

Map Reduce I

Big Data Management

Knowledge objectives

1. Enumerate several use cases of MapReduce
2. Explain some benefits of using MapReduce
3. Describe what the MapReduce is in the context of a DDBMS
4. Recognize the signature of Map and Reduce functions
5. Justify to which extent MapReduce is generic

Understanding objectives

1. Simulate the execution of a simple MapReduce algorithm from the user (agnostic of implementation details) perspective

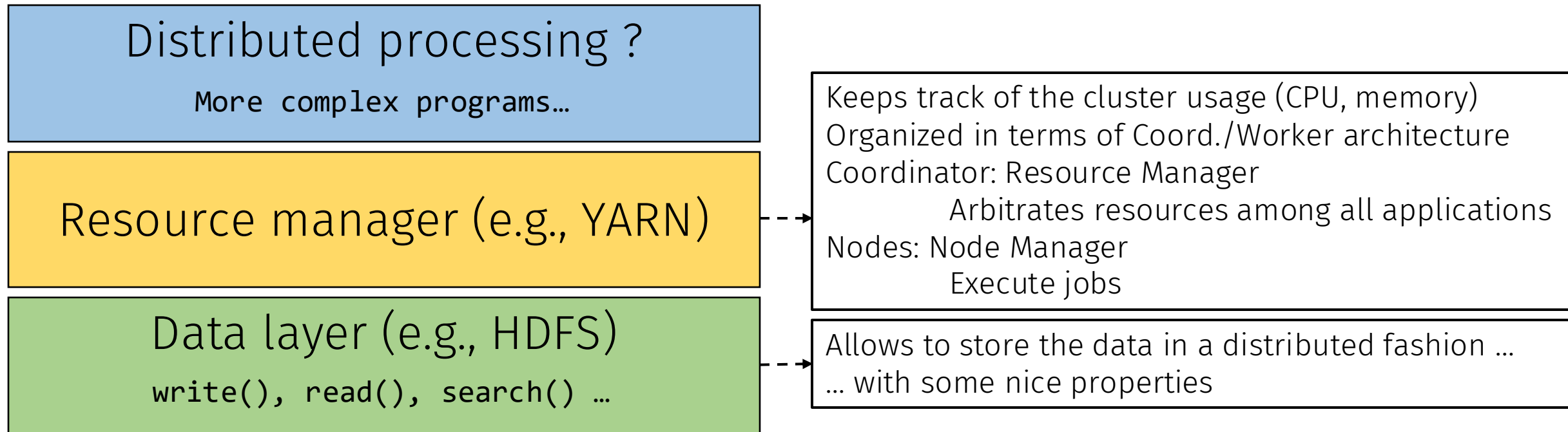
Application objectives

1. Identify the usefulness of MapReduce in a given use case
2. Define the key in the output of the map for a simple problem
3. Provide the pseudo-code of map and reduce functions for a simple problem

Distributed processing framework

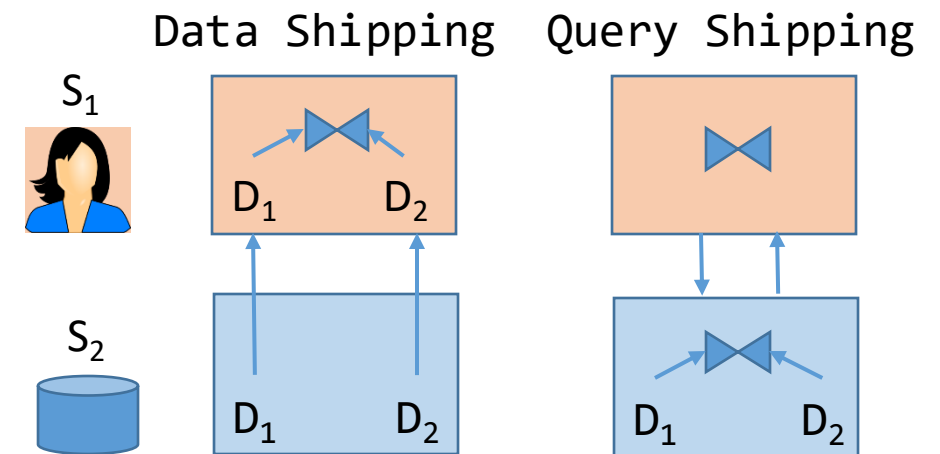
Overview and MapReduce

Distributed processing

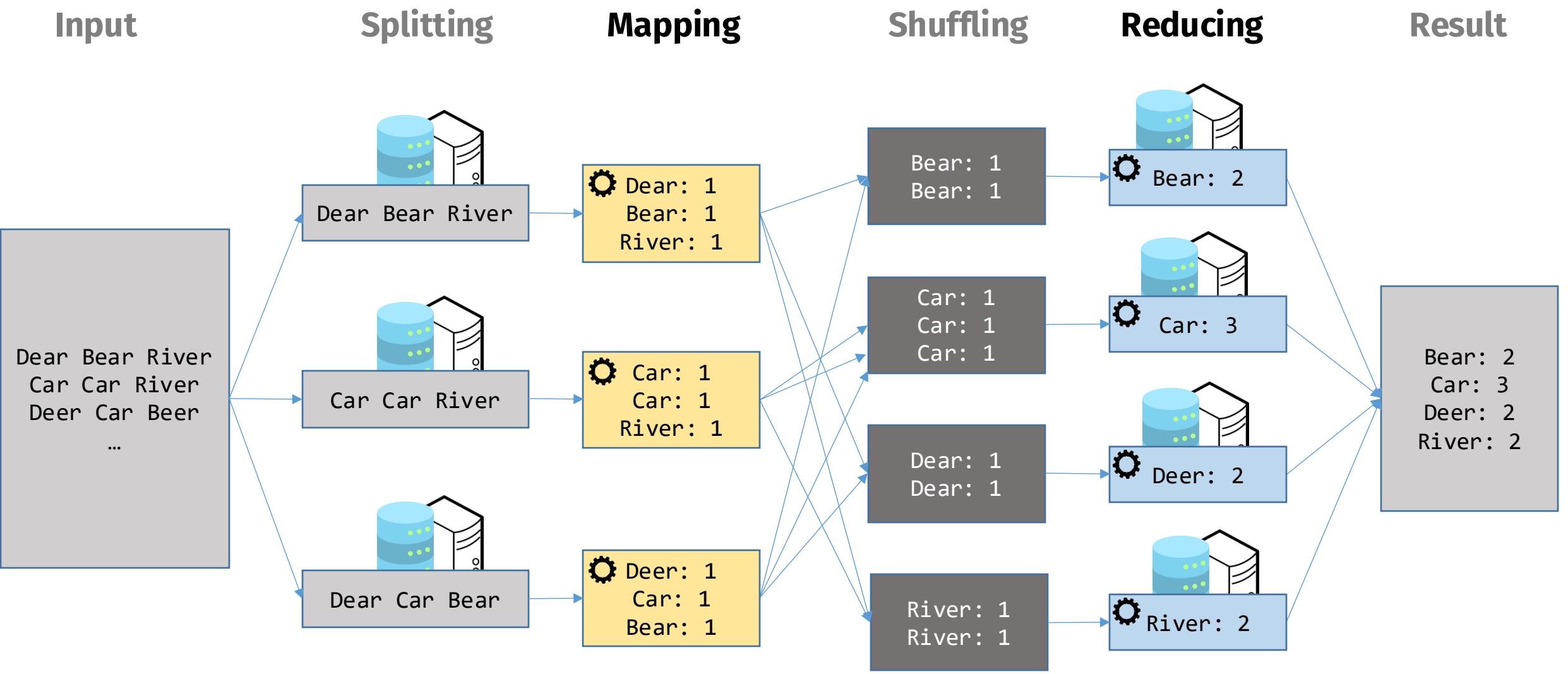


The main scenarios in data processing

- Data shipping
 - The data is retrieved from the stored site to the site executing the query
 - Avoid bottlenecks on frequently used data
 - Too much data may be moved – bandwidth intensive!
- Query shipping
 - The evaluation of the query is delegated to the site where it is stored
 - Avoid transferring large amounts of data
 - Overloads machines containing the data!
- Hybrid strategy
 - Dynamically decide data or query shipping



Word Count



MapReduce: a distributed programming model

Input

Dear Bear River
Car Car River
Deer Car Beer
...

MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

Google, Inc.

Abstract

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a *map* function that processes a key/value pair to generate a set of intermediate key/value pairs, and a *reduce* function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model, as shown in the paper.

Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication. This allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system.

Our implementation of MapReduce runs on a large cluster of commodity machines and is highly scalable: a typical MapReduce computation processes many terabytes of data on thousands of machines. Programmers find the system easy to use: hundreds of MapReduce programs have been implemented and upwards of one thou-

given day, etc. Most such computations are conceptually straightforward. However, the input data is usually large and the computations have to be distributed across hundreds or thousands of machines in order to finish in a reasonable amount of time. The issues of how to parallelize the computation, distribute the data, and handle failures conspire to obscure the original simple computation with large amounts of complex code to deal with these issues.

As a reaction to this complexity, we designed a new abstraction that allows us to express the simple computations we were trying to perform but hides the messy details of parallelization, fault-tolerance, data distribution and load balancing in a library. Our abstraction is inspired by the *map* and *reduce* primitives present in Lisp and many other functional languages. We realized that most of our computations involved applying a *map* operation to each logical "record" in our input in order to compute a set of intermediate key/value pairs, and then applying a *reduce* operation to all the values that shared the same key, in order to combine the derived data appropriately. Our use of a functional model with user-specified map and reduce operations allows us to parallelize large computations easily and to use re-execution as the primary mechanism for fault tolerance.

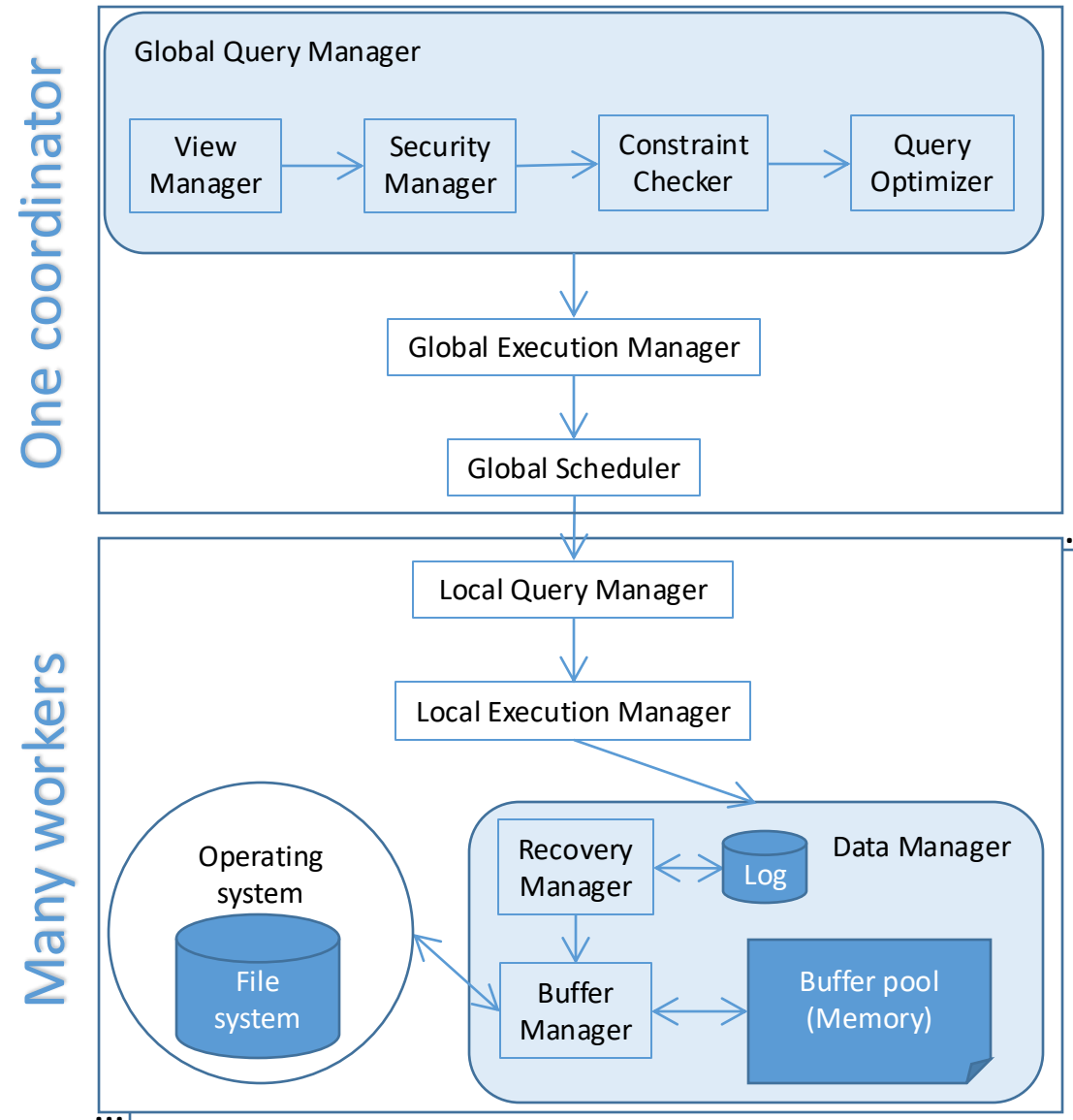
Result

Bear: 2
Car: 3
Deer: 2
River: 2

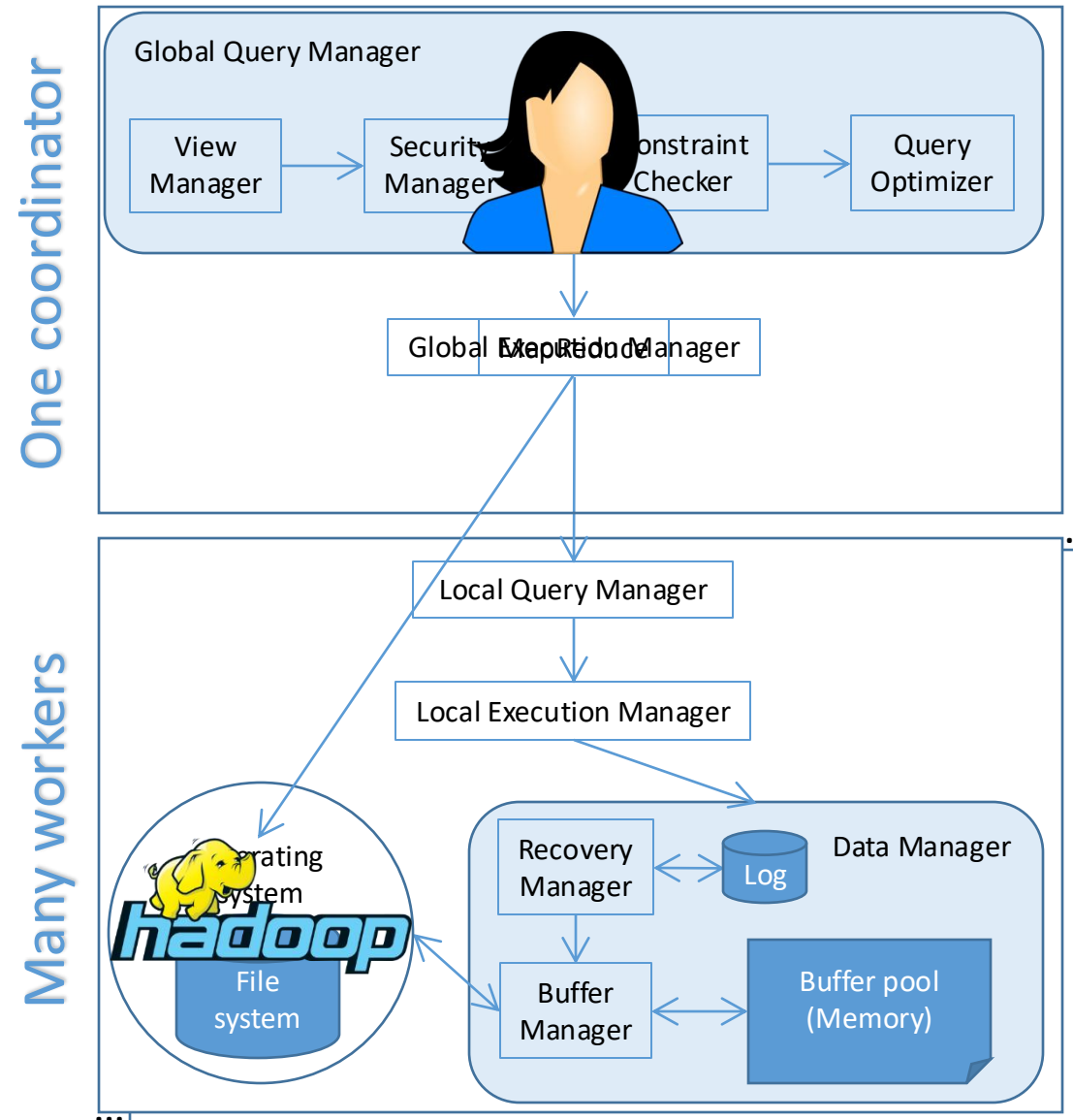
MapReduce

- The idea of MapReduce is to split the input into chunks that can be processed independently
 - These partial results are then merged into different groups in order to apply group operations (e.g., aggregations)
- Divide-and-conquer for very large data sets
 - Exploits the brute force of the cloud

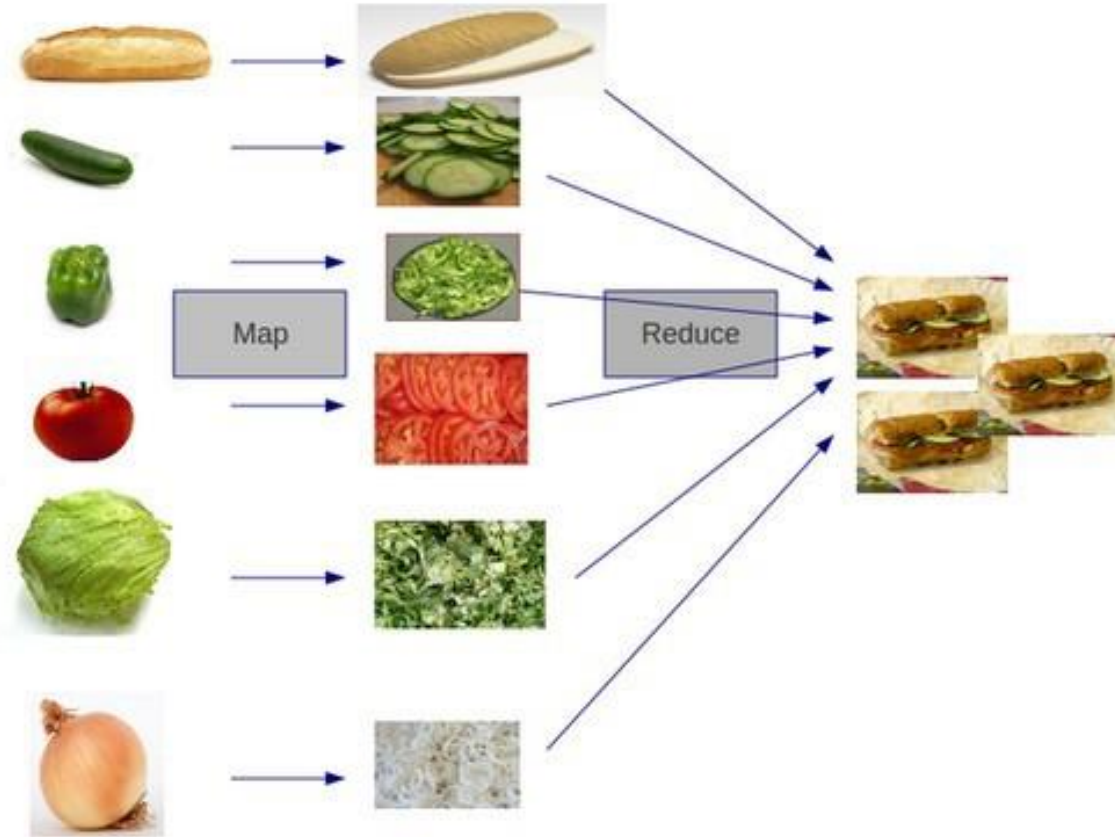
MapReduce as a DDBMS component



MapReduce as a DDBMS component



Chain production



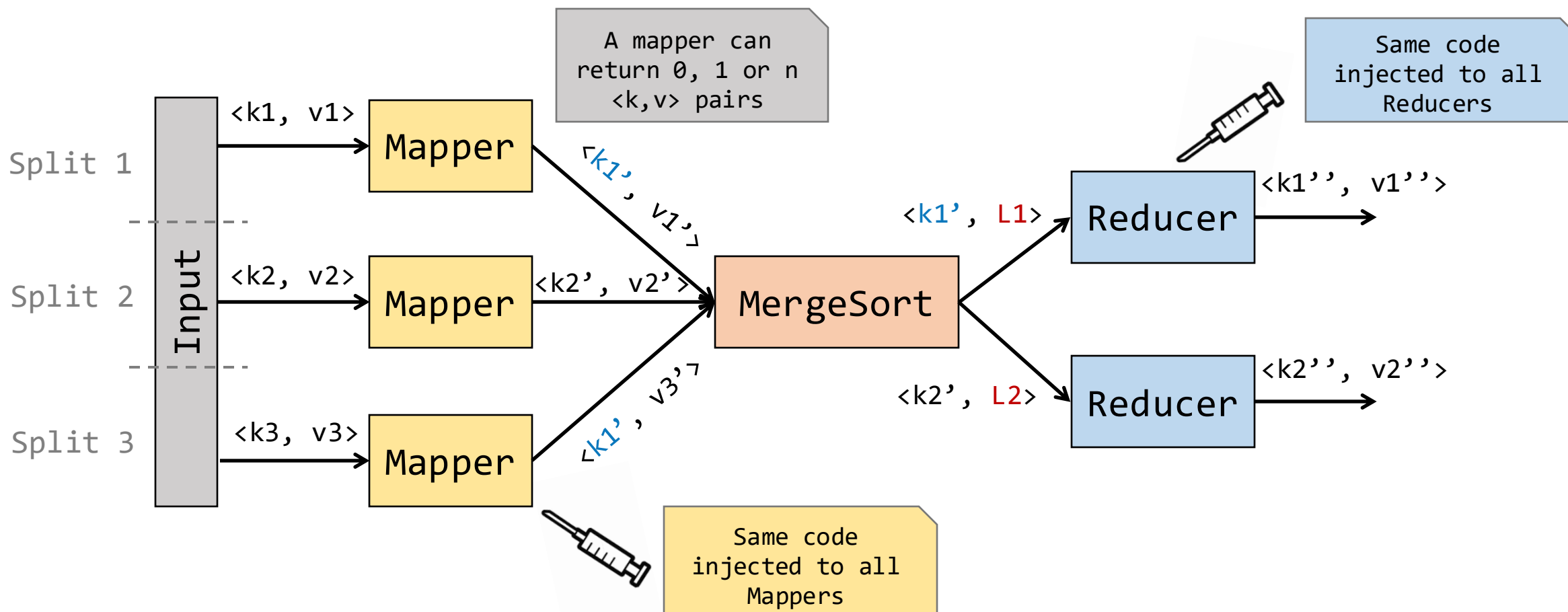
By Mohamed Nabeel

Components and use

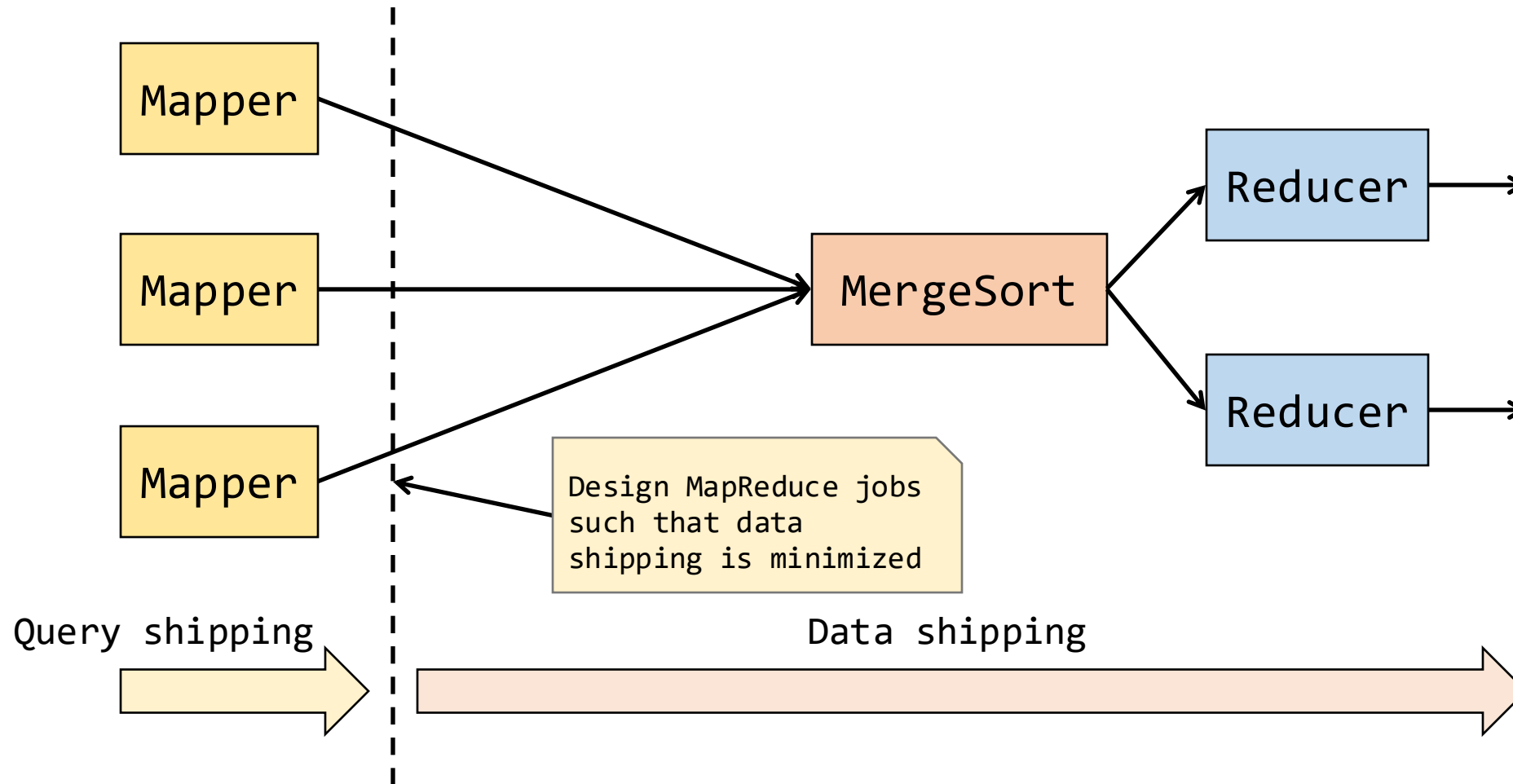
The MapReduce framework

Complex details are abstracted away from the developer

- No file I/O
- No networking code
- No synchronization



Anatomy of a MapReduce job



The MapReduce framework in detail

1. Input: read input from a DFS
2. Map: for each input $\langle \text{key}_{\text{in}}, \text{value}_{\text{in}} \rangle$
 - generate zero-to-many $\langle \text{key}_{\text{map}}, \text{value}_{\text{map}} \rangle$
3. Partition: assign sets of $\langle \text{key}_{\text{map}}, \text{value}_{\text{map}} \rangle$ to reducer machines
4. Shuffle: data are shipped to reducer machines using a DFS
5. Sort & Merge: reducers sort their input data by key
6. Reduce: for each key_{map}
 - the set $\text{value}_{\text{map}}$ is processed to produce zero-to-many $\langle \text{key}_{\text{red}}, \text{value}_{\text{red}} \rangle$
7. Output: writes the result of reducers to the DFS

Formal definition

- Single input
 - Data are represented as `<key, value>` pairs
 - Value can be anything (structured or not)
- Functional programming
 - **Map phase**, for each input `<key, value>` a function f is applied that returns a multiset of new `<key, value>` pairs:

$$f(\langle k, v \rangle) \mapsto \{ \langle k_1, v_1 \rangle, \dots, \langle k_n, v_n \rangle \}$$

- **Reduce phase**, all pairs with the same key are grouped and a function g is applied, which returns also a multiset of new `<key, value>` pairs:

$$g(\langle k, \{v_1, \dots, v_n\} \rangle) \mapsto \{ \langle k_1, v_1 \rangle, \dots, \langle k_m, v_m \rangle \}$$

The MapReduce framework

- What needs to be implemented?
- The code that processes input `<key,value>` pairs
 - The user must inject it
- The code that merges the partial results
 - Provided by the framework
- The code that processes grouped `<key, list of values>`
 - The user must inject it

MapReduce examples





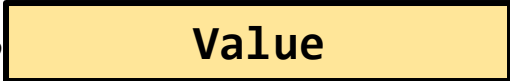
Word count


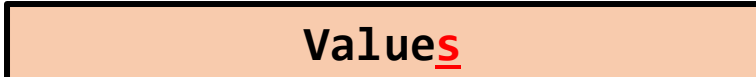



WordCount Code Example

```
public void map(LongWritable key, Text value) {  
    String line = value.toString();  
    StringTokenizer tokenizer = new StringTokenizer(line);  
    while (tokenizer.hasMoreTokens()) {  
        write(new Text(tokenizer.nextToken()), new IntWritable(1));  
    }  
}
```

```
public void reduce(Text key, Iterable<IntWritable> values) {  
    int sum = 0;  
    for (IntWritable val : values) {  
        sum += val.get();  
    }  
    write(key, new IntWritable(sum));  
}
```

WordCount Code Example

```
public void map(Key, Value) {  
    Blackbox  
    write(Key, Value);  
}  
}
```

```
public void reduce(Key, Values) {  
    Blackbox  
    write(Key, Value);  
}
```

MapReduce examples

Common friends

Friends in common

- In a social network (e.g., Facebook), we aim to compute the friends in common for every pair of users
 - This is a value that does not frequently change, so it can be precomputed
- Friends are stored as

Person -> [List of friends]

- A -> B C D
- B -> A C D E
- C -> A B D E
- D -> A B C E
- E -> B C D

Friends in common – Map task

- For every friend in the list, the mapper will generate a $\langle k, v \rangle$
 - **Key**: the input key concatenated with one friend in alphabetical order
 - **Value**: the whole list of friends
- Keys will be sorted, **a pair of friends go to the same reducer**

A -> B C D
(A B) -> B C D
(A C) -> B C D
(A D) -> B C D

B -> A C D E
(A B) -> A C D E
(B C) -> A C D E
(B D) -> A C D E
(B E) -> A C D E

C -> A B D E
(A C) -> A B D E
(B C) -> A B D E
(C D) -> A B D E
(C E) -> A B D E

...

Friends in common – Reduce task

- Reducers receive two lists of friends per pair of people

(A B) -> (B C D) (A C D E)
(A C) -> (B C D) (A B D E)
(A D) -> (B C D) (A B C E)

...

- The reduce function intersects the lists of values and generates the same key

(A B) -> (C D)
(A C) -> (B D)
(A D) -> (B C)

...

- ...when D visits A's profile we can lookup (A D) to see their common friends

Relational algebra in MapReduce

MapReduce Generality

- Supported in many store systems
 - HBase, MongoDB, CouchDB, etc.
- Programming paradigm is computationally complete
 - Any data process can be adapted to it
 - Some tasks better adapt to it than others
 - Not necessarily efficient
 - Optimization is very limited because of lack of expressivity
- Signature is closed
 - Iterations can be chained (you can have many mappers and reducers chained together)
 - Fault tolerance is not guaranteed in between
 - Resources are released to be just requested again
- Criticized for being too low-level
 - APIs for Ruby, Python, Java, C++, etc.
 - Attempts to build declarative languages on top
 - SQL-like
 - HiveQL
 - Cassandra Query Language (CQL)

Relational algebra operations

- **Unary operations**

- Selection
- Projection
- Renaming

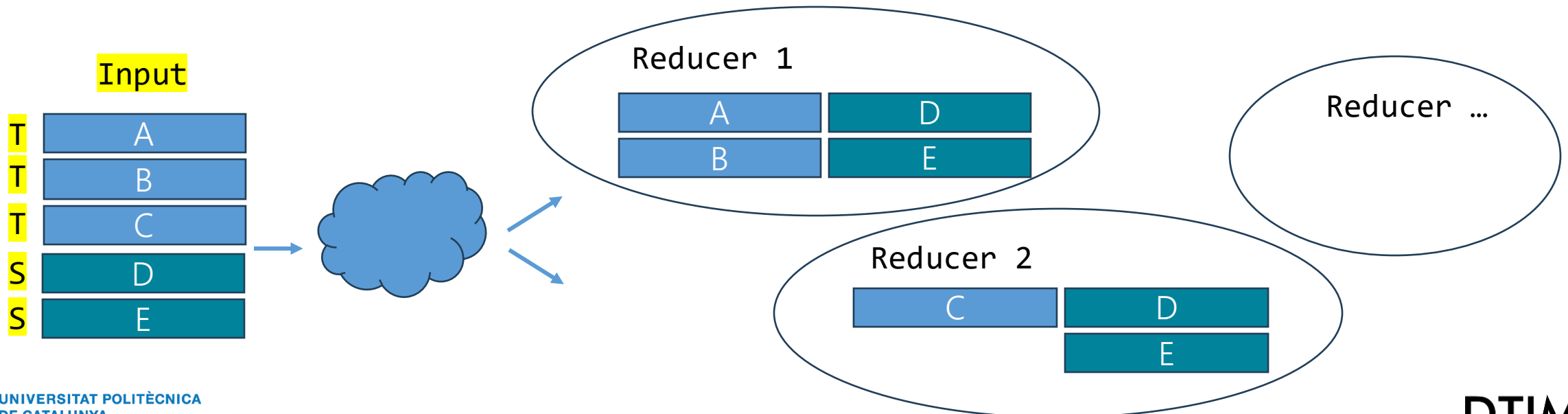
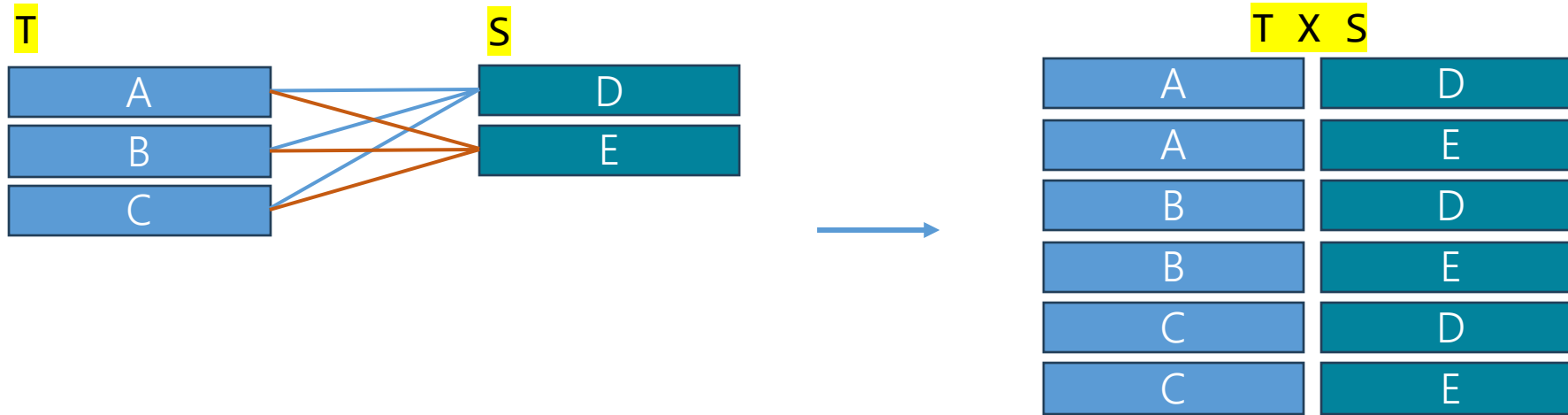
- **Binary operations**

- Union
- Intersection
- Difference
- Cartesian product
- Combination (join)

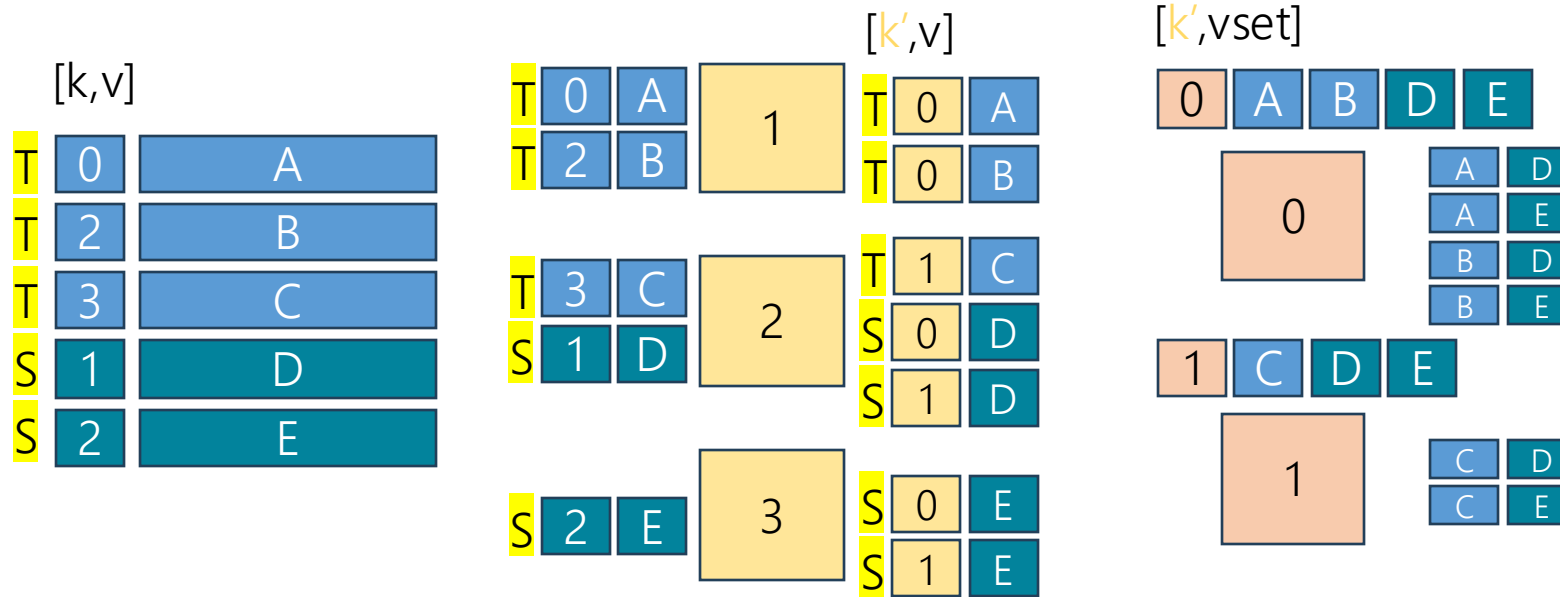
Relational operations: Projection

$$\pi_{a_{i_1}, \dots, a_{i_n}}(T) \Rightarrow \begin{cases} \text{map}(\text{key } k, \text{value } v) \mapsto [(\text{prj}_{a_{i_1}, \dots, a_{i_n}}(k \oplus v), 1)] \\ \text{reduce}(\text{key } ik, \text{vset } ivs) \mapsto [(ik)] \end{cases}$$

Relational operations: Cross Product



Relational operations: Cross Product



$$\begin{cases}
 \text{map}(\text{key } k, \text{value } v) \mapsto \\
 \begin{cases}
 [(h_T(k) \bmod D, k \oplus v)] & \text{if } \text{input}(k \oplus v) = T, \\
 [(0, k \oplus v), \dots, (D-1, k \oplus v)] & \text{if } \text{input}(k \oplus v) = S.
 \end{cases} \\
 \text{reduce}(\text{key } ik, \text{vset } ivs) \mapsto \\
 \left[\text{crossproduct}(T_{ik}, S) \mid \right. \\
 \quad T_{ik} = \{iv \mid iv \in ivs \wedge \text{input}(iv) = T\}, \\
 \quad \left. S = \{iv \mid iv \in ivs \wedge \text{input}(iv) = S\} \right]
 \end{cases}$$

Closing

Summary

- MapReduce usefulness and benefits
- MapReduce programming model
 - Expressivity
- Relational algebra in MapReduce

References

- J. Dean et al. *MapReduce: Simplified Data Processing on Large Clusters*. OSDI'04
- A. Pavlo et al. *A Comparison of Approaches to Large-Scale Data Analysis*. SIGMOD, 2009
- J. Dittrich et al. *Hadoop++: Making a Yellow Elephant Run Like a Cheetah (Without It Even Noticing)*. Proc. VLDB Endow. 3(1-2), 2010
- M. Stonebraker et al. *MapReduce and parallel DBMSs: friends or foes?* Communication of ACM 53(1), 2010
- S. Abiteboul et al. *Web data management*. Cambridge University Press, 2011
- A. Rajaraman et al. *Mining massive data sets*. Cambridge University Press, 2012
- P. Sadagale and M. Fowler. *NoSQL distilled*. Addison-Wesley, 2013