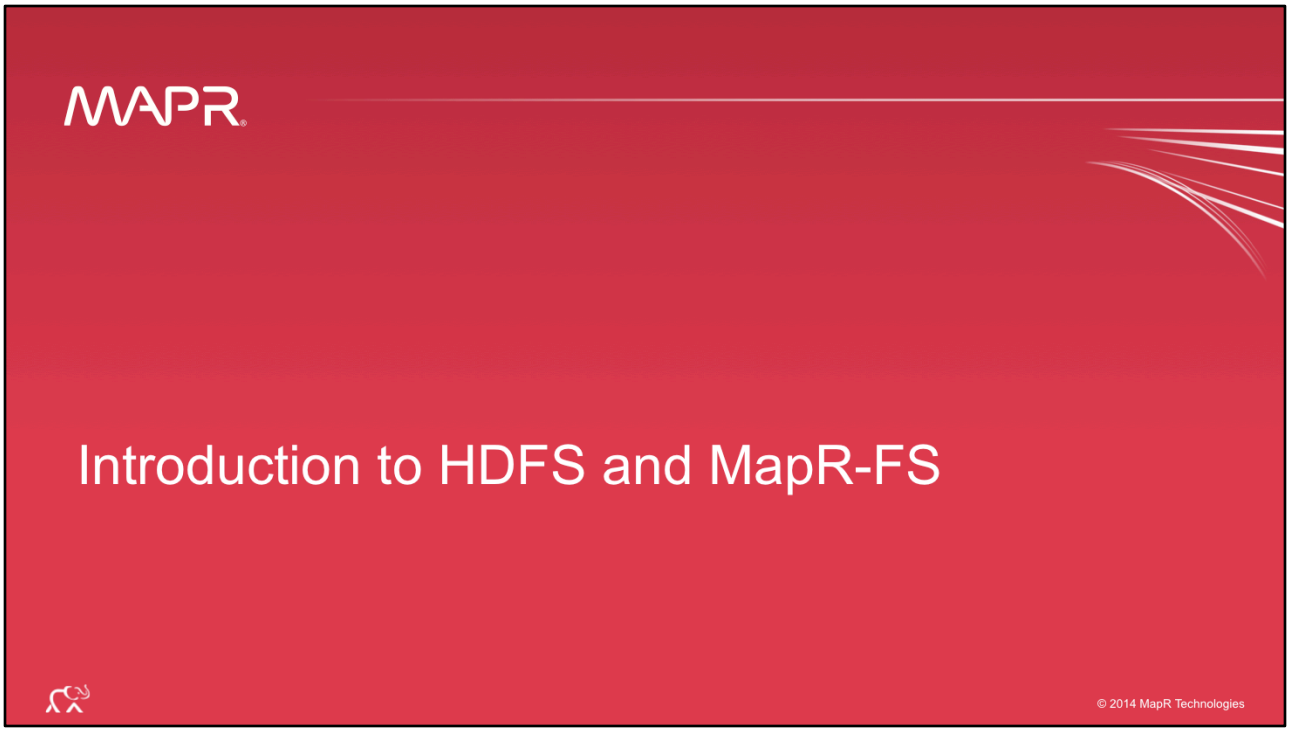


This lesson introduces and differentiates the HDFS and MapR-FS storage frameworks.

A Few Quotes to Start Us Off

Data is a precious thing and will last longer than the systems themselves.

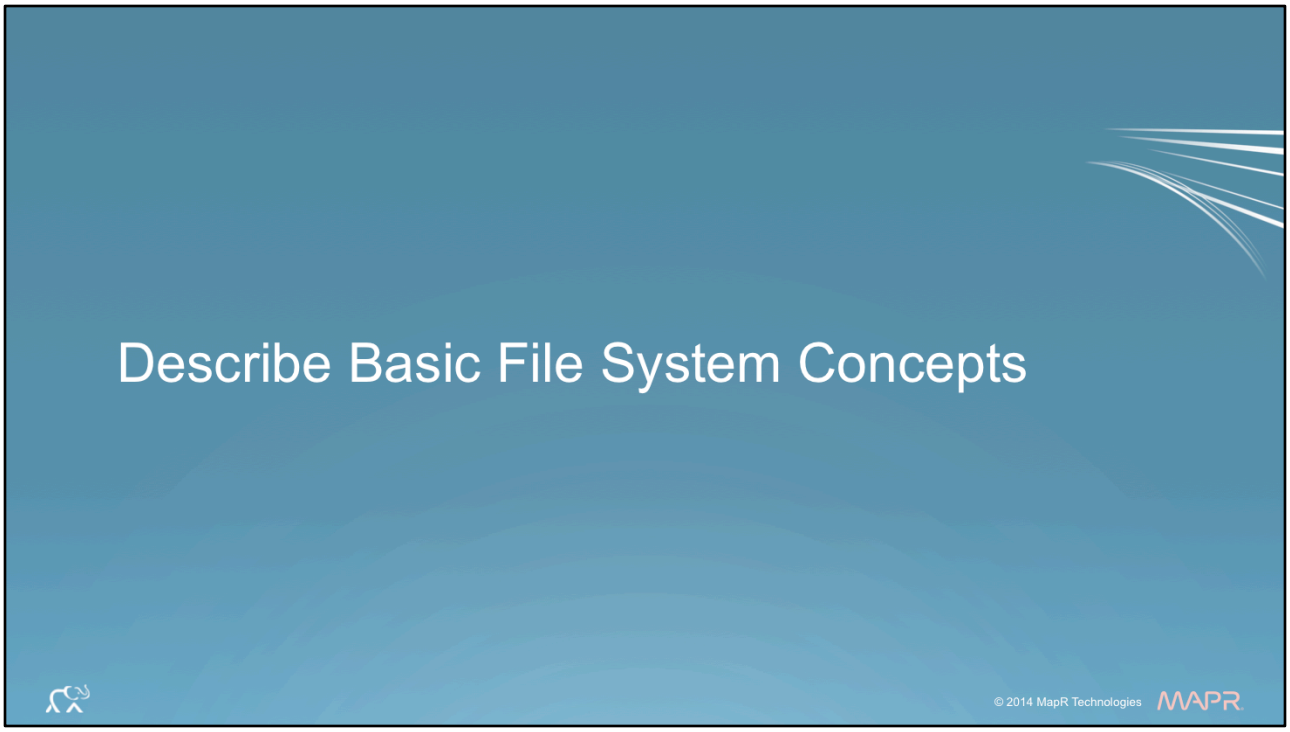
(Tim Berners-Lee, 2008)

Data is not information, information is not knowledge, knowledge is not understanding, understanding is not wisdom.

(Sir Arthur Conan Doyle)

Learning Objectives

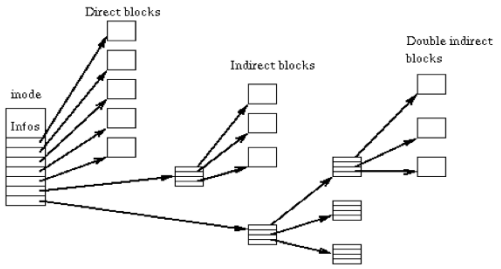
- Describe basic file system concepts
- Discuss an overview of HDFS
- Discuss an overview of MapR-FS
- Use the CLI to manage storage



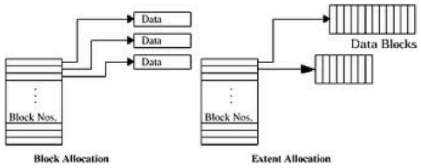
In this section, we will describe basic file system concepts.

Describe File System Concepts

- Logical structure that organizes files on a storage medium
- Dictates how data is stored and retrieved
- Contains data and metadata



ext2 file system



block vs extent-based allocation

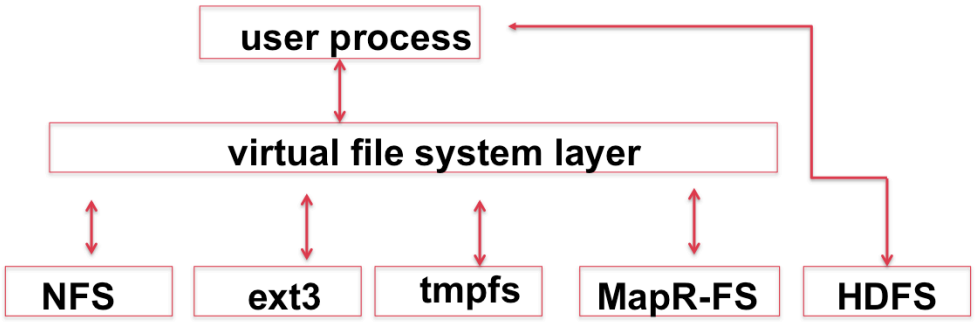
A file system is a logical structure (or rather a set of logical structures) that organizes files, directories, links, etc on a storage medium. A file system defines how file data is stored and retrieved. A file system is similar to a database in that it presents an interface for interacting with data, and that interface is abstracted from how data is actually stored and retrieved. A file system manages both the data itself as well as metadata that determines information such as file type, ownership, permissions, modification/access times, file names, and physical data locality.

The graphic above depicts the partial layout of an ext2 file system in Linux. The inode object stores all the metadata information relating to a file (e.g. owner, permissions, … etc), as well as the pointers to the actual locations of the data itself.

Many disk-based file systems such as ext2 use the concept of a pointer which resolves the location of the physical data blocks that comprise a file. Most modern disk-based file systems use 8KB as the size of a data block. A direct block is accessed directly from the inode, while an indirect block dereferences a set of pointers to another set of data blocks. Similarly, a double-indirect pointer points to a set of pointers, each of which point to a set of pointers that point to data blocks. File system blocks are allocated in the order of direct, indirect, and double-indirect as the size of the file grows.

Describe the Purpose of a Virtual File System

- Translation layer from generic file system to real file system
- Enables standard POSIX file access



 6

A virtual file system is an operating system layer that translates a set of "generic" (POSIX-compliant) file system access methods to actual implemented file system calls. As such, the client can use either operating system shell commands (e.g. cat, vi, tail, … etc) or system-level calls (e.g. open, close, read, write) to access files without knowing the underlying implementation of the file system.

NFS is a network-based file system that allows users to access files stored on a file server in the same way as if the files were stored local to that user.

Ext3 is a disk-based file system used in Linux distributions.

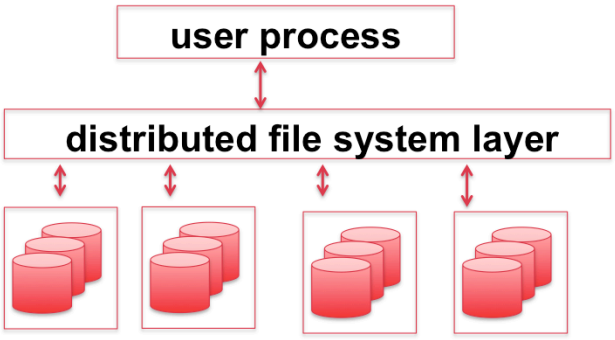
Tmpfs is a memory-based file system. Files stored in tmpfs do not survive reboots of the machine and as such is used for scratch space.

MapR-FS is a distributed file system that allows users to access files stored on a MapR cluster. Since it presents as a virtual file system, users can use system calls like open and close as well as user commands like tail and vi to access MapR-FS files. Note that a MapR file system can also be exported as an NFS file system.

HDFS is also a distributed file system, but it does not present as a virtual file system. As such, users cannot access files in HDFS using POSIX-compliant system calls like open and close or user-level commands like cat and grep.

Describe Distributed File System Concepts

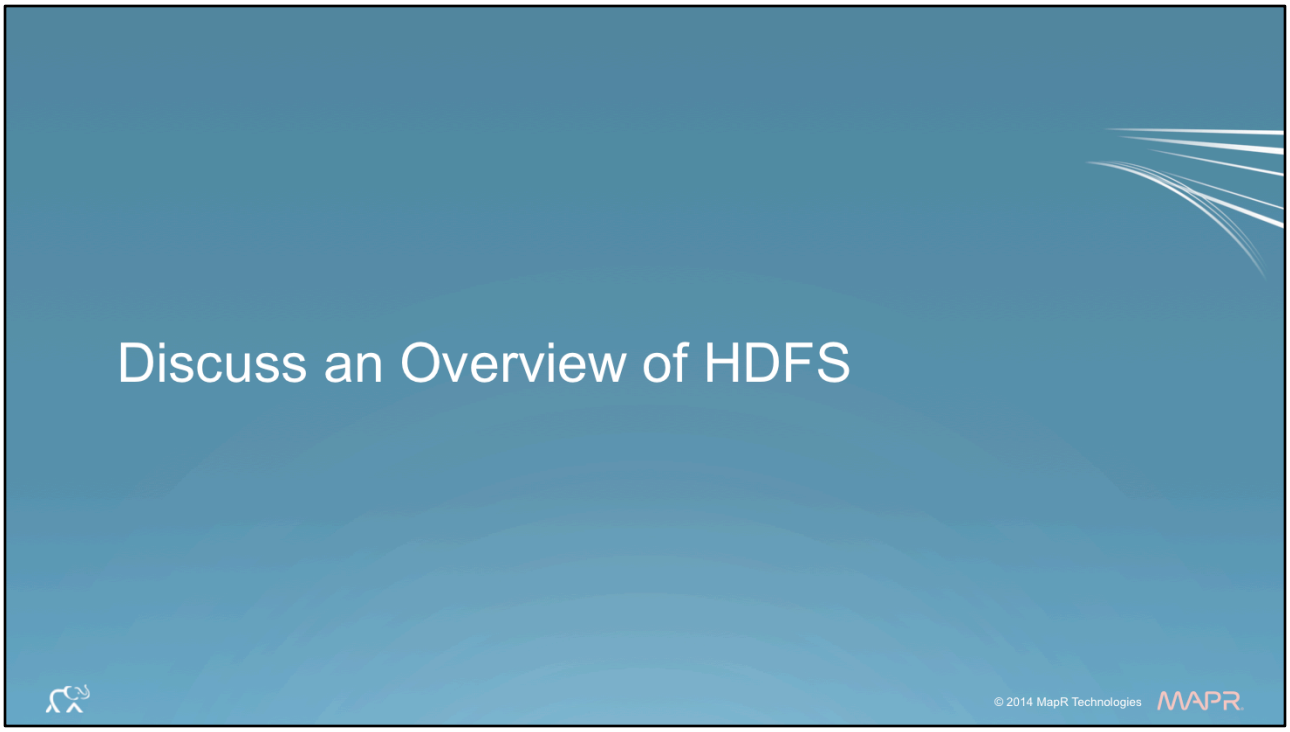
- Centrally stores metadata and distributes actual data
- Overcomes space, performance, and availability limitations of a single machine
- Abstracts data locality from client access



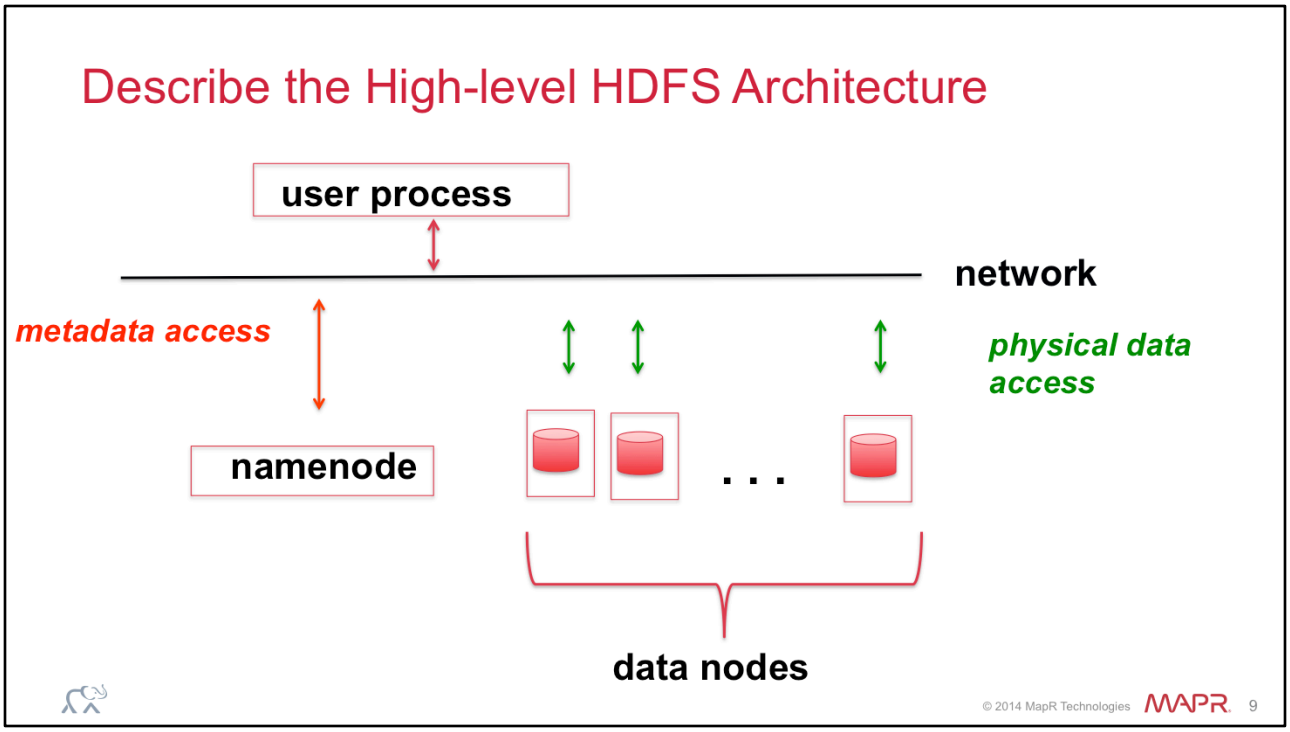
 7

A distributed file system extends the physical boundary of a normal file system (which resides on a single machine – either host or storage array) to a set of data nodes. This overcomes the space, performance, and availability limitations of hosting a file system on a single machine. Additionally, the actual physical locality of the file system data is abstracted from the client.

In the graphic above, a typical user process will first interact with the distributed file system layer (either directly or indirectly), and then the client will store and retrieve data over the network from the nodes that physically store the data.



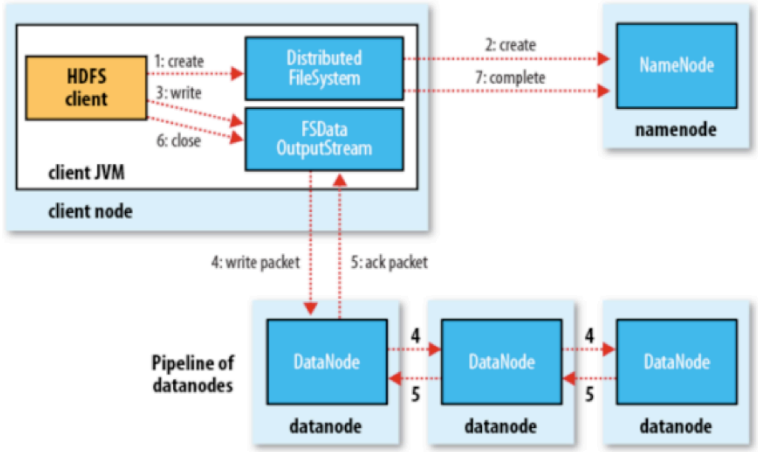
In this section, we will discuss an overview of HDFS.



The graphic above depicts a high-level architecture for HDFS. Note that since HDFS is a distributed file system, all file system operations are transacted across a network.

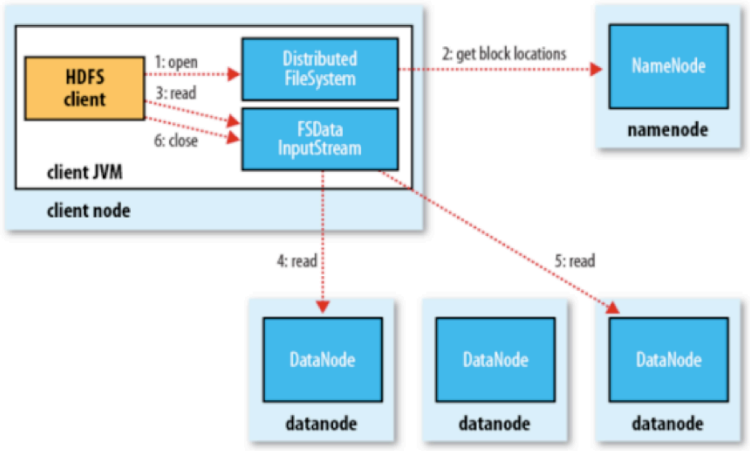
The namenode maintains metadata information for all the physical data blocks that comprise the files. HDFS clients always contact the namenode first for file operations. The HDFS clients use this metadata information to contact the data nodes to store and retrieve data stored in HDFS.

Discuss How Files are Written in HDFS



The graphic above shows how files are written to HDFS. The client begins by submitting a request to write to the file system to the namenode. The namenode splits up the blocks of data that comprise the write to different data nodes, and the client writes each block to each data node. The data nodes in turn replicate each block of data to another data node. Note that this process is executed serially from the client, block by block, to the primary data node until all the data is written to the cluster.

Discuss How Files are Read in HDFS



The graphic above shows how files are read from HDFS. The client begins the transaction by contacting the namenode to discover the locations of each block of the file. The blocks are read sequentially, block by block, from each data node until the read is complete.

Identify Limitations of HDFS

aspect	limitation
Block size	Same size used for I/O, replication, and sharding
Mutability	Write-once, read-many
POSIX semantics	Must use '`hadoop fs`' to access data
Availability	No snapshot or built-in mirroring capability
Scalability	Namenode only scales to 100M files
Performance	Written in Java and runs on block device

There are a few notable limitations of the HDFS framework, as described in the table above. The block size, while configurable, is overloaded to define the I/O size, the sharding size, and the na menode namespace. This one-size-fits-all approach does not exploit the inherent differences between these requirements.

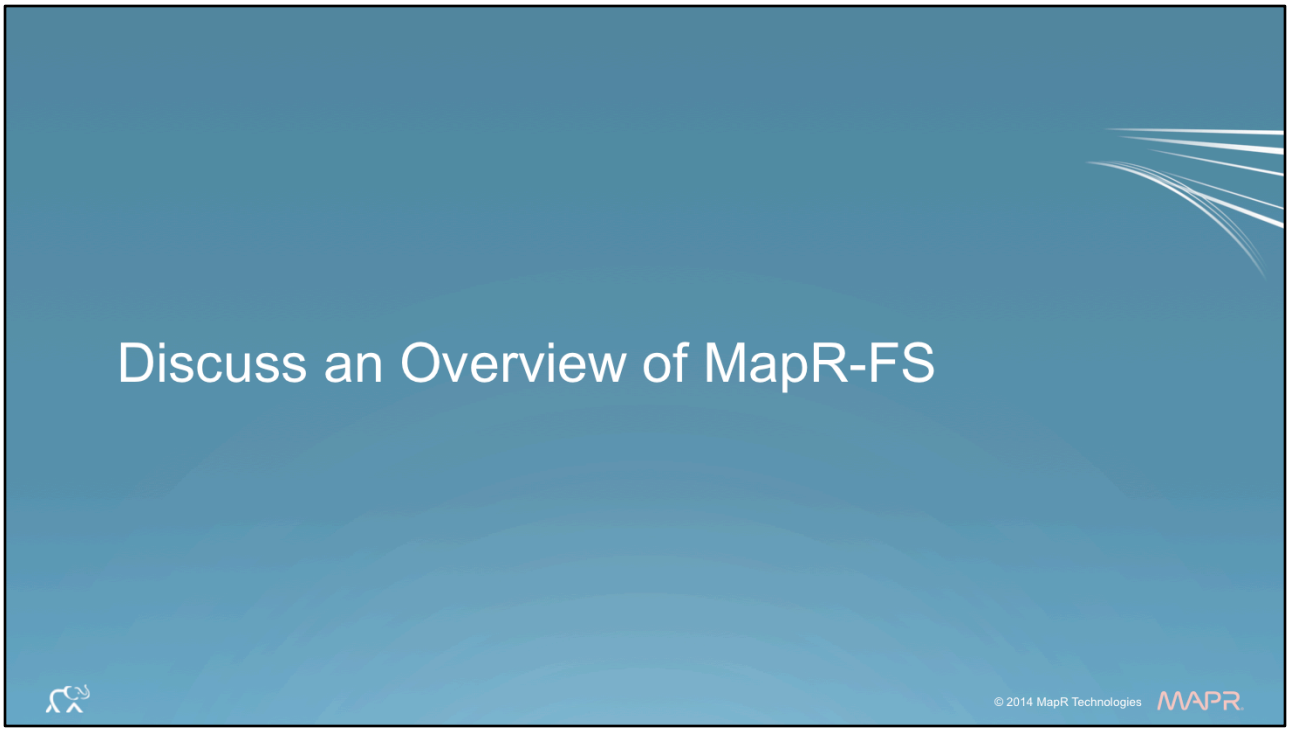
Data in HDFS is immutable – which is to say you write it once and then can read it ad infinitum. As such, if the source data changes, the data must be reloaded into the cluster (which is a time consuming operation).

HDFS does not support standard POSIX file symantics. You must use the 'hadoop fs' command in order to read and write the data. Users of HDFS must learn and incorporate this into their data flows.

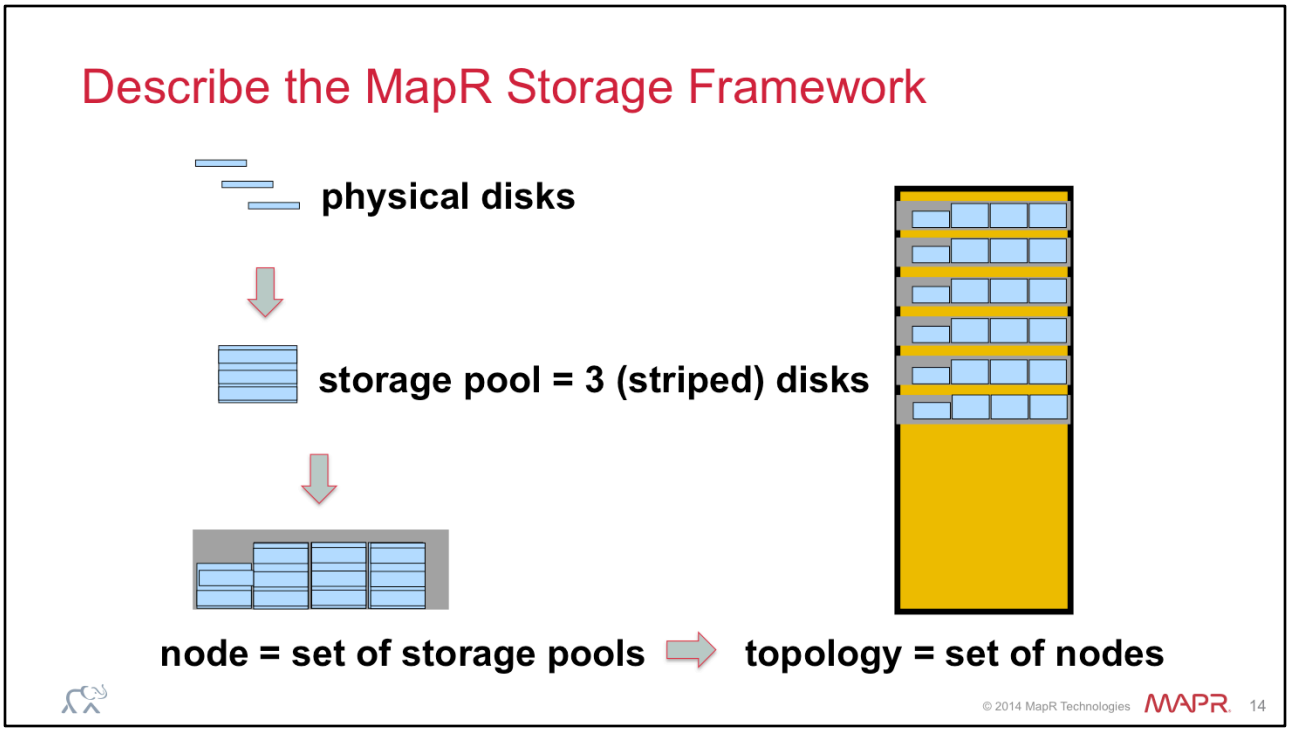
HDFS does not support snapshotting or remote mirroring, both of which enhance the availability and usability of the data set.

In terms of scalability, HDFS has an upper bound of 100 million files.

Since HDFS is written in Java, it is slower than filesystems written



In this section, we will discuss an overview of MapR-FS.



The hierarchy of how data is physically stored and organized in a MapR cluster is depicted above.

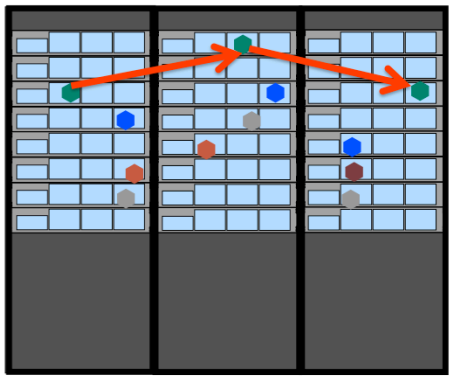
Each data node has a set of one or more disks. These disks are organized in groups of 3 as storage pools. When data is written to a storage pool, it is striped across the 3 disks. Depending on how many disks a node has, it will contain one or more storage pools. All the storage pools across all the data nodes contribute to the overall storage structure of a MapR.

Define Storage Pools and Containers

Data is written to *containers*

Containers are replicated

storage pool has many containers



The "container" is a the logical abstraction of storage within a MapR cluster that represents the unit of replication. A container size is configurable (16G by default). A container points to underlying storage pools (which reference a set of 3 striped disks).

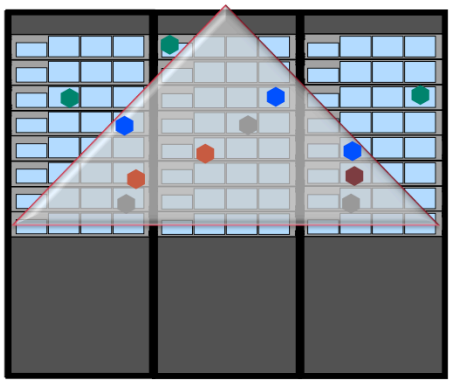
Containers are defined in terms of primary, secondary, and tertiary (more or less depending on the replication factor). By default, the replication factor is 3 but is configurable. Data is first written to the primary by the client, and the primary container then replicates that data to the secondary, which in turn replicates that data to the tertiary container.

Define Containers and Volumes

Volumes are MapR specific features

Volumes group data containers across the cluster

All replicas of a container are included in the same volume.



The highest level of data abstraction is the volume. A volume is a group of one or more containers and is used for encapsulating policies such as replication, permissions, mirroring, quotas, and more. Note that all the replicas of a given container are included in the same volume.

Identify MapR-FS Volume Features

Feature	Description
topologies	Provide placement policies for data
compression	Compresses data as it is being written to disk
mirrors	Copy data locally or remotely for protection
snapshots	Maintain point-in-time data and updates
quotas	Restrict total capacity per-user or per-group
permissions	Restrict access to users and groups
replication	Replicate containers in volume across cluster

Topologies
You can configure different topologies to enforce that certain data resides in certain locations (racks and nodes). In this way, you can design your data to exploit locality of reference for performance, availability, and any other arbitrary criteria that suit you. Then you can associate these topologies with your volume. HDFS has a similar concept of rack awareness.

Compression
Data in a volume is automatically compressed as it is being written to disk.

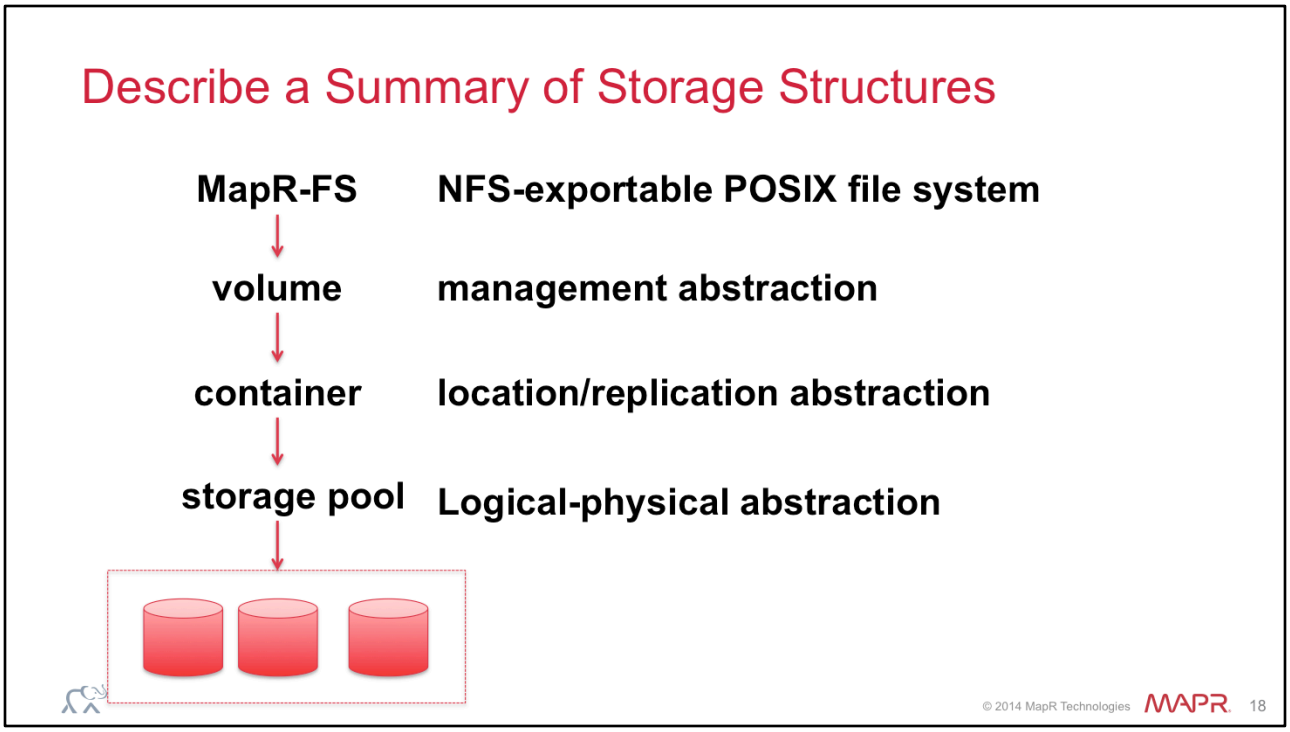
Mirrors
You can configure policies to mirror your data locally (within the same cluster) or remotely (to a different cluster). Mirrors provide a level of protection against multiple disk failures, and remote mirrors provide a level of disaster recovery.

Snapshots
Volume snapshots allow you to create point-in-time versions of data while maintaining the on-going consistency of a live volume. Snapshots provide a level of protection against user error, and they also allow you to return to a version of a data set at any time. You can also perform recurring snapshots automatically based on a calendar.

Quotas
Quota is upper bound for disk space utilization. Each volume, user, or group may have only one quota. User and group quotas apply to the total sum of the sizes of all volumes for which a user or group is the accountable entity. You can either configure “hard” quotas (prohibitive) or “soft” (advisory) quotas which set alarms when reached.

Permissions
Dump, restore, modify, delete, full control (for volume-level permissions). There are also standard UNIX permissions associated with the files and directories in the volume.

Replication
Containers that comprise the contents of a volume are replicated for redundancy. The default replication factor is 3 (the original plus two copies), but you can configure the replication factor on a per-volume basis.



The graphic above summarizes the storage structures that are used in the MapR storage framework. At the top of the diagram is a MapR file system which supports both POSIX semantics as well as NFS export. A MapR file system is written to a volume, which as we've discussed is a management abstraction for features such as replication, mirroring, topology, and so forth. Volume data are stored in containers which define the location and replication for the volume. A container is written to a storage pool, which is a set of one or more physical disks (by default 3).

Cite Differences b/w MapR-FS and HDFS

aspect	feature
Block size	Different sizes used for sharding, replicating, and performing I/O
Mutability	Full read-write capability
Access	Can NFS-mount MapR-FS volumes
POSIX semantics	Can use native OS to access data
Availability	Snapshots and local/remote mirroring support
Scalability	No limit to the number of files
Performance	Written in C and runs on raw device

 19

The table above defines the major differences between HDFS and the MapR-FS. MapR-FS allows administrators to define different sizes for I/O, chunking, and metadata distribution. The data in MapR-FS is stored in 16GB containers which means that fewer lookups are required to resolve the namespace. The data itself is fully read-write capable, making it a true file system. Moreover, the cluster can serve the data to clients outside the cluster using the NFS protocol. All told, the MapR-FS presents as a normal mountable file system with POSIX symantics. The MapR-FS provides snapshot and local/remote mirroring support to make the file system highly available. In terms of scalability, there is no limit to the number of files you can add to MapR-FS (you are limited only by the space offered by the node disks in your cluster). Last, since MapR-FS is written in C and compiled in object code, it performs at the speed of underlying hardware.

Compare Block Sizes in MapR-FS and HDFS

Storage unit	HDFS	MapR-FS
Unit of sharding	Block = 64 MB	Chunk = 256 MB
Unit of replication	Block = 64 MB	Container = 16-32 GB
Unit of I/O	Block = 64 MB	Block = 8 KB

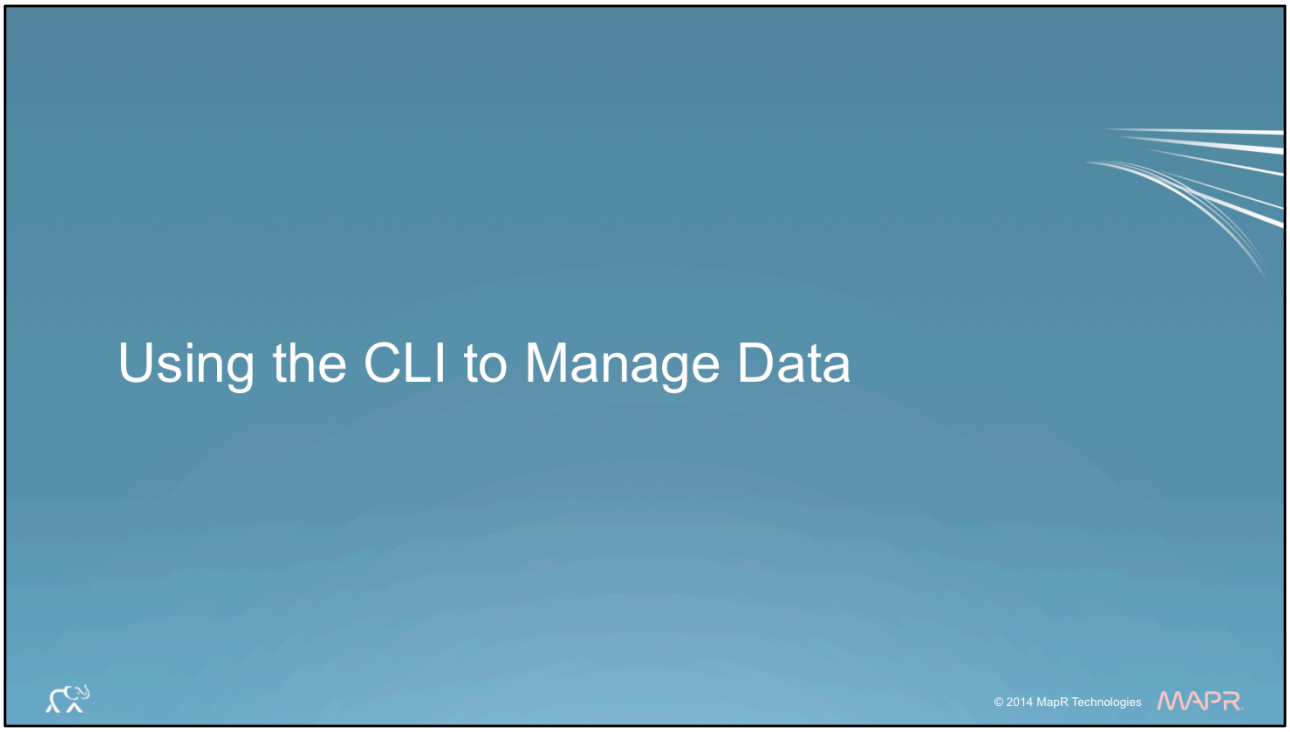
MapR-FS disambiguates use for each storage unit to optimize use case

One of the important differentiators between HDFS and MapR-FS is the definition and use of a block size. Where HDFS uses a single block size for sharding, replication, and file I/O, MapR-FS uses three different sizes. The importance of distinguishing these block sizes is in how the block is used. Using a single block size for all the uses is not efficient.

Having a larger shard size means that the metadata footprint associated with files is smaller. This permits users to store a larger number of files and make a fewer number of metadata lookups when accessing these files.

Having a larger replication size means that a fewer number of replication calls may be made.

Having a smaller I/O size means that files may be randomly read and written. This is perhaps the most critical reason for disambiguating block sizes in the distributed file system. HDFS is a write-once, append-and-read-many file system because of the large I/O size. MapR-FS on the other hand is a fully POSIX-compliant read-write file system.



In this section, we will discuss using the hadoop CLI to manage data.

Use the `hadoop fs` CLI

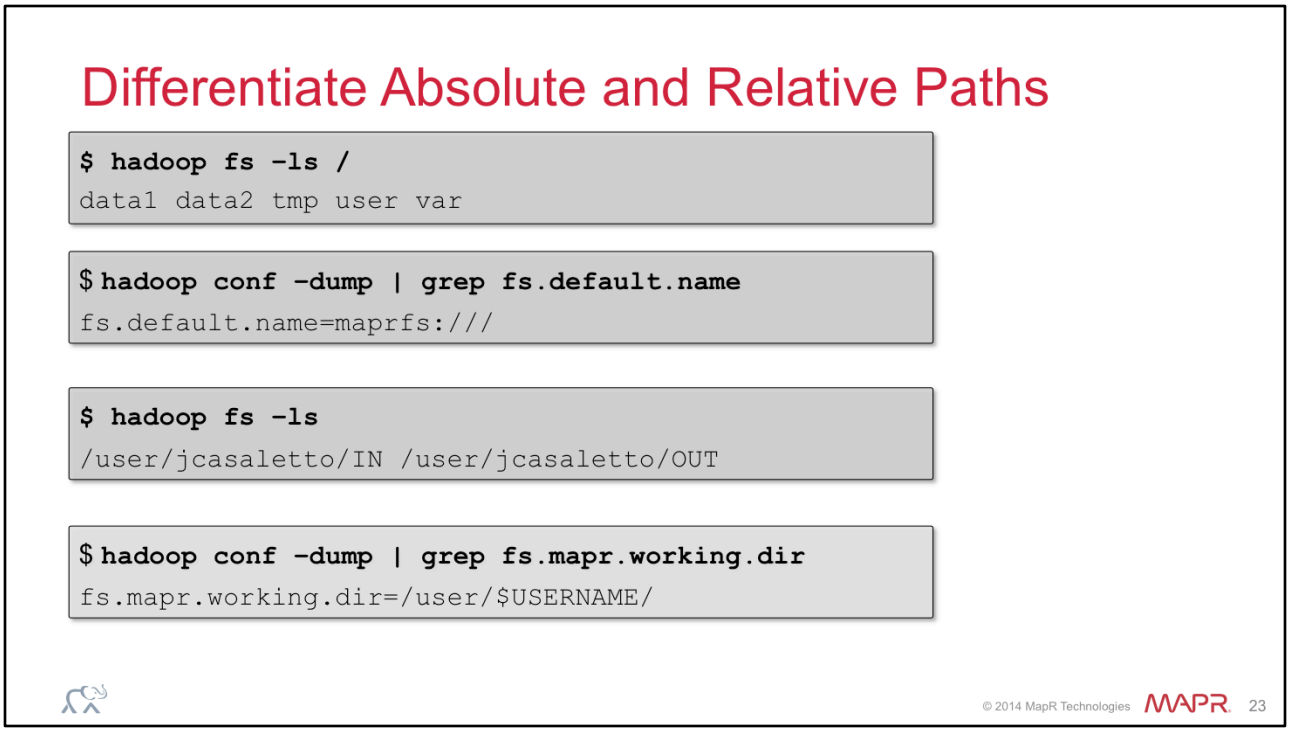
Usage: ***`hadoop fs [command] [args]`***

```
hadoop fs -mkdir mydir

hadoop fs -copyFromLocal /etc/hosts mydir

hadoop fs -lsr mydir

hadoop fs -cat mydir/hosts

hadoop fs -rm mydir/hosts
```

The commands above are examples of using the 'hadoop fs' command from the Apache Hadoop distribution.

The examples include creating a directory (mkdir), copying a file from the local file system to the distributed file system(copyFromLocal) , recursively listing directory contents (lsr), displaying file contents (cat), and removing a file (rm).

Note that the directory 'mydir' in this example is relative to the home directory of the user executing the command. This assumes that this user's home directory is mounted on a MapR-FS file system.



The slide above differentiates absolute and relative paths as interpreted by the 'hadoop fs' command.

The following two commands using absolute paths are equivalent, according to the configuration (fs.default.name) shown above:

hadoop fs -ls /
hadoop fs -ls maprfs:///

Similarly, the following two commands using relative paths are equivalent, according to the configuration (fs.mapr.working.dir) shown above:

hadoop fs -ls
hadoop fs -ls /user/$USERNAME

Use the `hadoop mfs` CLI

Usage: ***`hadoop mfs [command] [args]`***

```
hadoop mfs -ln mydir yourdir

hadoop mfs -setcompression off mydir

hadoop mfs -setchunksize 65536 mydir
```

The commands above are examples of using the 'hadoop mfs' command from the MapR Hadoop distribution. Note that the 'hadoop mfs' command has been written as a complement of functionality to the 'hadoop fs' command for MapR-specific features. These commands include creating a symbolic link (ln), turning off compression (setcompression), and setting the chunk size (setchunksize).

Use the Operating System CLI

```
mkdir /user/james/mydir

cp /etc/hosts /user/james/mydir

ls -R /user/james/mydir

ln -s /user/james/mydir /user/james/yourdir

rm  /user/james/mydir/hosts

tail / grep / awk / sed
```

One of the great benefits of using the MapR distribution for Hadoop is the ability to use operating system commands to manage your data. Examples in the slide above include creating a directory (mkdir), copying a file (cp), getting a long listing of a directory (ls –l), creating a symbolic link (ln), and removing a file (rm). Note that some commands either do not work or do not perform well on distributed file systems (examples include df, find and ls –R). Use the 'hadoop fs' or maprcli commands for these.