NDBI040

Big Data Management and NoSQL Databases

Lecture 3. Apache Hadoop

Doc. RNDr. Irena Holubova, Ph.D.
holubova@ksi.mff.cuni.cz

http://www.ksi.mff.cuni.cz/~holubova/NDBI040/

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

Apache Hadoop
Hadoop-related Projects

- Avro – a data serialization system
- Cassandra – a scalable multi-master database with no single points of failure
- Chukwa – a data collection system for managing large distributed systems
- HBase – a scalable, distributed database that supports structured data storage for large tables
- Hive – data warehouse infrastructure that provides data summarization and ad hoc querying
- Mahout – scalable machine learning and data mining library
- Pig – high-level data-flow language and execution framework for parallel computation
- ZooKeeper – high-performance coordination service for distributed applications

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

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
Data Characteristics

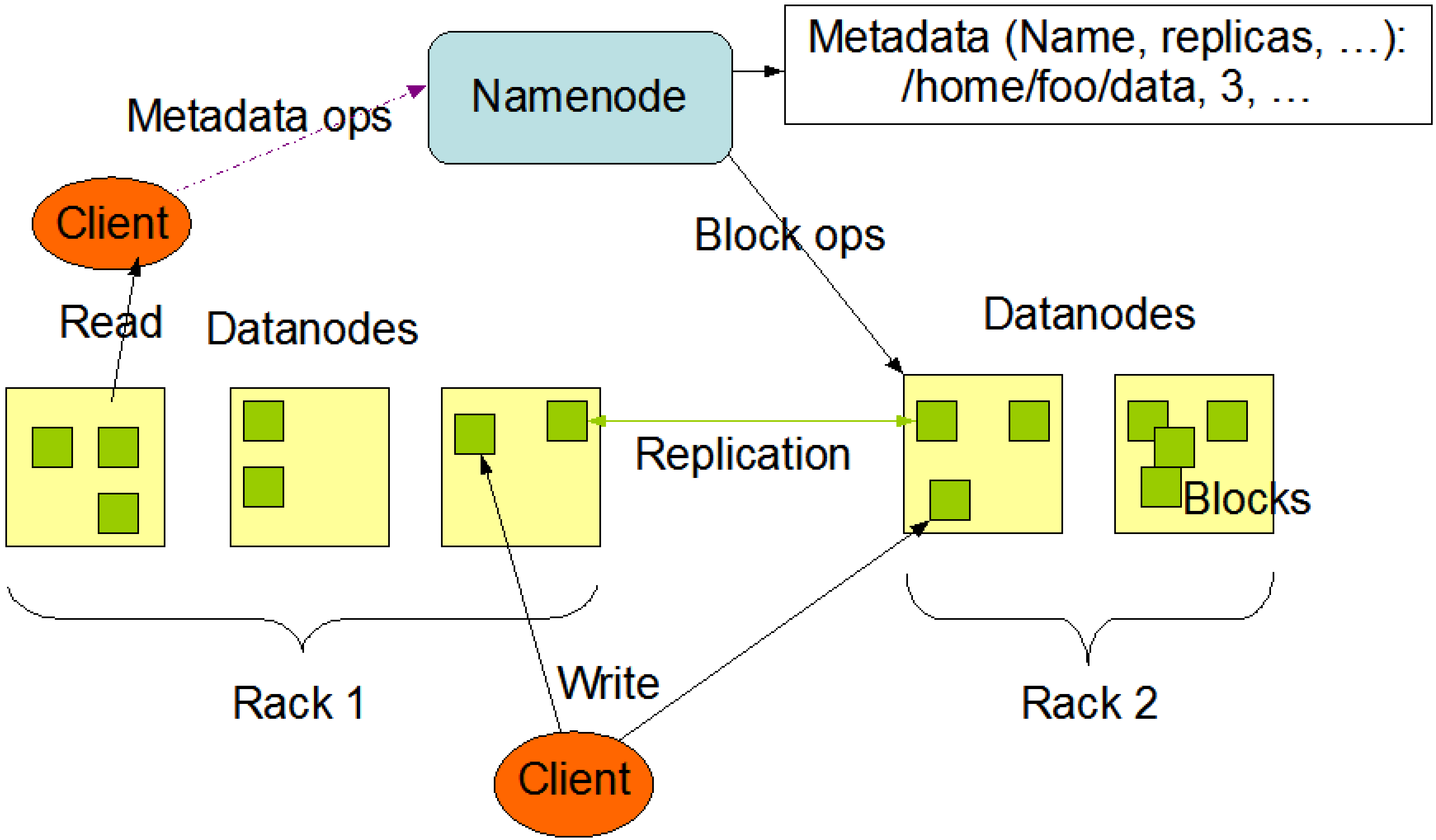
- Assumes:
  - Streaming data access
  - Batch processing rather than interactive user access
- Large data sets and files
- Write-once / read-many
  - A file once created, written and closed does not need to be changed
    - Or not often
  - This assumption simplifies coherency
- Optimal applications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

HDFS

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 Architecture



HDFS

Namespace

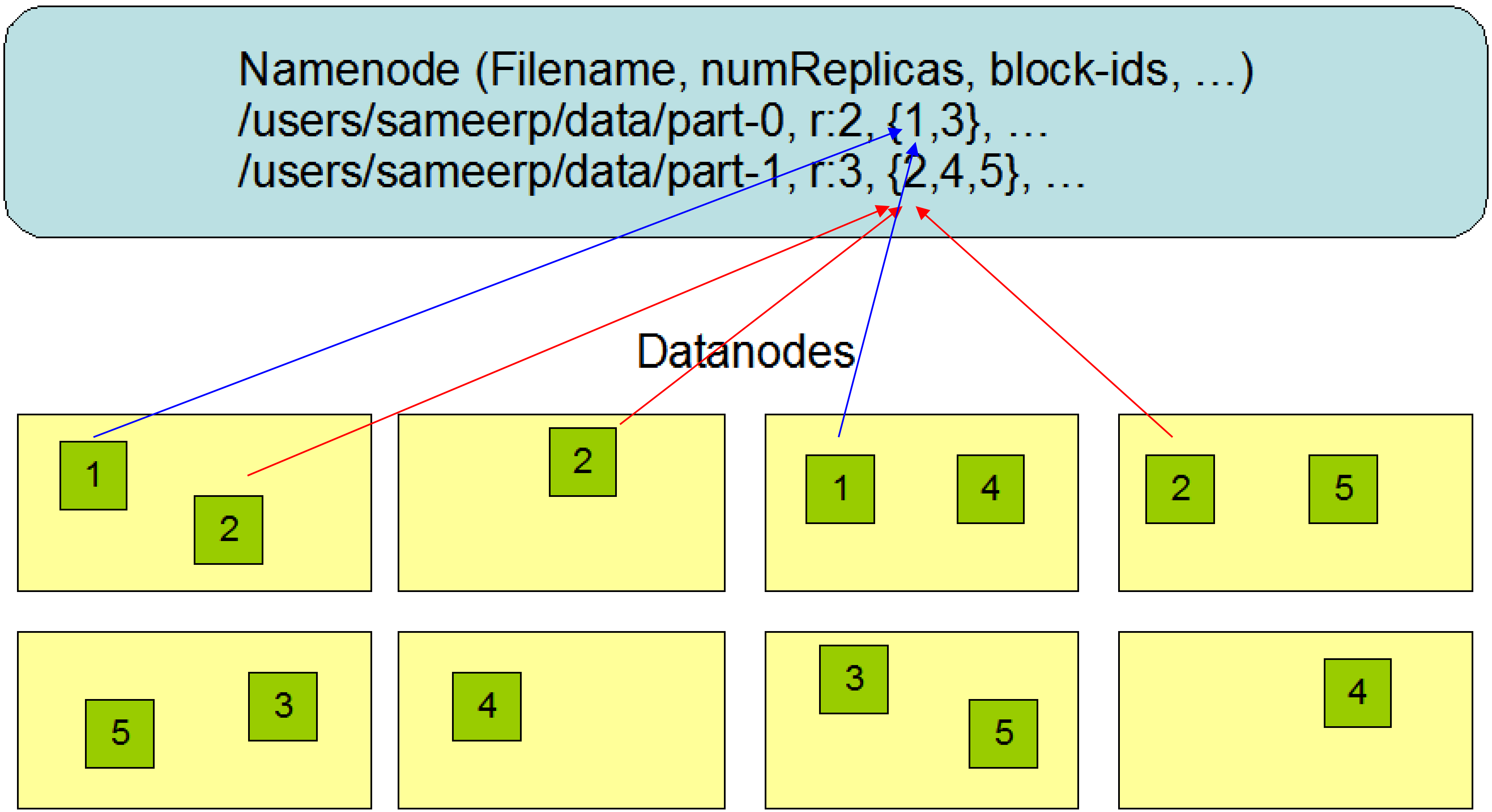
- Hierarchical file system
  - Directories and files
- Create, remove, move, rename, ...
- NameNode maintains the file system
  - Any meta information changes to the file system are recorded by the NameNode
- An application can specify the number of replicas of the file needed
  - Replication factor 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
- NameNode receives HeartBeat and BlockReport from each DataNode
  - BlockReport contains a list of all blocks on a DataNode

Block Replication



HDFS

Replica Placement

- Placement of the replicas is critical to reliability and performance
- Rack-aware replica placement = to take a node's physical location into account while scheduling tasks and allocating storage
  - Needs lots of tuning and experience
- Idea:
  - Nodes are divided into racks
  - Communication between racks through switches
  - Network bandwidth between machines on the same rack is greater than those in different racks
- NameNode determines the rack id for each DataNode

HDFS

Replica Placement

- First idea: replicas should be placed on different racks
  - Prevents losing data when an entire rack fails
  - Allows use of bandwidth from multiple racks when reading data
    - Multiple readers
  - Writes are expensive (transfer to different racks)
    - We need to write to all replicas
- Common case: replication factor is 3
  - Replicas are placed:
    - One on a node in a local rack
    - One on a different node in the local rack
    - One on a node in a different rack
  - Decreases the inter-rack write traffic

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 NameNode Works?

- When the filesystem starts up:
    1. It reads the FsImage and EditLog from disk
    2. It applies all the transactions from the EditLog to the in-memory representation of the FsImage
    3. It flushes out this new version into a new FsImage on disk = checkpoint
    4. It truncates the edit log
- Checkpoints are then built periodically
- Recovery = last checkpointed state

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
- When the file system starts up:
  1. It generates a list of all HDFS blocks = BlockReport
  2. It sends the report to NameNode

HDFS

Failures

- Primary objective: to store data reliably in the presence of failures
- Three common failures:
  - NameNode failure
  - DataNode failure
  - Network partition

HDFS

Failures

- Network partition can cause a subset of DataNodes to lose connectivity with NameNode
  - NameNode detects this condition by the absence of a Heartbeat message
  - NameNode marks DataNodes without HearBeat and does not send any IO requests to them
  - Data registered to the failed DataNode is not available to the HDFS
- The death of a DataNode may cause replication factor of some of the blocks to fall below their specified value → re-replication
  - Also happens when replica is corrupted, hard disk fails, replication factor is increased, …

HDFS

API

- Java API for application to use
  - Python access can be used
  - C language wrapper for Java API is available
- HTTP browser can be used to browse the files of a HDFS instance
- Command line interface called the FS shell
  - Lets the user interact with data in the HDFS
  - The syntax of the commands is similar to bash
  - e.g., to create a directory `/foodir`

```
/bin/hadoop fs -mkdir /foodir
```

Hadoop file system

- Browser interface is available to view the namespace

http://hadoop.apache.org/docs/r0.19.2/hdfs_shell.html

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

Preparing for 'grep' Example in Hadoop

- Hadoop's jobs operate within the HDFS
  - Read input from HDFS, write output to HDFS
- To prepare:
  - Download, e.g., a free electronic book
  - Load the file into HDFS

```
bin/hadoop fs -copyFromLocal book.txt
  /book.txt
```

Using 'grep' within Hadoop

```
bin/hadoop jar \
hadoop-0.18-2-examples.jar \
grep /book.txt /grep-result "search string"
```

input

output

```
bin/hadoop fs -ls /grep-result
```

How 'grep' in Hadoop Works

Bigger Example

- The program runs two Map/Reduce jobs in sequence
  - First job: counts how many times a matching string occurred
  - Second job: sorts matching strings by their frequency and stores the output in a single output file
- The first job:
  - Each mapper:
    - Takes a line as input and matches the given regular expression
    - Extracts all matching strings and emits (matching string, 1) pairs
  - Each reducer:
    - Sums the frequencies of each matching string
  - The output is a sequence of files containing the matching string and frequency
    - Combiner: sums the frequency of strings from a local map output

How 'grep' in Hadoop Works

- The second job:
  - Takes the output of the first job as input
    - Mapper is an inverse map
    - Reducer is an identity reducer
  - The number of reducers is one → the output is stored in one file
    - Sorted by the frequency in a descending order

MapReduce

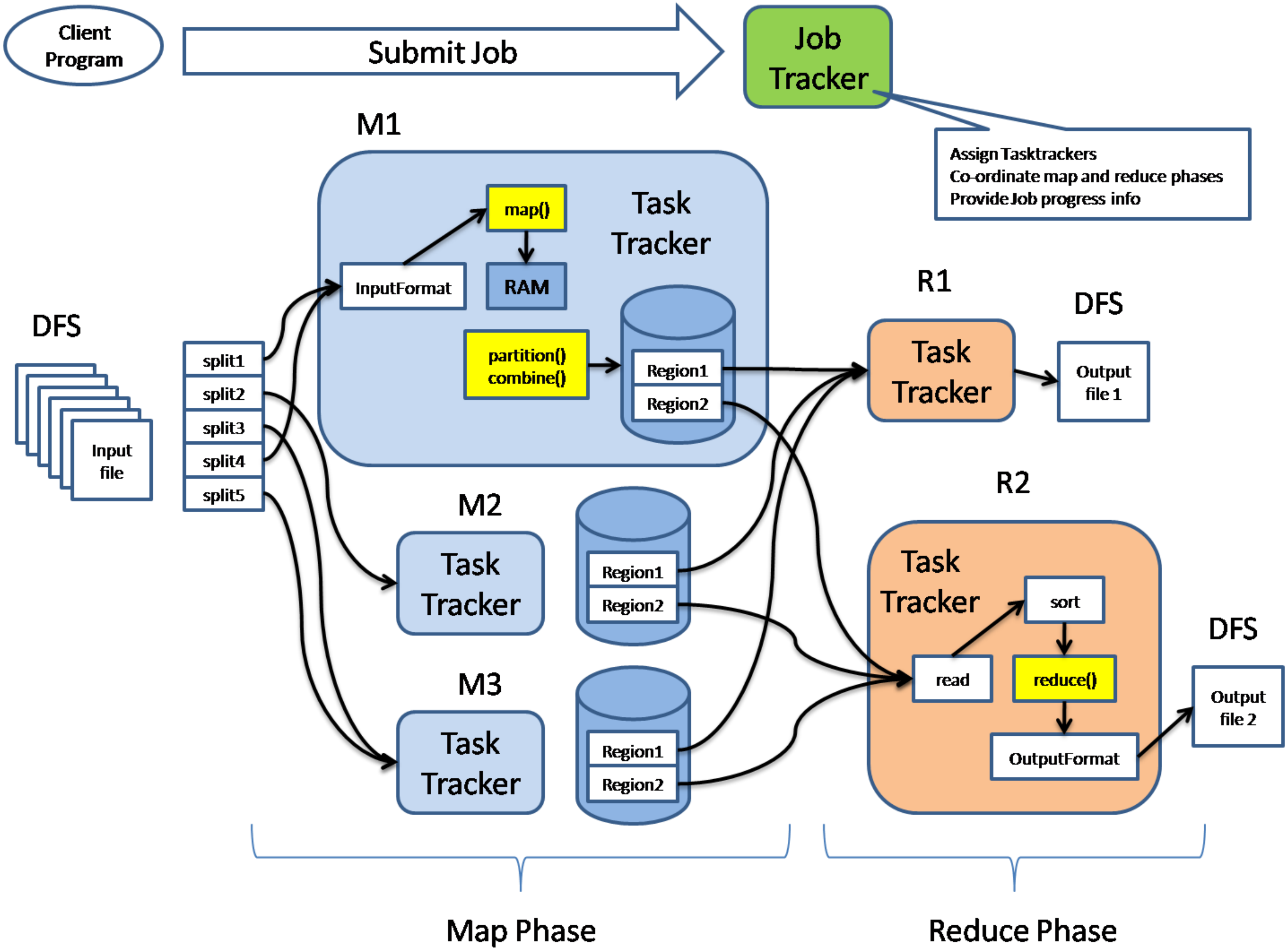
JobTracker (Master)

- 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

JobConf

- For launching program:
  1. Create a JobConf to define a job
     - Configuration
  2. Submit JobConf to JobTracker and wait for completion
- JobConf involves:
  - Classes implementing Mapper and Reducer interfaces
    - `JobConf.`**`setMapperClass`**`()`
    - `JobConf.`**`setReducerClass`**`()`
  - Input and output formats
    - `JobConf.`**`setInputFormat`**`(TextInputFormat.class)`
    - `JobConf.`**`setOutputFormat`**`(TextOutputFormat.class)`
  - Other options:
    - `JobConf.`**`setNumReduceTasks`**`()`
    - …

Job Launching

InputFormat, OutputFormat

- Define how the persistent data is read and written
- InputFormat
  - Splits the input to determine the partial input to each map task
  - Defines a RecordReader that reads key, value pairs that are passed to the map task
- OutputFormat
  - Given the key, value pairs and a filename, it writes the reduce task output to a persistent store

Job Launching
JobClient

- JobConf is passed to `JobClient.`**`runJob`**`()` or `JobClient.`**`submitJob`**`()`
  - `runJob()` blocks – waits until the job finishes
  - `submitJob()` does not block
    - Poll for status to make running decisions
    - Avoid polling with `JobConf.`**`setJobEndNotificationURI`**`()`
      - Provide a URI to be invoked when the job finishes
- JobClient
  - Determines proper division of input into InputSplits
  - Sends job data to master JobTracker server

Mapper

- The user provides an instance of Mapper
  - Should extend `MapReduceBase`
  - Should implement interface `Mapper`
    - Override function `map`
  - Emits (k_2,v_2) with `output.collect(k2, v2)`
- Exists in separate process from all other instances of Mapper
  - No data sharing

input key

input value

```
void map (WritableComparable key,
          Writable value,
          OutputCollector output,
          Reporter reporter)
```

facility to report progress

collects output keys and values

```java
public static class Map
   extends MapReduceBase
   implements Mapper<LongWritable, Text, Text, IntWritable> {

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

    public void map(LongWritable key,
                    Text value,
                    OutputCollector<Text, IntWritable> output,
                    Reporter reporter) throws IOException {
        String line = value.toString();
        StringTokenizer tokenizer = new StringTokenizer(line);

        while (tokenizer.hasMoreTokens()) {
             word.set(tokenizer.nextToken());
             output.collect(word, one);
          }
     }
}
```

What is Writable and Reporter?

- Hadoop defines its own “box” classes for strings (`Text`), integers (`IntWritable`), …
  - All values are instances of `Writable`
  - All keys are instances of `WritableComparable`
- `Reporter` allows simple asynchronous feedback
  - `incrCounter(Enum key, long amount)`
  - `setStatus(String msg)`

Partitioner

- Controls which of the *R* reduce tasks the intermediate key is sent for reduction

```
int getPartition(K2 key,
                 V2 value,
                 int numPartitions)
```

  - Outputs the partition number for a given key
  - One partition = one Reduce task

- `HashPartitioner` used by default
  - Uses `key.hashCode()` to return partition number
- JobConf sets Partitioner implementation

Reducer

```
reduce(WritableComparable key,
       Iterator values,
       OutputCollector output,
       Reporter reporter)
```

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

```java
public static class Reduce
    extends MapReduceBase
    implements Reducer<Text, IntWritable, Text, IntWritable> {

      public void reduce(Text key,
                         Iterator<IntWritable> values,
                         OutputCollector<Text, IntWritable> output,
                         Reporter reporter) throws IOException {
          int sum = 0;
          while (values.hasNext()) {
                sum += values.next().get();
          }
          output.collect(key, new IntWritable(sum));
       }
}
```

Design Questions to Ask

- From where will my input come?
  - InputFileFormat
- How is my input structured?
  - RecordReader
    - LineRecordReader, KeyValueRecordReader
  - (Do not reinvent the wheel.)
- Mapper and Reducer classes
  - Do Key (WritableComparator) and Value (Writable) classes exist?
- Do I need to count anything while job is in progress?
- Where is my output going?
- Executor class
  - What information do my map/reduce classes need?
  - Must I block, waiting for job completion?

References

- Apache Hadoop: http://hadoop.apache.org/
- http://wiki.apache.org/hadoop/
- Hadoop: **The Definitive Guide**, by Tom White, 2nd edition, Oreilly's, 2010
- Dean, J. and Ghemawat, S. 2008. **MapReduce: Simplified Data Processing on Large Clusters**. Communication of ACM 51, 1 (Jan. 2008), 107-113.