Modern Database Systems

MapReduce

Doc. RNDr. Irena Holubova, Ph.D.

Irena.Holubova@matfyz.cuni.cz

MapReduce Framework

- A programming model + implementation
- Developed by Google in 2008
  - To replace old, centralized index structure
- Distributed, parallel computing on large data

> Google: “A simple and powerful interface that enables automatic parallelization and distribution of large-scale computations, combined with an implementation of this interface that achieves high performance on large clusters of commodity PCs.”

- Programming model in general:
  - Mental model a programmer has about execution of application
  - Purpose: improve programmer's productivity
  - Evaluation: expressiveness, simplicity, performance

Programming Models

- Von Neumann model
  - Executes a stream of instructions (machine code)
  - Instructions can specify
    - Arithmetic operations
    - Data addresses
    - Next instruction to execute
    - …
  - Complexity
    - Billions of data locations and millions of instructions
    - Manages with:
      - Modular design
      - High-level programming languages

Programming Models

- Parallel programming models
  - Message passing
    - Independent tasks encapsulating local data
    - Tasks interact by exchanging messages
  - Shared memory
    - Tasks share a common address space
    - Tasks interact by reading and writing from/to this space
      - Asynchronously
  - Data parallelization
    - Data are partitioned across tasks
    - Tasks execute a sequence of independent operations

MapReduce Framework

- Divide-and-conquer paradigm
  - Map breaks down a problem into sub-problems
    - Processes a key/value pair to generate a set of intermediate key/value pairs
  - Reduce receives and combines the sub-solutions to solve the problem
    - Processes intermediate values associated with <u>the same</u> intermediate key
- Many real-world tasks can be expressed this way
  - Programmer focuses on map/reduce code
  - Framework cares about data partitioning, scheduling execution across machines, handling machine failures, managing inter-machine communication, …

MapReduce

A Bit More Formally

- Map
  - Input: a key/value pair
  - Output: a set of intermediate key/value pairs
    - Usually different domain
  - $(k_1, v_1) \rightarrow list(k_2, v_2)$
- Reduce
  - Input: an intermediate key and a set of all values <u>for that key</u>
  - Output: a possibly smaller set of values
    - The same domain
  - $(k_2, list(v_2)) \rightarrow (k_2, \text{possibly smaller } list(v_2))$

MapReduce

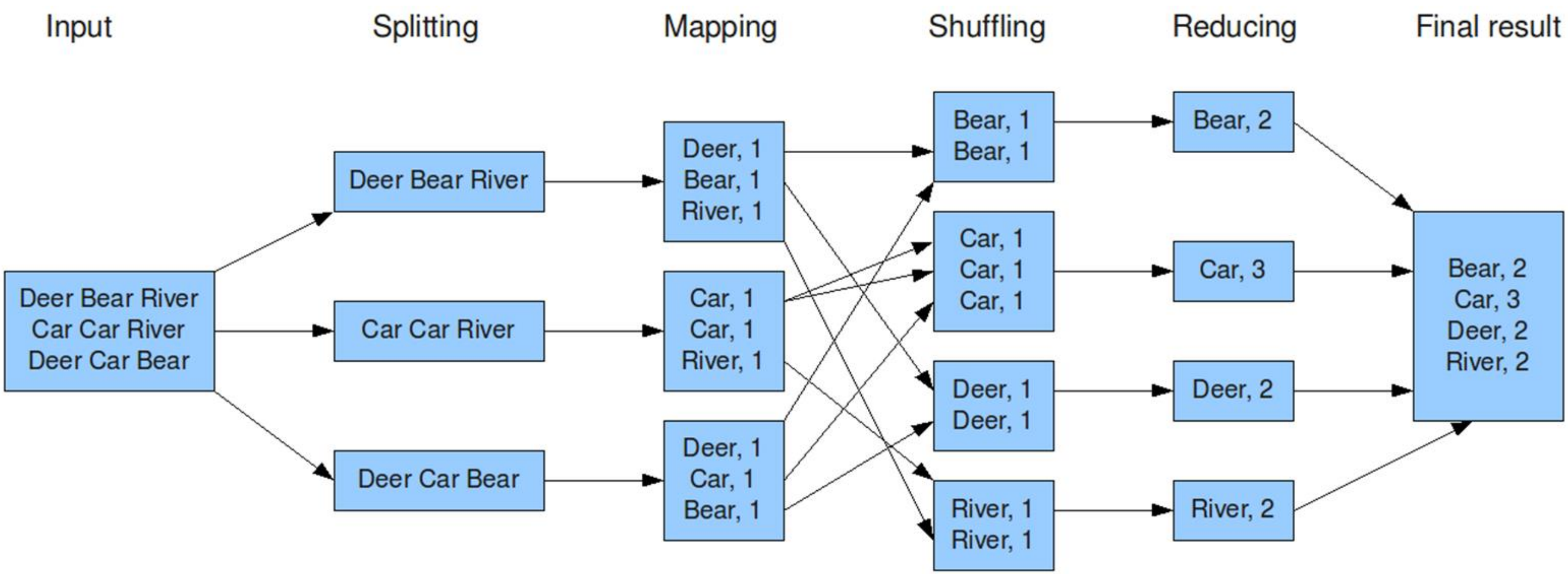
Example: Word Frequency

```
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(key, AsString(result));
```

MapReduce

Example: Word Frequency



MapReduce

More Examples

- distributed grep
  - Map: emits <word, line number> if it matches a supplied pattern
  - Reduce: identity
- URL access frequency
  - Map: processes web logs, emits <URL, 1>
  - Reduce: sums values and emits <URL, sum>
- reverse web-link graph
  - Map: <target, source> for each link to a target URL found in a page named source
  - Reduce: concatenates the list of all source URLs associated with a given target URL <target, list(source)>

MapReduce
More Examples

- term vector per host
  - “Term vector” summarizes the most important words that occur in a document or a set of documents
  - Map: emits <hostname, term vector> for each input document
    - The hostname is extracted from the URL of the document
  - Reduce: adds the term vectors together, throws away infrequent terms
- inverted index
  - Map: parses each document, emits <word, document ID>
  - Reduce: sorts the corresponding document IDs, emits <word, list(document ID)>
- distributed sort
  - Map: extracts the key from each record, and emits <key, record>
  - Reduce: emits all pairs unchanged

MapReduce
Application Parts

- Input reader
  - Divides the input into appropriate size 'splits'
    - Each assigned to a single Map function
  - Reads data from stable storage
    - e.g., a distributed file system
  - Generates key/value pairs
- Map function
  - User-specified processing of key/value pairs
- Partition function
  - Map function output is allocated to a reducer
  - Partition function is given the key (output of Map) and the number of reducers and returns the index of the desired reducer
    - Default is to hash the key and use the hash value modulo the number of reducers

MapReduce
Application Parts

- Compare function
  - Sorts the input for the Reduce function
- Reduce function
  - User-specified processing of key/values
- Output writer
  - Writes the output of the Reduce function to stable storage
    - e.g., a distributed file system

MapReduce

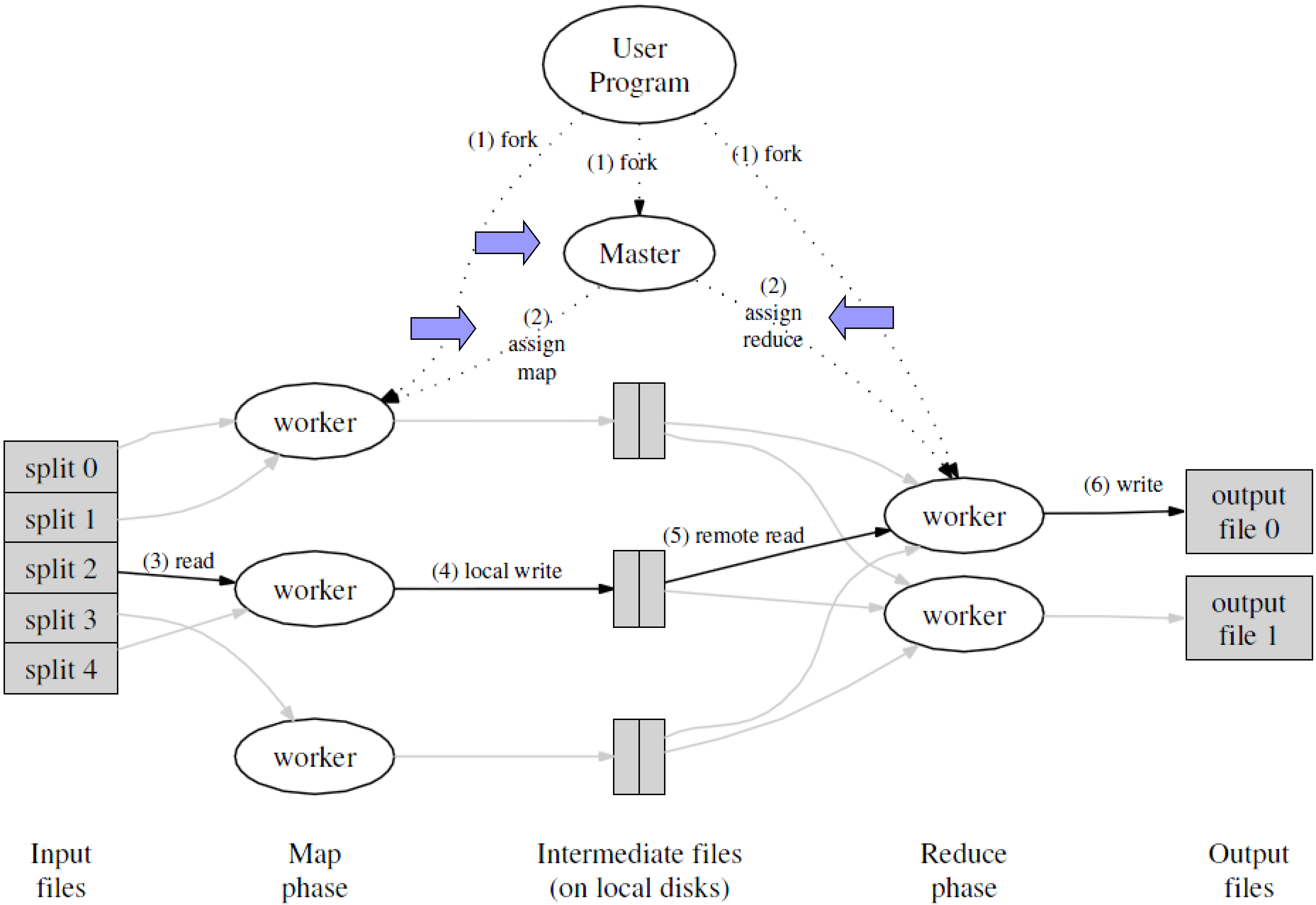
Execution (Google) – Step 1

1. MapReduce library in the user program splits the input files into *M* pieces
   - Typically 16 – 64 MB per piece
   - Controllable by the user via optional parameter
2. It starts copies of the program on a cluster of machines

MapReduce

Execution – Step 2

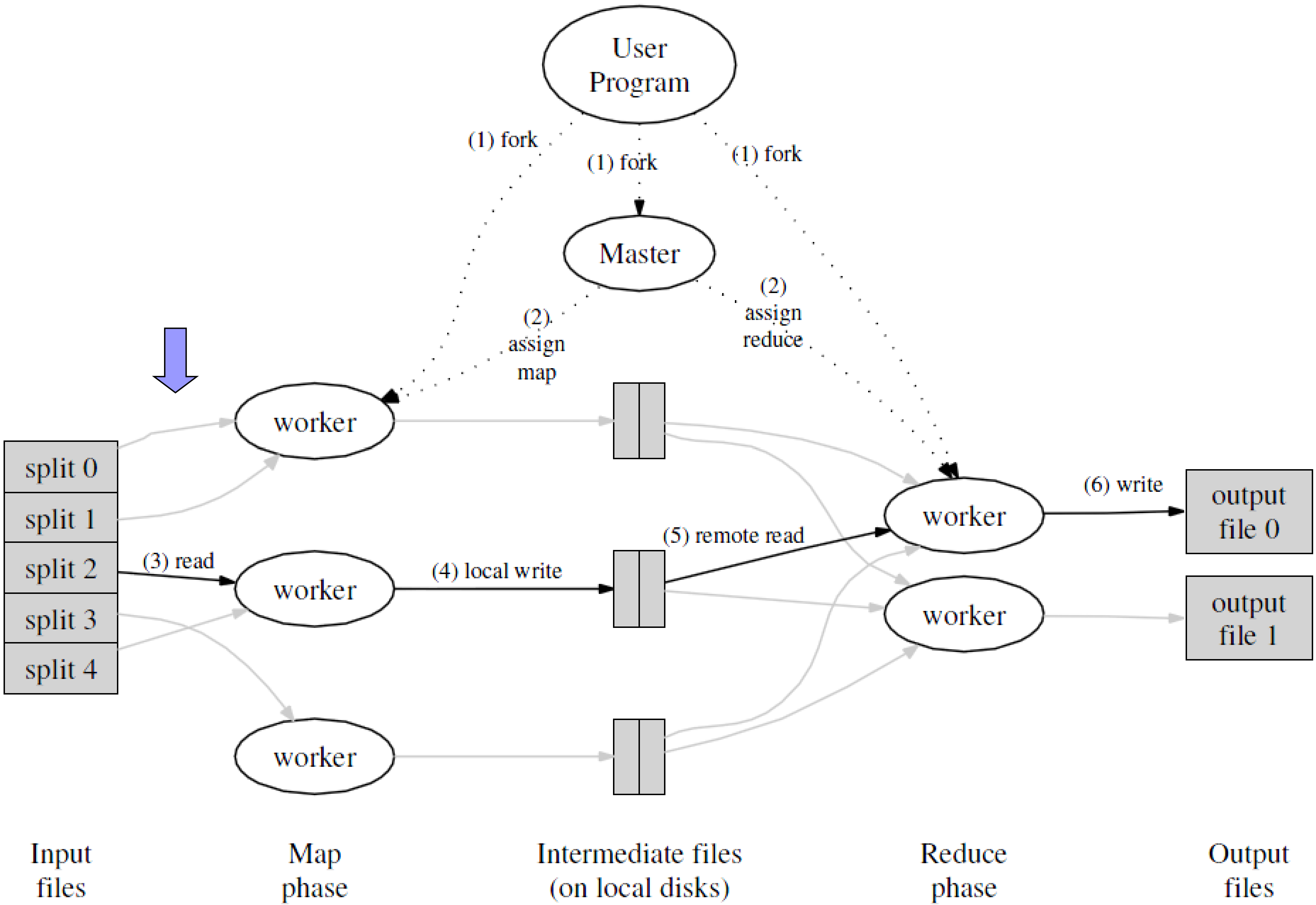
- Master = a special copy of the program
- Workers = other copies that are assigned work by master
- *M* Map tasks and *R* Reduce tasks to assign
- Master picks idle workers and assigns each one a Map task (or a Reduce task)


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

MapReduce

Execution – Step 3

- A worker who is assigned a Map task:
  - Reads the contents of the corresponding input split
  - Parses key/value pairs out of the input data
  - Passes each pair to the user-defined Map function
  - Intermediate key/value pairs produced by the Map function are buffered in memory

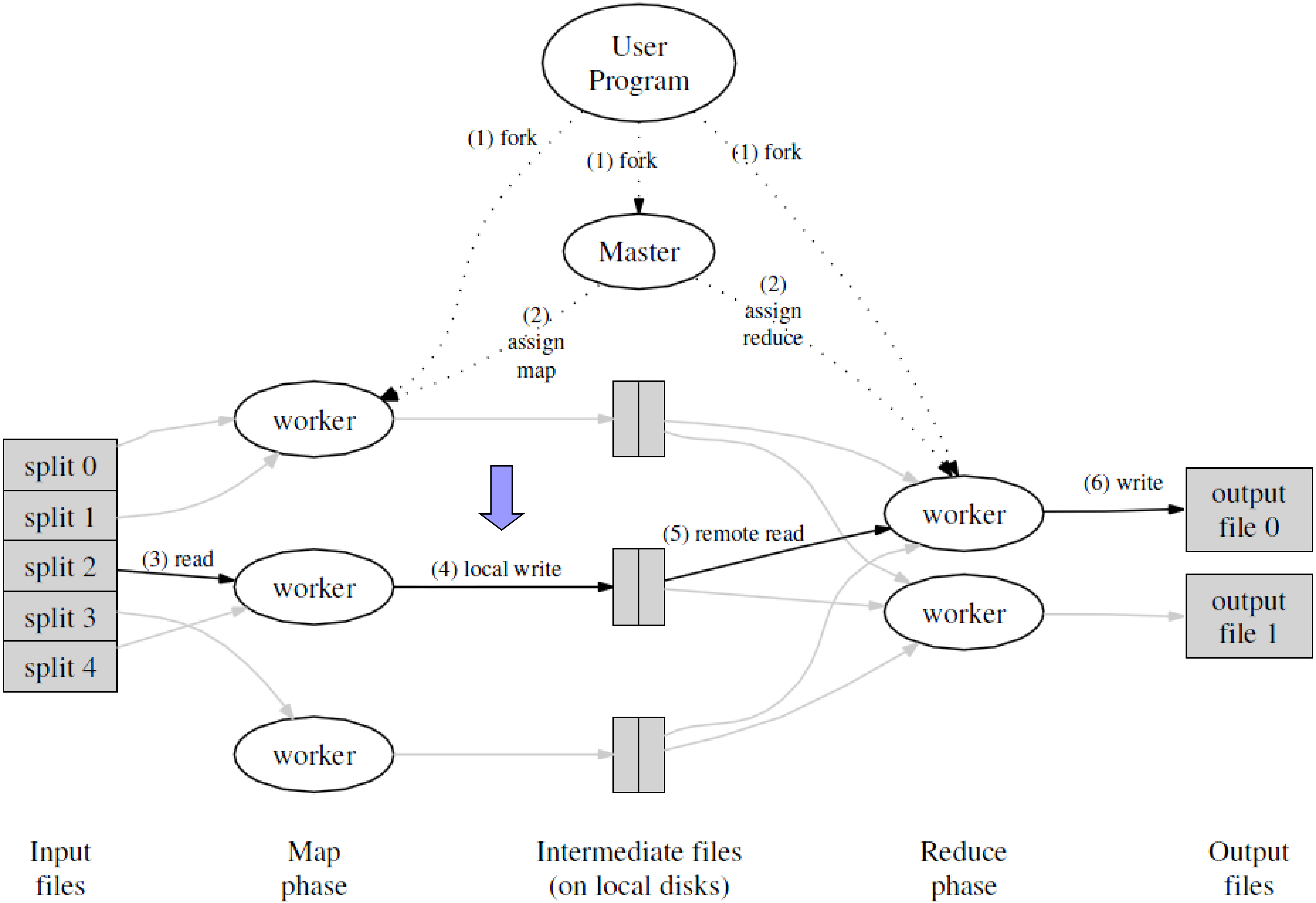


User Program
(1) fork
(1) fork
(1) fork
Master
(2) assign map
(2) assign reduce
worker
worker
worker
worker
worker
split 0
split 1
split 2
split 3
split 4
(3) read
(4) local write
(5) remote read
(6) write
output file 0
output file 1
Input files
Map phase
Intermediate files (on local disks)
Reduce phase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

MapReduce

Execution – Step 4

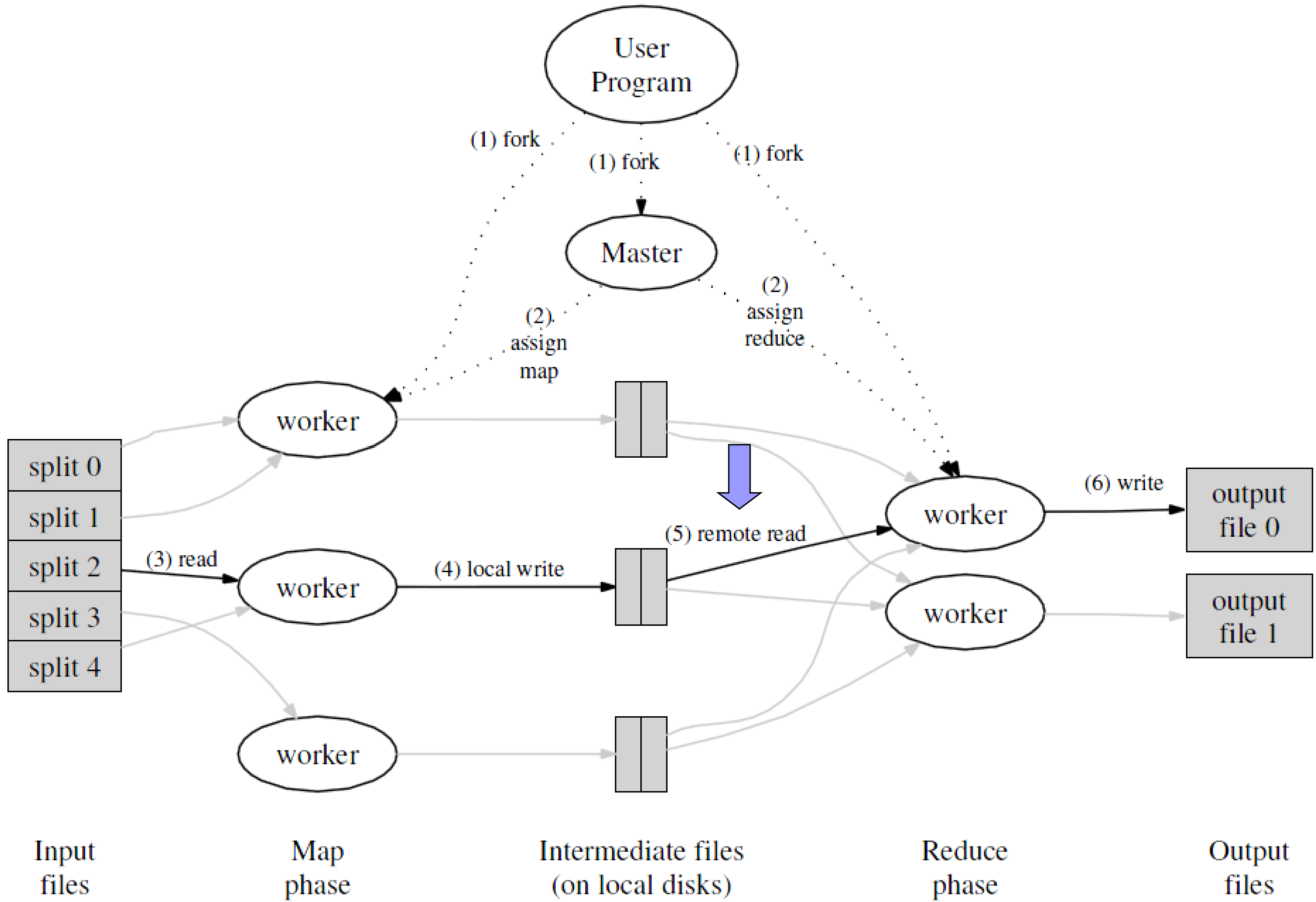
- Periodically, the buffered pairs are <u>written to local disk</u>
  - Partitioned into *R* regions by the partitioning function
- Locations of the buffered pairs on the local disk are passed back to the master
  - It is responsible for forwarding the locations to the Reduce workers


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

MapReduce

Execution – Step 5

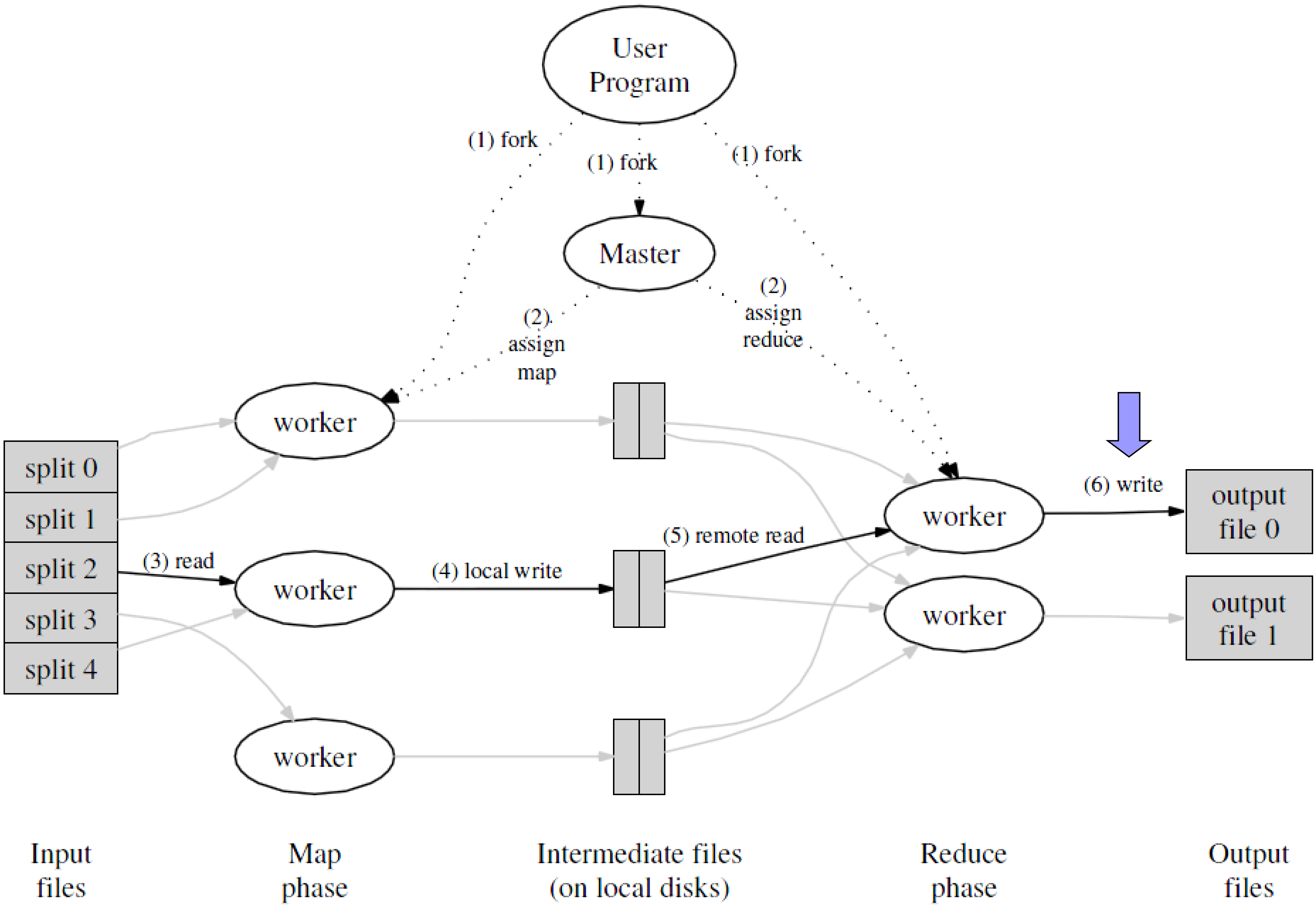
- Reduce worker is notified by the master about data locations
- It uses remote procedure calls to read the buffered data from local disks of the Map workers
- When it has read all intermediate data, it sorts it by the intermediate keys
  - Typically many different keys map to the same Reduce task
  - If the amount of intermediate data is too large, an external sort is used


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

MapReduce

Execution – Step 6

- A Reduce worker iterates over the sorted intermediate data
- For each intermediate key encountered:
  - It passes the key and the corresponding set of intermediate values to the user's Reduce function
  - The output is appended to a final output file for this Reduce partition



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

MapReduce

Function combine

- After a map phase, the mapper transmits over the network the entire intermediate data file to the reducer
- Sometimes this file is highly compressible
- User can specify function combine
  - Like a reduce function
  - It is run by the mapper before passing the job to the reducer
    - Over local data

MapReduce

Counters

- Can be associated with any action that a mapper or a reducer does
  - In addition to default counters
    - e.g., the number of input and output key/value pairs processed
- User can watch the counters in real time to see the progress of a job

MapReduce

Fault Tolerance

- A large number of machines process a large number of data → fault tolerance is necessary
- Worker failure
  - Master pings every worker periodically
  - If no response is received in a certain amount of time, master marks the worker as failed
  - <u>All</u> its tasks are reset back to their initial <u>idle</u> state → become eligible for scheduling on other workers

MapReduce
Fault Tolerance

- Master failure
  - Strategy A:
    - Master writes periodic checkpoints of the master data structures
    - If it dies, a new copy can be started from the last checkpointed state
  - Strategy B:
    - There is only a single master → its failure is unlikely
    - MapReduce computation is simply aborted if the master fails
    - Clients can check for this condition and retry the MapReduce operation if they desire

MapReduce

Stragglers

- Straggler = a machine that takes an unusually long time to complete one of the map/reduce tasks in the computation
  - Example: a machine with a bad disk
- Solution:
  - When a MapReduce operation is close to completion, the master schedules backup executions of the remaining in-progress tasks
  - A task is marked as completed whenever either the primary or the backup execution completes

MapReduce
Task Granularity

- *M* pieces of Map phase and *R* pieces of Reduce phase
  - Ideally both much larger than the number of worker machines
  - How to set them?
- Master makes $O(M + R)$ scheduling decisions
- Master keeps $O(M * R)$ status information in memory
  - For each Map/Reduce task: state (idle/in-progress/completed)
  - For each non-idle task: identity of worker machine
  - For each completed Map task: locations and sizes of the *R* intermediate file regions
- *R* is often constrained by users
  - The output of each Reduce task ends up in a separate output file
- Practical recommendation (Google):
  - Choose *M* so that each individual task is roughly 16 – 64 MB of input data
  - Make *R* a small multiple of the number of worker machines we expect to use

Real-World Example (Google)

Cluster Configuration

- 1,800 machines
- Each machine:
  - 2x 2GHz Intel Xeon processor
    - With Hyper-Threading enabled
  - 4GB memory
    - Approx. 1-1.5GB reserved by other tasks
  - 2x 160GB IDE disks
  - Gigabit Ethernet link
- Arranged in a two-level tree-shaped switched network with approximately 100-200 Gbps of aggregate bandwidth available at the root

Real-World Example 1

grep

- Search through approx. 1 terabyte of data looking for a particular pattern
  - Rare three-character pattern
  - Present in 92,337 records
- M = 15,000
- R = 1
- 1,764 workers assigned
- Entire computation?
  - 150 seconds
  - About a minute of start-up overhead

Real World Example 2

sort

- Sorting of approx. 1 terabyte of data
- Map: 3-line function
  - Extracts a 10-byte sorting key from a text line and emits the key and the original text line
- Reduce: identity
- M = 15,000
- R = 4,000
- About 1,700 workers assigned
- Entire computation?
  - 891 seconds
  - 5 stragglers increase the time of 44%

MapReduce Criticism

David DeWitt and Michael Stonebraker – 2008

1. MapReduce is a step backwards in database access based on
   - Schema describing data structure
   - Separating schema from the application
   - Advanced query languages
2. MapReduce is a poor implementation
   - Instead of indices it uses brute force
3. MapReduce is not novel (ideas more than 20 years old and overcome)
4. MapReduce is missing features common in DBMSs
   - Indices, transactions, integrity constraints, views, …
5. MapReduce is incompatible with applications implemented over DBMSs
   - Data mining, business intelligence, …

Note: Who is Michael Stonebraker?

- *1943
- Computer scientist – database researcher
- Academic prototypes form the core of various databases
  - Ingres, Postgres, C-store (Vertica), H-store (VoltDB), SciDB, …
- 2015 – Turing award (ACM)
  - “Nobel Prize of computing”
  - For concepts and practices underlying modern database systems
  - 2016 – Tim Berners Lee
    - For inventing the WWW

End of MapReduce?

- FaceBook used MapReduce in 2010
  - Hadoop

but…

- Google has shifted towards: Google Cloud DataFlow
  - Based on cloud and stream data processing
  - Idea: no need to maintain complex infrastructure
    - Data can be easily read, transformed and analyzed in a cloud

https://cloud.google.com/dataflow/

Hadoop MapReduce

- MapReduce requires:
  - Distributed file system
  - Engine that can distribute, coordinate, monitor and gather the results
- Hadoop: HDFS + JobTracker + TaskTracker
  - JobTracker (master) = scheduler
  - TaskTracker (slave per node) – is assigned a Map or Reduce (or other operations)
    - Map or Reduce run on a node → so does the TaskTracker
    - Each task is run on its own JVM

MapReduce

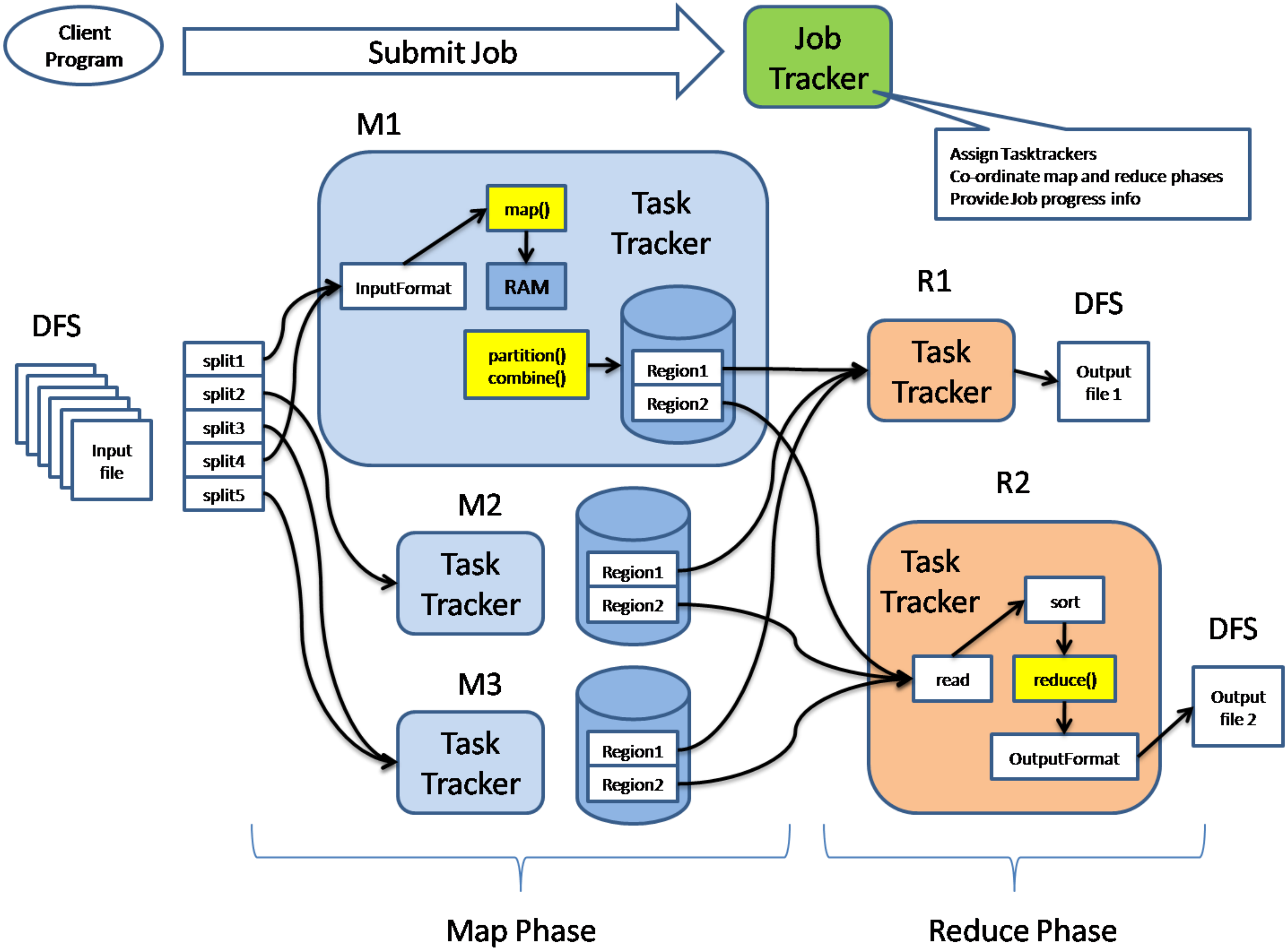
JobTracker (Master)

- Like a scheduler:
  1. A client application is sent to the JobTracker
  2. It “talks” to the NameNode (= HDFS master) and locates the TaskTracker (Hadoop client) <u>near</u> the data
  3. It moves the work to the chosen TaskTracker node

MapReduce

TaskTracker (Client)

- Accepts tasks from JobTracker
  - Map, Reduce, Combine, …
  - Input, output paths
- Has a number of slots for the tasks
  - Execution slots available on the machine (or machines on the same rack)
- Spawns a separate JVM for execution of a task
- Indicates the number of available slots through the hearbeat message to the JobTracker
  - A failed task is re-executed by the JobTracker


Client Program
Submit Job
Job Tracker
Assign Tasktrackers
Co-ordinate map and reduce phases
Provide Job progress info
M1
Task Tracker
map()
InputFormat
RAM
partition()
combine()
Region1
Region2
DFS
Input file
split1
split2
split3
split4
split5
M2
Task Tracker
Region1
Region2
M3
Task Tracker
Region1
Region2
R1
Task Tracker
DFS
Output file 1
R2
Task Tracker
sort
read
reduce()
OutputFormat
DFS
Output file 2
Map Phase
Reduce Phase

Job Launching

Job configuration

- For launching program:
  1. Create a Job to define a job
     - Using class Configuration
  2. Submit Job to the cluster and wait for completion
- Job involves:
  - Classes implementing Mapper and Reducer interfaces
    - `Job.setMapperClass()`
    - `Job.setReducerClass()`
  - Job outputs
    - `Job.setOutputKeyClass()`
    - `Job.setOutputValueClass()`
  - Other options:
    - `Job.setNumReduceTasks()`
    - …

Job Launching

Job

- `waitForCompletion()` – waits (blocks) until the job finishes
- `submit()` – does not block
- `monitorAndPrintJob()` – monitor a job and print status in real-time as progress is made and tasks fail

Mapper

- The <u>user</u> provides an instance of Mapper
  - Implements interface `Mapper`
    - Overrides function `map`
  - Emits (k_2,v_2) using `context.write(k2, v2)`
- Exists in separate process from all other instances of Mapper
  - No data sharing

input key

```
void map (Object key,
          Text value,
          Context context)
```

input value

collects output keys and values

```
public static class TokenizerMapper
      extends Mapper<Object, Text, Text, IntWritable>{

   private final static IntWritable one = new IntWritable(1);
   private Text word = new Text();

   public void map (Object key, Text value, Context context)
        throws IOException, InterruptedException {
     StringTokenizer itr
        = new StringTokenizer(value.toString());
     while (itr.hasMoreTokens()) {
       word.set(itr.nextToken());
       context.write(word, one);
     }
   }
 }
```

Reducer

```
reduce(Text key,
       Iterable<IntWritable> values,
       Context context)
```

- Keys & values sent to one partition all go to the same reduce task
- Calls are sorted by key

```java
public static class IntSumReducer
      extends Reducer<Text,IntWritable,Text,IntWritable> {
   private IntWritable result = new IntWritable();

   public void reduce (Text key,
                       Iterable<IntWritable> values,
                       Context context
                      )
        throws IOException, InterruptedException {
     int sum = 0;
     for (IntWritable val : values) {
       sum += val.get();
     }
     result.set(sum);
     context.write(key, result);
   }
 }
```

Basic Design Questions to Ask

- From where will my input come?
- How is my input structured?
- Mapper and Reducer classes
- Do I need to count anything while job is in progress?
- Where is my output going?
- Executor class
  - Must I block, waiting for job completion?

Resources

- Jeffrey Dean and Sanjay Ghemawat: MapReduce: **Simplified Data Processing on Large Clusters**, Google, Inc.
  - http://labs.google.com/papers/mapreduce.html
- Google Code: **Introduction to Parallel Programming and MapReduce**
  - code.google.com/edu/parallel/mapreduce-tutorial.html
- **Apache Hadoop:** http://hadoop.apache.org/
- **Hadoop Map/Reduce Tutorial**
  - http://hadoop.apache.org/docs/r0.20.2/mapred_tutorial.html
- **Open Source MapReduce**
  - http://lucene.apache.org/hadoop/
- Hadoop: **The Definitive Guide**, by Tom White, 2nd edition, Oreilly's, 2010
- David DeWitt and Michael Stonebraker: **Relational Database Experts Jump The MapReduce Shark**