B4M36DS2, BE4M36DS2: **Database Systems 2**

https://cw.fel.cvut.cz/b251/courses/b4m36ds2

Lecture 12

MapReduce, Apache Hadoop

Yuliia Prokop

prokoyul@fel.cvut.cz

13. 10. 2025

Author: Martin Svoboda (martin.svoboda@matfyz.cuni.cz)

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 the physical reality of the internal **internal implementation**
  - Allows us to work at a (much) **higher level of abstraction**
- The point is
  <u>how the intended user thinks to solve their tasks</u> 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 2004

> *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 distributed systems**.

Alternatives: Apache Spark, Apache Flink, Google Dataflow, Dask/Ray

History and Motivation

Google PageRank problem (2003) - one of the early implementations, now superseded by more sophisticated ranking algorithms

- How to rank tens of billions of web pages by their importance
  - … efficiently in a reasonable amount of time
  - … when data is scattered across hundreds of thousands of computers
  - … data files can be enormous (petabytes 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**
- Fault tolerance: **automatic failure handling**

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,
  - container orchestration
  - cloud resource management
  - data security and encryption

  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
- $(key, value) \rightarrow$ list of $(key, value)$

Reduce function

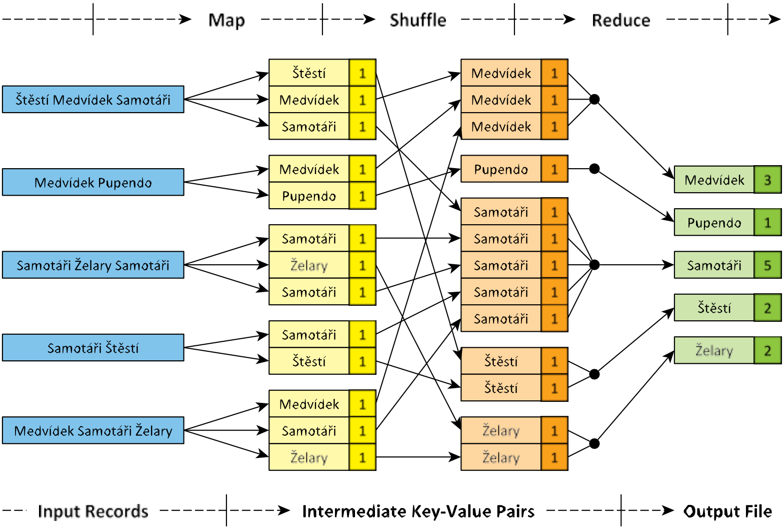
- *Input*: **intermediate key + list of (all) values** for this key
- *Output*: **possibly smaller list of values** for this key
  - Usually from the same domain
- $(key,$ list of $values) \rightarrow (key,$ list of $values)$

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

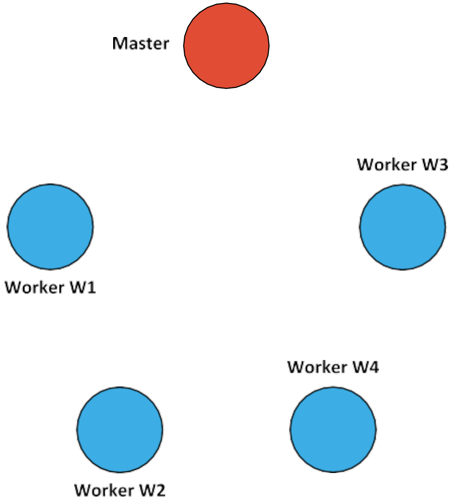
- **Reduce function** is executed **for each intermediate key**
- Output key-value pairs are generated

Cluster Architecture

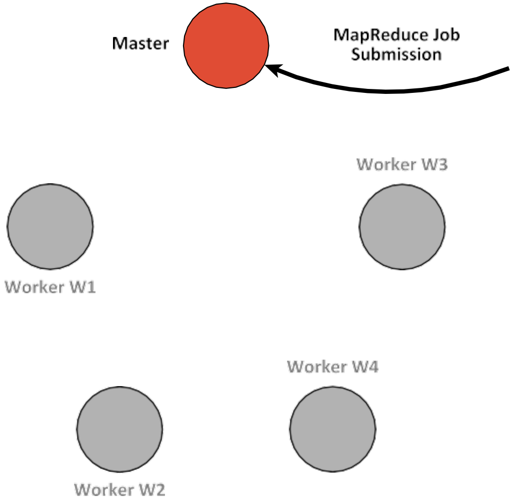
Master-slave (coordinator-worker or manager-worker) **architecture**

- 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

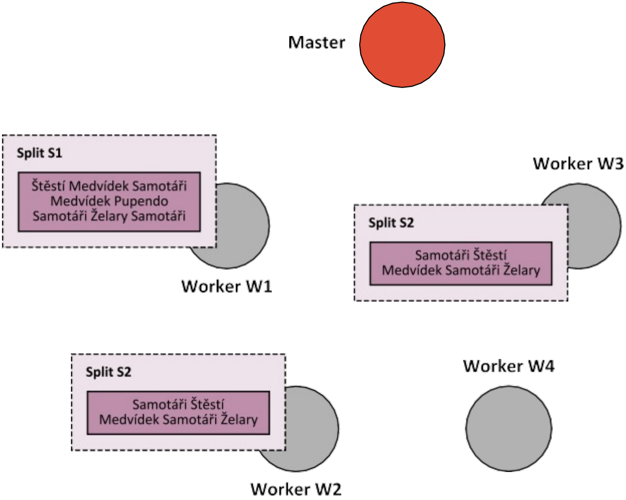
Submission of MapReduce jobs

- 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**

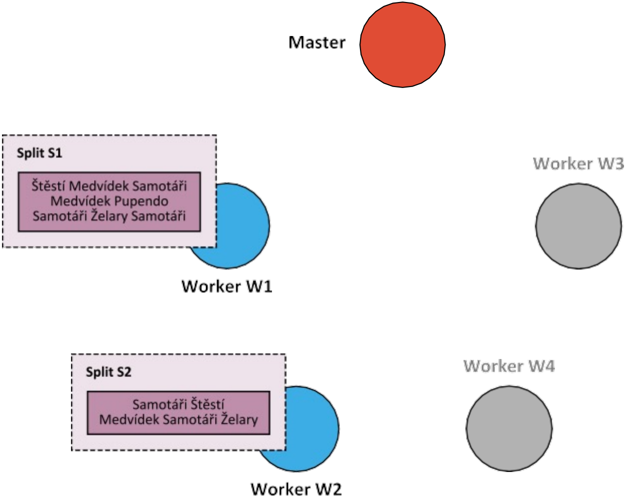
Localization of input files

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

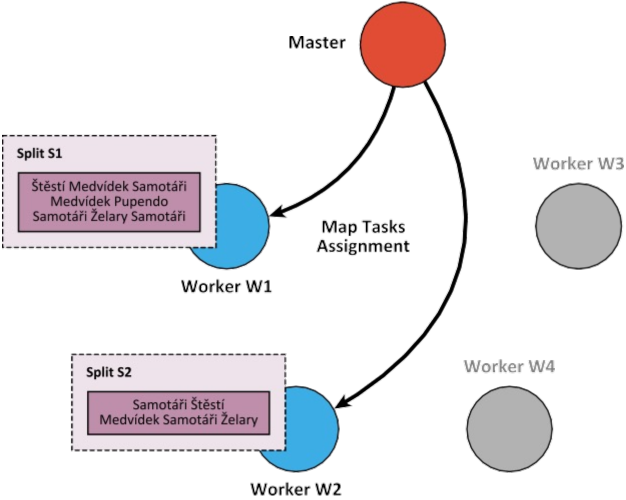
Input Splits Localization



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

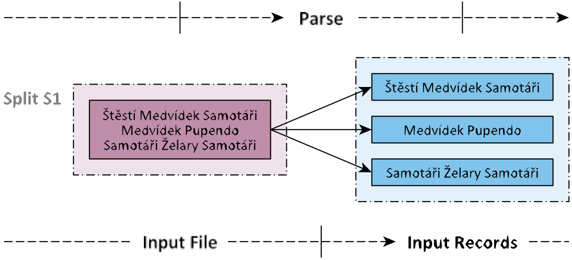
Individual steps…

- Input reader is used to **parse the 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 <u>locally</u> and organized into** regions
  - Either in memory with monitoring & adaptive spilling or compressed and flushed to a local hard drive when necessary
  - **Partition function** is used to determine the intended region
    - Intermediate <u>keys</u> (not values) are used
    - E.g. hash of the key modulo the overall number of reducers

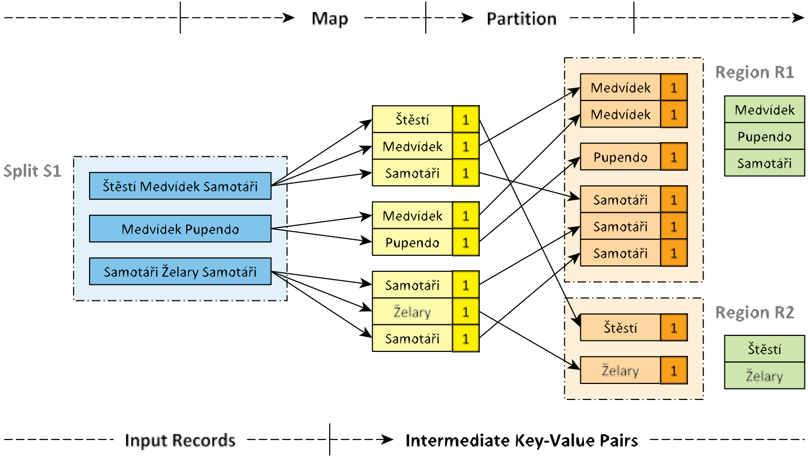
Input Parsing

Parsing phase

- **Each split is parsed** so that **input records are retrieved** (i.e. input key-value pairs are obtained)
  - Schema validation and type inference
  - Error handling for malformed data
  - Support for compressed/encoded formats

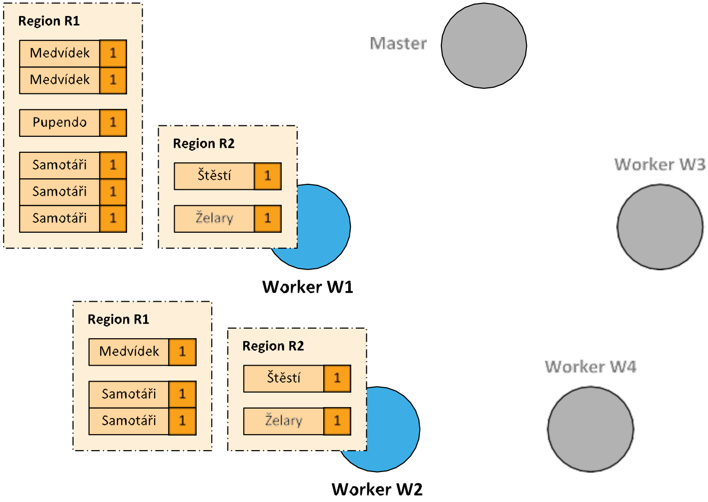


Map Phase



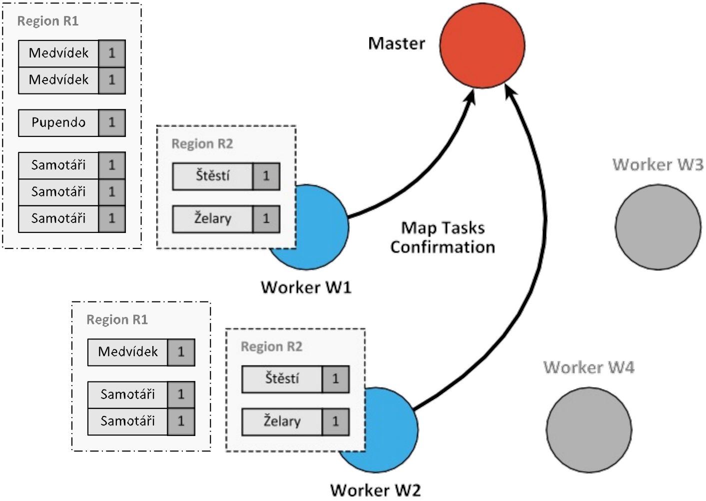
Map
Partition
Region R1
Medvídek 1
Medvídek 1
Pupendo 1
Samotáři 1
Samotáři 1
Samotáři 1
Medvídek
Pupendo
Samotáři
Split S1
Štěstí Medvídek Samotáři
Medvídek Pupendo
Samotáři Želary Samotáři
Štěstí 1
Medvídek 1
Samotáři 1
Medvídek 1
Pupendo 1
Samotáři 1
Želary 1
Samotáři 1
Region R2
Štěstí 1
Želary 1
Štěstí
Želary
Input Records
Intermediate Key-Value Pairs

Map Phase

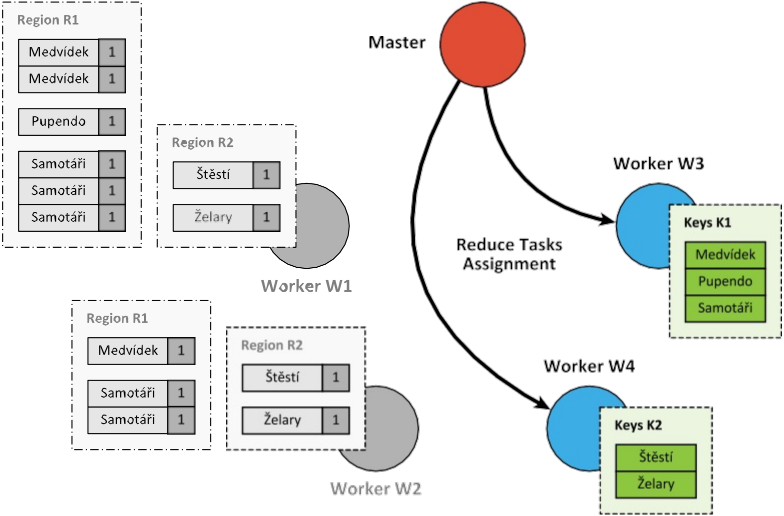


Region R1
Medvídek 1
Medvídek 1
Pupendo 1
Samotáři 1
Samotáři 1
Samotáři 1
Region R2
Štěstí 1
Želary 1
Master
Worker W3
Worker W1
Region R1
Medvídek 1
Samotáři 1
Samotáři 1
Region R2
Štěstí 1
Želary 1
Worker W4
Worker W2

Map Task Confirmation



Reduce Task Assignment



Reduce Task Execution

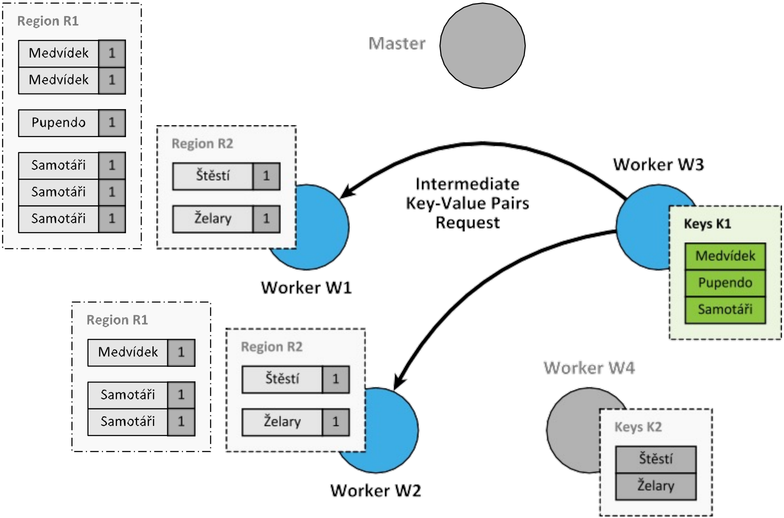
Reduce Task = reduction of selected key-value pairs by 1 worker

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

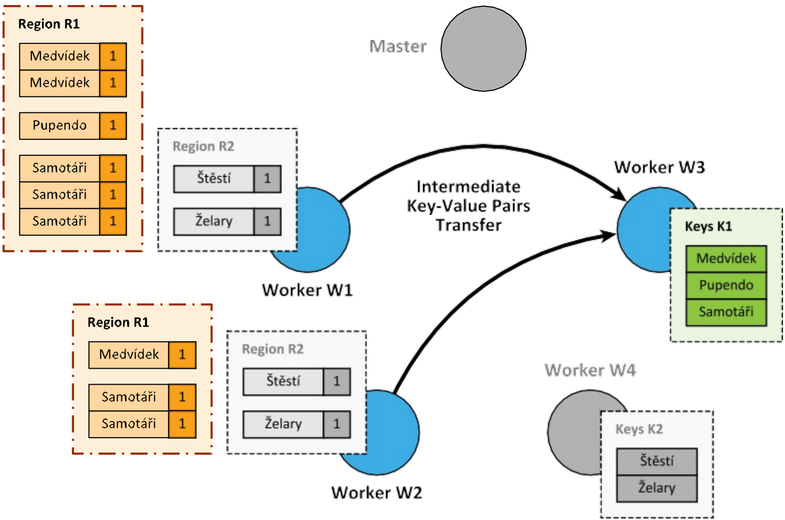
Individual steps…

- **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
  - External merge sort for large datasets
- **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

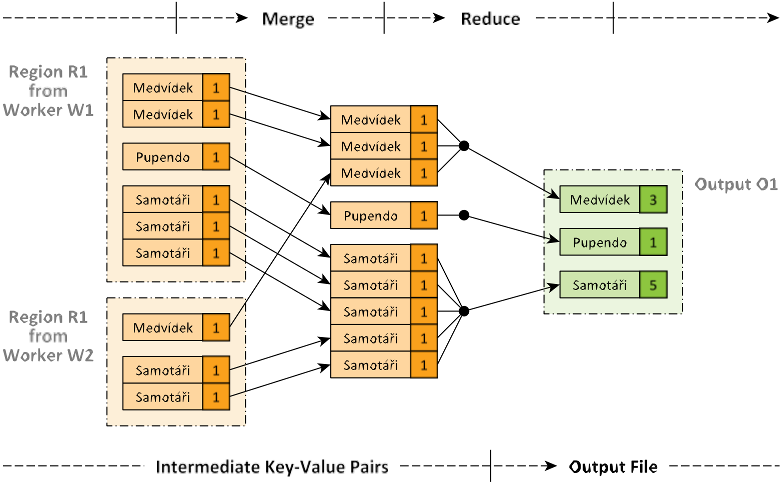
Region Data Retrieval



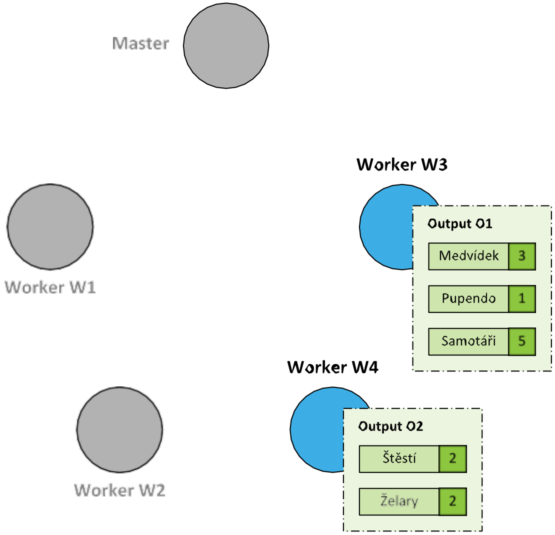
Region Data Retrieval



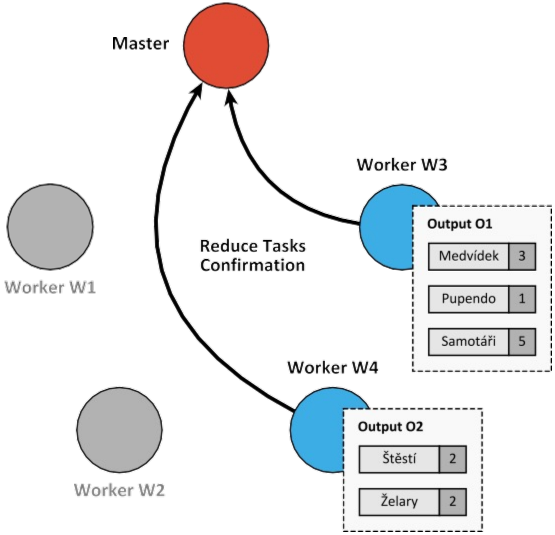
Reduce Phase



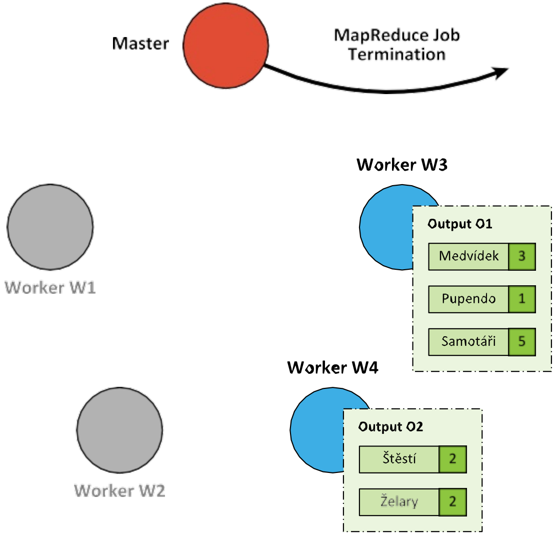
Reduce Phase



Reduce Task Confirmation



MapReduce Job Termination

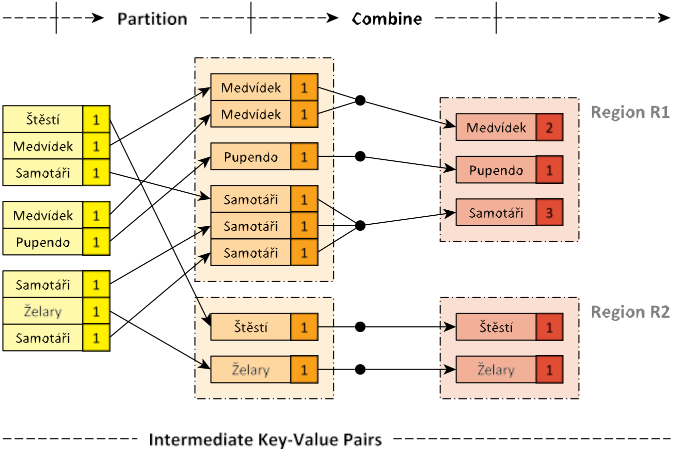


Combine Function

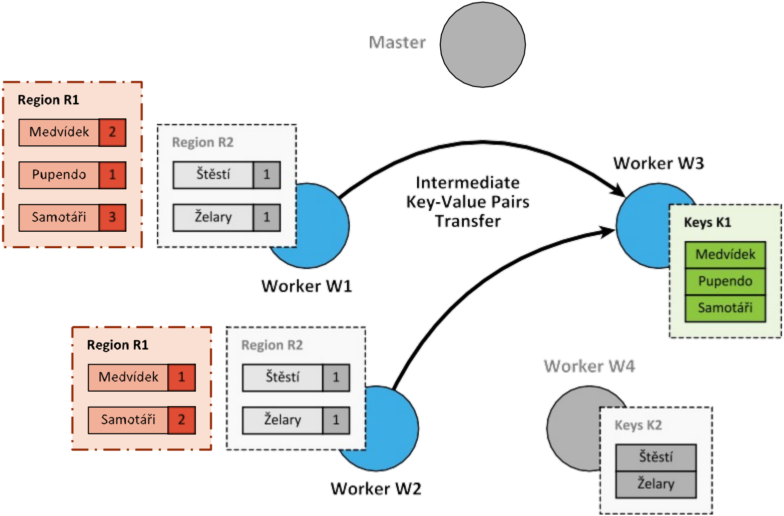
Optional **Combine function**

- Objective
  - **Decrease the amount of intermediate data**
    i.e. decrease the amount of data that is needed to be transferred from Mappers to Reducers
  - Optimize network and storage usage
- Analogous purpose and implementation to **Reduce function**
- **Executed locally by Mappers**
- However, only applicable when the reduction is…
  - **Commutative**
  - **Associative**
  - **Idempotent**: $f(f(x)) = f(x)$
  - Memory efficient
  - Cost-effective vs. raw transfer

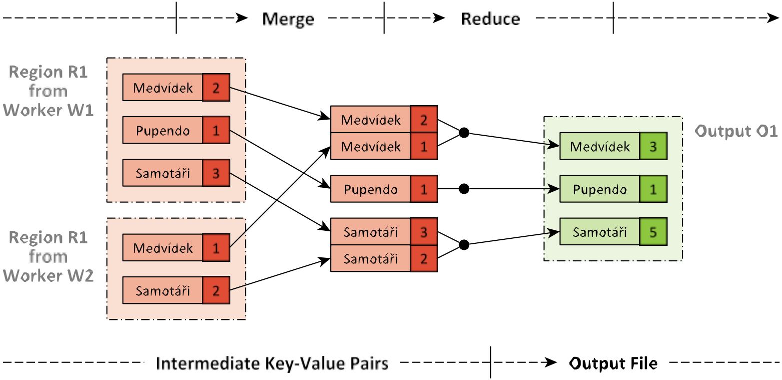
Improved Map Phase



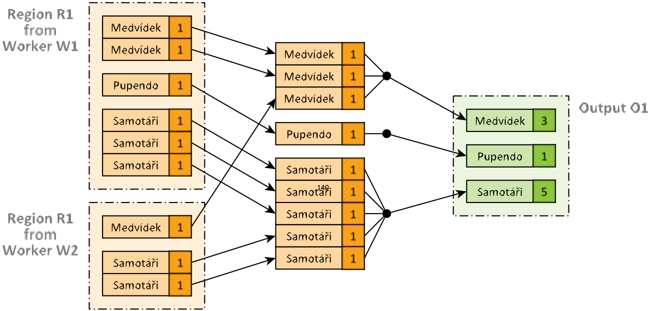
Improved Reduce Phase



Improved Reduce Phase



Data between Map and Reduce Phases



Medvídek, {1, 1, 1}
Pupendo, {1}
Samotáři, {1, 1, 1, 1, 1}

Key

List of values

Reduce Phase

In MapReduce, a **Reducer** processes all values associated with a **single key**. These values are represented as an **Iterable** in Java, not a collection.

An **Iterable** is an interface that allows objects to be the target of the "for-each" loop. It doesn't guarantee the ability to iterate over its elements multiple times like collections do. This means you can only iterate over an Iterable once.

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

Functions Overview

Input reader

- Parses a given input split and prepares input records

Map function

- Transforms input records into intermediate key-value pairs

Partition function

- Determines a particular Reduce**r** for a given intermediate key

Compare function

- Defines ordering between intermediate keys for sorting

Combine function

- Local pre-reduction on Mapper side (optional))

Reduce function

- Processes grouped values per intermediate key

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
    - Support monitoring and error tracking

Advanced Aspects

Fault tolerance

- When a large number of nodes process a large number of data $\Rightarrow$ **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**
- Detection via modern health checks and heartbeat mechanisms

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

Processing Big Data (Log Analysis)

Example: Counting the number of requests for each URL.

- **Task**: Count how many times each URL appears in server logs.
- **Input Data**: Log files (text lines):

```
192.168.1.1 - [10/Jun/2024] "GET /index.html"
192.168.1.2 - [10/Jun/2024] "GET /about.html"
192.168.1.1 - [10/Jun/2024] "GET /index.html"
```

Map Phase

- **Processing**: Extract the URL and emit **(URL, 1)**.
- **Map Output**: Key-value pairs:

```
("/index.html", 1)
("/about.html", 1)
("/index.html", 1)
```

Shuffle and Sort

- Grouping by key:

```
"/index.html" → [1, 1]
"/about.html" → [1]
```

Reduce Phase

- **Processing**: Sum all 1s for each key.
- **Reduce Output**: Final results:

```
("/index.html", 2)
("/about.html", 1)
```

Indexing Data (Search Engines)

Example: Building an inverted index that links words to document IDs.

- **Task**: Link words in documents to the documents where they appear.
- **Input Data**: Text documents:

```
doc1: "hello world"
doc2: "hello hadoop"
```

Map Phase

- **Processing**: Emit (word, document ID)
- **Map Output**:

```
("hello", "doc1")
("world", "doc1")
("hello", "doc2")
("hadoop", "doc2")
```

Shuffle and Sort

- Grouping by word:

```
"hello" → ["doc1", "doc2"]
"world" → ["doc1"]
"hadoop" → ["doc2"]
```

Reduce Phase

- **Processing**: Combine document IDs for each word into a list.
- **Reduce Output**:

```
("hello", ["doc1", "doc2"])
("world", ["doc1"])
("hadoop", ["doc2"])
```

Analytics and Business Reports

Example: Summing sales revenue by region.
- **Task**: Calculate total revenue per region.
- **Input Data**: Sales table:

```
region, amount
North, 100
South, 200
North, 300
```

Map Phase
- **Processing**: Emit (region, amount).
- **Map Output**:

```
("North", 100)
("South", 200)
("North", 300)
```

Shuffle and Sort
- Grouping by region:

```
"North" → [100, 300]
"South" → [200]
```

Reduce Phase
- **Processing**: Sum all amounts for each region.
- **Reduce Output**:

```
("North", 400)
("South", 200)
```

ETL Processes (Data Cleaning)

Example: Removing invalid records.
- **Task**: Filter out records with missing or invalid values.
- **Input Data**:

```
user, age
John, 25
Alice, null
Bob, 30
```

Map Phase
- **Processing**: Validate the data and emit valid records.
- **Map Output**:

```
("John", 25)
("Bob", 30)
```

Shuffle and Sort
- Grouping (optional):

Reduce Phase
- **Processing**: No aggregation required; output cleaned data.
- **Reduce Output**:

```
("John", 25)
("Bob", 30)
```

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 links (<a href>, JSON-LD, structured data)
- *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

The page http://page1.com contains:

```
<a href="http://target.com">Link</a>
<a href="http://other.com">Other</a>
```

Map output:

```
(http://target.com, http://page1.com)
(http://other.com, http://page1.com)
```

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);
}
```

```
http://target.com -> [http://page1.com, http://page2.com, http://page3.com]
```

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
- Fraudulent activity threats, security breaches detection
- Hardware or sensor network failure prediction
- Search quality analysis
- IoT data processing and analytics
- Real-time anomaly detection
- User behavior analytics
- Supply chain optimization

Source: http://www.cloudera.com/

Problems with MapReduce

- **High Disk I/O Costs**
  - Intermediate results are written to and read from disk.
- **Slow Execution for Iterative Jobs**
  - No in-memory processing; repetitive disk writes slow down machine learning and graph tasks.
- **Lack of Real-Time Support**
  - Designed for batch processing with high latency.
- **Rigid Programming Model**
  - Only Map and Reduce phases limit flexibility for complex workflows.
- **Complex Multi-Stage Workflows**
  - Chaining multiple jobs is cumbersome and inefficient.

Modern Alternatives to MapReduce

- **Apache Spark**
  - **Key Idea:** In-memory processing using Resilient Distributed Datasets (RDDs).
  - **Improvement:** Avoids disk I/O by storing intermediate results in RAM.
- **Apache Flink**
  - **Key Idea:** Real-time stream and batch processing with stateful computations.
  - **Improvement:** Designed for low-latency tasks; supports time windows and state management.
- **Google Dataflow / Apache Beam**
  - **Key Idea:** Unified model for batch and streaming data processing.
  - **Improvement:** Provides flexibility while abstracting execution details.
- **Dask and Ray**
  - **Key Idea:** Python-native frameworks for distributed parallel processing.
  - **Improvement:** Simplifies big data processing for Python developers.

The Core Concept: MapReduce Lives On

- **Map Phase**
  - All systems (Spark, Flink, Beam) retain the concept of parallel data transformation.
- **Shuffle Phase**
  - Data is grouped or partitioned by key, similar to MapReduce.
- **Reduce Phase**
  - Aggregation and summarization are still core principles.

What's Different?

- **In-Memory Execution**: Avoids repeated disk I/O.
- **Flexible Workflows**: DAG execution models enable complex chains.
- **Real-Time Streaming**: Frameworks like Flink and Beam handle data in motion.

Conclusion: MapReduce principles persist, but modern systems optimize execution, flexibility, and speed.

Apache Hadoop

Apache Hadoop

<u>Open-source</u> software framework

- https://hadoop.apache.org/
- **Distributed storage and processing** of very large data sets on clusters built from commodity hardware and cloud infrastructure
  - Implements a **distributed file system** (HDFS)
  - Implements a **MapReduce** programming model
- Part of the Hadoop ecosystem (YARN, HDFS, etc.)
- Derived from the original Google MapReduce and GFS
- Developed by Apache Software Foundation
- Implemented in Java with support for multiple programming languages
- 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
  - Container and GPU support
- Hadoop **MapReduce**
  - YARN-based implementation of the MapReduce model

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

Source: http://wiki.apache.org/hadoop/PoweredBy

HDFS

Hadoop **Distributed File System**

- Open-source, high-quality, cross-platform, pure Java
- **Highly scalable, high-throughput, fault-tolerant, erasure coding**
- Master-Slave (Primary-Secondary) architecture
- Optimal applications
    - Data lakes, MapReduce, web crawlers, data warehouses, AI/ML pipelines, …

HDFS: Assumptions

Data characteristics

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

Fault tolerance

- HDFS cluster may consist of thousands of nodes
  - Each component has a non-trivial probability of failure
- ⇒ <u>there is always some component that is non-functional</u>
  - 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
  - The default block size is typically **128 MB** (configurable per file or globally)
- 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
- Typically, a single namespace for the entire cluster, but HDFS Federation supports multiple namespaces for scalability

HDFS: Cluster Architecture

Master-slave architecture

- Master (Primary):
  - **NameNode**
  - Manages the **namespace**
  - Maintains **physical locations of file 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
- Slaves (Secondary): **DataNodes**
  - **Physically store file blocks** within their underlying file systems
  - **Serve read/write requests** from users
    - I.e. user data never flows through the NameNode
  - Have no knowledge about the namespace

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 (16 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**
  - Language bindings: Python, Go, C/C++
- **HTTP interface**
  - Browsing the namespace and downloading the contents of files
  - WebHDFS RESTful API
- 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**

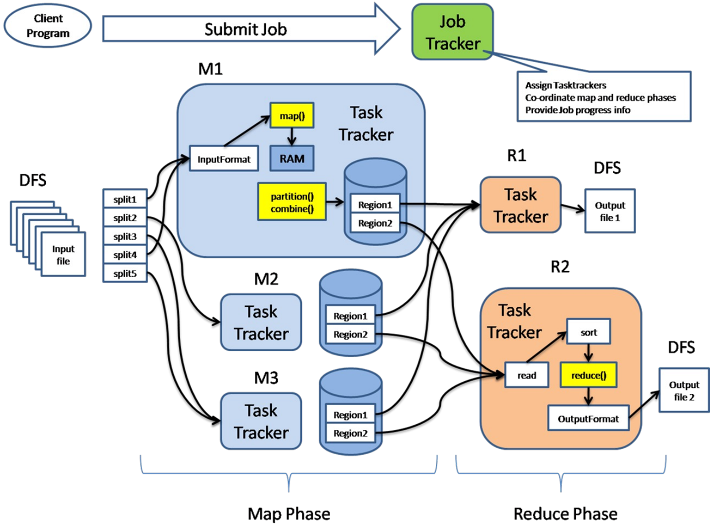
- MapReduce programming model implementation
- Requirements
  - **HDFS**
    - Input and output files for MapReduce jobs
  - **YARN**
    - Underlying distribution, coordination, monitoring and gathering of the results

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**
  - Manages containers for 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?