CSE 4510/5310
BIG DATA

Instructor: Fitzroy Nembhard, Ph.D.

Week 11

HDFS

FLORIDA TECH
FLORIDA'S STEM UNIVERSITY®

Distribution

- All slides included in this class are for the exclusive use of students and instructors associated with Management and Processing of Big Data(CSE 4510/5310) at the Florida Institute of Technology
- Redistribution of the slides is not permitted without the written consent of the author.

Goals

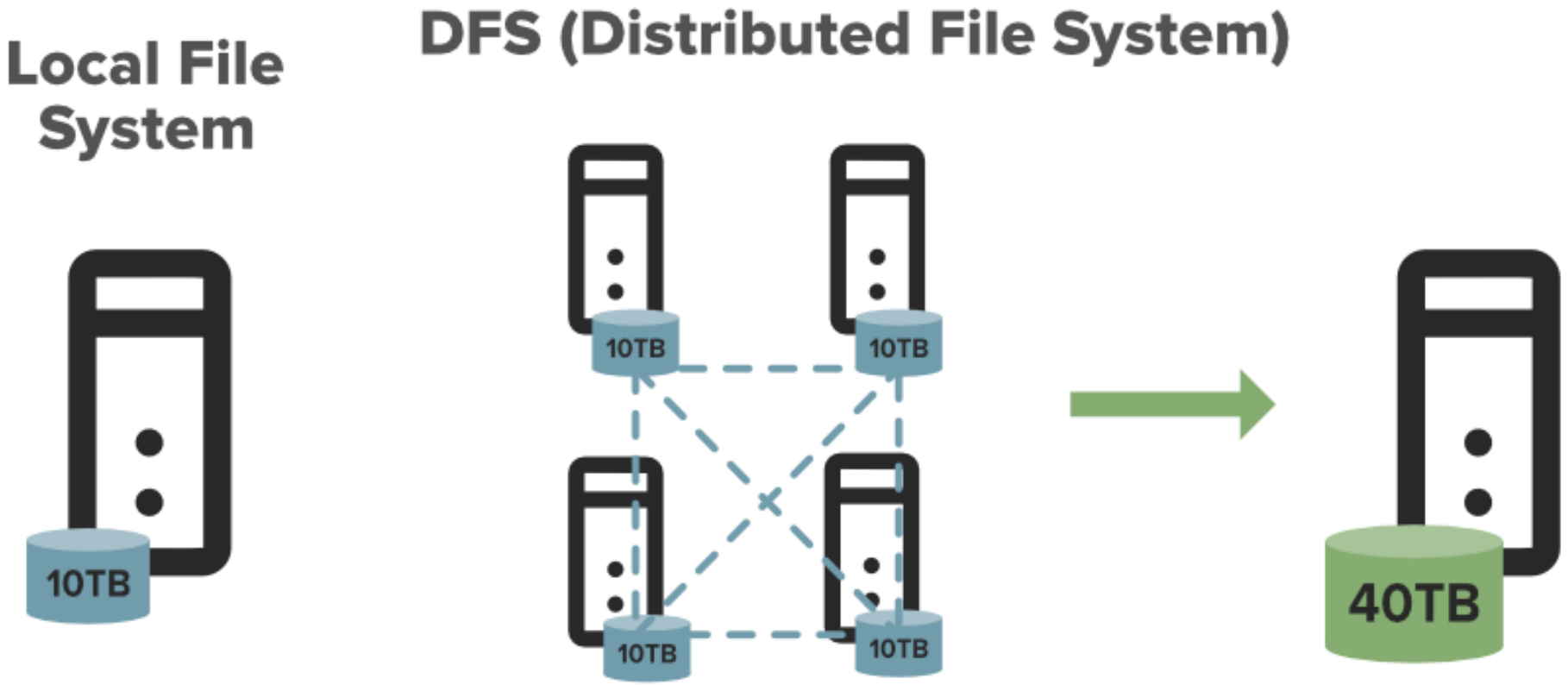
- To discuss Hadoop Distributed File System
  - HDFS design and architecture
  - Fault tolerance in HDFS
  - Create (Write) a file
  - Stream reading
  - Structured reading
  - Special Features
  - Setting up a Hadoop/HDFS Pseudo-cluster
  - Command-line interface (HDFS Shell)
  - Java API/Python API

Hadoop Components

HDFS	Distributed file system
MapReduce	Distributed computation framework
HBase	Column-oriented table service
Pig	Dataflow language and parallel execution framework
Hive	Data warehouse infrastructure
ZooKeeper	Distributed coordination service
Chukwa	System for collecting management data
Avro	Data serialization system

11.0 DISTRIBUTED FILE SYSTEMS

Distributed File Systems



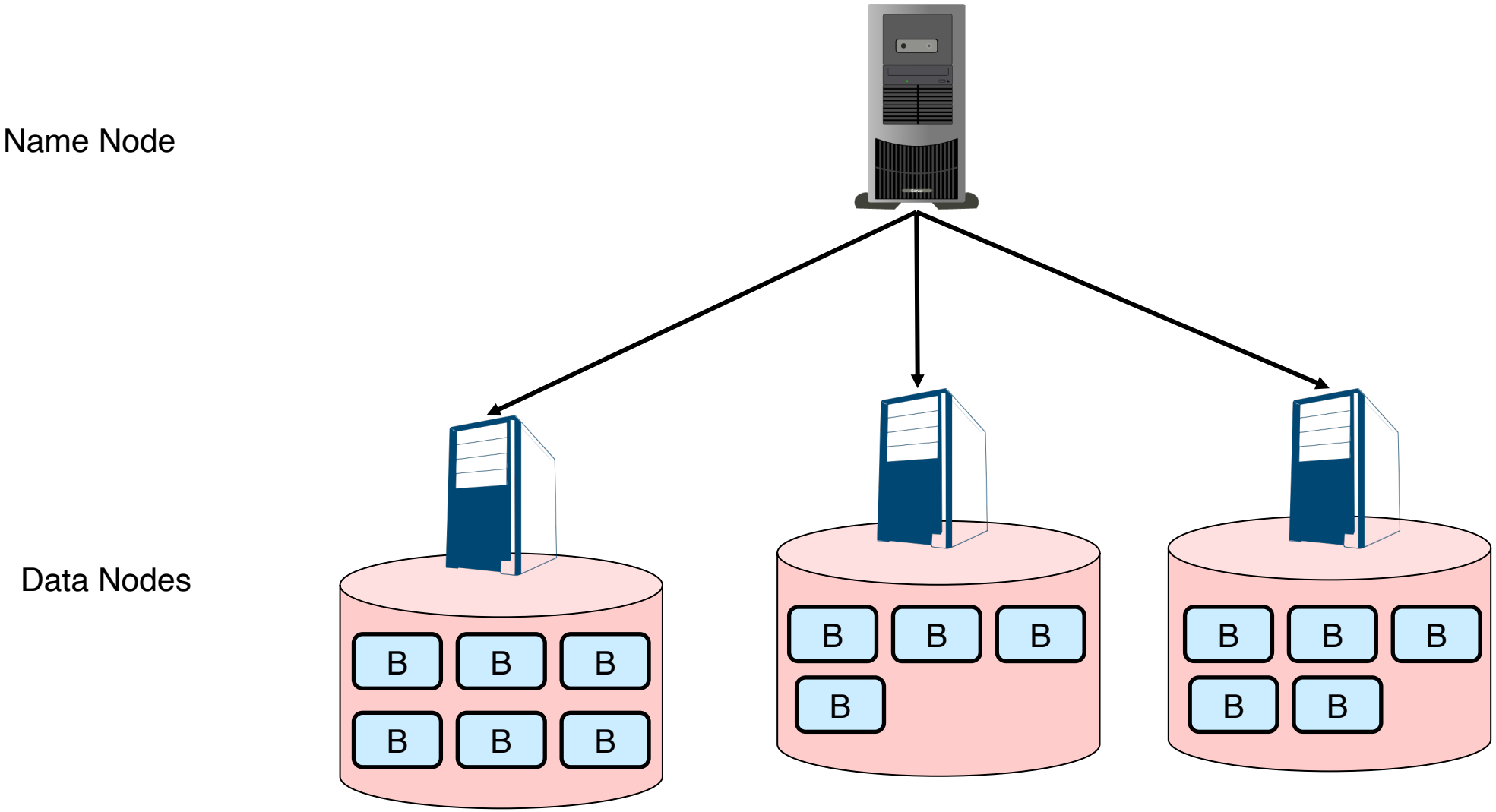
Source: Databricks.com

11.1 HDFS DESIGN & ARCHITECTURE

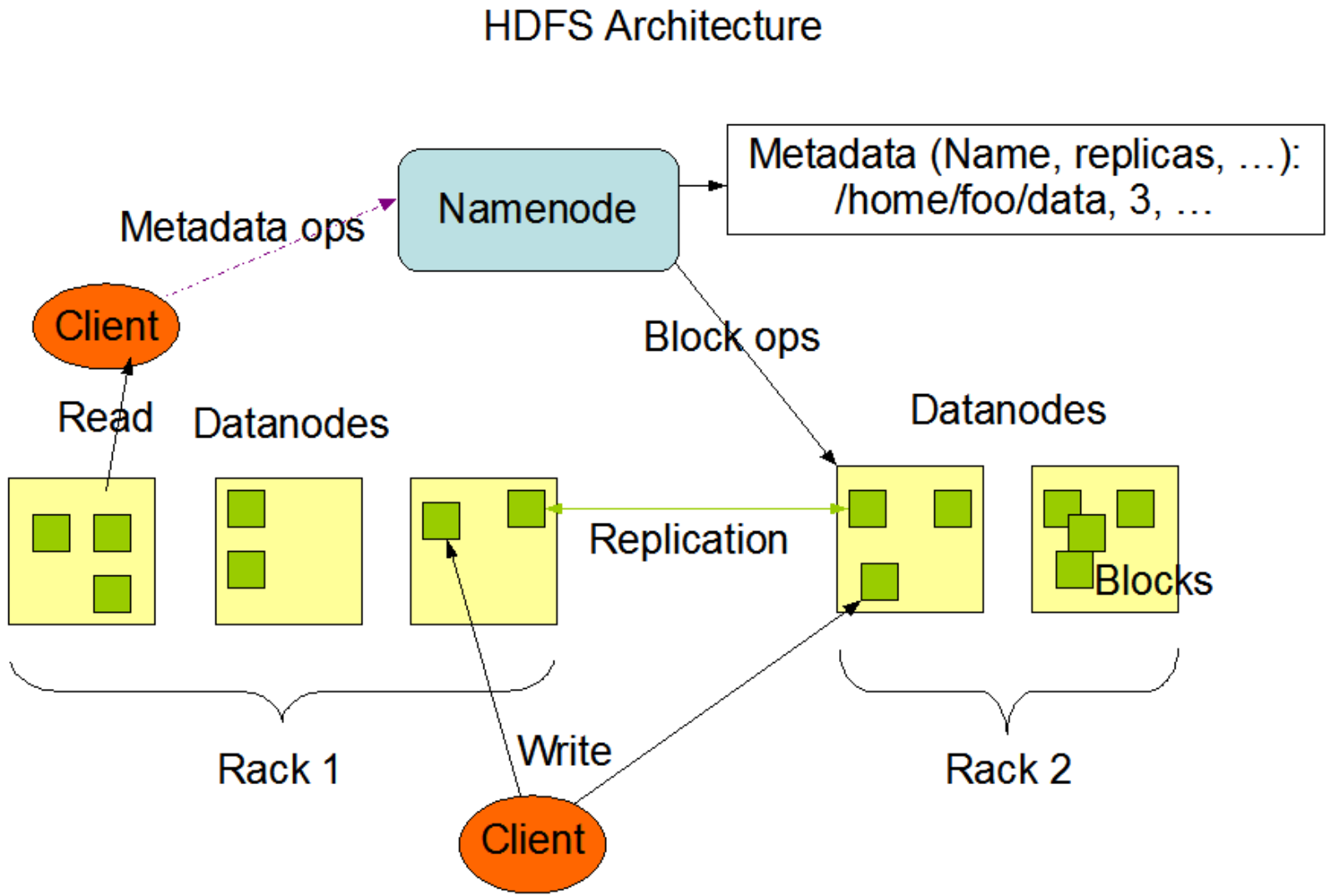
HDFS

- A distributed file system
- Built on the architecture of Google File System (GFS)
- Shares a similar architecture to many other common distributed storage engines such as Amazon S3 and Microsoft Azure
- HDFS is a stand-alone storage engine and can be used in isolation of the query processing engine
- Even if you do not use Hadoop MapReduce, you will probably still use HDFS

HDFS Architecture (Overview)



HDFS Architecture (Detailed View)

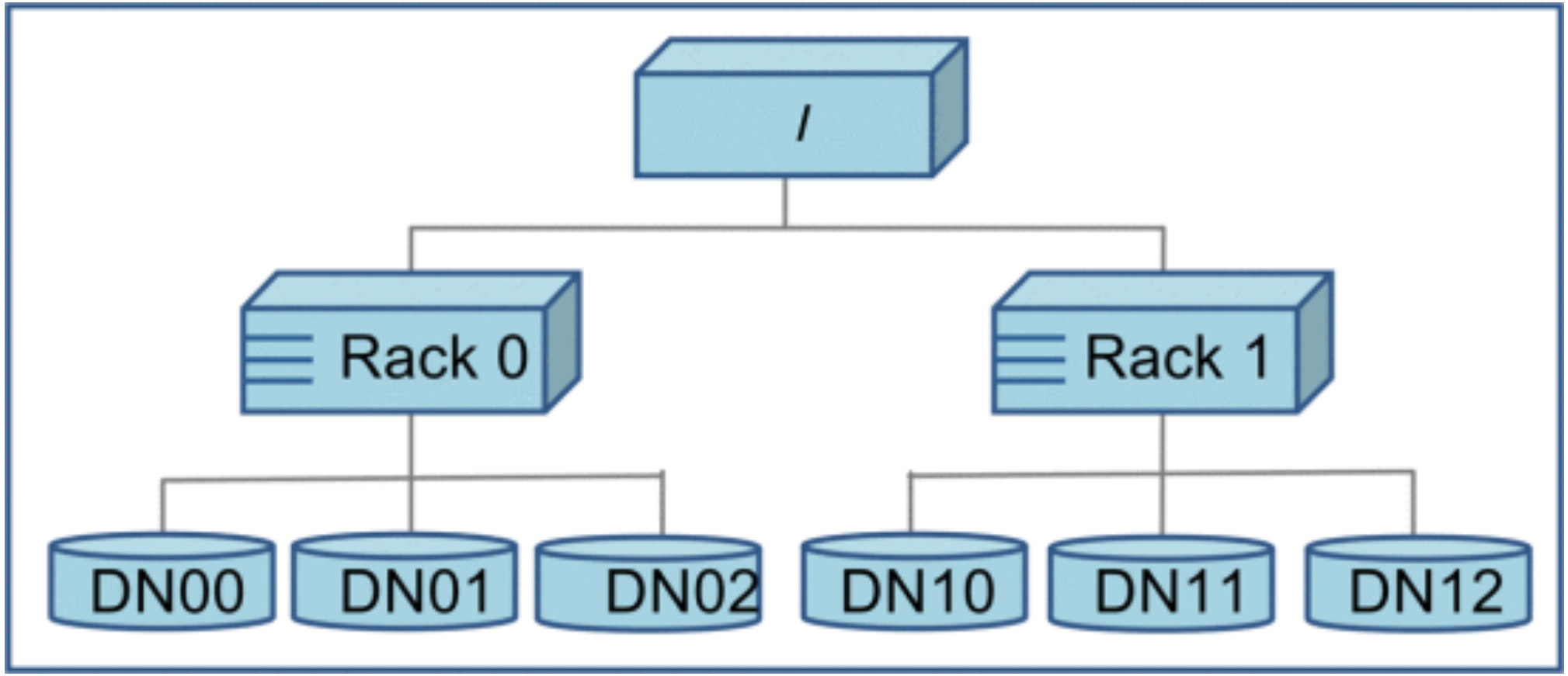


Physical Cluster Layout

Fully Qualified File Path

- `hdfs://masternode:9000/path/to/file`
- hdfs: the file system scheme.
  - Other protocols/schemes include ftp, s3, ...
- masternode: the name or IP address of the node that hosts the master of the file system
- 9000: the port on which the master node is listening
- /path/to/file: the absolute path of the file

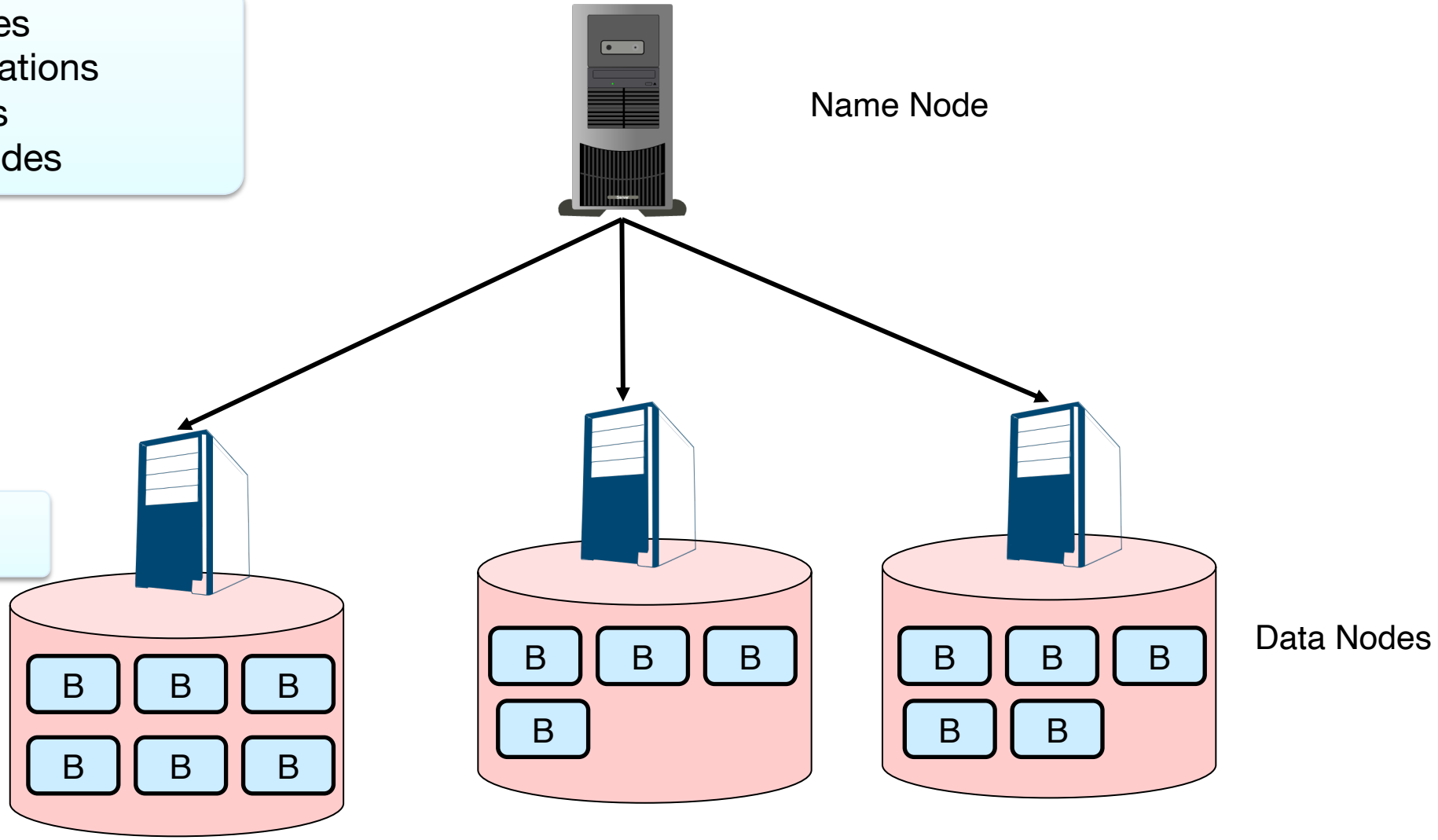
Example Cluster Topology



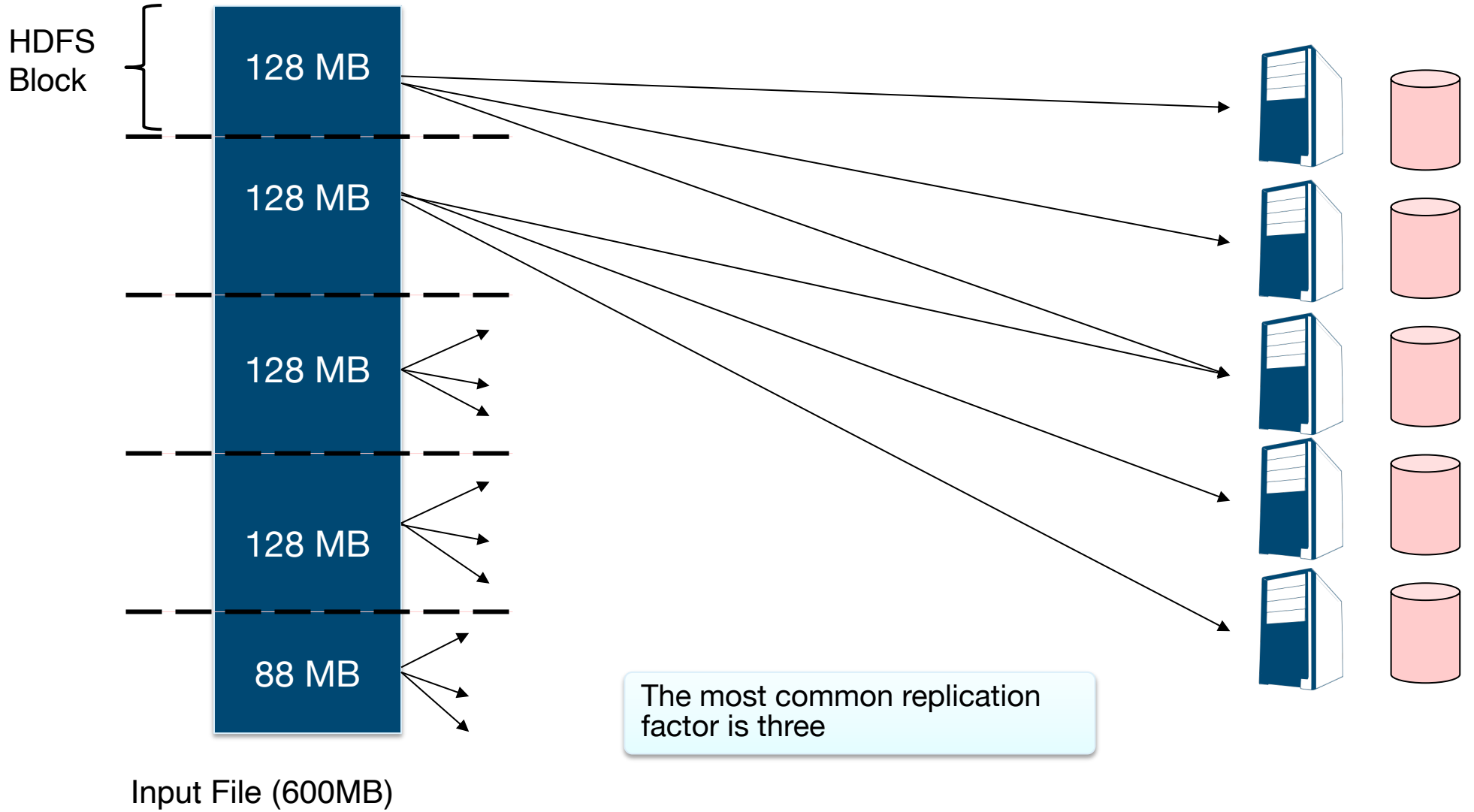
What is Where?

- File and directory names
- Block ordering and locations
- Capacity of data nodes
- Architecture of data nodes

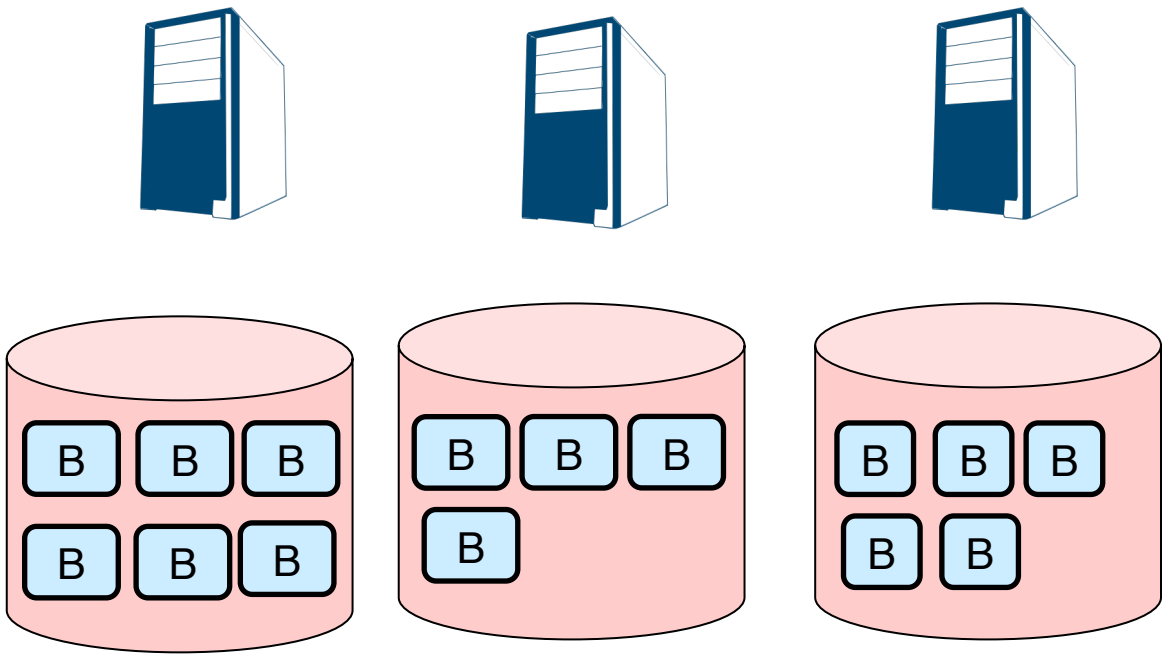
- Block data
- Name node location

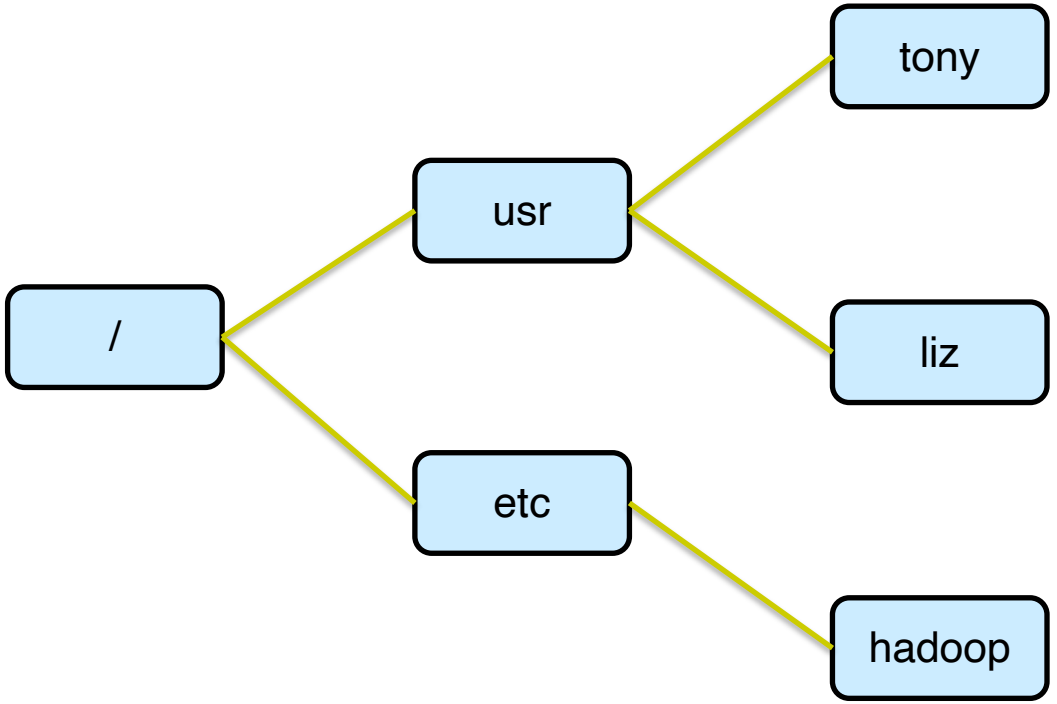


Data Loading



HDFS Storage: Analogy to Unix File System

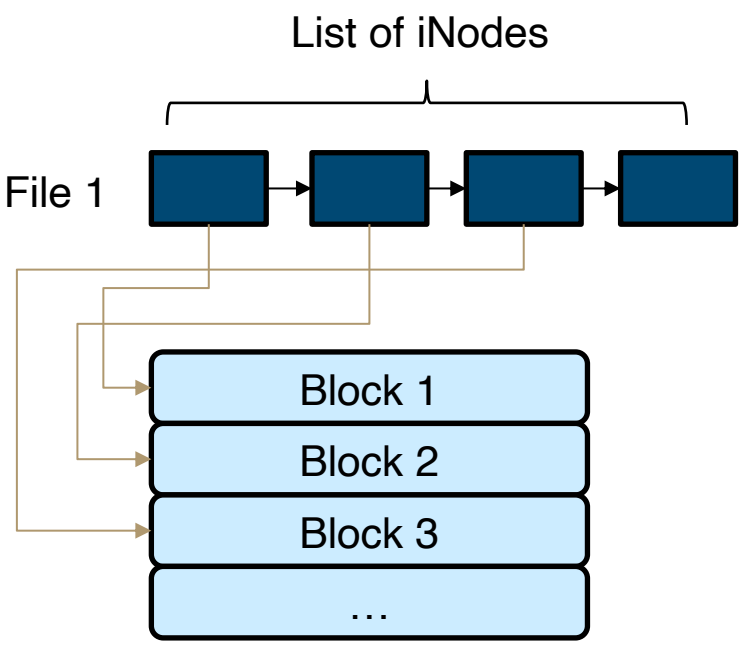




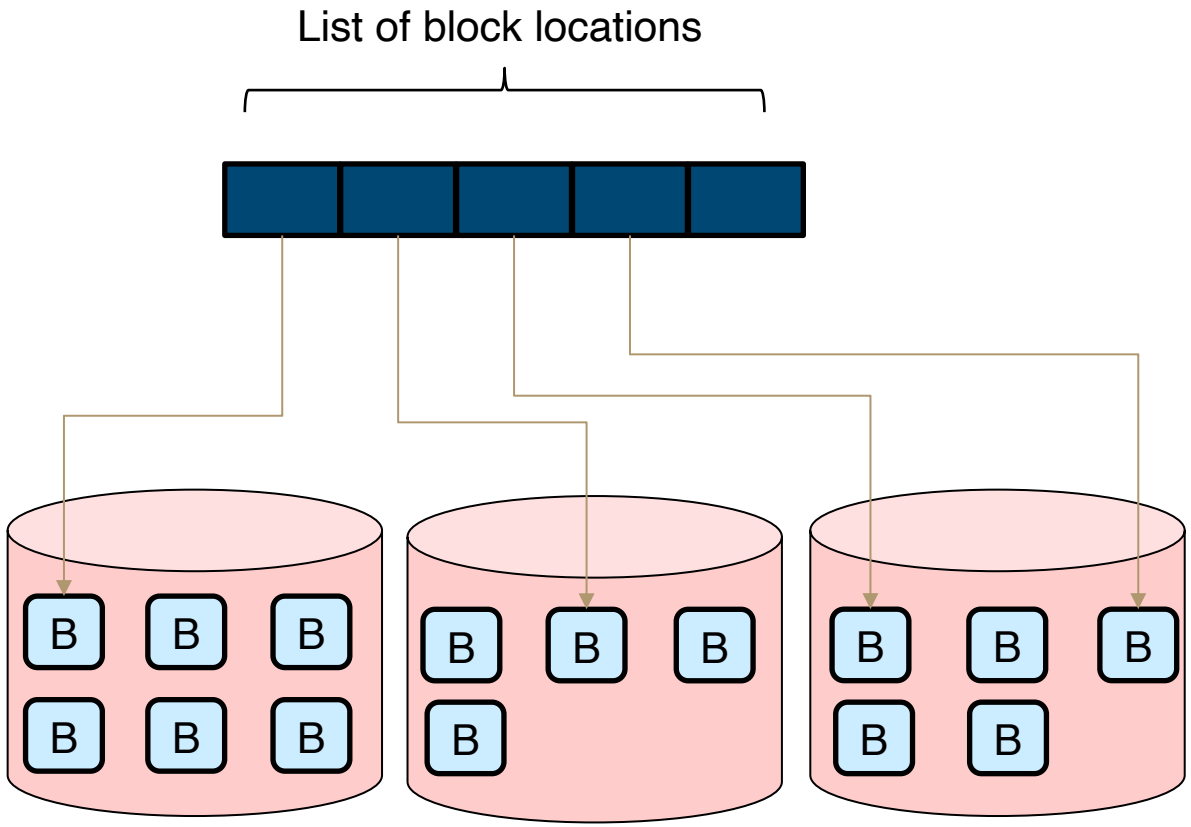
FAULT TOLERANCE

Analogy to Unix File System

- The physical model is comparable



Unix



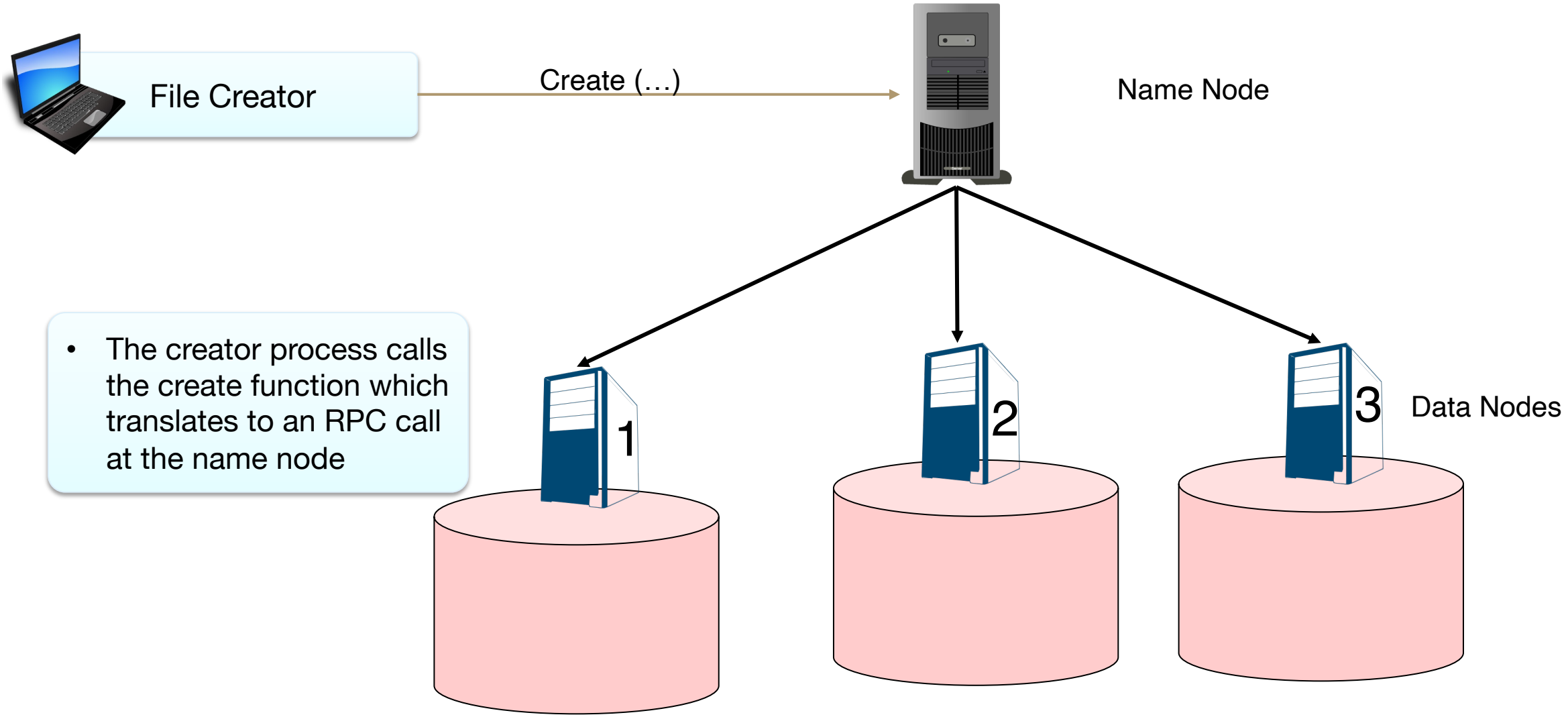
HDFS

Replication

- The default fault tolerance mechanism in HDFS is replication
- The most common replication factor is three
- If one or two nodes are temporarily unavailable, the data is still accessible
- If one or two nodes permanently fail, the master node replicates the under-replicated blocks to reach the desired replication factor
- Drawback: reduced disk capacity

WRITING TO HDFS

HDFS Create



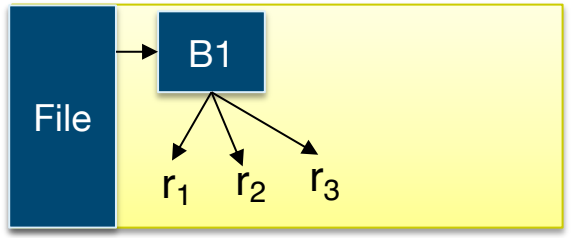
- The creator process calls the create function which translates to an RPC call at the name node

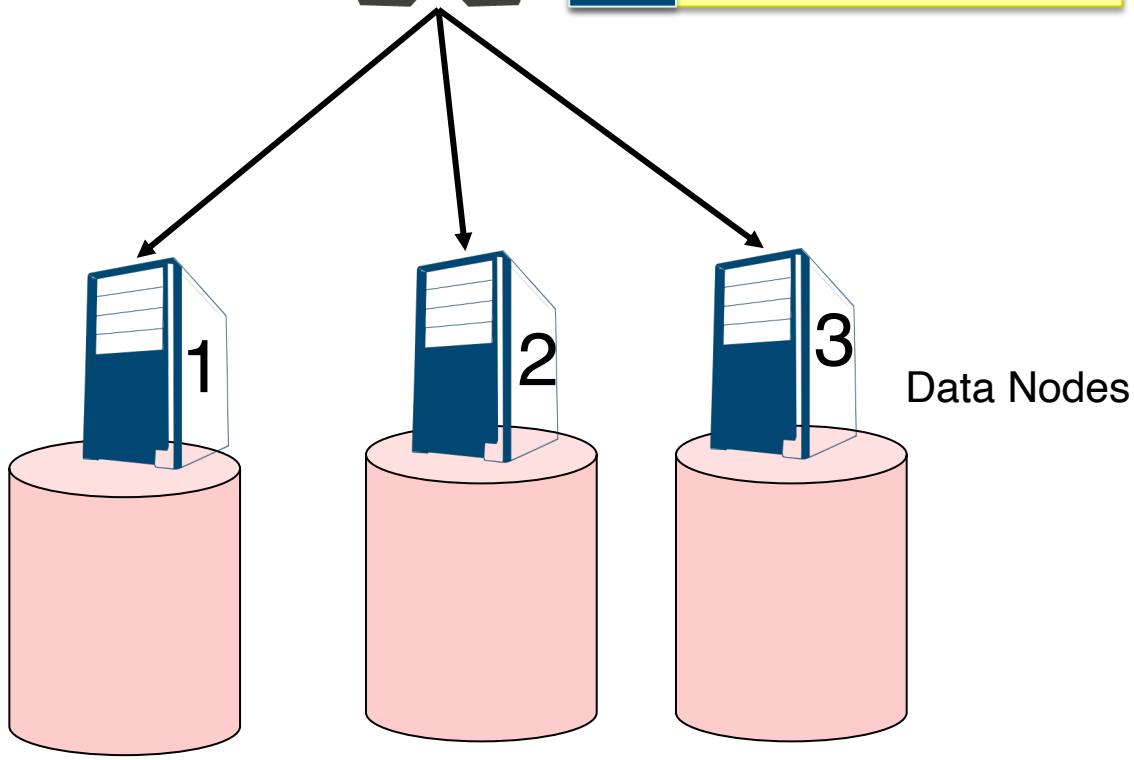
HDFS Create

File Creator

Create (…)

Name Node

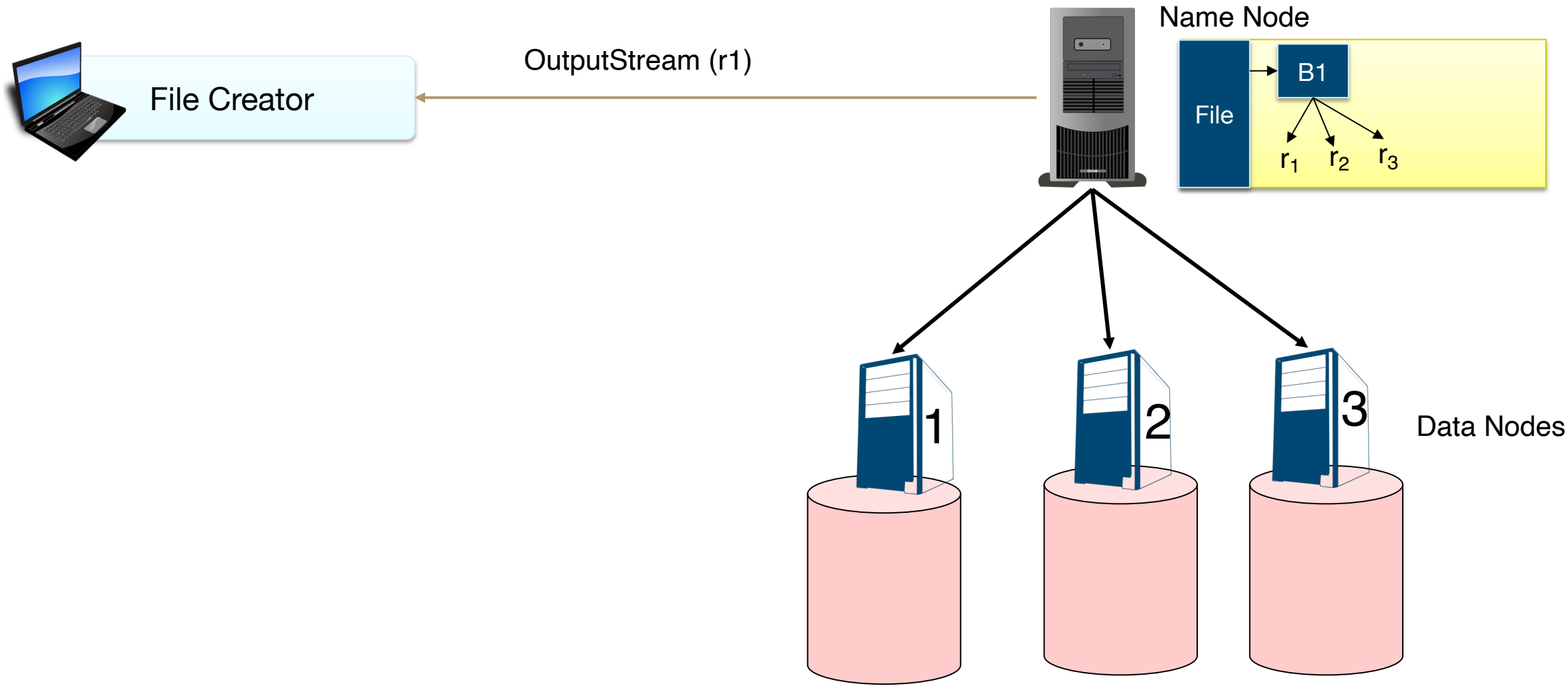




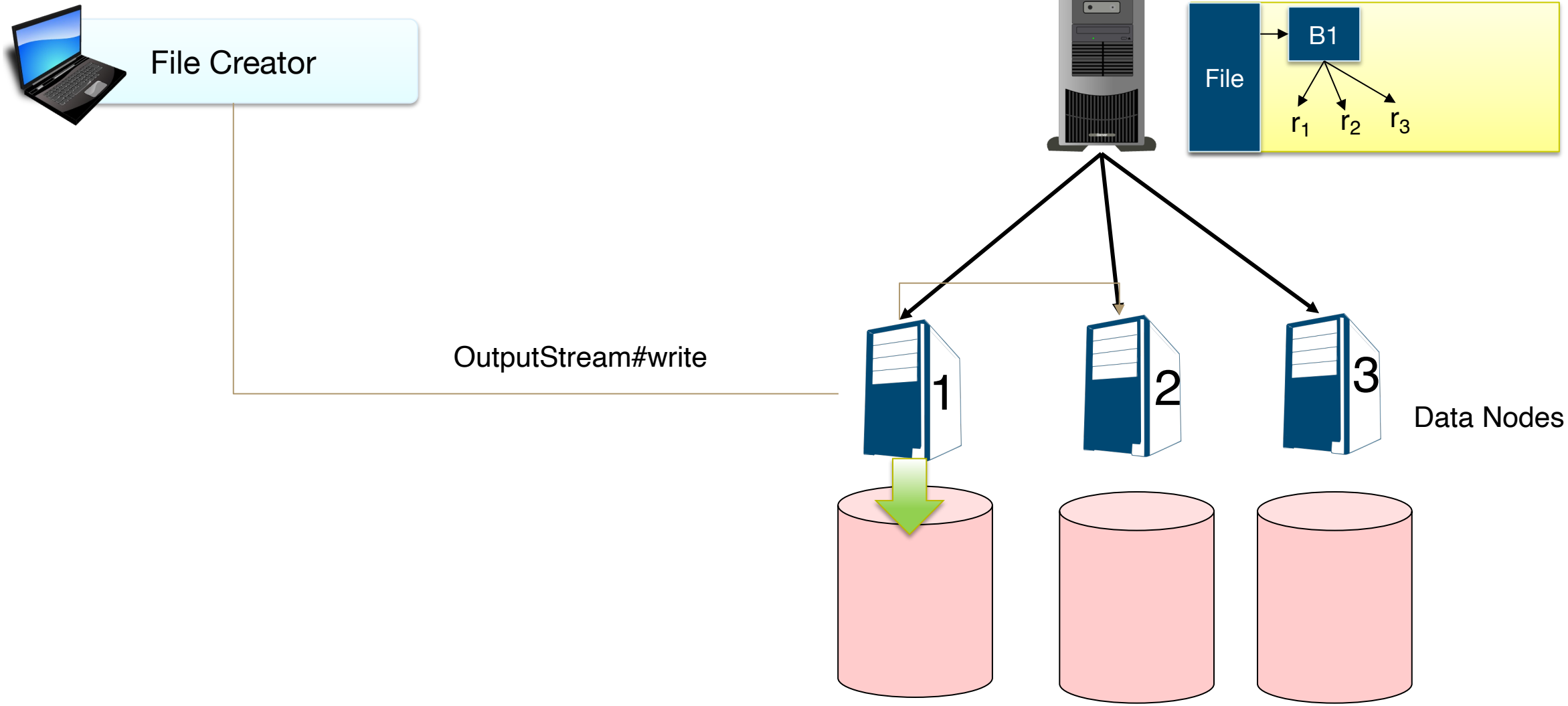
The master node creates one initial block with three replicas

1. First replica is assigned to a random machine
2. Second replica is assigned to another random machine in a different rack
3. Third replica is assigned to a random machine on the same rack of the second machine

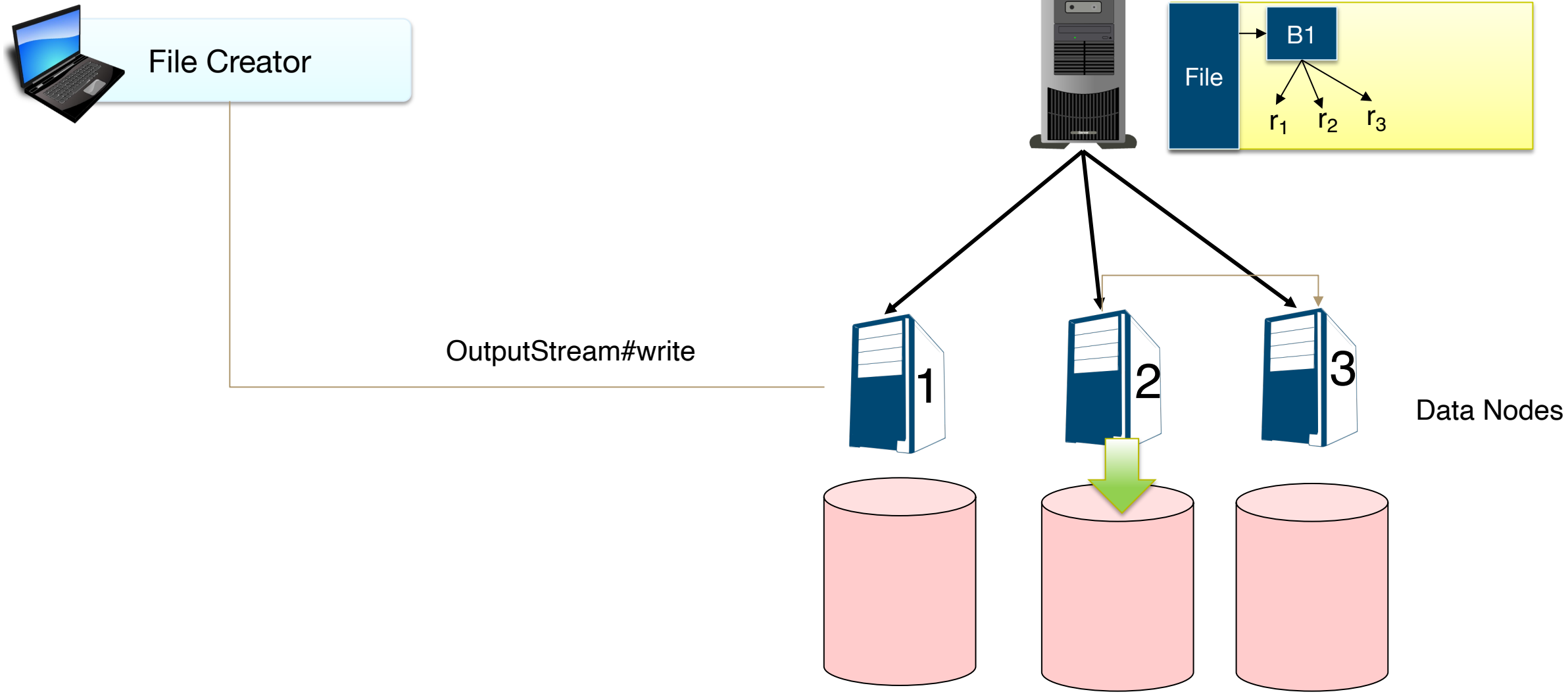
HDFS Create



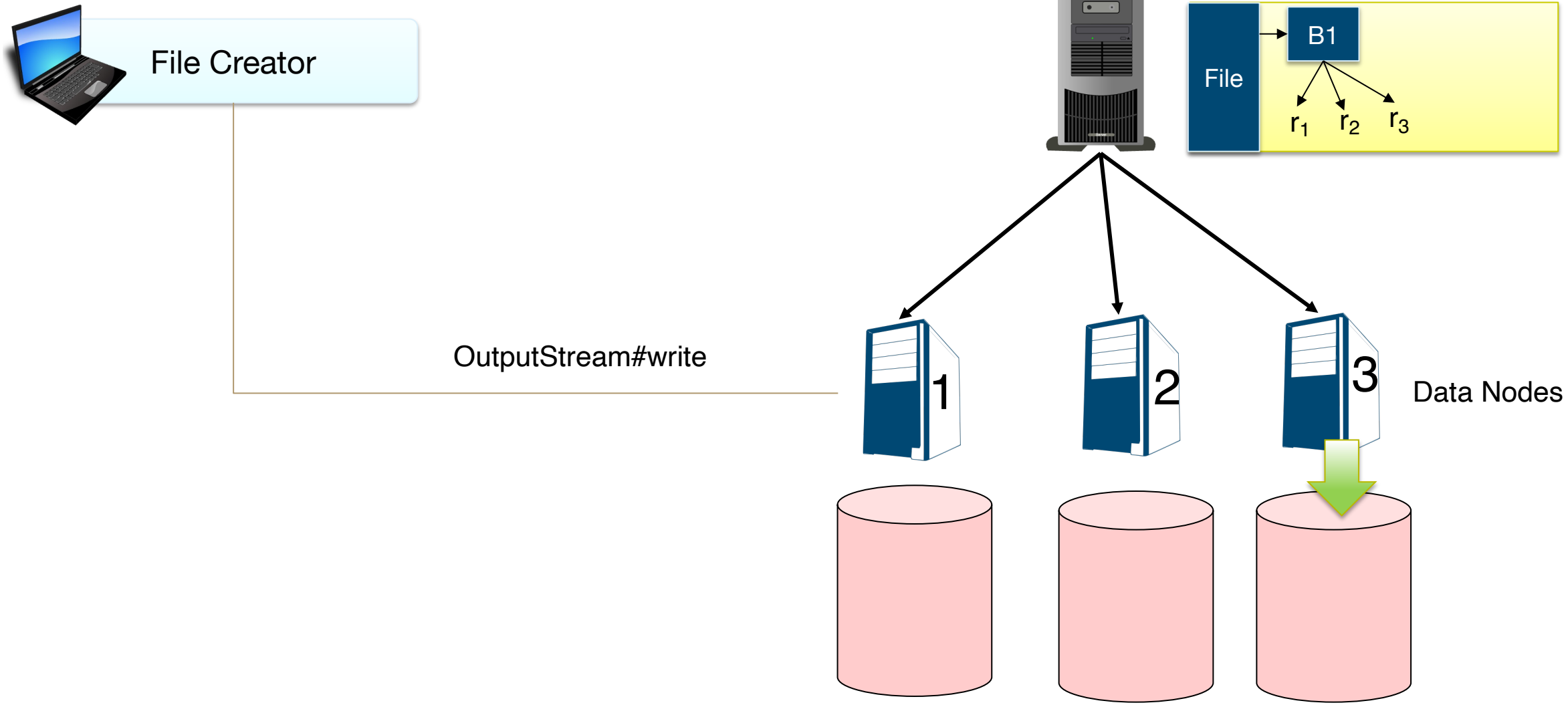
HDFS Create



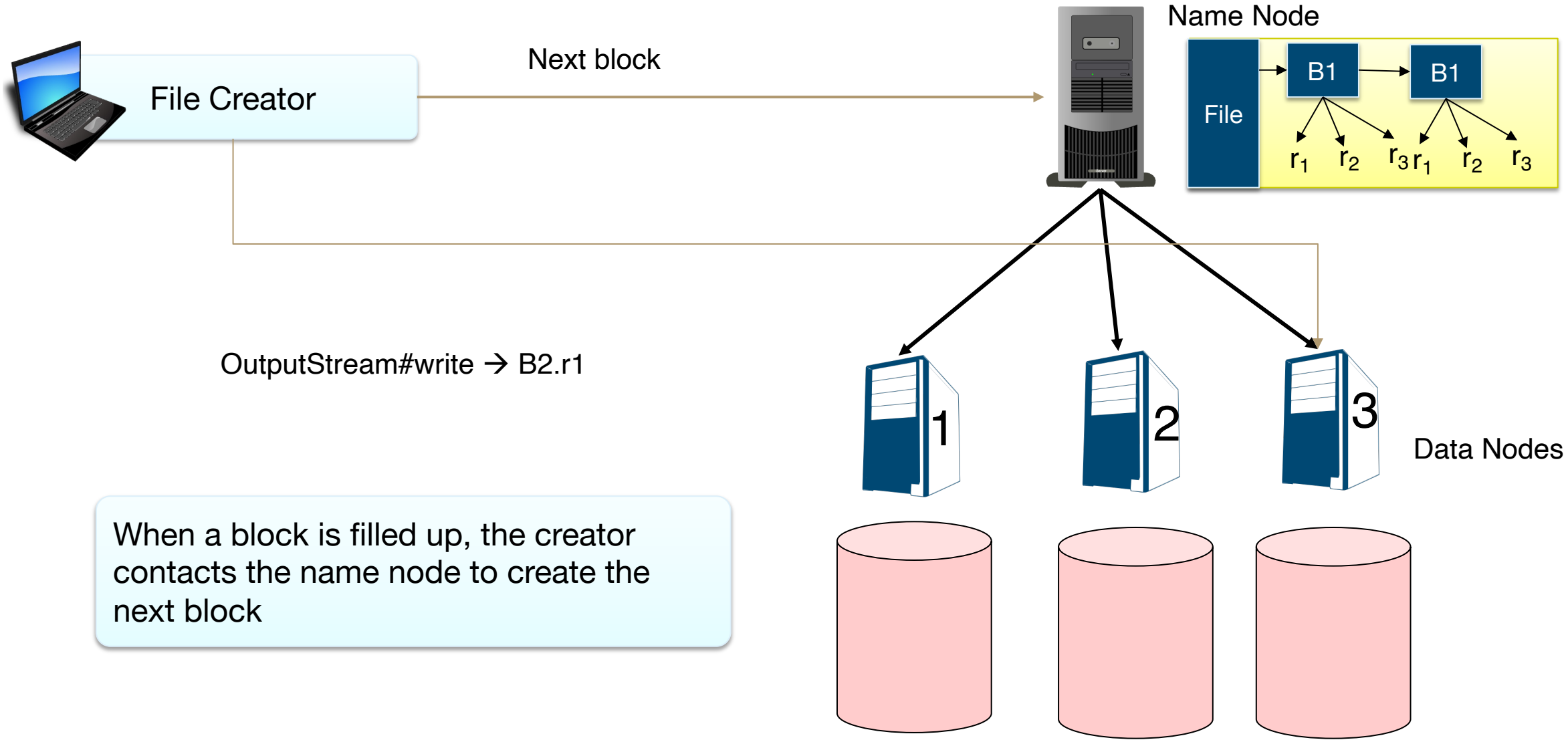
HDFS Create



HDFS Create



HDFS Create



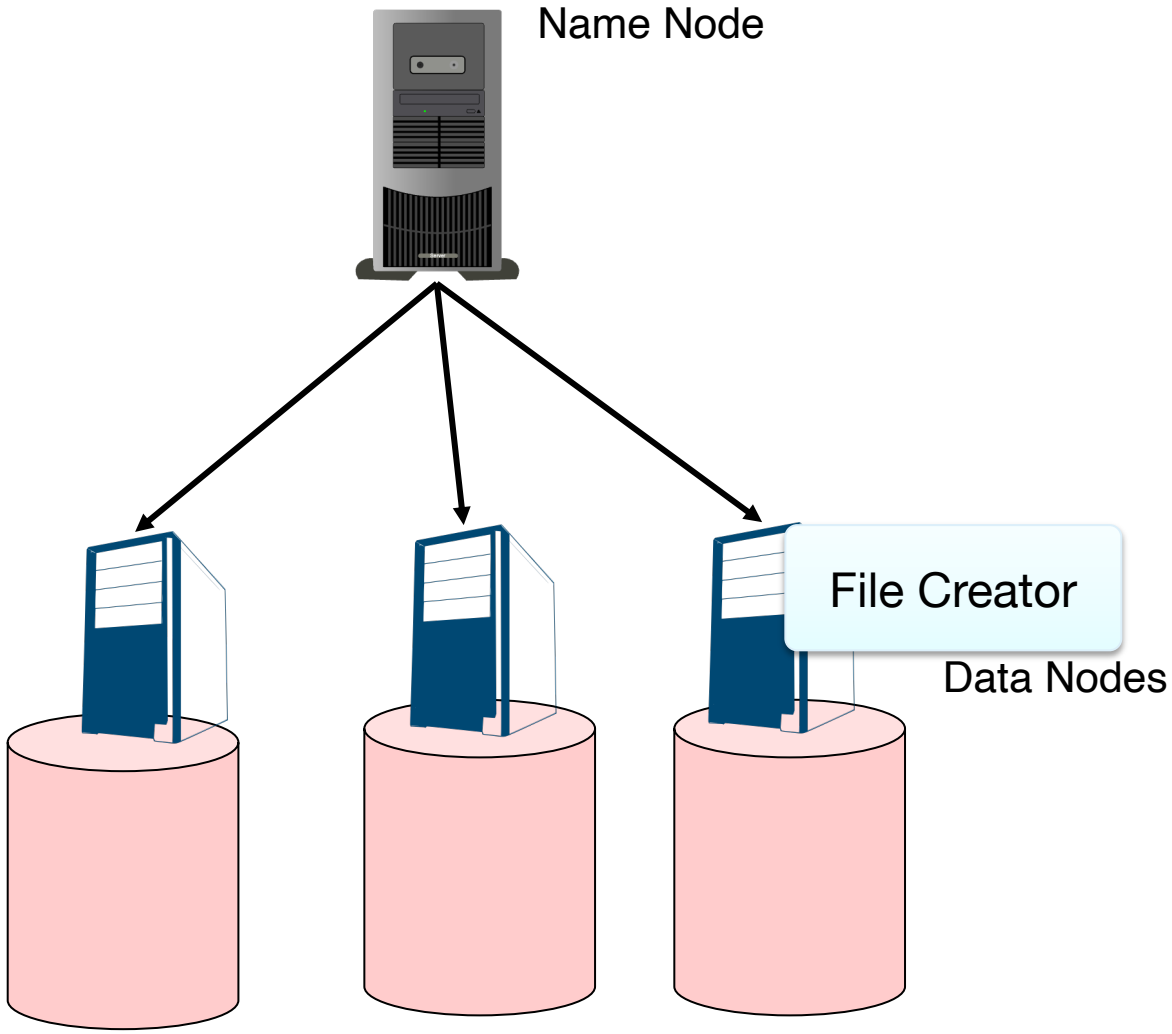
When a block is filled up, the creator contacts the name node to create the next block

Writing to HDFS

- Data transfers of replicas are pipelined
- The data does not go through the name node
- Random writing is not supported
- Appending to a file is supported but it creates a new block

Self Writing

- If the file creator is running on one of the data nodes, the first replica is always assigned to that node
- The second and third replicas are assigned as before, i.e., the second replica on a different rack and the third replica on the same rack as the second one.

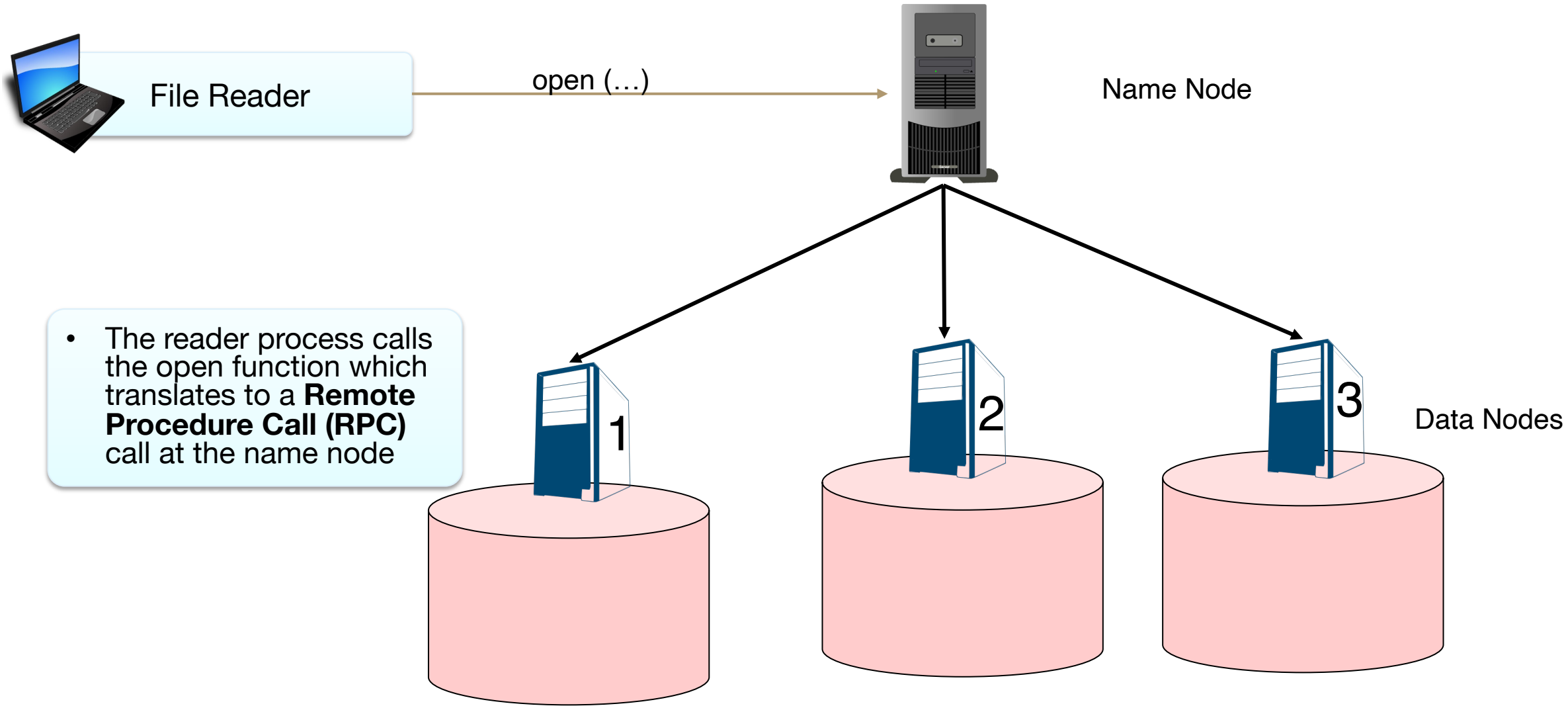


STREAM READING FROM HDFS

Reading From HDFS

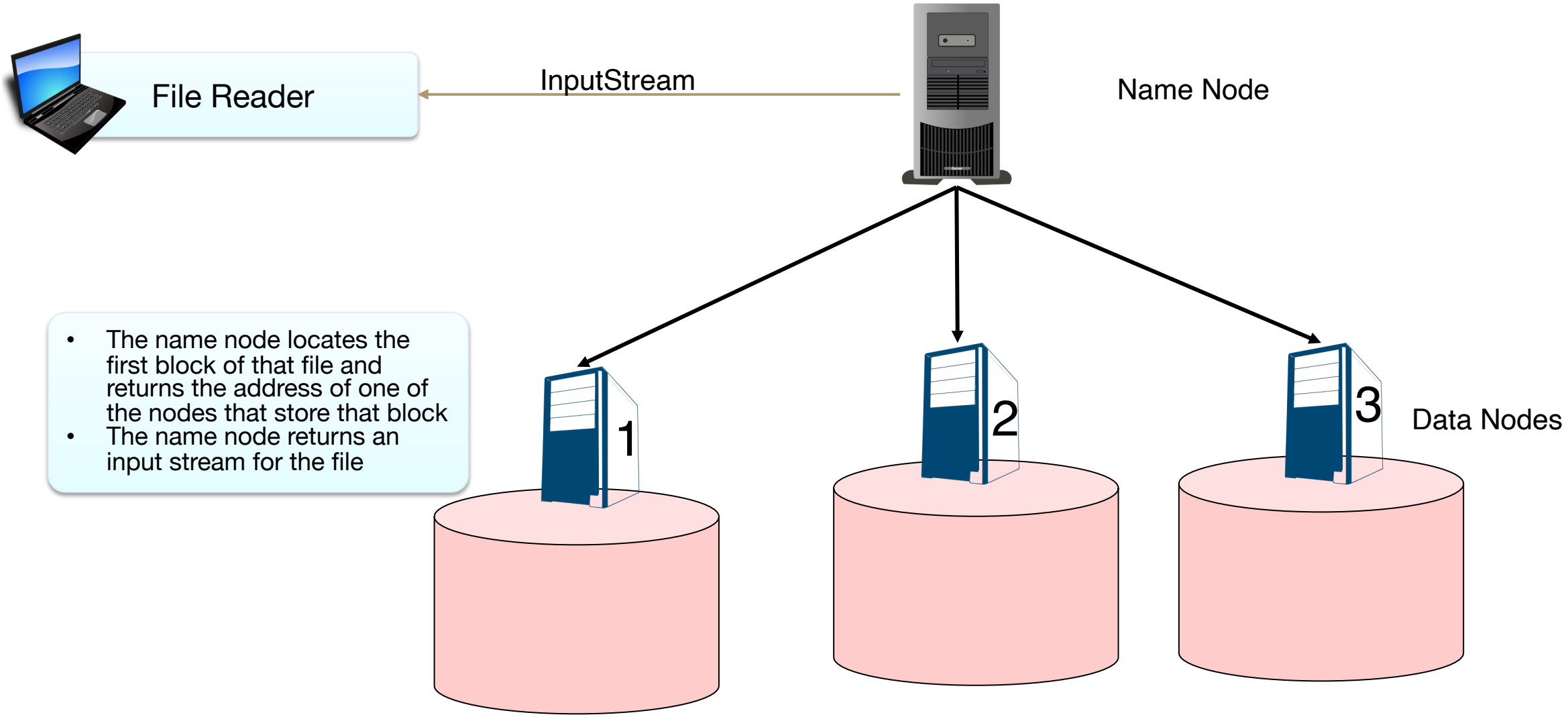
- Reading is relatively easier
- No replication is needed
- Replication can be exploited
- Random reading is allowed

HDFS Read



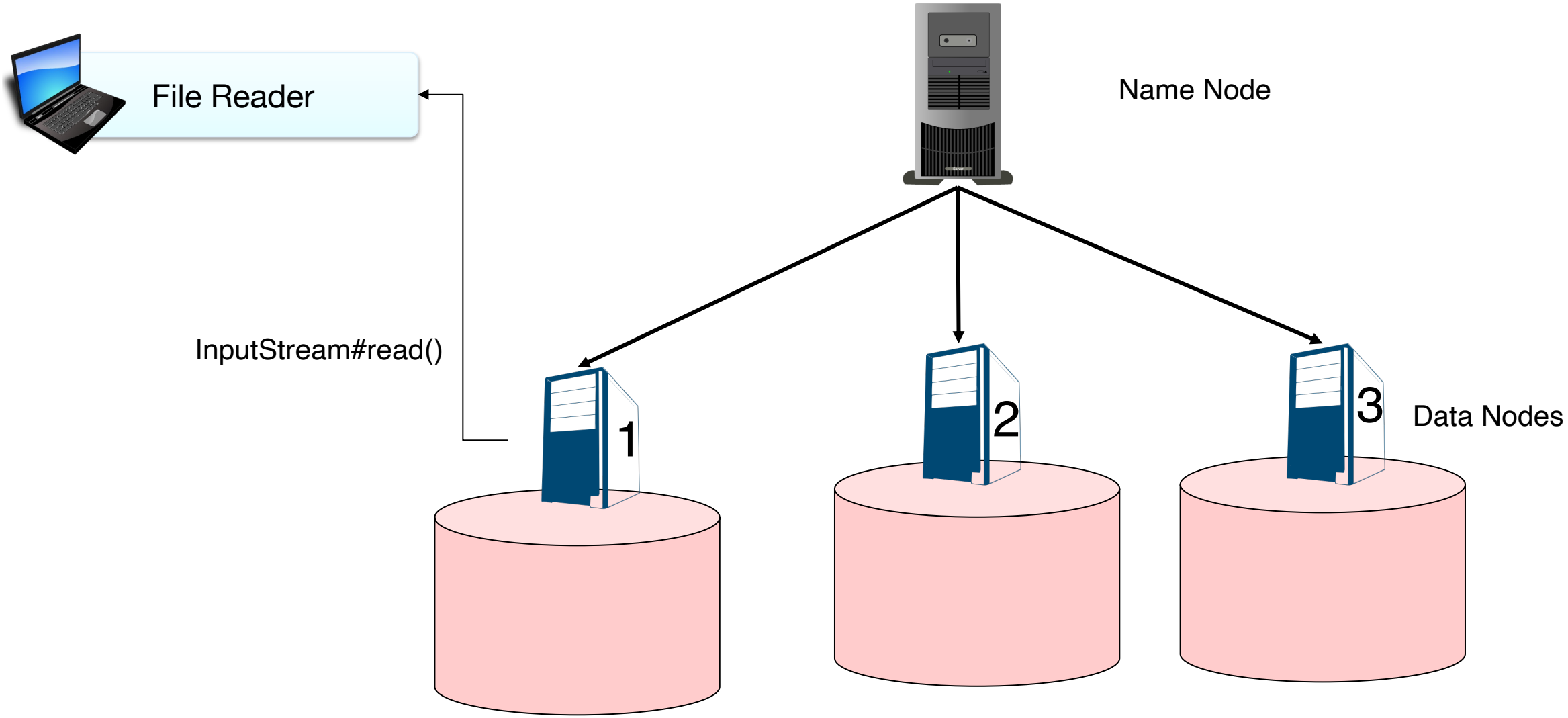
- The reader process calls the open function which translates to a **Remote Procedure Call (RPC)** call at the name node

HDFS Read

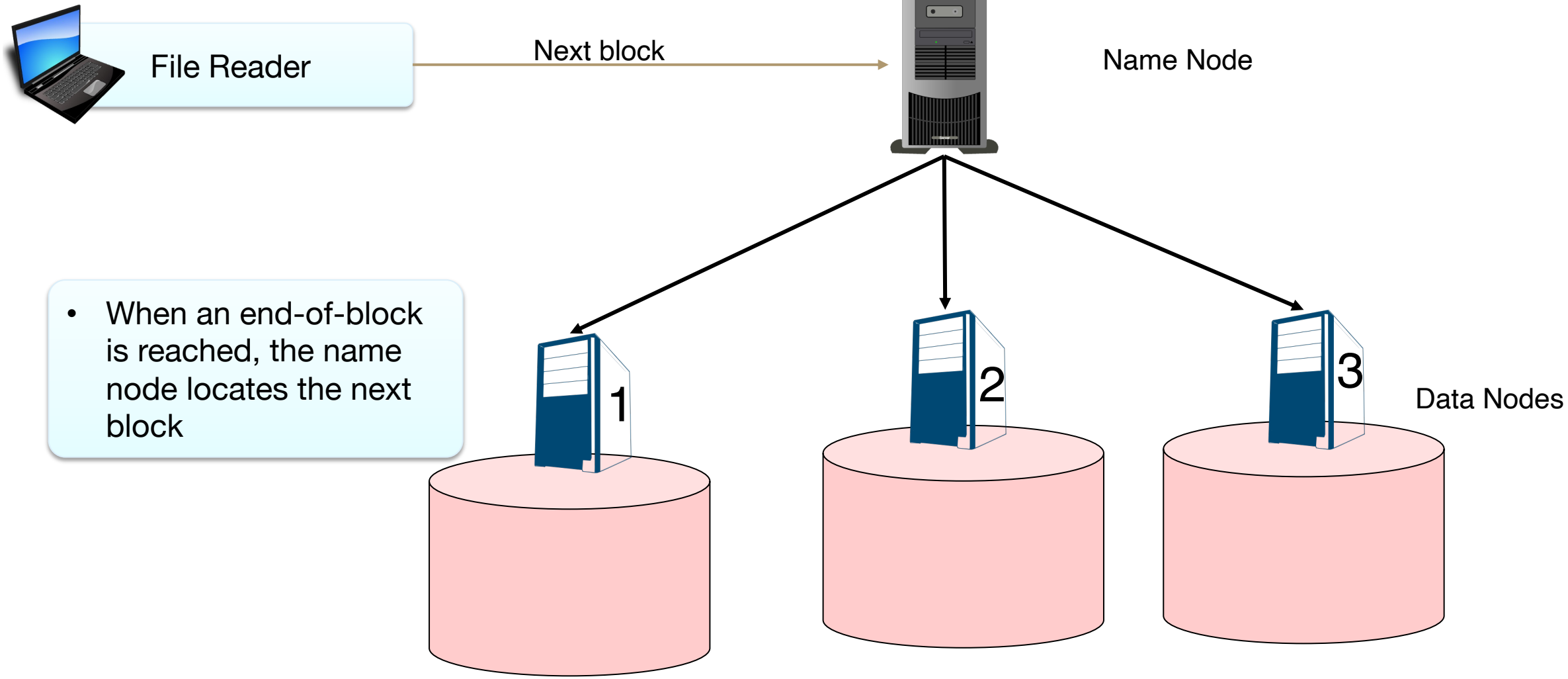


- The name node locates the first block of that file and returns the address of one of the nodes that store that block
- The name node returns an input stream for the file

HDFS Read

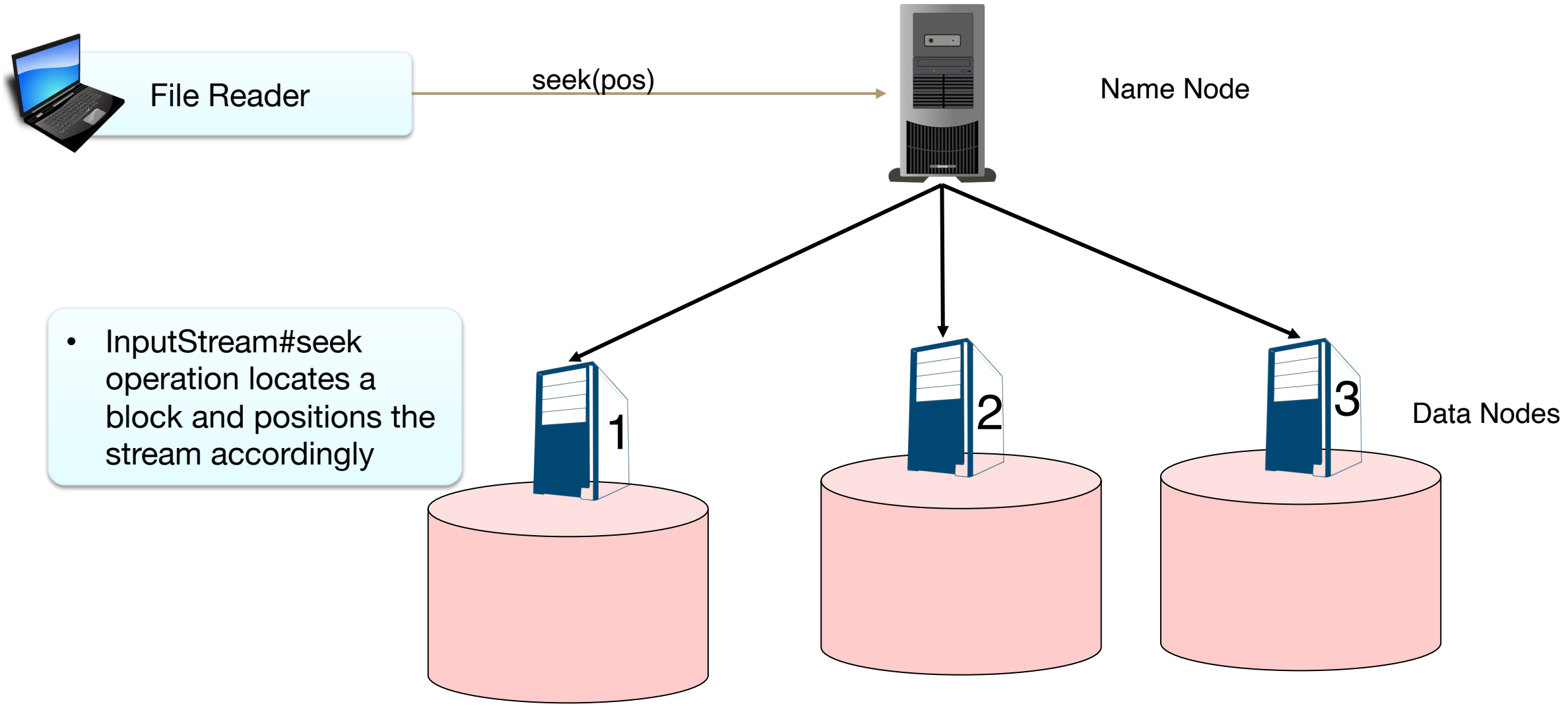


HDFS Read



- When an end-of-block is reached, the name node locates the next block

HDFS Read

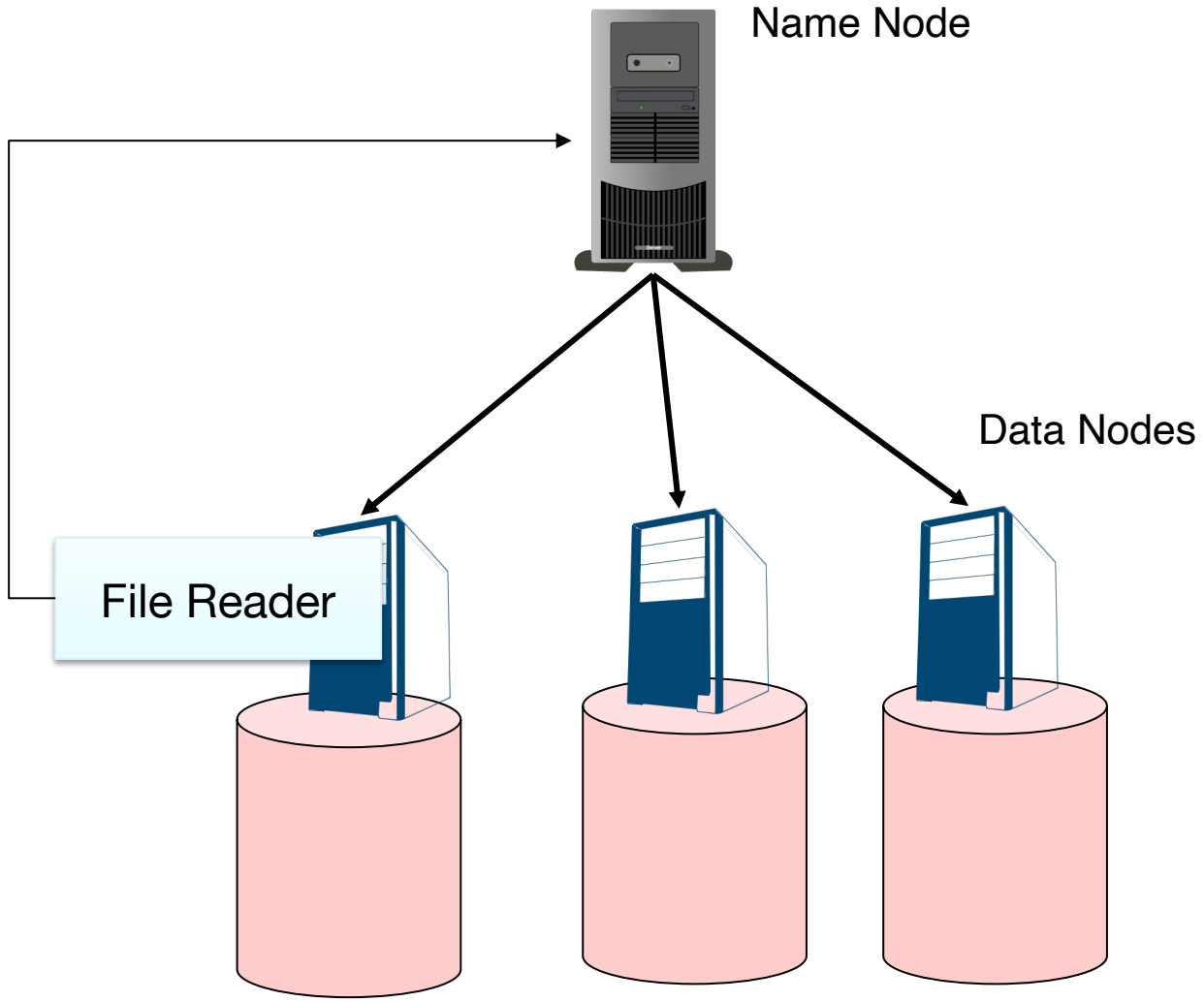


- InputStream#seek operation locates a block and positions the stream accordingly

Self Reading

1. If the block is locally stored on the reader, this replica is chosen to read
2. If not, a replica on another machine in the same rack is chosen
3. Any other random block is chosen

- When self-reading occurs, HDFS can make it much faster through a feature called short-circuit



Reading from HDFS

- The API is much richer than the simple open/seek/close API
  - We can retrieve block locations
  - We can choose a specific replica to read
- The same API is generalized to other file systems including the local FS and S3

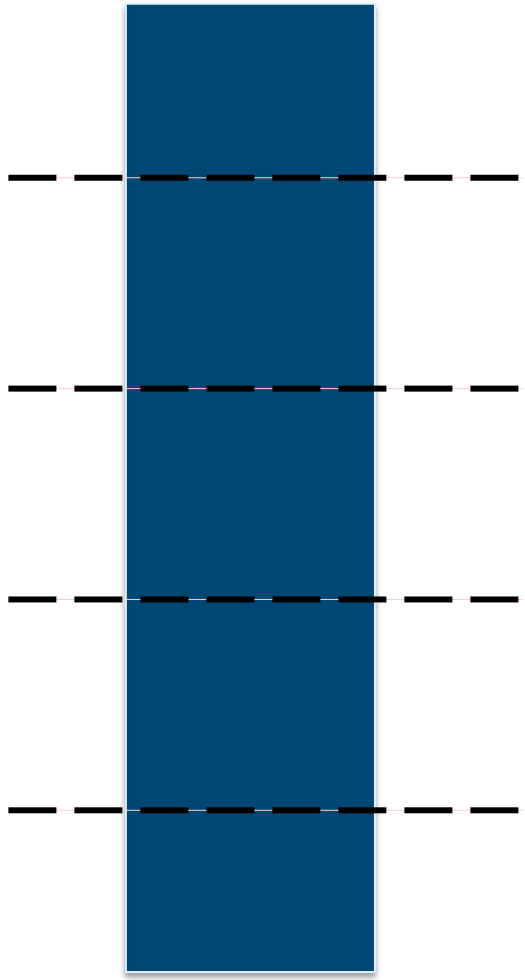
STREAM READING

Stream Reading

- In distributed big data processing, input files contain records not just raw bytes
- We need to a way to read records from files in HDFS
- For efficiency, we should split the file and read it in parallel

File Splitting

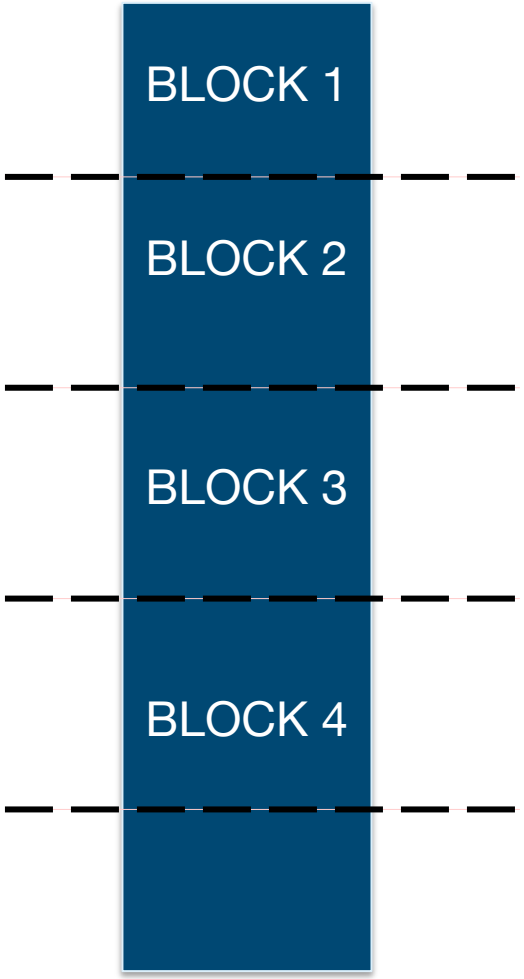
- How to split the file?
  - By record: For fixed-size records
  - By size: For variable-size records
- Considerations, splitting the file should be fast
- It Should not be required that we read the entire file to split it



Input File

Default File Splitting

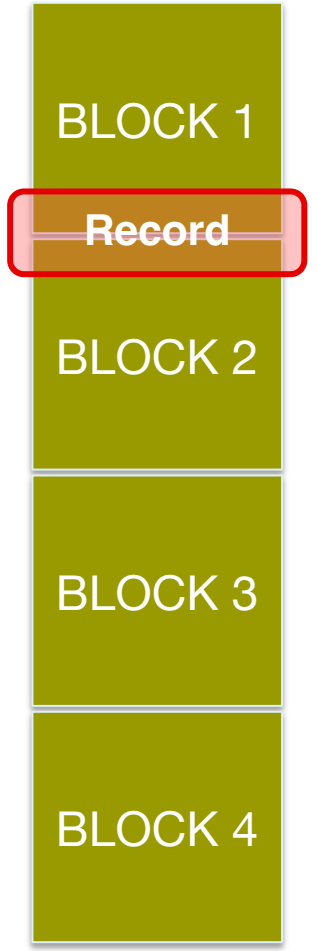
- A split is created for each block
- Advantages
  - Data locality
  - Efficiency
- If the file is too small, a single block might be further split



Input File

Read Data in One Split

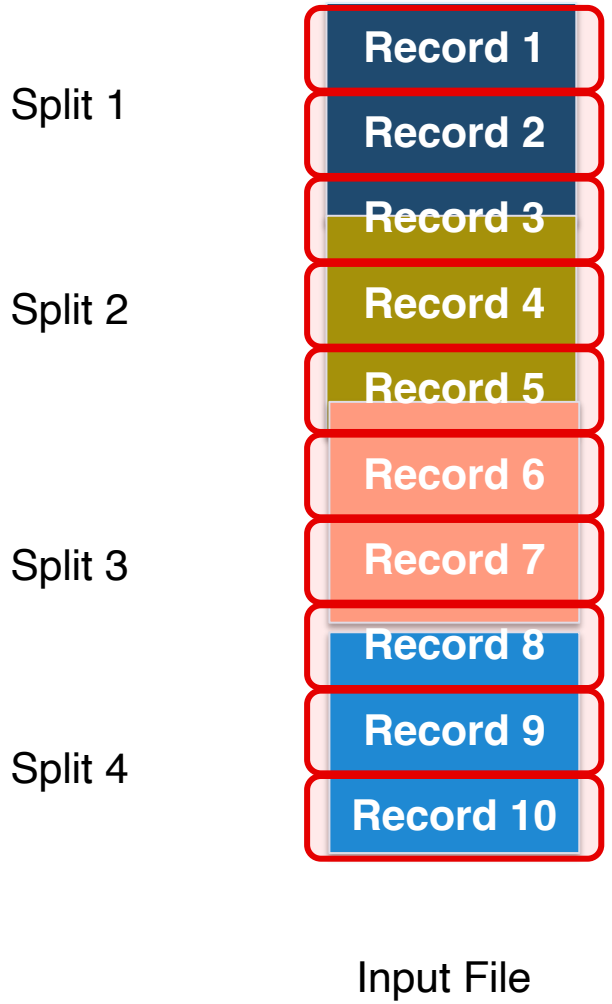
- Read all the records that are inside the split
- How to deal with records that overlap two splits?
- In each split, we should read the records that start in that split



Input File

Read Data in *Every* Split

- Read all the records that are inside the split
- How to deal with records that overlap two splits?
- In each split, we should read the records that start in that split



Reading Process

- Split the file based on the file metadata
  - File size, block sizes, # of nodes
- Each split is defined by:
  - File name, Start offset, Length
- For each split:
  - Seek to the start offset
  - Skip the first record (except for the first split)
  - Read until the beginning of the record goes beyond the start + length

SPECIAL FEATURES

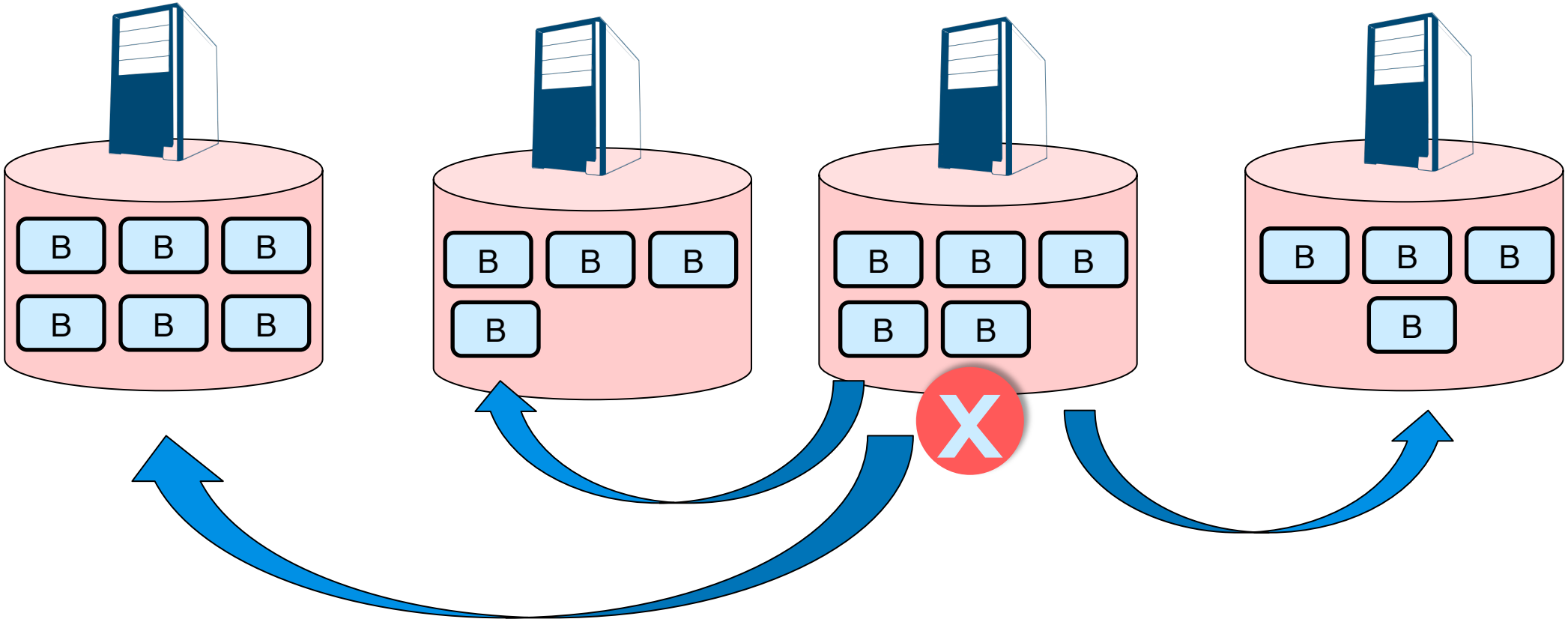
Special Features in HDFS

- Space Reclamation
- Node decommission
- Load balancer
- Cheap concatenation
- Integrity

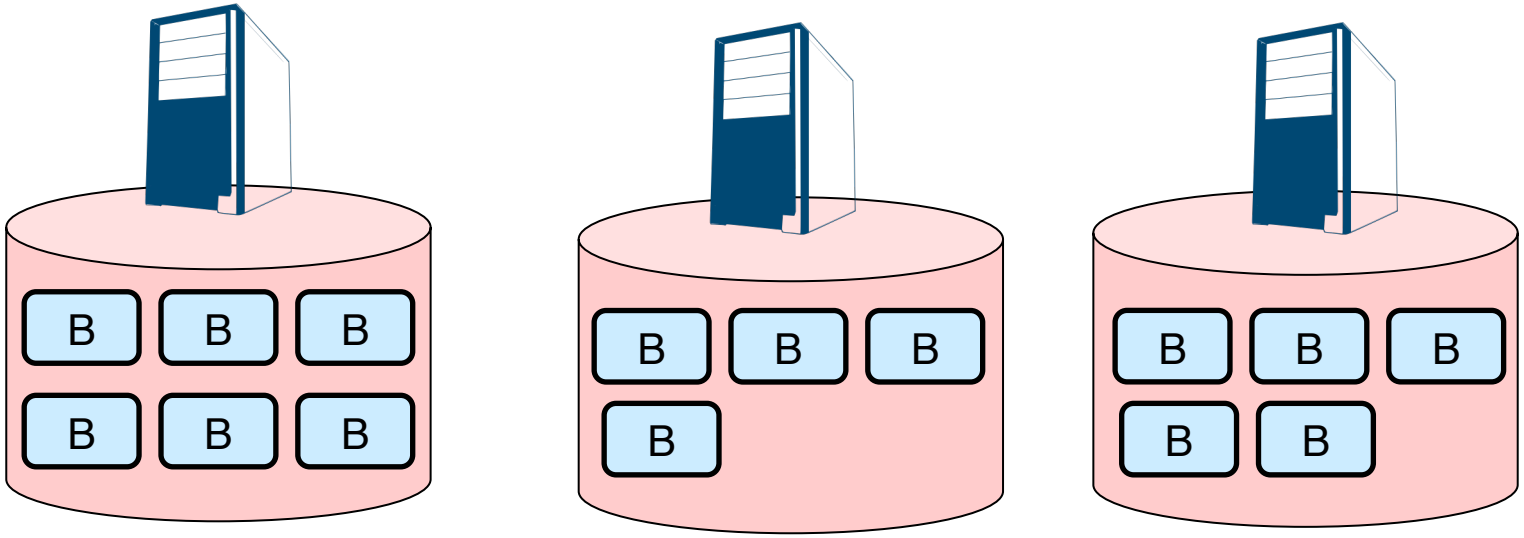
Space Reclamation

- When a file is deleted by a user or an application, it is not immediately removed from HDFS.
- Instead, HDFS first renames it to a file in the /trash directory.
- The file can be restored quickly as long as it remains in /trash. A file remains in /trash for a configurable amount of time.
- After the expiry of its life in /trash, the Namenode deletes the file from the HDFS namespace

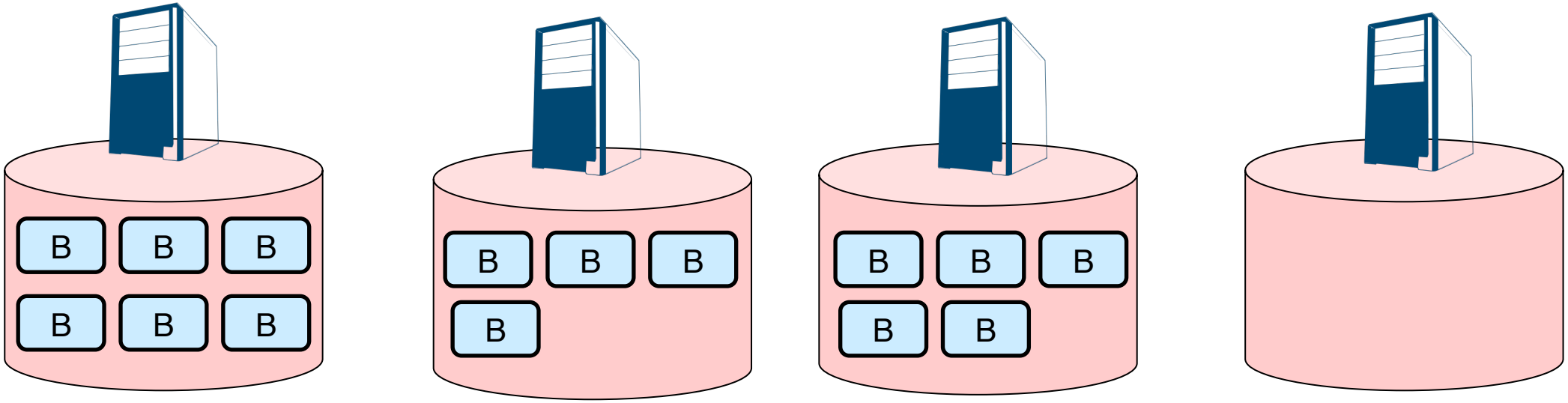
Node Decommission



Load Balancing

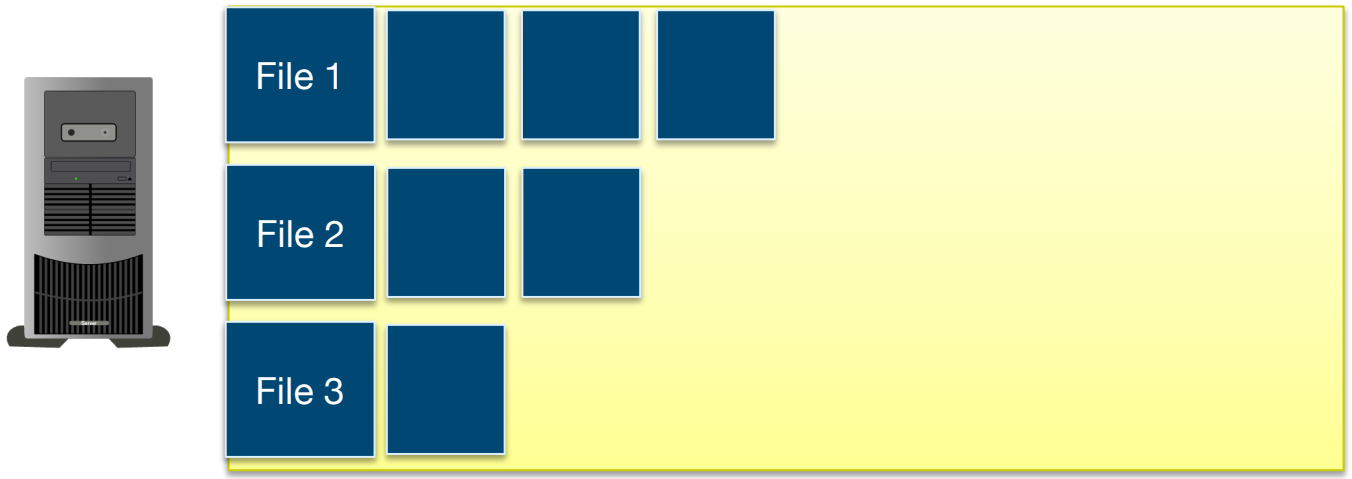


Load Balancing



Cheap Concatenation

- Concatenate File 1 + File 2 + File →File 4



- Rather than creating new blocks, HDFS can change the metadata in the name node to delete File 1, File 2, and File 3, and assign their blocks to a new File 4 in the right order.

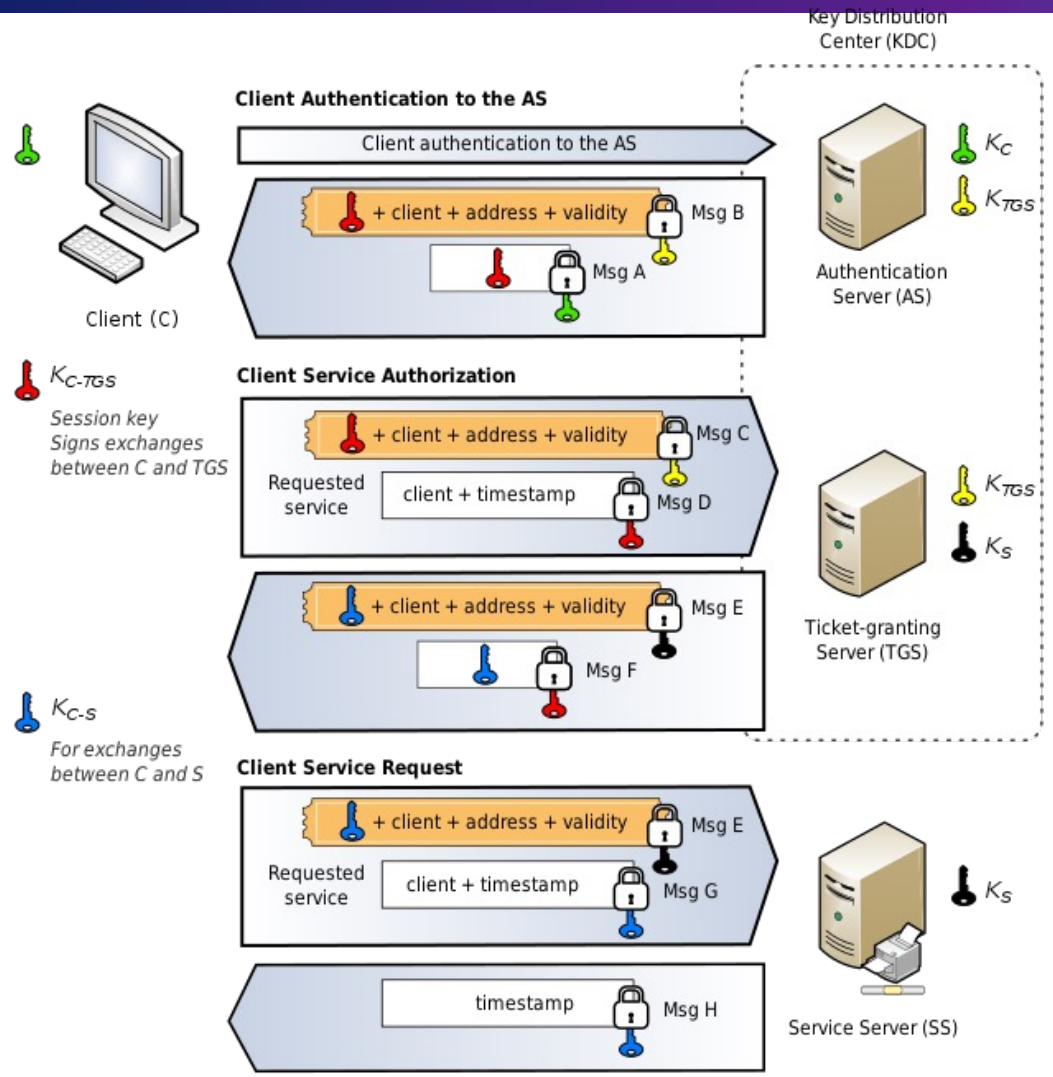
Integrity

- Every I/O operation on disks or the network may corrupt data
  - Users expect data not to be corrupted during storage or processing
  - Data integrity usually achieved with checksums
- HDFS transparently checksums all data during I/O
  - HDFS makes sure that storage overhead is roughly 1%
  - DataNodes are in charge of checksumming
    - With replication, the last replica performs the check
    - Checksums are timestamped and logged for statistcs on disks
  - Checksumming is also run periodically in a separate thread
    - Note that thanks to replication, error correction is possible

Security via Kerberos

- If Hadoop is configured with all of its defaults, Hadoop doesn't do any authentication of users
- Hadoop has the ability to require authentication, in the form of Kerberos principals.
- Kerberos is an authentication protocol which uses "tickets" to allow nodes to identify themselves
- In MapReduce, jobs are submitted via queues controlled by the scheduler. Administrators can define who is allowed to submit jobs to particular queues via MapReduce ACLs

Kerberos Negotiations



HDFS SHELL COMMAND-LINE INTERFACE (CLI)

Testing a Single Node Cluster

- Follow the instructions on the following site to install and set up Hadoop/HDFS:
  - https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-common/SingleCluster.html
- In a pseudo-distributed mode, each Hadoop daemon runs in a separate Java process

Hadoop Configuration

Filename	Format	Description
hadoop-env.sh	Bash script	Environment variables that are used in the scripts to run Hadoop.
core-site.xml	Hadoop configuration XML	I/O settings that are common to HDFS and MapReduce.
hdfs-site.xml	Hadoop configuration XML	Namenode, the secondary namenode, and the datanodes.
mapred-site.xml	Hadoop configuration XML	Jobtracker, and the tasktrackers.
masters	Plain text	A list of machines that each run a secondary namenode.
slaves	Plain text	A list of machines that each run a datanode and a tasktracker.

HDFS Configuration: etc/hadoop/core-site.xml

```
<configuration>
    <property>
        <name>fs.defaultFS</name>
        <value>hdfs://localhost:9000</value>
    </property>
</configuration>
```

HDFS Configuration: etc/hadoop/hdfs-site.xml:

```
<configuration>
    <property>
        <name>dfs.replication</name>
        <value>1</value>
    </property>
</configuration>
```

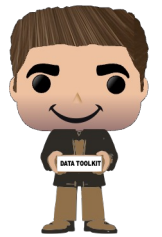
HDFS SHELL

- Used for common operations
- Its usage is similar to Unix shell commands
- Basic operations include
  - ls, cp, mv, mkdir, rm, ...
- HDFS-specific operations include
  - copyToLocal, copyFromLocal, setrep, appendToFile, ...

HDFS SHELL

- General Format
  - `hdfs dfs -<cmd> <arguments>`
- Instead of:
  - `mkdir –p myproject/mydir`
- We write:
  - `hdfs dfs -mkdir –p myproject/mydir`

Read more about commands and usage here:
https://hadoop.apache.org/docs/r3.3.0/hadoop-project-dist/hadoop-common/FileSystemShell.html



HDFS Commands in Summary

Command	Description
`-rm`	Removes file or directory
`-ls`	Lists files with permissions and other details
`-mkdir`	Creates a directory named path in HDFS
`-cat`	Shows contents of the file
`-rmdir`	Deletes a directory
`-put`	Uploads a file or folder from a local disk to HDFS
`-rmr`	Deletes the file identified by path or folder and subfolders
`-get`	Moves file or folder from HDFS to local file
`-count`	Counts number of files, number of directory, and file size
`-df`	Shows free space
`-getmerge`	Merges multiple files in HDFS
`-chmod`	Changes file permissions
`-copyToLocal`	Copies files to the local system
`-Stat`	Prints statistics about the file or directory
`-head`	Displays the first kilobyte of a file
`-usage`	Returns the help for an individual command
`-chown`	Allocates a new owner and group of a file

DFSAdmin

- The *DFSAdmin* command set is used for administering an HDFS cluster. These are commands that are used only by an HDFS administrator.
- Here are some sample action/command pairs:

Action	Command
Put a cluster in SafeMode	`bin/hadoop dfsadmin -safemode enter`
Generate a list of Datanodes	`bin/hadoop dfsadmin -report`
Decommission Datanode `datanodename`	`bin/hadoop dfsadmin -decommission datanodename`

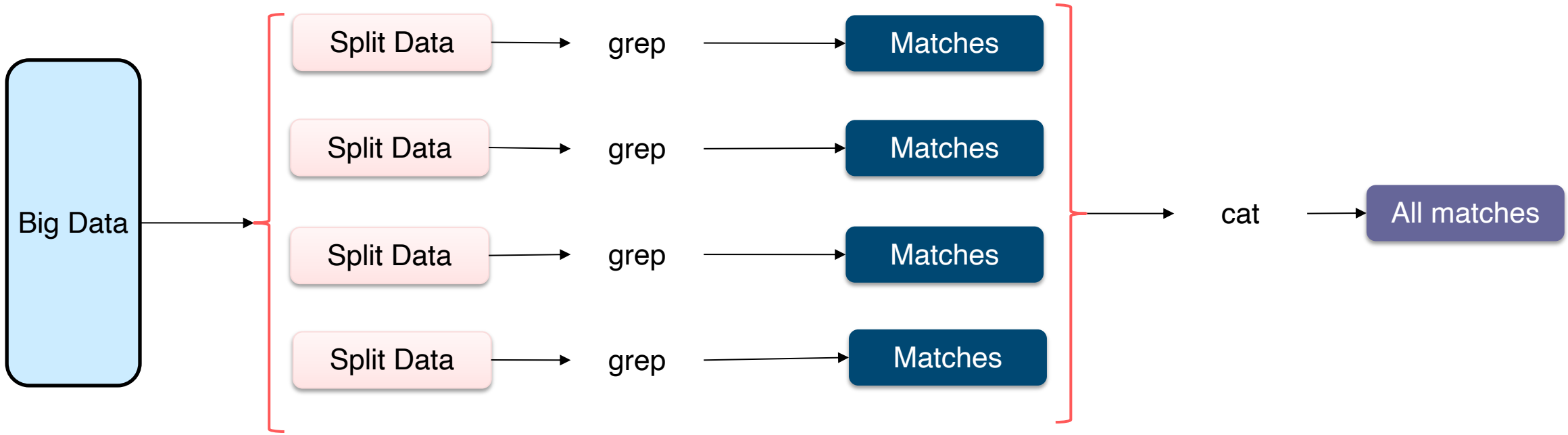
Set Up HDFS Using the HDFS Shell

Command	Description
`$ bin/hdfs namenode -format`	Format the file-system
`$ sbin/start-dfs.sh`	Start NameNode daemon and DataNode daemon
`http://localhost:9870/`	Browse the web interface for the NameNode
`$ bin/hdfs dfs -mkdir /user` `$ bin/hdfs dfs -mkdir /user/<username>`	Make the HDFS directories required to execute MapReduce jobs
`$ bin/hdfs dfs -mkdir input` `$ bin/hdfs dfs -put etc/hadoop/*.xml input`	Copy the input files into the distributed filesystem
`$bin/hadoop jar share/hadoop/mapreduce/hadoop-mapreduce-examples-3.3.4.jar grep input output 'dfs[a-z.]+'`	Run some of the examples provided
`$ bin/hdfs dfs -get output output` `$ cat output/*`	Copy the output files from the distributed filesystem to the local filesystem and examine them
`$ sbin/stop-dfs.sh`	Stop the daemons

Read more here: https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-common/SingleCluster.html

Example Hadoop Split

- `$bin/hadoop jar share/hadoop/mapreduce/hadoop-mapreduce-examples-3.3.4.jar grep input output 'dfs[a-z.]+'`
- See more examples here: https://jar-download.com/artifacts/org.apache.hadoop/hadoop-mapreduce-examples/3.2.0/source-code



HDFS API

HDFS API CLASSES

- **Configuration**: Holds system configuration such as where the master node is running and default system parameters, e.g., replication factor and block size
- **Path**: Stores a path to a file or directory
- **FileSystem**: An abstract class for file system operations

Java API

```
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FSDataInputStream;
import org.apache.hadoop.fs.FSDataOutputStream;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.commons.io.IOUtils;

public void writeFileToHDFS() throws IOException {
    Configuration configuration = new Configuration();
    configuration.set("fs.defaultFS", "hdfs://localhost:9000");
    FileSystem fileSystem = FileSystem.get(configuration);
    //Create a path
    String fileName = "read_write_hdfs_example.txt";
    Path hdfsWritePath = new Path("/usr/drfitz/" + fileName);
    FSDataOutputStream fsDataOutputStream =
fileSystem.create(hdfsWritePath,true);

    BufferedWriter bufferedWriter = new BufferedWriter(new
OutputStreamWriter(fsDataOutputStream, StandardCharsets.UTF_8));
    bufferedWriter.write("Java API to write data in HDFS");
    bufferedWriter.newLine();
    bufferedWriter.close();
    fileSystem.close();
}
```

```
<dependency>
    <groupId>org.apache.hadoop</groupId>
    <artifactId>hadoop-mapreduce-client-core</artifactId>
    <version>3.3.2</version>
</dependency>

<!-- https://mvnrepository.com/artifact/org.apache.hadoop/hadoop-common -->
<dependency>
    <groupId>org.apache.hadoop</groupId>
    <artifactId>hadoop-common</artifactId>
    <version>3.3.2</version>
</dependency>
```

Java API

```java
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FSDataInputStream;
import org.apache.hadoop.fs.FSDataOutputStream;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.commons.io.IOUtils;

public void readFileFromHDFS() throws IOException {
    Configuration configuration = new Configuration();
    configuration.set("fs.defaultFS", "hdfs://localhost:9000");
    FileSystem fileSystem = FileSystem.get(configuration);

    //Create a path
    String fileName = "read_write_hdfs_example.txt";
    Path hdfsReadPath = new Path("/usr/drfitz/data/" + fileName);

    //Init input stream
    FSDataInputStream inputStream = fileSystem.open(hdfsReadPath);

    //Classical input stream usage
    String out= IOUtils.toString(inputStream, StandardCharsets.UTF_8);
    System.out.println(out);

    inputStream.close();
    fileSystem.close();
}
```

```xml
<dependency>
    <groupId>org.apache.hadoop</groupId>
    <artifactId>hadoop-mapreduce-client-core</artifactId>
    <version>3.3.2</version>
</dependency>

<!-- https://mvnrepository.com/artifact/org.apache.hadoop/hadoop-common -->
<dependency>
    <groupId>org.apache.hadoop</groupId>
    <artifactId>hadoop-common</artifactId>
    <version>3.3.2</version>
</dependency>
```

Java API

- Create a new file
  - `FSDataOutputStream out = fs.create(path, ...);`
- Rename a file
  - `fs.rename(oldPath, newPath);`
- Delete a file
  - `fs.delete(path, recursive);`
  - `fs.deleteOnExit(path);`

Java API

- Open a file
  - FSDataInputStream in = fs.open(path, ...);
- Seek to a different location
  - in.seek(pos);
  - in.seekToNewSource(pos);

Java API

- Concatenate
  - `fs.concat(destination, src[]);`
- Get file metadata
  - `fs.getFileStatus(path);`
- Get block locations
  - `fs.getFileBlockLocations(path, from, to);`

Python API

- We may also use python to interact with HDFS
  - https://crs4.github.io/pydoop/api_docs/hdfs_api.html
  - https://programtalk.com/vs4/python/2656/pydoop/test/hdfs/test_hdfs_fs.py/

The End