SAARLAND UNIVERSITY
COMPUTER SCIENCE
Information Systems Group

Database Systems

WS 08/09

Prof. Dr. Jens Dittrich

Chair of Information Systems Group
http://infosys.cs.uni-saarland.de

Topics (5/6)

- large systems
  - global scale data management
  - map/reduce
  - pig
  - search engines
  - data warehouses and OLAP
- write-optimized system concepts
  - OLTP
  - publish/subscribe
  - streaming
  - moving objects
- management of geographical data
  - basic concepts
  - GIS, google maps

map/reduce

Introduction.

Motivation: Big Datasets

- Google web crawl
- Google web logs
- satellite data of all kinds (weather, astronomy, etc.)
- bioinformatics
- physics: collider data, e.g., CERN

Requirements

- index creation, e.g. inverted index
- click-log analysis
- looking for interesting patterns:
  - searching through the click-log
  - data mining (outliers, clusters, classification)
- file-oriented handling
- **flexible or no schema required**
- often string-oriented processing (but not necessarily)

How to Handle?

- how to handle these massive amounts of data?
- how to even inspect these datasets?
- obviously huge degree of parallelism
  - (ten-) thousands of machines
- Google strategy:
  - cheap standard hardware
  - shared nothing
  - geographically distributed
  - several replicas
- hardware failures will be the rule not the exception
- algorithms have to live with it

Goals

- provide users an **easy** programming model
- non-expert users should be able to make use of massive computing power and parallelism
- users should not fiddle around with
  - load balancing
  - data distribution (partitioning and replication)
  - synchronization of different threads
  - messaging
  - failover code
  - etc.

map() in python

- map() applies a unary function to each item in the output list
- the output is a list of function call results
- if the input list has N elements, the output list will have N elements
- each function call effects exactly one list element and is side-effect free
- **=> we could execute all N function calls concurrently**

Terminal — Python — 73×30

```
[jens-dittrichs-macbook-pro:~] jens% python
Python 2.5.1 (r251:54863, Jul 23 2008, 11:00:16)
[GCC 4.0.1 (Apple Inc. build 5465)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> def f(x): return pow(x,2)
...
>>> f(2)
4
>>> f(3)
9
>>> myList = range (1,20)
>>> myList
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
>>> map( f, myList )
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256, 289,
 324, 361]
>>> def g(x): return x % 2 != 0 and x % 3 != 0
...
>>> g(2)
False
>>> g(5)
True
>>> map (g, myList )
[True, False, False, False, True, False, True, False, False, False, True,
 False, True, False, False, False, True, False, True]
>>>
```

Counting words in Python: map()

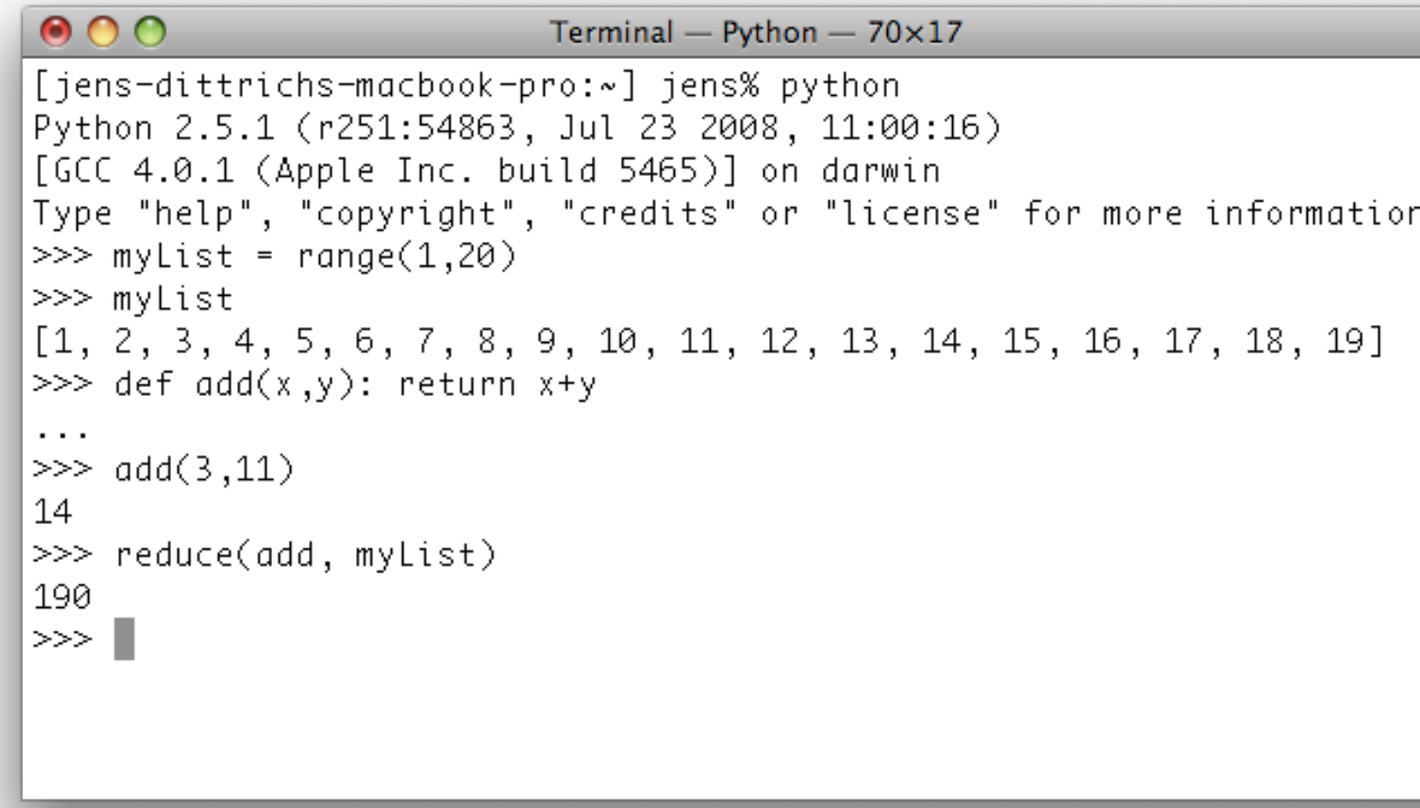
Terminal — Python — 77×25

```
Python 2.5.1 (r251:54863, Jul 23 2008, 11:00:16)
[GCC 4.0.1 (Apple Inc. build 5465)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> str = "this is a test\nthat is a rather\nstupid test\nand it is true that
 it does not\ncontain too much information."
>>> print str
this is a test
that is a rather
stupid test
and it is true that it does not
contain too much information.
>>> lines = str.split("\n")
>>> print lines
['this is a test', 'that is a rather', 'stupid test', 'and it is true that it
 does not', 'contain too much information.']
>>> output = []
>>> for line in lines:
...     output += map(lambda counter: [counter,1], line.split(" "))
...
>>> print output
[['this', 1], ['is', 1], ['a', 1], ['test', 1], ['that', 1], ['is', 1], ['a',
 1], ['rather', 1], ['stupid', 1], ['test', 1], ['and', 1], ['it', 1], ['is',
 1], ['true', 1], ['that', 1], ['it', 1], ['does', 1], ['not', 1], ['contain'
, 1], ['too', 1], ['much', 1], ['information.', 1]]
>>>
```

- mapping to (key, value)-pairs
- **keys**: strings
- **values**: word count (in this case always 1)

reduce() in python

- reduce() applies a binary function to the first pair of items in the list
- then it applies the same binary function to the result and the next item
- and so on...
- here: (((((((1+2)+3)+4)+5)+6)+7)+8)...
- however, we could also execute this as
- (((1+2)+(3+4))+(5+6))+(7+8)
- or ((1+2)+(3+4))+((5+6)+(7+8)), associativity
- or ((5+6)+(7+8))+ ((1+2)+(3+4)), commutativity
- **=> we could execute each bracket concurrently**



Counting words in Python: reduce()

Terminal — Python — 78×31

```
>>> dict = {}
>>> for item in output:
...     if dict.has_key(item[0]):
...             dict[item[0]] += [item[1]]
...     else:
...             dict[item[0]] = [item[1]]
...
>>> print dict
{'a': [1, 1], 'and': [1], 'not': [1], 'information.': [1], 'that': [1, 1], 'th
is': [1], 'much': [1], 'is': [1, 1, 1], 'it': [1, 1], 'rather': [1], 'stupid':
 [1], 'does': [1], 'contain': [1], 'test': [1, 1], 'true': [1], 'too': [1]}
>>> for item in dict.items():
...     item[0], reduce(lambda x,y: x+y, item[1])
...
('a', 2)
('and', 1)
('not', 1)
('information.', 1)
('that', 2)
('this', 1)
('much', 1)
('is', 3)
('it', 2)
('rather', 1)
('stupid', 1)
('does', 1)
('contain', 1)
('test', 2)
('true', 1)
('too', 1)
>>>
```

- collect tuples that share the same key
- then: one reduce()-call for **each** distinct key
- each call could be executed in parallel

Original word count example

```
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));
```

- two central functions need to be provided
- everything else handled by the framework

map/reduce

- the previous example illustrates the core idea of Google's map/reduce framework
- main observation: many data managing tasks can be expressed with two functions
  - **map**
    - maps an input (key,value) pair to a **list** of intermediate key-value pairs
    - map (k1, v1) -> list(k2, v2)
  - **reduce**
    - takes as input a key and a **list** of values for that key
    - maps this input to a **list** of values
    - reduce (k2, list(v2)) -> list(v2)
- these concepts are borrowed from functional programming
- but not exactly similar

Role of the map reduce library

- groups together all intermediate values associated with the same key
- passes these values to reduce()
- user may also specify some tuning parameters
- that‘s it!

Example: Distributed Grep

- map()
  - emits a line if it matches a supplied pattern
- reduce()
  - identity function
  - just copies input to the output
- discussion
  - map() acts as a filter
  - map basically implements a selection/filter operator
  - in DB terminology: parallel filter on horizontally partitioned data

Example: Count of URL Access Frequency

- map()
  - processes logs of web pages
  - emits (URL, 1) for each log input
- reduce()
  - adds together all values for the same URL
  - emits (URL, total count)
- Discussion
  - very similar to word count example
  - in DB terminology: simple aggregation

Example: Reverse Web-Link Graph

- Web = graph:
  - URLs = nodes
  - hyperlinks = edges
- Goal: find all URLs pointing to an individual URL
- map()
  - outputs (targetURL, sourceURL) pairs for each link to a target URL found in a source URL
- reduce()
  - concatenates the list of all source URLs associated with a given target URL
  - emits the pair (targetURL, list(sourceURL))
- Discussion
  - simple aggregation use-case
  - concatenates occurences in a list

Example: Term-Vector per Host

- term vector: summarizes most important words that occur in a document or set of documents
- desired output: (word, frequency)
- map()
  - emits (hostname, term vector) for each input document
  - hostname extracted from URL
- reduce()
  - aggregates individual tuples on host name
  - throws away infrequent terms
  - outputs (hostname, term vector)
- Discussion
  - simple aggregation use-case

Example: Inverted Index

- Goal: compute inverted text index for the entire Web (as Google sees it)
- map()
  - parses each document
  - emits (word, document ID) pairs
- reduce()
  - accepts all pairs **for a given word**
  - sorts the corresponding document IDs
  - emits (word, list(document ID)) pair
  - set of all output pairs forms a simple inverted index
  - again a form of aggregation

Example: Inverted Index plus Positions

- Goal: compute inverted text index for the entire Web **including word positions (for phrase search)**
- map()
  - parses each document
  - emits (word, **(**document ID**, list(positions))**) pairs
- reduce()
  - accepts all pairs **for a given word**
  - sorts the corresponding document IDs
  - emits (word, list(**(**document ID**, list(positions))**) pair
  - set of all output pairs forms a simple inverted index
  - again a form of aggregation

Distributed Sort

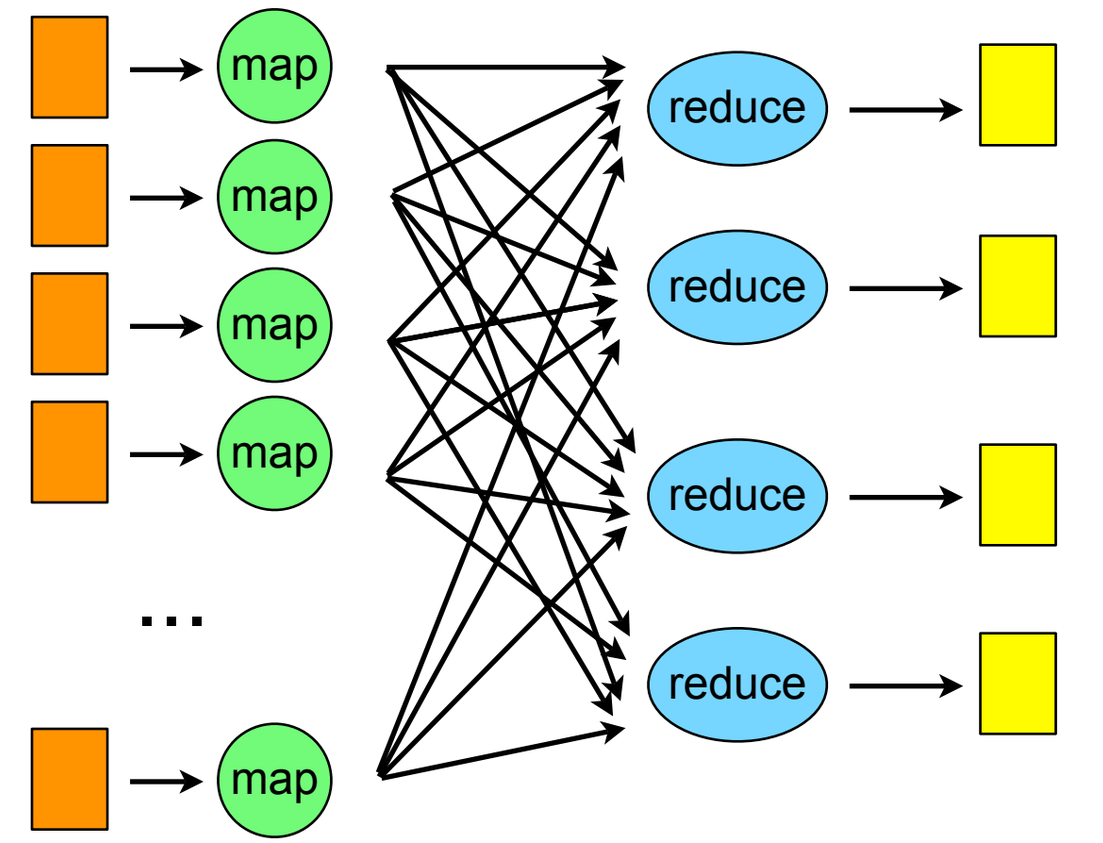
- map()
  - extracts the sort key from each record
  - emits (key, record) pairs
- reduce()
  - emits all pairs unchanged
- Discussion
  - requires extra property: output of a reducer is guaranteed to be sorted
  - therefore result will be avaliable in different output files (each of them sorted)

Implementation

- many different implementations possible: shared-nothing, GPU, multi-core, etc.
- implementation should be optimized for platform
- Google uses (in 2004)
  - dual CPU, x86, Linux, 2-4 GB of main memory
  - commodity networking hardware, 100 or 1000 megabits/second
  - single cluster consists of hundreds or thousands of machines
  - inexpensive IDE disks, attached directly to machines
  - distributed file system: Google file system (GFS)
  - GFS provides availability and reliability on top of unreliable hardware
  - user submits jobs to a scheduling system

Logical View on map/reduce

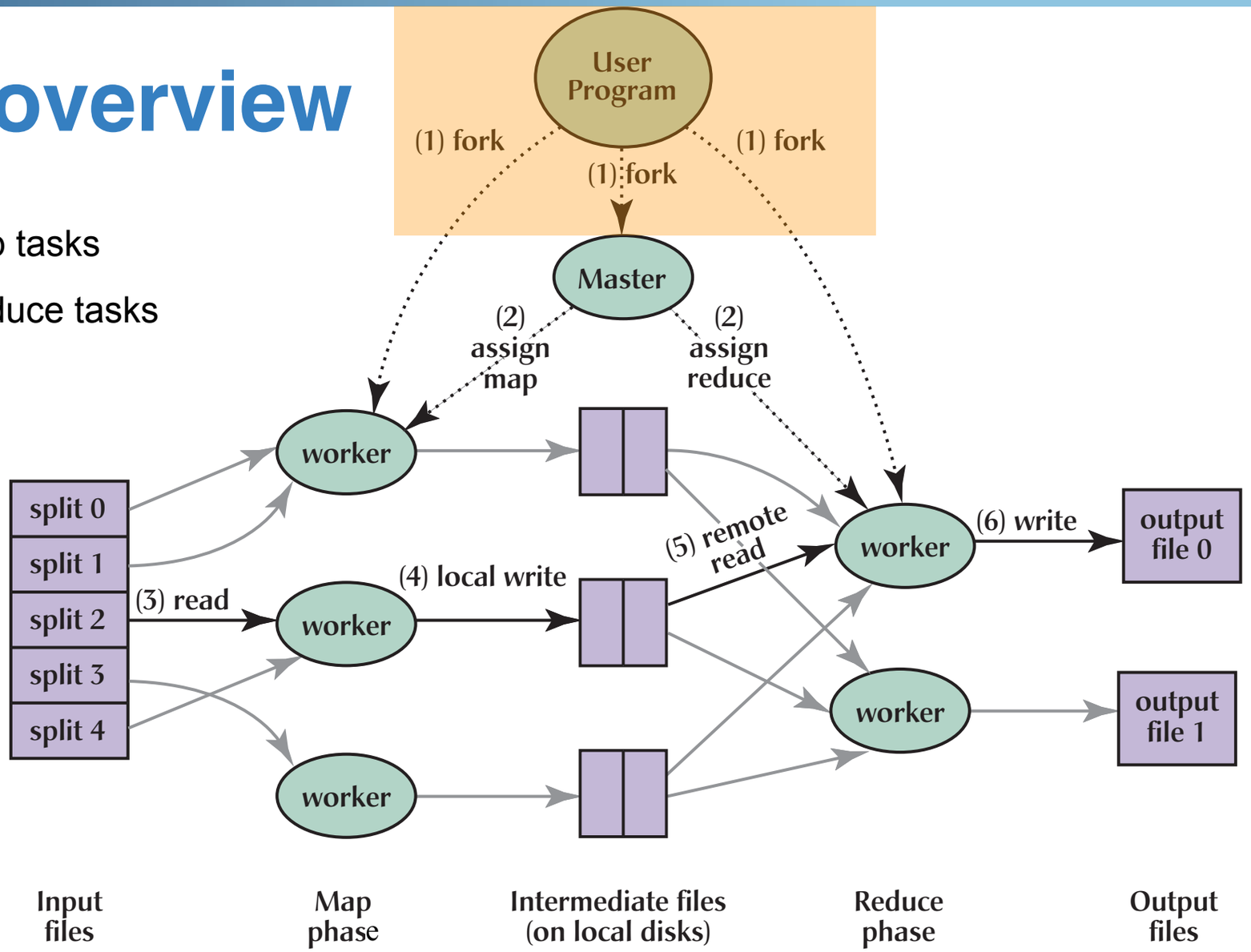
- input is partitioned into a set of **M** partitions, so-called **splits**
- typically 16 to 64 MB each, controlled by user
- input splits can be processed in parallel by different machines
- intermediate key space partitioned into **R** pieces using partitioning function (see below)



Execution overview

M=5 input files (splits) => 5 map tasks

R=2 output files (splits) => 2 reduce tasks

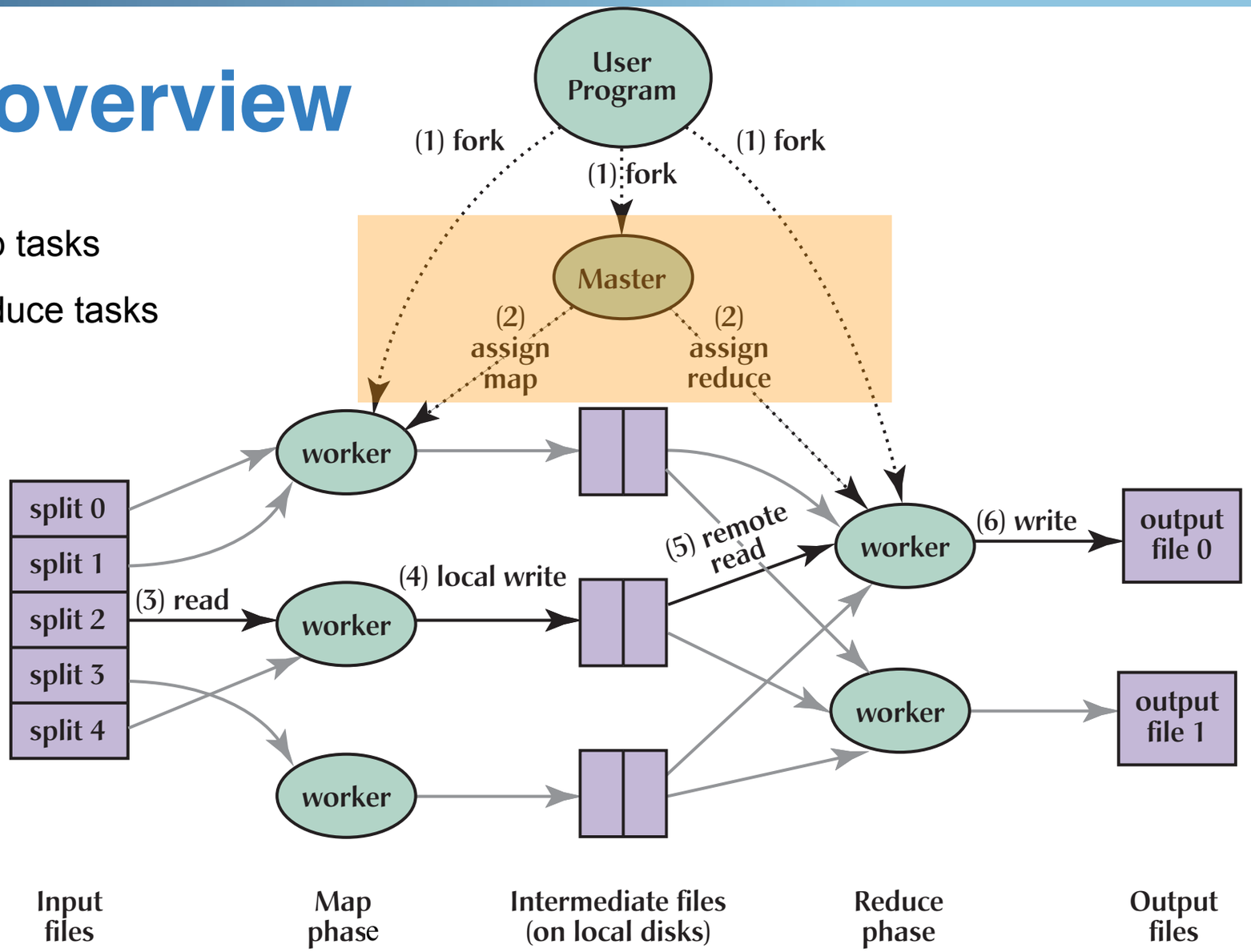


- (1)
  - MapReduce library splits input files into M pieces
  - starts many copies of the program on the cluster

Execution overview

M=5 input files (splits) => 5 map tasks

R=2 output files (splits) => 2 reduce tasks

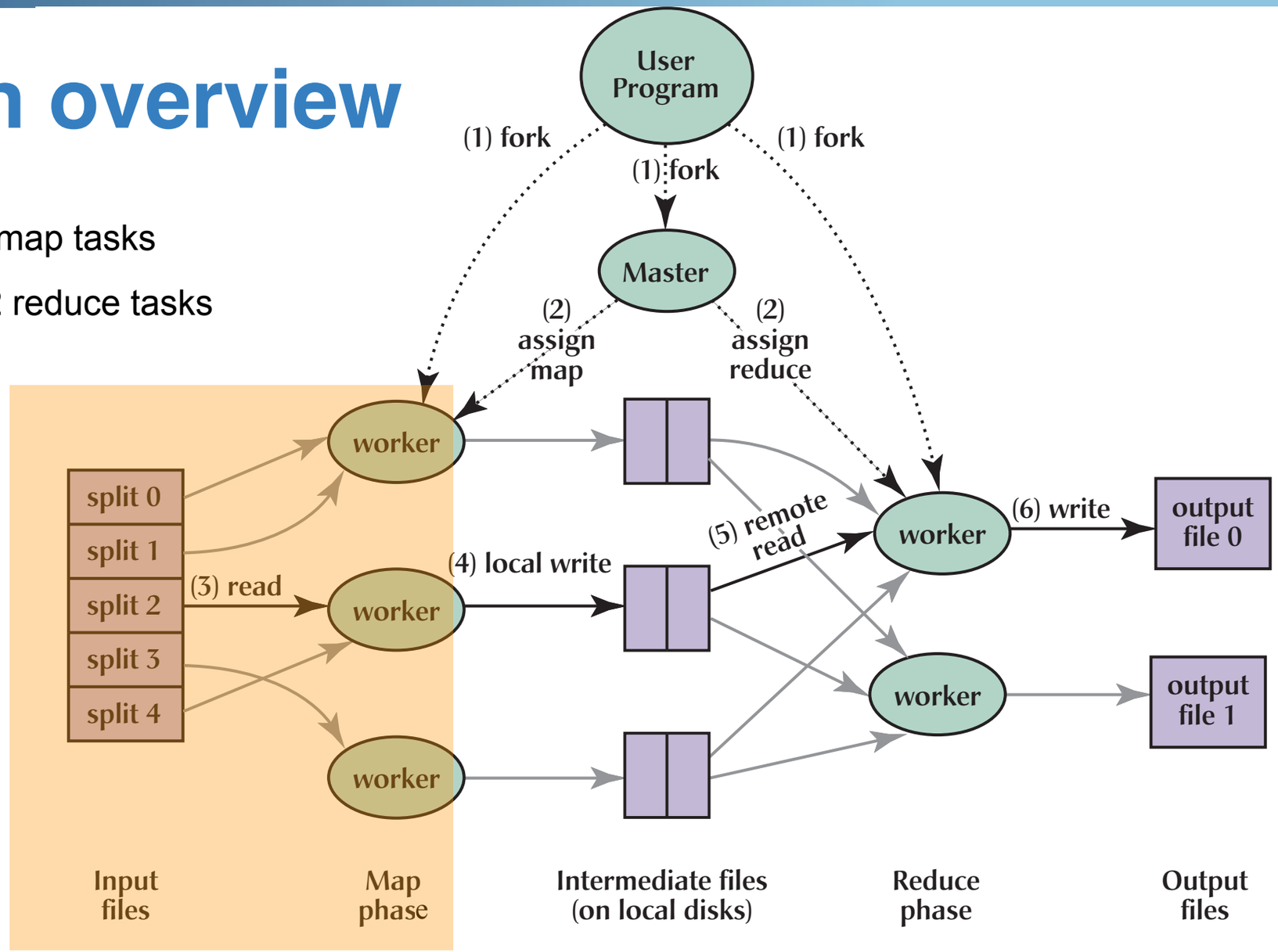


- (2)
  - master picks idle workers and assigns each one a map task or a reduce task

Execution overview

M=5 input files (splits) => 5 map tasks

R=2 output files (splits) => 2 reduce tasks

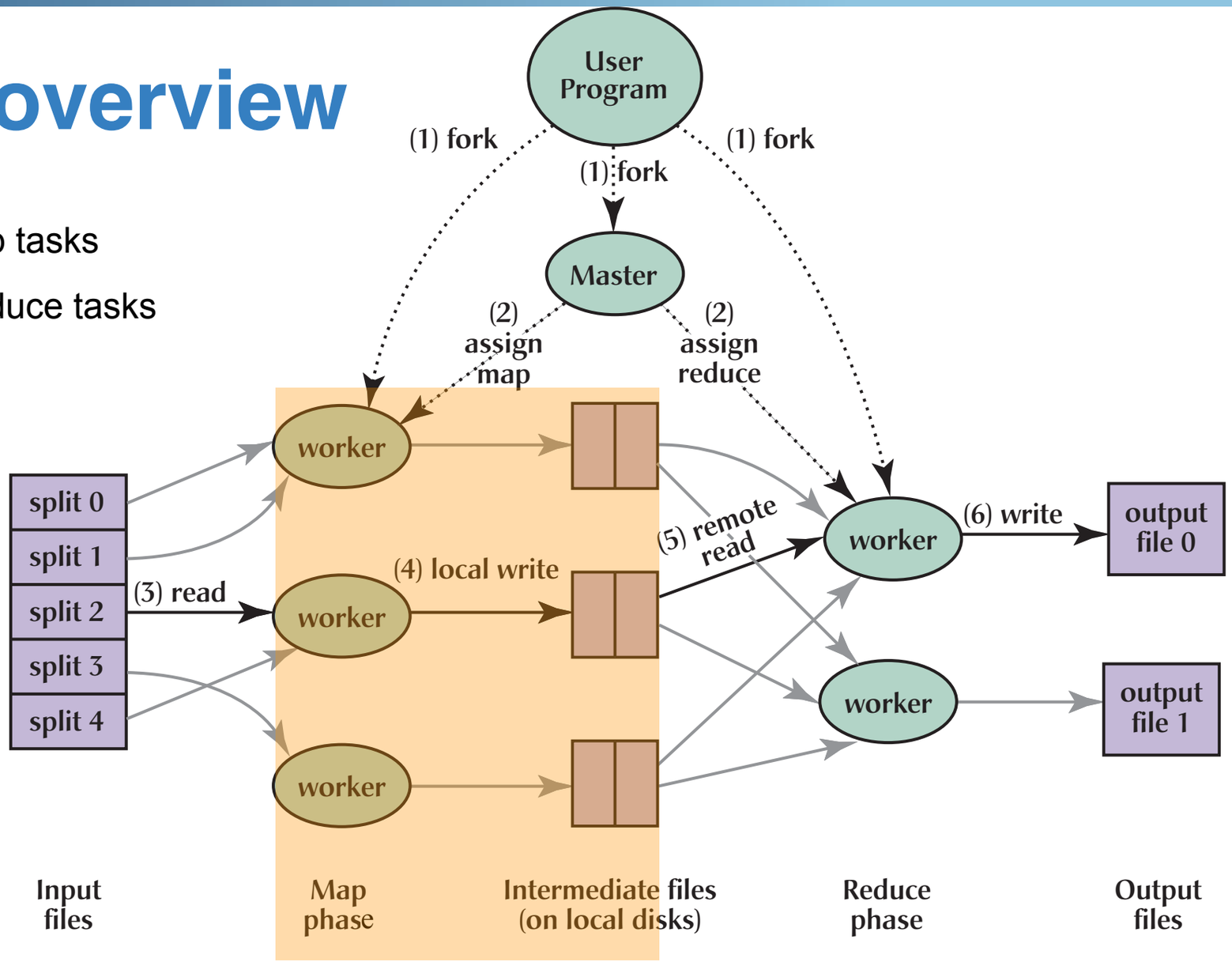


- (3)
  - a worker who is assigned a map task reads corresponding split
  - intermediate result pairs produced by map() are buffered in memory

Execution overview

M=5 input files (splits) => 5 map tasks

R=2 output files (splits) => 2 reduce tasks

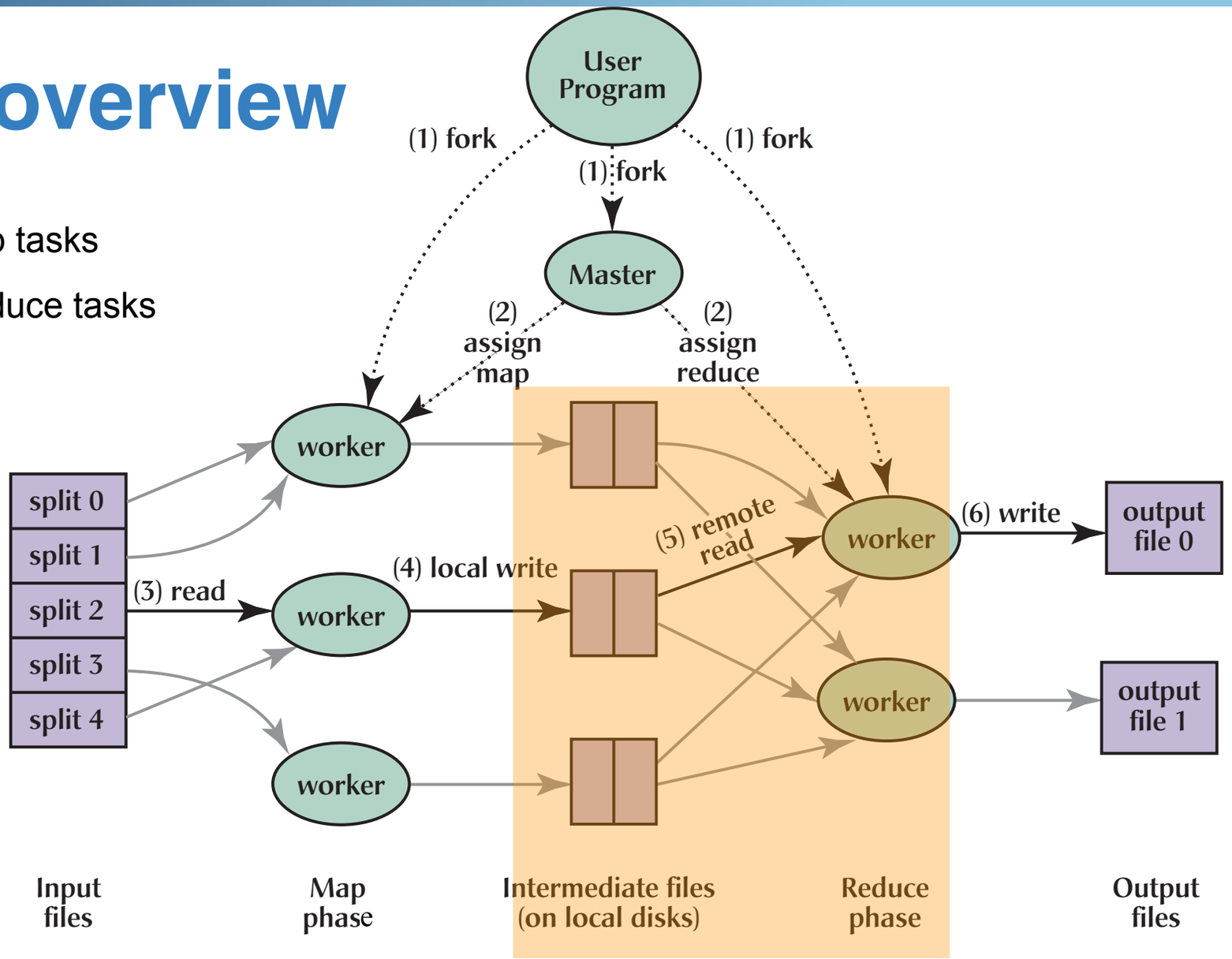


- (4)
  - periodically, buffered pairs are written to **local** disk
  - partitioned by **R** regions based on partitioning function (e.g., hash)
  - locations of these partitions published to master
  - master will later forward this data to the reducers

Execution overview

M=5 input files (splits) => 5 map tasks

R=2 output files (splits) => 2 reduce tasks

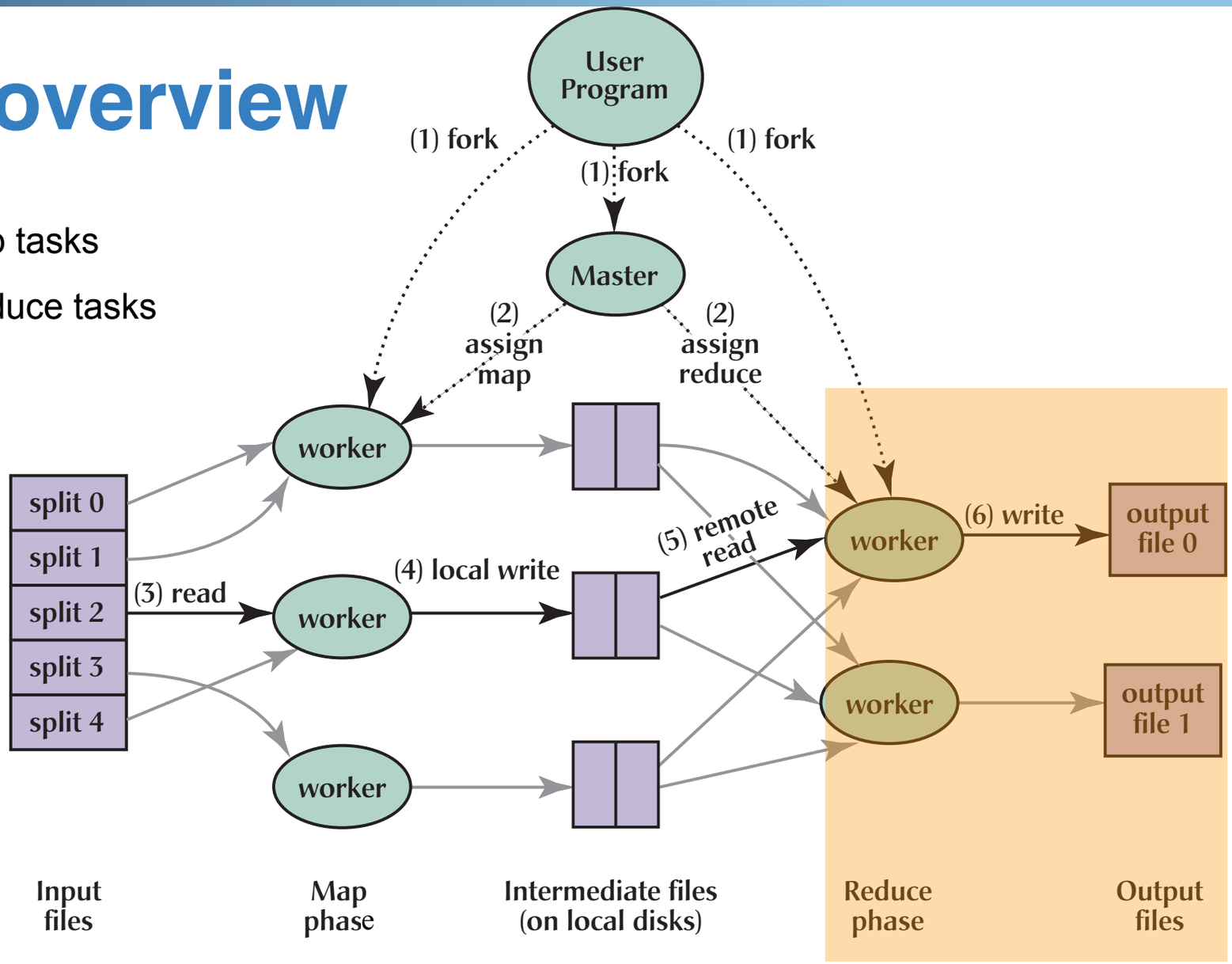


- (5)
  - reducer needs to read all input data (from possible many nodes/disks)
  - uses remote procedure calls
  - when reducer has read all input data
    => **sort** data by intermediate keys
  - if required, use external sorting

Execution overview

M=5 input files (splits) => 5 map tasks

R=2 output files (splits) => 2 reduce tasks

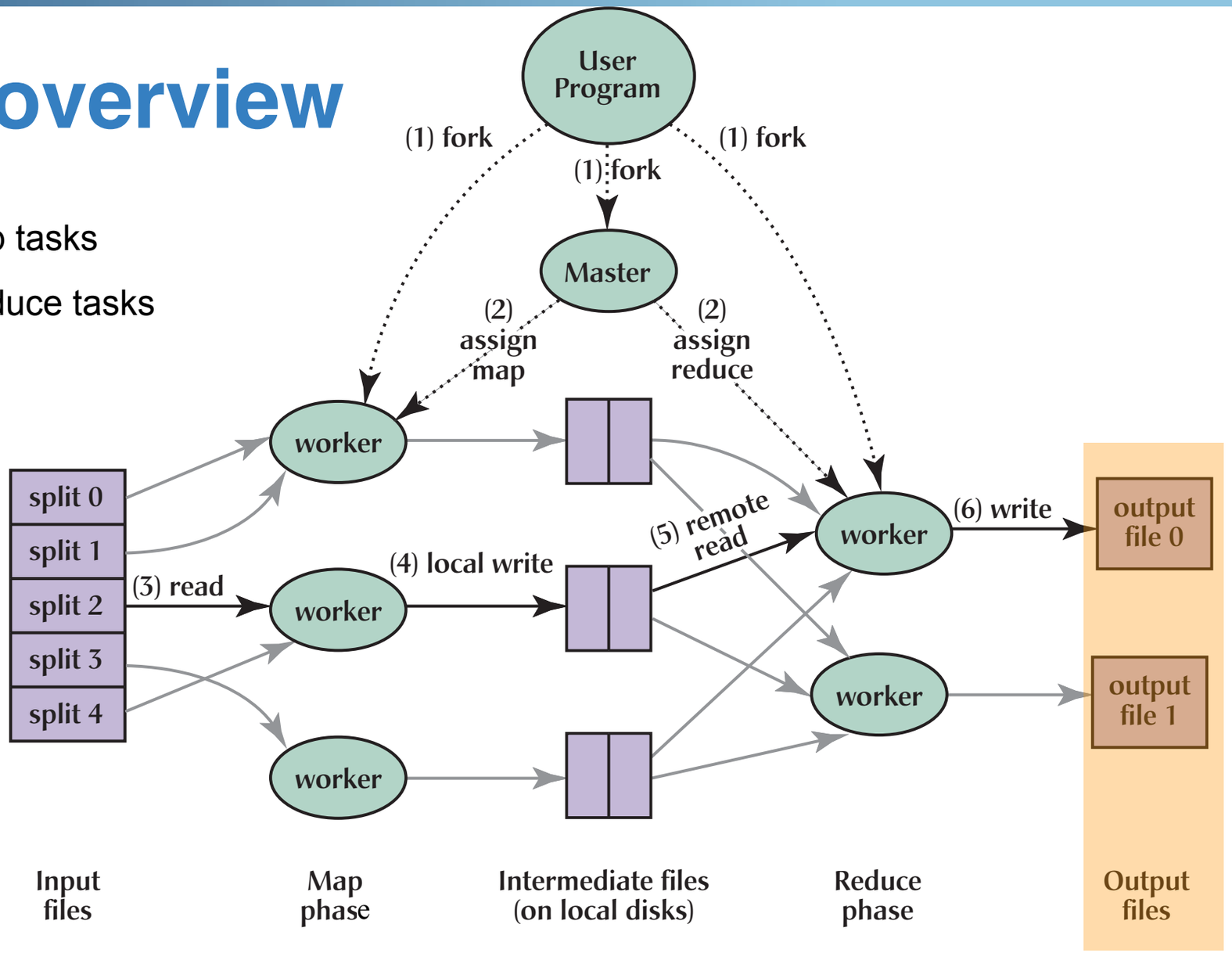


- (6)
  - reducer worker iterates over sorted intermediate data
  - for each unique key encountered:
    - pass key plus set of intermediate values found to a reducer()
  - output of the reduce function is appended to final output file for this reduce partition

Execution overview

M=5 input files (splits) => 5 map tasks

R=2 output files (splits) => 2 reduce tasks



- (7)
  - after completion: output available in R output files
  - typically these files are not combined into a single one but used as input for the next MapReduce call
  - or kept for an app that is able to handle partitioned data
  - example: inverted file -> query routed to machine having keyword IL

Fault Tolerance: Workers

- master pings workers periodically
- if no response within a timeout from a given worker
  => assume worker failed
- **any** task performed or in-progress by that worker wil be re-assigned to other workers
- note: even completed map tasks from that worker are re-scheduled
- reason: data produced by that worker may be inaccessible due to failure (data kept locally)
- completed **reduce tasks** are **not** re-scheduled
- reasons: output data kept on distributed files system

Fault Tolerance: Master

- only single master
- when master fails => all map/reduce work lost
- recovery
  - restart master
  - restart map/reduce jobs
- alternative: logging of master state (not implemented)
- discussion
  - better solution could be a hot standby

Backup Tasks

- overall execution time may be affected by single slow machines
- example:
  - machine with a bad disk
  - frequent correctable errors, but no failure
  - read performance may drop by an order of magnitude (or more)
- solution:
  - when the overall MapReduce task is close to completion
  - => master schedules backup executions of remaining in-progress tasks
  - tasks will be considered completed whenever either the primary of the backup execution completes
- significantly reduces the time to complete large tasks

Partitioning Function

- user specifies number of reduce tasks/output files R they desire
- data gets partitioned across these tasks by the framework
- based on a partitioning function
- default is a hash function hash(key) mod R
- may be adapted
- example
  - collect all entries for a single host in an output file
  - -> partition by host
  - hash(Hostname(URL)) mod R
- may also be replaced by an interval partitioning
- see discussion for Grace Hash Join and Horizontal Partitioning

Combiner Functions

- problem: some map()-tasks create considerable repetition
- example:
  - word count
  - each map will produce hundreds or thousands of records of the form (the, 1)
  - all of this is sent over the network
- this may be alleviated by allowing users to specify a combiner function
- combiner is executed in the machine performing the map task
- typically **same** code used for reduce() is used for combiner()

Combiner Functions

- only diff: output handling
  - output of reduce() -> file
  - output of combine() -> intermediate file -> reduce()
- Discussion
  - this concept is called **early aggregation** in databases
  - see Week 7, Slide 97
  - **drawbacks** in the Google approach:
    - differentiate between reduce() and combine()
    - not really necessary
  - reduce() and combine() should be considered operators!
  - actually it is the same operator: only one reduce()
  - operators should be placed by query optimizer
  - could even be inserted at runtime
- **= early reduce()**

Network Locality

- network may become a bottleneck
- system tries to cope with that by generating replicas for the input data
- master tries to schedule map tasks in a way that data is read from local disk
- if that does not work, master tries to schedule the map task near the data
- example: worker machine residing on the same network switch as the machine containing the data
- authors claim that most jobs can be assigned locally w.r.t. map
- Discussion: for reduce this is not possible in the general case

Setting M and R

- R often constrained
- recall that output of each reduce task generates a separate file
- in practice:
  - choose M such that each individual task is roughly 16 MB to 64 MB of input data
  - make R a small multiple of the number of worker machines
- example
  - M= 200,000
  - R= 6,000
  - using 2,000 worker machines

Experience at Google

- experience reported from the authors
  - “code simpler, smaller, easier to understand“
  - “distribution and parallelization hidden within MapReduce library“
  - “example: one part of the code “reduced“ from 3,800 to 700 LOC“
  - “map/reduce helps to keep conceptually unrelated computations separate, instead of mixing them together“
  - “makes it easier to change indexing process“
  - “example: one change in old system took month“
  - “took only a few days in MapReduce“

- open source implementation of map/reduce
- an apache project
- http://hadoop.apache.org/core/
- includes a distributed file system: HDFS
- all Java
- used by Amazon, AOL, Facebook, Last.fm, The New York Times, Yahoo!, etc.

Largest Hadoop Cluster at Yahoo!

- they believe this is the biggest hadoop cluster so far
- (as of Sept 2008)
- 4000 nodes, each:
  - 2 quad core Xeons @ 2.5ghz
  - 4x1TB SATA disks
  - 8G RAM
  - 1 gigabit ethernet
- 40 nodes per rack
- 4 gigabit ethernet uplinks from each rack to the core
- Red Hat Enterprise Linux AS release 4 (Nahant Update 5)
- Sun Java JDK 1.6.0_05-b13
- => **32,000 cores, 32TB main memory, 16PB disk space**!
- primarily an exercise to see how Hadoop works at this scale

Terabyte Benchmark

- task: sort one terabyte of data
- must read and write data from/to disk
- current winner: Hadoop, done by Yahoo!
- used 910 of the nodes
- total time: **209** seconds
- a 100 TB benchmark will be specified this year
- http://www.hpl.hp.com/hosted/sortbenchmark/

Multi-core Map/reduce: Phoenix

- map/reduce paradigm was also tried on multi-cores
- http://csl.stanford.edu/~christos/sw/phoenix/
- promising experimental results
- Phoenix equal to P-threads if algorithm matches MapReduce model
- P-threads is better for algorithms that do not fit MapReduce model
- Literature:
  "Evaluating MapReduce for Multi-core and Multiprocessor Systems", Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, Christos Kozyrakis. Symposium on High-Performance Computer Architecture (HPCA), Phoenix, AZ, 2007.

GPU Map/reduce: Mars

- tried NVIDIA G80 GPU, >100 cores
- experiments promising for GPUs
- also executes on multi-cores:
  claim to be better than Phoenix
- requires more research
- Literature:
  Mars: A MapReduce Framework on Graphics Processors, Bingsheng He, Wenbin Fang, Qiong Luo, Naga Govindaraju and Tuyong Wang. Parallel Architectures and Compilation Techniques (PACT), 2008.

map-reduce merge

- map/reduce may be extended to support binary operations
- examples: joins, set difference
- how does that work?
- core idea:
  - map and reduce two sets simultaneously
  - after reduce append a third phase: merge
- note: this is the core idea of almost all binary operators
- just “mapped“ to merge/reduce

Conclusions

- map/reduce extracts two ideas from functional programming
- allows users to easily run distributed data processing algorithms on a large cluster of machines
- users do not need to fiddle around with parallelization code, load balancing, failover, etc
- map/reduce may be extended to support binary operations such as joins, intersect, etc.
- open source implementations of map/reduce available: Hadoop

Literature

- Jeffrey Dean, Sanjay Ghemawat: MapReduce: Simplified Data Processing on Large Clusters. OSDI 2004:137-150
- Jeffrey Dean, Sanjay Ghemawat: MapReduce: simplified data processing on large clusters. Commun. ACM (CACM) 51(1):107-113 (2008)
- Hung-chih Yang, Ali Dasdan, Ruey-Lung Hsiao, Douglas Stott Parker Jr.: Map-reduce-merge: simplified relational data processing on large clusters. SIGMOD 2007:1029-1040
- Hadoop: http://hadoop.apache.org/core/

Next Topic: Pig.