

NDBI006: **Query Languages II**

<http://www.ksi.mff.cuni.cz/~svoboda/courses/202-NDBI006/>

Lecture 14

MapReduce

Martin Svoboda

svoboda@ksi.mff.cuni.cz

1. 6. 2021

Charles University, Faculty of Mathematics and Physics

Lecture Outline

MapReduce

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

Apache Hadoop

- HDFS (Hadoop Distributed File System)
- MapReduce

MongoDB

- MapReduce queries

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++, ...)

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** – different tasks over the same data
- **Data parallelism** – the same task over different 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**.

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**

Basic Idea

Divide-and-conquer paradigm

- Breaks down a given problem into simpler sub-problems
- Solutions of the sub-problems are then combined together

Two core functions

- **Map function**
 - Generates a set of so-called **intermediate key-value pairs**
- **Reduce function**
 - Reduces values associated with a given intermediate key

And that's all!

Basic Idea

And that's really 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!

Model Description

Map function

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

Reduce function

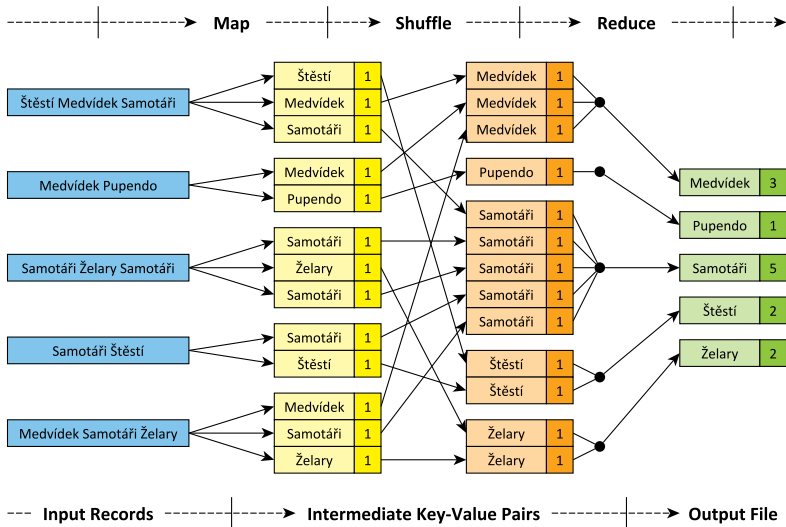
- **Input: intermediate key + list of (all) values** for this key
- **Output: possibly smaller list of values** for this key
 - Usually from the same domain
- $(key, \text{list of values}) \rightarrow (key, \text{list of values})$

Example: Word Frequency

```
/**
 * 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



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

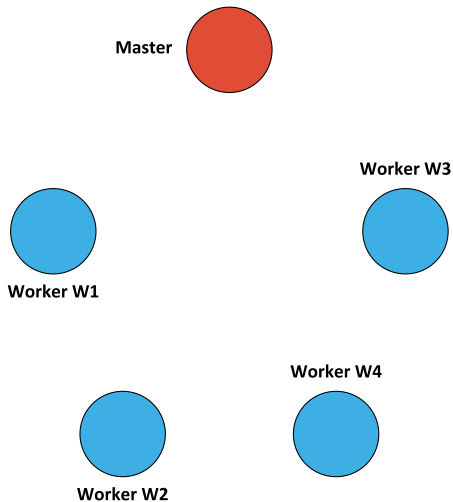
- **Reduce function** is executed **for each intermediate key**
- Output key-value pairs are generated

Cluster Architecture

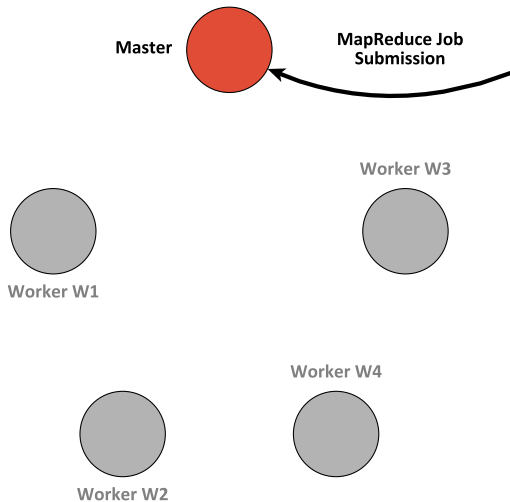
Master-slave architecture

- Two types of nodes, each with two basic roles
- **Master**
 - **Manages the execution of MapReduce jobs**
 - Schedules individual Map / Reduce tasks to idle workers
 - ...
 - **Maintains metadata about input / output files**
 - These are stored in the underlying distributed file system
- **Slaves (workers)**
 - **Physically store the actual data contents of files**
 - Files are divided into smaller parts called splits
 - Each split is stored by one / or even more particular workers
 - **Accept and execute assigned Map / Reduce tasks**

Cluster Architecture



MapReduce Job Submission



MapReduce Job Submission

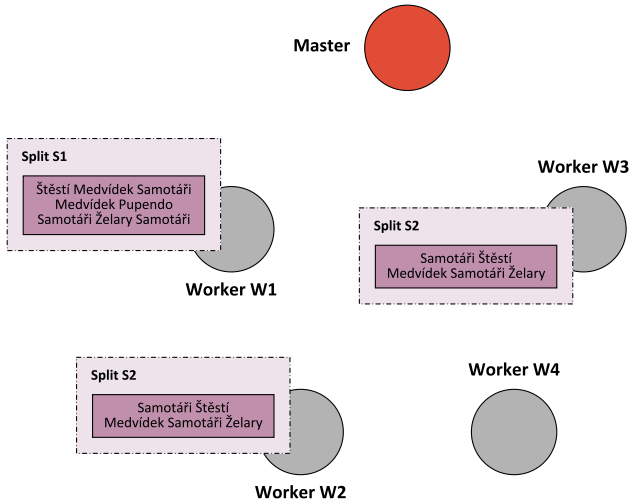
Submission of MapReduce jobs

- Jobs can only be submitted to the master node
- Client provides the following:
 - **Implementation** of (not only) **Map and Reduce functions**
 - Description of **input file** (or even files)
 - Description of **output directory**

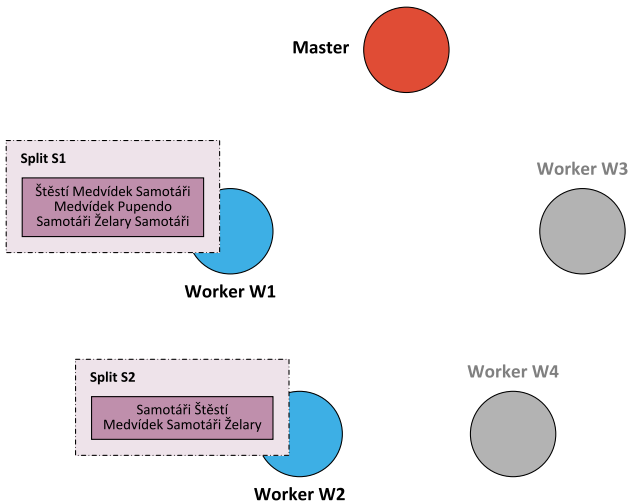
Localization of input files

- Master determines **locations of all involved splits**
 - I.e. workers containing these splits are resolved

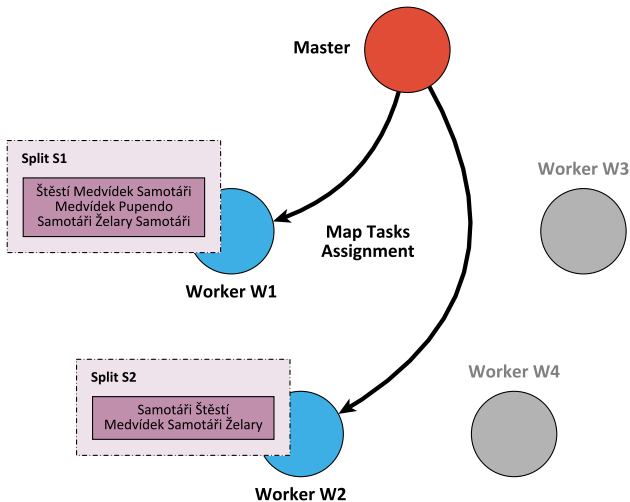
Input Splits Localization



Input Splits Localization



Map Task Assignment



Map Task Execution

Map Task = processing of 1 split by 1 worker

- Assigned by the master to an idle worker that is (preferably) already containing (physically storing) a given split

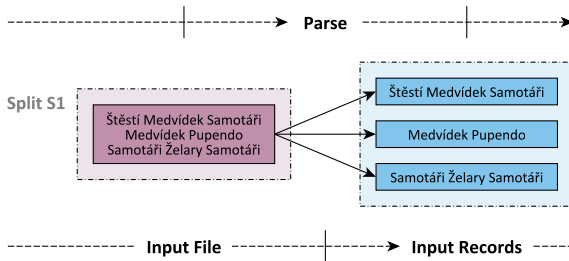
Individual steps...

- **Input reader** is used to **parse contents of the split**
 - I.e. **input records** are generated
- **Map function** is applied on each input record
 - Intermediate key-value pairs are emitted
- These pairs are **stored locally and organized into regions**
 - Either in the system memory,
or flushed to a local hard drive when necessary
 - **Partition function** is used to determine the intended region
 - Intermediate keys (not values) are used for this purpose
 - E.g. hash of the key modulo the overall number of reducers

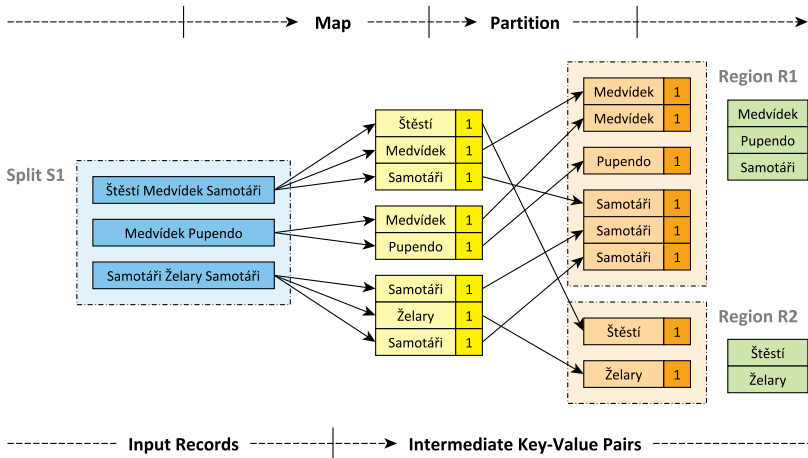
Input Parsing

Parsing phase

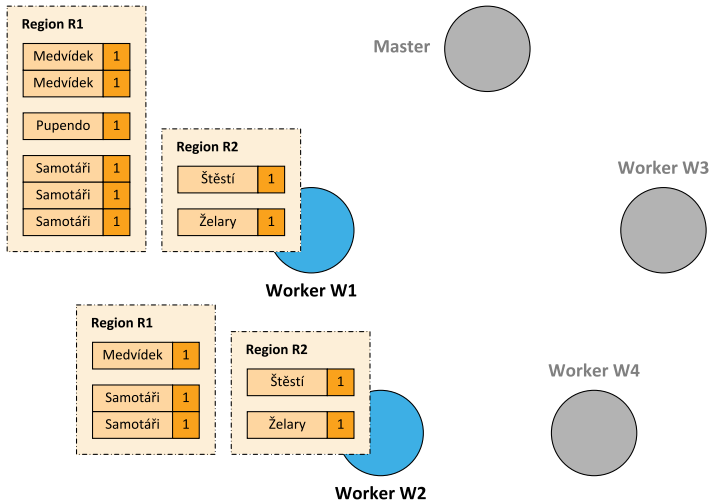
- **Each split is parsed** so that **input records** are retrieved (i.e. input key-value pairs are obtained)



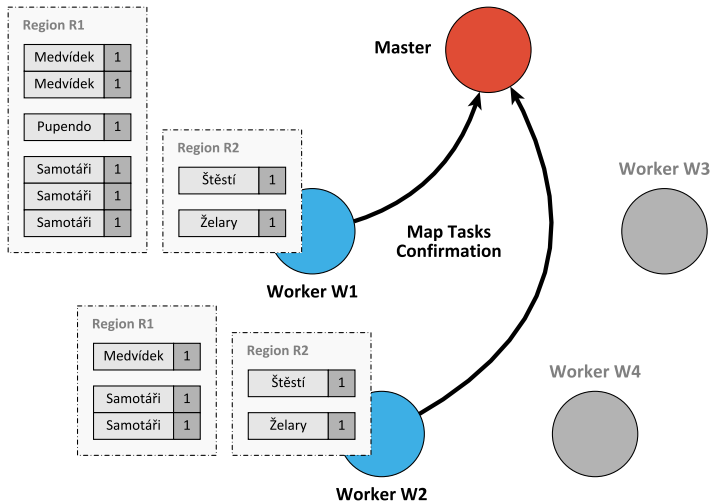
Map Phase



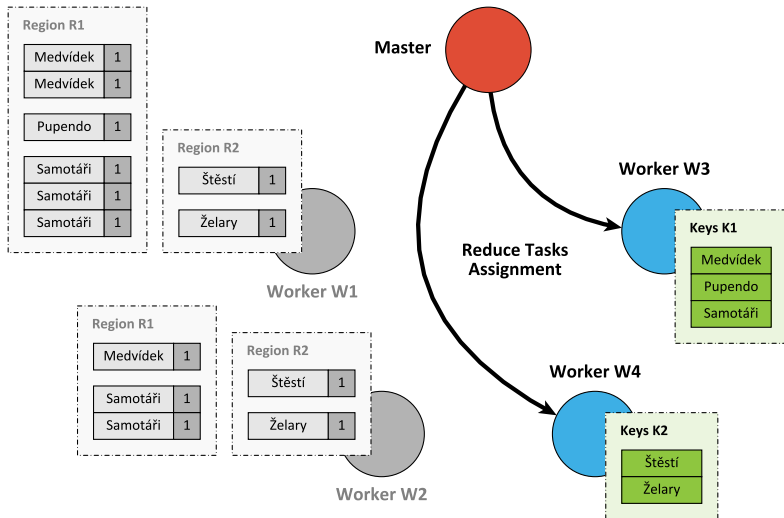
Map Phase



Map Task Confirmation



Reduce Task Assignment



Reduce Task Execution

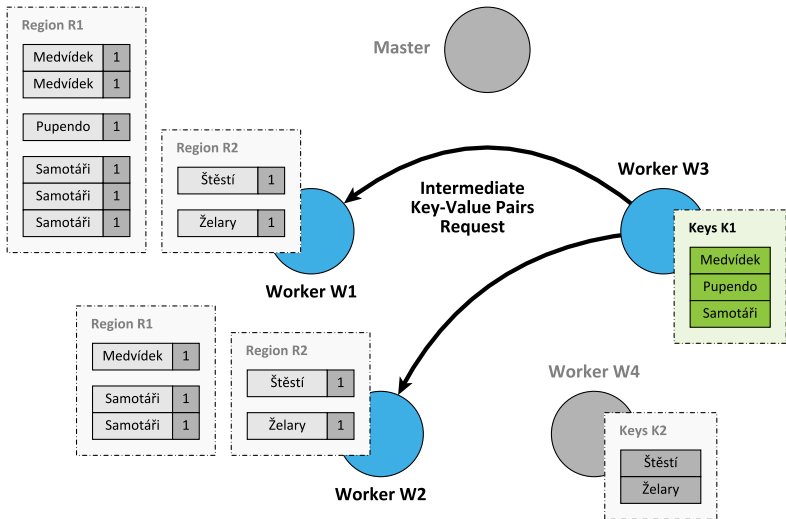
Reduce Task = reduction of selected key-value pairs by 1 worker

- Goal: processing of all emitted **intermediate key-value pairs belonging to a particular region**

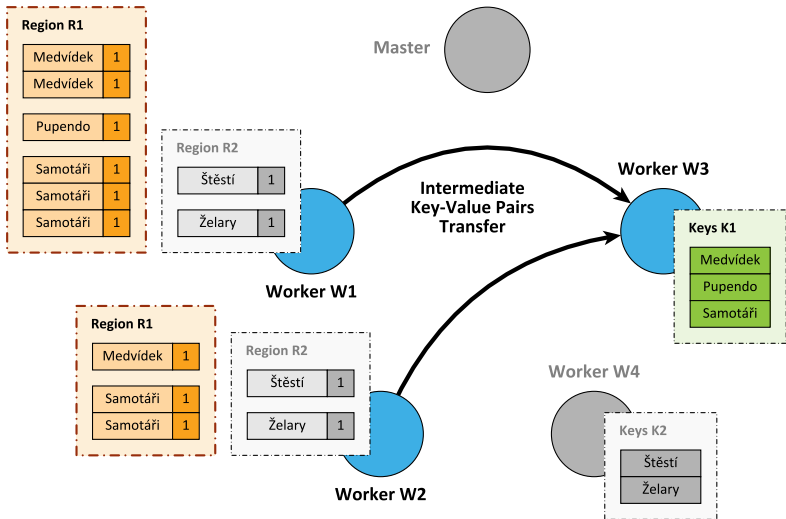
Individual steps...

- **Intermediate key-value pairs are first acquired**
 - All relevant mapping workers are addressed
 - Data of corresponding **regions are transferred** (remote read)
- Once downloaded, they are **locally merged**
 - I.e. sorted and grouped based on keys
- **Reduce function** is applied on each intermediate key
- **Output key-value pairs** are emitted and stored (**output writer**)
 - Note that each worker produces its own separate output file

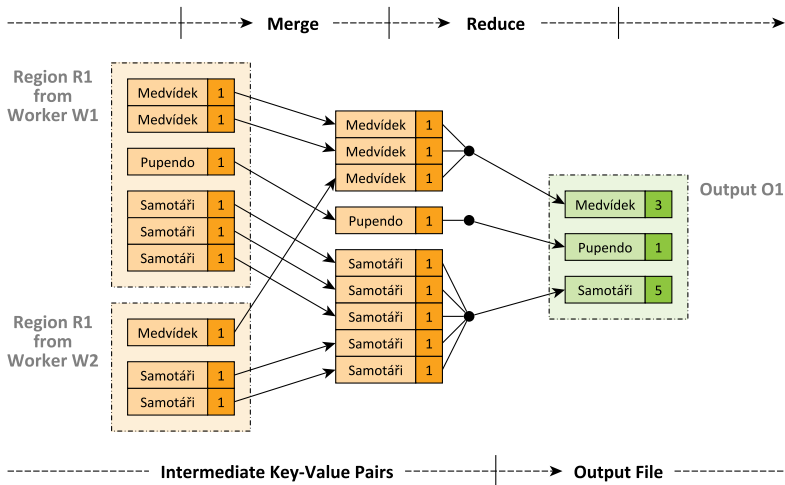
Region Data Retrieval



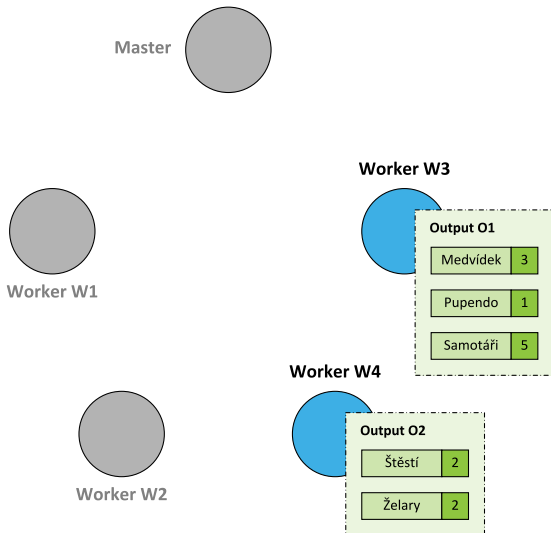
Region Data Retrieval



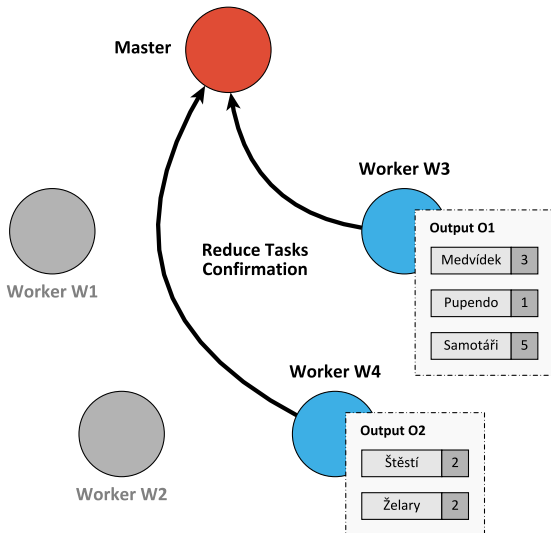
Reduce Phase



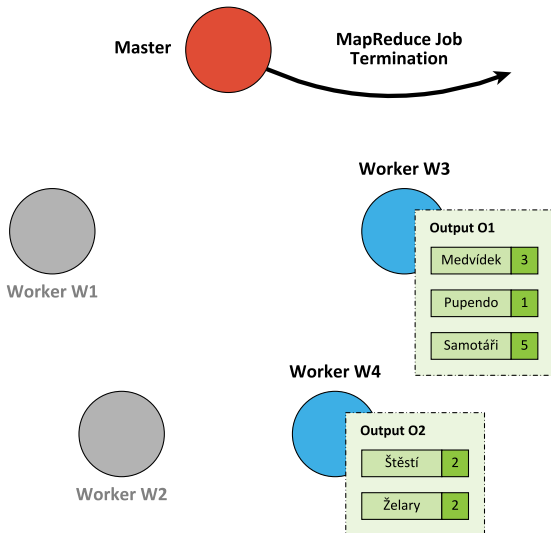
Reduce Phase



Reduce Task Confirmation



MapReduce Job Termination

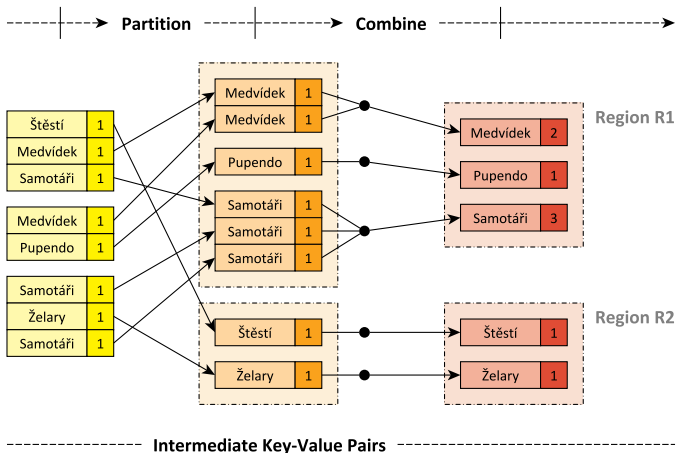


Combine Function

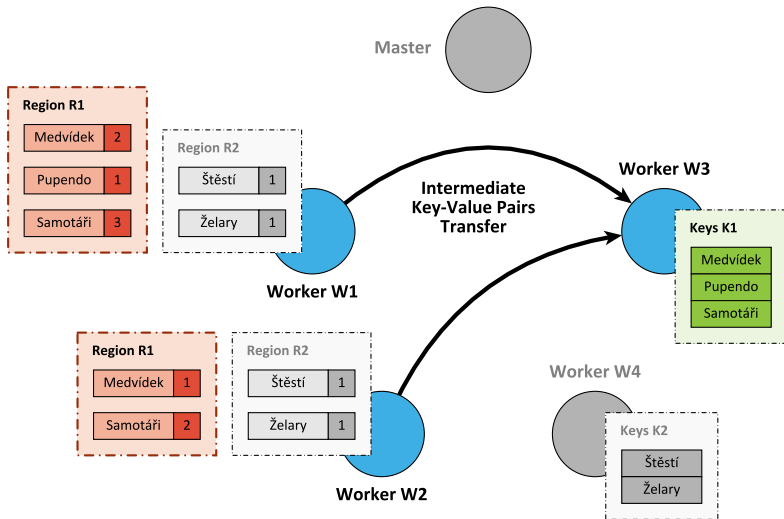
Optional **Combine** function

- Objective
 - **Decrease the amount of intermediate data**
i.e. decrease the amount of data that is needed to be transferred from Mappers to Reducers
- Analogous purpose and implementation to **Reduce function**
- **Executed locally by Mappers**
- However, only applicable when the reduction is...
 - **Commutative**
 - **Associative**
 - **Idempotent:** $f(f(x)) = f(x)$

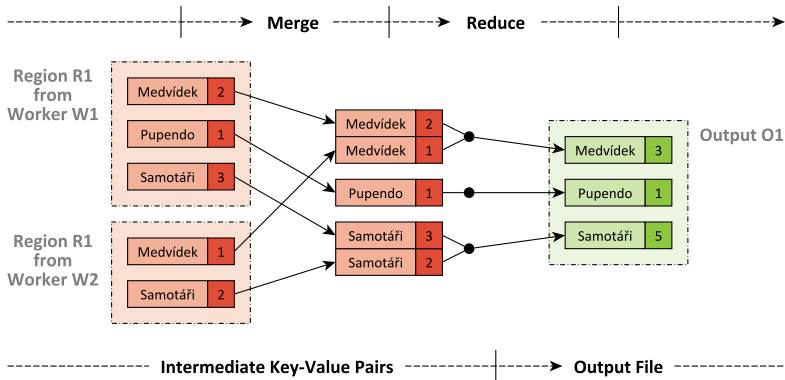
Improved Map Phase



Improved Reduce Phase



Improved Reduce Phase



Functions Overview

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

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

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

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 `...` 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

Reverse web-link graph

```
/**
 * 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 (year 2016)

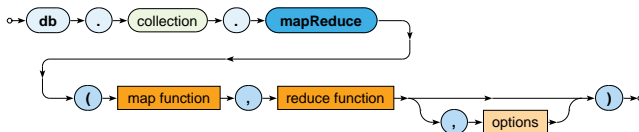
- **Facebook** – internal logs, analytics, machine learning, 2 clusters
1100 nodes (8 cores, 12 TB storage), 12 PB
300 nodes (8 cores, 12 TB storage), 3 PB
- **LinkedIn** – 3 clusters
800 nodes (2×4 cores, 24 GB RAM, 6×2 TB SATA), 9 PB
1900 nodes (2×6 cores, 24 GB RAM, 6×2 TB SATA), 22 PB
1400 nodes (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

MongoDB MapReduce



MapReduce

Executes a **MapReduce** job on a selected collection



- Parameters
 - **Map**: JavaScript implementation of the Map function
 - **Reduce**: JavaScript implementation of the Reduce function
 - **Options**

MapReduce

Map function

- Current document is accessible via `this`
- `emit(key, value)` is used for emissions

Reduce function

- Intermediate key and values are provided as arguments
- Reduced value is published via `return`

Options

- `query`: only matching documents are considered
- `sort`: they are processed in a specific order
- `limit`: at most a given number of them is processed
- `out`: output is stored into a given collection

MapReduce: Example

Count the number of movies filmed in each year, starting in *2005*

```
db.movies.mapReduce(  
  function() {  
    emit(this.year, 1);  
  },  
  function(key, values) {  
    return Array.sum(values);  
  },  
  {  
    query: { year: { $gte: 2005 } },  
    sort: { year: 1 },  
    out: "statistics"  
  }  
)
```


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?