More indepth look at MapReduce

NOT REQUIRED FOR THE EXAM!

Overview over this video

This video will show some more examples of MapReduce and then go in to more details with the framework

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))**

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

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 **(S,x,k)** in **values**:
        if **S**=M then:
            for each **(S',x',k')** in **values**:
                if **S'**=N then:
                    output pair **((x,x'),$k \times k'$)**

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 **(S,x,k)** in **values**:
        if **S**=M then:
            for each **(S',x',k')** in **values**:
                if **S'**=N then:
                    output pair **((x,x'),$k \times k'$)**

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

Second MapReduce

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

output pair **(pair,no)**

Second MapReduce

```
Map(pair<int,int> pair, double no):
    output pair (pair,no)
```

```
Reduce(pair<int,int> pair, Iterator<double> numbers):
    double result=0
    for each number in numbers:
        result=result + number
    output pair (pair,result)
```

Second MapReduce

```
Map(pair<int,int> pair, double no):
    output pair (pair,no)
```

```
Reduce(pair<int,int> pair, Iterator<double> numbers):
    double result=0
    for each number in numbers:
        result=result + number
    output pair (pair,result)
```

Number: $M_{i,j}N_{j,k}$

Second MapReduce

```
Map(pair<int,int> pair, double no):
    output pair (pair,no)
```

```
Reduce(pair<int,int> pair, Iterator<double> numbers):
    double result=0
    for each number in numbers:
        result=result + number
    output pair (pair,result)
```

Number: $M_{i,j}N_{j,k}$

Result: $P_{i,k} = \sum M_{i,j}N_{j,k}$

Relational algebra

Use MapReduce to implement relational algebra

Relational algebra

Use MapReduce to implement relational algebra

Tuples in R are like $(R, att_1 = val_1, att_2 = val_2, \ldots)$

Relational algebra

Use MapReduce to implement relational algebra

Tuples in R are like $(R, att_1 = val_1, att_2 = val_2, \dots)$

- E.g. if R has schema Employees(name,birthday), tuples are like: (Employees,name = Anne, birthday = 2000-11-1)

Relational algebra

Use MapReduce to implement relational algebra

Tuples in R are like $(R, att_1 = val_1, att_2 = val_2, \ldots)$

- E.g. if R has schema Employees(name,birthday), tuples are like: (Employees,name = Anne, birthday = 2000-11-1)

Input is **(t,t)** where **t** is some tuple

Relational algebra

Use MapReduce to implement relational algebra

Tuples in R are like $(R, att_1 = val_1, att_2 = val_2, \ldots)$

- E.g. if R has schema Employees(name,birthday), tuples are like: (Employees,name = Anne, birthday = 2000-11-1)

Input is **(t,t)** where **t** is some tuple

Two examples:

- Selection $\sigma_c(R)$

Relational algebra

Use MapReduce to implement relational algebra

Tuples in R are like $(R, att_1 = val_1, att_2 = val_2, \ldots)$

- E.g. if R has schema Employees(name,birthday), tuples are like: (Employees,name = Anne, birthday = 2000-11-1)

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**, **tuple**)

Selection $\sigma_c(R)$

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

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

Selection $\sigma_c(R)$

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

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**)

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 **($r \bowtie s, r \bowtie s$)**

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

$\boldsymbol{r} \bowtie \boldsymbol{s}$ is the natural join of the tuples r and s

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

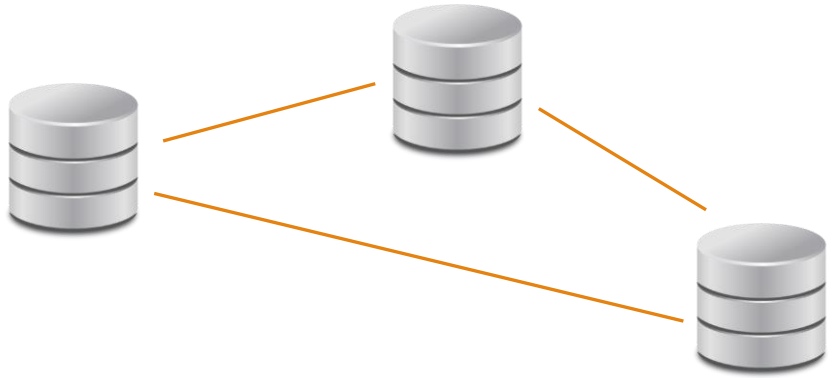
$\boldsymbol{r} \bowtie \boldsymbol{s}$ is the natural join of the tuples r and s

Similar for other joins

MapReduce: Closer Look

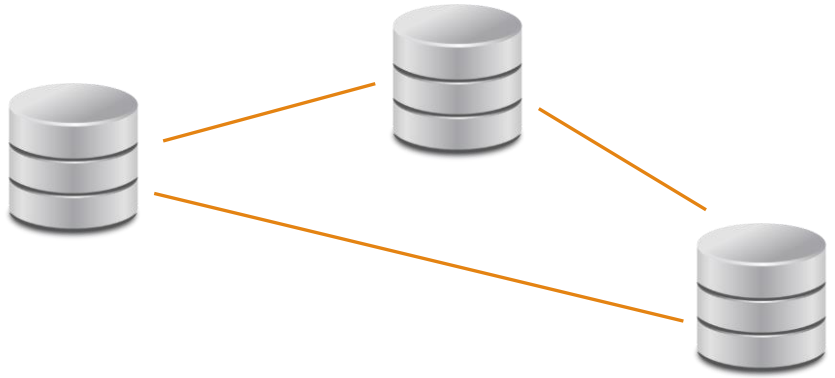
Distributed File System

MapReduce operates on a distributed file system



Distributed File System

MapReduce operates on a distributed file system

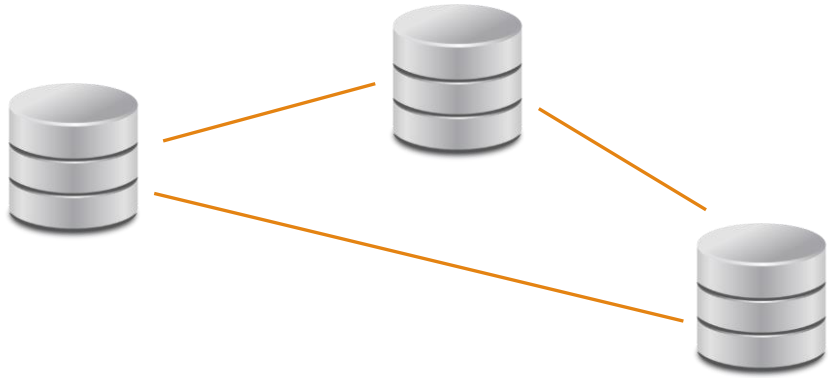


Similar to distributed DBMS

- All the data is distributed over the nodes/sites
- Replication is used for fault-tolerance

Distributed File System

MapReduce operates on a distributed file system



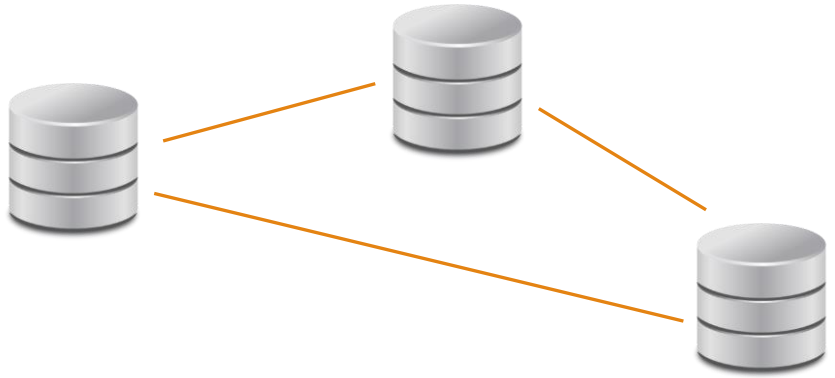
Similar to distributed DBMS

- 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

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, …

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)

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)

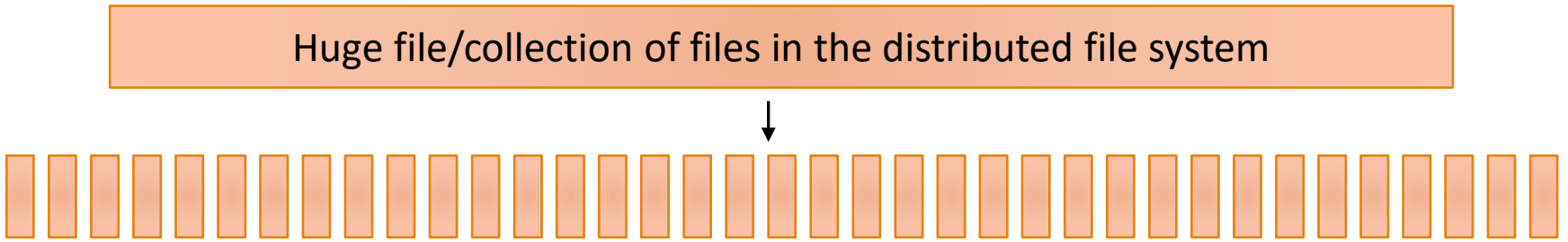
Huge file/collection of files in the distributed file system

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)

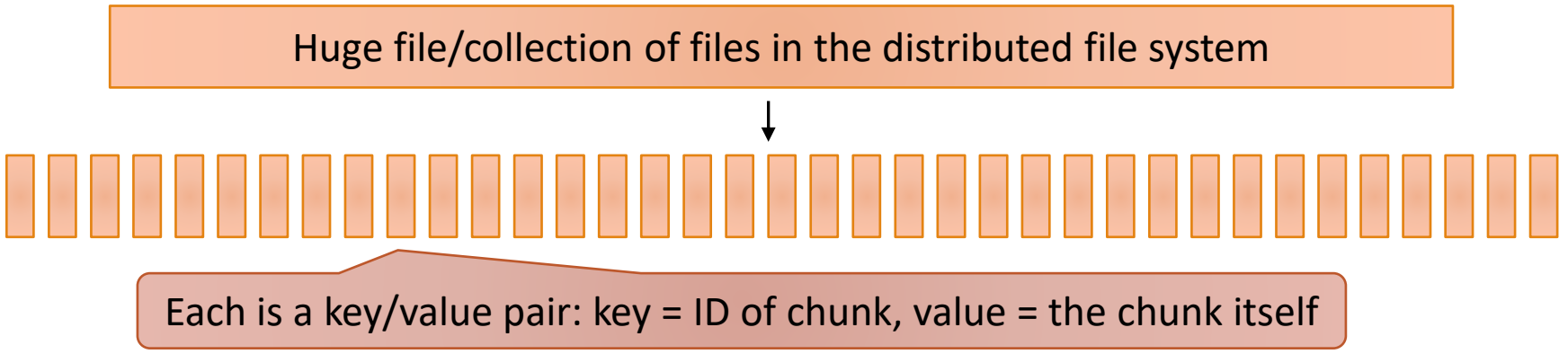


Inputs To MapReduce Programs

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 Computations

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

Not keys in the database sense!

MapReduce Computations

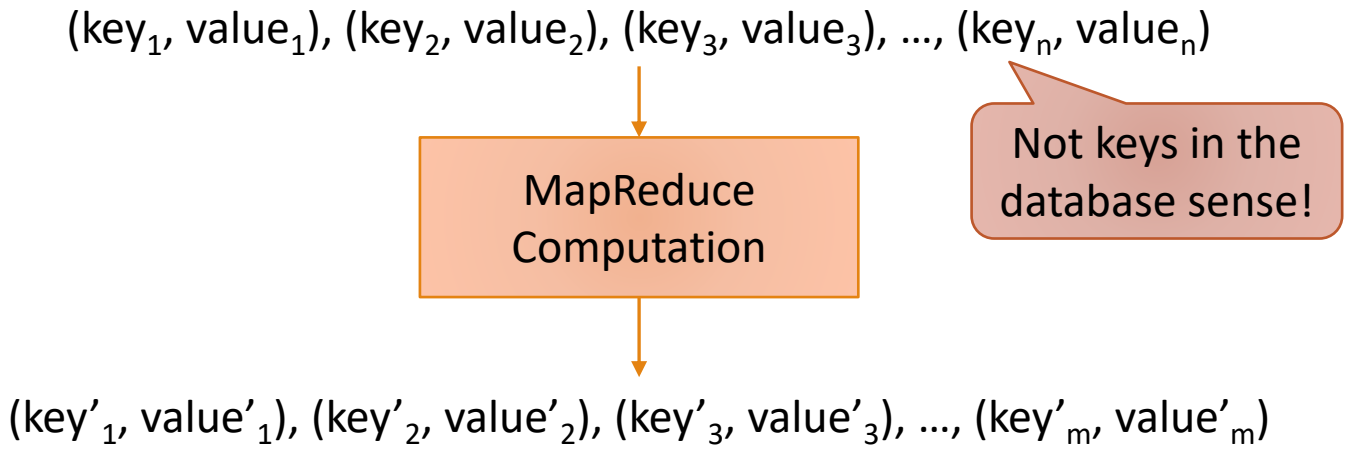
$(key_1, value_1), (key_2, value_2), (key_3, value_3), \ldots, (key_n, value_n)$

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.)

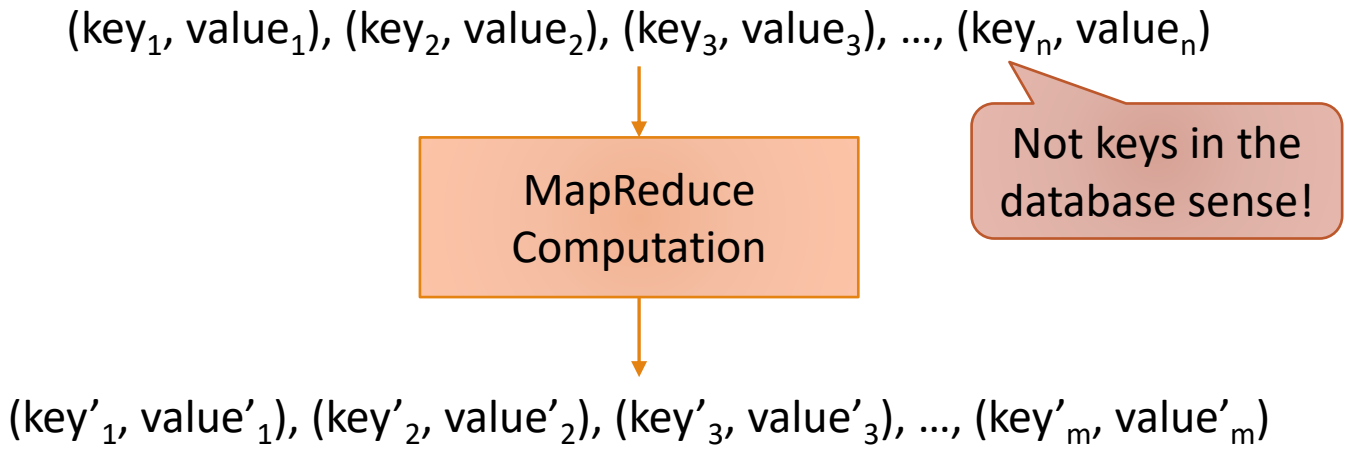
MapReduce Computations



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.)

MapReduce Computations



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

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

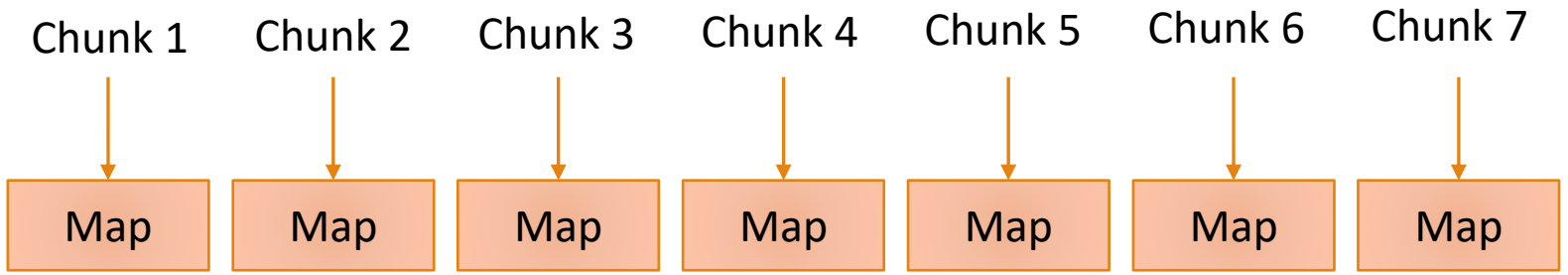
In practice also some additional code to set:

- Locations of input and output files
- Tuning parameters (e.g., number of machines, memory per Map/Reduce task, etc.)

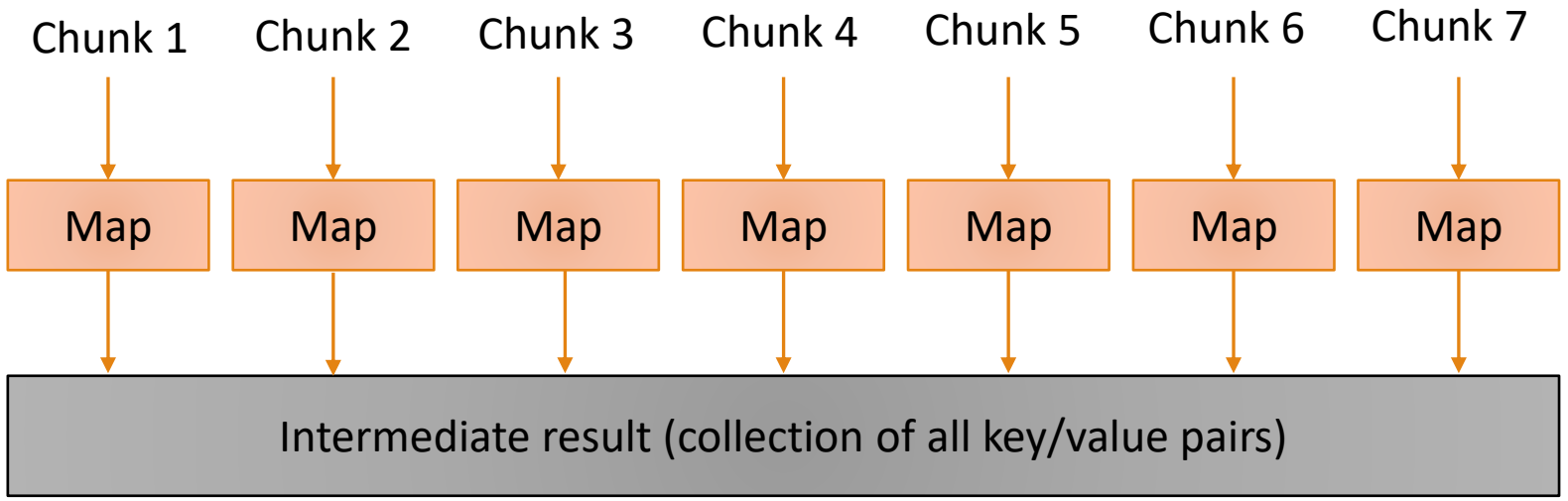
Execution

Chunk 1 Chunk 2 Chunk 3 Chunk 4 Chunk 5 Chunk 6 Chunk 7

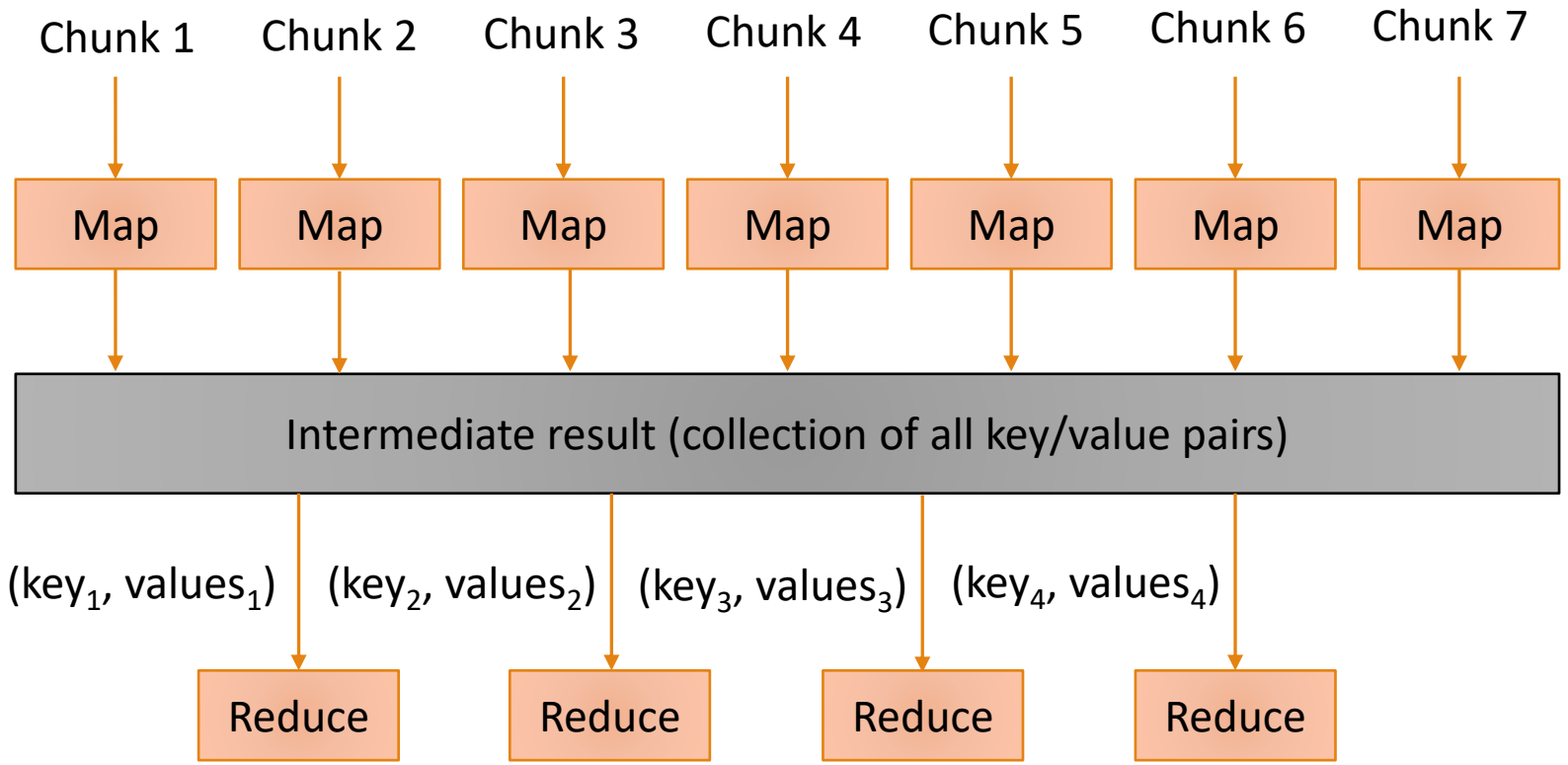
Execution


Chunk 1
Chunk 2
Chunk 3
Chunk 4
Chunk 5
Chunk 6
Chunk 7
Map
Map
Map
Map
Map
Map
Map

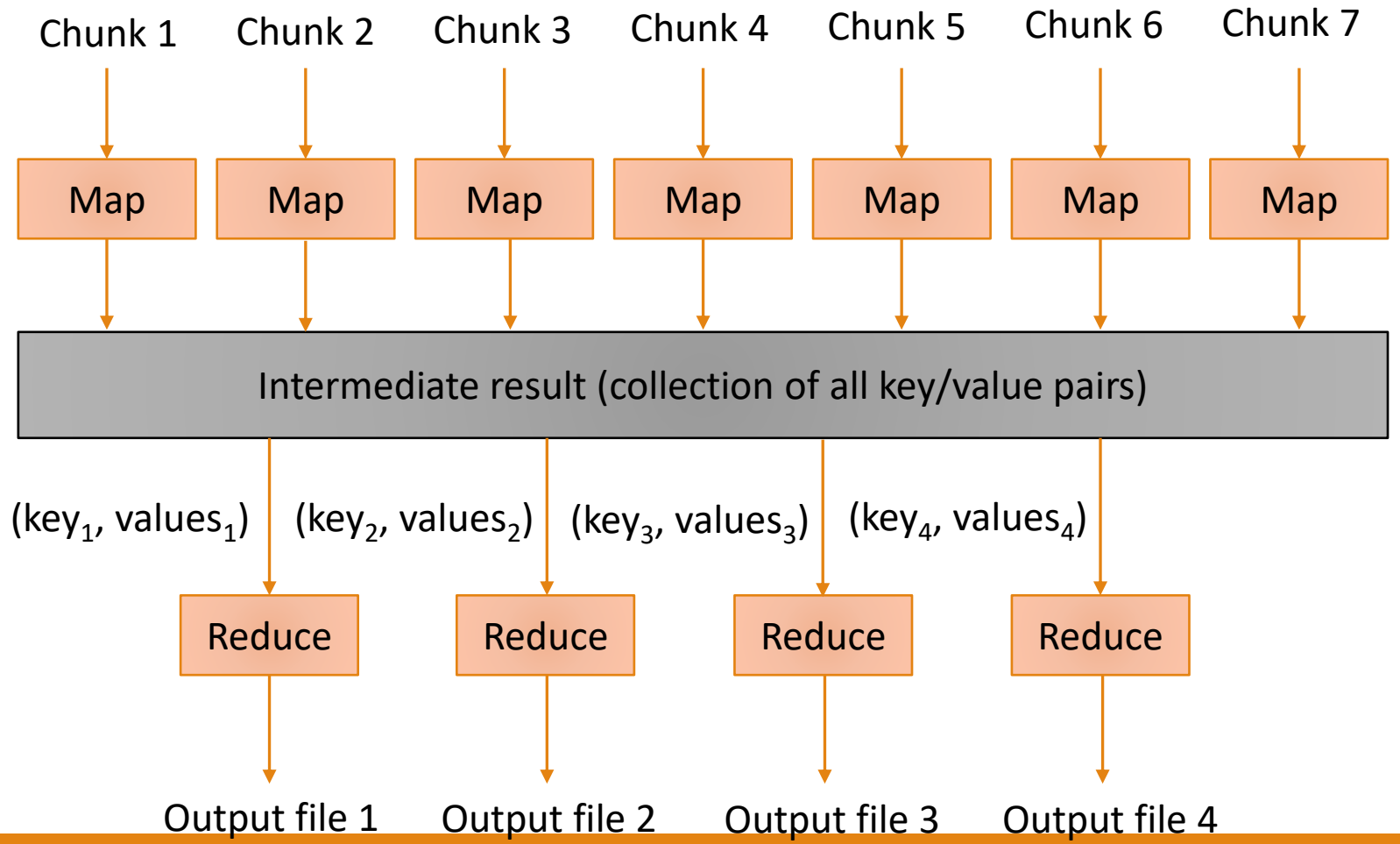
Execution


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)

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)
(key1, values1)
(key2, values2)
(key3, values3)
(key4, values4)
Reduce
Reduce
Reduce
Reduce

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)
(key1, values1)
(key2, values2)
(key3, values3)
(key4, values4)
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

We have seen:

- Counting the number of occurrences of each word
- Operators of relational algebra
- Matrix multiplication → PageRank

Expressiveness

MapReduce can be used to compute a number of interesting functions on large datasets

We have seen:

- Counting the number of occurrences of each word
- 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

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