732A54
Big Data Analytics

LINKÖPING UNIVERSITY

Introduction to MapReduce

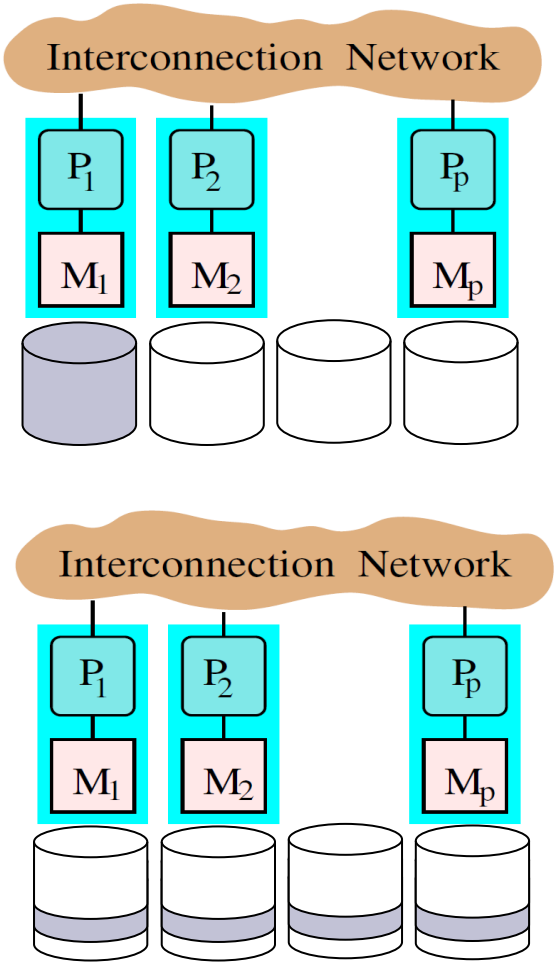
Christoph Kessler

IDA, Linköping University

Towards Parallel Processing of Big-Data

Big Data …

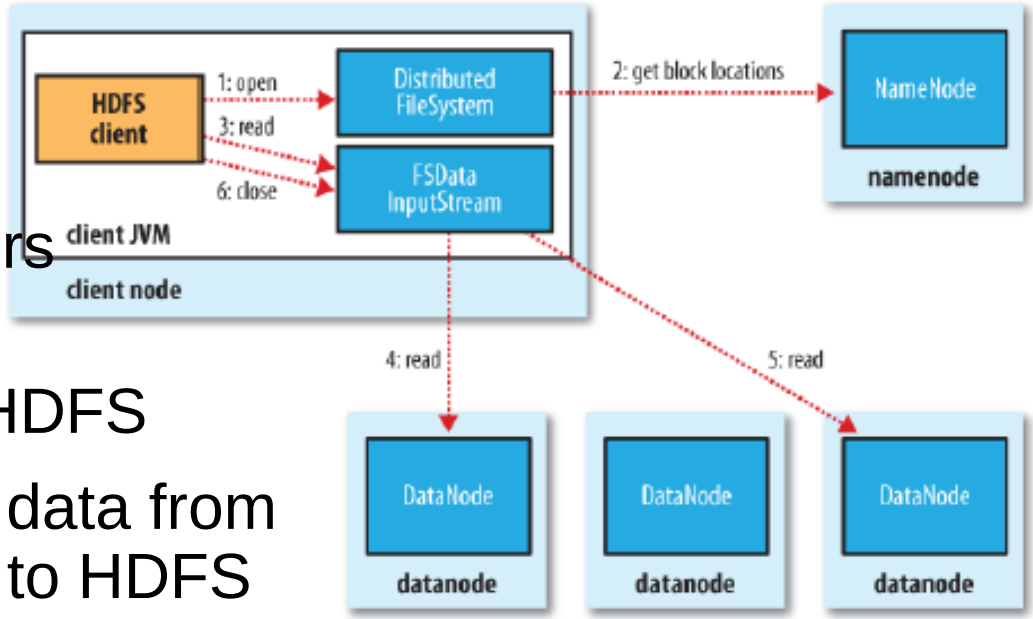
- too large to be read+processed in reasonable time by 1 server only
- too large to fit in main memory at a time
- Usually residing on **secondary storage** (local or remote)
  - Storage on a single hard disk (sequential access) would prevent parallel processing
  - Use lots of servers with lots of hard disks
    - i.e., standard server nodes in a cluster
  - **Distribute** the data across nodes to allow for parallel access



Distributed File System

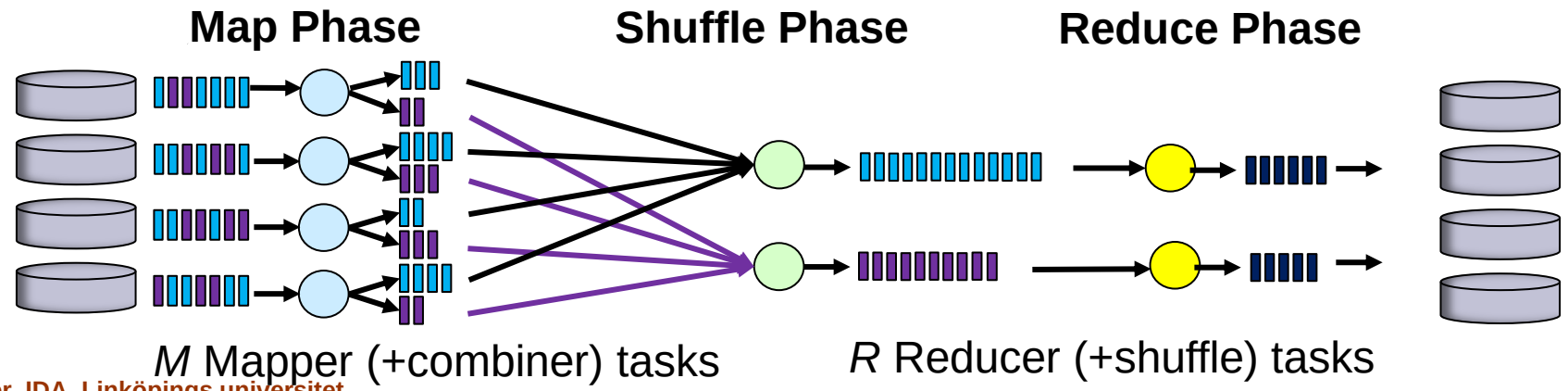
MapReduce works atop a **distributed file system**

- Large files are **distributed** (”*sharded*” = split into blocks of e.g. 64MB (shards) and spread out across cluster nodes)
  - Parallel access possible
  - Faster access to local chunks (higher bandwidth, lower latency)
- Also, replicas for fault tolerance
  - E.g. 3 copies of each block on different servers
  - Examples: Google GFS, Hadoop HDFS
- May need to first copy the data from ordinary (host) file system to HDFS



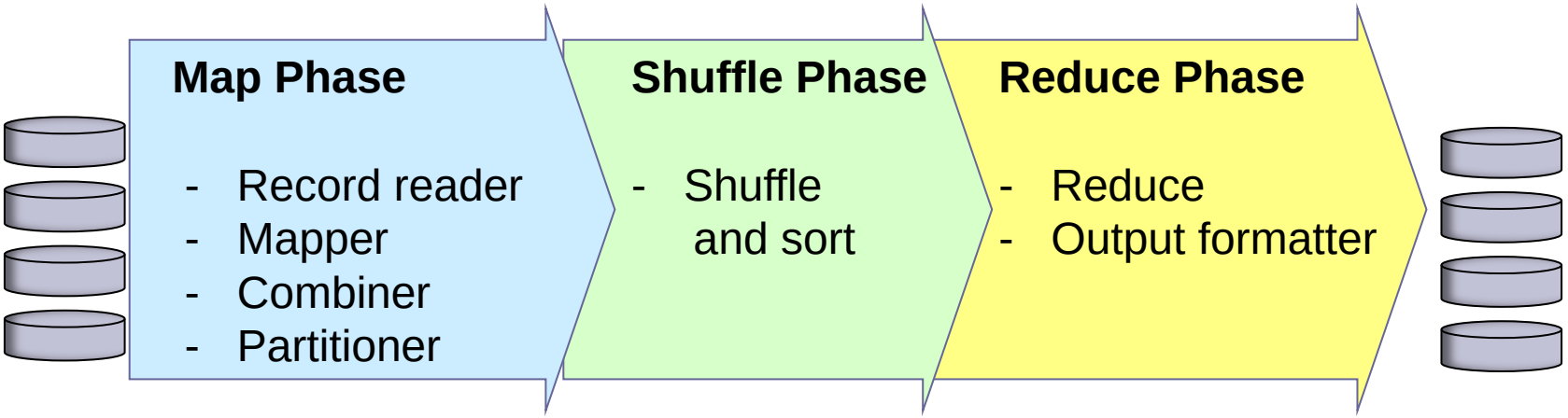
MapReduce Programming Model

- Designed to operate on LARGE distributed input data sets stored e.g. in HDFS nodes
- Abstracts from parallelism, data distribution, load balancing, data transfer, fault tolerance
- Implemented in **Hadoop** and other frameworks
- Provides a high-level parallel programming construct (= a skeleton) called **MapReduce**
  - A generalization of the data-parallel *MapReduce* skeleton of Lect. 1
  - Covers the following algorithmic design pattern:



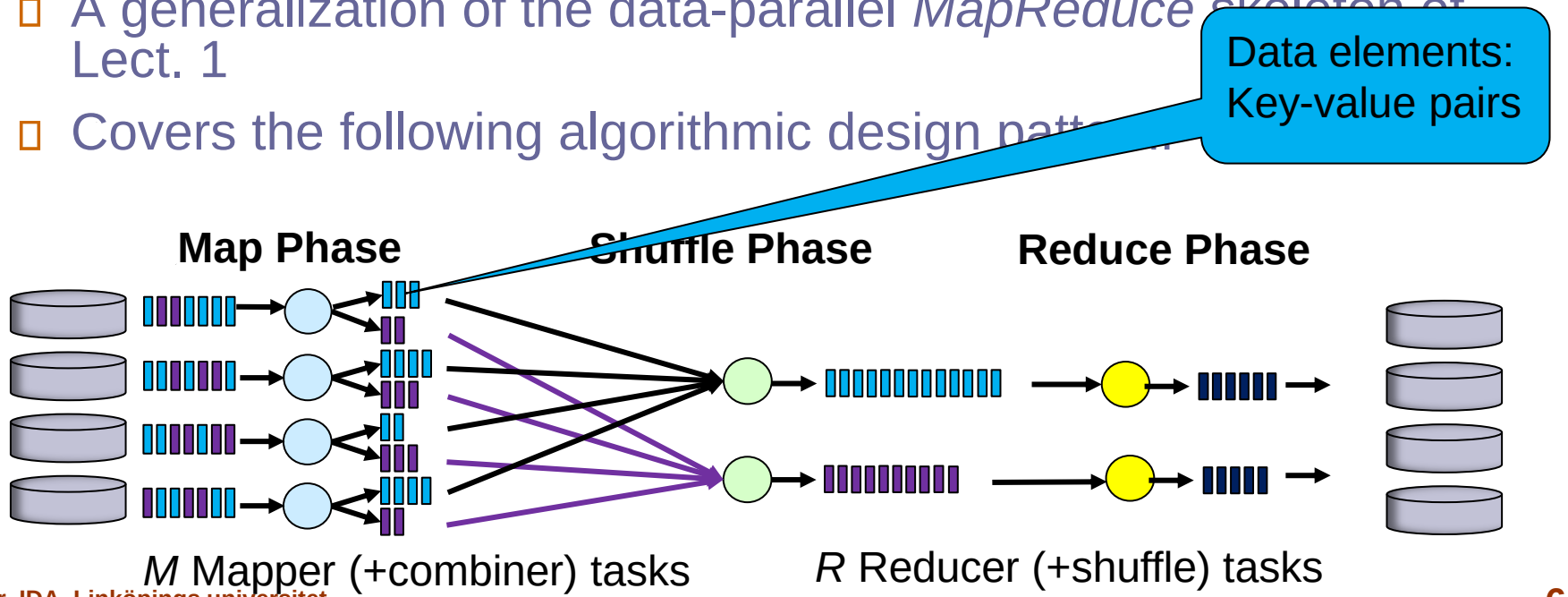
MapReduce Programming Model

- Designed to operate on LARGE input data sets stored e.g. in HDFS nodes
- Abstracts from parallelism, data distribution, load balancing, data transfer, fault tolerance
- Implemented in **Hadoop** and other frameworks
- Provides a high-level parallel programming construct (= a skeleton) called **MapReduce**
  - A generalization of the data-parallel MapReduce skeleton of Lect. 1
  - Covers the following algorithmic design pattern:



MapReduce Programming Model

- Designed to operate on LARGE distributed input data sets stored e.g. in HDFS nodes
- Abstracts from parallelism, data distribution, load balancing, data transfer, fault tolerance
- Implemented in **Hadoop** and other frameworks
- Provides a high-level parallel programming construct (= a skeleton) called **MapReduce**
  - A generalization of the data-parallel *MapReduce* skeleton of Lect. 1
  - Covers the following algorithmic design patt...

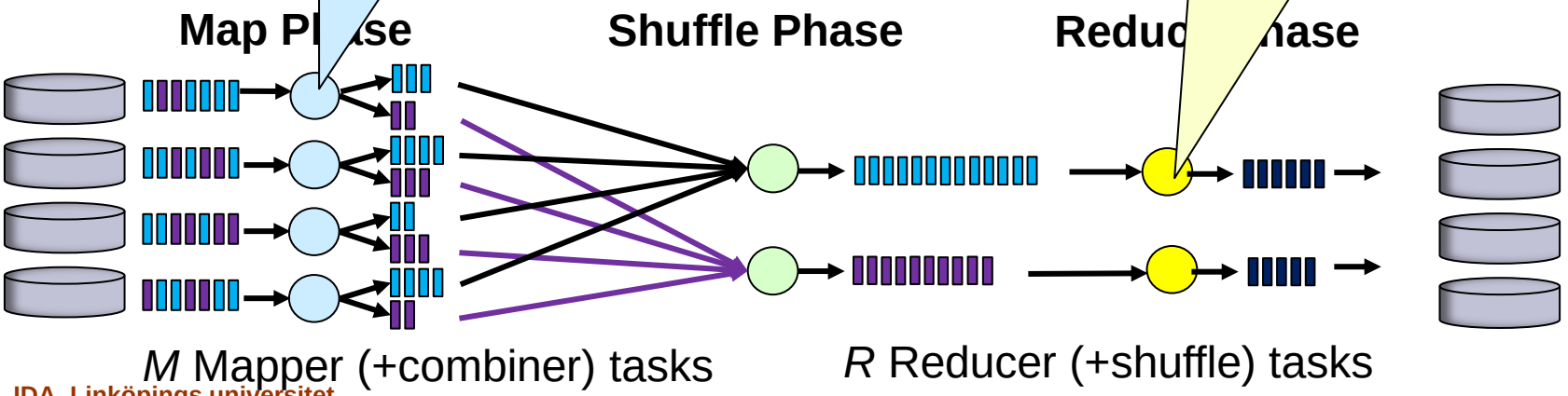


MapReduce Programming Model

- Designed to operate on LARGE distributed input data sets stored e.g. in HDFS nodes
- Abstracts from parallelism, data distribution, load balancing, data transfer, fault tolerance
- Implemented in **Hadoop** and other frameworks
- Provides a high-level parallel programming construct (= a skeleton) called **MapReduce**
  - A gene… par… of Lect. 1…
  - Covers the …owing algorithmic design pattern

Map function:
K_1 **x** V_1 → **List**(K_2 **x** V_2)

Reduce function:
K_2 **x List**(K_2 **x** V_2) → **List**(V_2)



Record Reader

Map Phase
- **Record reader**
- Mapper
- Combiner
- Partitioner

- Parses an input file block from stdin into **key-value pairs** that define input data records
  - **Key** in K_1 is typically positional information (location in file)
  - **Value** in V_1 = chunk of input data that composes a record

Mapper

Map Phase
- Record reader
- **Mapper**
- Combiner
- Partitioner

- Applies a **user-defined function** to each element (i.e., key/value pair coming from the Record reader).
  - Examples:
    - Filter function – drop elements that do not fulfill a constraint
    - Transformation function – calculation on each element
- Produces a list of zero or more new key/value pairs = **intermediate elements**
  - Key in K_2: index for grouping of data
  - Value in V_2: Data to be forwarded to reducer
  - Buffered in memory

Combiner

Map Phase
- Record reader
- Mapper
- **Combiner**
- Partitioner

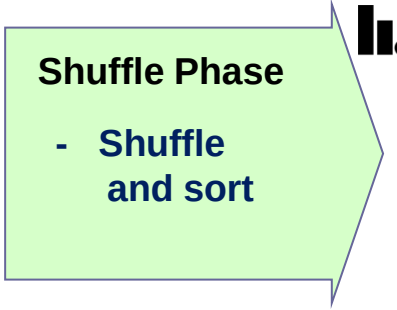
- An optional local reducer
  run in the mapper task as postprocessor
- Applies a **user-provided function** to aggregate values in the intermediate elements of one mapper task
- Reduction/aggregation could also be done by the reducer, but local reduction can improve performance considerably
  - *Data locality* – key/value pairs still in cache resp. memory of same node
  - *Data reduction* – aggregated information is often smaller
- Applicable if the user-defined *Reduce function* is commutative and associative
- Recommended if there is significant repetition of intermediate keys produced by each Mapper task

Partitioner

Map Phase
- Record reader
- Mapper
- Combiner
- **Partitioner**

- Splits the intermediate elements from the mapper/combiner into shards (64MB blocks stored in local files)
  - one shard per reducer
  - Default: *element* to *hashCode*(*element*.key) modulo *R* for even (round-robin) distribution of elements
    - Usually good for load balancing
- Writes the shards to the local file system

Shuffle-and-sort



- Downloads the needed files written by the partitioners to the node on which the reducer is running
- Sort the received (key,value) pairs by key into one list
  - Pairs with equivalent keys will now be next to each other (groups)
  - To be handled by the reducer
- No customization here beyond how to sort and group by keys

Reducer

Reduce Phase

- **Reducer**
- Output formatter

- Run a **user-defined reduce function** once per key grouping
  - Can aggregate, filter, and combine data
  - Output: 0 or more key/value pairs sent to output formatter.

Output Formatter

Reduce Phase

- Reducer
- **Output formatter**

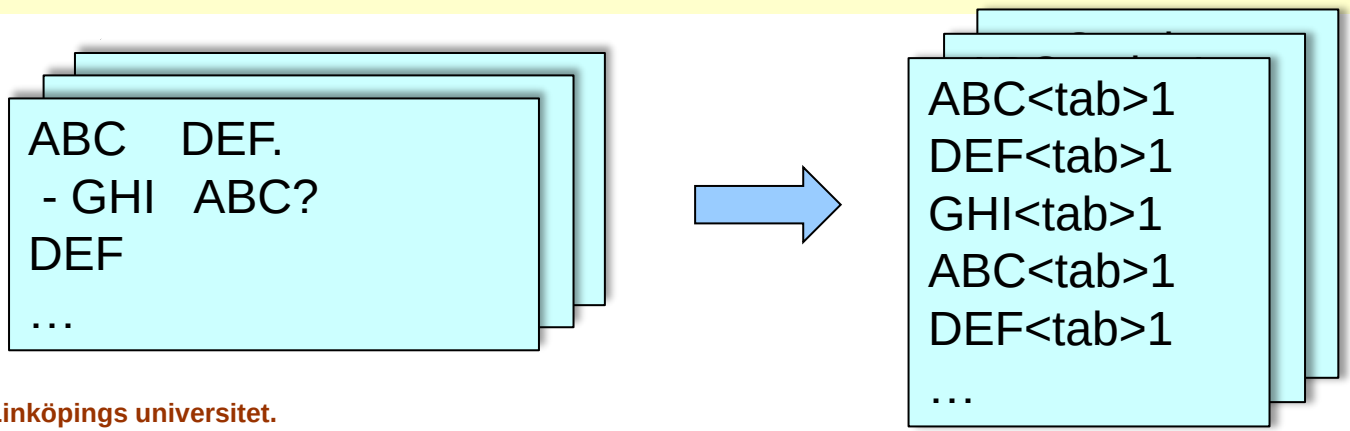
- Translates the final (key,value) pair from the reduce function and writes it to stdout → to a file in HDFS
  - Default formatting (key <TAB> value <NEWLINE>) can be customized

Example: Word Count

- Python code for the **Mapper** task:

```
import sys
for line in sys.stdin:
    # for each input document:
    # remove leading and trailing whitespace:
    line = line.strip()
    # split the line into words:
    words = line.split()
    # increase counters:
    for word in words:
        print '%s\t%s' % (word, 1)
```

Python code adapted from
MapReduce tutorial, Princeton U., 2015

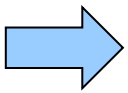


Example: Word Count

- Python code for the **Combiner** task:

```
import sys
for line in sys.stdin:
    # for each document create dictionary of words:
    wordcounts = dict()
    line = line.strip()
    words = line.split()
    for word in words:
        if word not in wordcounts.keys(): wordcounts[word] = 1
        else: wordcounts[word] += 1
    # emit key-value pairs only for distinct words per document
    for w in wordcounts.keys():
        print '%s\t%s' % (w, wordcounts[w])
```

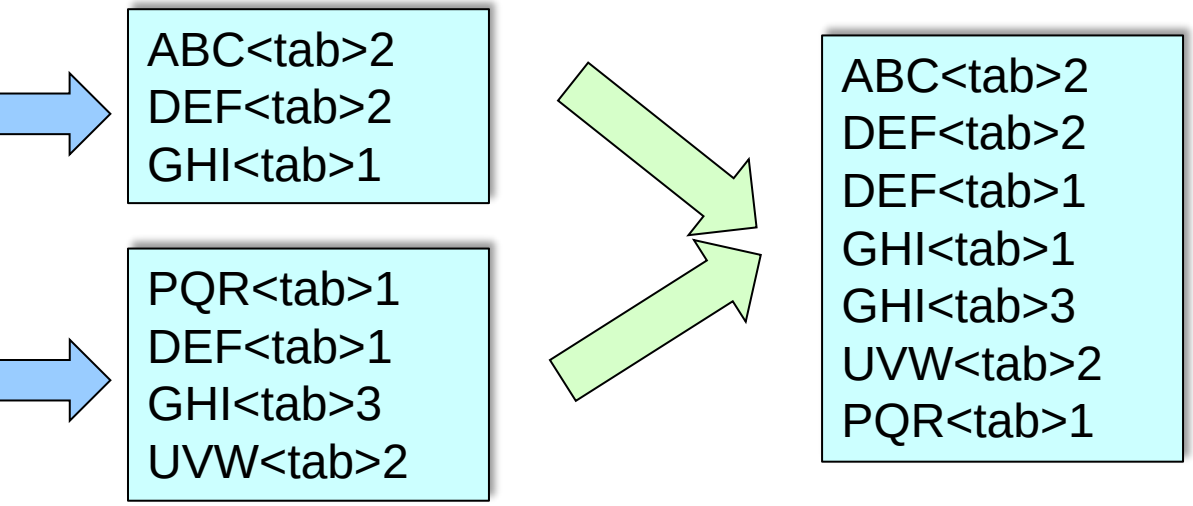
ABC<tab>1
DEF<tab>1
GHI<tab>1
ABC<tab>1
DEF<tab>1



ABC<tab>2
DEF<tab>2
GHI<tab>1

Example: Word Count

- Effect of Shuffle-And-Sort:



Example: Word Count

- Python code for the **Reducer** task:

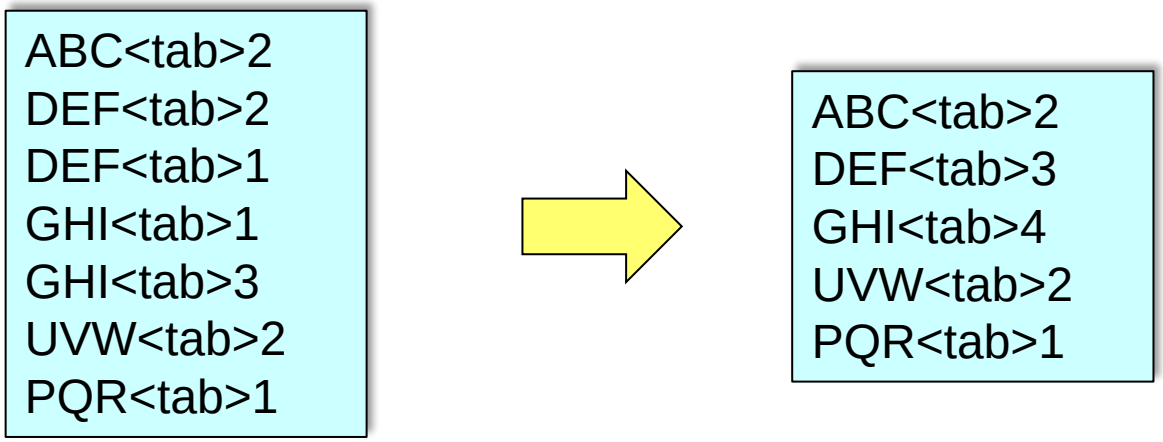
```
import sys
current_word = None
current_count = 0
word = None
for line in sys.stdin:
    # remove leading and trailing whitespace
    line = line.strip()
    # parse the input we got from mapper:
    word, count = line.split('\t', 1)
    # convert count from string to int:
    try:
        count = int(count)
    except ValueError:
        # silently ignore invalid line
        continue
    ....
    if current_word == word:
        current_count += count
    else:
        # new word – print tuple for
        #   the previous one to stdout:
        if current_word:
            print '%s\t%s' %
                        (current_word,
                         current_count)
        current_count = count
        current_word = word

# loop done, write the last tuple:
if current_word == word:
    print '%s\t%s' % (current_word,
                      current_count)
```

NB words come in sorted order – if word is same as the last one, just add its count

Example: Word Count

- Effect of Reducer:



Special Cases of MapReduce

Map only (Reduce is identity function)

- **Data Filtering**
  - E.g. distributed grep
- **Data Transformation**

Shuffle-and-sort only:

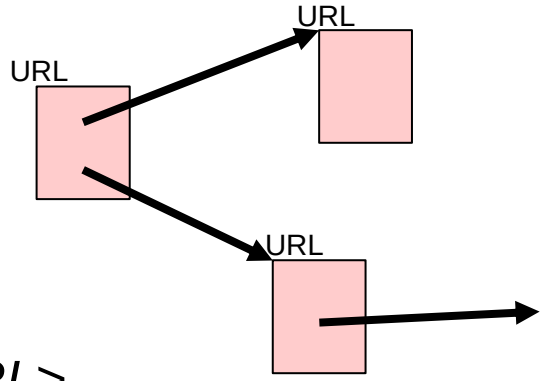
- **Sorting values by key**
  - Mapper extracts *key* from *record* and forms <*key*, *record*> pairs
  - Shuffle-and-sort phase does the sorting by *key*

Reduce only: (Map is identity function, Combiner for local reduce)

- **Reductions (summarizations)**:
  - Find global maximum/minimum, global sum, average, median, standard deviation, …
  - Find top-10

Further Examples for MapReduce

- **Count URL frequencies** (a variant of wordcount)
  - Input: logs of web page requests <*URL*, 1>
  - Reduce function adds together all values for same *URL*

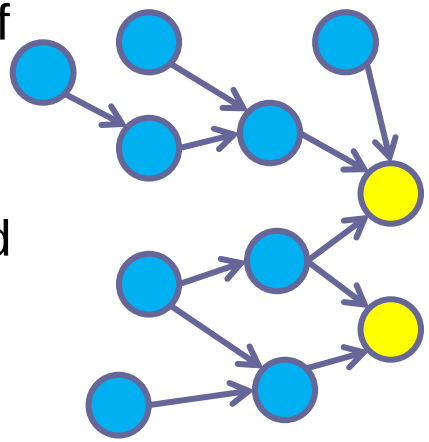


- **Construct reverse web-link graph**
  - Input: <*sourceURL*, *targetURL*> pairs
  - Mapper reverses: <*targetURL*, *sourceURL*>
  - Shuffle-and-sort → <*targetURL*, list of all URLs pointing to *targetURL*>
  - no reduction → Reduce function is identity function

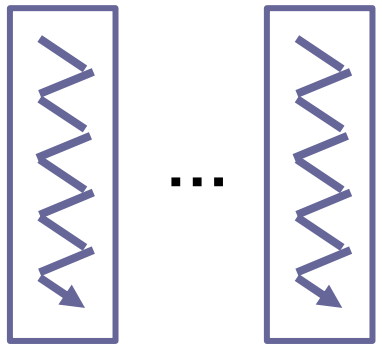
- **Indexing web documents**
  - Input: list of documents (e.g. web pages)
  - Mapper parses documents and builds sequences <*word*, *documentID*>*
  - Shuffle-and-sort produces for each *word* a list of all *documentID*s where *word* occurs (Reduce function is identity)

MapReduce Implementation / Execution Flow

- User application calls **MapReduce** and waits.
- MapReduce library implementation **splits** the input data (if not already done) in *M* blocks (of e.g. 64MB) and **creates** *P* MapReduce processes on different cluster nodes:
  1 master and *P*-1 workers.
- **Master creates** *M* mapper tasks and *R* reducer tasks, and **dispatches** them to idle workers (dynamic scheduling)
  - Worker executing a **Mapper** task reads its block of input, applies the *Map (*and local *Combine) function*, and buffers (key,value) pairs in memory.
    Buffered pairs are periodically written to local disk, locations of these files are sent to Master.
  - Worker executing a **Reducer** task is notified by Master about locations of intermediate data to shuffle+sort and **fetches** them by remote memory access request, then **sorts** them by key (K_2).
    It applies the *Reduce function* to the sorted data and appends its output to a local file.
- When all mapper and reducer tasks have completed, the master wakes up the user program and returns the locations of the *R* output files.



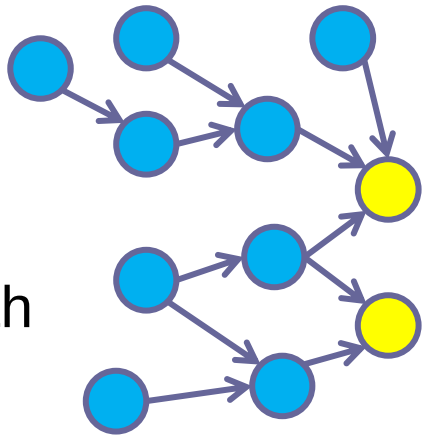
Run-time scheduler



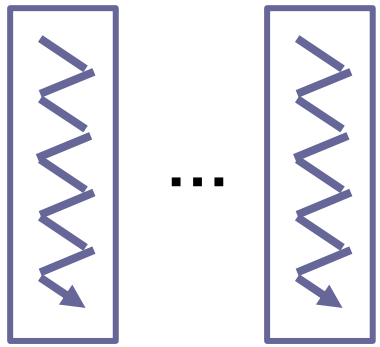
Worker processes
on different cluster nodes

MapReduce Implementation: Fault Tolerance

- **Worker failure**
  - Master pings every worker periodically.
  - Master marks a dead worker's tasks for re-execution → eventually reassigned to other workers
    - Completed map tasks (as their local files with intermediate data are no longer accessible) and unfinished map and reduce tasks
    - Reducer tasks using data from a failed map task are informed by master about the new worker



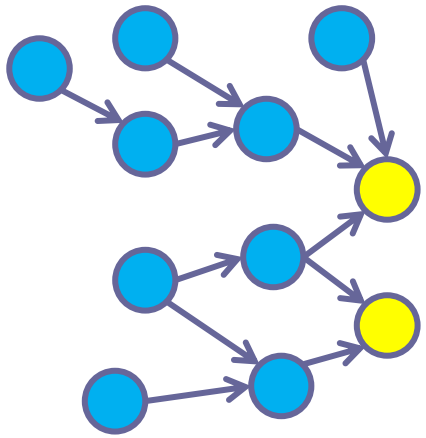
- **Master failure**
  - Less likely (1 Master, *P*-1 Workers)
  - Use checkpointing, a new master can restart from latest checkpoint

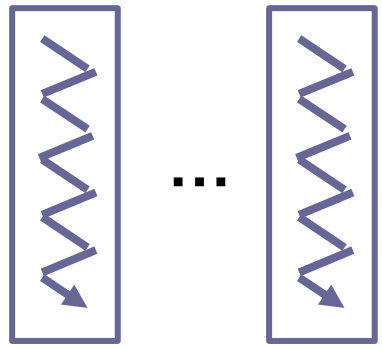


Worker processes
on different cluster nodes

MapReduce Implementation: Data Locality

- For data storage fault tolerance, have 3 copies of each 64MB data block, each stored on a different cluster node
- Master uses **Locality-aware scheduling:**
  - Schedule a mapper task to a worker node holding one copy of its input data block
  - Or on a node that is near a copy holder (e.g. a neighbor node in the network topology)



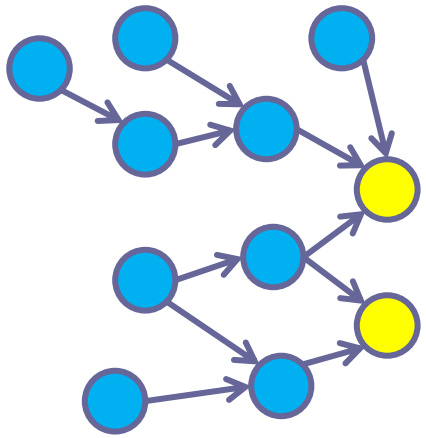


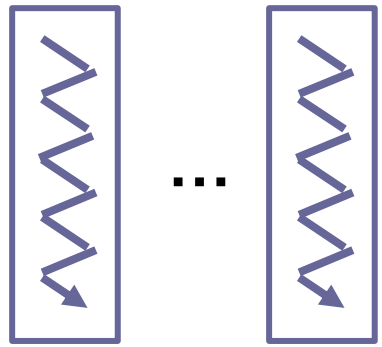
Worker processes
on different cluster nodes

MapReduce Implementation: Granularity

Numbers *M, R* and work of tasks (block size) might be tuned

- Default: *M = input file size / block size*
  - User can set other value
- *M, R* should be >> *P*
  - For flexibility in dynamic load balancing
  - Hadoop recommends ~10…100 mappers per cluster node, or more if lightweight
- Not too large, though…
  - ~ *M+R* scheduling decisions by master
  - Block size should be reasonably large (e.g. 64MB) to keep relative impact of communication and task overhead low





Worker processes
on different cluster nodes

References

- J. Dean, S. Ghemawat: MapReduce: Simplified Data Processing on Large Clusters. Proc. *OSDI* 2004. Also in: *Communications of the ACM* 51(1), 2008.
- D. Miner, A. Shook: *MapReduce Design Patterns*. O'Reilly, 2012.
- Apache Hadoop: https://hadoop.apache.org

Questions for Reflection

- A MapReduce computation should process 12.8 TB of data in a distributed file with block (shard) size 64MB. How many mapper tasks will be created, by default? (Hint: 1 TB (Terabyte) = 10^{12} byte)
- Discuss the design decision to offer just one MapReduce construct that covers both mapping, shuffle+sort and reducing. Wouldn't it be easier to provide one separate construct for each phase? What would be the performance implications of such a design operating on distributed files?
- Reformulate the wordcount example program to use no Combiner.
- Consider the local reduction performed by a Combiner:
  Why should the user-defined Reduce function be associative and commutative? Give examples for reduce functions that are associative and commutative, and such that are not.
- Extend the wordcount program to discard words shorter than 4 characters.
- Write a wordcount program to only count *all* words of odd and of even length. There are several possibilities.
- Show how to calculate a database join with MapReduce.
- Sometimes, workers might be temporarily slowed down (e.g. repeated disk read errors) without being broken. Such workers could delay the completion of an entire MapReduce computation considerably.
  How could the master speed up the overall MapReduce processing if it observes that some worker is late?