HADOOP ARCHITECTURE

BIG DATA PROCESSING

Félix Cuadrado

felix.cuadrado@qmul.ac.uk

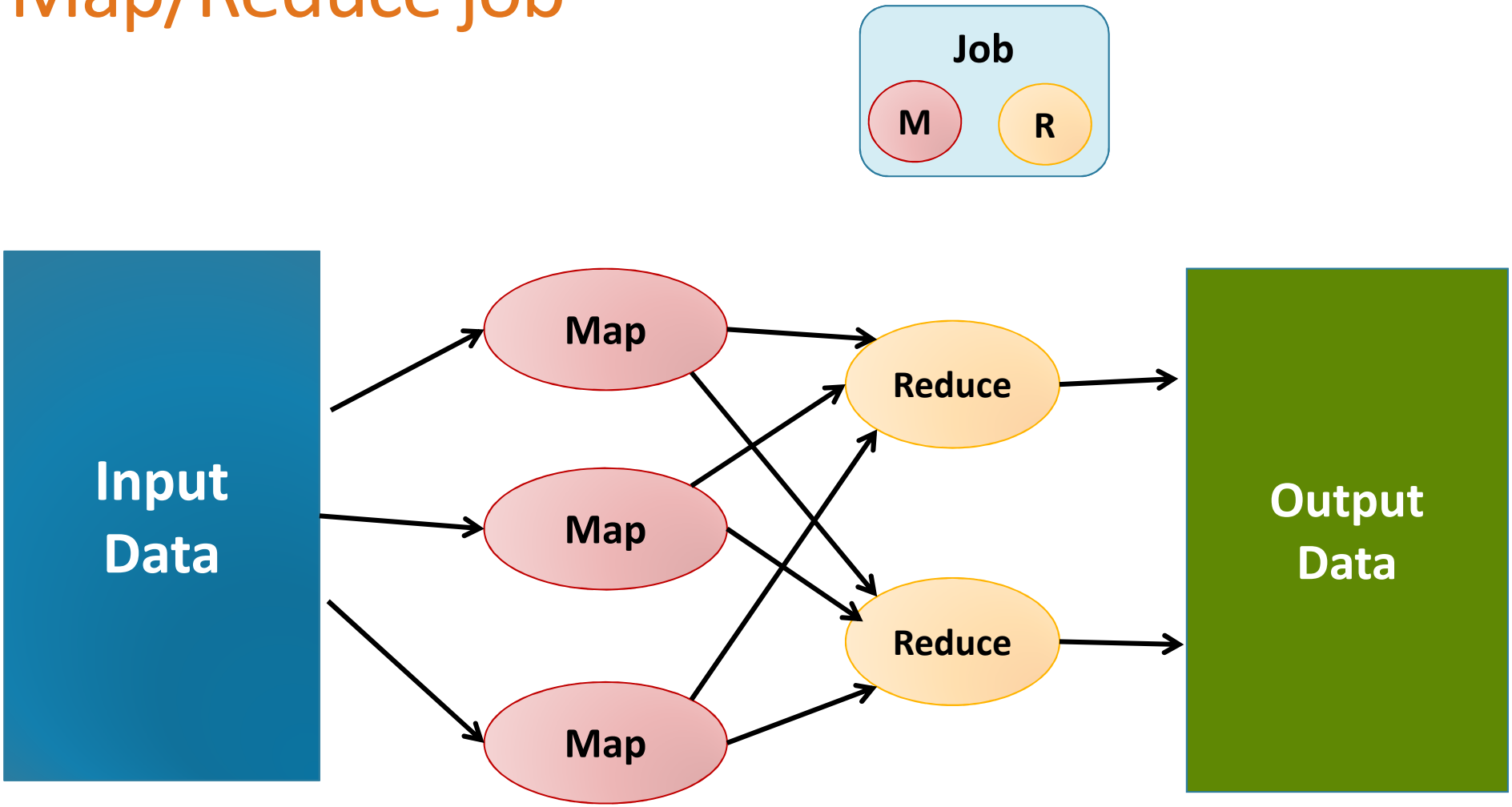
Queen Mary University of London

School of Electronic Engineering and Computer Science

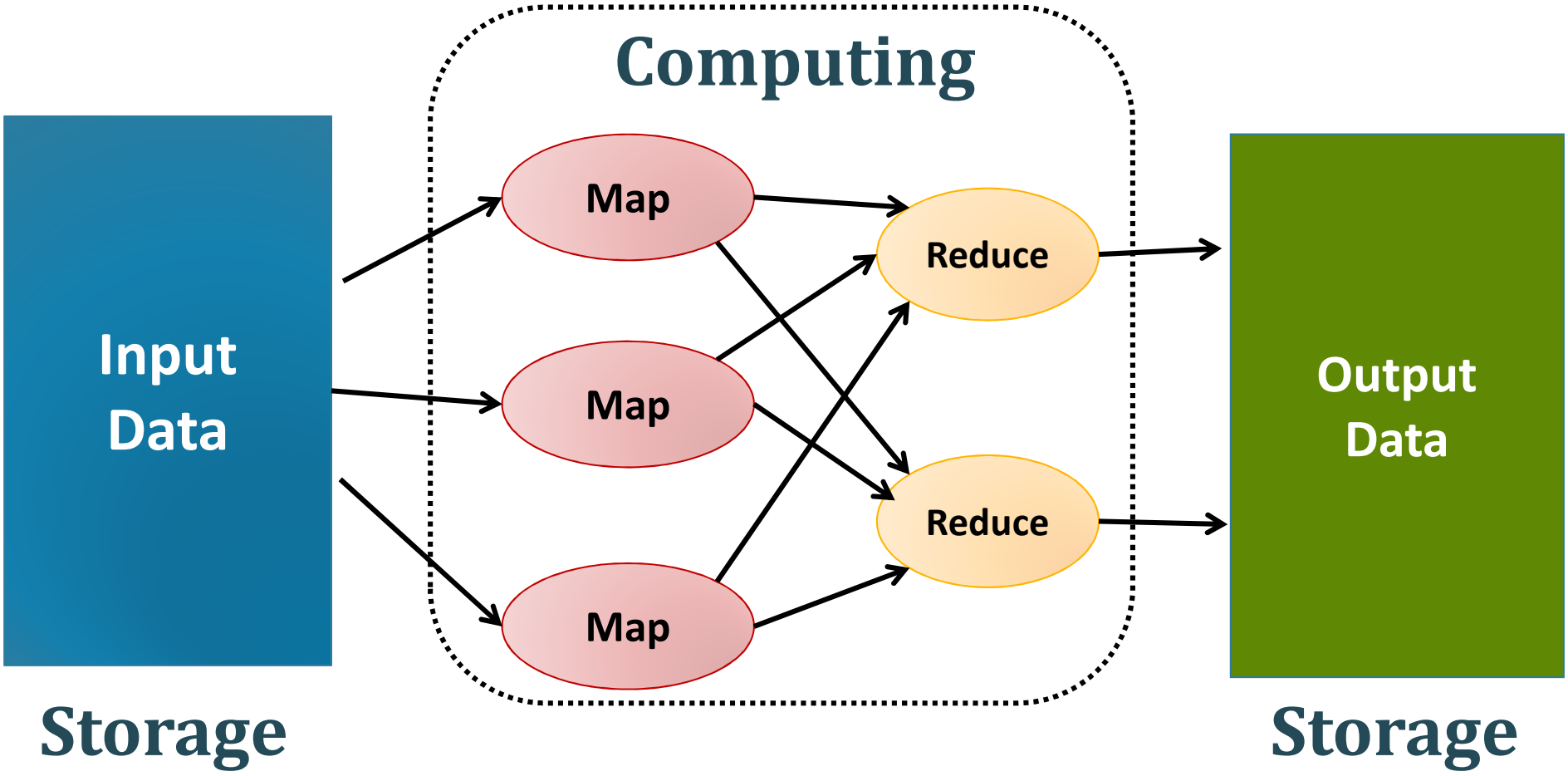
Contents

- **Apache Hadoop architecture**
- Hadoop job execution: YARN
- Hadoop storage: HDFS
- MapReduce programming patterns

Map/Reduce job



Map/Reduce framework roles



Hadoop Architecture

- Hadoop executes on a cluster of networked PCs
- Each node runs a set of daemons

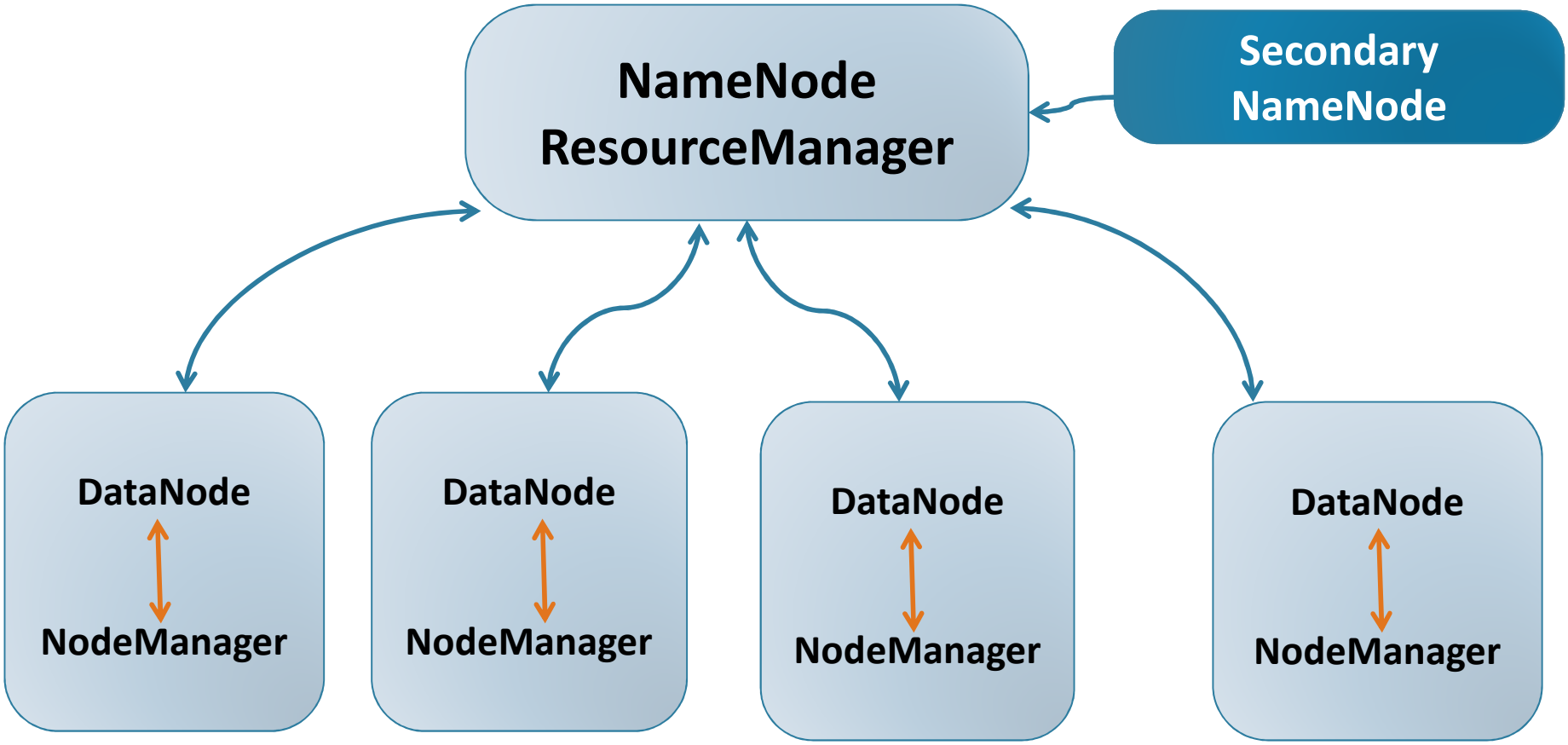
Computing
- ResourceManager
- NodeManager

Storage
- NameNode
- SecondaryNameNode
- DataNode

Master Slave Architecture

- Master (1)
  - Is aware of all the slave nodes
  - Receives external requests
  - Decides who executes what, and when
  - Speaks with slaves
- Slave (1..*)
  - Worker node
  - Executes the tasks the master tells it to do

Hadoop Master-Slave architecture



Queen Mary University of London

Contents

- Apache Hadoop architecture
- **Hadoop job execution: YARN**
- Hadoop storage: HDFS
- MapReduce programming patterns

Hadoop computation tasks

- Resource management
  - Being aware of what resources are in the cluster
  - Which resources are available now
- Job allocation
  - How many resources are needed to compute the job
  - Which nodes should execute each of the tasks
- Job execution
  - Coordinate task execution from workers
  - Make sure the job completes, deal with failures

Hadoop Job allocation

- Resource management needs to estimate **how many Map and Reduce tasks** are needed for a given job
  - Based on input dataset
  - Based on job definition
- Ideally, one different node will be allocated for each different Map/Reduce tasks
  - Otherwise multiple tasks can go to same node

Job execution: Complete MapReduce job flow

1. Split (logically) input data into computing chunks
2. Assign one chunk to a (co-located) NodeManager
3. Run 1..* **Mappers**
4. Shuffle and Sort
5. Run 1..* **Reducers**
6. Results from Reducers create the job output

How many Mappers are needed?

- Mapper parallelization:
  - Each Mapper processes a different **input split**
  - Input dataset size is known
- Nmappers = input size / split size
  - If input has multiple small files, more Mappers can be invoked (Hadoop inefficiency)

How many Reducers are needed?

- Reducer parallelization
  - **keys are partitioned** across the reducers
  - Hard to automatically estimate what is the right number
  - Too many Reducers can complicate too much shuffle and sort.
- Nreducers = **User defined parameter**
  - (in MapReduce job definition)

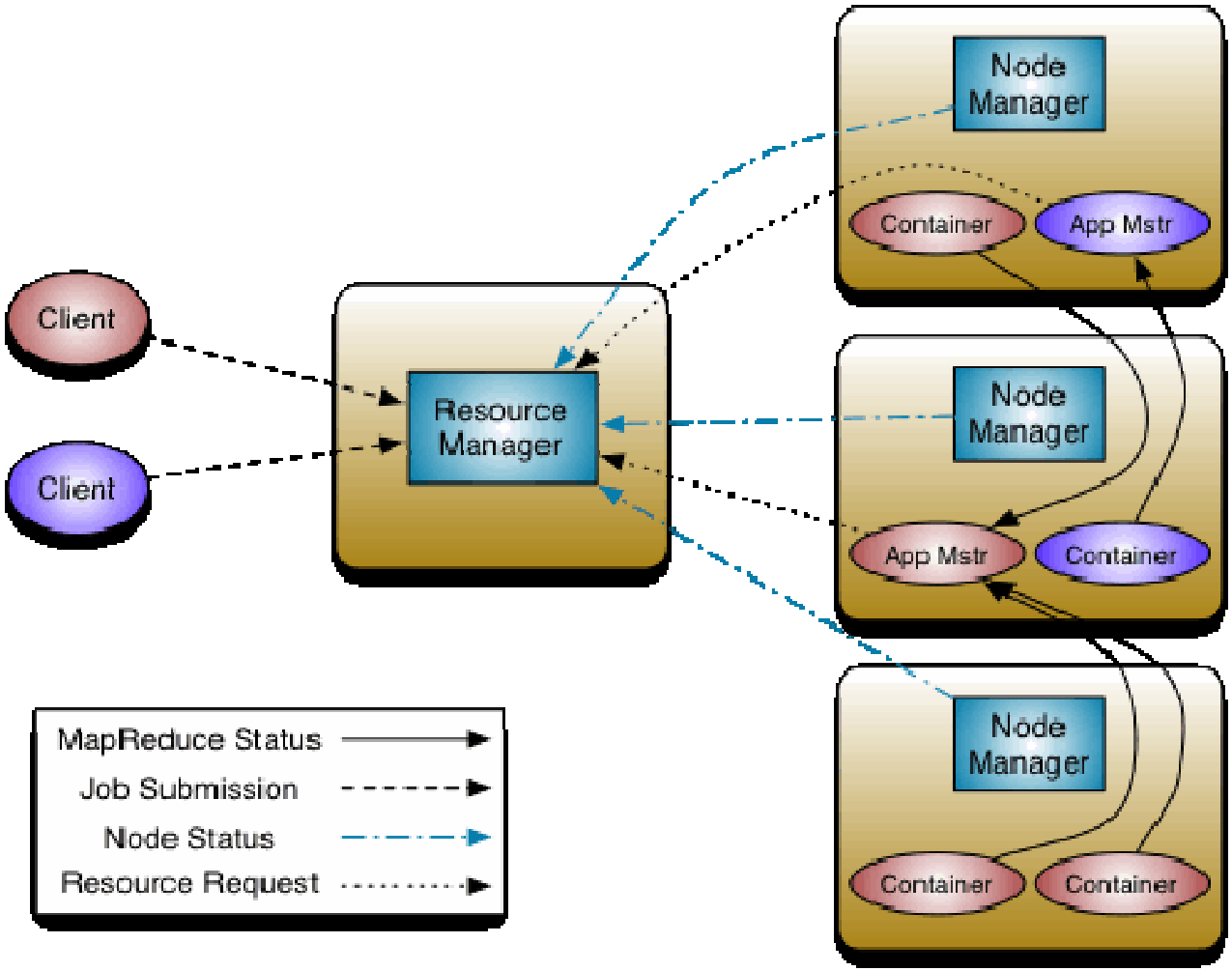
Hadoop Execution daemons

- ResourceManager (1 per cluster)
  - Receives job requests from Hadoop Clients
  - Creates one ApplicationMaster per job to manage it
  - Allocates Containers in slave nodes, with assigned resources
  - Keeps track of health of NodeManager nodes
- NodeManager (1..* per cluster)
  - Coordinates execution of Map and Reduce tasks at node
  - Sends heartbeat messages to ResourceManager

ApplicationMaster

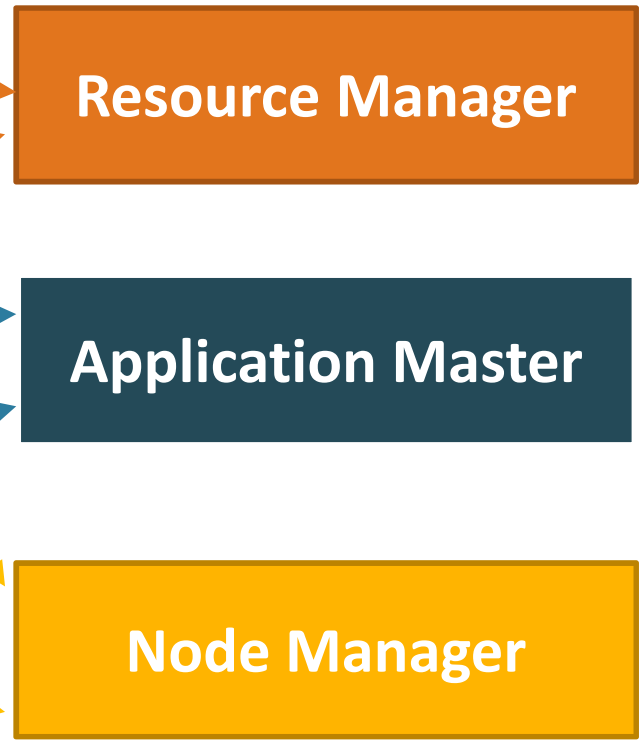
- One per job. Implements the specific computing framework
  - After creation, negotiates with ResourceManager how many resources will be required for the job
  - Decides which nodes will run Map and Reduce jobs among the Containers given by the ResourceManager
  - Reports to the ResourceManager about the progress and completion of the whole job
  - Is destroyed when the job is completed

Job Execution Architecture (YARN)



Responsibilities on computation tasks

- Resource management
- Job allocation
- Job execution



Contents

- Apache Hadoop architecture
- Hadoop job execution: YARN
- **Hadoop storage: HDFS**
- MapReduce programming patterns

HDFS

- Hadoop Distributed Filesystem
  - Shared storage among the nodes of the Hadoop cluster
- Storage for Input and output of MapReduce jobs
- HDFS is Tailored for MapReduce jobs
  - Large block size (128MB default)
    - But not too large, blocks define the minimum parallelization unit
  - HDFS is not a POSIX compliant Filesystem
    - Tradeoffs for improving data processing throughput

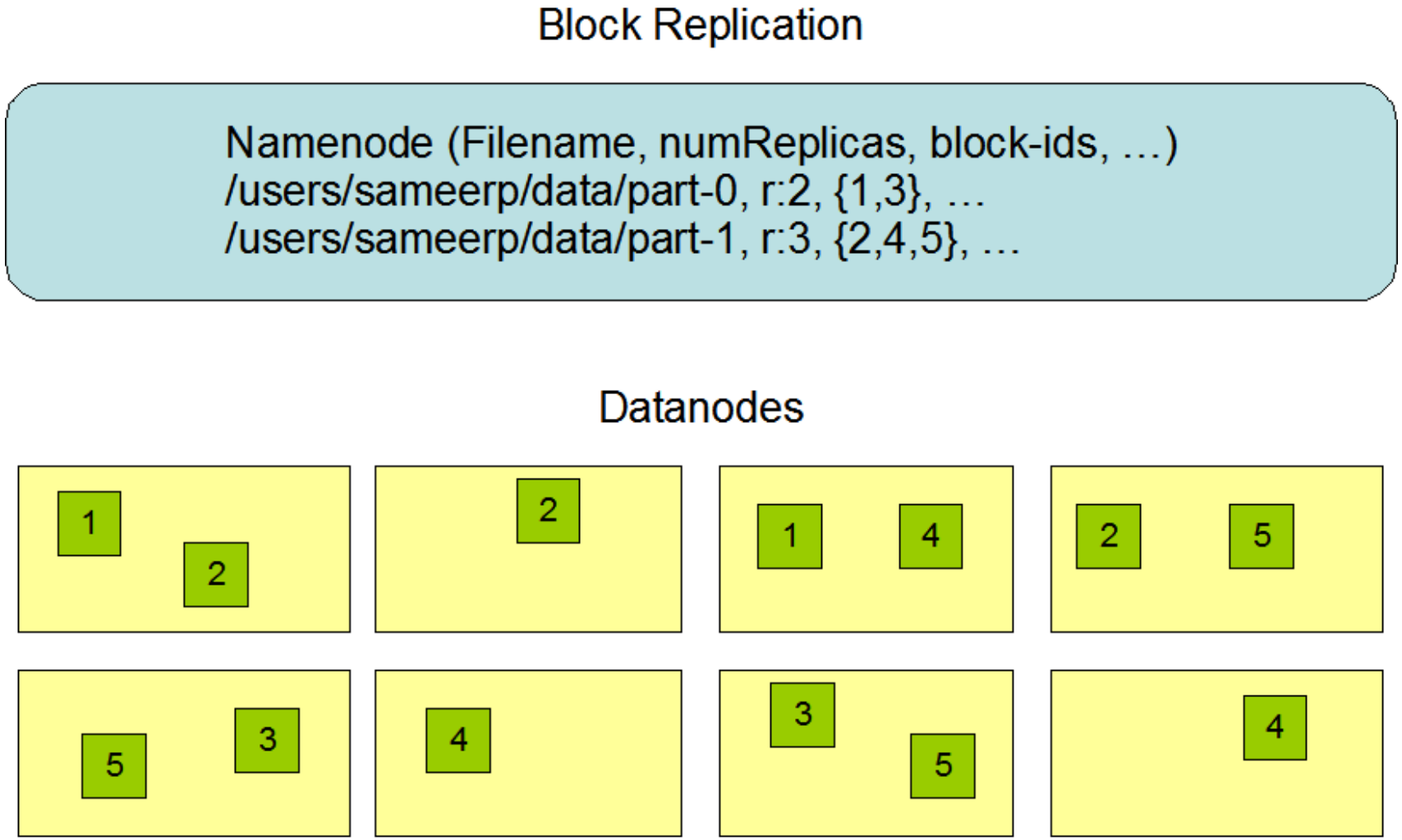
HDFS Data distribution

- Data distribution is a key element of the MapReduce model and architecture
- “Move computation to data” principle
- Blocks are replicated over the cluster
  - Default ratio is three times
  - spread replicas among different physical locations
  - Improves reliability

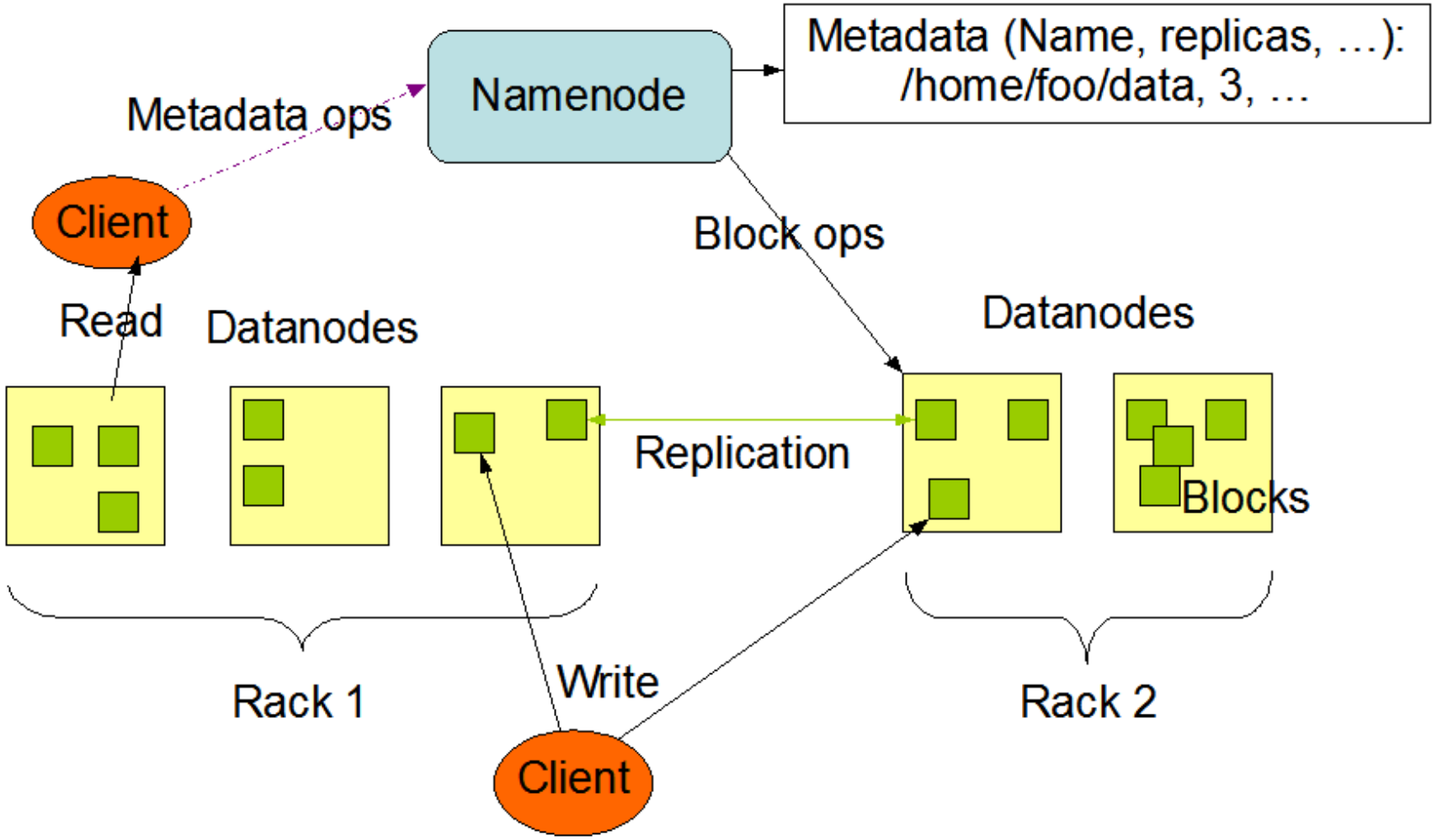
Hadoop Storage Daemons

- DataNode (1..* per cluster)
  - Stores blocks from the HDFS
  - Report periodically to NameNode list of stored blocks
- NameNode (1 per cluster)
  - Keeps index table with (all) the locations of each block
  - Heavy task, no computation responsibilities
  - Single point of failure
- Secondary Namenode (1 per cluster)
  - Communicates periodically with NameNode
  - Stores backup copy of index table

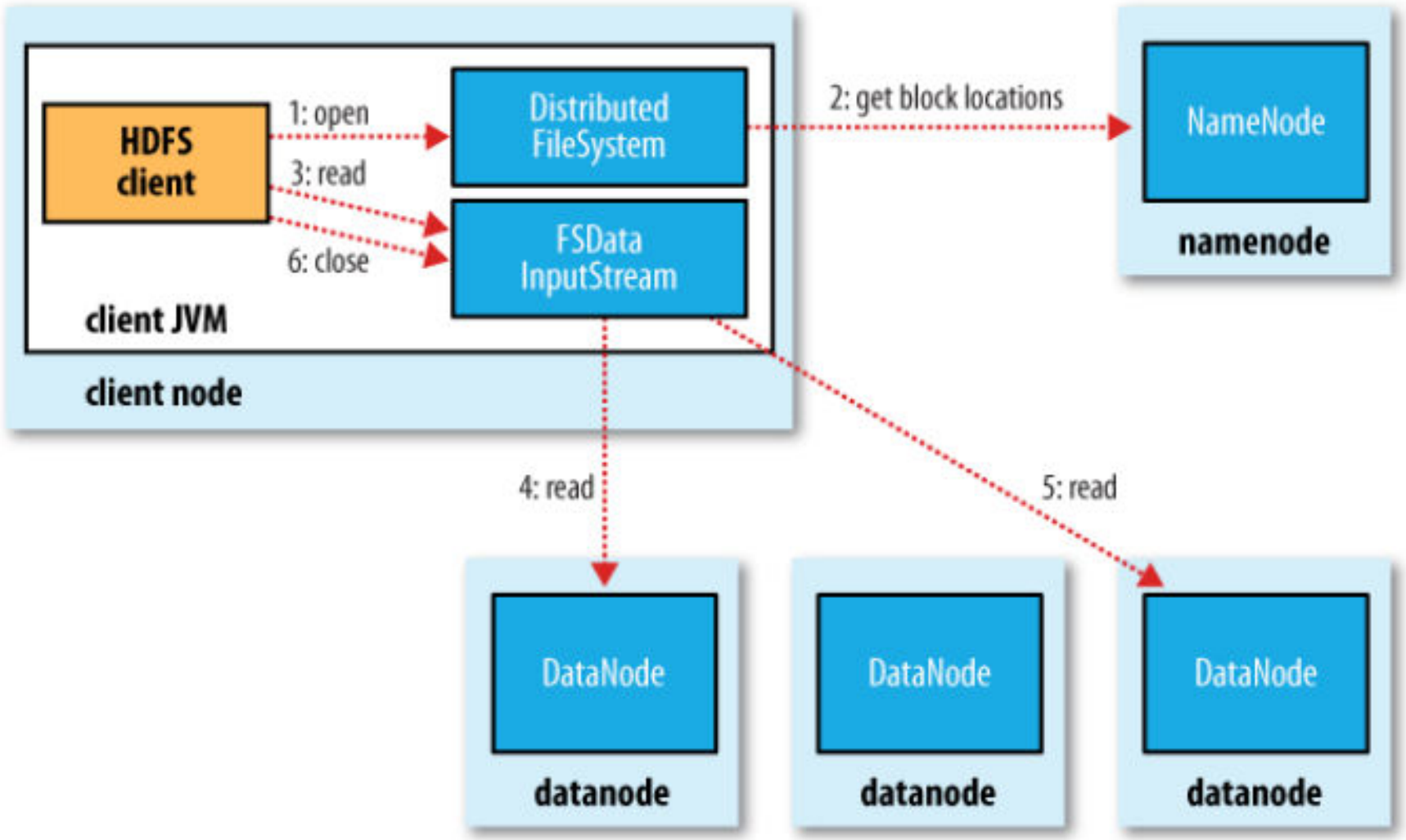
Data replication



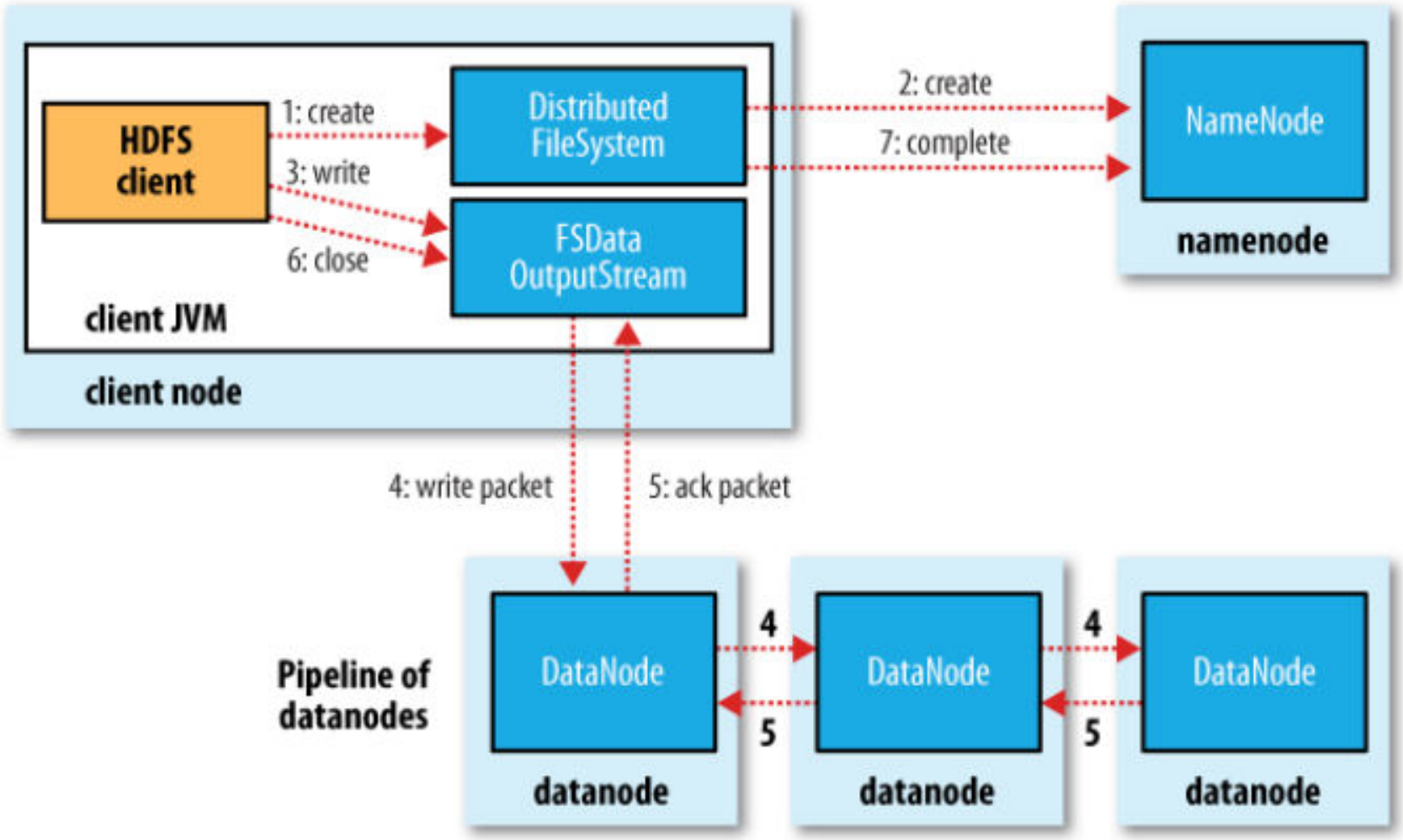
HDFS Usage


Metadata ops
Namenode
Metadata (Name, replicas, …):
/home/foo/data, 3, …
Client
Block ops
Read
Datanodes
Datanodes
Replication
Blocks
Rack 1
Write
Rack 2
Client

HDFS File Read operation



HDFS File Write Operation



MR Job input and output data

1. **Input data -> Mappers**
   - Mappers are assigned input splits from HDFS input path
     - (default 64MB)
   - Data locality optimization: ApplicationManager attempts to assign Mappers where data block is stored
2. **Reducers -> Output data**
   - Reducer output copied to HDFS
     - One file per Reducer
   - For reliability concerns, HDFS replication

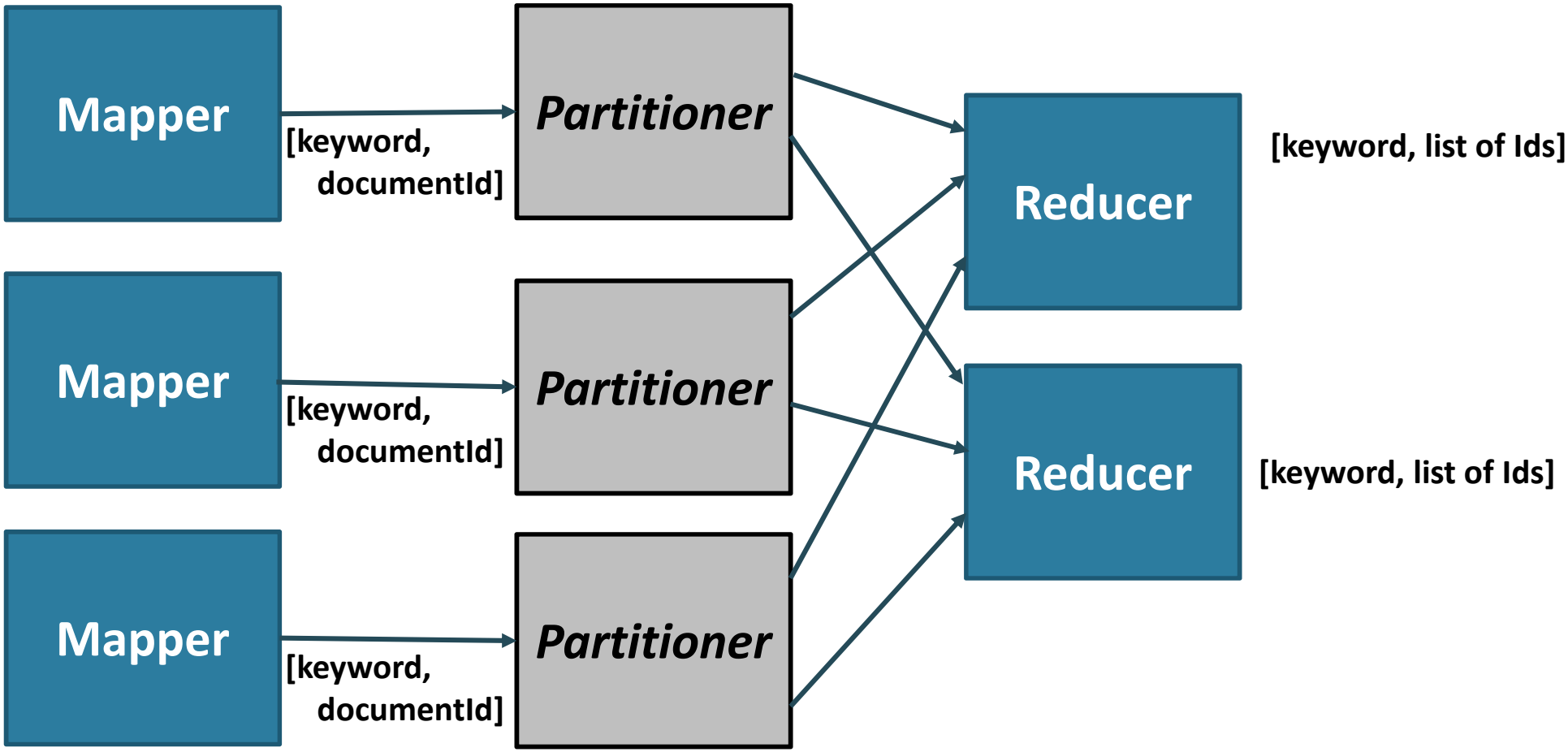
Contents

- Apache Hadoop architecture
- Hadoop job execution: YARN
- Hadoop storage: HDFS
- **MapReduce programming patterns**

Inverted index

- Goal: Generate index from a dataset to allow faster searches for specific features
- Examples:
  - Building index from a textbook.
  - Finding all websites that match a search term

Inverted index Structure



Inverted Index Mapper

```
public void map (String docId, String text){

    String[] features = findFeatures(text);
      for(String feature: features){
           emit(feature, docId);
      }
}
```

Inverted Index Reducer

```
public void reduce (String feature,
                                String[] docIds){

    emit(feature, formatNicely(docIds))

}
```

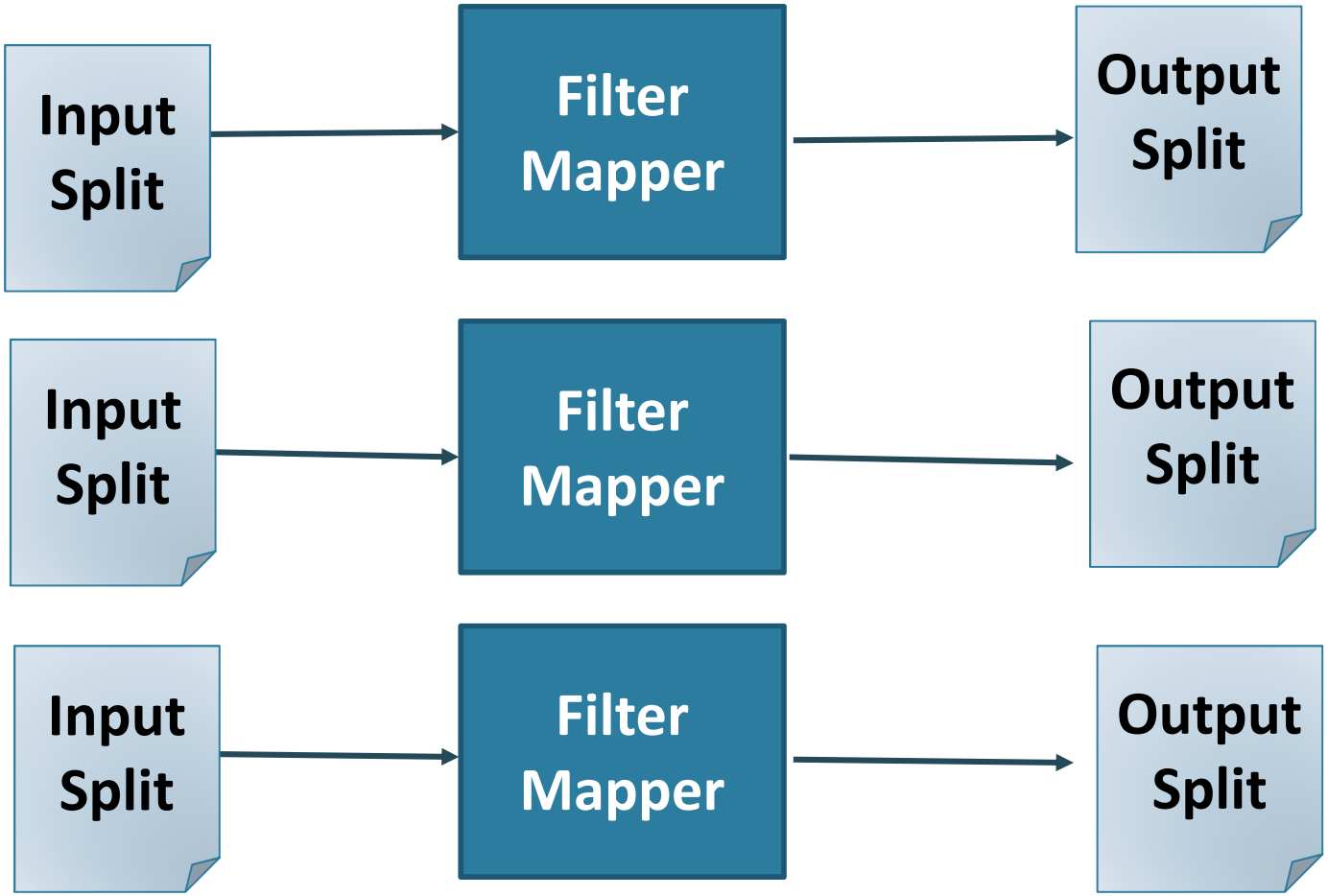
Writing Map and Reduce functions

- Mapper
  - Find features in Input
  - Emit [feature, document identifier]
- Reducer
  - Identity function (emits the list of results provided in shuffle and sort)
- Combiner?

Filtering

- Goal: Filter out records/fields that are not of interest for further computation.
- Speedup the actual computation thanks to a reduced size of the dataset
- Examples:
  - distributed grep.
  - Tracking a thread of events (logs from the same user)
  - data cleansing
- Mapper only job

Filtering Structure



Top ten elements

- Goal: Retrieve a small number of records, relative to a ranking
- Examples:
  - build top sellers view
  - find outliers in the data.

Top Ten Map and Reduce functions

- Mapper
  - Emits null/unique key, value with (ranking, record)
- Reducer
  - Sort all values by ranking, emit k times null/unique key, value with (ranking, record)
- Combiner
  - Same as Reducer

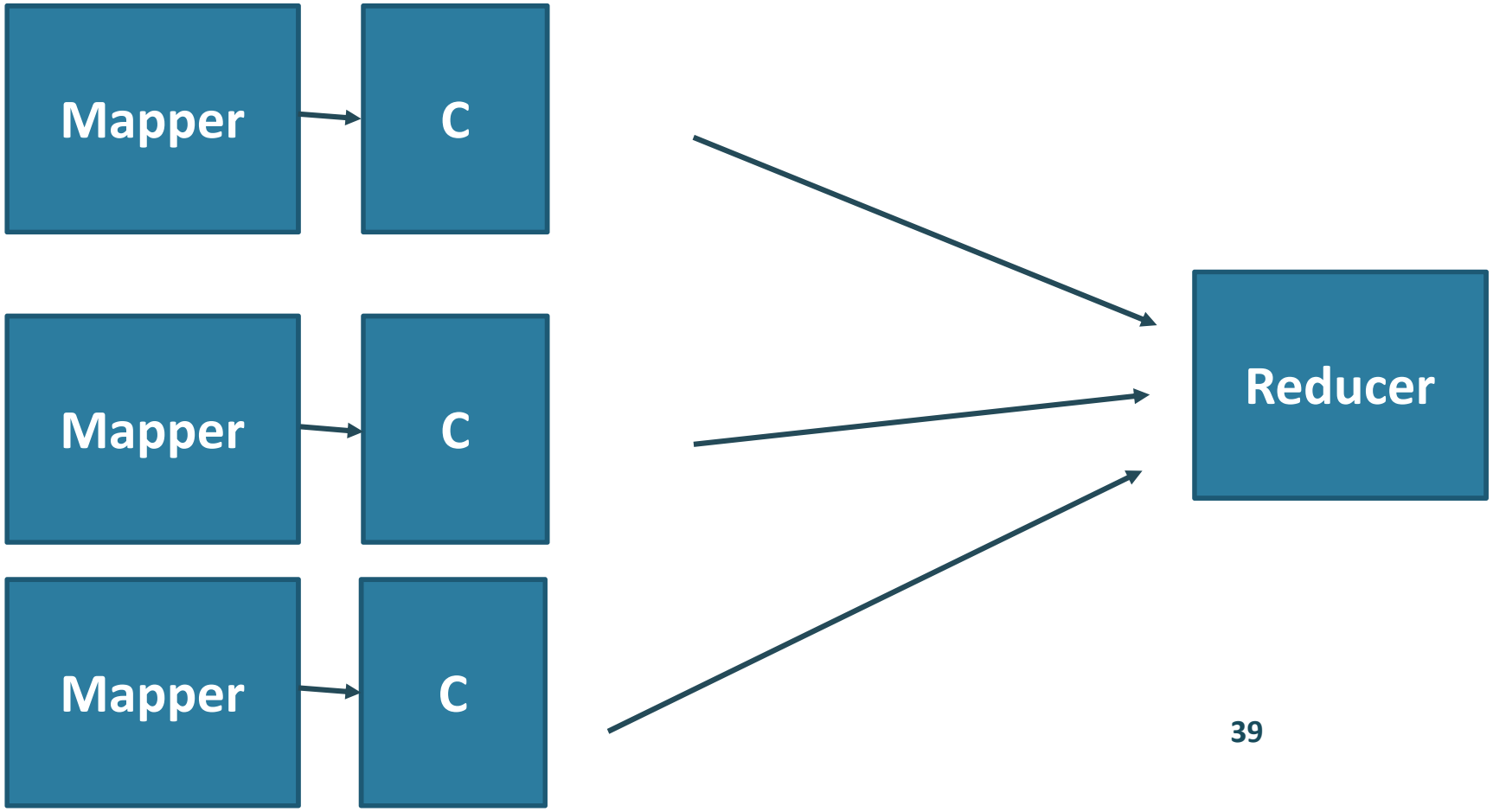
Top Ten Mapper

```
public void map (String studentId, double grade){

      emit(null, new Pair(studentId,grade));

}
```

Top Ten Reducer

```
public void reduce (null, Pair[] studentResults){
      list top10 = studentResults.sort().getTop(10);
      int rank = 1;
      for(Pair student: topTen){
            emit(rank, student.getId());
            rank++;
      }
}
```

Top Ten Structure



39

Top Ten Performance

- How many Reducers?
  - Performance issues?
- What happens if we don't use Combiners?
- Performance depends greatly on the number of elements, (to a lesser extend on the size of data)
- Minimum requirement: the ranking data for a whole input split must fit into memory of a single Mapper

Recommended reading

- Hadoop YARN: Yet Another Resource Negotiator
  - http://www.socc2013.org/home/program/a5-vavilapalli.pdf
- How MapReduce works, Chapter 6, Hadoop: The Definitive Guide, 5th Edition.
  - Available in QMUL Safaribooksonline
- HDFS design: http://hadoop.apache.org/docs/r1.2.1/hdfs_design.html