AIM3 – Scalable Data Analysis and Data Mining

03 – Hadoop, Parallel Joins, Ecosystem

Sebastian Schelter, Christoph Boden, Volker Markl

Fachgebiet Datenbanksysteme und Informationsmanagement
Technische Universität Berlin

http://www.dima.tu-berlin.de/

Scalability

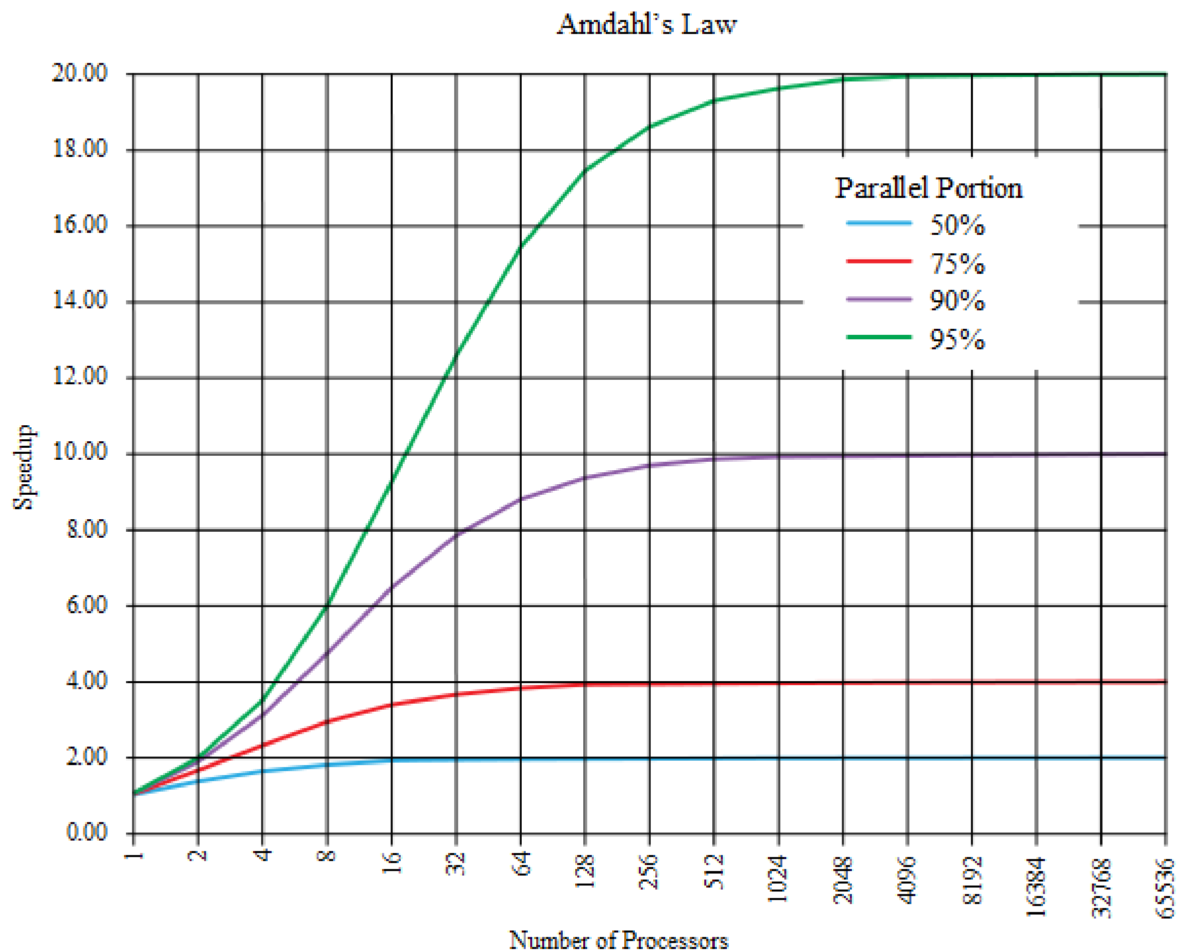
Parallel Speedup

- The Speedup is defined as: $S_p = {}^{T_1}/_{T_p}$
  - T_1: runtime of the sequential program
  - T_p: runtime of the parallel program on p processors

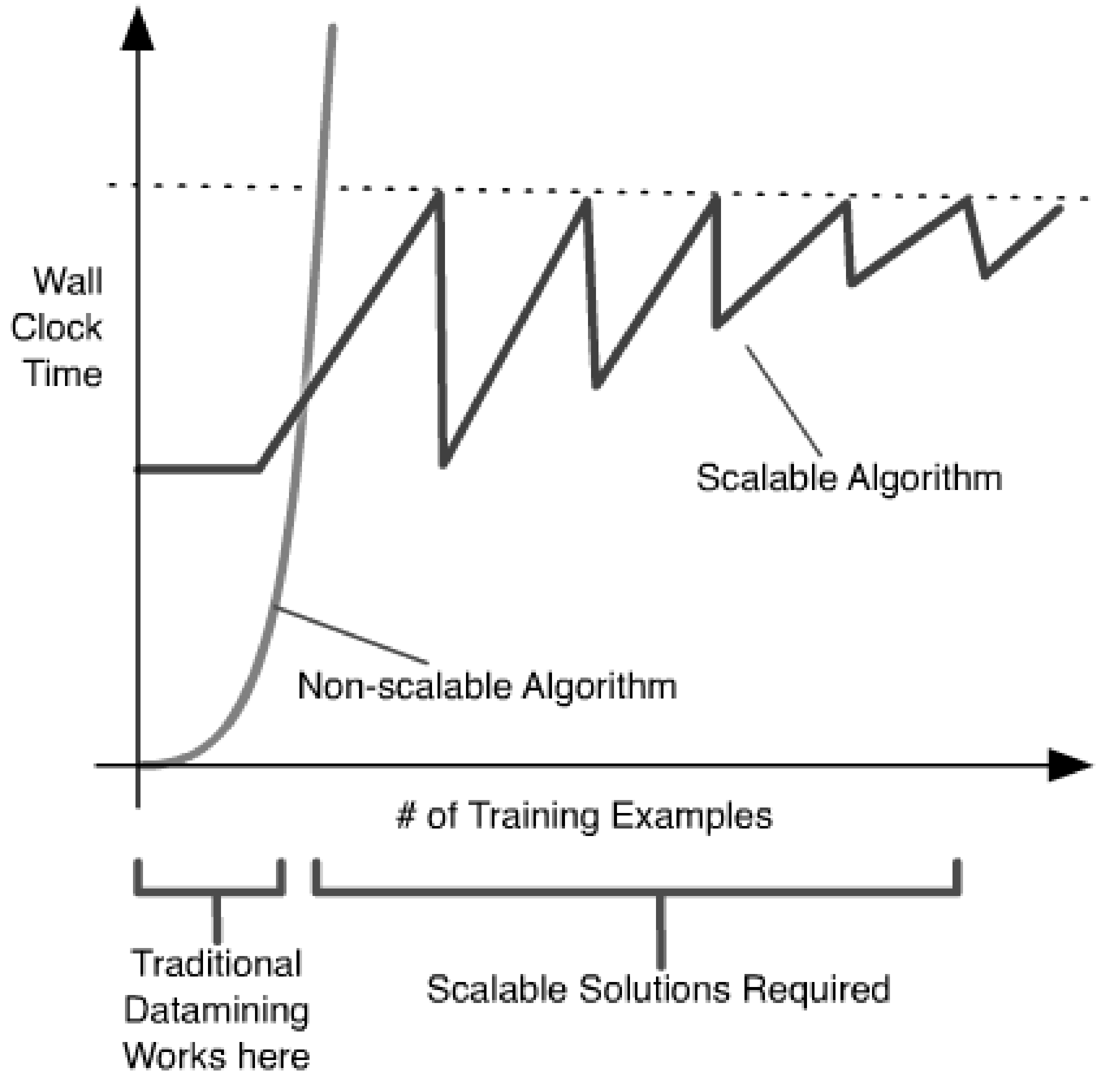
- Ahmdal's Law: „The maximal speedup is determined by the non-parallelizable part of a program" :
  - $S_{max} = \frac{1}{(1-f)+{}^{f}/_{p}}$ f: fraction of the program that can be parallelized
  - → Ideal speedup: S=p for f=1.0 (linear speedup)
  - However - since usually f<1.0 -, S is bound by a constant ! (e.g. ~10 for f=0.9)
  - → Fixed problems can only be parallelized to a certain degree!

- Gustavson's Law: „More processors are usually added to solve a larger problem in the same time" :
  - $S_{max} = (1-f) + Pf$ P: number of processors AND problem size
  - The larger the problem gets, the better the speedup gets.
  - →A growing problem can be effectively handled using parallelization!

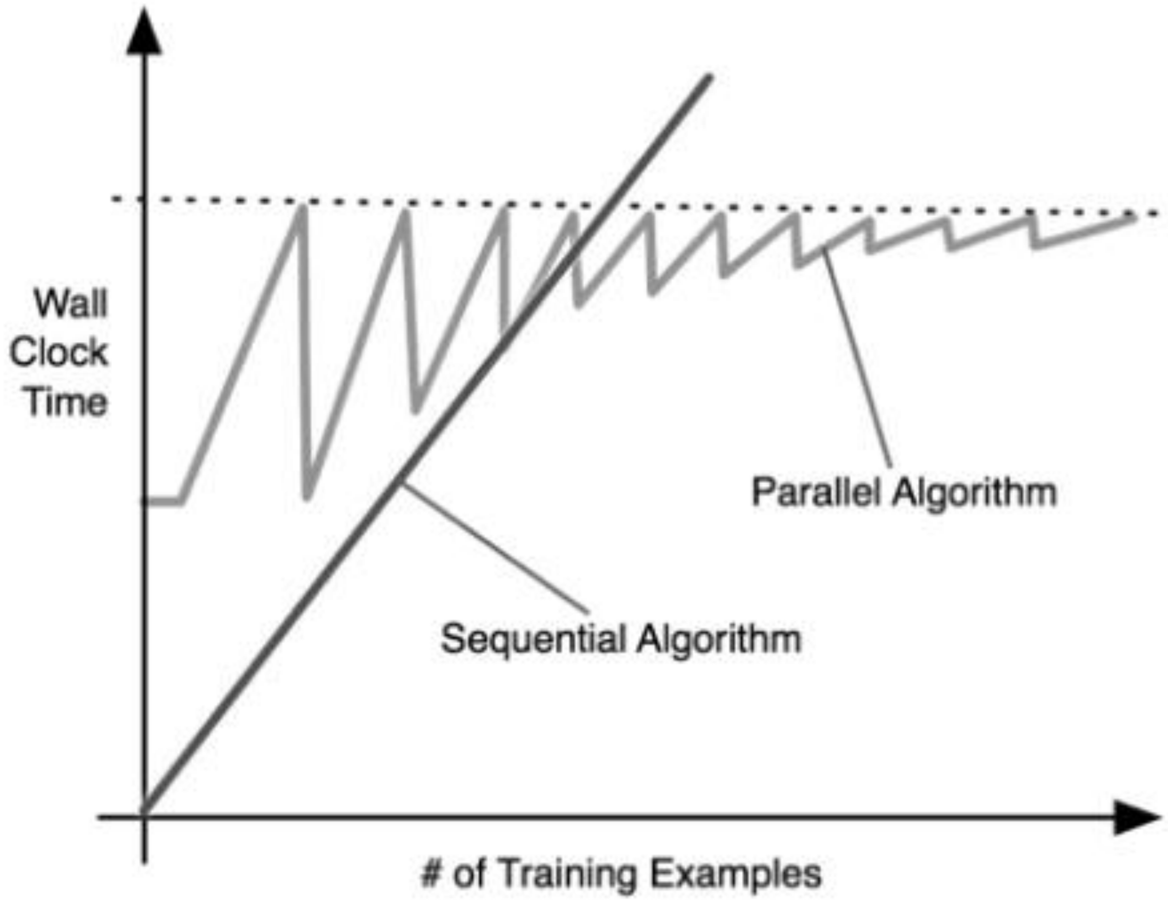
Parallel Speedup



need for scalable algorithms



from sequential to parallel



Hadoop in Detail

combine – a third function

- network is typically the most scarce resource in a distributed environment
- remember the wordcount example:
  - for each document, emit *(word,1)* tuples from the mapper
  - the reducer sums up all counts per word
  - → unnecessary network traffic, we could pre-aggregate all the tuples sent from a single mapper-instance

- MapReduce offers a third optional function **combine**

  (k, list(v)) → (k,list(v))

  - implementation must be **idempotent**
  - no guarantee that combine will see each tuple
  - very often, the reducer can also be used as combiner

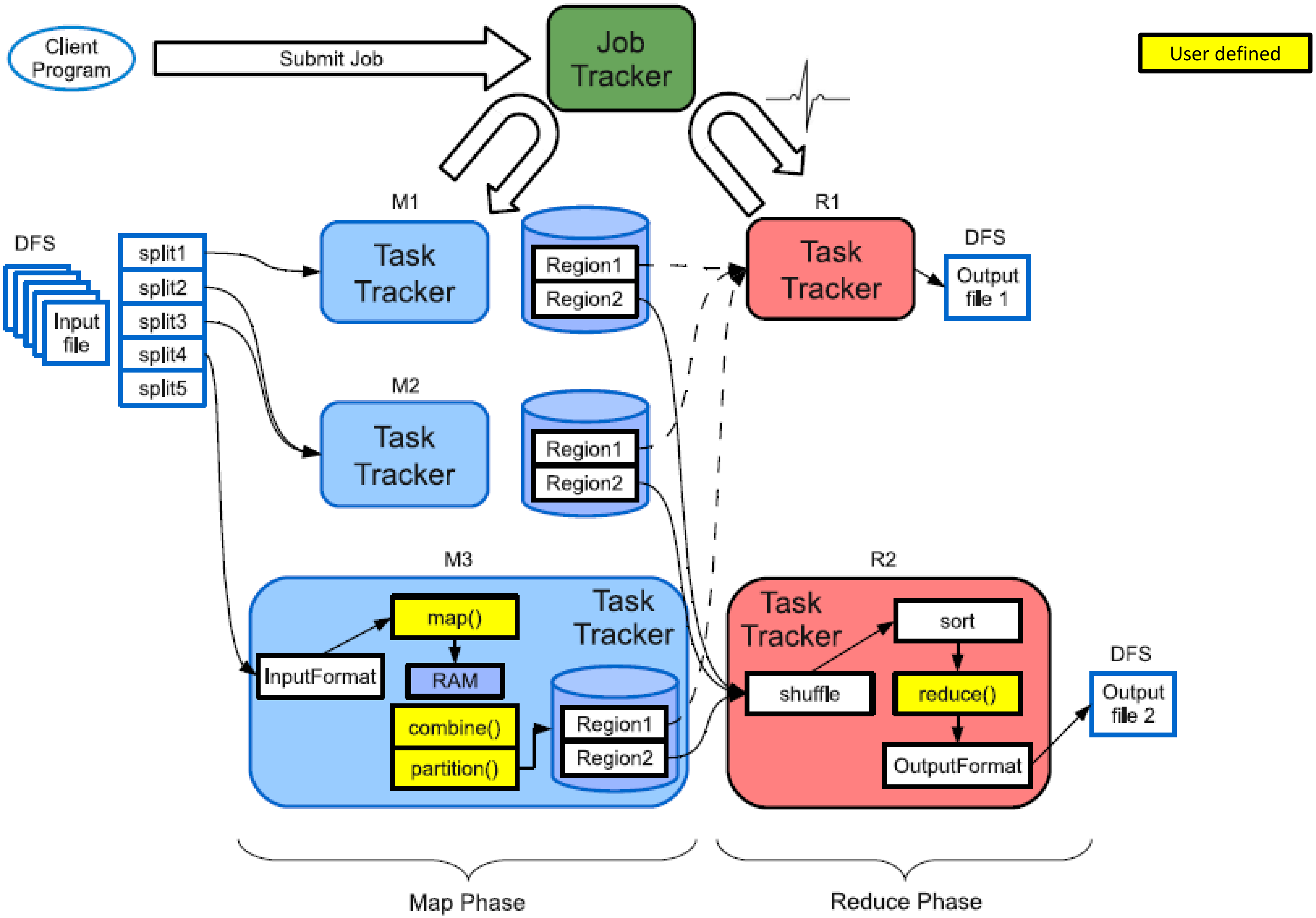
Hadoop Map/Reduce Engine

- Jobs are executed like a Unix pipeline:
  - `cat * | grep | sort | uniq -c | cat     > output`
  - `Input | Map   | Shuffle & Sort | Reduce | Output`

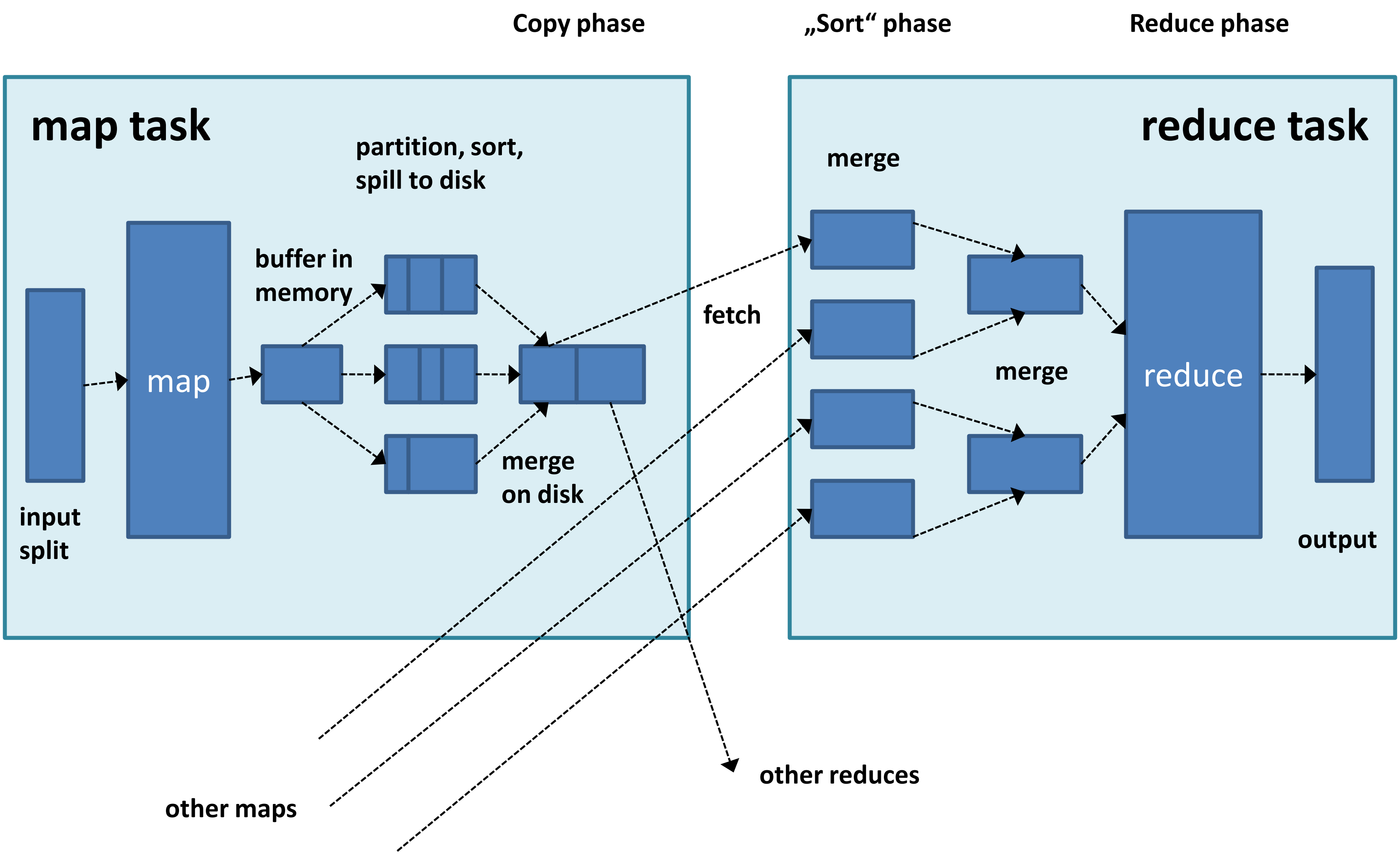
- Workflow
  - *input phase:* generates a number of FileSplits from input files (one per Map task)
  - *map phase:* executes a user function to transform input kv-pairs into a new set of kv-pairs
  - *sort & shuffle:* sort and distribute the kv-pairs to output nodes
  - *reduce phase:* combines all kv-pairs with the same key into new kv-pairs
  - *output phase* writes the resulting pairs to files

- All phases are distributed with many tasks doing the work
  - Framework handles scheduling of tasks on cluster
  - Framework handles recovery when a node fails

Hadoop Map/Reduce Engine



shuffle and sort



Secondary Sort

- suppose we run a large search engine and would like to find the top ten search queries per city from a file holding *(city, query, count)* tuples
- we would emit *(city, {query, count})* pairs in the mapper, but how would we find the the top ten queries in the reducer?
  - as we don‛t know in which order the *{query, count}* values arrive, we would have to buffer the ten best and iterate through all values ☹

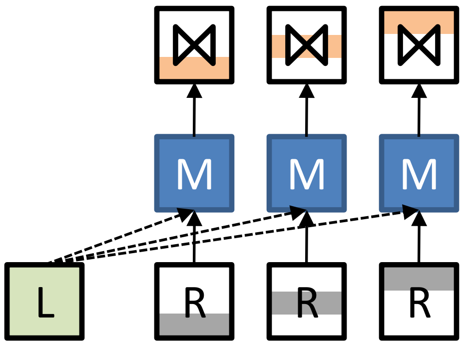
- Hadoop‛s Secondary Sort capabilities solve that problem!

  - create a combined key in the mapper *({city, count}, {query, count})*
  - Hadoop supports using different comparators for grouping and sorting
  - for grouping we only use the first key attribute: *{**city**, count}*
  - For sorting we use both *{**city**, **count**}*

Parallel Joins in MapReduce

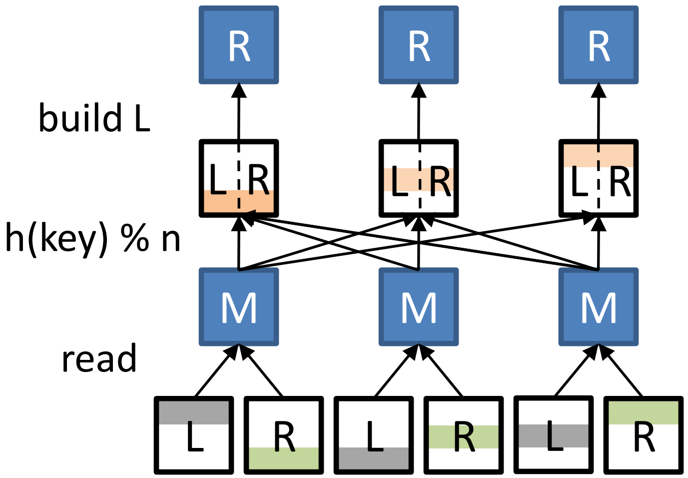
Broadcast Join

- Equi-Join: $L(A,X) \bowtie R(X,C)$
  - assumption: $|L| << |R|$



- Idea
  - broadcast L to each node completely before the map phase begins
  - Utilities like Hadoop's distributed cache can take this part
- Mapper
  - only over R
  - step 1: read assigned input split of R into a hash-table (build phase)
  - step 2: scan local copy of L and find matching R tuples (probe)
  - step 3: emit each such pair
  - Alternatively read L into Hash-Table, then read R and probe
- No need for partition / sort / reduce processing
  - Mapper outputs the final join result

Repartition Join (Reduce-side Join)



- Equi-Join: L(A,X) ⋈ R(X,C)
  - assumption: |L| < |R|

- Mapper L(A,X) ⋈ R(X,C)
  - identical processing logic for L and R
  - evaluate local predicates to filter unneeded tuples (optional)
  - emits each tuple once
  - the intermediate key is a pair of
    - the value of the actual join key X
    - an annotation identifying to which relation the tuple belongs to (L or R)

- Partition and sort
  - modulo division of the join key hash value
  - input is sorted primary on the join key, secondary on the relation name
  - output: a sequence of L(*i*), R(*i*) blocks of tuples for ascending join key *i*

- Reduce
  - collect all L-tuples for the current L(*i*) block in a hash map
  - combine them with each R-tuple of the subsequent R(*i*)-tuple block

Hadoop Ecosystem

Hive

- Data warehouse infrastructure built on top of Hadoop, providing:
  - Data Summarization
  - Ad hoc querying
- Simple query language: Hive QL (based on SQL)
- Extendable via custom mappers and reducers
- Subproject of Hadoop
- No „Hive format“
- http://hadoop.apache.org/hive/

Hive - Example

```
LOAD DATA INPATH `/data/visits` INTO TABLE visits


INSERT OVERWRITE TABLE visitCounts
SELECT url, category, count(*)
FROM visits
GROUP BY url, category;


LOAD DATA INPATH '/data/urlInfo' INTO TABLE urlInfo


INSERT OVERWRITE TABLE visitCounts
SELECT vc.*, ui.*
FROM visitCounts vc JOIN urlInfo ui ON (vc.url = ui.url);


INSERT OVERWRITE TABLE gCategories
SELECT category, count(*)
FROM visitCounts
GROUP BY category;


INSERT OVERWRITE TABLE topUrls
SELECT TRANSFORM (visitCounts) USING 'top10';
```

JAQL

- Higher level query language for JSON documents
- Developed at IBM‘s Almaden research center
- Supports several operations known from SQL
  - Grouping, Joining, Sorting
- Built-in support for
  - Loops, Conditionals, Recursion
- Custom Java methods extend JAQL
- JAQL scripts are compiled to MapReduce jobs
- Various I/O
  - Local FS, HDFS, Hbase, Custom I/O adapters
- http://www.jaql.org/

JAQL - Example

```
registerFunction(„top“, „de.tuberlin.cs.dima.jaqlextensions.top10“);

$visits = hdfsRead(„/data/visits“);

$visitCounts =
$visits
-> group by $url = $
       into { $url, num: count($)};

$urlInfo = hdfsRead(„data/urlInfo“);

$visitCounts =
join $visitCounts, $urlInfo
where $visitCounts.url == $urlInfo.url;

$gCategories =
$visitCounts
-> group by $category = $
       into {$category, num: count($)};

$topUrls = top10($gCategories);

hdfsWrite(“/data/topUrls”, $topUrls);
```

Pig

- A platform for analyzing large data sets

- Pig consists of two parts:
  - PigLatin: A Data Processing Language
  - Pig Infrastructure: An Evaluator for PigLatin programs
  - Pig compiles Pig Latin into physical plans
  - Plans are to be executed over Hadoop

- Interface between the declarative style of SQL and low-level, procedural style of MapReduce

- http://hadoop.apache.org/pig/

Pig - Example

```
visits      = load '/data/visits' as (user, url, time);

visitCounts = foreach visits generate url, count(visits);

urlInfo     = load '/data/urlInfo'
               as (url, category, pRank);

visitCounts = join visitCounts by url, urlInfo by url;

gCategories = group visitCounts by category;

topUrls     = foreach gCategories
               generate top(visitCounts,10);

store topUrls into '/data/topUrls';
```

Example taken from:
"Pig Latin: A Not-So-Foreign Language For Data Processing" Talk, Sigmod 2008

Mahout

- a scalable machine learning library
- Scalable to reasonably large data sets: the core algorithms are implemented on top of Apache Hadoop using the map/reduce paradigm.
- Currently Mahout supports mainly four use cases:
  - Recommendation mining takes users' behavior and from that tries to find items users might like.
  - Clustering takes e.g. text documents and groups them into groups of topically related documents.
  - Classification learns from exisiting categorized documents what documents of a specific category look like and is able to assign unlabelled documents to the (hopefully) correct category.
  - Frequent itemset mining takes a set of item groups (terms in a query session, shopping cart content) and identifies, which individual items usually appear together.

ZooKeeper

- open-source server which enables highly reliable distributed coordination.

- a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services. All of these kinds of services are used in some form or another by distributed applications.

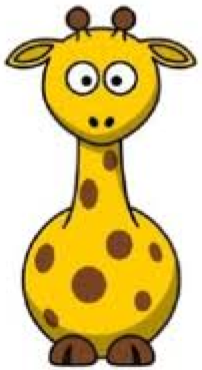
HBase

- HBase is open-source, distributed, versioned, column-oriented store modeled after Google's Bigtable

- features
  - random, realtime read/write access to your Big Data.
  - hosting of very large tables -- billions of rows X millions of columns -- atop clusters of commodity hardware
  - provides Bigtable-like capabilities on top of Hadoop and HDFS

Giraph

- Large scale graph processing system ontop of Hadoop
- implementation of Google Pregel
- adds fault-tolerance with the use of ZooKeeper as its centralized coordination service.



- Giraph follows the bulk-synchronous parallel model relative to graphs where vertices can send messages to other vertices during a given superstep.