Large-Scale Data Management and Distributed Systems

II. MapReduce and Hadoop

Vania Marangozova

Vania.Marangozova@imag.fr

2023-2024

References

- Coursera – Big Data, University of California San Diego
- The lecture notes of V. Leroy
- Designing Data-Intensive Applications by Martin Kleppmann
- Mining of Massive Datasets by Leskovec et al.
- Lecture notes of T.Ropars

Agenda

- Introduction to MapReduce
- The Hadoop Eco-System
- HDFS (Hadoop Distributed File System)
- Hadoop MapReduce

MapReduce at Google

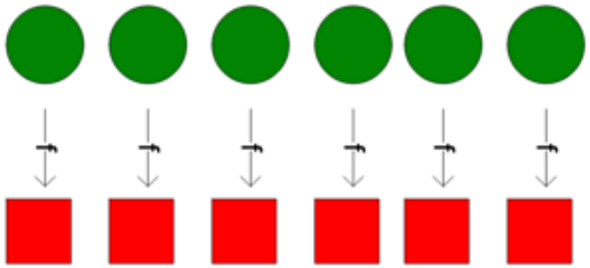
- Publication
  - The Google file system, S. Ghemawat et al. SOSP 2003.
  - MapReduce: simplified data processing on large clusters, J.Dean and S. Ghemawat. OSDI 2004.
- Main ideas
  - Data represented as key-value pairs
  - Two main operations on data: Map and Reduce
  - A distributed file system: **compute where the data are located**

Use of MapReduce at Google

- Used to implement several tasks:
  - Building the indexing system for Google Search
  - Extracting properties of web pages
  - Graph processing
  - etc.
- Google does not use MapReduce anymore
  - Moved on to more efficient technologies
    https://www.datacenterknowledge.com/archives/2014/06/25/google-dumps-mapreduce-favor-new-hyper-scale-analytics-system
    - We will study BigTable (data storage) in this course
  - The main principles are still valid

MapReduce: Map Function

MAP : TRANSFORMATION

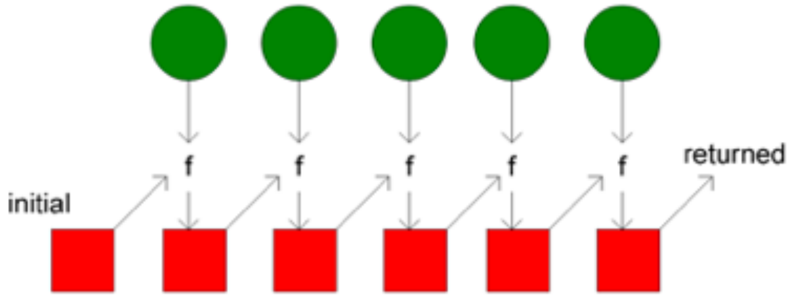


$\mathtt{map}(f,[x_0 \ldots, x_n])=[f(x_0),\ldots,f(x_n)]$

```
map (*2, [1,2,3]) = [(*2 1),(*2 2),(*2 3)] = [2,4,6]
square(x)=x*x
map(square, [1,2,3]) = [1,4,9]
```

MapReduce: Reduce Fuction

REDUCE : AGGREGATION

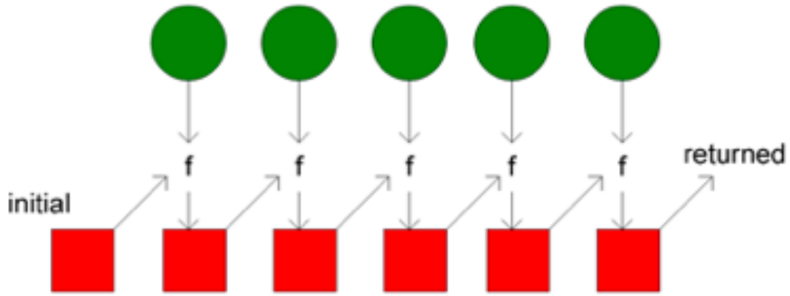



```
reduce(f,[x0 ...,xn ])= f(f(f(x0, x1), x2), x3...)

reduce(+, [2,4,6]) = 12
reduce(*, [2,4,6]) = 48
```

MapReduce: Reduce Fuction

REDUCE : AGGREGATION



reduce(f,[x_0 ...,x_n])= f(f(f(x_0, x_1), x_2), x_3...)

reduce(+, [2,4,6]) = 12
reduce(*, [2,4,6]) = 48

In MapReduce, with **<Key, Value>**
Reduce is applied to all the elements with the same key

Why MapReduce became very popular?

- Main advantages
  - Simple to program
  - Scales to large number of nodes
    - Targets scale out (share-nothing) infrastructures
  - Handles failures automatically

Simple to program

Provides a distributed computing execution framework

- Simplifies parallelization
  - Defines a programming model
  - Handles distribution of the data and the computation
- Fault tolerant
- Detects failures
- Automatically takes corrective actions

Code once (expert), benefit to all

- Limits the operations that a user can run on data
- Inspired from functional programming (MapReduce)
- Allows expressing several algorithms
  - But not all algorithms can be implemented in this way

Scales to large number of nodes

- Data parallelism
  - Running the same task on different (distributed) data pieces in parallel.
  - As opposed to Task parallelism that runs different tasks in parallel (e.g., in a pipeline)
- Move the computation instead of the data
  - The distributed file system is central to the framework
    - GFS in the case of Google
    - Heavy use of partitioning
  - The tasks are executed where the data are stored
    - Moving data is costly

Fault Tolerance

- Motivations
  - Failures are the norm rather than the exception.
    The Google file system, S. Ghemawat et al, 2003
    - In Google datacenters, jobs can be preempted at any time
    - MapReduce jobs have low priority and have high chances of being preempted
    - A 1-hour task has 5% chances of being preempted
  - Dealing with stragglers (slow machines)

Fault Tolerance

- Mechanisms
  - Data are replicated in the distributed file system
  - Results of computation are written to disk
  - Failed tasks are re-executed on other nodes
  - Tasks can be executed multiple times in parallel to deal with stragglers
    - Towards the end of a computation phase

A First MapReduce Program

WordCount

Description

- Input: A set of lines including words
  - Pairs $<$ line number, line content $>$
  - The initial keys are ignored in this example
- Output: A set of pairs $<$ word, nb of occurrences $>$

Input

- $<$ 1, "aaa bb ccc" $>$
- $<$ 2, "aaa bb" $>$

Output

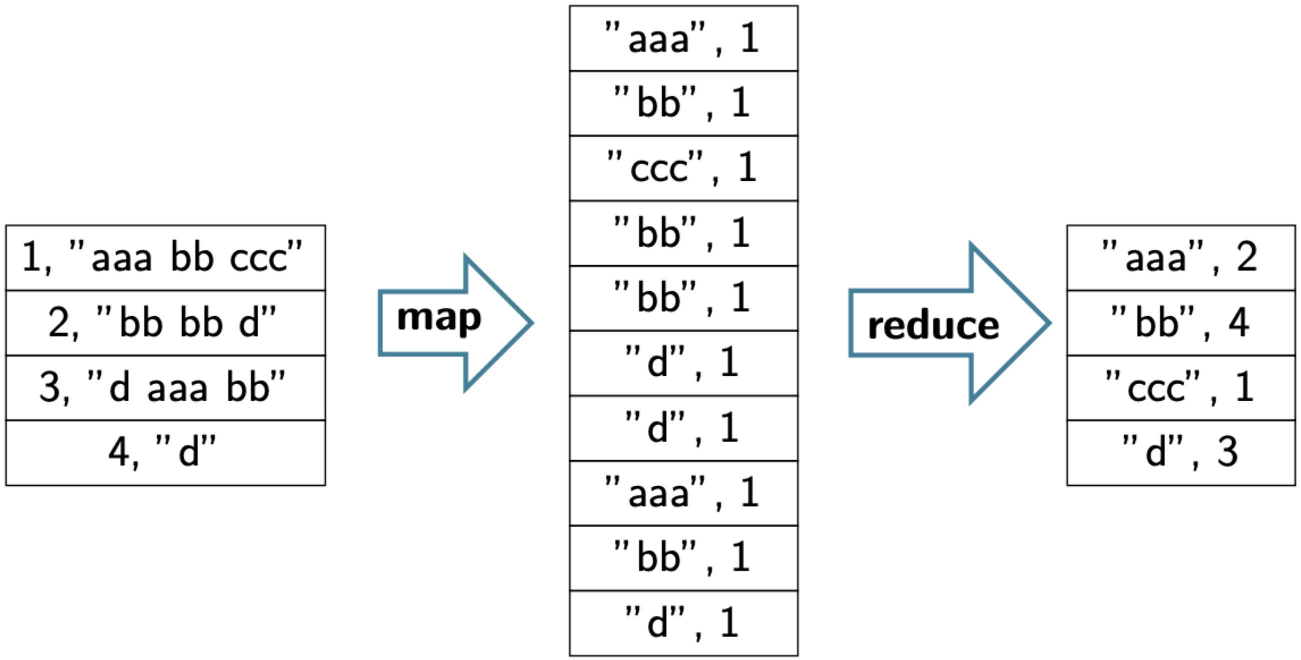
- $<$ "aaa", 2 $>$
- $<$ "bb", 2 $>$
- $<$ "ccc", 1 $>$

WordCount

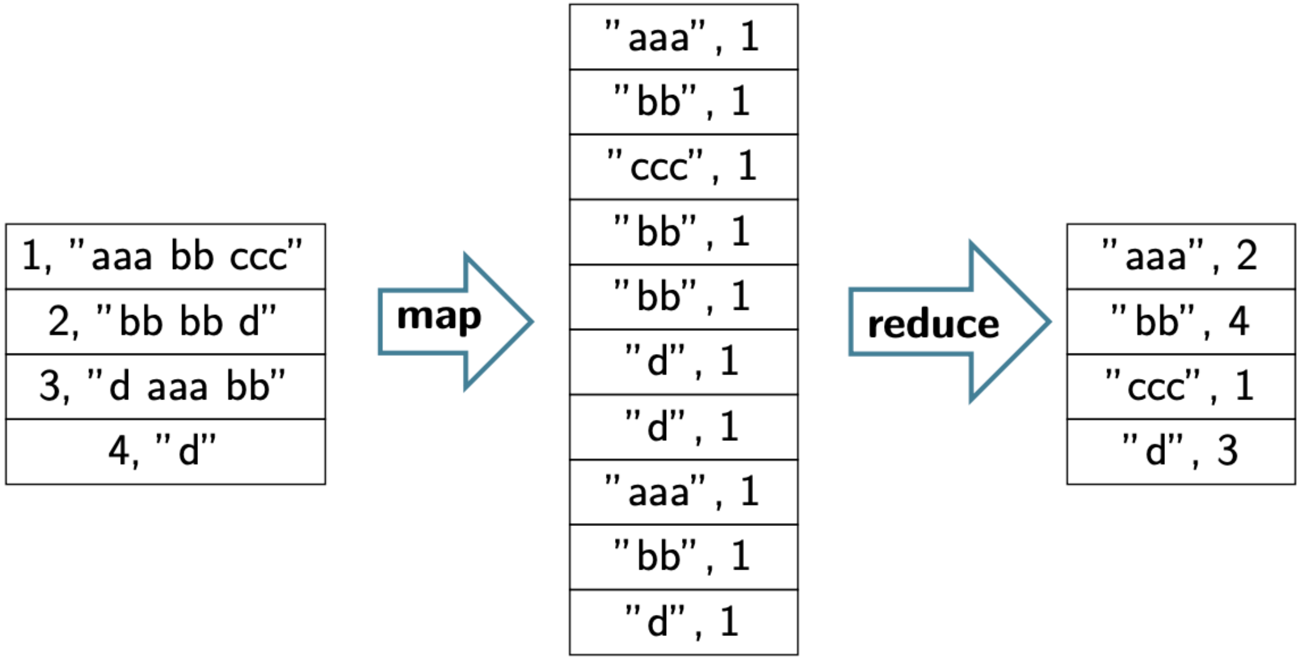
```
map(key, value):  /* pairs of {line num, content} */
  foreach word in value.split():
    emit(word, 1)


reduce(key, values):  /* {word, list nb occurences} */
  result = 0
  for value in values:
    result += value
  emit(key, result) /* -> {word, nb occurences} */
```

WordCount


1, "aaa bb ccc"
2, "bb bb d"
3, "d aaa bb"
4, "d"
map
"aaa", 1
"bb", 1
"ccc", 1
"bb", 1
"bb", 1
"d", 1
"d", 1
"aaa", 1
"bb", 1
"d", 1
reduce
"aaa", 2
"bb", 4
"ccc", 1
"d", 3

WordCount



How is it implemented in a distributed environment? (stay tuned)

Example: Web index

Description

Construct an index of the pages in which a word appears.

- Input: A set of web pages
  - Pairs $<$ URL, content of the page $>$
- Output: A set of pairs $<$ word, set of URLs $>$

Web Index

```
map(key, value):  /* pairs of {URL, page_content} */
  foreach word in value.parse():
    emit(word, key)
```

```
reduce(key, values): /* {word, URLs} */
  list=[]
  for value in values:
    list.add(value)
  emit(key, list)  /* {word, list of URLs} */
```

About batch and stream processing

- Batch processing
  - A batch processing system takes a large amount of input data, runs a job to process it, and produces some output data.
  - Offline system
    - All inputs are already available when the computation starts
  - In this lecture, we are discussing batch processing.
- Stream processing
  - A stream processing system processes data shortly after they have been received
  - Near real-time system
  - The amount of data to process is unbounded
    - Data arrives gradually over time

Agenda

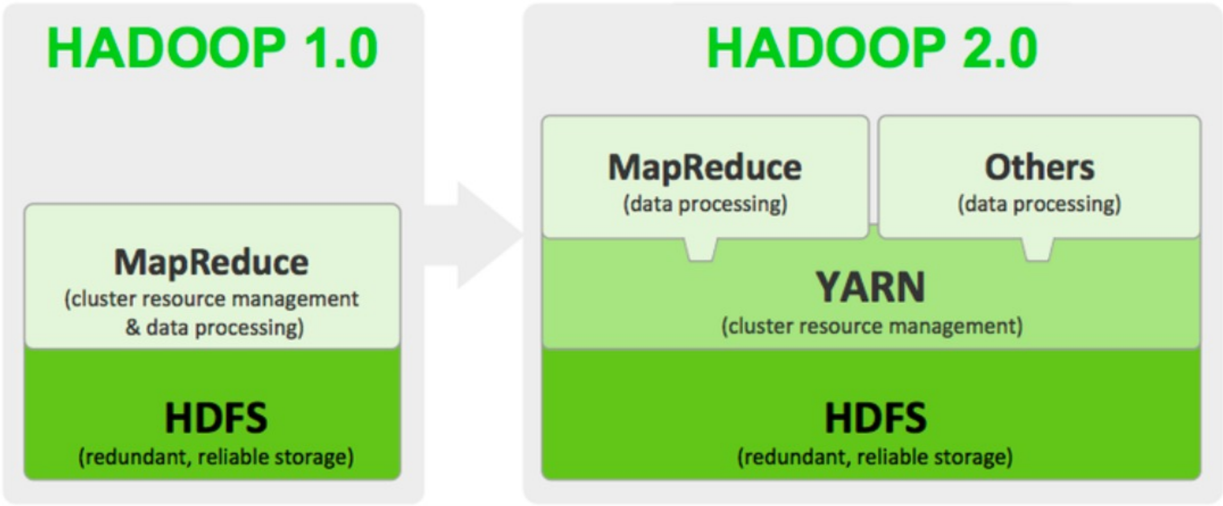
- Introduction to MapReduce
- The Hadoop Eco-System
- HDFS (Hadoop Distributed File System)
- Hadoop MapReduce

Apache Hadoop

History

- Open source implementation of a MapReduce framework
  - Implemented by people working at Yahoo!
  - Inspired from the publications of Google
  - Released in 2006
- Evolution
  - A full ecosystem
  - Used by many companies
  - Facebook Big Data stack is still inspired by (and even making use of) Hadoop
    https://www.datanami.com/2020/08/31/ how-facebook-accelerates-sql-at-extreme-scale/
    https://www.simplilearn.com/how-facebook-is-using-big-data-article

Hadoop Evolution



The Hadoop Ecosystem

- The main blocks
  - HDFS: The distributed file system
  - Yarn: The cluster resource manager
  - MapReduce: The processing engine

- Other blocks
  - Hive: Provide SQL-like query language
  - Pig: High-level language to create MapReduce applications
    - Notion of Pipeline
  - Giraph: Graph processing
  - …

Yarn: A resource management framework

- Dynamically allocates the resources of a cluster to jobs
- Allows multiple engines to run in parallel on the cluster
  - Not all jobs have to be MapReduce jobs
  - Increases resource usage
- Main components of the system
  - ResourceManager: Allocates resources to applications and monitors the available nodes
  - ApplicationMaster: Negotiates resources access for one application with the RM; Coordinates the application’s tasks execution
  - The NodeManager: Launches tasks on nodes and monitors resource usage
- Has been replaced by other frameworks (Mesos, Kubernetes)

Agenda

- Introduction to MapReduce
- The Hadoop Eco-System
- HDFS (Hadoop Distributed File System)
- Hadoop MapReduce

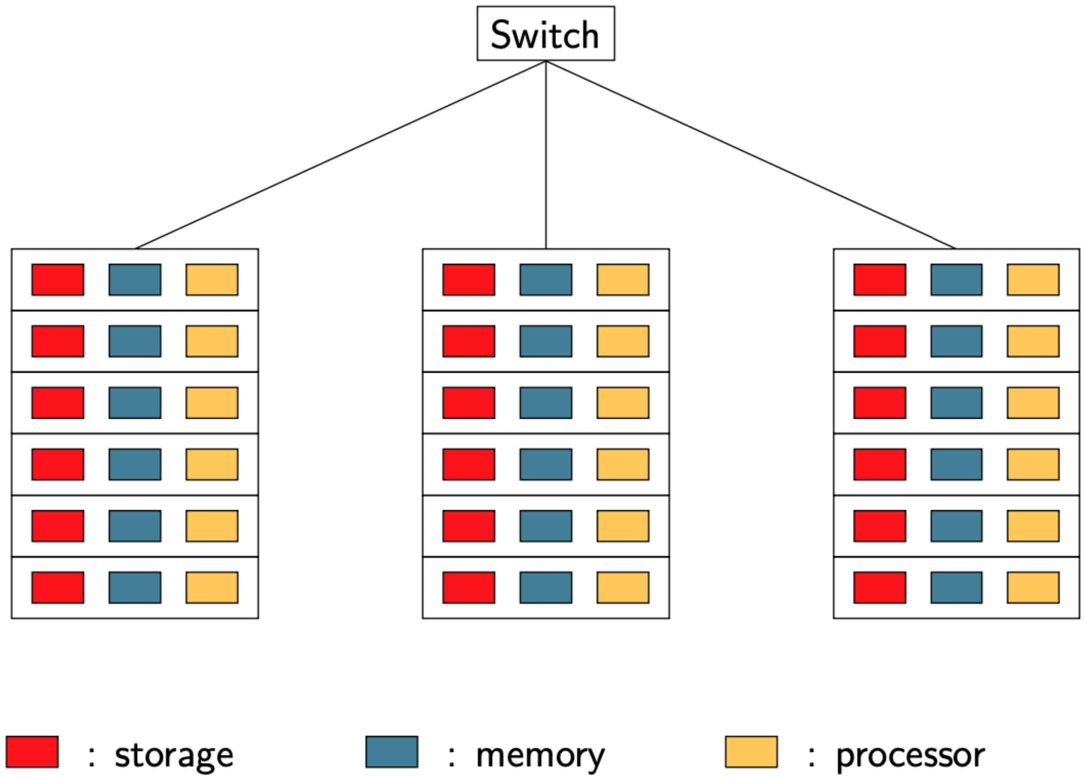
HDFS

- Purpose
  - Store and provide access to large datasets in a share-nothing infrastructure

- Challenges
  - Scalability
  - Fault tolerance

Target infrastructure (reminder)

Cluster of commodity machines



Main principles

Achieving scalability and fault tolerance

Main assumptions

- Storing large datasets
  - Provide large aggregated bandwidth
  - Allow storing large amount of files (millions)
- Batch processing (i.e., simple access patterns)
  - The file system is not POSIX-compliant
  - Assumes sequential read and writes (no random accesses)
    - Write-once-read-many file accesses
    - Supported write operations: Append and Truncate
    - Stream reading
- Optimized for throughput (not latency)

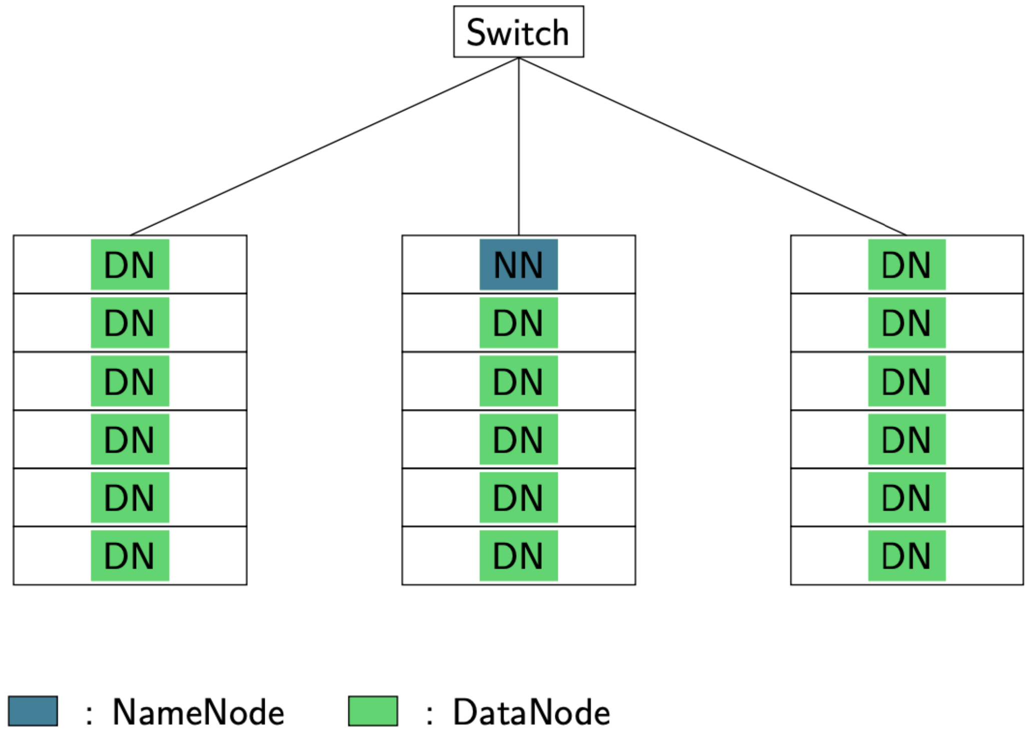
Main principles

- Partitioning
  - Files are partitioned into blocks
  - Blocks are distributed over the nodes of the system
  - Default block size in recent versions: 128MB
- Replication
  - Multiple replicas of each block are created
  - Replication is topology aware (rack awareness)
  - Default replication degree is 3

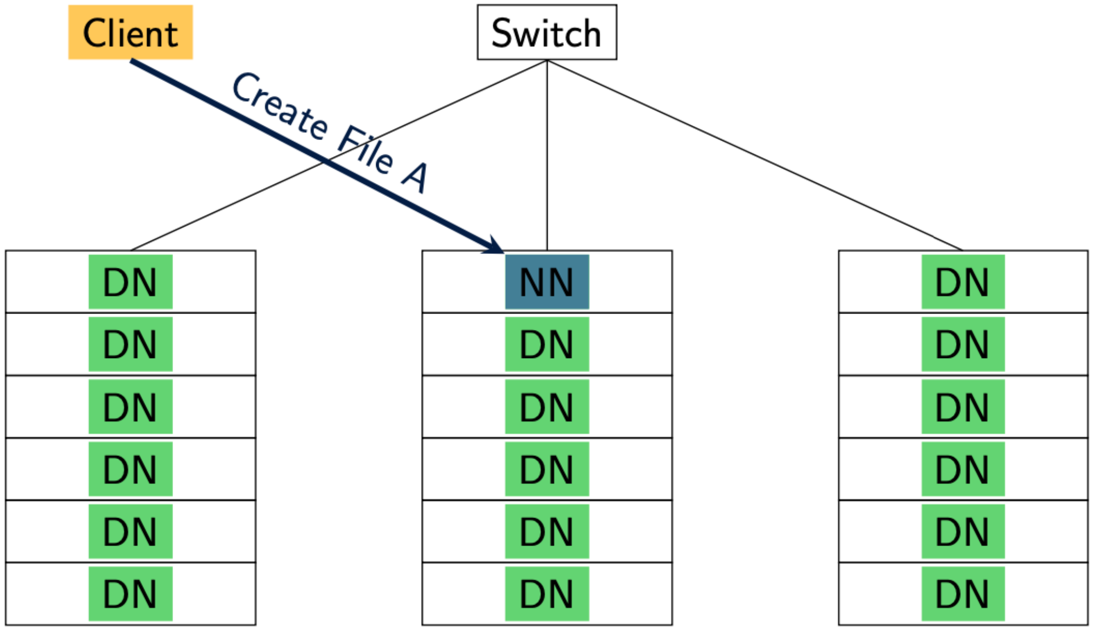
A Master-Slave Architecture

- A set of DataNodes
  - One daemon per node in the system
  - A network service allowing to access the file blocks stored on that node
    - It is responsible for serving read and write requests
- One NameNode
  - Keeps track of where blocks are stored
  - Monitors the DataNodes
  - Entry point for clients

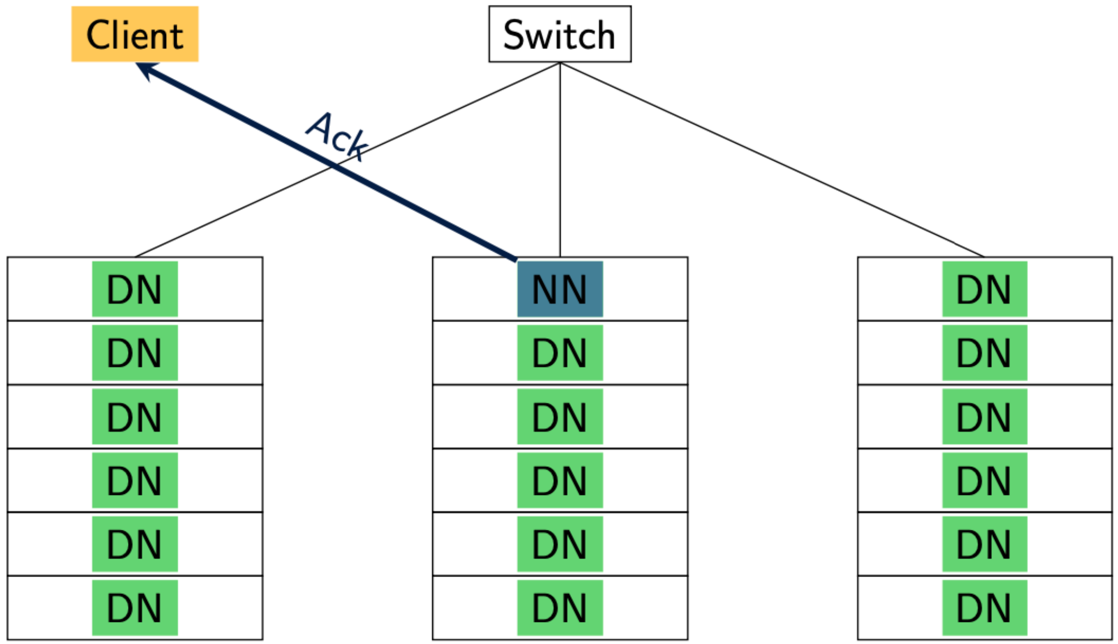
HDFS architecture



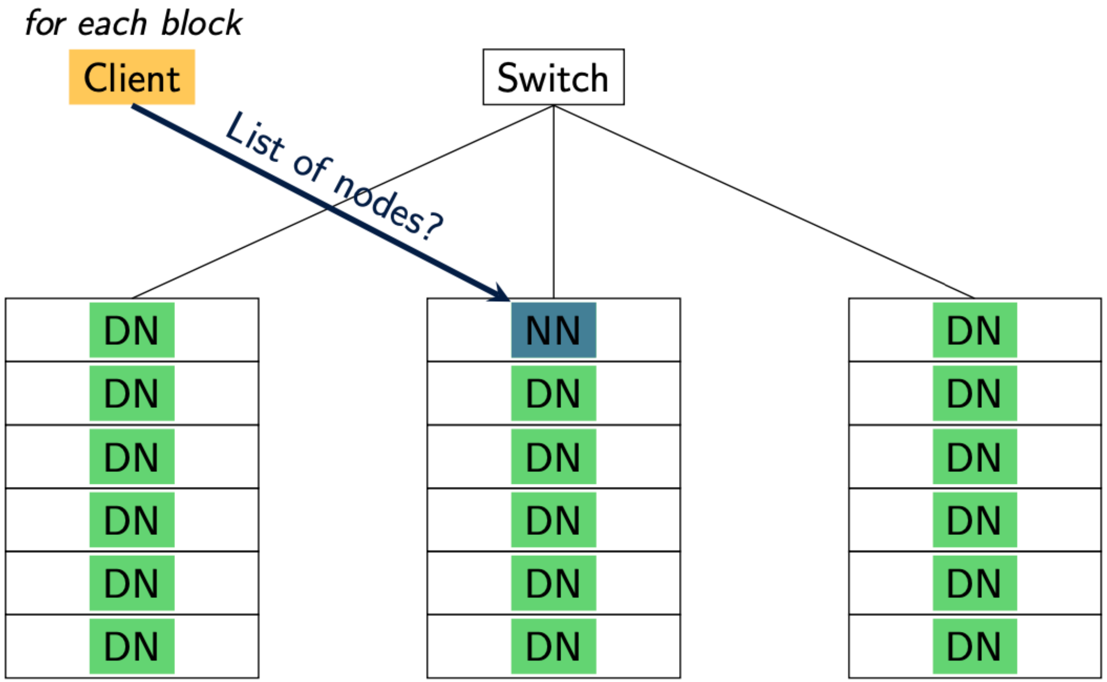
Write a File


Client
Switch
Create File A
DN
NN
DN
DN
DN
DN
DN
DN
DN
DN
DN
DN
DN
DN
DN
DN
DN
DN
DN

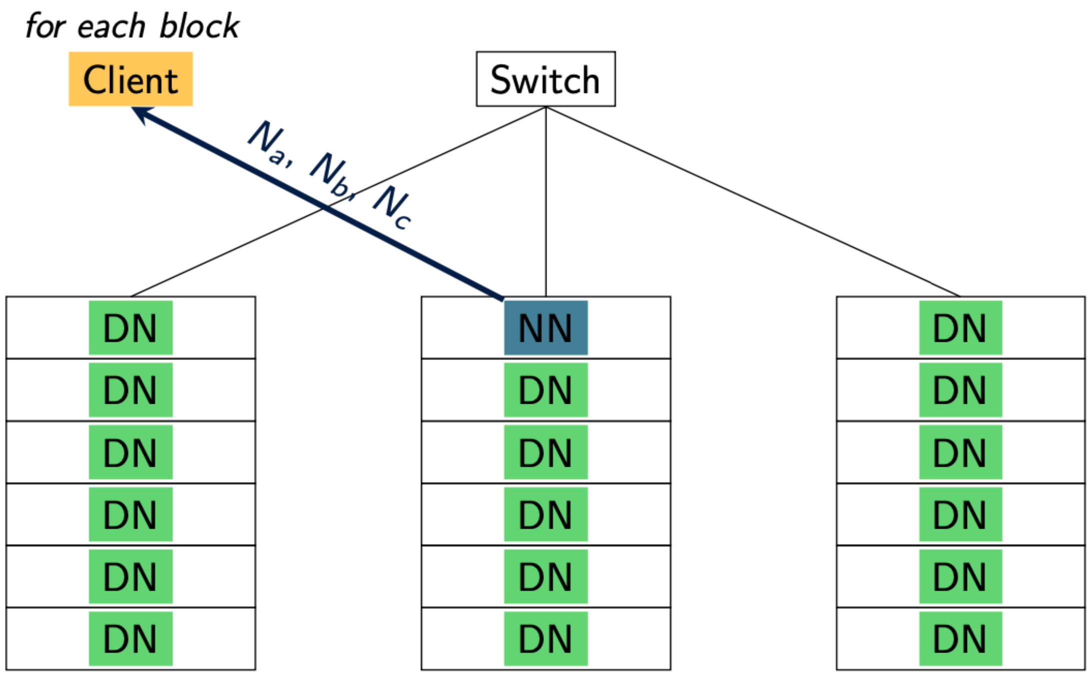
Write a File



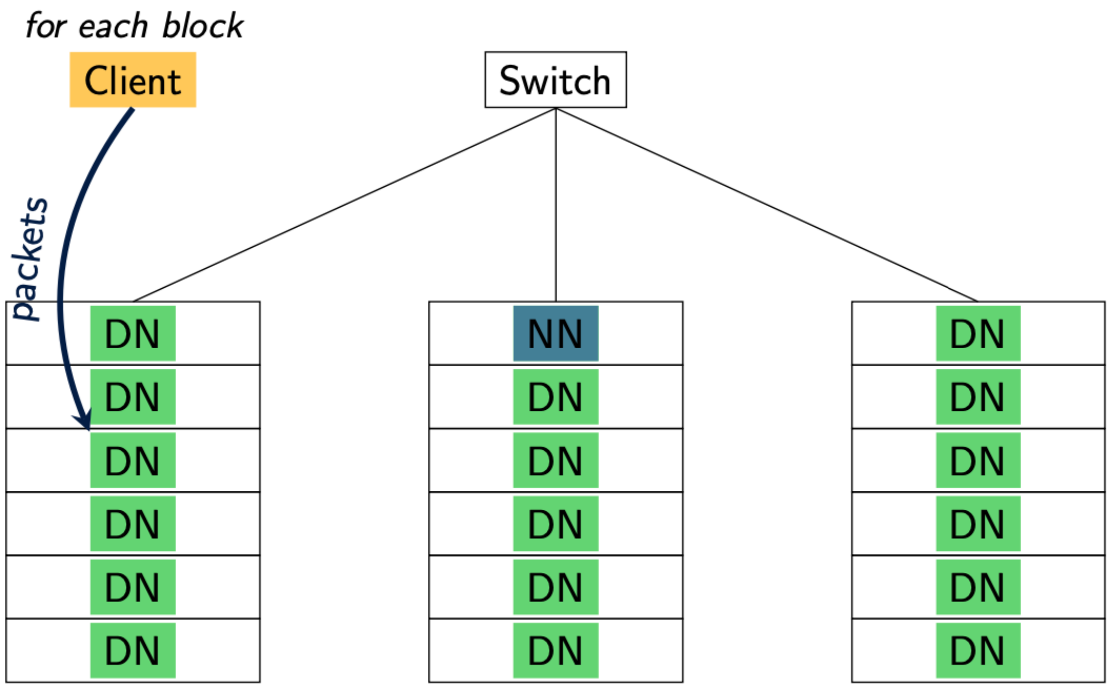
Write a File


for each block
Client
Switch
List of nodes?
DN
DN
DN
DN
DN
DN
NN
DN
DN
DN
DN
DN
DN
DN
DN
DN
DN
DN

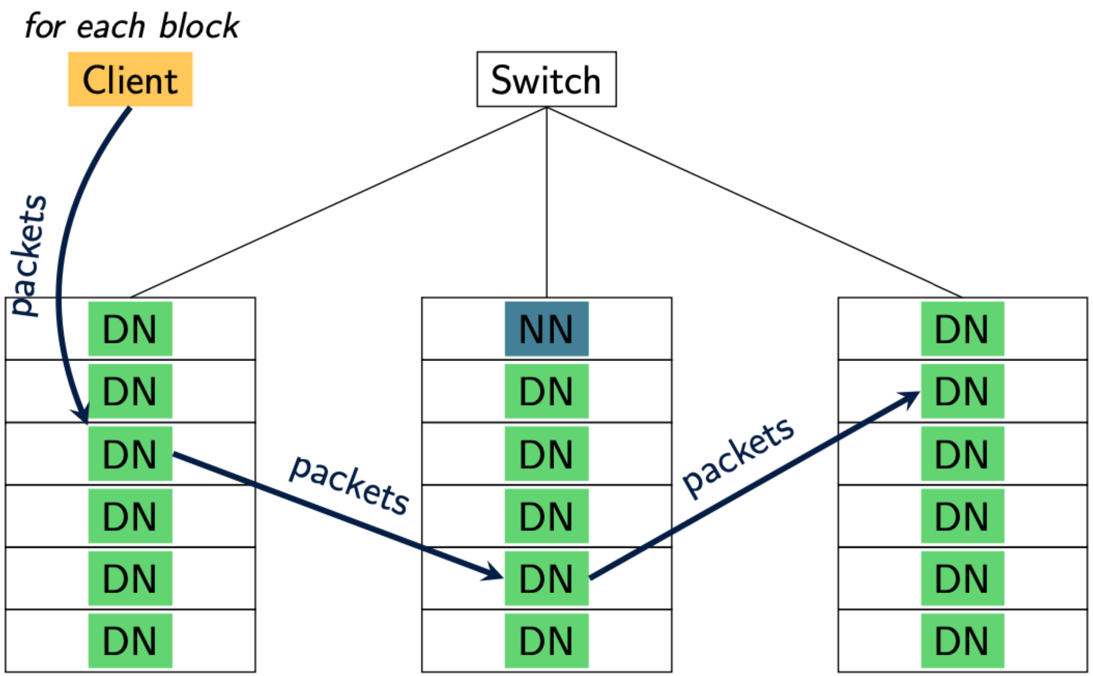
Write a File



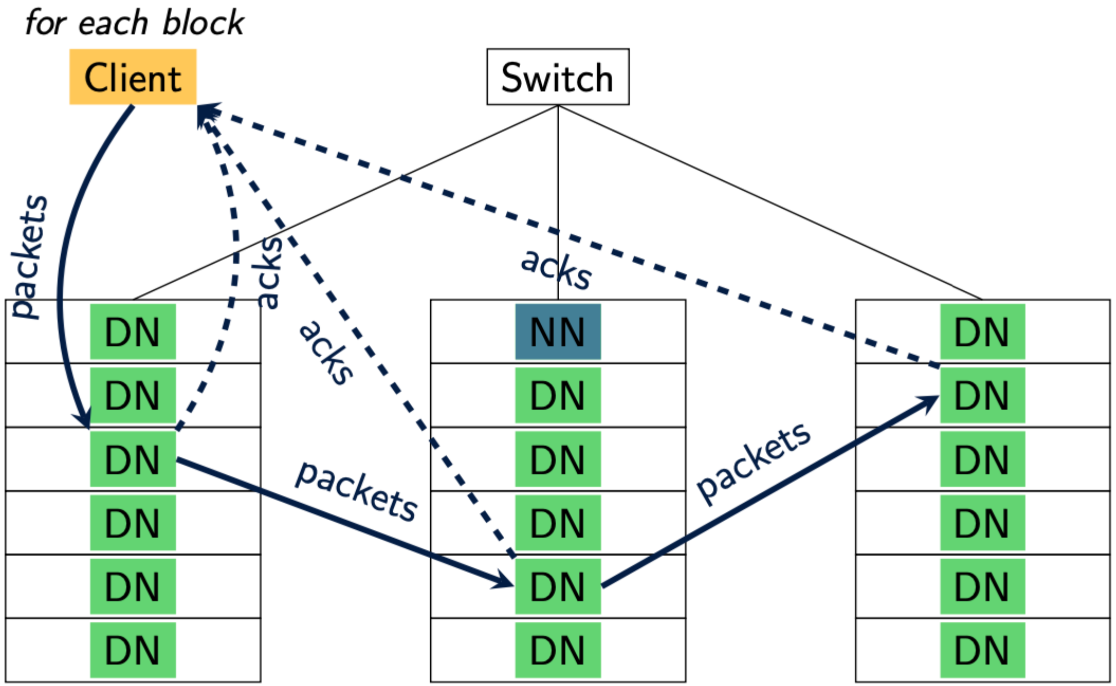
Write a File


for each block
Client
Switch
packets
DN
DN
DN
DN
DN
DN
NN
DN
DN
DN
DN
DN
DN
DN
DN
DN
DN
DN

Write a File

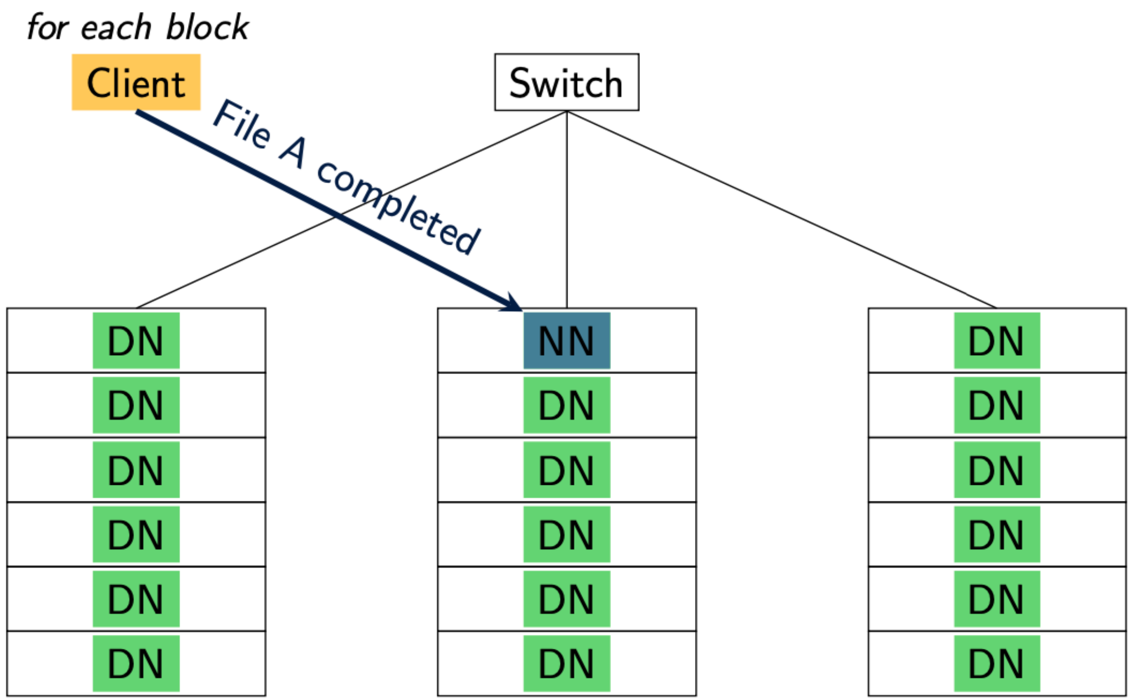

for each block
Client
Switch
packets
DN
DN
DN
DN
DN
DN
NN
DN
DN
DN
DN
DN
DN
DN
DN
DN
DN
DN
packets
packets

Write a File



for each block
Client
Switch
packets
acks
acks
acks
packets
packets
NN
DN

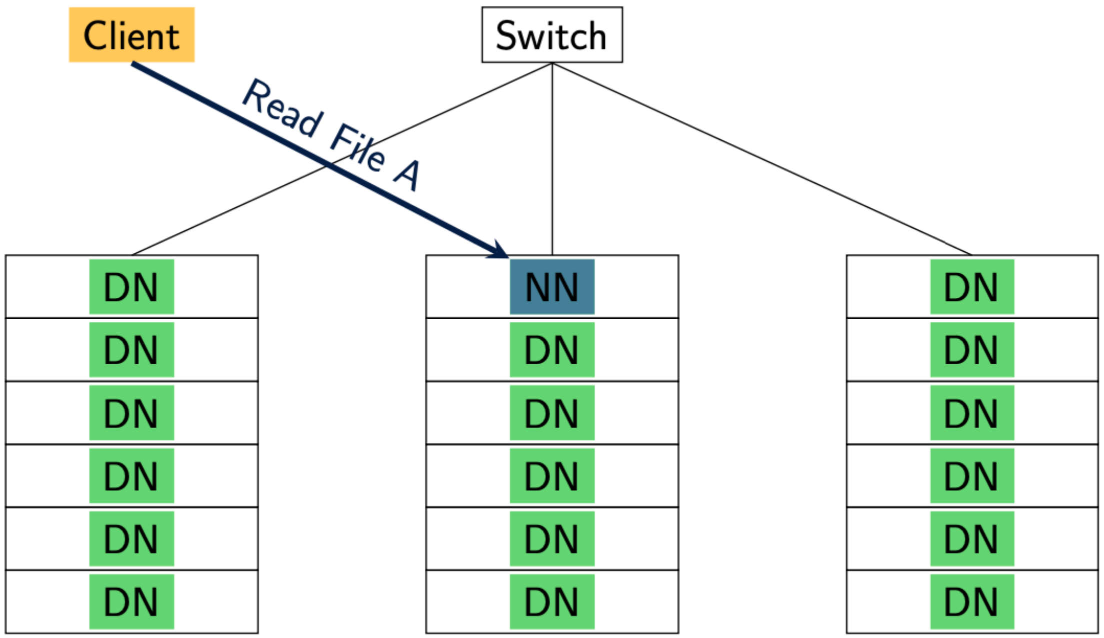
Write a File



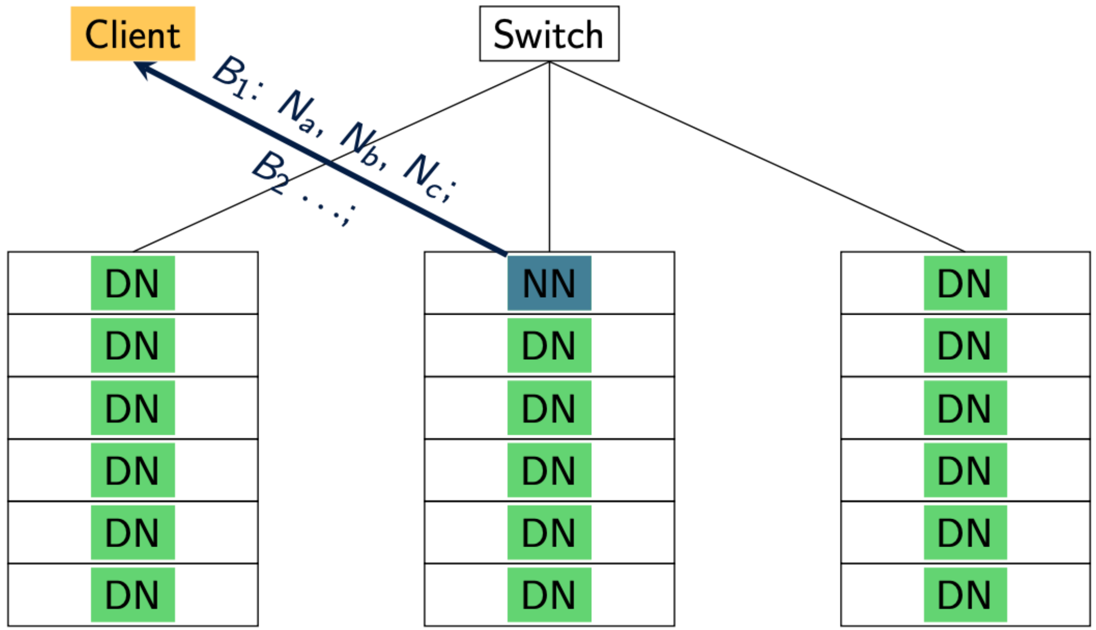
Write a File: Summary

1. The client contacts the NameNode to request new file creation
   - The NameNode makes all required checks (Permissions, file does not exists, etc.)
2. The NameNode allows the client to write the file
3. The client splits the data to be written into blocks
   - For each block, it asks the NameNode for a list of destination nodes
   - The returned list is sorted in increasing distance from the client
4. Each block is written in a pipeline
   - The client picks the closest node to write the block
   - The DataNode receives the packets (portions) and forwards them to the next DataNode in the list
5. Once all blocks have been created with a sufficient replication degree, the client acknowledges file creation completion to the name node.
6. The NameNode flushes information about the file to disk

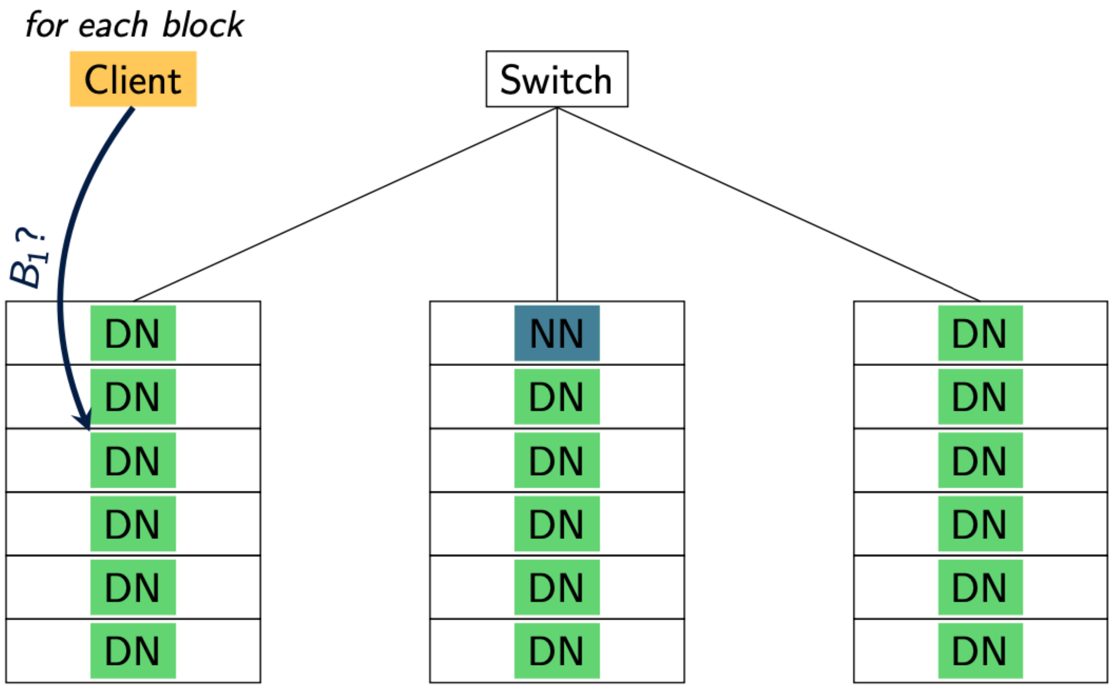
Read a File



Read a File

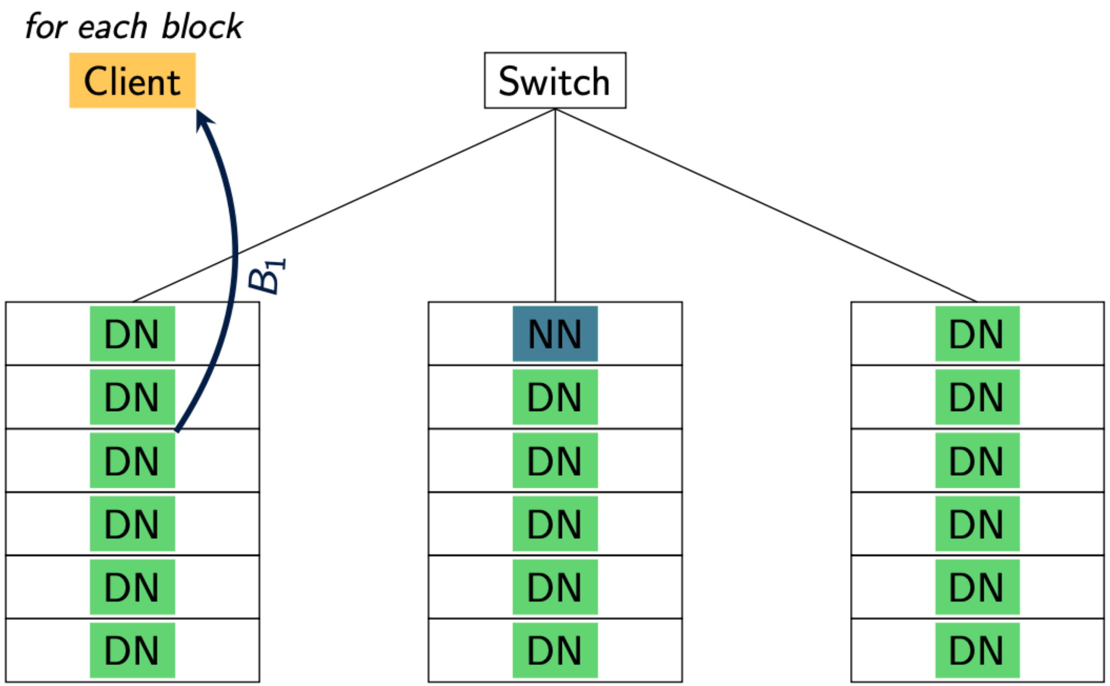


Read a File

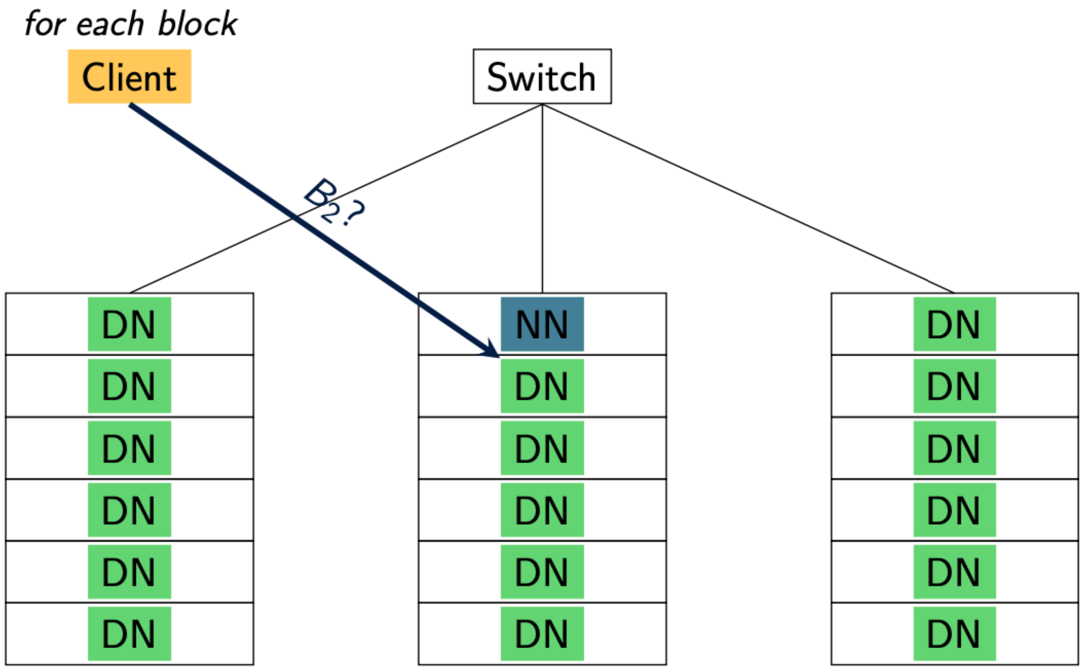


for each block
Client
Switch
B_1 ?
DN
DN
DN
DN
DN
DN
NN
DN
DN
DN
DN
DN
DN
DN
DN
DN
DN
DN

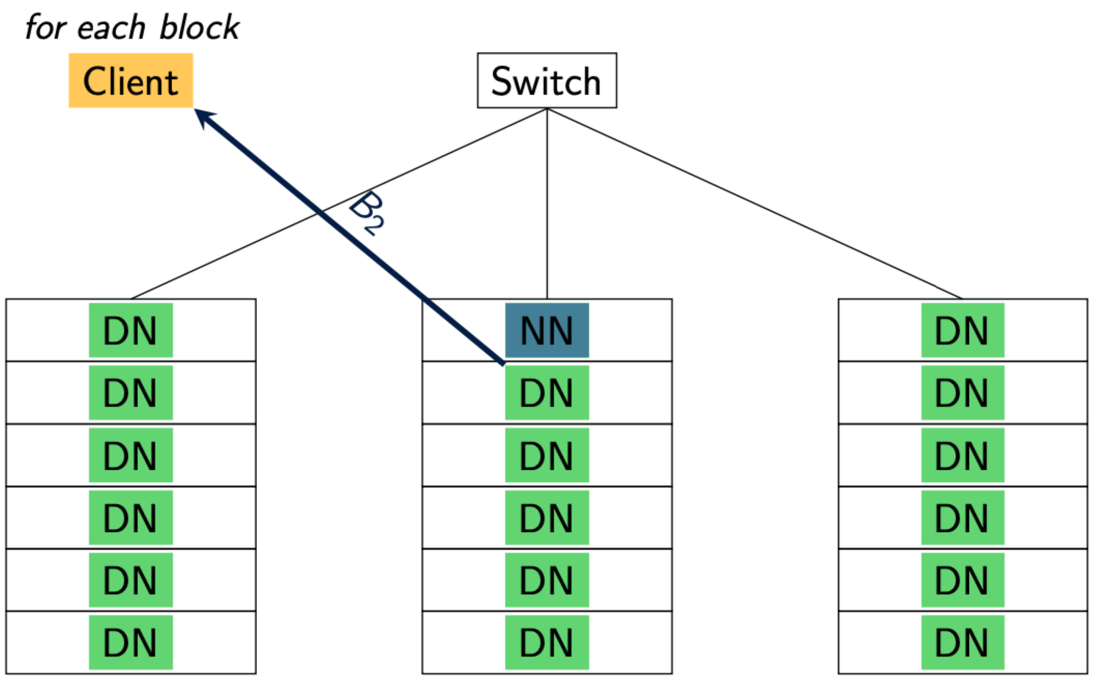
Read a File


for each block
Client
Switch
B_1
DN
DN
DN
DN
DN
DN
NN
DN
DN
DN
DN
DN
DN
DN
DN
DN
DN
DN

Read a File


for each block
Client
Switch
B_2?
NN
DN
DN
DN
DN
DN
DN
DN
DN
DN
DN
DN
DN
DN
DN
DN
DN
DN

Read a File



Read a File: Summary

1. The client contacts the NameNode to have info about a file
2. The NameNode returns the list of all blocks
   - For each block, it provides a list of nodes hosting the block
   - The list is sorted according to the distance from the client
3. The client can start reading the blocks sequentially in order
   - By default, contacts the closest DataNode
   - If the node is down, contacts the next one in the list

Supported File Formats

- Text/CSV files
- JSON records
- Sequence files (binary key-value pairs)
  - Can be used to store photos, videos, etc
- Defining custom formats
  - Avro
  - Parquet
  - ORC

Agenda

- Introduction to MapReduce
- The Hadoop Eco-System
- HDFS (Hadoop Distributed File System)
- Hadoop MapReduce

In a Nutshell

- A distributed MapReduce framework
  - Map and Reduce tasks are distributed over the nodes of the system
  - Runs on top of HDFS
    - Move the computation instead of the data
  - Fault tolerant

- 2 main primitives
  - Map (transformation)
  - Reduce (aggregation)

In a Nutshell

- Key/Value pairs
  - MapReduce manipulate sets of Key/Value pairs
  - Keys and values can be of any types

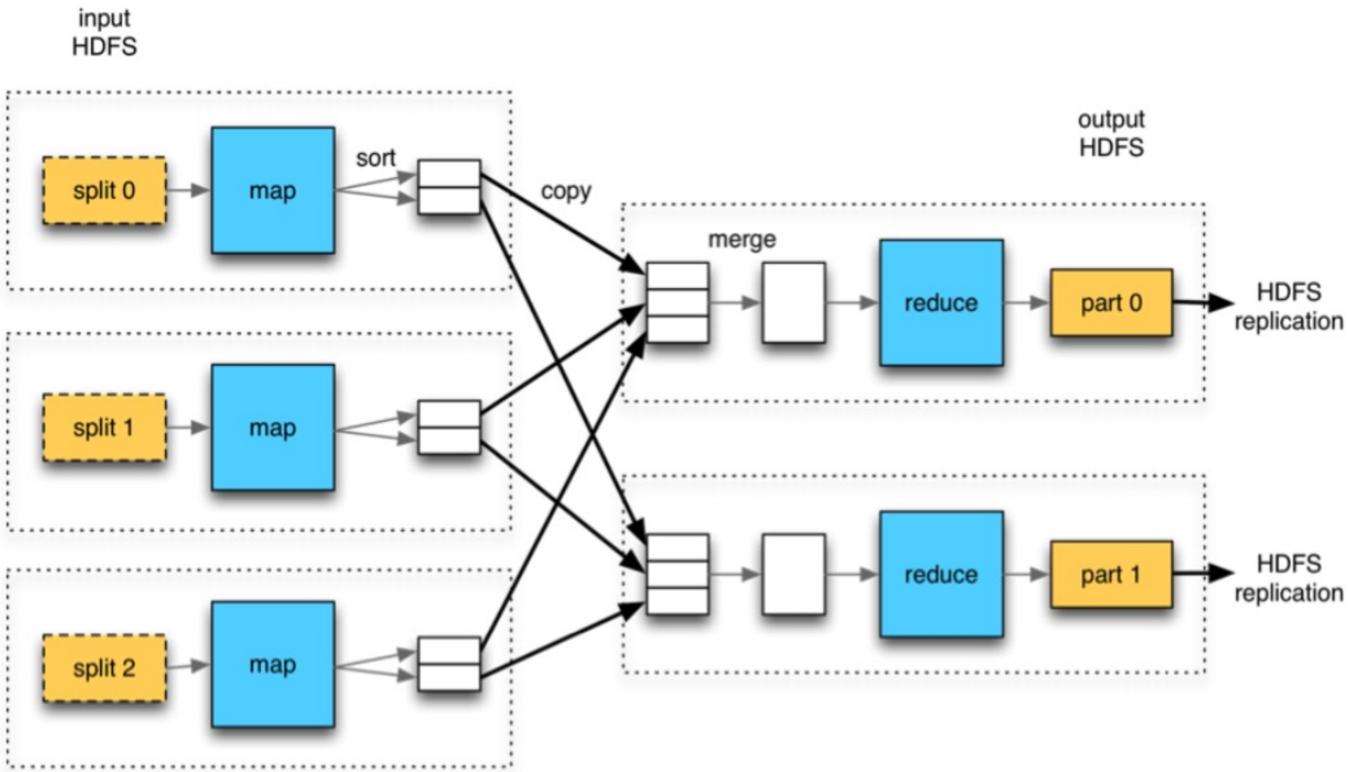
- Functions to apply
  - The user defines the functions to apply
  - In Map, the function is applied independently to each pair
  - In Reduce, the function is applied to all values with the same key

MapReduce operations

- About the Map operation
  - A given input pair may map to zero, one, or many output pairs
  - Output pairs need not be of the same type as input pairs
- About the Reduce operation
  - Applies operation to all pairs with the same key
  - 3 steps:
    - Shuffle and Sort: Groups and merges the output of mappers by key
    - Reduce: Applies the reduce operation to the new key/value pairs

Distributed Execution

Figure from https://www.supinfo.com/articles/single/2807-introduction-to-the-mapreduce-life-cycle



Distributed Execution

The Details

- Map tasks
  - As many as the number of blocks to process
  - Executed on a node hosting a block (when possible)
  - Data read from HDFS

- Reduce tasks
  - Number selected by the programmer
  - Key-value pairs are distributed over the reducers using a hash of the key
  - The output is stored in HDFS

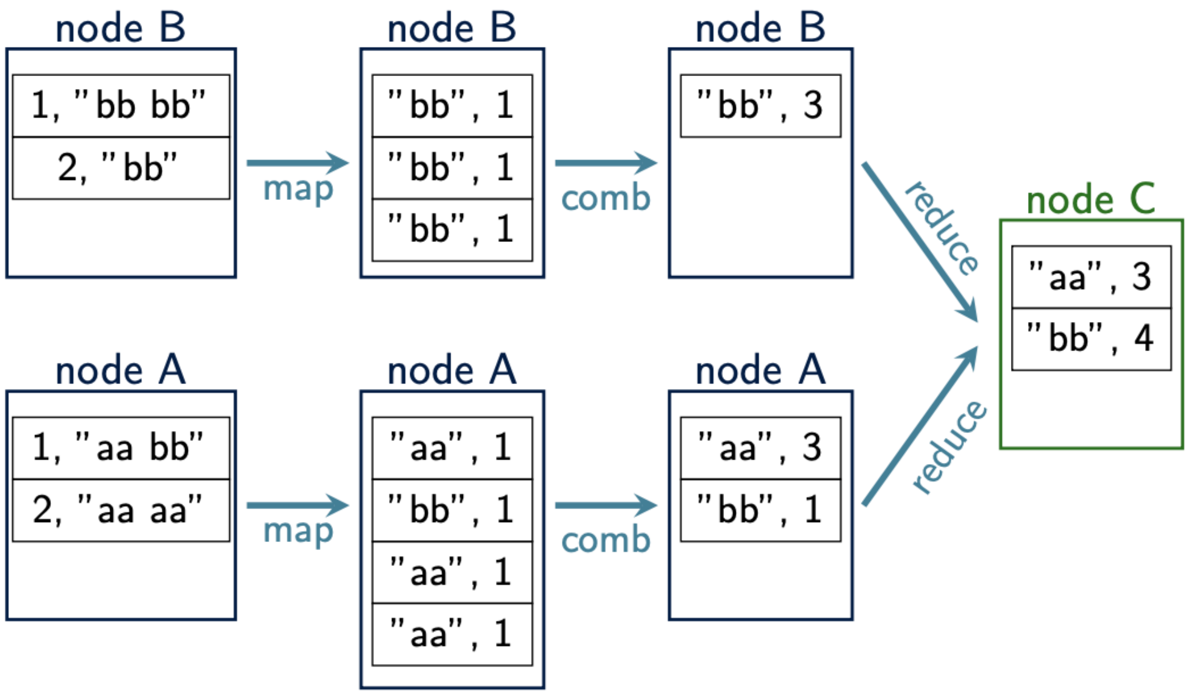
Data Management

Moving data from the Map to the Reduce tasks

1. Output of map tasks are partitioned. The result is stored locally
   - As many partitions are created as the number of reducers
   - By default, a partitioning function based on the hash of the key is used
   - The user can specify its own partitioning function
2. The reducers fetch the data from the map tasks
   - They connect to the map nodes to fetch data (shuffle)
   - This can start as soon as some map tasks finish (customizable)
3. The reducers sort the data by key (sort)
   - Can start only when all map tasks are finished

Reducing the amount of data transferred

- Combiner
  - User-defined function for local aggregation on the map tasks
  - Applied after the partitioning function



About More Complex Programs
Workflows

- Sequence of Map and Reduce operations
  - The output of one job is the input of the next job
  - Example: Getting the word that occurs to most often in a text
    - Job 1: counting the number of occurrence of each word
    - Job 2: Find the word with the highest count
- Implementation
  - No specific support in Hadoop
  - Data simply go through HDFS

References

- Mandatory reading (preparation for next course)
  - Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing, M. Zaharia et al. NSDI, 2012.
- Suggested reading
  - Chapter 10 of Designing Data-Intensive Applications by Martin Kleppmann
  - HDFS Cartoon: https://wiki.scc.kit.edu/gridkaschool/upload/1/18/Hdfs-cartoon.pdf
  - MapReduce illustration: https://words.sdsc.edu/words-data-science/mapreduce