Prof. Dr. Florian Heinz
florian.heinz@sysv.de

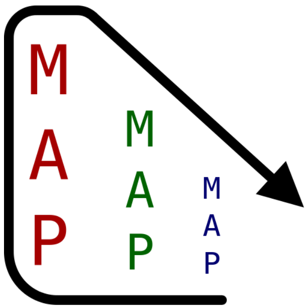
Modern Database Concepts

The Map-Reduce Algorithm

CC BY

OSTBAYERISCHE
TECHNISCHE HOCHSCHULE
REGENSBURG

Map Reduce



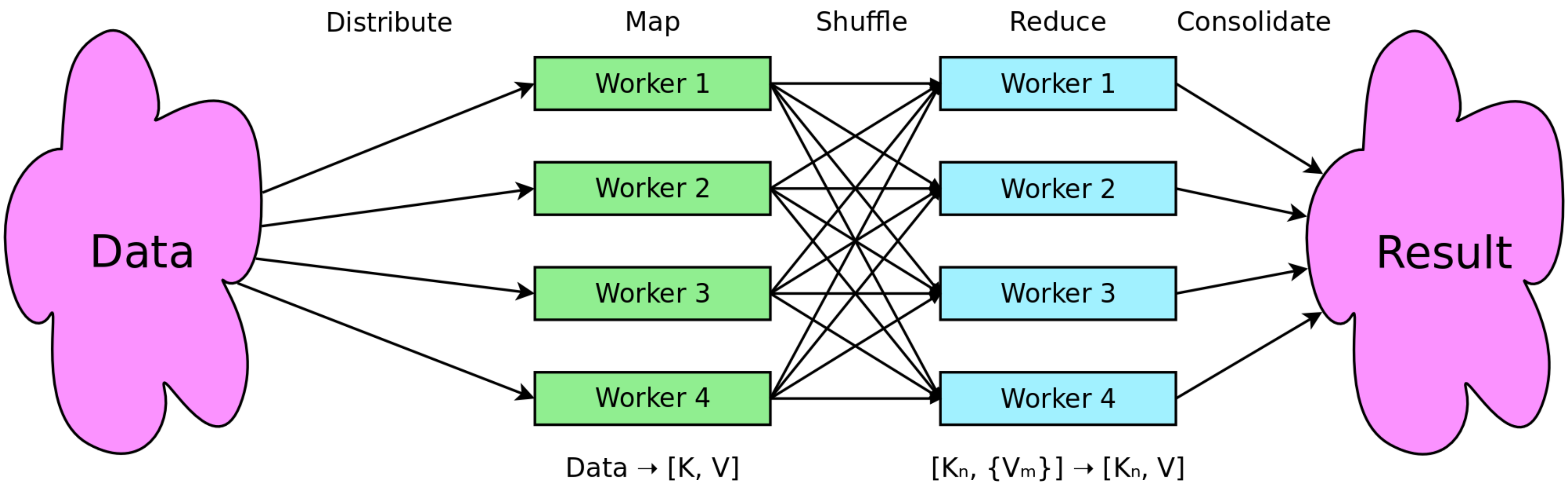
- Processing Schema for very big amounts of data
- Gained popularity in 2004 through Google
- Paper: "MAPREDUCE: SIMPLIFIED DATA PROCESSING ON LARGE CLUSTERS" by Dean & Ghemawat, 2004

  https://static.googleusercontent.com/media/research.google.com/en//archiv

  osdi04.pdf
- Useful for computations that can be massively parallelized

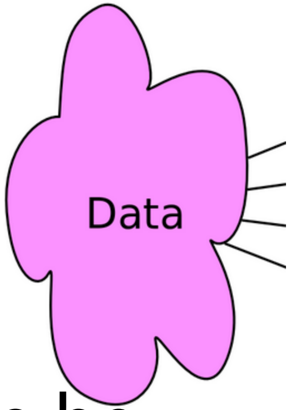
Map Reduce

Overview:

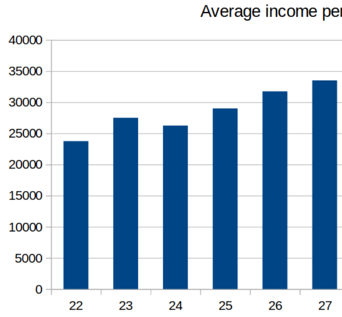


Map Reduce: Data

- Data format can be arbitrary, no special structure required
- Example: Raw text files, XML documents, Images, Audio Files
- Suitable for data in **Data Lakes**
- Drawback: Map and Reduce functions have to be programmed at low level, e.g. in Java



Running Example: Data



Input: CSV files with personal data (e.g. 100 files with many GB each)

```
# name,gender,birthdate,income
Anna Smith,female,1994-05-25,52000
Peter Miller,male,1996-07-02,45000
... <1 million lines>
```

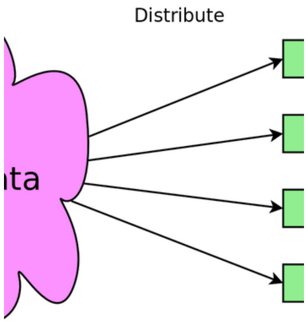
Task: Calculate the average yearly income per age of the people

Similar to SQL Query:

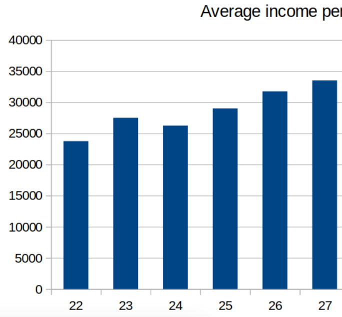
```
SELECT age(birthdate) AS age,avg(income) AS income FROM persons
                                                     GROUP BY age
```

Map Reduce: Distribute

- Distribution of the data items is usually handled by the framework
- A single large input file is split up into individual blocks on different nodes:
  - Sharding for load distribution ...
  - ... but also replication for fault tolerance
- A function to split the input into individual data items has to be provided by the user
- Examples: split based on words, lines or (binary) records
- The resulting items will be processed in parallel



Running Example: Distribute



Input: CSV files with personal data (e.g. 100 files, 1 GB each)

```
# name,gender,birthdate,income
Anna Smith,female,1994-05-25,52000
Peter Miller,male,1996-07-02,45000
... <100 files, each with 1 million records>
```

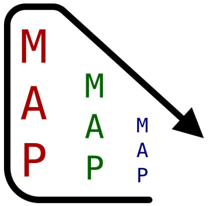
Output: Individual data items (e.g. 100 mio. items)

Large files are usually already split up in blocks on several nodes on the distributed storage

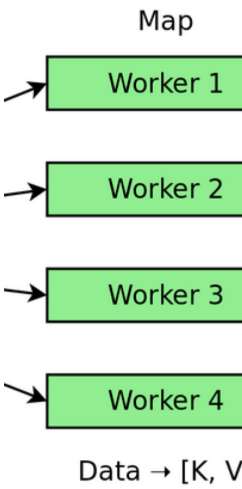
Distribution phase: Files are split up into individual records (CSV-lines), that go into the map phase.

MapReduce framework should try hard to process the data blocks locally.

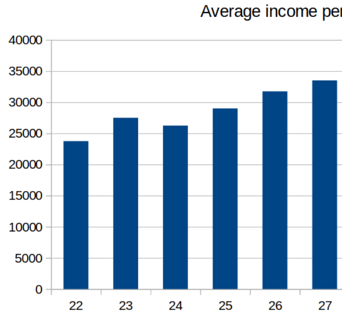
Map Reduce: Map



- First real processing step
- Map function has to be provided by the user
- Maps a data item into a Key/Value pair
- Massively parallel execution
  → Most of the work should happen in this step
- Data mapped to the same key is related and processed together in the reduce step



Running Example: Map



Input: CSV lines with personal data (e.g. 100 million records)

```
Anna Smith,female,1994-05-25,52000 -> (27,52000)
Peter Miller,male,1996-07-02,45000 -> (25,45000)
... <100 million records>
```

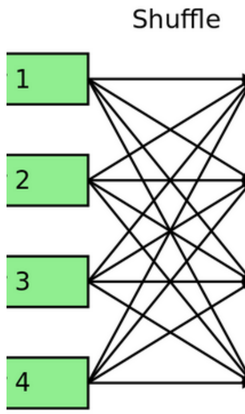
Output: Key-Value pairs (e.g. 100 million pairs)

Each record is mapped to a Key/Value pair `(age, income)`

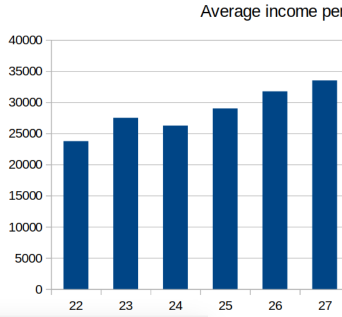
Parallel processing on many tasks and potentially distributed over the network

Map Reduce: Shuffle

- Redistributes Key/Value pairs
- Handled by the MapReduce framework
- Preparation for reducing
- All pairs with the same key go to the same node
- Expensive operation: potentially high communication overhead
- optionally: Preceding **Combine** step to reduce node-local data prior to shuffling



Running Example: Shuffle



Input: Key-Value pairs (e.g. 100 mio. pairs)

```
(27,52000), (27,43000), (27,78000), (27,30000) -> Worker2
(25,45000), (25,34000), (25,23000), (25,41000) -> Worker4
... <many key/value pairs with about 50 different keys>
```

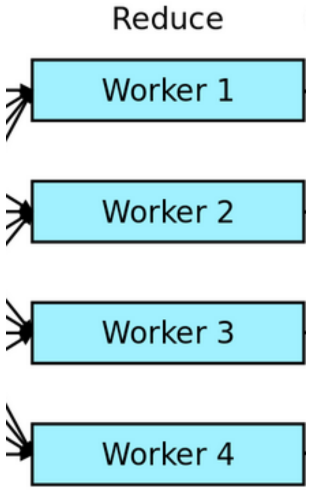
Same ages are distributed to the same workers for reducing (calculating the average)

In this example: 100 mio. pairs with 50 keys are distributed to 4 worker nodes for 50 reduction tasks

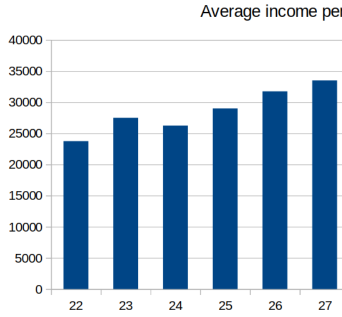
An optional **Combine** step could pre-aggregate the local (age,income) pairs grouped by age on each node. The result would be <= 200 pairs of (age, (count,avgincome)) (<= 50 on each node). The additional count is important for the final reduce step to weigh the intermediate results correctly

Map Reduce: Reduce

- A single reduce step processes all data with the same key
- Result: usually a single Key/Value-pair (optionally more than one)
- Level of parallelism depends on number of different keys
- Bordercase: If only one key is used, a single reduce step will have to process all data (no parallelism)



Running Example: Reduce



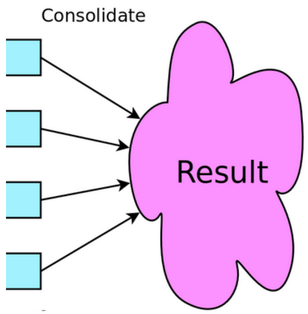
Input: One Key/Value-set pair per reduce task from kv-pairs with the same key

```
Worker 2.1: (27, {52000,43000,78000,30000}) -> (27,50750)
Worker 4.1: (25, {45000,34000,23000,41000}) -> (25,35750)
... <many reduce tasks on all workers>
```

One reduce task gets all income values for a single age.

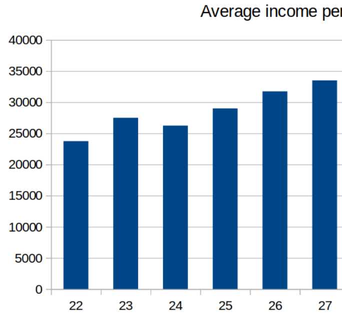
The reduce task calculates the average income and emits a key/value pair with `(age, avgincome)`

Map Reduce: Consolidate



- Collects results from the reduce step
- Usually written to a common data object
- e.g.: file on a distributed storage
- After that, the MapReduce algorithm is finished.

Running Example: Consolidate



Input: Key/Value pairs from the reduce tasks

```
27,50750
25,35750
... <many output records>
```

Output: File with the result records

The key/value pairs from the reduce tasks are written to a common storage, i.e. a file on a distributed filesystem

After the MapReduce algorithm has terminated, this file can be fetched to collect the results

Other Examples:

Word Count: Determine frequency of words

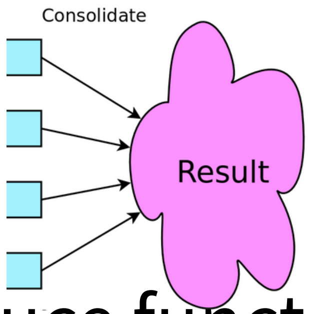
- Map: <word> -> (<word>, 1)
- Reduce: (<word>, {1,...}) -> (<word>, n)

Create Inverted Index:

- Map: <word> -> (<word>, <docid>)
- Reduce: (<word>, {doc1,doc2,doc1,doc3,...}) -> (<word>, [doc1,doc2,doc3,...])

Map Reduce: Conclusion

Advantages:



- Massive parallel computations
- Works on arbitrary data
- User only has to provide Map and Reduce functions

Problems:

- Computation effort must significantly outweigh the communications overhead
- Low level programming needed

Apache Hadoop

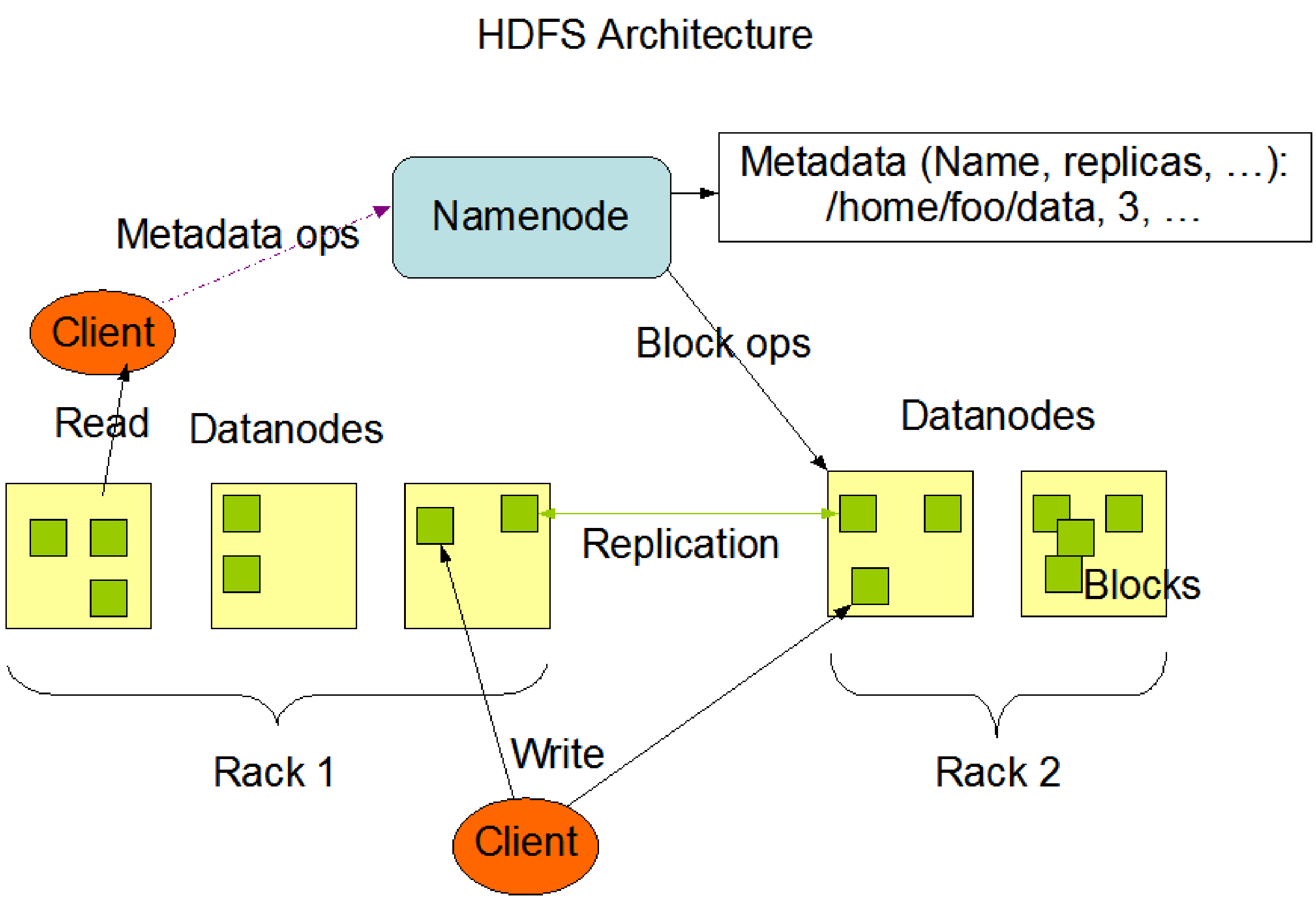
A Distributed Computing framework

Suitable for **Data Lakes**

Base: HDFS (Hadoop Distributed File System)

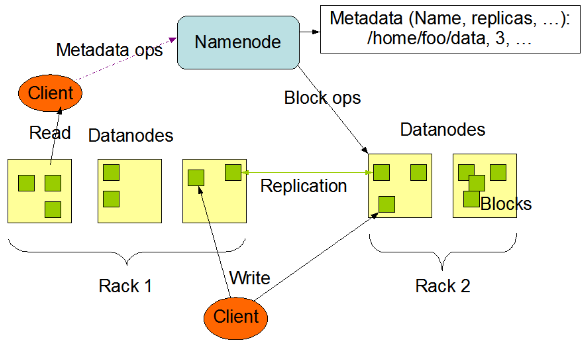
```
$ bin/hdfs dfs -mkdir -p /ex1/input
$ bin/hdfs dfs -mkdir -p /ex1/output
$ bin/hdfs dfs -put *.csv /ex1/input
$ bin/hdfs dfs -ls /ex1/input
Found 6 items
-rw-r--r-- user grp 38023 2022-05-01 21:34 /ex1/input/data1.csv
-rw-r--r-- user grp 37998 2022-05-01 21:34 /ex1/input/data2.csv
-rw-r--r-- user grp 37971 2022-05-01 21:34 /ex1/input/data3.csv
-rw-r--r-- user grp 37913 2022-05-01 21:34 /ex1/input/data4.csv
-rw-r--r-- user grp 37988 2022-05-01 21:34 /ex1/input/data5.csv
-rw-r--r-- user grp 37993 2022-05-01 21:34 /ex1/input/data6.csv
```

Apache Hadoop



Apache Hadoop

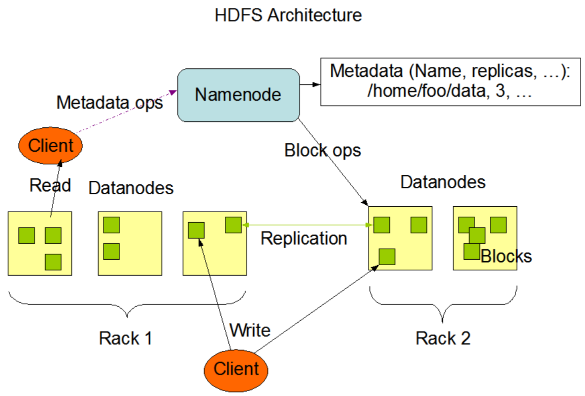
HDFS Architecture



Namenode:

- Stores Metadata of the File System
- Determines, which blocks go to which Datanode
- Client asks Namenode, where to find specific data.
  (e.g. bytes 15000-27000 of a specific file)
- Single point of failure, but high-availability with standby servers possible

Apache Hadoop



Datanode:

- Stores the actual data
- Files are splitted into blocks of fixed size
- With replication factor n, each data block exists on n different nodes
- Client transmits data directly from/to datanodes

More information: https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html

Apache Hadoop

MapReduce on input files

Generates output files

User has to provide java classes for:

- Input split function
- Map function
- Reduce function

Apache Hadoop

Provides mechanisms for:

- Distributing data amongst the cluster
- Executing the map and reduce functions with minimal communication overhead
- Fault tolerance by replicating data shards and reassigning work to active nodes

Apache Hadoop

Example AvgIncome:

```
$ javac -cp $(bin/hadoop classpath) AvgIncome.java
$ jar -cf ai.jar AvgIncome*.class
$ bin/hadoop jar ai.jar AvgIncome /ex1/input /ex1/output
$ bin/hdfs dfs -cat /ex1/output/part-r-00000
27   48156
28   50481
[...]
42   50696
```

Apache Hadoop

Map Function AvgIncome:

```
public static class CSVMapper
  extends Mapper<Object, Text, IntWritable, IntWritable> {

 public void map(Object key, Text value, Context context)
              throws IOException, InterruptedException {
  try {
   StringTokenizer itr = new StringTokenizer(value.toString());
   String name = itr.nextToken(",");
   String gender = itr.nextToken(",");
   LocalDate birthdate = LocalDate.parse(itr.nextToken(","));
   Integer income = Integer.parseInt(itr.nextToken(","));
   LocalDate now = LocalDate.now();
   Period period = Period.between(birthdate, now);

   IntWritable age = new IntWritable(period.getYears());
   IntWritable inc = new IntWritable(income);
   context.write(age, inc);
  } catch (Exception e) { }
 }
}
```

Apache Hadoop

Reduce Function AvgIncome:

```
public static class AvgCalculateReducer extends
  Reducer<IntWritable,IntWritable,IntWritable,IntWritable> {

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