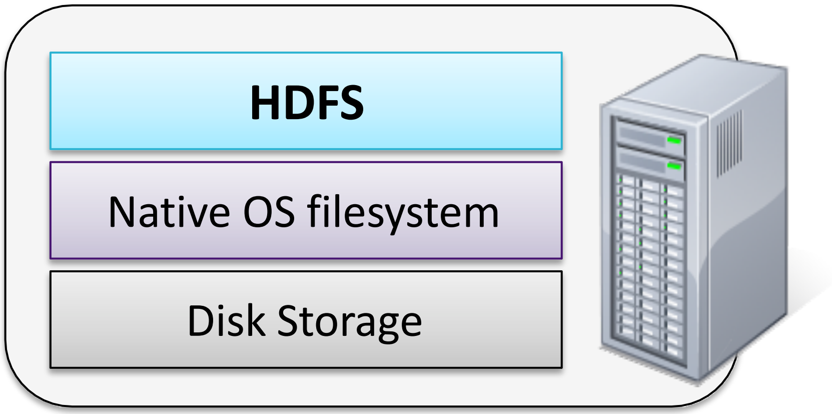
The Hadoop Distributed File System (HDFS)

HDFS: The Hadoop Distributed File System

- **HDFS is a filesystem written in Java**
  - Based on Google's GFS (Google File System)
- **Sits on top of a native filesystem**
  - Such as ext3, ext4, or xfs
- **Provides redundant storage for massive amounts of data**
  - Using industry-standard hardware
- **At load time, data is distributed across all nodes**
  - Provides for efficient processing



HDFS Features

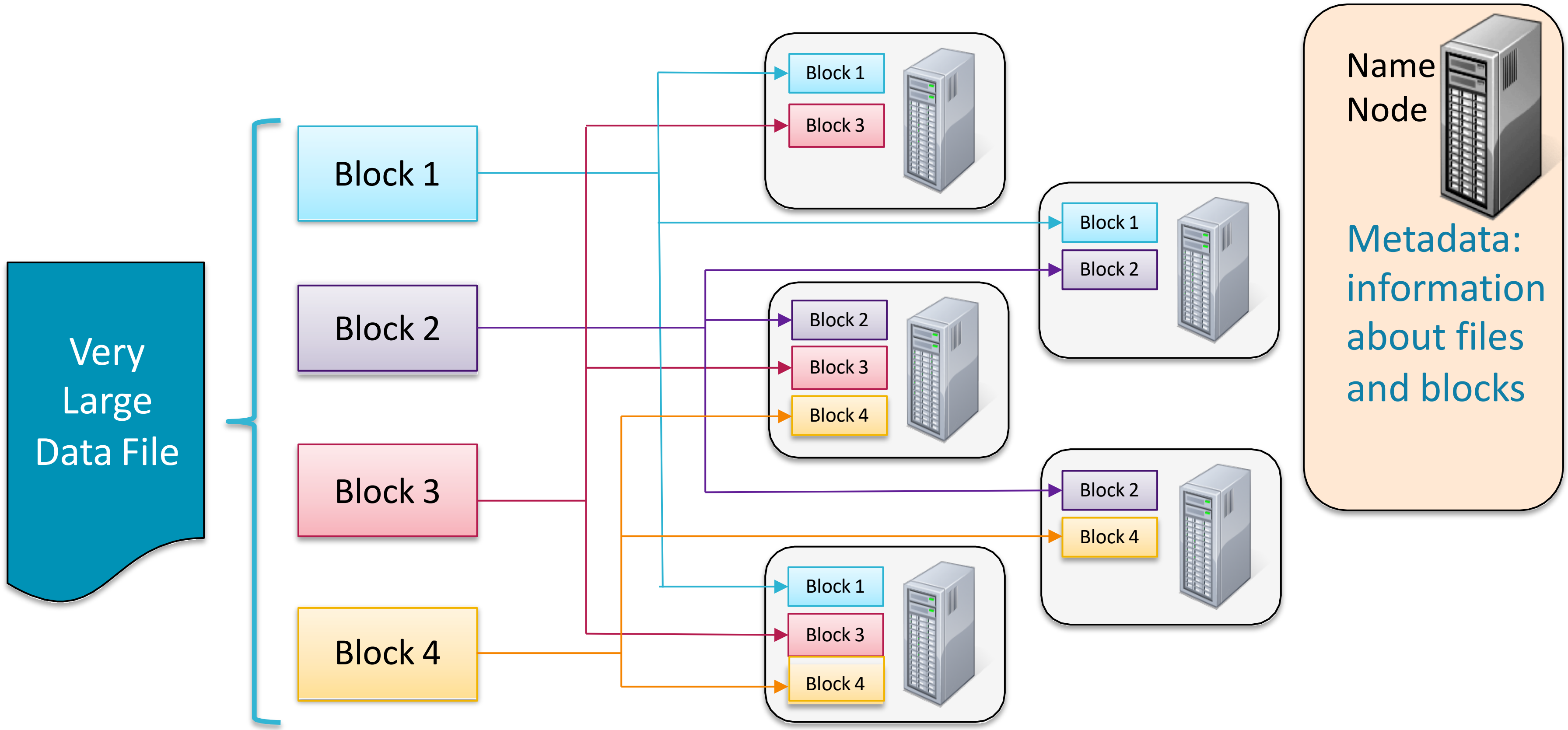
- **High performance**
- **Fault tolerance**
- **Relatively simple centralized management**
  - Master-worker architecture
- **Security**
  - Two levels from which to choose
- **Optimized for distributed processing**
  - Data locality
- **Scalability**

HDFS Design Assumptions

- **Components will fail**
- **“Modest” number of large files**
  - Millions of large files, not hundreds of millions of small files
  - Each file is likely to be 100MB or larger
    - Multi-gigabyte files typical
- **Files are write-once**
  - Data can be appended to a file, but the file’s existing contents cannot be changed
- **Large streaming reads**
  - Favor high sustained throughput over low latency

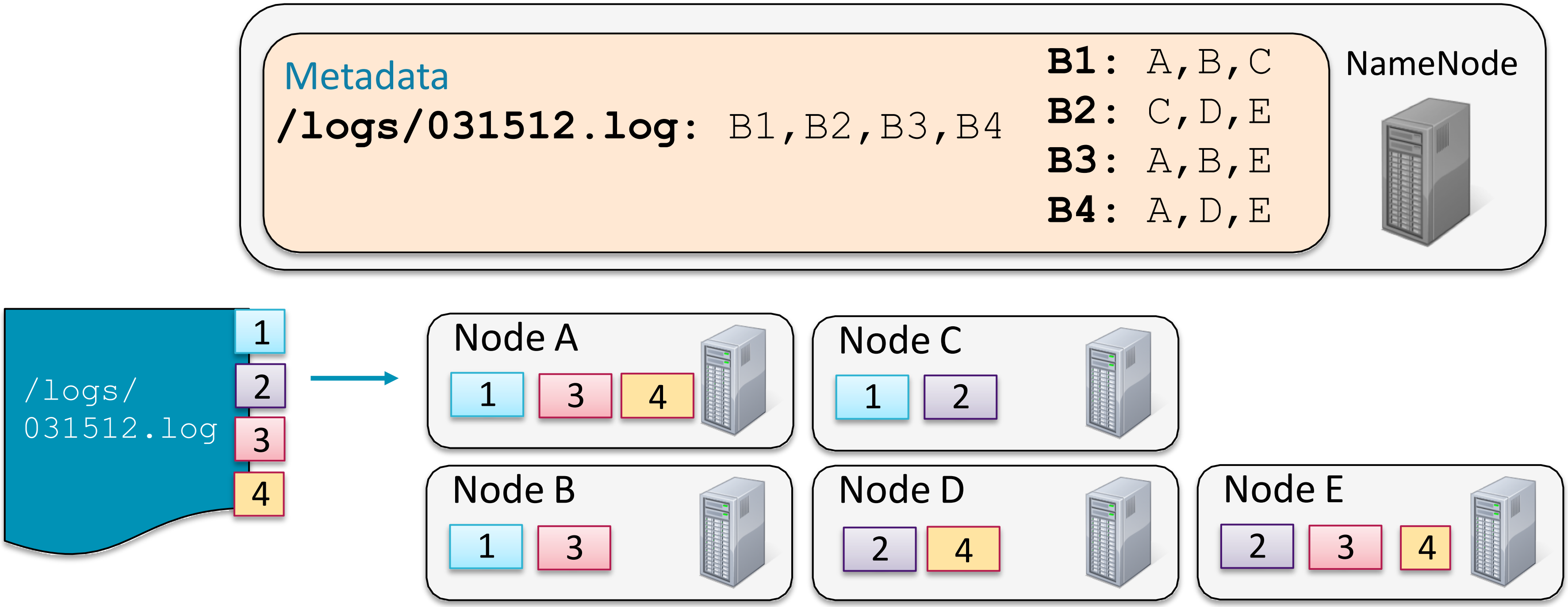
HDFS Blocks

- **When a file is added to HDFS, it is split into blocks**
  - Similar concept to native filesystems, but *much* larger block size
  - Default block size is 512MB (configurable)



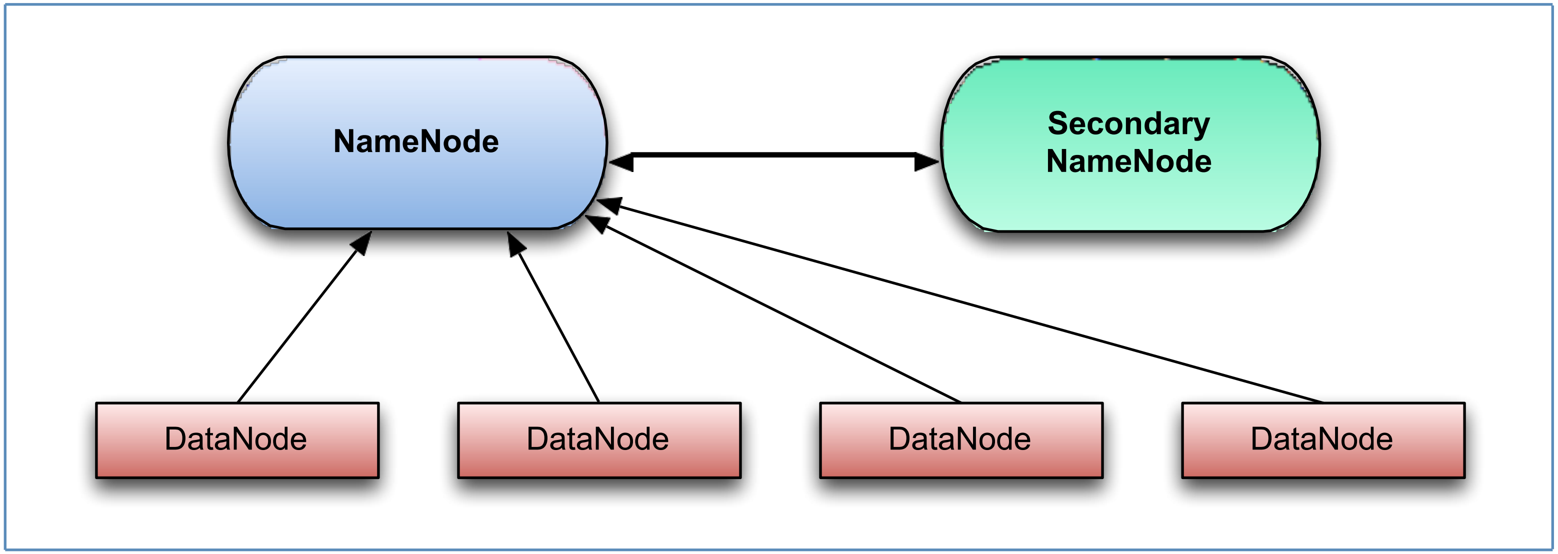
HDFS Replication

- **Blocks are replicated to nodes throughout the cluster**
  - Based on the *replication factor* (default is three)
- **Replication increases reliability and performance**
  - Reliability: data can tolerate loss of all but one replica
  - Performance: more opportunities for data locality



HDFS Without High Availability

- **You can deploy HDFS with or without high availability**
- **Without high availability, there are three daemons**
  - NameNode (master)
  - SecondaryNameNode (master)
  - DataNode (worker)



The HDFS NameNode

- **The NameNode holds all *metadata* in RAM**
  - Information about file locations in HDFS
  - Information about file ownership and permissions
  - Names of the individual blocks
  - Locations of the blocks
- **Metadata is stored on disk and read when the NameNode daemon starts up**
  - Filename is `fsimage`
- **Changes to the metadata are stored in RAM**
  - Changes are also written to an edits log

The Worker Nodes

- **Actual contents of the files are stored as *blocks* on the worker nodes**
- **Each worker node runs a DataNode daemon**
  - Controls access to the blocks
  - Communicates with the NameNode
- **Blocks are simply files on the worker nodes' underlying filesystem**
  - Named `blk_`*`xxxxxxx`*
  - Nothing on the worker node provides information about what underlying file the block is a part of
    - That information is *only* stored in the NameNode's metadata
- **Each block is stored on multiple different nodes for redundancy**
  - Default is three replicas

The HDFS SecondaryNameNode: Caution!

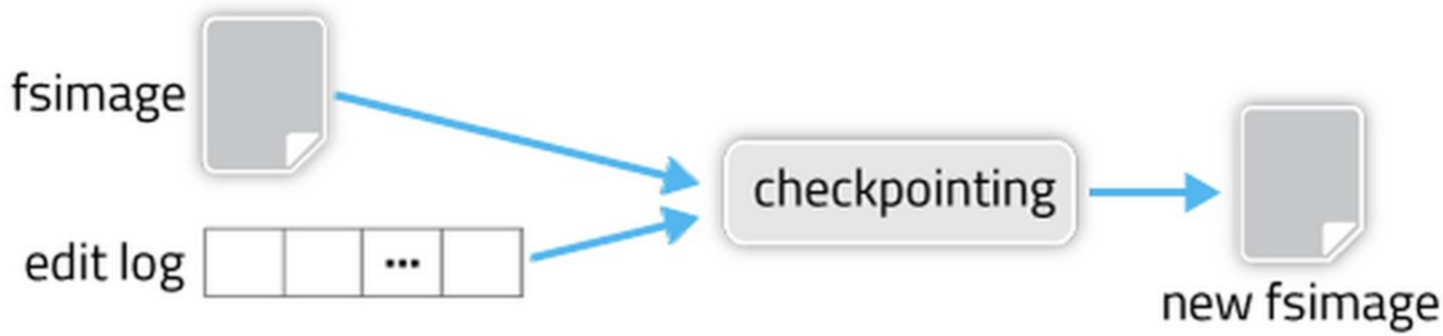
- **The SecondaryNameNode is *not* a failover NameNode!**
  - It performs memory-intensive administrative functions for the NameNode
    - NameNode keeps information about files and blocks (the *metadata*) in memory
    - NameNode writes metadata changes to an `edit` log
    - Secondary NameNode periodically combines a prior snapshot of the file system metadata and edit log into a new snapshot
    - New snapshot is transmitted back to the NameNode
- **SecondaryNameNode should run on a separate machine in a large installation**
  - It requires as much RAM as the NameNode
- **SecondaryNameNode only exists when high availability is not configured**

File System Metadata Snapshot and Edit Log

- **The `fsimage` file contains a file system metadata snapshot**
  - It is ***not*** updated at every write
- **HDFS write operations are recorded in the NameNode's edit log**
  - The NameNode's in-memory representation of the file system metadata is also updated
- **Applying all changes in the `edits` file(s) during a NameNode restart could take a long time**
  - The files could also grow to be huge

Checkpointing the File System Metadata

- **The SecondaryNameNode periodically constructs a checkpoint using this process:**
  1. Compacts information in the edits log
  2. Merges it with the most recent `fsimage` file
  3. Clears the edits log
- **Benefit: faster NameNode restarts**
  - The NameNode can use the latest checkpoint and apply the contents of the smaller edits log

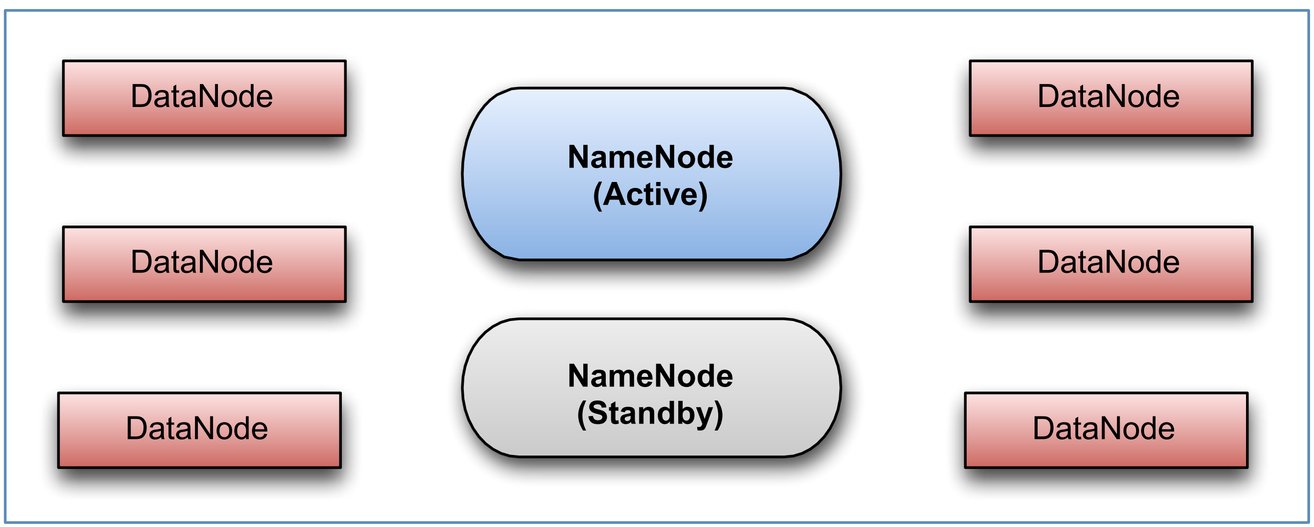


Single Point of Failure

- **In this mode of operation, each Hadoop cluster has a single NameNode**
  - The Secondary NameNode is *not* a failover NameNode
- **The NameNode is a single point of failure (SPOF)**
- **In practice, this is not a major issue**
  - HDFS will be unavailable until NameNode is replaced
  - There is very little risk of data loss for a properly managed system
- **Recovering from a failed NameNode is relatively easy**
  - We will discuss this process in detail later

HDFS With High Availability

- **Deploy HDFS with high availability to eliminate the NameNode SPOF**
- **Two NameNodes: one active and one standby**
  - Standby NameNode takes over when active NameNode fails
  - Standby NameNode also does checkpointing (SecondaryNameNode no longer needed)



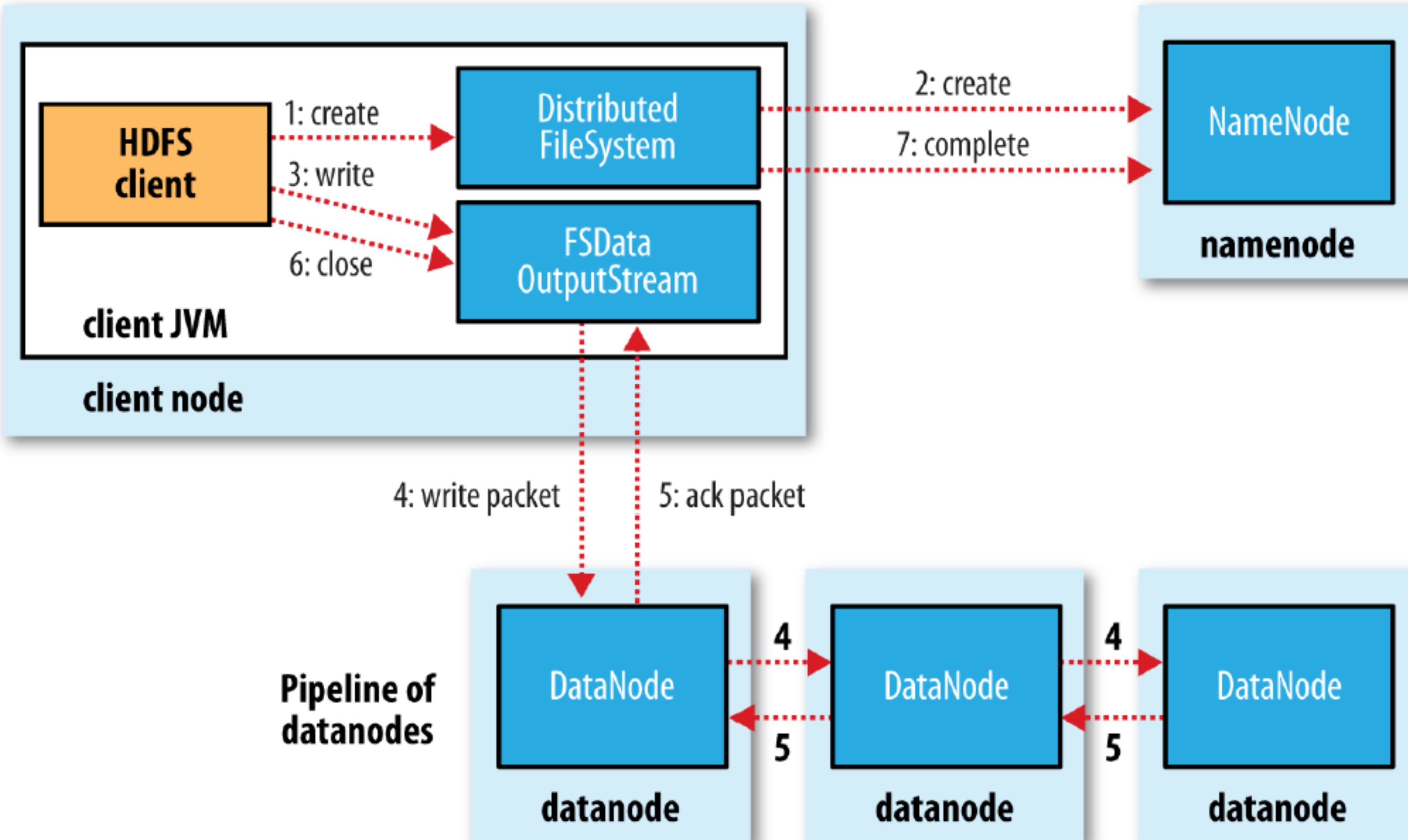
HDFS Read Caching

- **Applications can instruct HDFS to *cache* blocks of a file**
  - Blocks are stored on the DataNode in off-heap RAM
  - Can result in significant performance gains for subsequent reads
  - Most useful for applications where the same file will be read multiple times
- **It is possible for a user to manually cache files**
  - Impala is caching aware
    - Tables can be cached from within the Impala shell, or set to be cached when they are created
  - Caching-aware applications will attempt to read a block from the node on which it is cached
- **HDFS caching provides benefits over standard OS-level caching**
  - Avoids memory-to-memory copying

Configuring HDFS Read Caching

- **Cloudera Manager enables HDFS Read Caching by default**
- **Amount of RAM per DataNode to use for caching is controlled by `dfs.datanode.max.locked.memory`**
  - Can be configured per role group
  - Default is 4GB
  - Set to 0 to disable caching

Anatomy of a File Write (1)



Anatomy of a File Write (2)

1. **Client connects to the NameNode**
2. **NameNode places an entry for the file in its metadata, returns the block name and list of DataNodes to the client**
3. **Client connects to the first DataNode and starts sending data**
4. **As data is received by the first DataNode, it connects to the second and starts sending data**
5. **Second DataNode similarly connects to the third**
6. **ack packets from the pipeline are sent back to the client**
7. **Client reports to the NameNode when the block is written**

Anatomy of a File Write (3)

- **If a DataNode in the pipeline fails**
  - The pipeline is closed
  - A new pipeline is opened with the two good nodes
  - The data continues to be written to the two good nodes in the pipeline
  - The NameNode will realize that the block is under-replicated, and will re-replicate it to another DataNode
- **As blocks of data are written, the client calculates a checksum for each block**
  - Sent to the DataNode along with the data
  - Written together with each data block
  - Used to ensure the integrity of the data when it is later read

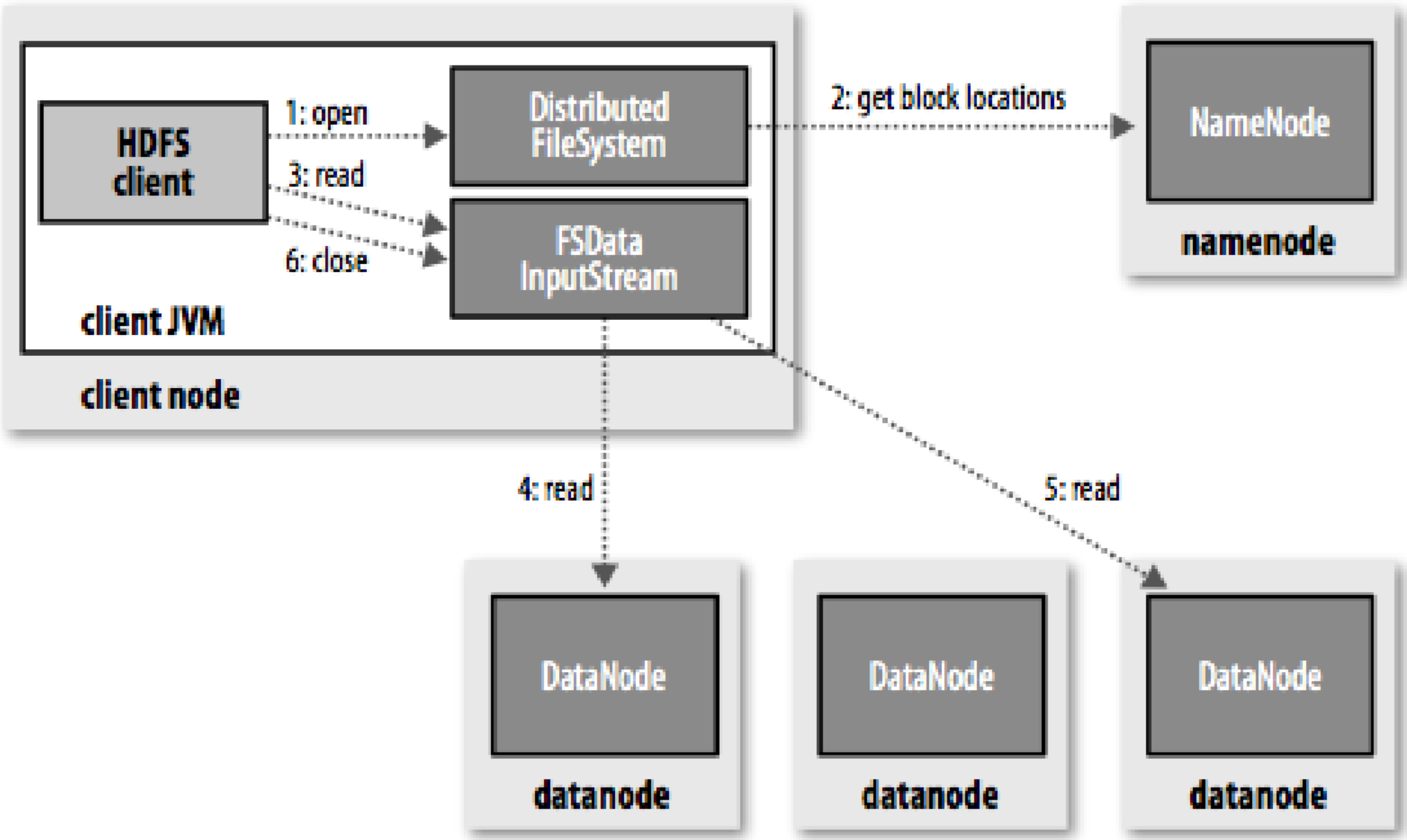
Hadoop is 'Rack-aware'

- **Hadoop understands the concept of 'rack awareness'**
  - The idea of where nodes are located, relative to one another
  - Helps the ResourceManager allocate processing resources on nodes closest to the data
  - Helps the NameNode determine the 'closest' block to a client during reads
  - In reality, this should perhaps be described as being 'switch-aware'
- **HDFS replicates data blocks on nodes on different racks**
  - Provides extra data security in case of catastrophic hardware failure
- **Rack-awareness is determined by a user-defined script**
  - Rack Topology configuration details through Cloudera Manager discussed later in this course

HDFS Block Replication Strategy

- **First copy of the block is placed on the same node as the client**
  - If the client is not part of the cluster, the first block is placed on a random node
    - System tries to find one which is not too busy
- **Second copy of the block is placed on a node residing on a different rack**
- **Third copy of the block is placed on different node in the same rack as the second copy**

Anatomy of a File Read (1)



Anatomy of a File Read (2)

1. **Client connects to the NameNode**
2. **NameNode returns the name and locations of the first few blocks of the file**
   - Block locations are returned closest-first
3. **Client connects to the first of the DataNodes, and reads the block**
   - If the DataNode fails during the read, the client will seamlessly connect to the next one in the list to read the block

Dealing With Data Corruption

- **As a client is reading the block, it also verifies the checksum**
  - 'Live' checksum is compared to the checksum created when the block was stored
- **If they differ, the client reads from the next DataNode in the list**
  - The NameNode is informed that a corrupted version of the block has been found
  - The NameNode will then re-replicate that block elsewhere
- **The DataNode verifies the checksums for blocks on a regular basis to avoid 'bit rot'**
  - Default is every three weeks after the block was created

Data Reliability and Recovery

- **DataNodes send *heartbeats* to the NameNode**
  - Every three seconds
- **After a period without any heartbeats, a DataNode is assumed to be lost**
  - NameNode determines which blocks were on the lost node
  - NameNode finds other DataNodes with copies of these blocks
    - These DataNodes are instructed to copy the blocks to other nodes
  - Three-fold replication is actively maintained
- **A DataNode can rejoin a cluster after being down for a period**
  - The NameNode will ensure that blocks are not over-replicated by instructing DataNodes to remove excess copies
    - Note that this does not mean all blocks will be removed from the DataNode which was temporarily lost!

The NameNode Is Not a Bottleneck

- **Note: the data *never* travels via a NameNode**
  - For writes
  - For reads
  - During re-replication

NameNode: Memory Allocation (1)

- **When a NameNode is running, all metadata is held in RAM for fast response**
- **Default Java Heap Size of the NameNode is 1GB**
  - At least 1GB recommended for every million HDFS blocks
- **Items stored by the NameNode:**
  - Filename, permissions, etc.
  - Block information for each block

NameNode: Memory Allocation (2)

- **Why HDFS prefers fewer, larger files:**
  - Consider 1GB of data, HDFS block size 128MB
  - Stored as 1 x 1GB file
    - Name: 1 item
    - Blocks: 8 items
    - <u>Total items in memory: 9</u>
  - Stored as 1000 x 1MB files
    - Names: 1000 items
    - Blocks: 1000 items
    - <u>Total items in memory: 2000</u>

HDFS File Permissions

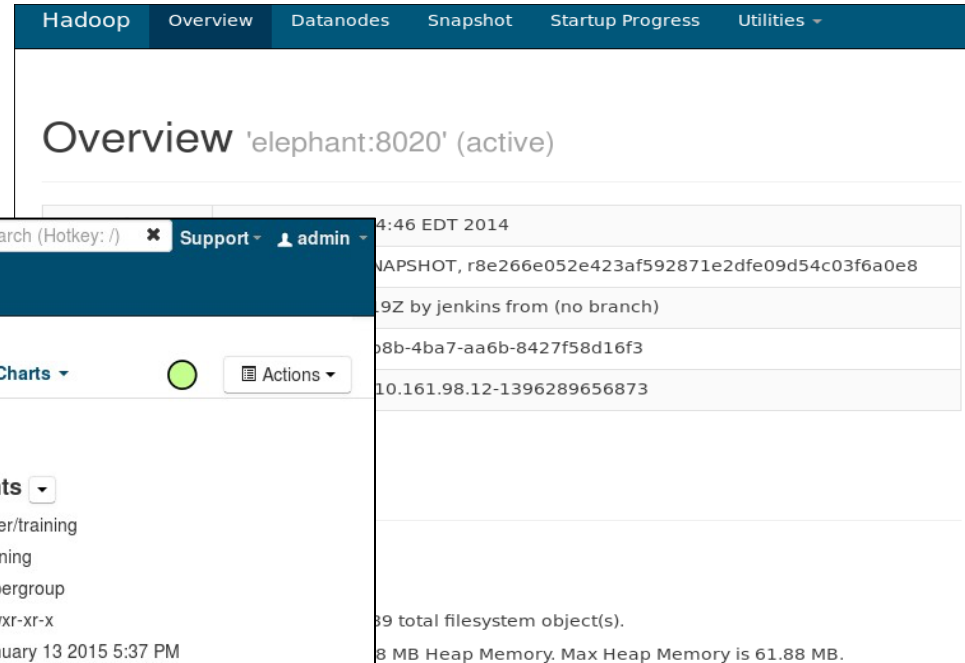
- **Files in HDFS have an owner, a group, and permissions**
  - Very similar to Unix file permissions
- **File permissions are read (`r`), write (`w`), and execute (`x`) for each of owner, group, and other**
  - `x` is ignored for files
  - For directories, `x` means that its children can be accessed
- **HDFS permissions are designed to stop good people doing foolish things**
  - Not to stop bad people doing bad things!
  - HDFS believes you are who you tell it you are

Hadoop Security Overview

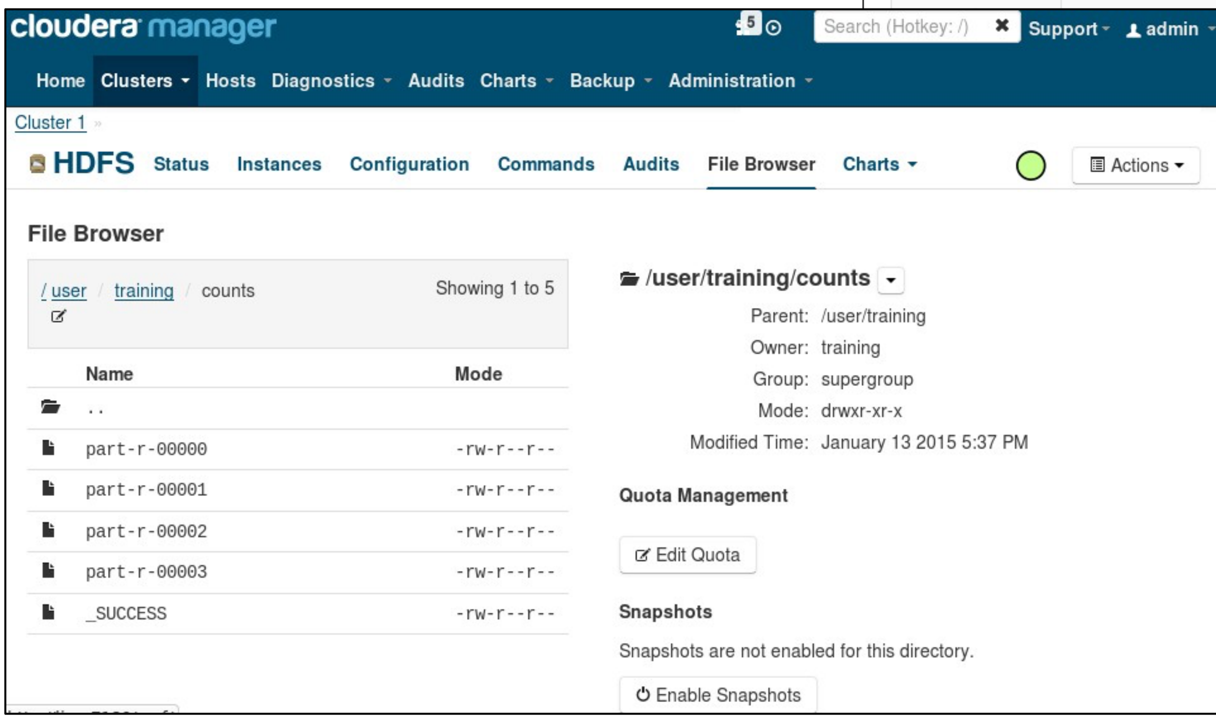
- **Authentication**
  - Proving that a user or system is who he or she claims to be
  - Hadoop can provide strong authentication control via Kerberos
    - Cloudera Manager simplifies Kerberos deployment
  - Authentication via LDAP is available with Cloudera Enterprise
- **Authorization (access control)**
  - Allowing people or systems to do some things but not other things
  - CDH has traditional POSIX-style permissions for files and directories
  - Access Control Lists (ACLs) for HDFS
  - Role-based access control provided with Apache Sentry
- **Data encryption**
  - OS filesystem-level, HDFS-level, and Network-level options
- **We will cover Hadoop security in more depth later**

Web UIs for HDFS

The NameNode Web UI on port 50070



The Cloudera Manager – HDFS File Browser

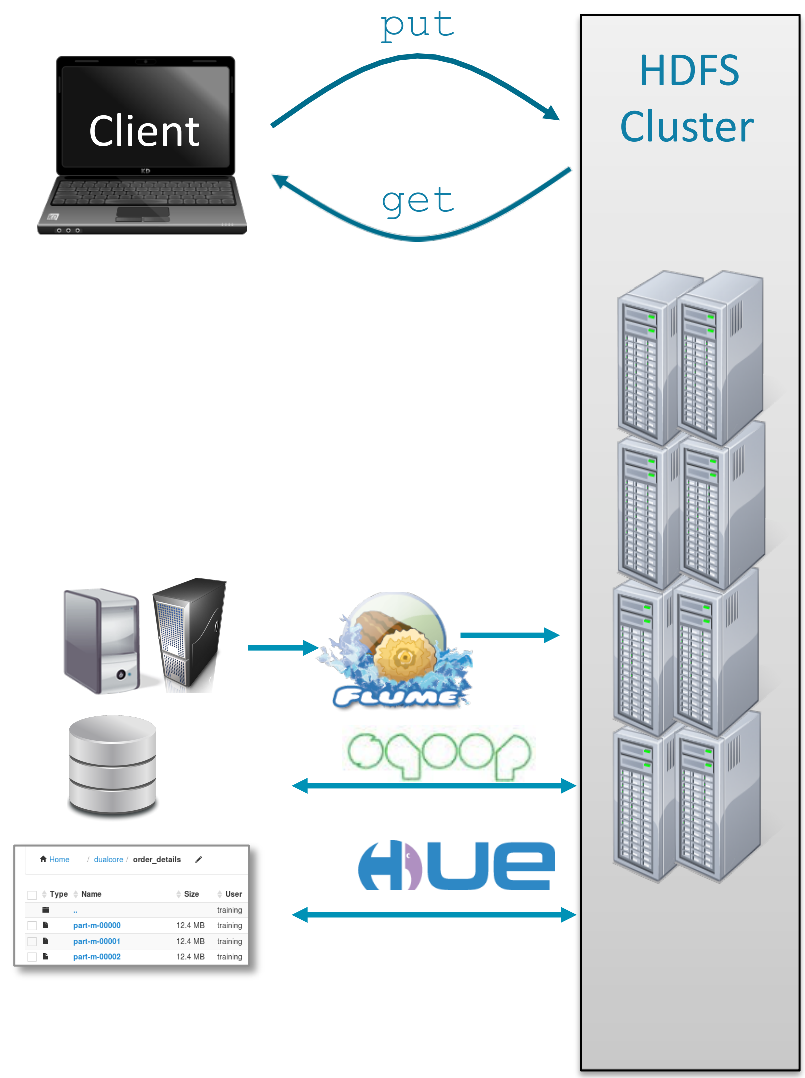


Options for Accessing HDFS

- **From the command line**
  - FsShell: **`hdfs dfs`**
- **From Cloudera Manager**
  - HDFS page, File Browser tab

`hdfs://host:port/file…`

- **From the NameNode Web UI**
  - Utilities > Browse the file system
- **Other programs**
  - Java API
    - Used by MapReduce, Spark, Impala, Hue, Sqoop, Flume, etc.
  - RESTful interface



Accessing HDFS via the Command Line

- **HDFS is not a general purpose filesystem**
  - Not built into the OS, so only specialized tools can access it
- **End users typically access HDFS via the `hdfs dfs` command**
  - Actions are specified with subcommands (prefixed with a minus sign)
  - Most subcommands are similar to corresponding UNIX commands
- **Display the contents of the `/user/fred/sales.txt` file**

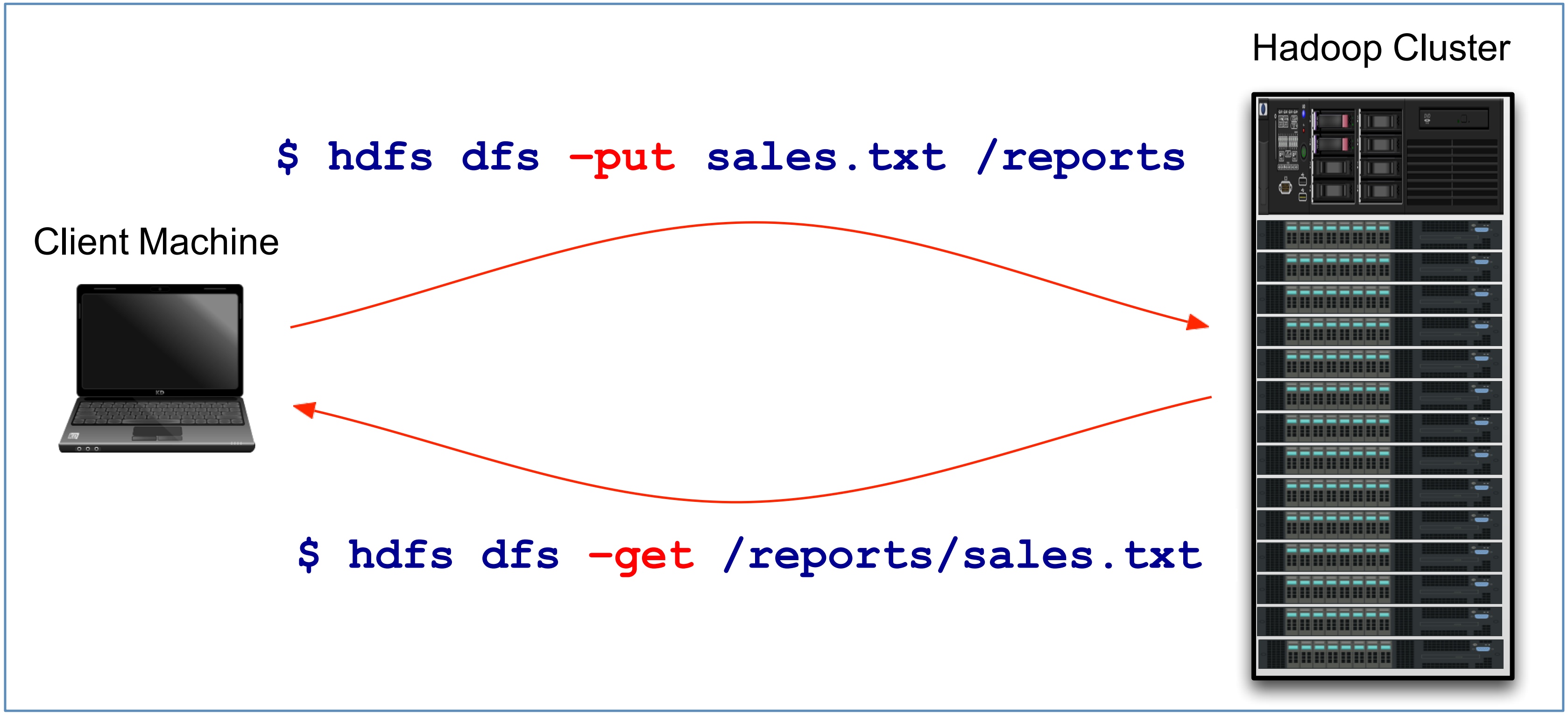
```
$ hdfs dfs -cat /user/fred/sales.txt
```

- **Create a directory (below the root) called `reports`**

```
$ hdfs dfs -mkdir /reports
```

Copying Local Data To and From HDFS Using the Command Line

- **Remember that HDFS is distinct from your local filesystem**
  - The `hdfs dfs -put` command copies local files **to** HDFS
  - The `hdfs dfs -get` fetches a local copy of a file **from** HDFS



More `hdfs dfs` Command Examples

- **Copy file `input.txt` from local disk to the user's directory in HDFS**

```
$ hdfs dfs -put input.txt input.txt
```

  - This will copy the file to `/user/`*`username`*`/input.txt`

- **Get a directory listing of the HDFS root directory**

```
$ hdfs dfs -ls /
```

- **Delete the file `/reports/sales.txt`**

```
$ hdfs dfs -rm /reports/sales.txt
```

Hands-On Exercise: Exploring HDFS