Acknowledgements

I am indebted to Prof. Dr.-Ing. Sebastian Michel, Prof. Johan Gamper, and Dr. Holubova for providing me slides.
The Elwedritsch is a cryptid or mythical creature that supposedly inhabits the Palatinate of Germany. It is described as being a chicken-like creature with antlers. It also has scales instead of feathers. However, it is said that their wings are of little use. That is why they live mainly in underbrush and under vines. Sometimes Elwetritschen are depicted with antlers of a stag and their beaks often appear to be very long. In the second half of the 20th century, artists increasingly portrayed Elwetritschen as female by adding breasts. Elwetritschen supposedly originate from crossbreeding chickens, ducks, and geese with mythical wood creatures such as goblins and elves. Being a fowl, they naturally lay eggs, which as a result of descending from forest spirits, grow during breeding season. Eggs in various sizes are artistically depicted at the Elwetritschenbrunnen in Neustadt an der Weinstraße. Geographical Distribution: The area in which tales of the Elwetritsch are spread expands from the Palatinate Forest in the west of Germany towards the east across the Upper Rhine Plain to the southern parts of the Odenwald. The mythical creature also appears in the north of Baden-Württemberg. In the Main-Tauber-Kreis, where they are known as “Ilwedridsche”, the children are told that at night the creatures sleep in the crowns of the willow trees standing next to the river Tauber. In Neustadt an der Weinstraße, which is said to be the “capital” of the Elwetritsches, there is an Elwetritsche-fountain, created by Gernot Rumpf. Other sources consider Dahn in the southwestern Palatinate, which also has an Elwetritsche-fountain, Erfweiler or other villages as secret capitals of these creatures. The idea is very similar to the "snipe hunt." The Elwetritsch is supposedly very shy, but also very curious. A hunting party consists of a "Fänger" (catcher), equipped with a big potato sack and a lantern, and the "Treiber" (beaters). The catcher is led into the woods where the Elwetritsch is supposed to live, instructed to wait in a clearing with his sack and lantern, while the beaters go off, supposedly to flush out the Elwetritsch. The light of the lantern is said to be attractive to the curious creature, so it will come to investigate and will then be caught by the catcher. While he waits, everyone heads back to the pub or wherever the party had previously assembled, to wait for the catcher to realize he has been fooled.

Imagine this file is several TB or PB in size!
The Elwedritsch is a cryptid or mythical creature that supposedly inhabits the Palatinate of Germany. It is described as being a chicken-like creature with antlers. It also has scales instead of feathers. However, it is said that their wings are of little use. That is why they live mainly in underbrush and under vines. Sometimes Elwetriritschen are depicted with antlers of a stag and their beaks often appear to be very long. In the second half of the 20th century, artists increasingly portrayed Elwetriritschen as female by adding breasts. Elwetriritschen supposedly originate from crossbreeding chickens, ducks, and geese with mythical wood creatures such as goblins and elves. Being a fowl, they naturally lay eggs, which as a result of descending from forest spirits, grow during breeding season. Eggs in various sizes are artistically depicted at the Elwetriritschenbrunnen in Neustadt an der Weinstraße. Geographical Distribution: The area in which tales of the Elwetriritsch are spread expands from the Palatinate Forest in the west of Germany towards the east across the Upper Rhine Plain to the southern parts of the Odenwald. The mythical creature also appears in the north of Baden-Württemberg. In the Main-Tauber-Kreis, where they are known as “Ilwedridsche”, the children are told that at night the creatures sleep in the crowns of the willow trees standing next to the river Tauber. In Neustadt an der Weinstraße, which is said to be the “capital” of the Elwetriritsches, there is an Elwetriritsche-fountain, created by Gernot Rumpf. Other sources consider Dahn in the southwestern Palatinate, which also has an Elwetriritsche-fountain, Erfweiler or other villages as secret capitals of these creatures. The idea is very similar to the "snipe hunt." The Elwetriritsch is supposedly very shy, but also very curious. A hunting party consists of a "Fänger" (catcher), equipped with a big potato sack and a lantern, and the "Treiber" (beaters). The catcher is led into the woods where the Elwetriritsch is supposed to live, instructed to wait in a clearing with his sack and lantern, while the beaters go off, supposedly to flush out the Elwetriritsch. The light of the lantern is said to be attractive to the curious creature, so it will come to investigate and will then be caught by the catcher. While he waits, everyone heads back to the pub or wherever the party had previously assembled, to wait for the catcher to realize he has been fooled.
MR: Scale-out Architecture

- Many machines (hundreds, thousands)
- Data is spread across machines
- Processing tasks initiated (ideally) where data resides
Screenshot of HDFS (Hadoop/MR) Filesystem UI. Showing info on a large file of Twitter tweets/updates, stored in 509 blocks (chunks) over several machines.
Map and Reduce: Key Idea

- Spread task of processing data on machines
- According to map and reduce rules/functions
- No need to deal with node failures, load balancing, etc. system takes care of this.
- Divide-and-conquer paradigm
  - **Map** breaks down a problem into sub-problems
    - Processes input data to generate a set of intermediate key/value pairs
    - map 'length' [ [] [a] [a,b] [a,b,c] ] ⇒ [0,1,2,3]
  - **Reduce** receives and combines the sub-solutions to solve the problem
    - Processes intermediate values associated with the same intermediate key
    - Combines all the values using a binary function (e.g., +)
    - reduce '+' [1,2,3,4,5] ⇒ 15
Map Reduce from High Level

Intermediate Results
Brief History of MapReduce

• First described in an article in 2004.
  – MapReduce paradigm and how it is used in Google (Google file system, etc.)

Jeffrey Dean, Sanjay Ghemawat: MapReduce: Simplified Data Processing on Large Clusters. OSDI 2004: 137-150

• Many MapReduce (MR) implementations
• Hadoop is arguably the most prominent one
• Will look at MR in general and Hadoop specifically
Architectural Issues

• Data lies in a *distributed file system*
• Block based, **big chunks** (usually **64MB** or **128MB**)
• Chunks are **replicated and distributed** over machines

• If possible, **data processing is moved to data hosting machines.**
MapReduce
A Bit More Formally

Map
- Input: a key/value pair
- Output: a set of intermediate key/value pairs
  - Usually different domain
  - \((k_1, v_1) \rightarrow \text{list}(k_2, v_2)\)

Reduce
- Input: an intermediate key and a set of values for that key
- Output: a possibly smaller set of values
  - The same domain
  - \((k_2, \text{list}(v_2)) \rightarrow (k_2, \text{possibly smaller list}(v_2))\)
General Scheme

Input files
Map phase
Intermediate files (on local disk)
Reduce phase
Output files
MapReduce
Example: Word Frequency

```java
map(String key, String value):
    // key: document name
    // value: document contents
    for each word w in value:
        EmitIntermediate(w, "1");
```

```java
reduce(String key, Iterator values):
    // key: a word
    // values: a list of counts
    int result = 0;
    for each v in values:
        result += ParseInt(v);
    Emit(key, AsString(result));
```
MapReduce
Example: Word Frequency
MapReduce
More Examples

- **distributed grep**
  - Map: emits <word, line number> if it matches a supplied pattern
  - Reduce: identity

- **URL access frequency**
  - Map: processes web logs, emits <URL, 1>
  - Reduce: sums values and emits <URL, sum>

- **reverse web-link graph**
  - Map: <target, source> for each link to a target URL found in a page named source
  - Reduce: concatenates the list of all source URLs associated with a given target URL <target, list(source)>
MapReduce
More Examples

- **term vector per host**
  - “Term vector” summarizes the most important words that occur in a document or a set of documents
  - Map: emits <hostname, term vector> for each input document
    - The hostname is extracted from the URL of the document
  - Reduce: adds the term vectors together, throws away infrequent terms

- **inverted index**
  - Map: parses each document, emits <word, document ID>
  - Reduce: sorts the corresponding document IDs, emits <word, list(document ID)>

- **distributed sort**
  - Map: extracts the key from each record, and emits <key, record>
  - Reduce: emits all pairs unchanged
MapReduce – Map

Map
- generates key-value pairs as intermediate results

Example: Word counting

```
map(documentName, value) {
    for each word w in value
        emit(w, "1");
}
```

- Emits a value pair for each word, consisting of the word itself and one occurrence
- Multiple occurrence result in multiple pairs

Note: Input (documentName, value) is also a key-value pair, but for a different domain than the emitted pairs
Reduce

- Aggregates intermediate results

**Example: Word counting (cont’d)**

```java
reduce(word, values) {
    int result = 0;
    for each w in values
        result += w;
    emit(word, result);
}
```

- Sums up occurrences and emits result

- List of word occurrences: Generated by the framework by collecting and grouping intermediate results with same key (shuffle)
- Provided to Reduce operation as iterator, which allows handling also very large datasets

- Note: Input pairs of Reduce are of the same domain as output pairs of Map (Relaxed in Yahoo’s Hadoop)
- Multiple reduce workers can be called on same key
  - But with different values
  - Results will be intermediate results that have to go through the reduce step again
Implementation (1)

Input files  | Map phase  | Intermediate files (on local disk) | Reduce phase  | Output files
split 0      | (3) read   | (5) remote read                   |               | output file 0
split 1      | (1) fork   | (4) local write                   |               | output file 1
split 2      |             | (2) assign map                     |               |
split 3      |             | (2) assign reduce                  |               |
split 4      |             | (1) fork                           |               |
Input data is split into M disjunctive partitions for the parallel execution of Map operation

About 16–64 MB per partition (controllable by parameter)

Split can be done on parallel, too
Implementation (3)

- Copies of program is started on machines of the cluster (1)
- One instance is appointed to the master, the remaining instances become workers
Implementation (4)

- Master assigns map and reduce tasks to running worker instance (2) (considers locality)
  - M map tasks
  - R reduce tasks
Implementation (5)

- Worker with map task
- Reads assigned partition (3)
- Processes all input pairs with map function
- Buffers output pairs in local main memory
- Flushes buffer periodically to disk (4)
  - Separated into R partitions based on partitioning function (e.g. hash(key) mod R)
  - Storage location is reported to the master, who coordinates hand-over to reducers
## Implementation (6)

- **Worker with reduce task**
- Informed by master
- Gets location of intermediate results and reads them (5)
- **Shuffle:** sorting pairs by key to group them
  - Worker iterates over sorted pairs
  - Process each key together with all its values with the reduce function
- **Write result of reduce function into the output file, that is associate with reducer’s input partition (6)**
Implementation (7)

- Returns to user program after all map and reduce task have been processed completely
- Output consists of R files (one per reducer task)
- Could be used as input for another map reduce program
Fault Tolerance

**Worker**
- Get ping periodically if everything is okay
- No answer implies crash
- Assigned tasks get reassigned by master
- Also successfully finished map tasks get reassigned, since output resides on local disk
- Worker with reduce task get informed

**Master**
- Periodically backups its state (check pointing)
- In case of crash: Restart and recovery of last check point

*Fault Tolerance is main advantage of MapReduce!*
- Parallelization is just one concern
Distributed File System

How does the data get to the workers?
- Do not move data to the workers!
- Move workers to the data!
- Data vs. program code: Ship what is smaller! → program code
- Data is stored on local disk of cluster nodes
- Start workers on node that have data locally available

Data stored in distributed file system
- GFS (Google File System) for Google’s MapReduce
- HDFS (Hadoop Distributed File System) for Hadoop

Google's MapReduce
- Map write intermediate results to local file system
- Reduce gets data via RPC and write results to GFS.
MR Example (1)

Weather data

1. Reading unstructured weather data
   - 0029029070999991901010106004+64333+023450FM-12+00059999V0202701N015919999999N0000001N9-00781+99999102001ADDGF108991999999999999999999
   - 00290290709999990101010106004+64333+023450FM-12+00059999V0202701N015919999999N0000001N9-00781+99999102001ADDGF108991999999999999999999
   - 00290290709999991901010113004+64333+023450FM-12+00059999V0202701N015919999999N0000001N9-00781+99999102001ADDGF108991999999999999999999

   1901-01-01
   13:00
   -7,2°C
MR Example (2)

*Weather data*

2. Tokenize file into records
   - Parser gives byte position of the individual records
   - Records can then be distributed to multiple map tasks
   - $(k1, v1) = (\text{long}, \text{String})$
3. Map: Transforms records into intermediate result
- Required data is extracted from the records
- Results in many key-value pairs

<table>
<thead>
<tr>
<th>Year</th>
<th>Temperatur</th>
</tr>
</thead>
<tbody>
<tr>
<td>1950</td>
<td>0</td>
</tr>
<tr>
<td>1950</td>
<td>22</td>
</tr>
<tr>
<td>1949</td>
<td>111</td>
</tr>
<tr>
<td>1949</td>
<td>78</td>
</tr>
</tbody>
</table>
MR Example (4)

Weather data

4. Shuffle: Groups intermediate key-value pairs

- Sorts by key
- Collects values into lists
- Each Mapper write sorted output into the file system
- For each year, one reduced task is executed on the cluster

<table>
<thead>
<tr>
<th>Year</th>
<th>Temperatur</th>
</tr>
</thead>
<tbody>
<tr>
<td>1949</td>
<td>111</td>
</tr>
<tr>
<td></td>
<td>78</td>
</tr>
<tr>
<td>1950</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>22</td>
</tr>
</tbody>
</table>
5. Reduce: Aggregate value per key

- In this example maximum aggregation
- One value per key remains

<table>
<thead>
<tr>
<th>Year</th>
<th>Temperatur</th>
</tr>
</thead>
<tbody>
<tr>
<td>1949</td>
<td>111</td>
</tr>
<tr>
<td>1950</td>
<td>22</td>
</tr>
</tbody>
</table>
MR Example (6)

Weather data

6. Output into file
Summary
Application Parts

- **Input reader**
  - Reads data from stable storage
    - e.g., a distributed file system
  - Divides the input into appropriate size 'splits'
  - Prepares key/value pairs

- **Map function**
  - User-specified processing of key/value pairs

- **Partition function**
  - Map function output is allocated to a reducer
  - Partition function is given the key (output of Map) and the number of reducers and returns the index of the desired reducer
    - Default is to hash the key and use the hash value modulo the number of reducers
Summary
Application Parts

- **Compare function**
  - Sorts the input for the Reduce function

- **Reduce function**
  - *User-specified* processing of key/values

- **Output writer**
  - Writes the output of the Reduce function to stable storage
    - e.g., a distributed file system
MapReduce
Execution (Google) – Step 1

1. MapReduce library in the user program splits the input files into $M$ pieces
   - Typically 16 – 64 MB per piece
   - Controllable by the user via optional parameter

2. It starts copies of the program on a cluster of machines
MapReduce
Execution – Step 2

- **Master** = a special copy of the program
- **Workers** = other copies that are assigned work by master
- **M** Map tasks and **R** Reduce tasks to assign
- Master picks **idle** workers and assigns each one a Map task (or a Reduce task)
MapReduce
Execution – Step 3

- A worker who is assigned a Map task:
  - Reads the contents of the corresponding input split
  - Parses key/value pairs out of the input data
  - Passes each pair to the user-defined Map function
  - Intermediate key/value pairs produced by the Map function are buffered in memory
MapReduce
Execution – Step 4

- Periodically, the buffered pairs are written to local disk
  - Partitioned into $R$ regions by the partitioning function
- Locations of the buffered pairs on the local disk are passed back to the master
  - It is responsible for forwarding the locations to the Reduce workers
The diagram illustrates the Hadoop Distributed File System (HDFS) workflow, which is a key component of the MapReduce framework used in distributed computing.

1. **User Program**: The user program initiates the process by creating a MapReduce job.
2. **Master Node**: The master node distributes the job to worker nodes.
3. **Map Phase**: Each worker node reads input files and applies the map function to them.
4. **Intermediate Files**: The map function outputs intermediate files that are stored on local disks.
5. **Reduce Phase**: Workers then read these intermediate files and apply the reduce function, with each worker node handling a subset of the data.
6. **Output Files**: The reduce function outputs the final results, which are written to output files.

The diagram uses arrows to indicate the flow of data and processes, with labels indicating the steps involved in the MapReduce workflow.
MapReduce
Execution – **Step 5**

- Reduce worker is notified by the master about data locations
- It uses **remote procedure calls** to read the buffered data from local disks of the Map workers
- When it has read all intermediate data, it sorts it by the intermediate keys
  - Typically many different keys map to the same Reduce task
  - If the amount of intermediate data is too large, an external sort is used
MapReduce
Execution – Step 6

- A Reduce worker iterates over the sorted intermediate data

- For each intermediate key encountered:
  - It passes the key and the corresponding set of intermediate values to the user's Reduce function
  - The output is appended to a final output file for this Reduce partition
MapReduce

Function combine

- After a map phase, the mapper transmits over the network the entire intermediate data file to the reducer
- Sometimes this file is highly compressible
- User can specify function combine
  - Like a reduce function
  - It is run by the mapper before passing the job to the reducer
    - Over local data
MapReduce

Counters

- Can be associated with any action that a mapper or a reducer does
  - In addition to default counters
    - e.g., the number of input and output key/value pairs processed
- User can watch the counters in real time to see the progress of a job
MapReduce
Fault Tolerance

- A large number of machines process a large number of data → fault tolerance is necessary

- Worker failure
  - Master pings every worker periodically
  - 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 → become eligible for scheduling on other workers
MapReduce
Fault Tolerance

- Master failure
  - Strategy A:
    - Master writes periodic checkpoints of the master data structures
    - If it dies, a new copy can be started from the last checkpointed state
  - Strategy B:
    - There is only a single master → its failure is unlikely
    - MapReduce computation is simply aborted if the master fails
    - Clients can check for this condition and retry the MapReduce operation if they desire
MapReduce

Stragglers

- **Straggler** = a machine that takes an unusually long time to complete one of the map/reduce tasks in the computation
  - Example: a machine with a bad disk

- **Solution:**
  - When a MapReduce operation is close to completion, the master schedules *backup executions* of the remaining in-progress tasks
  - A task is marked as completed whenever either the primary or the backup execution completes
MapReduce
Task Granularity

- $M$ pieces of Map phase and $R$ pieces of Reduce phase
  - Ideally both much larger than the number of worker machines
  - How to set them?
- Master makes $O(M + R)$ scheduling decisions
- Master keeps $O(M \times R)$ status information in memory
- For each Map/Reduce task: state (idle/in-progress/completed)
- For each non-idle task: identity of worker machine
- For each completed Map task: locations and sizes of the $R$ intermediate file regions
- $R$ is often constrained by users
  - The output of each Reduce task ends up in a separate output file
- Practical recommendation (Google):
  - Choose $M$ so that each individual task is roughly 16 – 64 MB of input data
  - Make $R$ a small multiple of the number of worker machines we expect to use
Real-World Example (Google)
Cluster Configuration

- 1,800 machines
- Each machine:
  - 2x 2GHz Intel Xeon processor
    - With Hyper-Threading enabled
  - 4GB memory
    - Approx. 1-1.5GB reserved by other tasks
  - 2x 160GB IDE disks
  - Gigabit Ethernet link
- Arranged in a two-level tree-shaped switched network with approximately 100-200 Gbps of aggregate bandwidth available at the root
Real-World Example 1

grep

- Search through approx. 1 terabyte of data looking for a particular pattern
  - Rare three-character pattern
  - Present in 92,337 records
- $M = 15,000 \ (10^{12}/64*10^6)$
- $R = 1$
- 1,764 workers assigned
- Entire computation: 150 seconds
  - About a minute of start-up overhead
Example: Grep

• Given: file
• Want: all lines that contain certain pattern

• Map(String key, String value)
  
  if value.contains(pattern):
  
      emit(value, "")

This is a **map only** task (no reducer; no grouping by key): output is written directly to distributed file system
Real World Example 2

sort

- Sorting of approx. 1 terabyte of data
- Map: 3-line function
  - Extracts a 10-byte sorting key from a text line and emits the key and the original text line
- Reduce: identity
- $M = 15,000$
- $R = 4,000$
- About 1,700 workers assigned
- Entire computation: 891 seconds
  - 5 stragglers increase the time of 44%
Real World Example 3

word count

- **Task:** Count # of occurrences of each word in a collection of documents
- **Input:** Large number of text documents
- **Output:** Word count across all the documents

- **Map:** Parse data and output \((word, "1")\) for every word in a document.
- **Reduce:** For each word, sum all occurrences and output \((word, total\ ct)\)

```java
map(String key, String value);
//key: document name
//value: document contents
foreach word w in value do
    EmitIntermediate(w, "1");
reduce(String key, Iterator values);
//key: a word
//values: a list of counts
int result = 0;
foreach v in values do
    result += ParseInt(v);
 Emit(key, AsString(result));
```
Real World Example 3

word count

It will be seen that this mere painstaking burrower and grub-worm of a poor devil of a Sub-Sub appears to have gone through the long Vaticans and street-stalls of the earth, picking up whatever random allusions to whales he could anyways find in any book whatsoever, sacred or profane. Therefore you must not, in every case at least, take the higgledy-piggledy whale statements, however authentic, in these extracts, for veritable gospel cetology. Far from it. As touching the ancient authors generally, as well as the poets here appearing, these extracts are solely valuable or entertaining, as affording a glancing bird’s eye view of what has been promiscuously said, thought, fancied, and sung of Leviathan, by many nations and generations, including our own.
Real World Example 3

word count
Co-occurrences Example 4
Pairs of keys

• Given: text file
• Want: for terms a, b, how often does a and b occur close together, e.g., within sentence?
• That is, output = ([a,b], count)

• How can this be computed?
Co-occurrences Example 4

Pairs of keys

• **Solution 1: pairs approach**
  
  – mapper for string s:
    • **for all** term pairs (a,b) in s: emit({a,b}, 1)
  
  – reducer just aggregates counts

• **Solution 2: “stripes” approach**
  
  – mapper for string s:
    • **collect all** t_i that co-occur with a
    • emit (a,{t_1, t_2, .... t_n})
  
  – reducer aggregates

*What is the difference?*
MapReduce Criticism
David DeWitt and Michael Stonebraker – 2008

1. MapReduce is a step backwards in database access based on
   - Schema describing data structure
   - Separating schema from the application
   - Advanced query languages
2. MapReduce is a poor implementation
   - Instead of indexes is uses brute force
3. MapReduce is not novel (ideas more than 20 years old and overcome)
4. MapReduce is missing features common in DBMSs
   - Indexes, transactions, integrity constraints, views, …
5. MapReduce is incompatible with applications implemented over DBMSs
   - Data mining, business intelligence, …
End of MapReduce?

- FaceBook used MapReduce in 2010
  - Hadoop

but…

- Google has recently (June 2014) announced a shift towards: Google Cloud DataFlow
  - Based on cloud and stream data processing
  - Idea: no need to maintain complex infrastructure
    - Data can be easily read, transformed and analyzed in a cloud

http://googledevelopers.blogspot.fr/2014/06/cloud-platform-at-google-io-new-big.html
Resources

- Jeffrey Dean and Sanjay Ghemawat: MapReduce: *Simplified Data Processing on Large Clusters*, Google, Inc.
- Google Code: *Introduction to Parallel Programming and MapReduce*
  - [code.google.com/edu/parallel/mapreduce-tutorial.html](http://code.google.com/edu/parallel/mapreduce-tutorial.html)
- Hadoop Map/Reduce Tutorial
  - [http://hadoop.apache.org/docs/r0.20.2/mapred_tutorial.html](http://hadoop.apache.org/docs/r0.20.2/mapred_tutorial.html)
- Open Source MapReduce
- David DeWitt and Michael Stonebraker: *Relational Database Experts Jump The MapReduce Shark*