The MapReduce computation:



- To implement this computation, we have to write two functions: **Map** & **Reduce**

The Map Function

- The Map function is applied to a single key/value pair and produces a list of zero or more key/value pairs

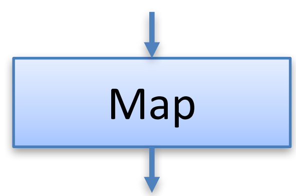
Map(String **filepath**, String **contents**):

  for each word **w** in **contents**:

    output pair (w, “1”)

- Example:

(“file1.txt”, “Hello World! Hello UK! Hello Liverpool!”)



(“Hello”, “1”), (“World!”, “1”), (“Hello”, “1”), (“UK!”, “1”), (“Hello”, “1”), (“Liverpool!”, “1”)

Grouping

- After the Map functions have been applied, we group all values by key

("Hello", 1), ("World!", 1), ("Hello", 1), ("UK!", 1), ("Hello", 1), ("Liverpool!", 1)

("Hello", (1, 1, 1)), ("World!", (1)), ("UK!", (1)), ("Liverpool!", (1))

- The list of values for a key is the input to Reduce

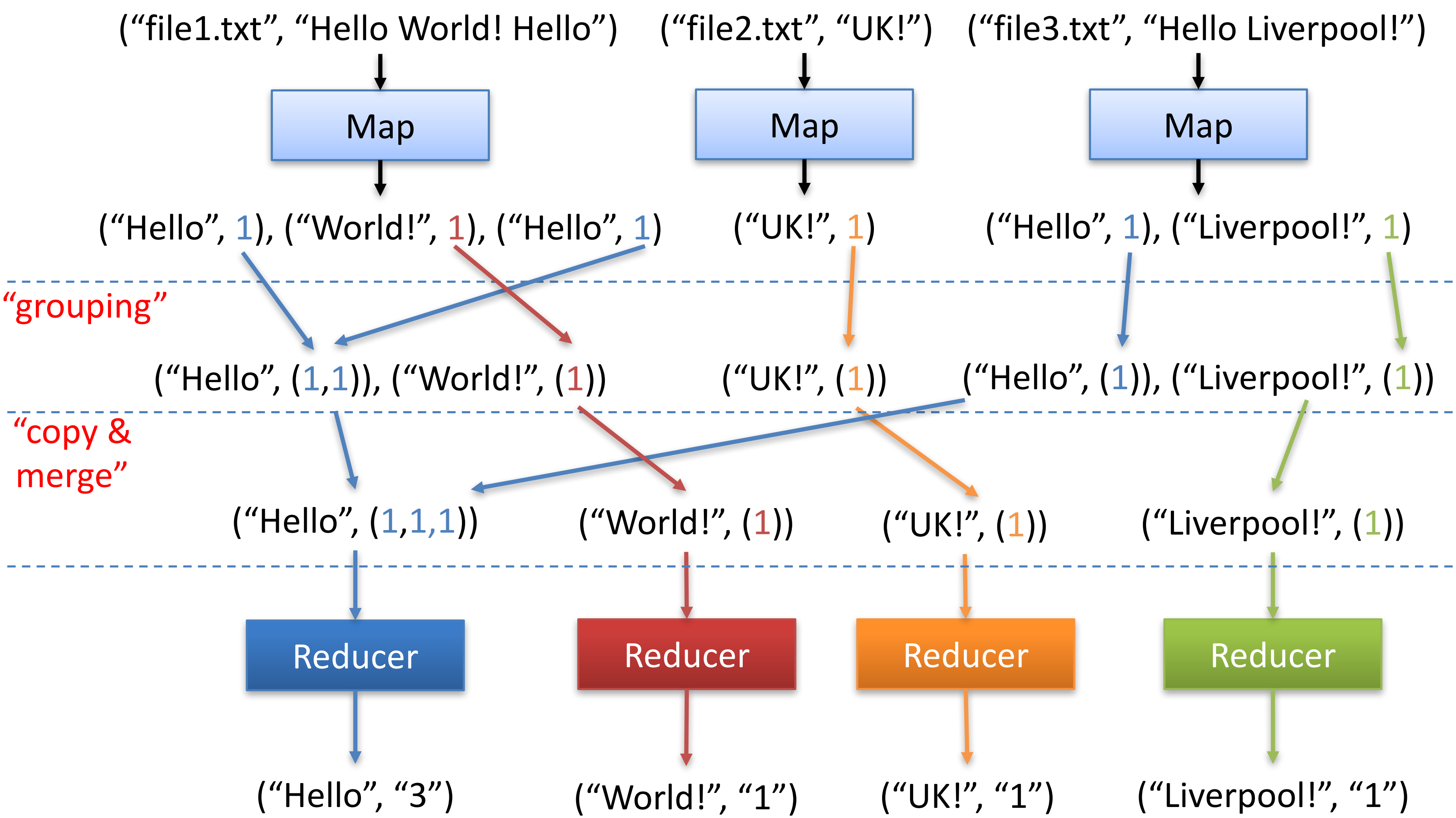
The Reduce Function

- The Reduce function takes a key and a list of values as input and outputs a list of key/value pairs

```
Reduce(String word, Iterator<String> values):
    int count = 0
    for each v in values:
        count = count + parseInt(v)
    output pair (word, toString(count))
```

- Example:
  – Reduce(“Hello”, (“1”,”1”,”1”)) outputs (“Hello”, “3”)

Putting Things Together



The Beauty of It…

- We only had to implement these two functions…

```
Map(String filepath, String contents):
    for each word w in contents:
        output pair (w, "1")
```

```
Reduce(String word, Iterator<String> values):
    int count = 0
    for each v in values:
        count = count + parseInt(v)
    output pair (word, toString(count))
```

- MapReduce takes care of everything else:
  - Parallel execution, including allocation of Map/Reduce to machines
  - Failure

Matrix multiplication

- Matrix multiplication: Given (r,s)- and (s,t)-matrix M and N resp., compute P s.t.

$$P_{i,k} = \sum_{j=1}^{s} M_{i,j} N_{j,k}$$

- (P can be computed in O($n^{2.373}$) for $n = r = s = t$)
- Easiest (also faster) for MapReduce:
  use 2 MapReduce computations

First MapReduce

- matrix $\in \{N, M\}$ and triplet $\in \mathbb{Z}_+ \times \mathbb{Z}_+ \times \mathbb{R}$

Map(String **matrix**, String **triplet**):
    let **(i,j,v)** = **triplet**
    if **matrix** = *N* then output pair **(i, (N,j,v))**
    else output pair **(j, (M,i,v))**

If $M_{i,j} = v$ then **(M,"(i,j,v)")** is an input, similar for N

Reduce(int **no**, Iterator<String> **values**):
    for each **(i,j,k)** in **values**:
        if **i**=M then:
            for each **(i',j',k')** in **values**:
                if **i'**=N then:
                    output pair **((j,j'),$\boldsymbol{k} \times \boldsymbol{k}'$)**

Creates the pair ((i,k), $M_{i,j} N_{j,k}$)

Second MapReduce

Map(pair<int,int> **pair**, double **no**):

  output pair **(pair,no)**

Reduce(pair<int,int> **pair**, Iterator<double> **nos**):

  double **result**=0

  for each **no** in **nos**:

    result=**result** + **no**

  output pair **(pair,result)**

Relational algebra

- Use MapReduce to implement relational algebra
- Tuples in R are like $(R, att_1 = val_1, att_2 = val_2, \ldots)$
- Input is **(t,t)** where **t** is some tuple
- Two examples:
  - Selection $\sigma_c(R)$
  - Natural join $R \bowtie S$

Selection $\sigma_c(R)$

Map(String **tuple**, String **tupleCopy**):
  if **tuple** satisfies **c** then output pair (**tuple**, **tupleCopy**)

Reduce(String **tuple**, Iterator<String> **tupleCopies**):
  output pair **(tuple,tuple)**

- Similar for everything but joins (and intersection and difference): Map does all the work

Natural join $R \bowtie S$

- Scheme: R(A,B) and S(B,C) – A,B,C are sets of attributes

Map(String **tuple**, String **tupleCopy**):

  Let **Co** be the attribute and value pairs in **B**
  output pair (**Co**, **tuple**)

Reduce(String **Co**, Iterator<String> **tuples**):

  for each **r** in **tuples**:

    if **r.relation** = R then:

      for each **s** in **tuples**:

        if **s.relation** = S then:

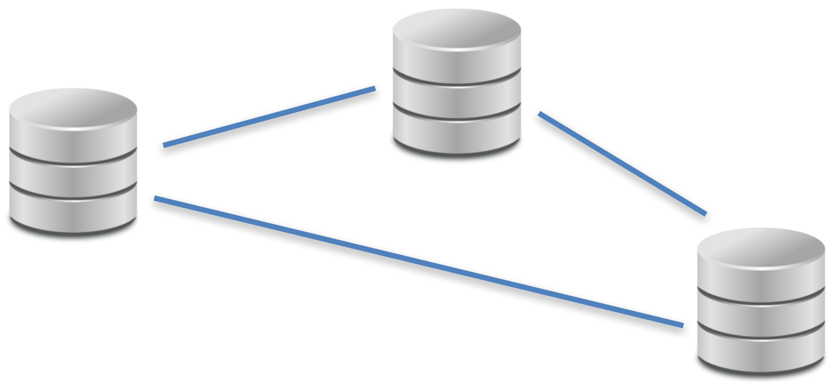
          output pair $(\boldsymbol{r \bowtie s}, \boldsymbol{r \bowtie s})$

- Similar for other joinrs

MapReduce: Closer Look

Distributed File System

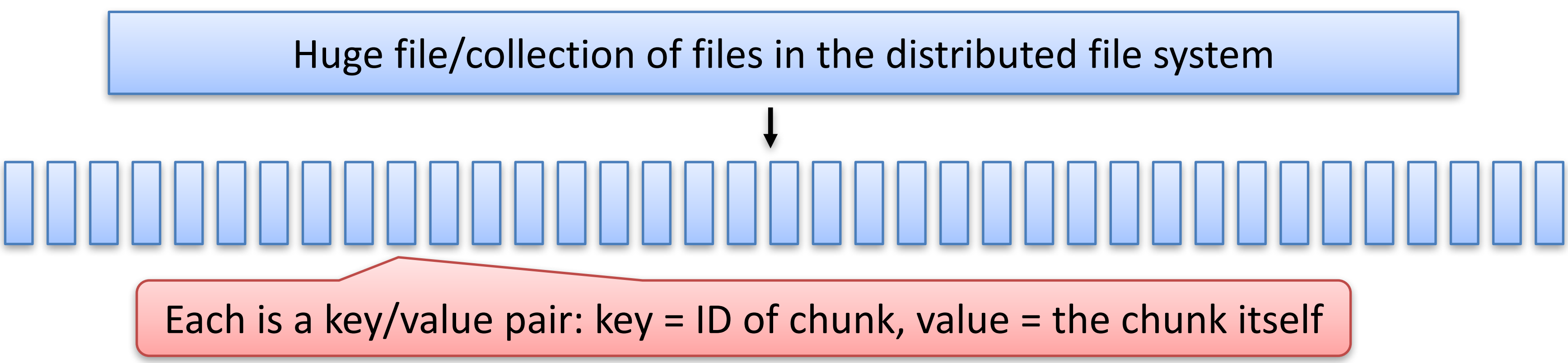
- MapReduce operates on a distributed file system



- Similar to distributed DBMS
  - All the data is distributed over the nodes/sites
  - Replication is used for fault-tolerance
- Appears to the user as if it was a regular file system
- Implementations: Google File System, Hadoop File System

Inputs To MapReduce Programs

- Input files to MapReduce programs can be very large (GBs/TBs in size)
  - Collections of web pages, links in a social network, Twitter feeds, stock market data, …
  - Satellite images, data from scientific experiments, …
- MapReduce splits these into chunks (~16-64 MB) and provides an ID for each chunk (e.g., the filename)



MapReduce Computations

$(key_1, value_1), (key_2, value_2), (key_3, value_3), \ldots, (key_n, value_n)$

↓

MapReduce Computation

↓

$(key'_1, value'_1), (key'_2, value'_2), (key'_3, value'_3), \ldots, (key'_m, value'_m)$

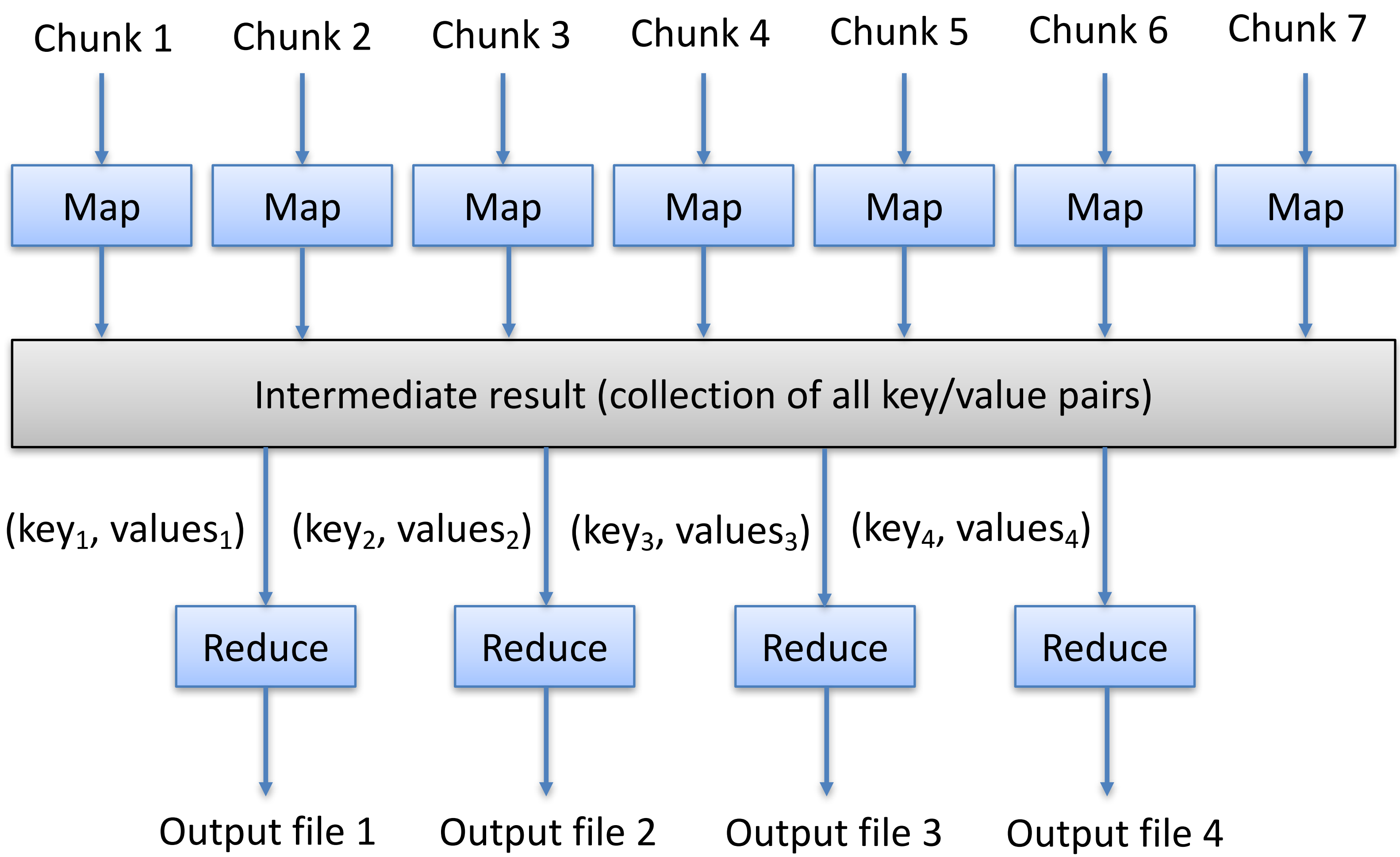
Not keys in the database sense!

- Keys/values can be arbitrary objects
  - Keys are typically small (e.g., integers, strings, etc.)
  - Values might be larger (e.g., entire text documents, images, etc.)
- Each MapReduce computation is expressed in terms of two functions: **Map** & **Reduce**

Implementation

- **Map**(String key, String value):
  - Returns a list of key/value pairs
- **Reduce**(String key, Iterator<String> values):
  - Returns a list of key/value pairs
- In practice also some additional code to set:
  - Locations of input and output files
  - Tuning parameters (e.g., number of machines, memory per Map/Reduce task, etc.)

Execution


Chunk 1
Chunk 2
Chunk 3
Chunk 4
Chunk 5
Chunk 6
Chunk 7
Map
Map
Map
Map
Map
Map
Map
Intermediate result (collection of all key/value pairs)
(key_1, $values_1$)
(key_2, $values_2$)
(key_3, $values_3$)
(key_4, $values_4$)
Reduce
Reduce
Reduce
Reduce
Output file 1
Output file 2
Output file 3
Output file 4

Expressiveness

- MapReduce can be used to compute a number of interesting functions on large datasets
  – Inverted index: for each word, return the list of all documents that contain it
  – Operators of relational algebra
  – Matrix multiplication → PageRank

- MapReduce is not a universal parallelism framework
  – Not every problem that is parallelisable can be expressed and solved nicely in MapReduce

- Bottle-neck: communication & disk access

Summary

- MapReduce is a framework for solving problems on large volumes of data
  - Simple implementation: Map & Reduce
  - MapReduce takes care of the rest
- Many interesting problems can be solved using MapReduce
- Big Data tools are based on MapReduce

Thanks for listening!

... this will not be covered in the exam.