MapReduce-style data processing

Software Languages Team
University of Koblenz-Landau
Ralf Lämmel and Andrei Varanovich

© 2012, 101companies & Software Languages Team (University of Koblenz-Landau)

Related meanings of MapReduce

- Functional programming with ‘map’ & ‘reduce’
- An algorithmic skeleton for data parallelism
- Google’s related programming model
- Related programming techniques for data technologies

© 2012, 101companies & Software Languages Team (University of Koblenz-Landau)

Functional programming with ‘map’ & ‘reduce’

We use Haskell here for illustration; ‘reduce’ is called ‘foldr’ in Haskell.

© 2012, 101companies & Software Languages Team (University of Koblenz-Landau)

The higher-order function ***map***

map f l applies the function f to each element of the list l and produces the list of results.

```
> :t map
map :: (a -> b) -> [a] -> [b]
> map ((+) 1) [1,2,3]
[2,3,4]
> map ((*) 2) [1,2,3]
[2,4,6]
> map even [1,2,3]
[False,True,False]
```

The type of map

Increment the numbers.

Double the numbers.

Test numbers to be even.

© 2012, 101companies & Software Languages Team (University of Koblenz-Landau)

The higher-order function ***foldr***

foldr f x l combines all elements of the list l with the binary operation f starting from x.

```
> :t foldr
foldr :: (a -> b -> b) -> b -> [a] -> b
> foldr (+) 0 [1,2,3]
6
> foldr (&&) True [True,True,False]
False
```

The type of foldr

Sum up the numbers.

‘And‘ the Booleans.

© 2012, 101companies & Software Languages Team (University of Koblenz-Landau)

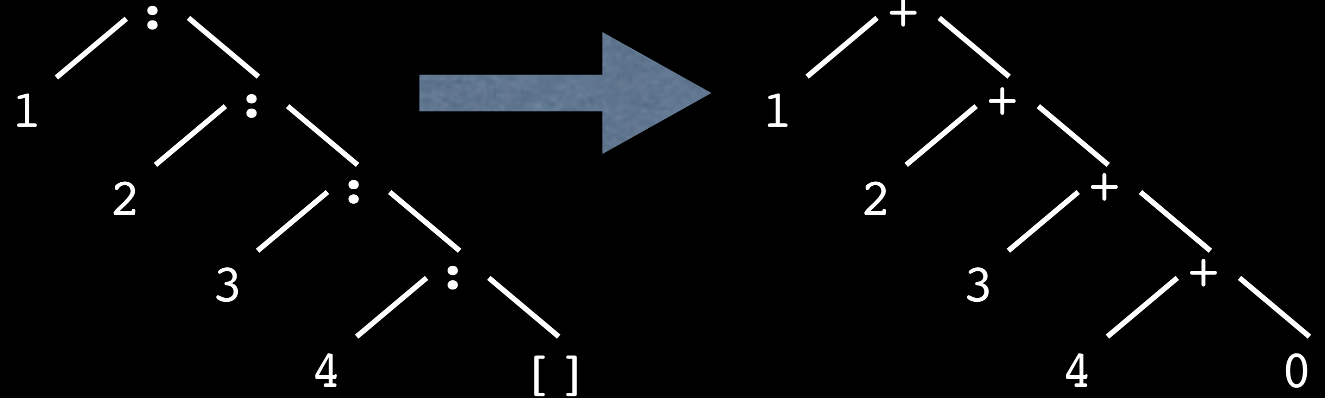
Typical forms of reduction

```
sum = foldr (+) 0
product = foldr (*) 1
and = foldr (&&) True
or = foldr (||) False
```

© 2012, 101companies & Software Languages Team (University of Koblenz-Landau)

Another way to think of *foldr*

```
> let l1 = [1,2,3,4]
> sum l1
10
```




```
> product l1
24
```

[1,2,3,4] = 1:(2:(3:(4:[])))

```
sum, product :: Num a => [a] -> a
sum = foldr (+) 0
product = foldr (*) 1
```

How to replace (:)

How to replace [ ]

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f k [] = k
foldr f k (x:xs) = f x (foldr f k xs)
```

Datatypes for companies

```
data Company
 = Company Name [Department]
data Department
 = Department Name Manager [Department] [Employee]
data Employee = Employee Name Address Salary
type Manager = Employee
type Name = String
type Address = String
type Salary = Float
```

[101implementation:haskell]

© 2012, 101companies & Software Languages Team (University of Koblenz-Landau)

Company structure in Haskell

```
company =
 Company
  "meganalysis"
  [ Department "Research"
      (Employee "Craig" "Redmond" 123456)
      []
      [ Employee "Erik" "Utrecht" 12345,
        Employee "Ralf" "Koblenz" 1234
      ],
    Department "Development"
      (Employee "Ray" "Redmond" 234567)
      [ Department "Dev1"
          (Employee "Klaus" "Boston" 23456)
          [ Department "Dev1.1"
              (Employee "Karl" "Riga" 2345)
              []
              [ Employee "Joe" "Wifi City" 2344 ]
          ]
          []
      ]
      []
  ]
```

© 2012, 101companies & Software Languages Team (University of Koblenz-Landau)

Cutting salaries in Haskell

```haskell
cut :: Company -> Company
cut (Company n ds) = Company n (map dep ds)
 where
  dep :: Department -> Department
  dep (Department n m ds es)
   = Department n (emp m) (map dep ds) (map emp es)
   where
    emp :: Employee -> Employee
    emp (Employee n a s) = Employee n a (s/2)
```

[101implementation:haskell]

© 2012, 101companies & Software Languages Team (University of Koblenz-Landau)

Totaling salaries in Haskell

```
total :: Company -> Float
total (Company n ds) = sum (map dep ds)
 where
  dep :: Department -> Float
  dep (Department _ m ds es)
   = sum (emp m : map dep ds ++ map emp es)
   where
    emp :: Employee -> Float
    emp (Employee _ _ s) = s
```

[101implementation:haskell]

© 2012, 101companies & Software Languages Team (University of Koblenz-Landau)

The basic idea of data parallelism

© 2012, 101companies & Software Languages Team (University of Koblenz-Landau)

Think of a huge company
(Millions of employees)

- Too big to store on one disk!
- We assume a ***flat*** representation.
- Compute total in ***parallel*** on many machines.

© 2012, 101companies & Software Languages Team (University of Koblenz-Landau)

Datatypes for flat companies

```
type Company = Name -- Name of company
type Department = (
      Name,           -- Name of department
      Maybe Name,     -- Name of ancestor department
      Name            -- Name of associated company
   )
type Employee = (
      Name,         -- Name of employee
      Name,         -- Name of associated department
      Name,         -- Name of associated company
      Address,      -- Address of employee
      Salary,       -- Salary of employee
      Bool          -- Manager?
   )
type Name = String
type Address = String
type Salary = Float
```

[101implementation:haskellFlat]

© 2012, 101companies & Software Languages Team (University of Koblenz-Landau)

Flat companies

```
companies :: [Company]
companies = ["meganalysis"]

departments :: [Department]
departments = [
     ("Research", Nothing, "meganalysis"),
     ("Development", Nothing, "meganalysis"),
     ("Dev1", Just "Development", "meganalysis"),
     ("Dev1.1", Just "Dev1", "meganalysis")
   ]

employees :: [Employee]
employees = [
     ("Craig", "Research", "meganalysis", "Redmond", 123456, True),
     ("Erik", "Research", "meganalysis", "Utrecht", 12345, False),
     ("Ralf", "Research", "meganalysis", "Koblenz", 1234, False),
     ("Ray", "Development", "meganalysis", "Redmond", 234567, True),
     ("Klaus", "Dev1", "meganalysis", "Boston", 23456, True),
     ("Karl", "Dev1.1", "meganalysis", "Riga", 2345, True),
     ("Joe", "Dev1.1", "meganalysis", "Wifi City", 2344, False)
   ]
```

© 2012, 101companies & Software Languages Team (University of Koblenz-Landau)

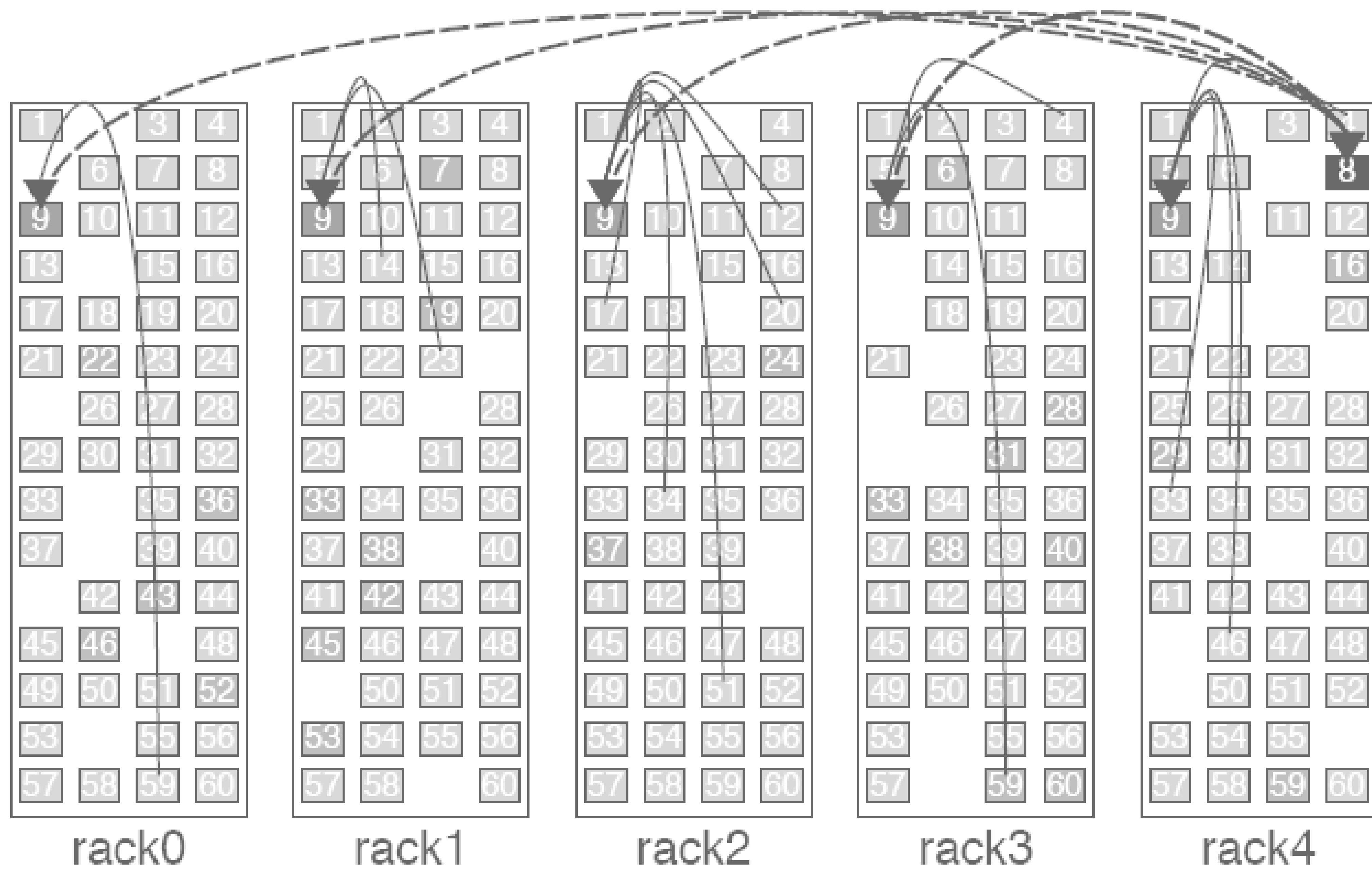
Total flat companies

```
-- Total all salaries (perhaps even of several companies)
total :: [Employee] -> Float
total = sum . map (\(_e, _d, _c, _a, s, _m) -> s)
```

[101implementation:haskellFlat]

© 2012, 101companies & Software Languages Team (University of Koblenz-Landau)

Clusters of machines for parallel map-reduce



http://labs.google.com/papers/sawzall.html

Total flat companies on many machines

```
total :: [[Employee]] -> Float

total
  = sum
  . map (sum . map (\(_e, _d, _c, _a, s, _m) -> s))
```

[101implementation:haskellFlat]

© 2012, 101companies & Software Languages Team (University of Koblenz-Landau)

Total flat companies

```haskell
-- Total all salaries grouped by company times department
totalPerDepartment :: [Employee] -> Map (Name, Name) Float
totalPerDepartment = foldr insert empty
 where
  insert (_e, d, c, _a, s, _m)
   = insertWith (+) (c, d) s
```

How to think parallel here?

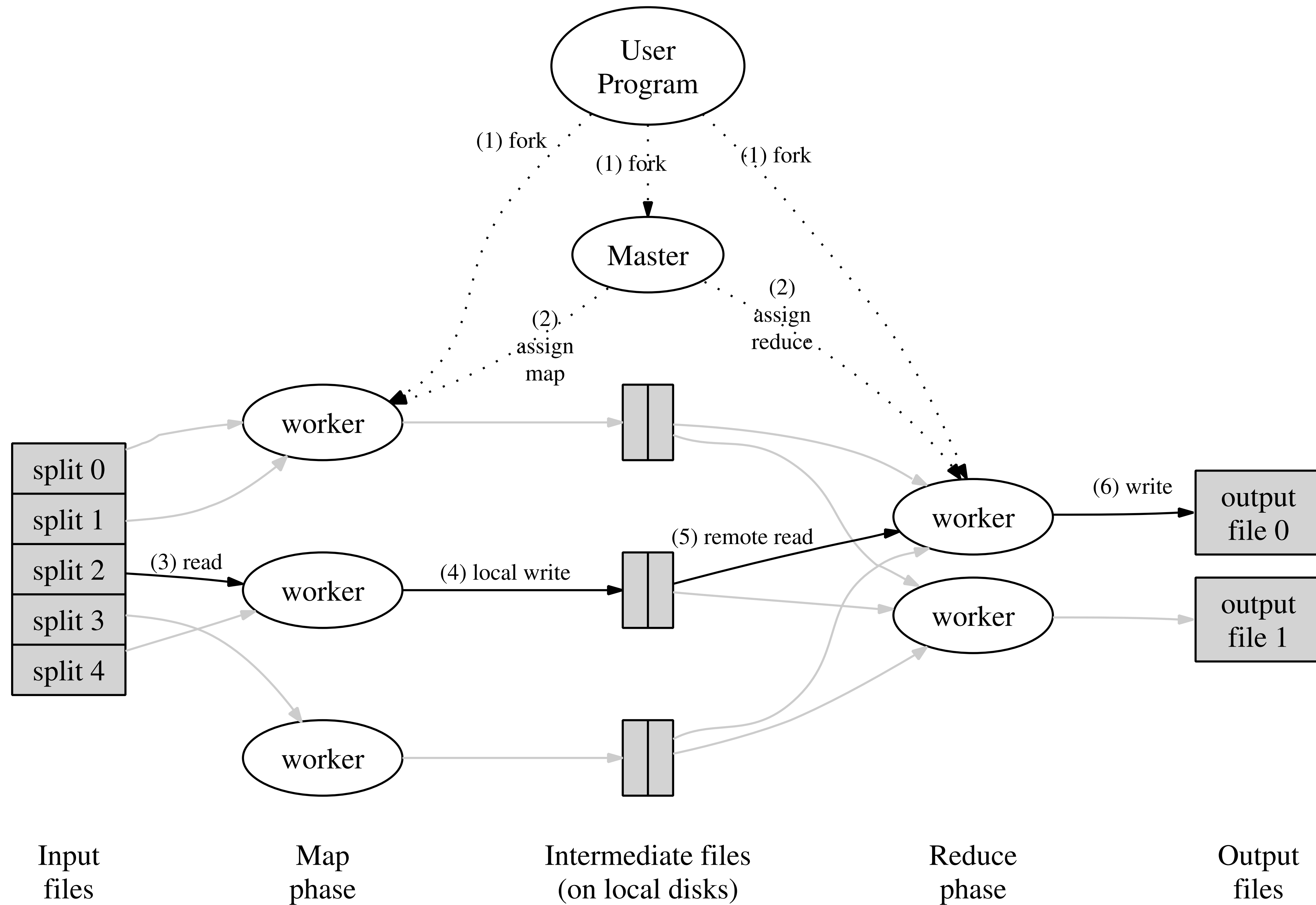
Output

```
fromList [
   (("meganalysis","Dev1"),23456.0),
   (("meganalysis","Dev1.1"),4689.0),
   (("meganalysis","Development"),234567.0),
   (("meganalysis","Research"),137035.0)
]
```

[101implementation:haskellFlat]

© 2012, 101companies & Software Languages Team (University of Koblenz-Landau)

Google's MapReduce Programming Model



[http://labs.google.com/papers/mapreduce.html]

© 2012, 101companies & Software Languages Team (University of Koblenz-Landau)

Large Scale Data Processing

- Process lots of data to produce other data.
- Use hundreds or thousands of CPUs.
- Automatic parallelization and distribution.

© 2012, 101companies & Software Languages Team (University of Koblenz-Landau)

The MapReduce programming model

Input & Output: sets of key/value pairs

Programmer specifies two functions:

map (in_key, in_value) -> list(out_key, intermediate_value)

Processes input key/value pair.
Produces set of intermediate pairs.

reduce (out_key, list(intermediate_value)) -> list(out_value)

Combines all intermediate values for a particular key.
Produces a set of merged output values (usually just one).
The domains for intermediate and output values coincide.

© 2012, 101companies & Software Languages Team (University of Koblenz-Landau)

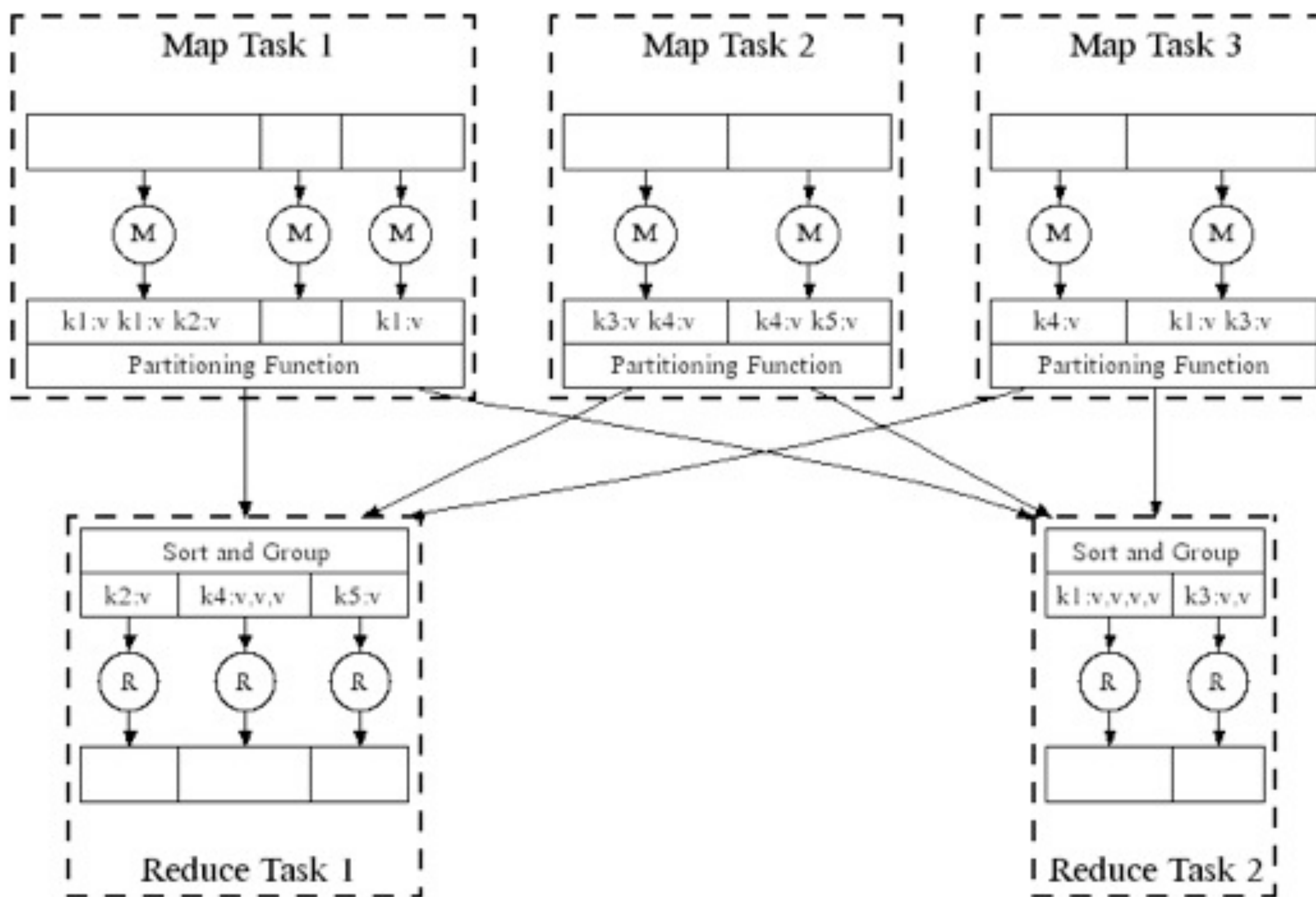
An example: counting the number of occurrences of each word in a collection of documents

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

```
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(AsString(result));
```

© 2012, 101companies & Software Languages Team (University of Koblenz-Landau)

Distribution: many map and reduce tasks



[http://labs.google.com/papers/mapreduce.html]

© 2012, 101companies & Software Languages Team (University of Koblenz-Landau)

Control of job execution

- Automatic division of job into tasks
- Automatic placement of computation near data
- Automatic load balancing
- Recovery from failures & stragglers

User focuses on application, not on complexities of distributed computation.

© 2012, 101companies & Software Languages Team (University of Koblenz-Landau)

Fault tolerance

- Cheap nodes fail, especially if you have many
  - Mean time between failures for 1 node = 3 years
  - Mean time between failures for 1000 nodes = 1 day
- Solution: Build fault-tolerance into system
  - If a node crashes: Re-launch its current tasks on other nodes and re-run any maps the node previously ran.
  - If a task crashes: Retry on another node. (OK for a map because it has no dependencies. OK for reduce because map outputs are on disk.)

© 2012, 101companies & Software Languages Team (University of Koblenz-Landau)

Network as a bottleneck

- **Limited bandwidth (especially for commodity network)**
- **Solution: Push computation to the data**

© 2012, 101companies & Software Languages Team (University of Koblenz-Landau)

‘Stragglers’

. **If a task is going slowly (straggler):**

Launch second copy of task on another node (“speculative execution”). Take the output of whichever copy finishes first, and kill the other.

Surprisingly important in large clusters:

Stragglers occur frequently due to failing hardware, software bugs, misconfiguration, etc. Single straggler may noticeably slow down a job.

© 2012, 101companies & Software Languages Team (University of Koblenz-Landau)

Apache Hadoop

Apache Hadoop is an open-source software framework that supports data-intensive distributed applications [...]. It enables applications to work with thousands of computational independent computers and petabytes of data. Hadoop was derived fromGoogle's MapReduce and Google File System (GFS) papers. The entire Apache Hadoop "platform" is now commonly considered to consist of the Hadoop kernel, MapReduce and HDFS, as well as a number of related projects [...]. Hadoop is a top-level Apache project being built and used by a global community of contributors, [...] written in the Java programming language.

[http://en.wikipedia.org/wiki/Apache_Hadoop] 13 Sep 2012

© 2012, 101companies & Software Languages Team (University of Koblenz-Landau)

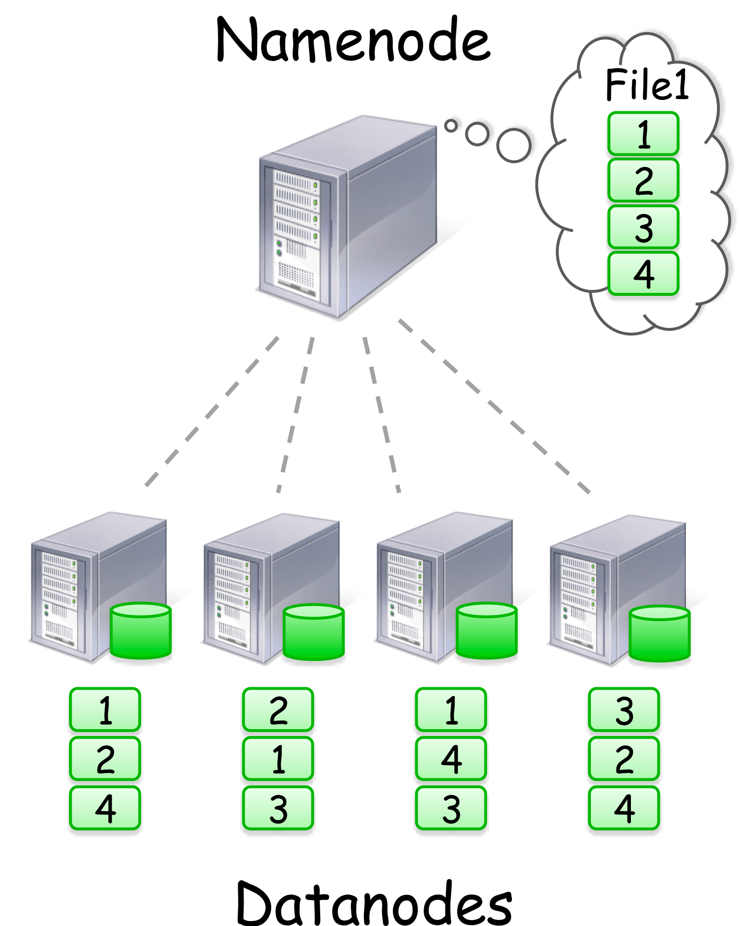
Hadoop components

- Distributed file system (HDFS)
  - Single namespace for entire cluster
  - Replicates data 3x for fault-tolerance
- MapReduce framework
  - Executes user jobs specified as “map” and “reduce” functions
  - Manages work distribution & fault-tolerance

© 2012, 101companies & Software Languages Team (University of Koblenz-Landau)

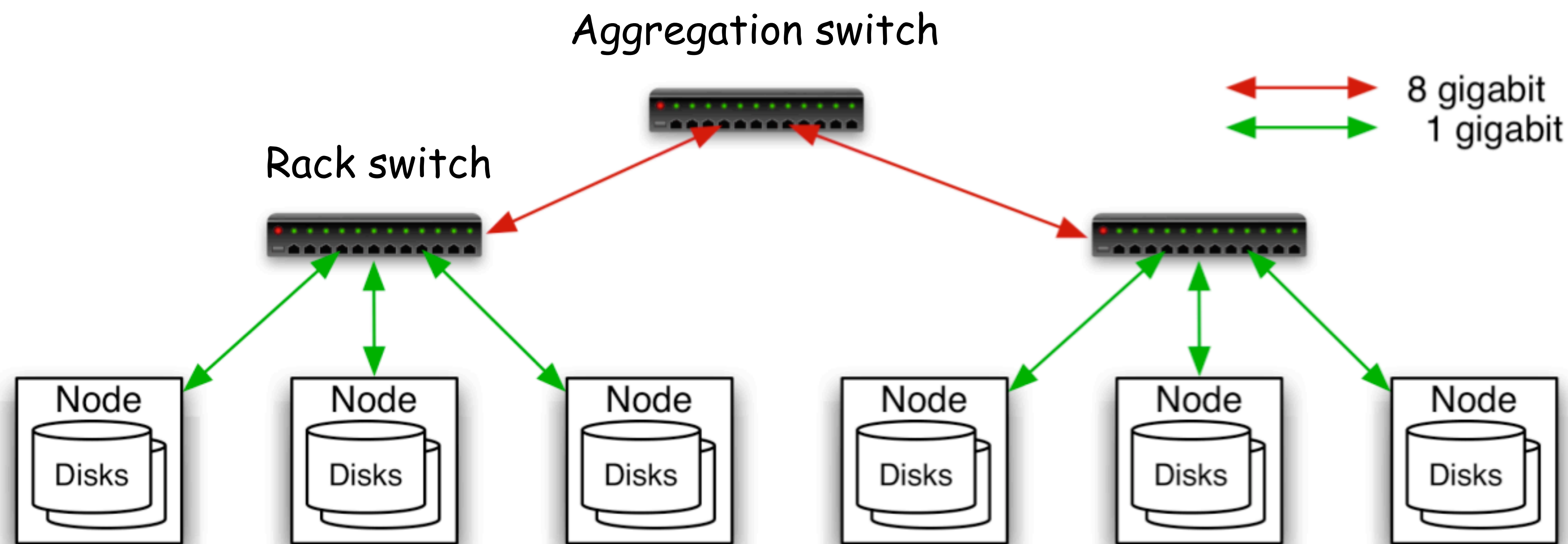
HDFS

- Files split into 128MB blocks
- Blocks replicated across several datanodes (usually 3)
- Single namenode stores metadata (file names, block locations, etc)
- Optimized for large files, sequential reads
- Files are append-only



© 2012, 101companies & Software Languages Team (University of Koblenz-Landau)

A Hadoop Cluster



40 nodes/rack, 1000-4000 nodes in cluster
1 Gbps bandwidth within rack, 8 Gbps out of rack
Node specs (Yahoo terasort):
    8 x 2GHz cores, 8 GB RAM, 4 disks (= 4 TB

© 2012, 101companies & Software Languages Team (University of Koblenz-Landau)

Demo

101implementation:hadoop

A *101companies* implementation using ***Hadoop***

© 2012, 101companies & Software Languages Team (University of Koblenz-Landau)

```
public static class TotalMapper extends
    Mapper<Text, Employee, Text, DoubleWritable> {
  private static String name;

  protected void setup(Context context) throws IOException,
      InterruptedException {
    name = context.getConfiguration().get(Total.QUERIED_NAME);
  }

  protected void map(Text key, Employee value, Context context)
      throws IOException, InterruptedException {
    if (value.getCompany().toString().equals(name))
      context.write(value.getCompany(), value.getSalary());
  }
}
```

© 2012, 101companies & Software Languages Team (University of Koblenz-Landau)

```java
public static class TotalReducer extends
    Reducer<Text, DoubleWritable, Text, DoubleWritable> {

  protected void reduce(Text key, Iterable<DoubleWritable> values,
      Context context) throws IOException, InterruptedException {
    double total = 0;
    for (DoubleWritable value : values) {
      total += value.get();
    }
    context.write(key, new DoubleWritable(total));
  }
}
```

© 2012, 101companies & Software Languages Team (University of Koblenz-Landau)

Summary

You learned about ...

- functional programming with map & reduce,
- related opportunities of parallelization,
- Google's MapReduce programming model,
- and the Hadoop implementation.

© 2012, 101companies & Software Languages Team (University of Koblenz-Landau)

Resources

- Google's MapReduce Programming Model -- Revisited

  http://userpages.uni-koblenz.de/~laemmel/MapReduce/

© 2012, 101companies & Software Languages Team (University of Koblenz-Landau)