MI-PDB, MIE-PDB: **Advanced Database Systems**

http://www.ksi.mff.cuni.cz/~svoboda/courses/2015-2-MIE-PDB/

Lecture 10:

MapReduce, Hadoop

26. 4. 2016

Lecturer: **Martin Svoboda**
svoboda@ksi.mff.cuni.cz

Author: **Irena Holubová**
Faculty of Mathematics and Physics, Charles University in Prague

Course NDBI040: **Big Data Management and NoSQL Databases**

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

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

MapReduce

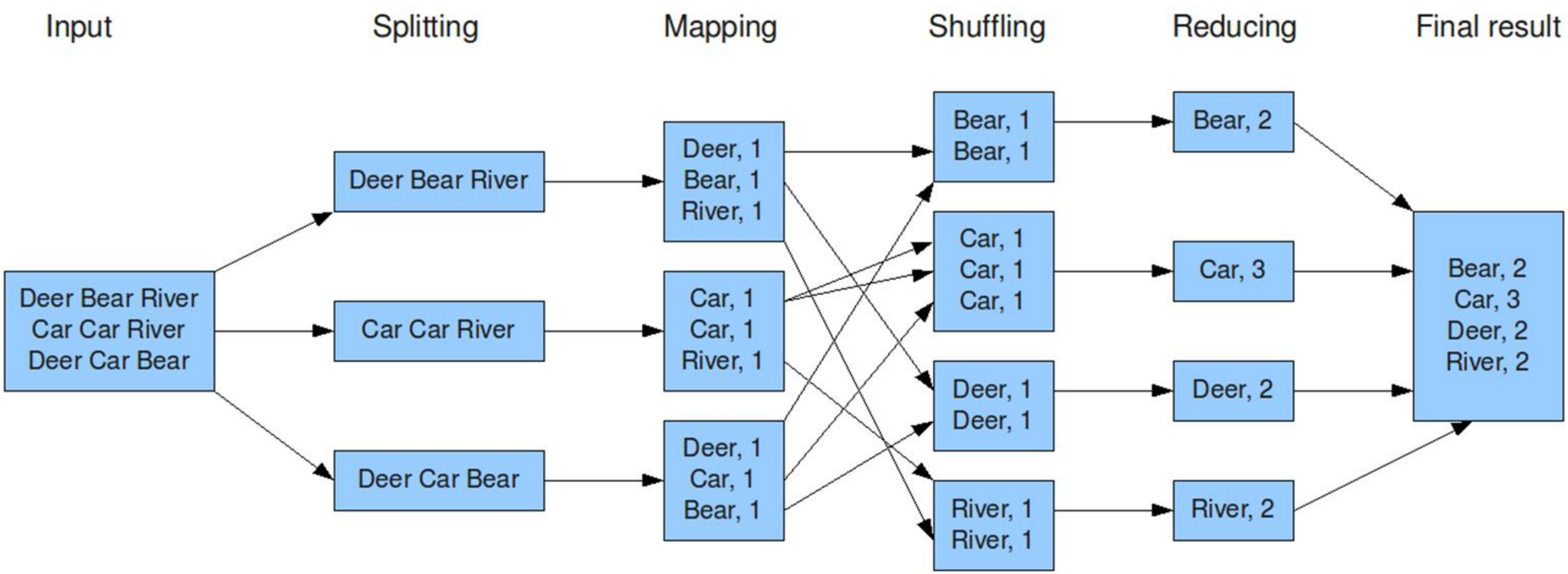
Example: Word Frequency

```
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

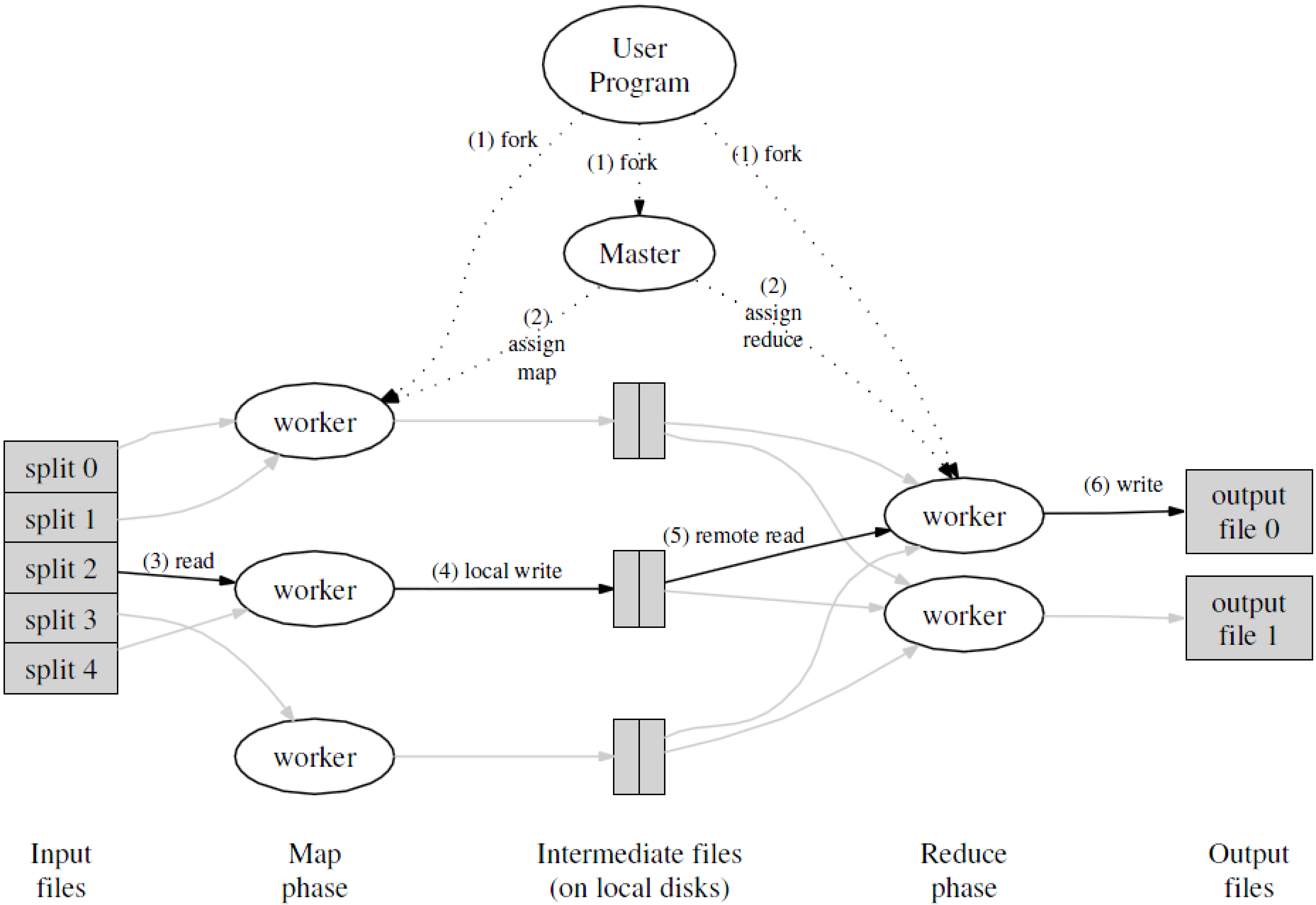


MapReduce
Application Parts

- Input reader
  - Reads data from stable storage
    - e.g., a distributed file system
  - Divides the input into appropriate size 'splits'
  - Prepares 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



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 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 <u>idle</u> workers and assigns each one a Map task (or a Reduce task)

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

MapReduce

Execution – Step 4

- Periodically, the buffered pairs are written to local disk
  - 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

MapReduce

Execution – Step 5

- Reduce worker is notified by the master about data locations
- It uses <u>remote procedure calls</u> 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

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

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
  - All its tasks are reset back to their initial idle 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

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 indexes is uses brute force
3. MapReduce is not novel (ideas more than 20 years old and overcome)
4. MapReduce is missing features common in DBMSs
   - Indexes, transactions, integrity constraints, views, …
5. MapReduce is incompatible with applications implemented over DBMSs
   - Data mining, business intelligence, …

Apache Hadoop

- Open-source software framework
- Running of applications on large clusters of commodity hardware
  - Multi-terabyte data-sets
  - Thousands of nodes
- Implements MapReduce
- Derived from Google's MapReduce and Google File System (GFS)
  - Not open-source

http://hadoop.apache.org/

Apache Hadoop

Modules

- Hadoop Common
  - Common utilities
  - Support for other Hadoop modules
- Hadoop Distributed File System (HDFS)
  - Distributed file system
  - High-throughput access to application data
- Hadoop YARN
  - Framework for job scheduling and cluster resource management
- Hadoop MapReduce
  - YARN-based system for parallel processing of large data sets

HDFS (Hadoop Distributed File System)
Basic Features

- Free and open source
- High quality
- Crossplatform
  - Pure Java
  - Has bindings for non-Java programming languages
- Fault-tolerant
- Highly scalable

HDFS
Data Characteristics

- Assumes:
  - Streaming data access
  - Batch processing rather than interactive user access
- Large data sets and files
- Write-once / read-many
  - A file once created, written and closed does not need to be changed
    - Or not often
  - This assumption simplifies coherency
- Optimal applications for this model: MapReduce, web-crawlers, …

HDFS

Fault Tolerance

- Idea: “failure is the norm rather than exception”
  - A HDFS instance may consist of thousands of machines
    - Each storing a part of the file system’s data
  - Each component has non-trivial probability of failure

→ Assumption: “There is always some component that is non-functional.”

  - Detection of faults
  - Quick, automatic recovery

HDFS

NameNode, DataNodes

- Master/slave architecture
- HDFS exposes file system <u>namespace</u>
- <u>File</u> is internally split into one or more <u>blocks</u>
  - Typical block size is 64MB (or 128 MB)
- NameNode = master server that manages the file system namespace + regulates access to files by clients
  - Opening/closing/renaming files and directories
  - Determines mapping of blocks to DataNodes
- DataNode = serves read/write requests from clients + performs block creation/deletion and replication upon instructions from NameNode
  - Usually one per node in a cluster
  - Manages storage attached to the node that it runs on

HDFS

Namespace

- Hierarchical file system
  - Directories and files
- Create, remove, move, rename, ...
- NameNode maintains the file system
  - Any meta information changes to the file system are recorded by the NameNode
- An application can specify the number of replicas of the file needed
  - Replication factor of the file
  - The information is stored in the NameNode

HDFS

Data Replication

- HDFS is designed to store very large files across machines in a large cluster
  - Each file is a sequence of blocks
  - All blocks in the file are of the same size
    - Except the last one
    - Block size is configurable per file
- Blocks are replicated for fault tolerance
  - Number of replicas is configurable per file

HDFS

How NameNode Works?

- Stores HDFS namespace
- Uses a transaction log called EditLog to record every change that occurs to the file system's meta data
  - E.g., creating a new file, change in replication factor of a file, ..
  - EditLog is stored in the NameNode's local file system
- FsImage – entire file system namespace + mapping of blocks to files + file system properties
  - Stored in a file in NameNode's local file system
  - Designed to be compact
    - Loaded in NameNode's memory
    - 4 GB of RAM is sufficient

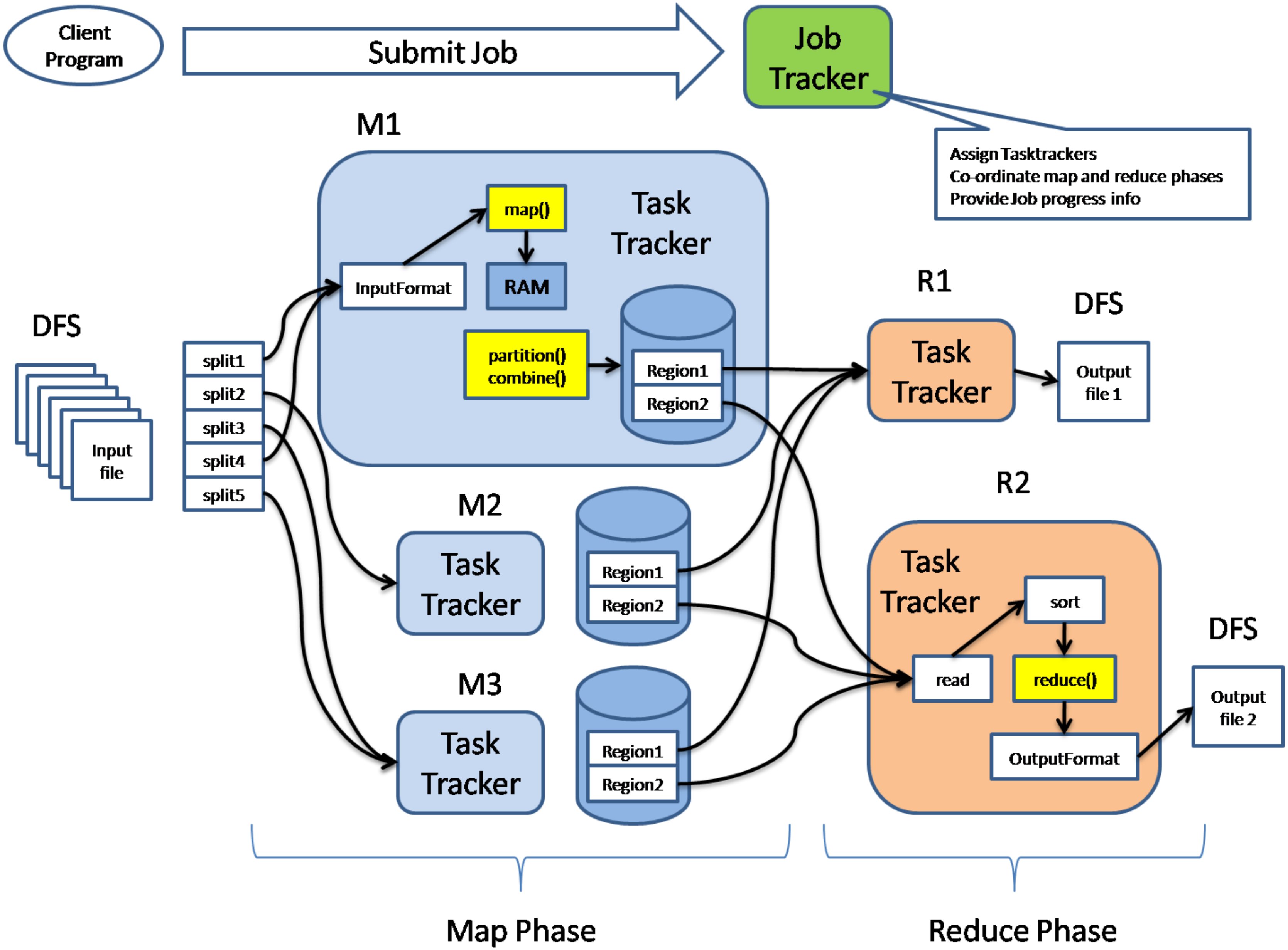
HDFS

How DataNode Works?

- Stores data in files in its local file system
  - Has no knowledge about HDFS file system
- Stores each block of HDFS data in a separate file
- Does not create all files in the same directory
  - Local file system might not be support it
  - Uses heuristics to determine optimal number of files per directory

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


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

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