

**NDBI040: Big Data Management and NoSQL Databases**

<http://www.ksi.mff.cuni.cz/~svoboda/courses/171-NDBI040/>

Lecture 5

# **MapReduce, Apache Hadoop**

**Martin Svoboda**

svoboda@ksi.mff.cuni.cz

31. 10. 2017

**Charles University in Prague**, Faculty of Mathematics and Physics

**Czech Technical University in Prague**, Faculty of Electrical Engineering

# Lecture Outline

## MapReduce

- Programming model and implementation
- Motivation, principles, details, ...

## Apache Hadoop

- HDFS – *Hadoop Distributed File System*
- MapReduce

# Programming Models

What is a **programming model**?

- **Abstraction of an underlying computer system**
  - Describes a **logical view** of the provided functionality
  - Offers a **public interface**, resources or other constructs
  - Allows for the expression of **algorithms and data structures**
  - Conceals physical reality of the **internal implementation**
  - Allows us to work at a (much) **higher level of abstraction**
- The point is  
how the intended user thinks in order to solve their tasks  
and not necessarily how the system actually works

# Programming Models

## Examples

- Traditional **von Neumann model**
  - **Architecture of a physical computer** with several components such as a central processing unit (CPU), arithmetic-logic unit (ALU), processor registers, program counter, memory unit, etc.
  - Execution of a **stream of instructions**
- **Java Virtual Machine (JVM)**
- ...

Do not confuse programming models with

- **Programming paradigms** (procedural, functional, logic, modular, object-oriented, recursive, generic, data-driven, parallel, ...)
- **Programming languages** (Java, C++, ...)

# Programming Models

## Parallel Programming Models

### Process interaction

*Mechanisms of mutual communication of parallel processes*

- **Shared memory** – shared global address space, asynchronous read and write access, synchronization primitives
- **Message passing**
- **Implicit interaction**

### Problem decomposition

*Ways of problem decomposition into tasks executed in parallel*

- **Task parallelism**
- **Data parallelism** – independent tasks on disjoint partitions of data
- **Implicit parallelism**

# MapReduce

# MapReduce Framework

What is **MapReduce**?

- **Programming model + implementation**
- Developed by Google in 2008

*Google:*

A simple and powerful interface that enables **automatic parallelization and distribution of large-scale computations**, combined with an implementation of this interface that achieves high performance on **large clusters of commodity PCs**.

# MapReduce Framework

## A bit of history and motivation

### Google PageRank problem (2003)

- How to rank tens of billions of web pages by their importance
  - ... efficiently in a reasonable amount of time
  - ... when data is scattered across thousands of computers
  - ... data files can be enormous (terabytes or more)
  - ... data files are updated only occasionally (just appended)
  - ... sending the data between compute nodes is expensive
  - ... hardware failures are rule rather than exception
- Centralized index structure was no longer sufficient
- Solution
  - **Google File System** – a distributed file system
  - **MapReduce** – a programming model



# MapReduce Framework

## MapReduce **programming model**

- **Cluster** of commodity personal computers (nodes)
  - Each running a host operating system, mutually interconnected within a network, communication based on IP addresses, ...
- **Data is distributed among the nodes**
- **Tasks executed in parallel across the nodes**

## Classification

- Process interaction: **message passing**
- Problem decomposition: **data parallelism**

# MapReduce Model

## Basic Idea

### Divide-and-conquer paradigm

- **Map** function
  - Breaks down a problem into sub-problems
  - Processes input data in order to **generate a set of intermediate key-value pairs**
- **Reduce** function
  - Receives and combines sub-solutions to solve the problem
  - Processes and possibly **reduces intermediate values associated with the same intermediate key**

And that's all!

# MapReduce Model

## Basic Idea

And that's all!

It means...

- We only need to **implement *Map* and *Reduce* functions**
- Everything else such as
  - input data distribution,
  - scheduling of execution tasks,
  - monitoring of computation progress,
  - inter-machine communication,
  - handling of machine failures,
  - ...

is managed automatically by the framework!

# MapReduce Model

A bit more formally...

## Map function

- **Input: an input key-value pair** (input *record*)
- **Output: a set of intermediate key-value pairs**
  - Usually from a different domain
  - Keys do not have to be unique
- $(key, value) \rightarrow \text{list of } (key, value)$

## Reduce function

- **Input: an intermediate key + a set of (all) values** for this key
- **Output: a possibly smaller set of values** for this key
  - From the same domain
- $(key, \text{list of } values) \rightarrow (key, \text{list of } values)$

# Example: Word Frequency

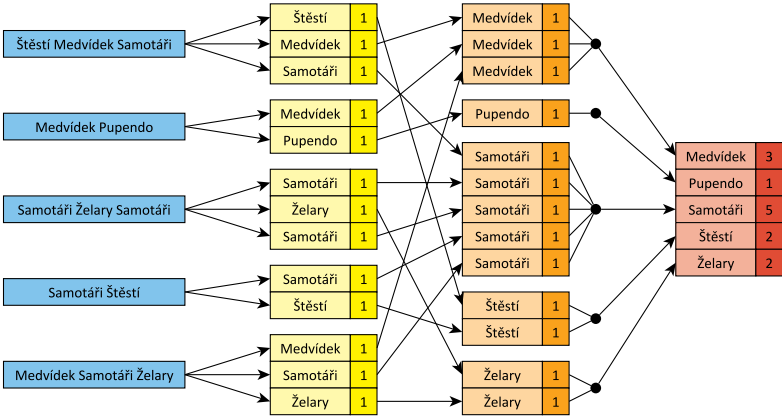
## Implementation

```
/**
 * Map function
 * @param key Document identifier
 * @param value Document contents
 */
map(String key, String value) {
    foreach word w in value: emit(w, 1);
}
```

```
/**
 * Reduce function
 * @param key Particular word
 * @param values List of count values generated for this word
 */
reduce(String key, Iterator values) {
    int result = 0;
    foreach v in values: result += v;
    emit(key, result);
}
```

# Logical Phases

-----> Mapping -----> Shuffling -----> Reducing ----->



Input records -----> Intermediate key-value pairs -----> Output file

# Logical Phases

## Mapping phase

- **Map function** is executed **for each input record**
- Intermediate key-value pairs are emitted

## Shuffling phase

- Intermediate key-value pairs are **grouped and sorted** according to the keys

## Reducing phase

- **Reduce function** is executed **for each intermediate key**
- Final output is generated

# Framework Architecture

## Master-slave architecture

- Master
  - **Manages the entire execution** of MapReduce jobs
  - **Schedules individual Map / Reduce tasks** to idle workers
- Slave (*worker*)
  - **Accepts Map / Reduce tasks from the master**

## Input / output files

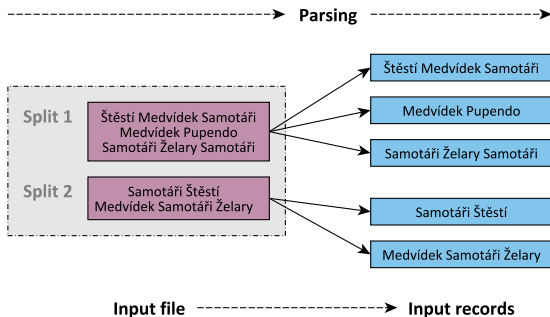
- Stored in the **underlying distributed file system**
- Actual contents of these files...
  - **Divided into smaller splits**
  - **Physically stored by individual slaves**



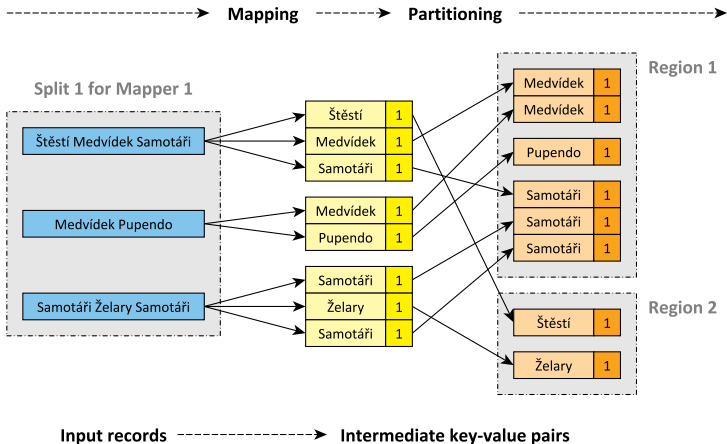
# Input Parsing

## Parsing phase

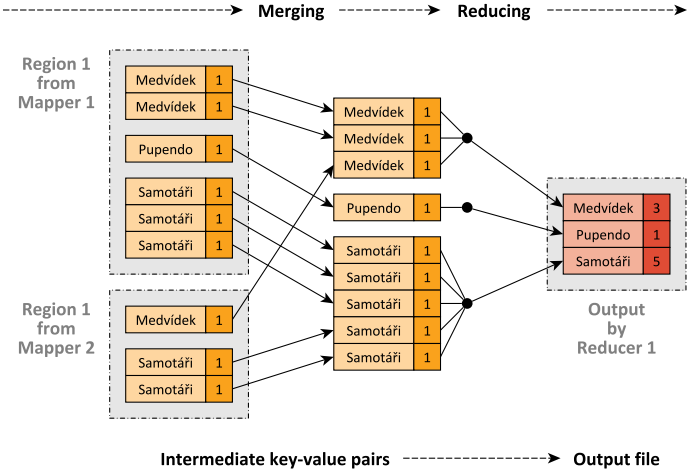
- **Each split is parsed** so that **input records** are retrieved (i.e. input key-value pairs are obtained)



# Mapping Phase



# Reducing Phase



# Execution Functions

## Input reader

- Parses a given input split and **prepares input records**

## Map function

## Partition function

- **Determines a particular Reducer** for a given intermediate key

## Compare function

- Mutually **compares two intermediate keys**

## Combine function

## Reduce function

## Output writer

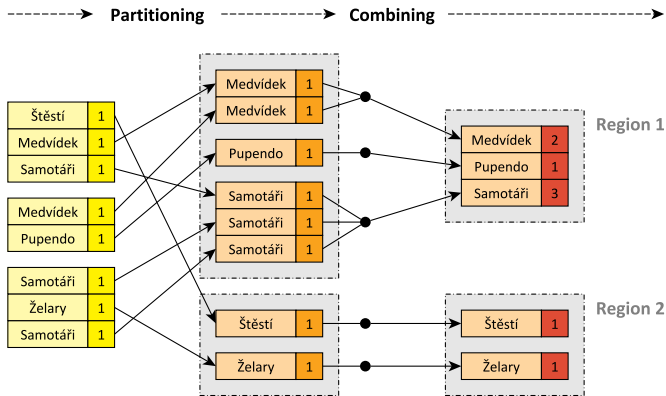
- **Writes the output** of a given Reducer

# Combine Function

## Optional **Combine** function

- Analogous purpose and implementation to the **Reduce function**
- Objective
  - **Decrease the amount of intermediate data**  $\Rightarrow$   
i.e. decrease the amount of data transferred to Reducers
- Executed locally by Mapper before the shuffling phase
- Only works for commutative and associative functions!

# Combine Function



Intermediate key-value pairs

# Advanced Aspects

## Counters

- Allow to track the progress of a MapReduce job in real time
  - **Predefined counters**
    - E.g. numbers of launched / finished Map / Reduce tasks, parsed input key-value pairs, ...
  - **Custom counters** (user-defined)
    - Can be associated with any action that a Map or Reduce function does

# Advanced Aspects

## Fault tolerance

- When a large number of nodes process a large number of data  
⇒ **fault tolerance is necessary**

## Worker failure

- Master periodically pings every worker; if no response is received in a certain amount of time, master marks the worker as failed
- **All its tasks are reset back to their initial idle state and become eligible for rescheduling on other workers**

## Master failure

- Strategy A – periodic checkpoints are created; if master fails, a new copy can then be started
- Strategy B – master failure is considered to be highly unlikely; users simply resubmit unsuccessful jobs



# Advanced Aspects

## Stragglers

- **Straggler** = node that takes unusually long time to complete a task it was assigned
- Solution
  - When a MapReduce job is close to completion, the master schedules **backup executions** of the remaining in-progress tasks
  - A given task is considered to be completed whenever either the primary or the backup execution completes

# Advanced Aspects

## Task granularity

- Intended **numbers of Map and Reduce tasks**
- Practical recommendation (by Google)
  - **Map tasks**
    - Choose the number so that each individual Map task has roughly 16 – 64 MB of input data
  - **Reduce tasks**
    - Small multiple of the number of worker nodes we expect to use
    - Note also that the **output of each Reduce task ends up in a separate output file**

# Additional Examples

## URL access frequency

- *Input*: HTTP server access logs
- *Map*: parses a log, emits (accessed URL, 1) pairs
- *Reduce*: computes and emits the sum of the associated values
- *Output*: overall number of accesses to a given URL

## Inverted index

- *Input*: text documents containing words
- *Map*: parses a document, emits (word, document ID) pairs
- *Reduce*: emits all the associated document IDs sorted
- *Output*: list of documents containing a given word

# Additional Examples

## Distributed sort

- *Input*: records to be sorted according to a specific criterion
- *Map*: extracts the sorting key, emits (key, record) pairs
- *Reduce*: emits the associated records unchanged

## Reverse web-link graph

- *Input*: web pages with `<a href="...">...</a>` tags
- *Map*: emits (target URL, current document URL) pairs
- *Reduce*: emits the associated source URLs unchanged
- *Output*: list of URLs of web pages targeting a given one

# Additional Examples

## Sources of links between web pages

```
/**
 * Map function
 * @param key    Source web page URL
 * @param value  HTML contents of this web page
 */
map(String key, String value) {
    foreach <a> tag t in value: emit(t.href, key);
}
```

```
/**
 * Reduce function
 * @param key    URL of a particular web page
 * @param values List of URLs of web pages targeting this one
 */
reduce(String key, Iterator values) {
    emit(key, values);
}
```

# Use Cases: General Patterns

## Counting, summing, aggregation

- When the overall number of occurrences of certain items or a different aggregate function should be calculated

## Collating, grouping

- When all items belonging to a certain group should be found, collected together or processed in another way

## Filtering, querying, parsing, validation

- When all items satisfying a certain condition should be found, transformed or processed in another way

## Sorting

- When items should be processed in a particular order with respect to a certain ordering criterion

# Use Cases: Real-World Problems

Just a few **real-world examples**...

- Risk modeling, customer churn
- Recommendation engine, customer preferences
- Advertisement targeting, trade surveillance
- Fraudulent activity threats, security breaches detection
- Hardware or sensor network failure prediction
- Search quality analysis
- ...

# Apache Hadoop





# Apache Hadoop

## Open-source software framework

- <http://hadoop.apache.org/>
- **Distributed storage and processing** of very large data sets on clusters built from commodity hardware
  - Implements a **distributed file system**
  - Implements a **MapReduce** programming model
- Derived from the original Google MapReduce and GFS
- Developed by Apache Software Foundation
- Implemented in Java
- Operating system: cross-platform
- Initial release in 2011

# Apache Hadoop

## Modules

- Hadoop **Common**
  - Common utilities and support for other modules
- Hadoop **Distributed File System** (HDFS)
  - High-throughput distributed file system
- Hadoop **Yet Another Resource Negotiator** (YARN)
  - Cluster resource management
  - Job scheduling framework
- Hadoop **MapReduce**
  - YARN-based implementation of the MapReduce model

# Apache Hadoop

## Hadoop-related projects

- Apache **Cassandra** – wide column store
- Apache **HBase** – wide column store
- Apache **Hive** – data warehouse infrastructure
- Apache **Avro** – data serialization system
- Apache **Chukwa** – data collection system
- Apache **Mahout** – machine learning and data mining library
- Apache **Pig** – framework for parallel computation and analysis
- Apache **ZooKeeper** – coordination of distributed applications
- ...

# Apache Hadoop

## Real-world Hadoop users

- **Facebook** – internal logs, analytics, machine learning, 2 clusters  
1100 nodes (8 cores, 12 TB storage), 12 PB  
300 nodes (8 cores, 12 TB storage), 3 PB
- **LinkedIn** – 3 clusters  
800 nodes (2×4 cores, 24 GB RAM, 6×2 TB SATA), 9 PB  
1900 nodes (2×6 cores, 24 GB RAM, 6×2 TB SATA), 22 PB  
1400 nodes (2×6 cores, 32 GB RAM, 6×2 TB SATA), 16 PB
- **Spotify** – content generation, data aggregation, reporting, analysis  
1650 nodes, 43000 cores, 70 TB RAM, 65 PB, 20000 daily jobs
- **Yahoo!** – 40000 nodes with Hadoop, biggest cluster  
4500 nodes (2×4 cores, 16 GB RAM, 4×1 TB storage), 17 PB

# HDFS

## Hadoop Distributed File System



- Open-source, high quality, cross-platform, pure Java
- **Highly scalable, high-throughput, fault-tolerant**
- Master-slave architecture
- Optimal applications
  - MapReduce, web crawlers, data warehouses, ...

# HDFS: Assumptions

## Data characteristics

- **Large data sets** and files
- **Streaming data access**
- **Batch processing** rather than interactive users
- **Write-once, read-many**

## Fault tolerance

- HDFS cluster may consist of thousands of nodes
  - Each component has a non-trivial probability of failure
- ⇒ there is always some component that is non-functional
  - I.e. failure is the norm rather than exception, and so
  - **automatic failure detection and recovery** is essential

# HDFS: File System

Logical view: Linux-based **hierarchical file system**

- **Directories and files**
- Contents of files is divided into blocks
  - Usually **64 MB**, configurable per file level
- User and group **permissions**
- Standard **operations** are provided
  - Create, remove, move, rename, copy, ...

## Namespace

- Contains names of all directories, files, and other metadata
  - I.e. all data to capture the whole logical view of the file system
- Just a single namespace for the entire cluster

# HDFS: Cluster Architecture

## Master-slave architecture

- Master: **NameNode**
  - **Manages the file system namespace**
  - **Manages file blocks** (mapping of logical to physical blocks)
  - **Provides the user interface** for all the operations
    - Create, remove, move, rename, copy, ... file or directory
    - **Open and close file**
  - Regulates access to files by users
- Slave: **DataNode**
  - **Physically stores file blocks** within the underlying file system
  - **Serves read/write requests from users**
    - I.e. user data never flows through the NameNode
  - **Has no knowledge about the file system**



# HDFS: Replication

**Replication** = maintaining of **multiple copies of each file block**

- Increases read throughput, increases fault tolerance
- **Replication factor** (number of copies)
  - Configurable per file level, usually 3

## Replica placement

- Critical to reliability and performance
- **Rack-aware strategy**
  - Takes the physical location of nodes into account
  - **Network bandwidth between the nodes on the same rack is greater than between the nodes in different racks**
- Common case (replication factor 3):
  - Two replicas on two different nodes in a local rack
  - Third replica on a node in a different rack

# HDFS: NameNode

## How the **NameNode** Works?

- **FsImage** – data structure describing the whole file system
  - Contains: **namespace + mapping of blocks + system properties**
  - Loaded into the system memory (4 GB RAM is sufficient)
  - Stored in the local file system, periodical checkpoints created
- **EditLog** – **transaction log** for all the metadata changes
  - E.g. when a new file is created, replication factor is changed, ...
  - Stored in the local file system
- **Failures**
  - **When the NameNode starts up**
    - FsImage and EditLog are read from the disk, transactions from EditLog are applied, new version of FsImage is flushed on the disk, EditLog is truncated

# HDFS: DataNode

## How each **DataNode** Works?

- Stores physical file blocks
  - Each block (replica) is stored as a separate local file
  - Heuristics are used to place these files in local directories
- Periodically sends **HeartBeat** messages to the NameNode
- **Failures**
  - **When a DataNode fails** or in case of a **network partition**, i.e. when the NameNode does not receive a HeartBeat message within a given time limit
    - The NameNode no longer sends read/write requests to this node, re-replication might be initiated
  - **When a DataNode starts up**
    - Generates a list of all its blocks and sends a **BlockReport** message to the NameNode

# HDFS: API

## Available **application interfaces**

- **Java API**
  - Python access or C wrapper also available
- **HTTP interface**
  - Browsing the namespace and downloading the contents of files
- **FS Shell – command line interface**
  - Intended for the user interaction
  - Bash-inspired commands
  - E.g.:
    - `hadoop fs -ls /`
    - `hadoop fs -mkdir /mydir`

# Hadoop MapReduce

## Hadoop **MapReduce**



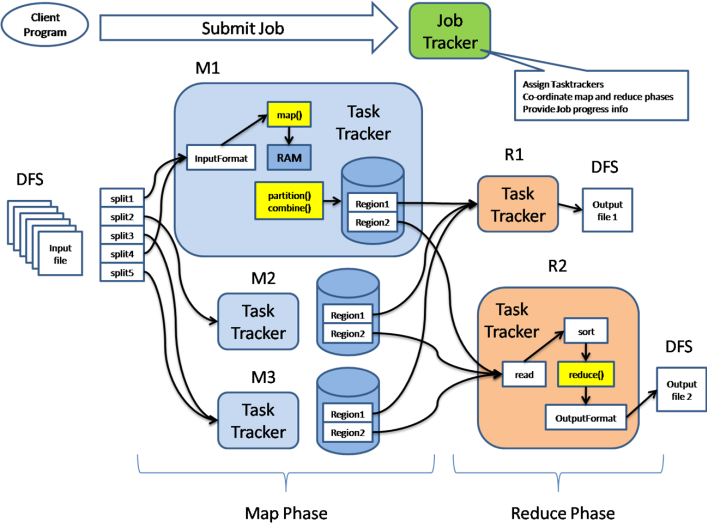
- MapReduce programming model implementation
- Requirements
  - **HDFS**
    - Input and output files for MapReduce jobs
  - **YARN**
    - Underlying distribution, coordination, monitoring and gathering of the results

# Cluster Architecture

## Master-slave architecture

- Master: **JobTracker**
  - **Provides the user interface for MapReduce jobs**
  - Fetches input file data locations from the NameNode
  - Manages the entire execution of jobs
    - Provides the progress information
  - **Schedules individual tasks** to idle TaskTrackers
    - Map, Reduce, ... tasks
    - Nodes close to the data are preferred
    - Failed tasks or stragglers can be rescheduled
- Slave: **TaskTracker**
  - **Accepts tasks from the JobTracker**
  - Spawns a separate JVM for each task execution
  - Indicates the available task slots via **HearBeat** messages

# Execution Schema



# Java Interface

## Mapper class

- Implementation of the **map function**
- Template parameters
  - KEYIN, VALUEIN – types of input key-value pairs
  - KEYOUT, VALUEOUT – types of intermediate key-value pairs
- Intermediate pairs are emitted via `context.write(k, v)`

```
class MyMapper extends Mapper<KEYIN, VALUEIN, KEYOUT, VALUEOUT> {
    @Override
    public void map(KEYIN key, VALUEIN value, Context context)
        throws IOException, InterruptedException
    {
        // Implementation
    }
}
```



# Java Interface

## Reducer class

- Implementation of the **reduce function**
- Template parameters
  - KEYIN, VALUEIN – types of intermediate key-value pairs
  - KEYOUT, VALUEOUT – types of output key-value pairs
- Output pairs are emitted via `context.write(k, v)`

```
class MyReducer extends Reducer<KEYIN, VALUEIN, KEYOUT, VALUEOUT> {
    @Override
    public void reduce(KEYIN key, Iterable<VALUEIN> values, Context context)
        throws IOException, InterruptedException
    {
        // Implementation
    }
}
```

# Example

## Word Frequency

- *Input*: Documents with words
  - Files located at `/home/input` HDFS directory
- *Map*: parses a document, emits (word, 1) pairs
- *Reduce*: computes and emits the sum of the associated values
- *Output*: overall number of occurrences for each word
  - Output will be written to `/home/output`

## MapReduce **job execution**

```
hadoop jar wc.jar WordCount /home/input /home/output
```

# Example: Mapper Class

```
public class WordCount {
    ...
    public static class MyMapper
        extends Mapper<Object, Text, Text, IntWritable>
    {
        private final static IntWritable one = new IntWritable(1);
        private Text word = new Text();
        @Override
        public void map(Object key, Text value, Context context)
            throws IOException, InterruptedException
        {
            StringTokenizer itr = new StringTokenizer(value.toString());
            while (itr.hasMoreTokens()) {
                word.set(itr.nextToken());
                context.write(word, one);
            }
        }
    }
    ...
}
```

# Example: Reducer Class

```
public class WordCount {
    ...
    public static class MyReducer
        extends Reducer<Text, IntWritable, Text, IntWritable>
    {
        private IntWritable result = new IntWritable();
        @Override
        public void reduce(Text key, Iterable<IntWritable> values,
            Context context) throws IOException, InterruptedException
        {
            int sum = 0;
            for (IntWritable val : values) {
                sum += val.get();
            }
            result.set(sum);
            context.write(key, result);
        }
    }
    ...
}
```



# Lecture Conclusion

## MapReduce criticism

- MapReduce **is a step backwards**
  - Does not use database schema
  - Does not use index structures
  - Does not support advanced query languages
  - Does not support transactions, integrity constraints, views, ...
  - Does not support data mining, business intelligence, ...
- MapReduce **is not novel**
  - Ideas more than 20 years old and overcome
  - Message Passing Interface (MPI), Reduce-Scatter

The end of MapReduce?