# Peer-to-Peer and Cloud Computing

## MapReduce and Related Technologies

14th January 2019

Universität Augsburg
Institut für Informatik
Lehrstuhl für Organic Computing

# Gliederung

ORGANIC COMPUTING

MapReduce

The “next generation” of Big Data algorithms

# MapReduce

# Motivation for MapReduce

- initially created by Google Inc.
- over the years: “hundreds of special-purpose computations that process large amounts of raw data”
    - crawled documents $\rightarrow$ inverted indices
    - web request logs $\rightarrow$ frequencies
    - etc.
- computations themselves are straightforward
- however: a *lot* of data
  $\Rightarrow$ distributed (parallel) computation necessary

# What MapReduce provides

ORGANIC COMPUTING

- simple programming model
- messy details are hidden, notably
  - parallelisation,
  - fault-tolerance,
  - data distribution and
  - load balancing

# Roots of MapReduce: **map**

- inspired by functional programming (esp. Lisp)
- example usage of Lisp's **map**:

```
(map 'list #'(lambda (x) (+ x 1)) '(1 2 3 4 5))
 ⇒  (2 3 4 5 6)
```

- takes a function and a sequence and
- applies that function to every element of the sequence returning a new sequence

# Roots of MapReduce: `reduce`

- `reduce` also comes from functional languages (esp. Lisp):

  ```
  (reduce #'(lambda (x y) (* x y)) '(1 2 3 4 5))
   ⇒  (120)
  ```

- takes a function and a sequence
- the first element serves as the first *accumulator value*
- then, the next accumulator value is the value of the binary function applied to the *current accumulator value* and the *next element*

# The MapReduce programming model

A MapReduce application has to implement two functions with the following signatures:

- `map :: (k₁, v₁) → [(k₂, v₂)]`
- `reduce :: (k₂, [v₂]) → [v₂]`

Note: The `map`-function from MapReduce has a different signature than the *original* `map`-function from functional programming (e.g. from Lisp)! It more resembles a function that would be used *with* a `map`-function from functional programming.

# Classic example: Count occurrences of words

- given a set of documents,
- calculate the number of occurrences for each word
- algorithmically simple task, however: assume a very large set of documents

# (Very) naïve implementation

```
public Map<Word, Integer>
  count(Set<Document> documents) {
    Map<Word, Integer> counts = new Map<>();
    for (Document d : documents) {
      for (Word w: d.toWords()) {
        counts.merge(w, 1, Integer::add);
      }
    }
}
```

MapReduce separates processing a single document from “merging” that document’s word counts with the other documents’ word counts.

# Mapping over documents

ORGANIC COMPUTING

```
public void map(Key key, Document document) {
  for (Word w : document.toWords()) {
    emitIntermediate(w, 1);
  }
}
```

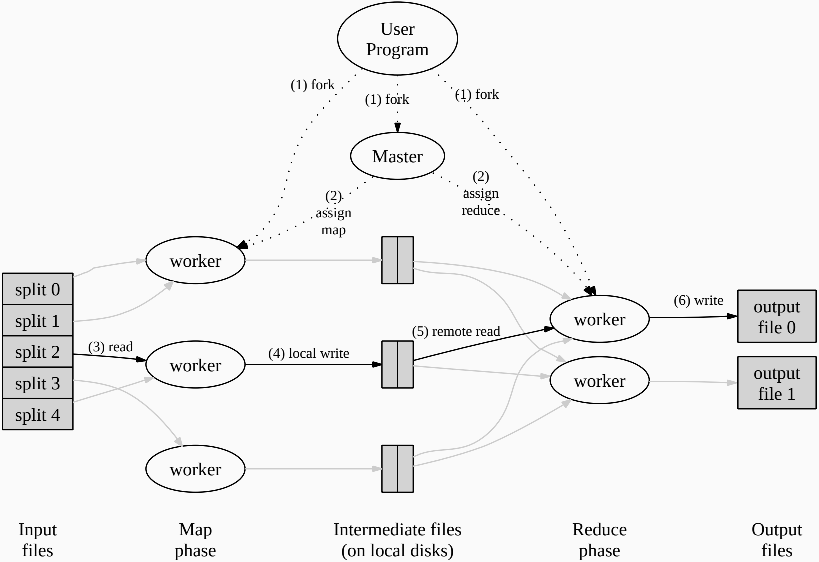
Note: While `key` is not used here, it may be required in another application.

Why use `emitIntermediate` instead of aggregating a list and `return`ing?

# Reducing

```java
public void reduce(Word w, Set<Integer> counts) {
  Integer sum = 0;
  for (Integer i : counts) {
    sum += i;
  }
  emit(w, sum);
}
```

# Execution



User Program
(1) fork
(1) fork
(1) fork
Master
(2) assign map
(2) assign reduce
worker
worker
worker
worker
worker
split 0
split 1
split 2
split 3
split 4
(3) read
(4) local write
(5) remote read
(6) write
output file 0
output file 1
Input files
Map phase
Intermediate files (on local disks)
Reduce phase
Output files

# Execution details

- input is partitioned into pieces of a user-given size (e.g. 16MB)
  $\Rightarrow$ distributed `map`'ing
- key space of intermediary results is partitioned using user-given partitioning function
  $\Rightarrow$ distributed `reduce`'ing
- "emitting" (intermediary) results $\approx$ writing them to a shared memory
- role of the master node:
  - picks idle workers for map or reduce tasks
  - gives them the memory address of their input

# Why is *parallel execution* so easily possible?

Because there are no dependencies between `map` calls (*partition* of input space)!

# Fault tolerance

- on worker failure: simply reschedule all its tasks on other workers (no additional overhead necessary because no dependencies!)
- on master failure: either
    - use latest backup of its state and restart it, or
    - simply re-run the whole MapReduce task

# Notable implementation: Hadoop MapReduce

ORGANIC COMPUTING

- Hadoop project consists of

  **Hadoop Distributed File System (HDFS)** high-throughput access to application data

  **Hadoop YARN** framework for job scheduling and cluster resource management

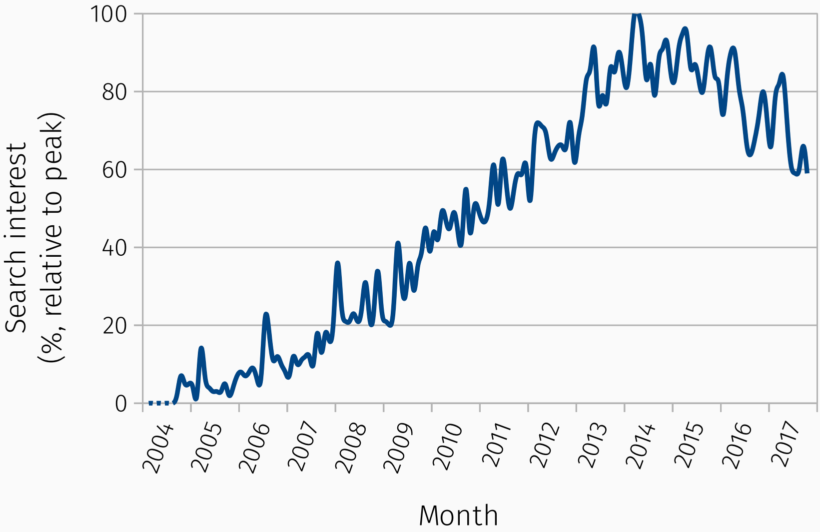
  **Hadoop MapReduce** YARN-based implementation of MapReduce

- Open source!
- YARN and HDFS are general purpose

  $\Rightarrow$ they are relied on by other projects as well

# The “next generation” of Big Data algorithms

# Google trend graph for “MapReduce” keyword



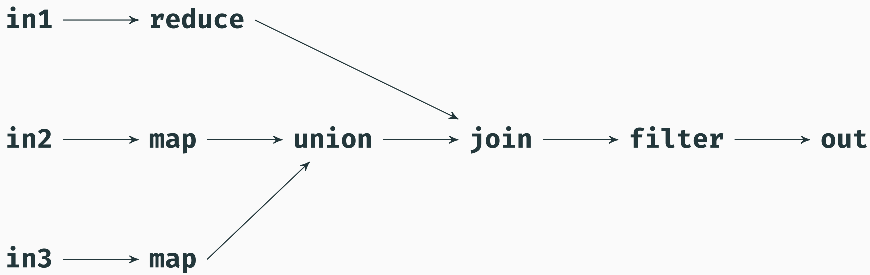
# Why?

- from a developer's perspective: MapReduce unnecessarily restricted!
  - Why only exactly support the MapReduce pipeline?
  - Why only support functional "primitives" `map` and `reduce`?
  - E.g., why not also `filter :: (a → Bool) → [a] → [a]`
- from a performance perspective: iterative applications (such as machine learning) not properly supported!
  - don't behave well if too much I/O is involved
  - MapReduce always writes results to (distributed) disk $\Rightarrow$ too much overhead through I/O

# DAGs—actually just real functions

- more general than MapReduce: Directed Acyclic Graphs (DAGs)
- MapReduce is a DAG itself:
  **`in`** ⟶ **`map`** ⟶ **`reduce`** ⟶ **`out`**
- DAGs are really composed (pure!) functions
  ⇒ *functional programming*

# Example for DAG and the corresponding function




This DAG equals the following function:

```
out(in1, in2, in3) =
  filter(
    join(
      reduce(in1),
      union(
        map(in2)),
        map(in3)))
```

ORGANIC COMPUTING

- "A fast and general engine for large-scale data processing."
- more general than "just" MapReduce
- many more parallelised functional operations
- builds on a distributed storage (e.g. HDFS or a simple shared file system)
- holds more stuff in memory than (original) MapReduce
  $\Rightarrow$ greatly(!) increased performance

# MapReduce example using Apache Spark

ORGANIC COMPUTING

- Scala has proper type inference $\Rightarrow$ only difference to "non-distributed" (functional) Scala code to initially create an implicit `SparkContext`

```scala
// sc: SparkContext
val lines = sc.textFile("data.txt")
val lineLengths = lines.map(s => s.length)
val totalLength =
  lineLengths.reduce((a, b) => a + b)
```

- As long as you keep programming *purely functional*, no need to think about parallelism at all!
- Need `lineLengths` somewhere else in your application? $\rightarrow$ simply add `lineLengths.persist()` and that constant is written to the distributed storage

# Conclusion

ORGANIC COMPUTING

- increasingly less knowledge required for using parallelism
- *functional programming* is becoming more and more important

- Jeffrey Dean and Sanjay Ghemawat. “MapReduce: Simplified Data Processing on Large Clusters”. In: *Communications of the ACM* 51.1 (2008), pp. 107–113.
- Google. *MapReduce on Google Trends*. 2017. URL: `https://trends.google.com/trends/explore?date=all&q=MapReduce` (visited on 10/11/2017).
- LispWorks Ltd. *Lisp Function MAP*. 2017. URL: `http://clhs.lisp.se/Body/f_map.htm` (visited on 10/11/2017).

- LispWorks Ltd. *Lisp Function REDUCE*. 2017. URL: http://clhs.lisp.se/Body/f_reduce.htm (visited on 10/11/2017).
- Apache Spark Team. *Apache Spark Programming Guide*. 2017. URL: https://spark.apache.org/docs/latest/rdd-programming-guide.html (visited on 10/10/2017).
- Apache Spark Team. *Apache Spark Website*. 2017. URL: https://spark.apache.org/ (visited on 10/10/2017).