Hadoop Distributed File System (HDFS)

Big Data Programming
Lecture 2

79% of enterprise executives agree that companies that do not embrace Big Data will lose their competitive position and could face extinction. Even more, 83%, have pursued Big Data projects to seize a competitive edge.

Source: Accenture

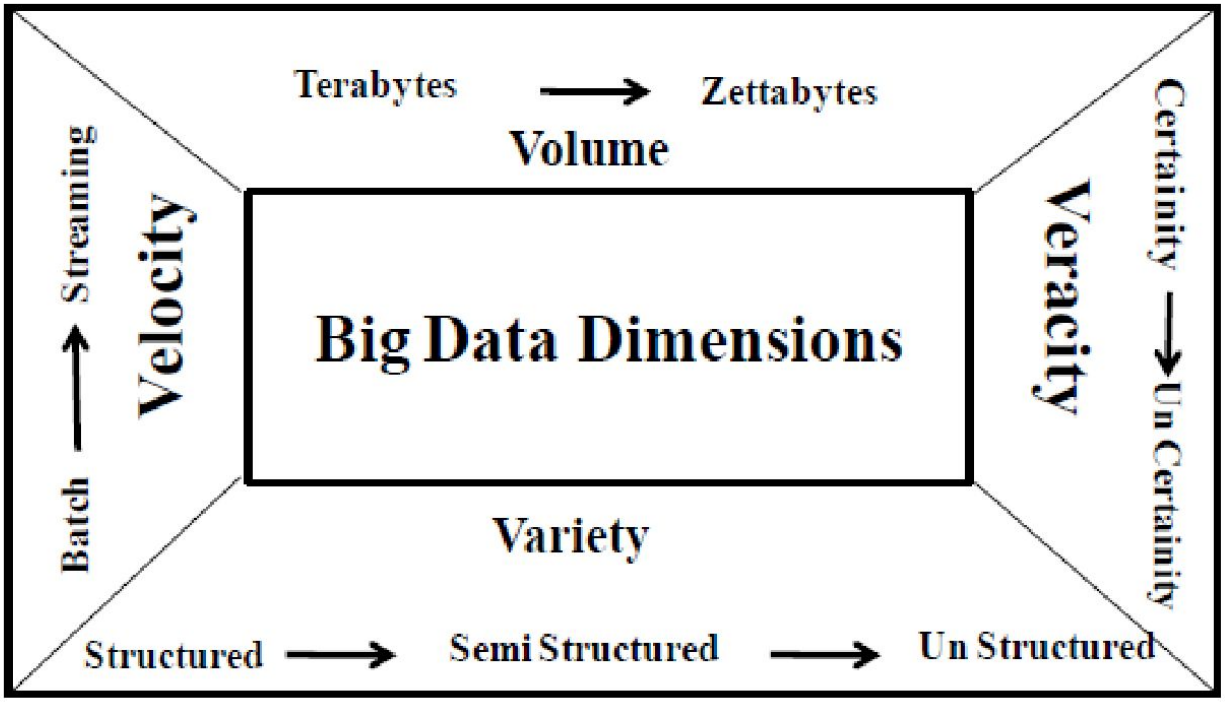
"About 78% of organizations now report using AI (including data/analytics) in at least one business function, but ~85% of large-scale data/AI projects fail due to poor alignment and technical challenges. And although most firms pilot initiatives, 74% struggle to scale and embed data/AI into core operations. These trends suggest that while the will to adopt is high, execution remains the real hurdle."

S. Suhothayan

Big Data

- the amount of data just beyond technology's capability to store, manage and process efficiently.
- data whose scale, diversity, and complexity require new architecture, techniques, algorithms, and analytics to manage it and extract value and hidden knowledge from it…

Data vs Big Data



The FOUR V's of Big Data

From traffic patterns and music downloads to web history and medical records, data is recorded, stored, and analyzed to enable the technology and services that the world relies on every day. But what exactly is big data, and how can these massive amounts of data be used?

As a leader in the sector, IBM data scientists break big data into four dimensions: **Volume, Velocity, Variety and Veracity**

Depending on the industry and organization, big data encompasses information from multiple internal and external sources such as transactions, social media, enterprise content, sensors and mobile devices. Companies can leverage data to adapt their products and services to better meet customer needs, optimize operations and infrastructure, and find new sources of revenue.

By 2015
4.4 MILLION IT JOBS
will be created globally to support big data, with 1.9 million in the United States

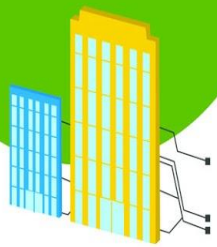
Volume
SCALE OF DATA

40 ZETTABYTES
[43 TRILLION GIGABYTES]
of data will be created by 2020, an increase of 300 times from 2005

It's estimated that
2.5 QUINTILLION BYTES
[2.3 TRILLION GIGABYTES]
of data are created each day

6 BILLION PEOPLE
have cell phones

WORLD POPULATION: 7 BILLION



Most companies in the U.S. have at least
100 TERABYTES
[100,000 GIGABYTES]
of data stored

Variety
DIFFERENT FORMS OF DATA

As of 2011, the global size of data in healthcare was estimated to be
150 EXABYTES
[161 BILLION GIGABYTES]

30 BILLION PIECES OF CONTENT
are shared on Facebook every month

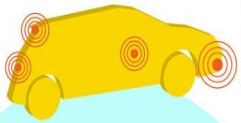
By 2014, it's anticipated there will be
420 MILLION WEARABLE, WIRELESS HEALTH MONITORS

4 BILLION+ HOURS OF VIDEO
are watched on YouTube each month

400 MILLION TWEETS
are sent per day by about 200 million monthly active users

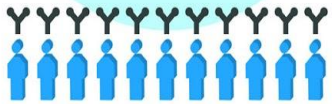
Velocity
ANALYSIS OF STREAMING DATA

The New York Stock Exchange captures
1 TB OF TRADE INFORMATION
during each trading session



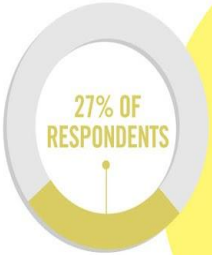
Modern cars have close to
100 SENSORS
that monitor items such as fuel level and tire pressure

By 2016, it is projected there will be
18.9 BILLION NETWORK CONNECTIONS
– almost 2.5 connections per person on earth



Veracity
UNCERTAINTY OF DATA

1 IN 3 BUSINESS LEADERS
don't trust the information they use to make decisions



in one survey were unsure of how much of their data was inaccurate

Poor data quality costs the US economy around
$3.1 TRILLION A YEAR

Sources: McKinsey Global Institute, Twitter, Cisco, Gartner, EMC, SAS, IBM, MEPTEC, QAS

IBM

A DAY IN DATA

The exponential growth of data is undisputed, but the numbers behind this explosion – fuelled by internet of things and the use of connected devices – are hard to comprehend, particularly when looked at in the context of one day

500m

tweets are sent every day

Twitter

4PB

of data created by Facebook, including

350m photos

100m hours of video watch time

Facebook Research

DEMYSTIFIYING DATA UNITS

From the more familiar 'bit' or 'megabyte', larger units of measurement are more frequently being used to explain the masses of data

Unit		Value	Size
b	bit	0 or 1	1/8 of a byte
B	byte	8 bits	1 byte
KB	kilobyte	1,000 bytes	1,000 bytes
MB	megabyte	$1{,}000^2$ bytes	1,000,000 bytes
GB	gigabyte	$1{,}000^3$ bytes	1,000,000,000 bytes
TB	terabyte	$1{,}000^4$ bytes	1,000,000,000,000 bytes
PB	petabyte	$1{,}000^5$ bytes	1,000,000,000,000,000 bytes
EB	exabyte	$1{,}000^6$ bytes	1,000,000,000,000,000,000 bytes
ZB	zettabyte	$1{,}000^7$ bytes	1,000,000,000,000,000,000,000 bytes
YB	yottabyte	$1{,}000^8$ bytes	1,000,000,000,000,000,000,000,000 bytes

*A lowercase "b" is used as an abbreviation for bits, while an uppercase "B" represents bytes.

463EB

of data will be created every day by 2025

IDC

95m

photos and videos are shared on Instagram

Instagram Business

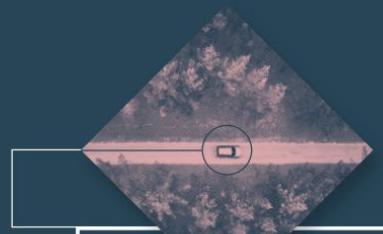
294bn

billion emails are sent

Radicati Group

320bn emails to be sent each day by 2021

306bn emails to be sent each day by 2020



65bn

messages sent over WhatsApp and two billion minutes of voice and video calls made

Facebook

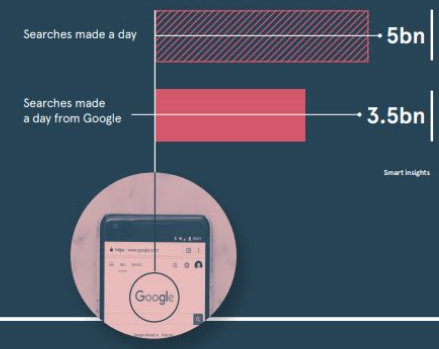
3.9bn

people use emails

4TB

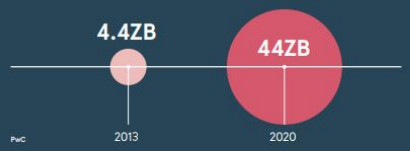
of data produced by a connected car

Intel



Smart Insights

ACCUMULATED DIGITAL UNIVERSE OF DATA



PwC

28PB

to be generated from wearable devices by 2020

Statista

RACONTEUR

Hadoop

- First released in 2006 as and reached it's peak popularity between 2014-2017
- Main focuse on data analytics and storage
- Consists of manly 4 modules
    - **HDFS :** Hadoop Distributed File System is a file system that can run on commodity hardware with better throughput than traditional file systems.
    - **YARN :** “Yet Another Resource Negotiator” is used for task and resource management. scheduling jobs of a computation cluster cluster.
    - **MapReduce :** Big data processing engine that supports the parallel computation of large data sets. It is the default processing engine available on Hadoop.
    - **Hadoop Common :** Hadoop Common provides a common set of libraries that can be used across all the other Hadoop modules

Contents

- Introduction to HDFS
- HDFS Components and Data Model
- HDFS Architecture
- HDFS Operations
- HDFS API and Example

Introduction to HDFS

Distributed File System (DFS)

- File system is responsible for
  - organization, storage, retrieval, naming, sharing and protection of files
  - controlling access to the data
  - performing low-level operations such as buffering frequently used data and issuing disk I/O requests
- Resource sharing is the motivation behind distributed systems
  - distributed file system allows sharing files
- Distributed file system allow users of physically distributed computers to share data and storage resources using a common file system

Hadoop Distributed File System HDFS

- Distributed file system written in Java based on the Google's GFS
- Provides redundant storage for massive amounts of data
- Very large DFS that works best with a smaller number of large files
  - millions as opposed to billions of files
  - typically 100MB or more per file
- Uses commodity hardware
  - files are replicated to handle hardware failure
  - HDFS detects failures and recovers from them
- Optimized for:
  - streaming reads of large files and not random reads
    - 10K nodes, 100 million files, 10PB large
  - batch processing where data locations are exposed so that computations can move to where data resides providing very high aggregate bandwidth

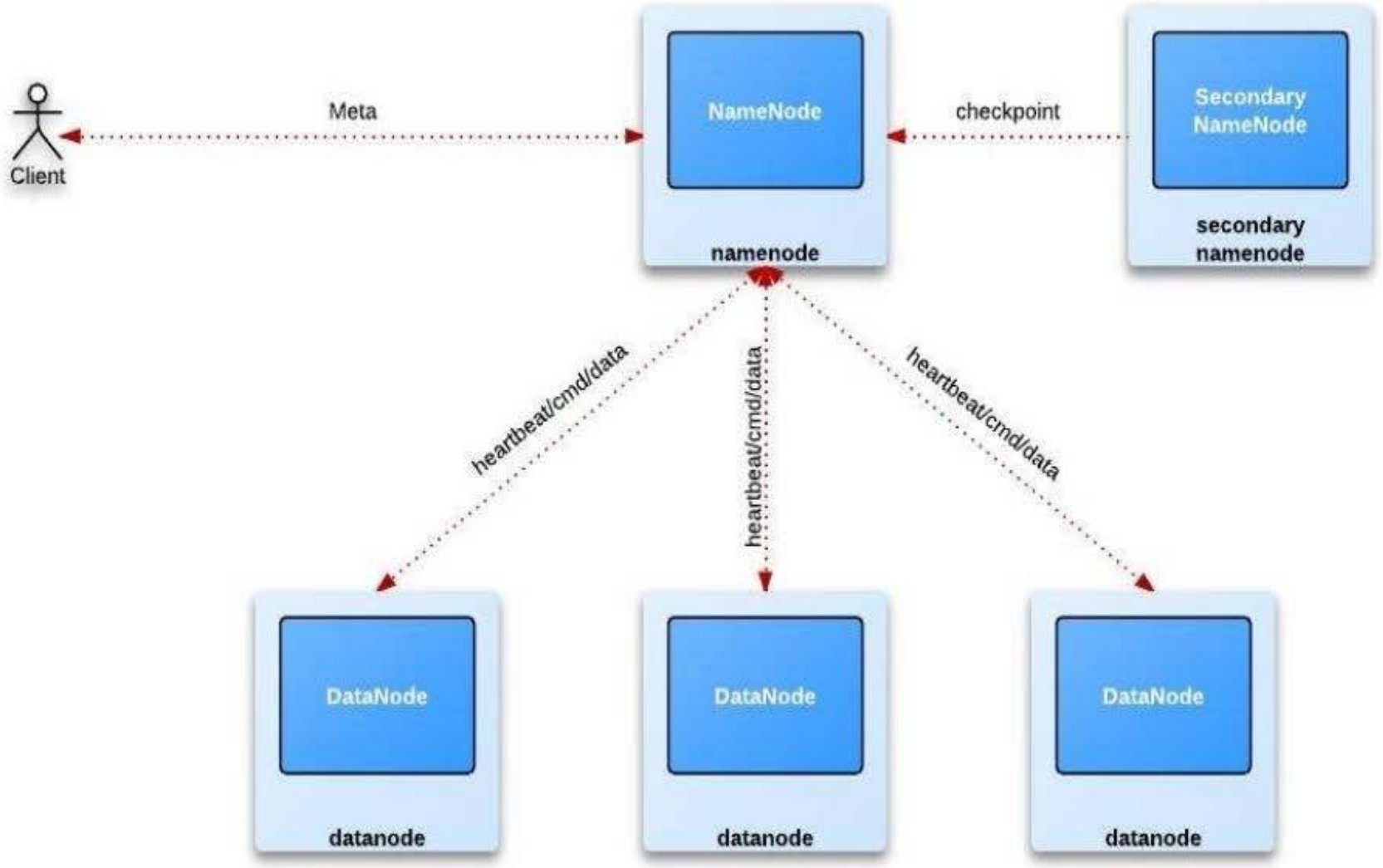
HDFS Characteristics

- Single cluster with computation and data
  - Processes huge amounts of data
- Scalable to store and process petabytes of data
- Economical:
  - It distributes the data and processing across clusters of commonly available computers,
  - clusters PCs into a storage and computing platform,
  - it makes more efficient use of available CPU resources, RAM on individual machines etc.
- Efficient:
  - by distributing the data, Hadoop can process it in parallel on the nodes where the data is located. This makes it extremely rapid
  - computation is moved to the data location
- Reliable and Fault Tolerant:
  - Hadoop automatically maintains multiple copies of data
  - automatically redeploys computing tasks based on failures.

HDFS Characteristic

- Architecture has a master-slave based architecture
  - Single NameNode to manage file system metadata slave nodes
  - multiple DataNodes for storing data
- Single Namespace for entire cluster
- Data coherency
  - write-once-read-many access model. why write-once?
  - client can only append existing files due to HDFS is optimized for sequential data access
- Files are broken up into blocks and each block replicated on multiple DataNodes
- Accessed through intelligent client that
  - can find location of blocks
  - can access data directly from DataNode

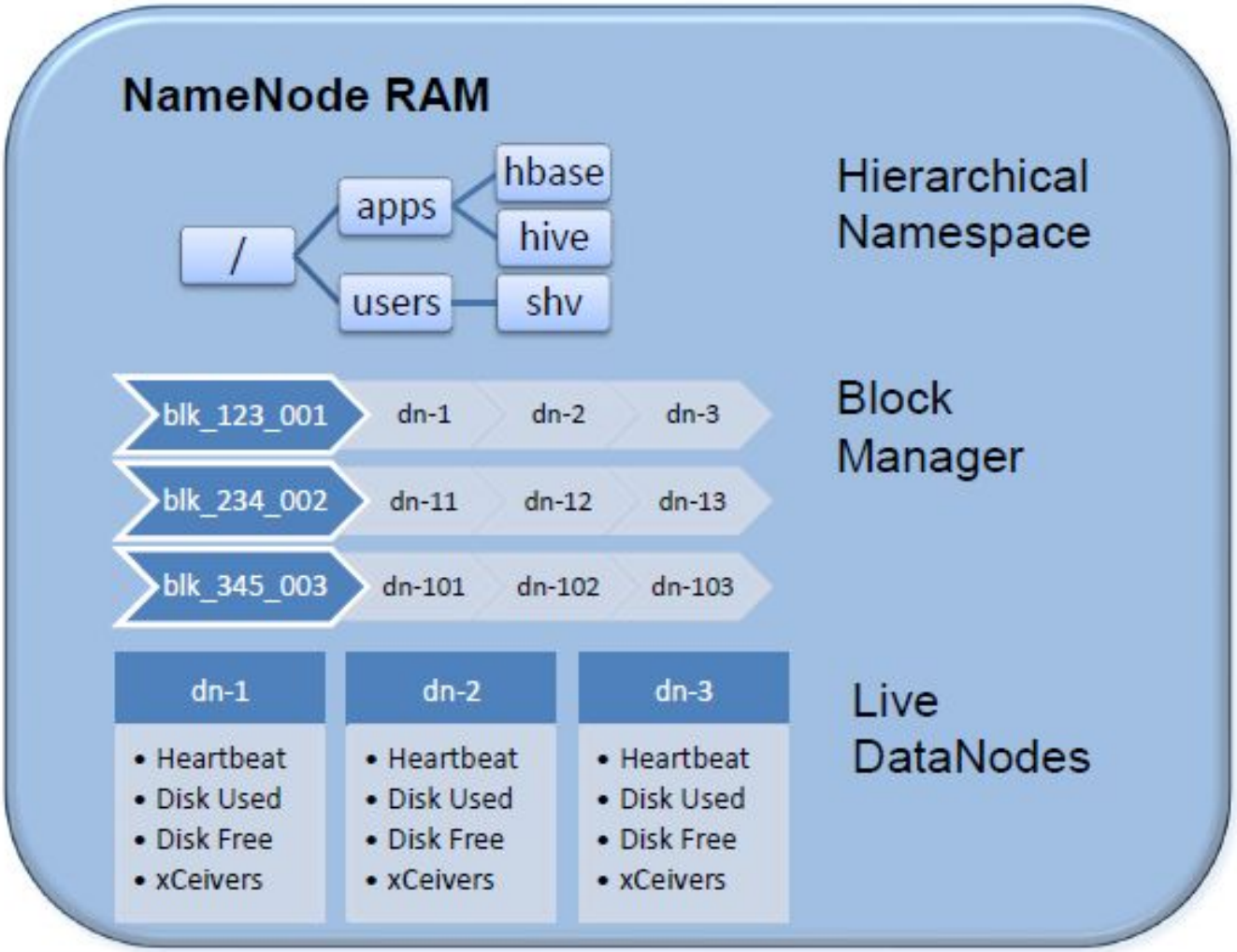
HDFS Architecture



HDFS Components: NameNode

- Mange cluster configuration
  - maps a file to a set of blocks
  - maps a block to the DataNodes where it resides
- Controls read/write access to files
- Detects DataNode failures
  - handles block replication choosing new DataNodes for new replicas
  - checkpoints namespace and journals namespace changes for reliability
  - periodic checkpointing is done to support system recovery back to the last checkpointed state in case of a crash
- Maintain the balance of,
  - Disk usage
  - balances communication traffic to DataNodes
- Keeps image of entire file system namespace and file block map in memory
- 4GB of local RAM is sufficient to support the above data structures
  - that can represent the huge number of files and directories
  - approximatly 2.75PB

HDFS Components : NameNode

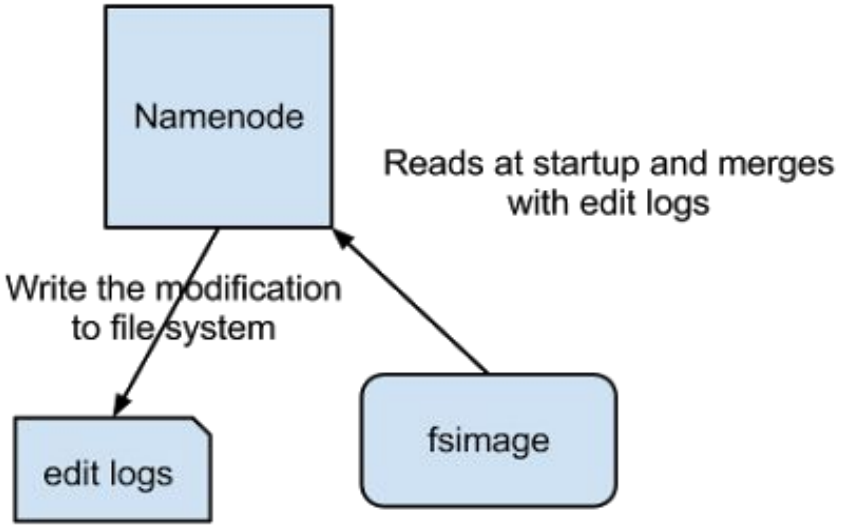


HDFS Components: NameNode Metadata

- Types of HDFS metadata
  - list of files
  - list of Blocks for each file
  - list of DataNodes for each block
- Metadata is stored in the main memory to guarantee fast access
- ***fsimage*** file
  - stores the entire file system namespace including mapping of blocks to files and file system properties, i.e. file system metadata
  - stored in NameNode's local file system
- ***edit logs*** file
  - NameNode uses a transaction log (edit logs) to record every change that occurs to the filesystem metadata.
  - for example: creating a new file, deleting a file, changing replication factor of a file, etc.
- Stored in the NameNode's local file system

HDFS Components: MetaData

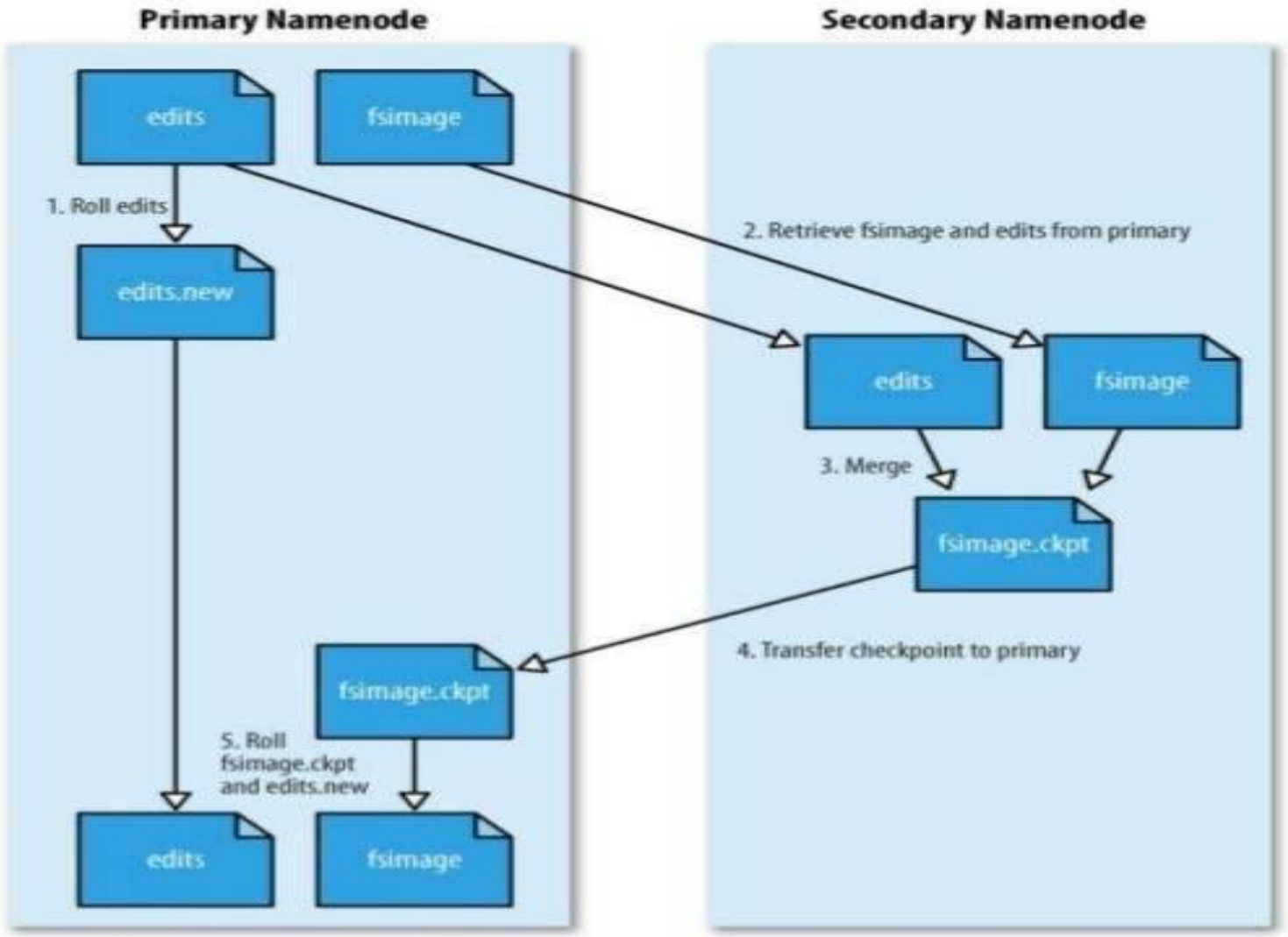
- When a Hadoop application is starting,
  - NameNode loads the fsimage file, which stores the HDFS file system information,
  - and the edit logs file from its local file system
  - then stores a copy of the fsImage on the file system as a checkpoint
- Updates to the file system (for example add/remove blocks) do not change the fsimage file, they are written to the edits log file



HDFS Components: Secondary NameNode

- **Update the fsimage file**
  - Periodically reads the edit logs file and applies the changes to the fsimage file bringing it up to date
- **Copy the new fsimage to the primary NameNode**
  - Copies fsimage and edit logs file from NameNode and merges them into a new fsimage file stored in a temporary directory
- **Purge edit logs in NameNode**
  - Edit logs on NameNode is purged after the merge operation
- **Upload new fsimage file to the NameNode**
  - Uploads new fsimage to the NameNode to allow the NameNode to restart faster when required
- Secondary NameNode is not a backup of the primary NameNode

HDFS Components: Secondary NameNode



HDFS Components : DataNode

- Serves read/write requests from clients
- Stores each block of HDFS data in a separate file in its local file system
- Perform replication tasks upon instruction by the NameNode
- Have no knowledge about the whole HDFS filesystem
- Uses heuristics to determine optimal number of files per directory and creates directories appropriately
- Generates a list of all HDFS blocks and send this report to NameNode in a Block Report when the filesystem starts up it

HDFS Components : DataNode Responsibilities

- **Block Server**
  - stores data in the local file system
  - stores metadata of a block (e.g. CRC - Cyclic Redundancy Check for integrity)
  - serves data and metadata to clients
- **Block Report:**
  - Periodically sends a report of all existing blocks to the NameNode
- **Heartbeat Manager**
  - Periodically sends heartbeat to NameNode to detect node failures
- **Facilitator of Data Pipelining**
  - forwards data to other specified DataNodes

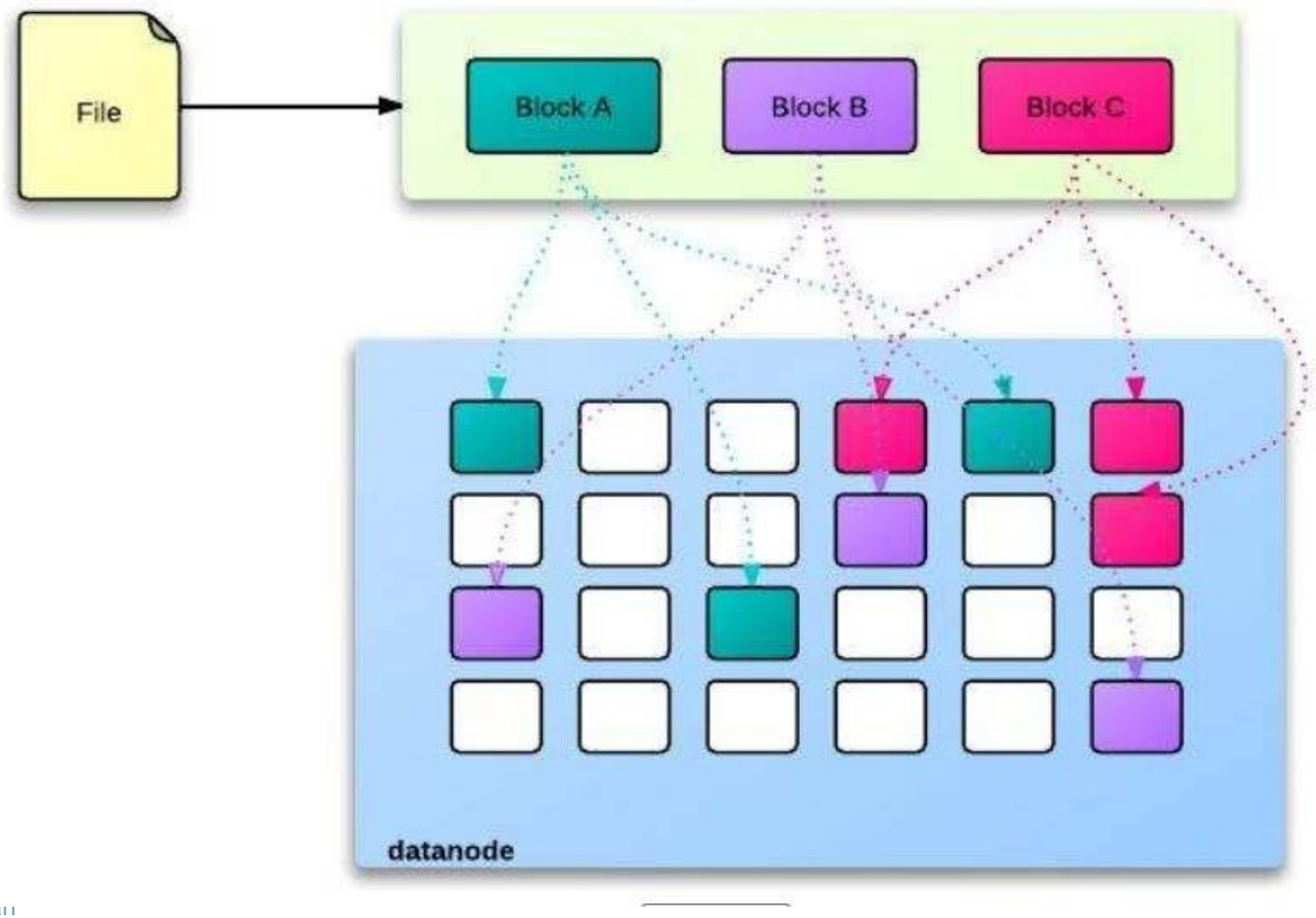
HDFS Data Model

- Data is organized into **hierarchical file system** with files and directories
- Supports create, remove, move, rename etc, file operations
- Files and directories are represented by inodes on the NameNode where,
  - inodes contain attributes such as permission, access and modification time, etc.
- Files are divided into uniform sized blocks and distributed across the Hadoop cluster nodes
  - typically block size is 64 MB or 128 MB
  - expose block placement so that computation can be migrated near data
- Blocks are split across many DataNodes at load time,
  - Different blocks from the same file will be stored on different machines
- Blocks are replicated across multiple DataNodes

HDFS Data Model

- NameNode maintains the file system and keeps track of which blocks make up a file and where they are stored
- Any meta information changes to the file system recorded by the NameNode
- Handles hardware failures via,
  - **Replication**
    - replicate blocks are created to handle hardware failure
    - one replica on local node, second replica on a remote rack and third replica on local rack,
    - additional replicas are randomly placed
    - application can specify the number of replicas of the file needed as the replication factor of the file
    - the replication factor stored in the NameNode.
  - **Checksums**
    - checksums of data is generated for corruption detection and recovery
  - **Continuous operation**
    - continues operation as nodes / racks added / removed

HDFS Data Model



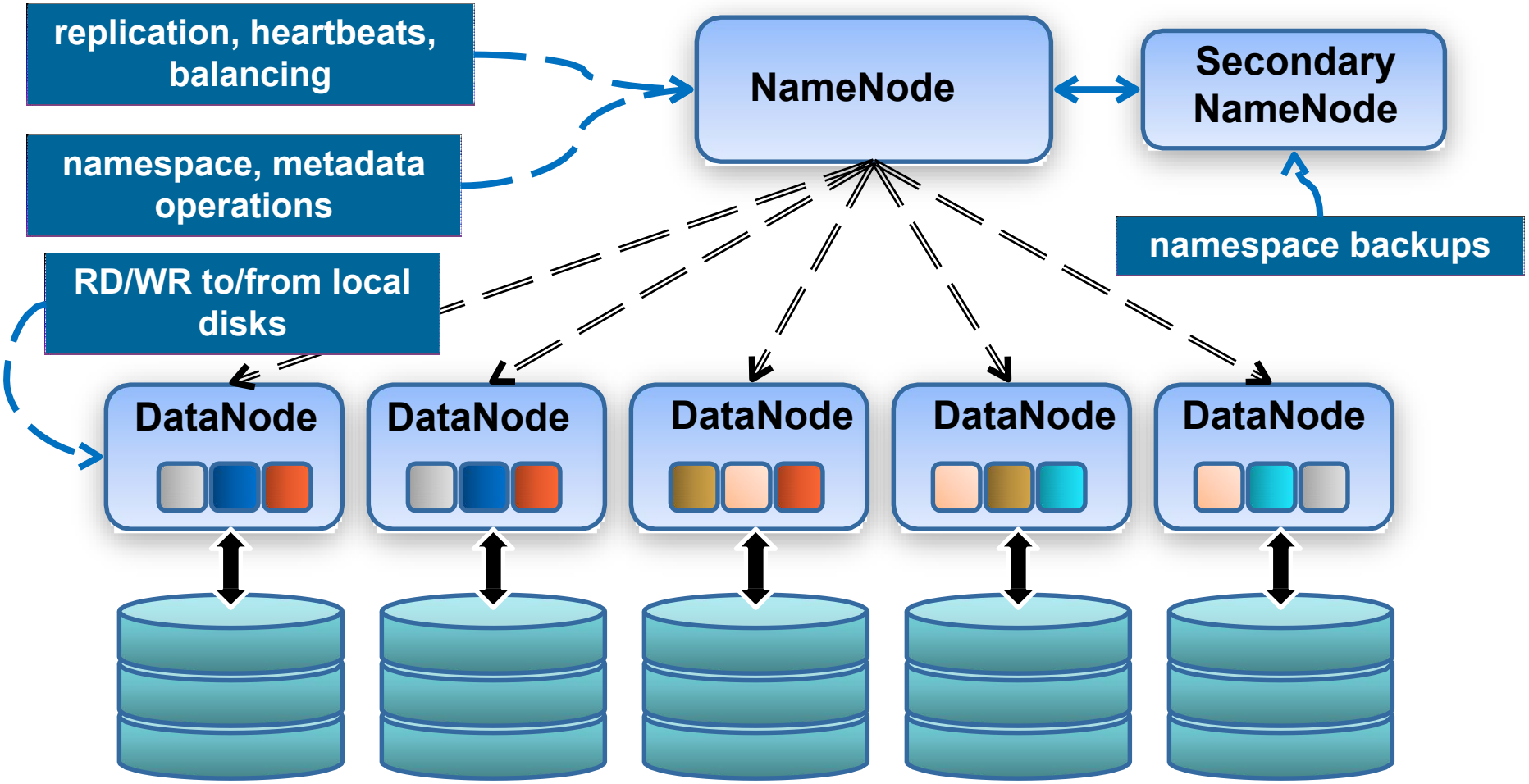
HDFS Data Model

- Optimized for fast batch processing
  - Data location exposed to allow to move computing to data
  - Stores data in chunks/blocks on every node in the cluster
  - Provides very high aggregate bandwidth
- Large streaming reads and small random reads
  - Running a MapReduce job on a large log file
  - Retrieving specific rows from a large CSV file (many seeks required)
- Enables multiple clients to append to a file
- Understands rack locality
  - Data placement exposed so that computation can be migrated to data
- Client talks to both NameNode and DataNodes
  - Data is not sent through the NameNode but clients access data directly from DataNode
  - Throughput of file system scales nearly linearly with the number of nodes.

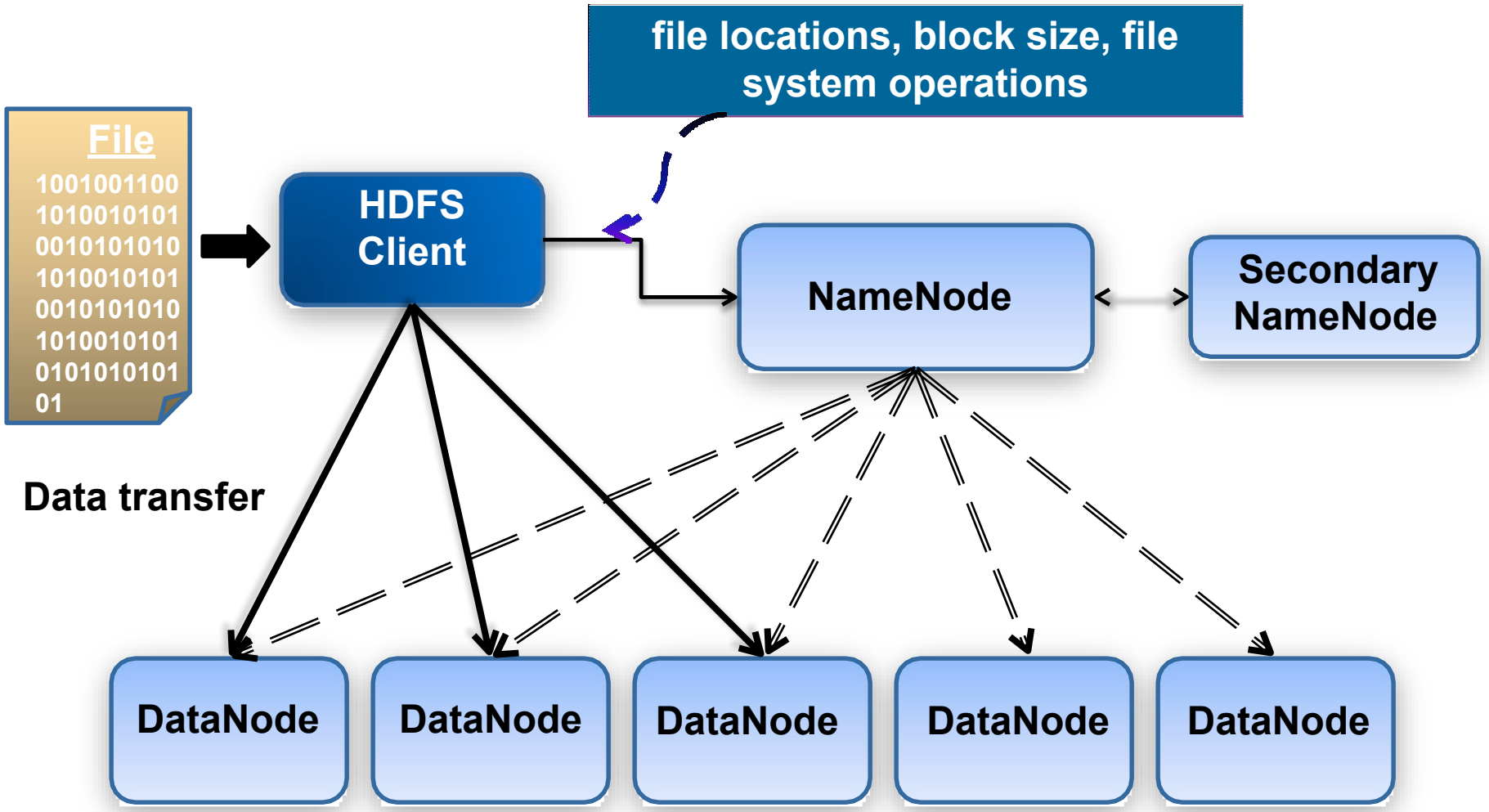
HDFS Architecture and Basic Operations

S. Suhothayan

HDFS Architecture



HDFS Architecture



HDFS Setting up a Cluster

Startup and SafeMode:

- When the **NameNode starts**, it enters **SafeMode**.
- **SafeMode** is a special state where the NameNode is in a read-only mode and **does not allow changes to the filesystem**, such as block replication or modifications to metadata.
- **Goal of SafeMode**: The purpose of SafeMode is to allow the NameNode to establish communication with the **DataNodes** and to verify the health and replication status of the file system's blocks.

DataNode Heartbeats and BlockReports:

- Each **DataNode** sends a **heartbeat** message to the NameNode to indicate that it is alive and functioning.
- Along with the heartbeat, the DataNodes send **BlockReports**. These reports provide the **NameNode** with a list of all blocks stored on each DataNode.
- This exchange allows the **NameNode** to track which DataNodes are up and which blocks they have.

Replication Verification in SafeMode:

- The **NameNode verifies the replication status** of all blocks as it receives BlockReports from DataNodes.
- Each block has a target replication factor (e.g., 3 replicas by default). The NameNode checks whether the **replication factor for each block** is sufficient.
- **In SafeMode**, block replication **does not occur**, meaning the NameNode will not instruct DataNodes to replicate blocks while still in SafeMode.

HDFS Setting up a Cluster Cont.

SafeMode Threshold:

- **SafeMode exit is based on a threshold**. The NameNode waits until a **configurable percentage** (e.g., 99.9%) of the blocks have been confirmed to have an acceptable number of replicas.
- This threshold can be configured using the `dfs.namenode.safemode.threshold-pct` parameter. Until this percentage of blocks are safe (i.e., meet their replication factor), the NameNode remains in SafeMode.

Exiting SafeMode:

- Once the threshold is met and a sufficient percentage of blocks are considered safe, the NameNode **exits SafeMode**.
- Exiting SafeMode means the NameNode resumes normal operation, allowing clients to write new data and modifying metadata.

Post-SafeMode Block Replication:

- After leaving SafeMode, the NameNode **analyzes blocks that are under-replicated**. These are blocks that do not meet their configured replication factor.
- The **NameNode initiates block replication** by instructing available DataNodes to create additional replicas of these under-replicated blocks on other DataNodes.
- The replication process is **asynchronous** and is managed by the NameNode as a background task.

HDFS Writing and Data Staging

Client Initiates File Creation:

- When an HDFS client requests to create a file, the request **does not immediately reach the NameNode** with the full data.
- Instead, the client first interacts with the **NameNode** to request the creation of a file, at which point the **NameNode only records the file metadata** in its namespace (i.e., it records the file path in the hierarchy but does not allocate any blocks yet).
- **Note**: The actual file data is not written at this stage. The NameNode simply reserves the file name and path.

Client Buffers Data Locally:

- The client caches the data into a **temporary buffer (typically 64KB)** in local memory (or disk) and starts writing to it.
- When the data reaches the size of an HDFS block (e.g., 128 MB by default), the client is ready to write the first block to HDFS.

Contacting NameNode for Block Allocation:

- Once the client has enough data to fill a block, it contacts the **NameNode** to allocate a block.
- The **NameNode allocates a block** for the file and assigns **DataNodes** to store the replicas of this block (based on the replication factor, typically 3).
- The **NameNode sends the client** the list of DataNodes where the block should be stored (including the primary DataNode and replica locations).

HDFS Writing and Data Staging Cont.

Client Writes Data to DataNodes:

- The client starts sending the block to the first DataNode. Once the first DataNode has received the block, it forwards it to the next DataNode, and so on, until the block is fully replicated across the assigned DataNodes.
- This process is called a **pipeline write**: The data flows through the DataNodes in a chain to ensure that each block is replicated according to the replication factor.
    a. Client contacts the first DataNode and sends the packet, the first node stores data and contacts the second DataNode and forwards the packet, and later contacts to the third DataNode
    b. Packet is removed from the ”ack queue” only when it has been acknowledged by all DataNodes in the pipeline
    c. If a DataNode fails while data is being written to it the partial data block will be deleted if the failed DataNode recovers later
    d. Client reports to the NameNode when the packet has been written

Client Acknowledgement:

- Once all DataNodes confirm that the block has been replicated successfully, the client moves on to the next block.
- The client keeps writing blocks until the entire file is written. Each block follows the same process: data is buffered locally, then the NameNode is contacted for block allocation, and the block is written to the DataNodes.

HDFS Writing and Data Staging Cont.

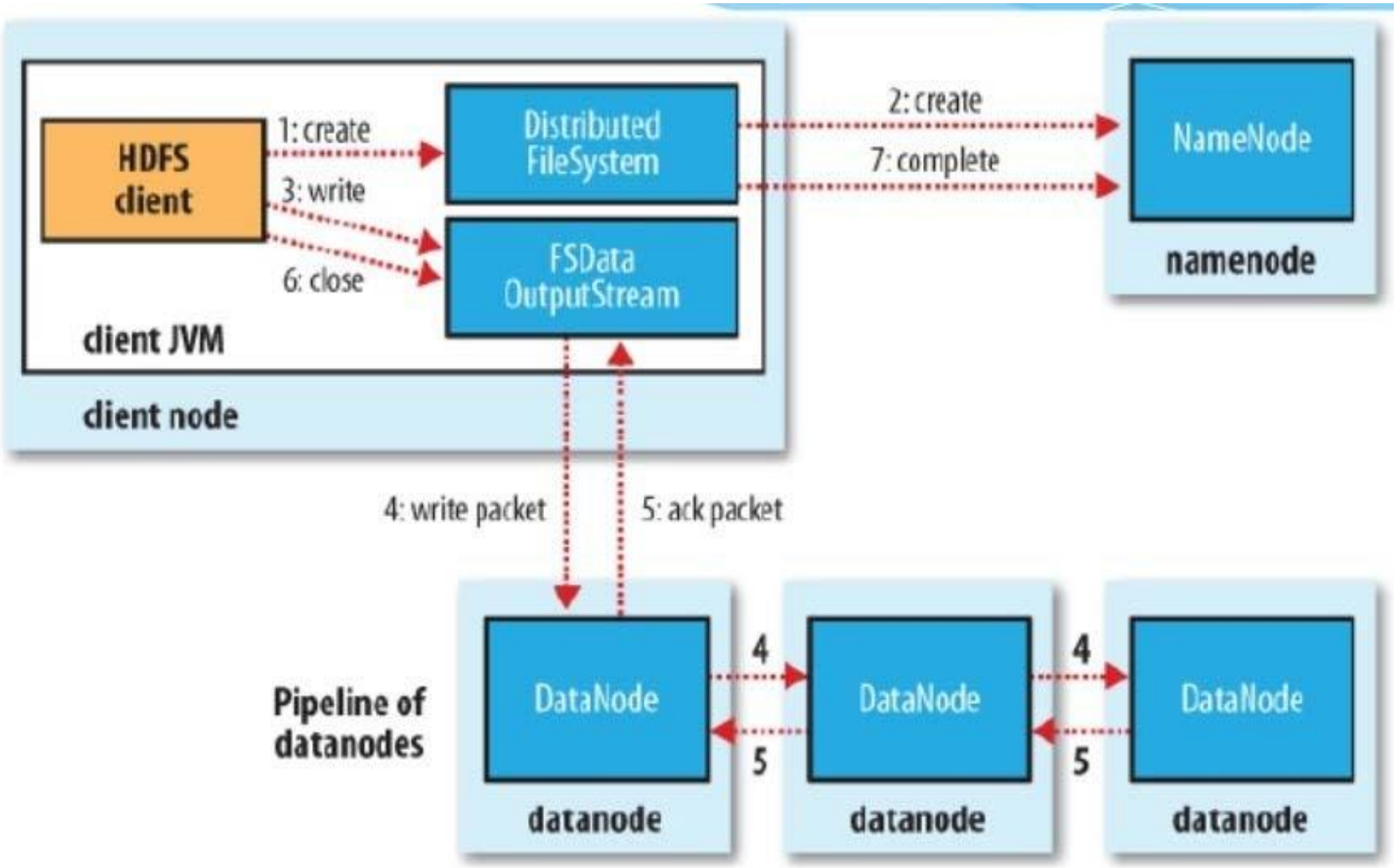
Closing the File:

- After all the blocks are written, the client sends a **close request** to the NameNode.
- The **NameNode finalizes the file** by committing the file's metadata (including its block locations) to the **in-memory namespace and edit log**. This ensures the file is properly recorded in the HDFS namespace.

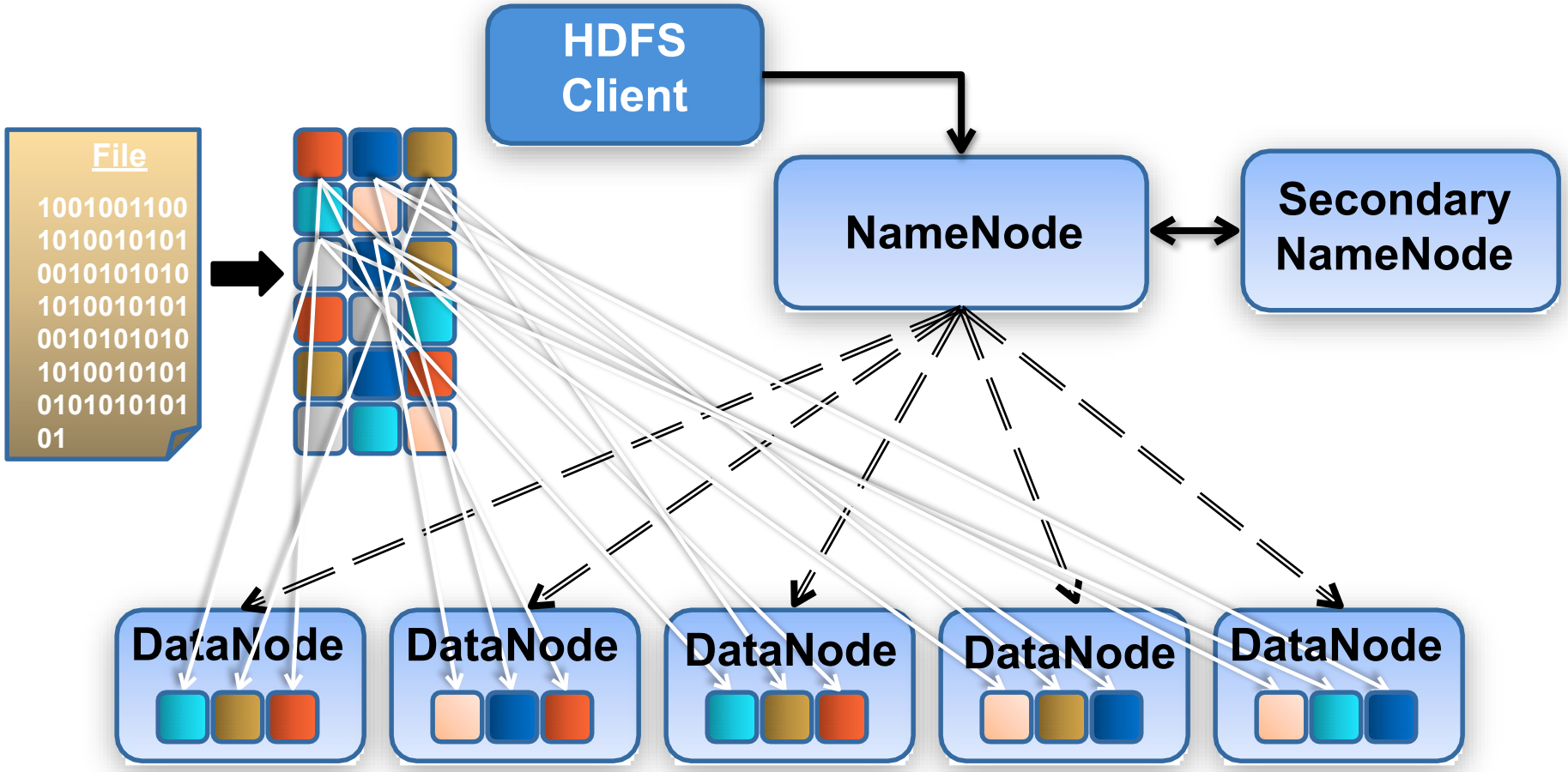
Data Loss Scenario:

- **If the NameNode crashes before the file is closed**, the file may be lost because it hasn't been fully committed. The file metadata is incomplete until the file is closed, and the **NameNode has not finalized the write**.
- Unclosed files or files with incomplete blocks will be discarded, as HDFS assumes that incomplete files (without proper closure) are corrupt.

HDFS Write Operation



HDFS Write Operation



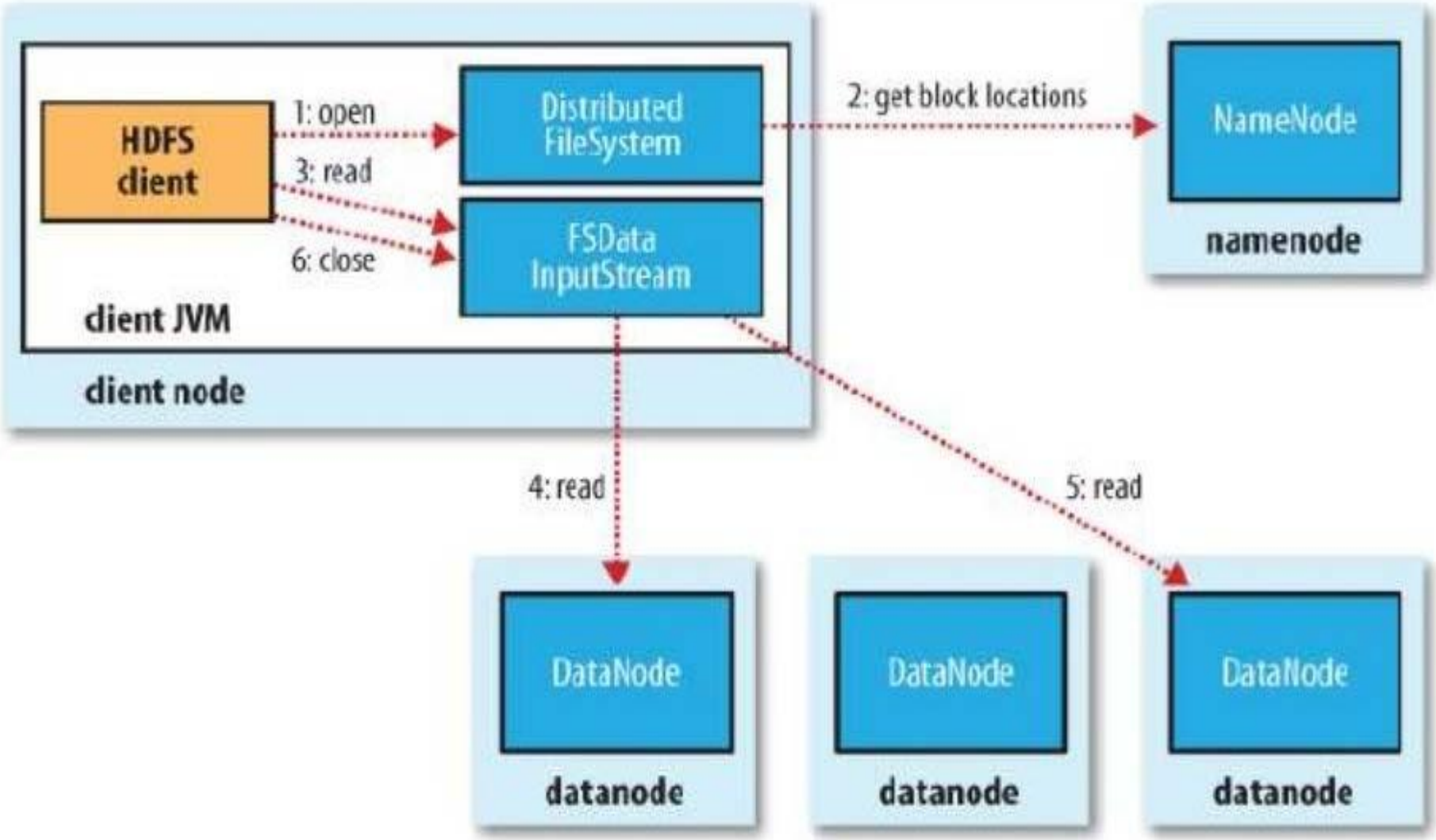
HDFS Read Operation

- **When a client wants to read data**
  - it must communicate with the NameNode to determine which blocks make up a file and on which DataNodes those blocks are stored
  - Having this information, client reads data **directly** with the DataNodes

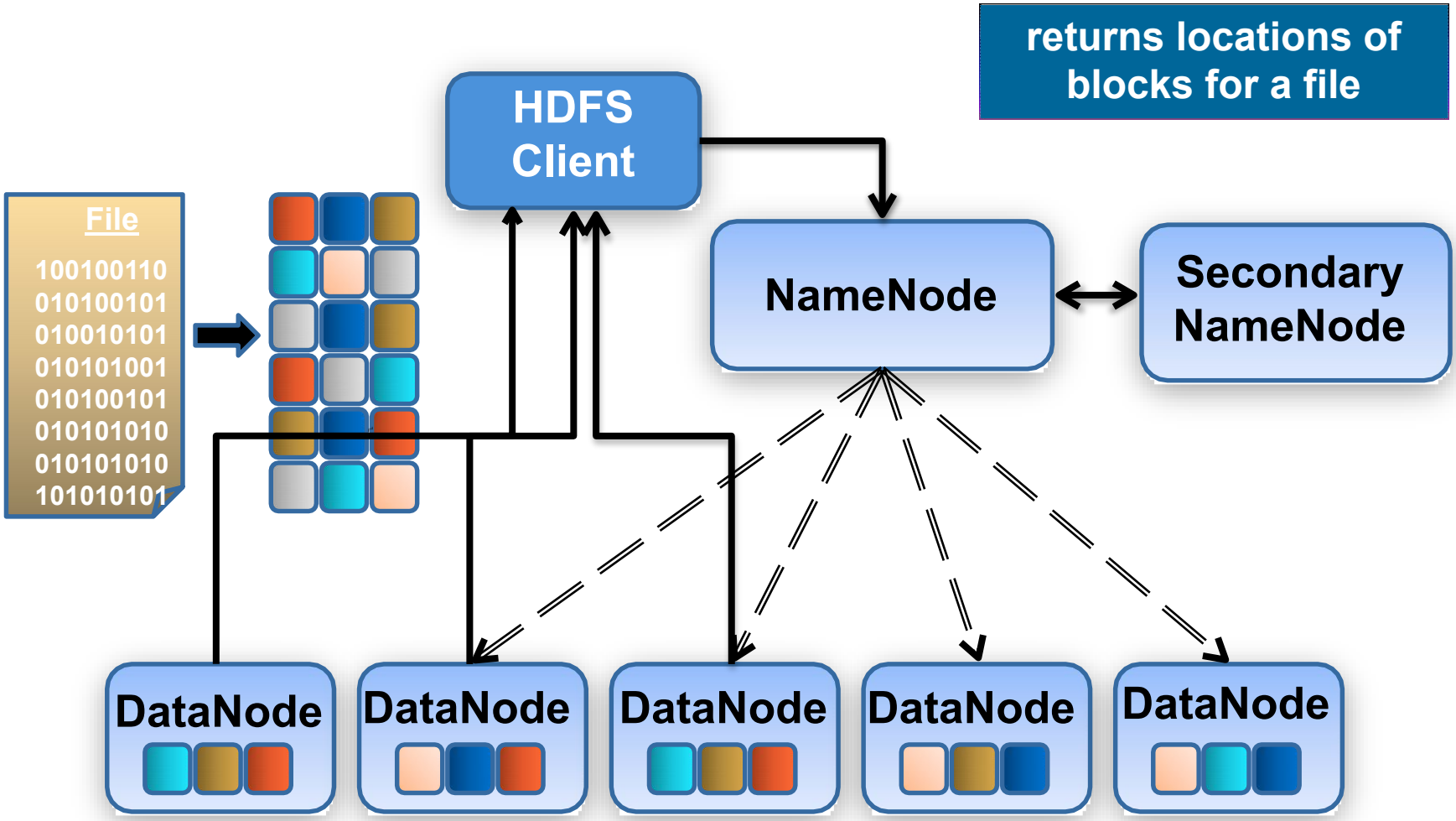
- **Read operation steps:**

1. Client connects to the NameNode with a file name
2. NameNode checks whether the file exists, whether the client has the required permission, etc.
3. NameNode returns locations of DataNodes that have the required block (locality is considered)
4. DataNodes, that contain the requested data, form a pipeline assuming that replication level is three, i.e. there are three DataNodes in the pipeline
5. Client contacts the first DataNode and reads the first block of the file, next the client finds the best DataNode for the second block, …. and completed the read operation
6. Client verifies checksums for each block transferred from the DataNode
7. If the Client finds an error communicating with a DataNode or a block is corrupted it will try to contact the next closest DataNode to retrieve the block

HDFS Read Operation



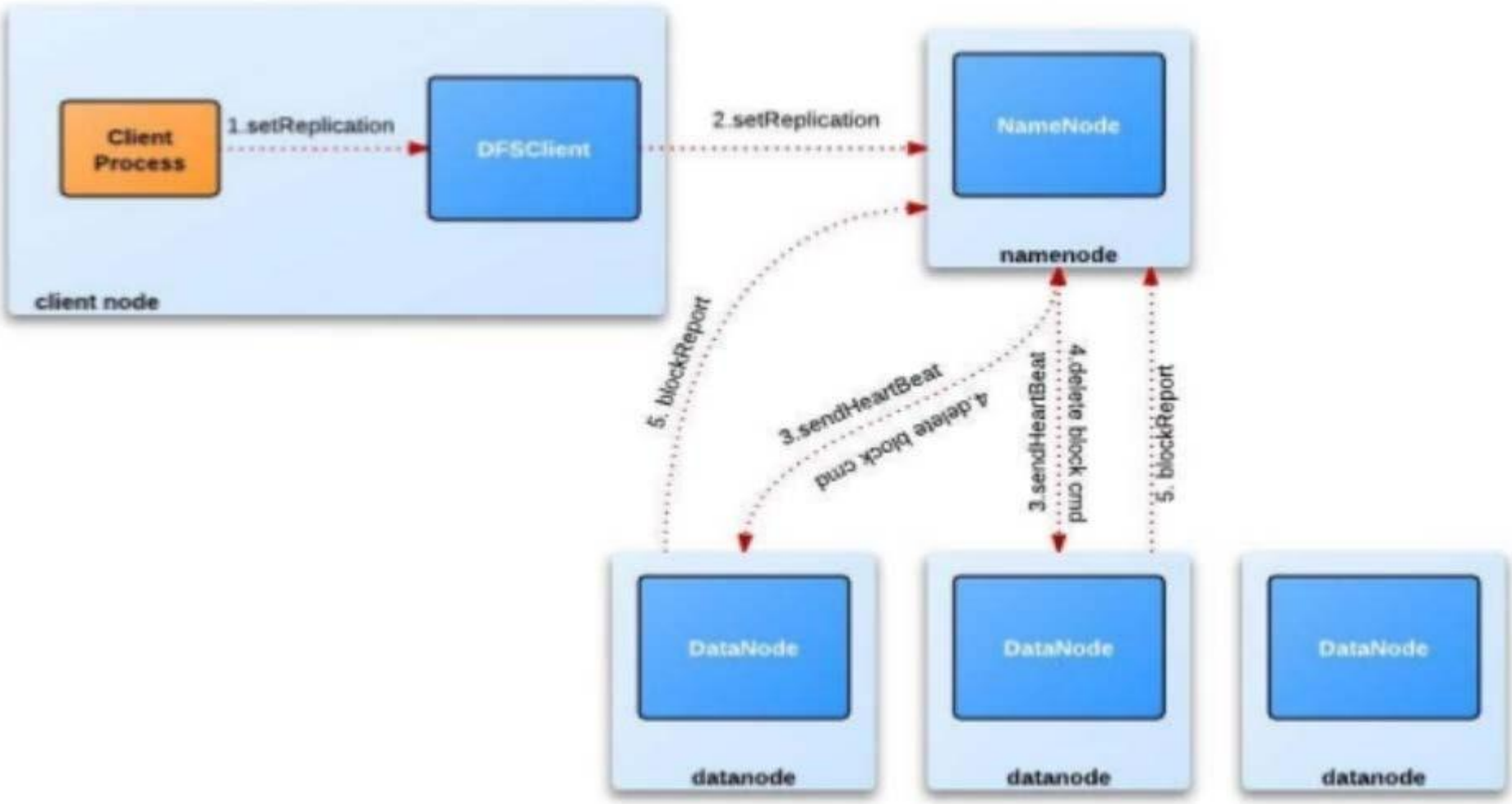
HDFS Read Operation



HDFS Replication

- HDFS is designed to store very large files across machines in a large cluster
- Each file is a sequence of blocks and all blocks in the file except the last are of the same size
- Blocks are replicated to provide fault tolerance
- Block size and replicas are configurable per file
- NameNode receives a Heartbeat and a BlockReport (contains all the blocks on a DataNode) from each DataNode in the cluster
  - **Replication selection**
    - Selecting replica for read operation HDFS tries to minimize the bandwidth consumption and latency
    - If there is a replica on the Reader node then that is preferred
    - HDFS cluster may span multiple data centers -> replica in the local data center is preferred over the remote
  - **Replication pipelining**
    - When the client receives response from a NameNode for replication, it flushes its block in small pieces (4KB) to the first replica, that in turn copies it to the next replica and so on to pipeline data from a DataNode to the next one

HDFS Replication



HDFS API

- Supports most common file and directory operations:
  - create, open, close, delete, read, write, seek, list, etc.
- Files are write once and have exclusively one writer
- There are specific HDFS operations:
  - set replication,
  - get block locations
- Provides supports for owners, permissions
- HDFS APIs
  - Java API (DistributedFileSystem)
  - C wrapper (libhdfs)
  - HTTP protocol
  - WebDAV protocol
  - Shell Commands

HDFS Shell Commands

User commands

- hdfs dfs – runs filesystem commands on the HDFS
- hdfs fsck – runs a HDFS filesystem checking command

Administration commands

- hdfs dfsadmin – runs HDFS administration commands

list directory contents

```
hdfs dfs -ls
hdfs dfs -ls /
hdfs dfs -ls -R /var
```

display the disk space used by files

```
hdfs dfs -du -h /
hdfs dfs -du /hbase/data/hbase/namespace/
hdfs dfs -du -h /hbase/data/hbase/namespace/
hdfs dfs -du -s /hbase/data/hbase/namespace/
```

HDFS Shell Commands

Copy data to HDFS

```
hdfs dfs -mkdir tdata
hdfs dfs -ls
hdfs dfs -copyFromLocal tutorials/data/geneva.csv tdata
hdfs dfs -ls -R
```

Copy the file back to local filesystem

```
cd tutorials/data/
hdfs dfs -copyToLocal tdata/geneva.csv geneva.csv.hdfs
md5sum geneva.csv geneva.csv.hdfs
```

HDFS Shell Commands

Removing a file

```
hdfs dfs -rm tdataset/tfile.txt
hdfs dfs -ls -R
```

list the blocks of a file and their locations

```
hdfs fsck /user/cloudera/tdata/geneva.csv -files -
blocks -locations
```

print missing blocks and the files they belong to

```
hdfs fsck / -list-corruptfileblocks
```

HDFS Shell Commands

list of acl of files

```
hdfs dfs -getfacl tdata/geneva.csv
```

list the file statistics – (%r – replication factor)

```
hdfs dfs -stat "%r" tdata/geneva.csv
```

Click to add text

write to hdfs reading from stdin

```
echo "blah blah blah" | hdfs dfs -put -
       tdataset/tfile.txt
hdfs dfs -ls -R
hdfs dfs -cat tdataset/tfile.txt
```

LAB

Starting the Docker Image

- Clone https://github.com/ramindu-msc/hadoop-docker-compose
  ```
  git clone https://github.com/ramindu-msc/iit
  ```
- Navigate to Run the docker image
  ```
  cd  iit/
  sudo chmod 755 lab1/
  sudo docker compose -f lab1/docker-compose.yaml up -d
  sudo docker ps
  ```
- Check the containers are started with exposed ports using
  ```
  docker ps
  ```

```
iitgcpuser@instance-20250930-170349:~/iit$ sudo docker ps
CONTAINER ID   IMAGE                                          COMMAND                  CREATED              STATUS                        PORTS
 NAMES
edf4a2bcee0b   bde2020/hadoop-nodemanager:2.0.0-hadoop3.2.1-java8       "/entrypoint.sh /run…"   About a minute ago   Up About a minute (healthy)   8042/tcp
 nodemanager1
fcc1e8ec1216   bde2020/hadoop-nodemanager:2.0.0-hadoop3.2.1-java8       "/entrypoint.sh /run…"   About a minute ago   Up About a minute (healthy)   8042/tcp
 nodemanager2
d5837314f900   bde2020/hadoop-resourcemanager:2.0.0-hadoop3.2.1-java8   "/entrypoint.sh /run…"   About a minute ago   Up About a minute (healthy)   0.0.0.0:8088->8088/tcp, [::]:8088->8088/tcp
 resourcemanager
ad0ff72a2af2   bde2020/hadoop-datanode:2.0.0-hadoop3.2.1-java8          "/entrypoint.sh /run…"   About a minute ago   Up About a minute (healthy)   9864/tcp
 datanode1
5701d51c6bb1   bde2020/hadoop-datanode:2.0.0-hadoop3.2.1-java8          "/entrypoint.sh /run…"   About a minute ago   Up About a minute (healthy)   9864/tcp
 datanode2
d4e87c8e890a   bde2020/hadoop-namenode:2.0.0-hadoop3.2.1-java8          "/entrypoint.sh /run…"   About a minute ago   Up About a minute (healthy)   0.0.0.0:9000->9000/tcp, [::]:9000->9000/tcp, 0.0.0.0:9870->9870/tcp, [::]:9870->9870/tcp
 namenode
```

- Check the cluster overview dashboard (Refresh if it gives you error messages after loading since it takes a bit of time to initialise the cluster) by navigating to (http://localhost:9870/dfshealth.html#tab-overview in your browser
- Access the shell of the docker container
  ```
  sudo docker exec -it namenode bash
  ```

Activities:

- Check your local working directory:
  ```
  pwd
  ```
- Check your HDFS working directory
  ```
  hadoop fs -ls /
  ```
  or
  ```
  hdfs dfs -ls /
  ```
- Create folder and Exit from the namenode
  ```
  mkdir /tmp/files
  exit
  ```
- Create a new local folder and name it “ml-100k” and Verify if the local folder has been created. Then verify the current directory.
  ```
  mkdir ml-100k
  ls
  pwd
  cd ml-100k
  ```
- Download the Movielens dataset from wget http://media.sundog-soft.com/hadoop/ml-100k/u.data to the ml-100k directory
  ```
  wget http://media.sundog-soft.com/hadoop/ml-100k/u.data
  ```

Activities: Cont.

- Display a sample data of the u.data downloaded
```
head u.data
```

- Copy the data set to your docker container
```
sudo docker cp /home/iitgcpuser/iit/ml-100k/u.data namenode:/tmp/files
```

- Create the HDFS directories and verify it's created
```
sudo docker exec -it namenode bash
hdfs dfs -mkdir -p /user/test/ml-100k/
hdfs dfs -ls /user/test/ml-100k/
```

- Then upload the u.data to the location
```
hdfs dfs -put /tmp/files/u.data /user/test/ml-100k/u.data
```

- now verify if the u.data has been successfully uploaded.
```
hdfs dfs -ls /user/test/ml-100k/u.data
```

- Check how the data is stored in the nodes
```
hdfs fsck /user/test/ml-100k/u.data -files -blocks -locations -racks
```