Ingeniería Industrial
FACULTAD DE CIENCIAS
FÍSICAS Y MATEMÁTICAS
UNIVERSIDAD DE CHILE

INTRODUCTION TO BIG DATA

Juan D. Velásquez
Felipe E. Vildoso

Ingeniería Industrial

FACULTAD DE CIENCIAS
FÍSICAS Y MATEMÁTICAS
UNIVERSIDAD DE CHILE

CHAPTER 2

Ingeniería Industrial
FACULTAD DE CIENCIAS
FÍSICAS Y MATEMÁTICAS
UNIVERSIDAD DE CHILE

LECTURE 8

Chapter 2

Ingeniería Industrial

FACULTAD DE CIENCIAS
FÍSICAS Y MATEMÁTICAS
UNIVERSIDAD DE CHILE

DATA STORAGE ON THE BATCH LAYER

HOW TO PHYSICALLY STORE THE DATA IN THE BATCH LAYER?

The master dataset is typically **too large to exist on a single server**.

You must choose **how you'll distribute** your data across multiple machines.

The way you store your master dataset will impact how you consume it, so it's vital to **devise your storage strategy with your usage patterns in mind**.

STORAGE REQUIREMENTS FOR THE MASTER DATASET

You must consider how your data will be **written** and how it will be **read**.

The batch layer affects both areas.

STORAGE REQUIREMENTS FOR THE MASTER DATASET

Two key properties of data: **data is immutable and eternally** true.

- Each piece of your data will be written once and only once.
- There is no need to ever alter your data, the only write operation will be to add a new data unit to your dataset.

The storage solution must therefore be optimized to handle a **large**, **constantly growing** set of data.

STORAGE REQUIREMENTS FOR THE MASTER DATASET

The batch layer is also responsible for computing functions on the dataset to **produce the batch views**.

- The batch layer storage system needs to be good at reading lots of data at once.
- Random access to individual pieces of data is not required.

With this **"write once, bulk read many times"** paradigm in mind, we can create a checklist of requirements for the data storage.

Write

- Efficient appends of data
- Scalable Storage

Read

- Support for parallel processing

Both

- Tunable storage and processing costs
- Enforceable immutability

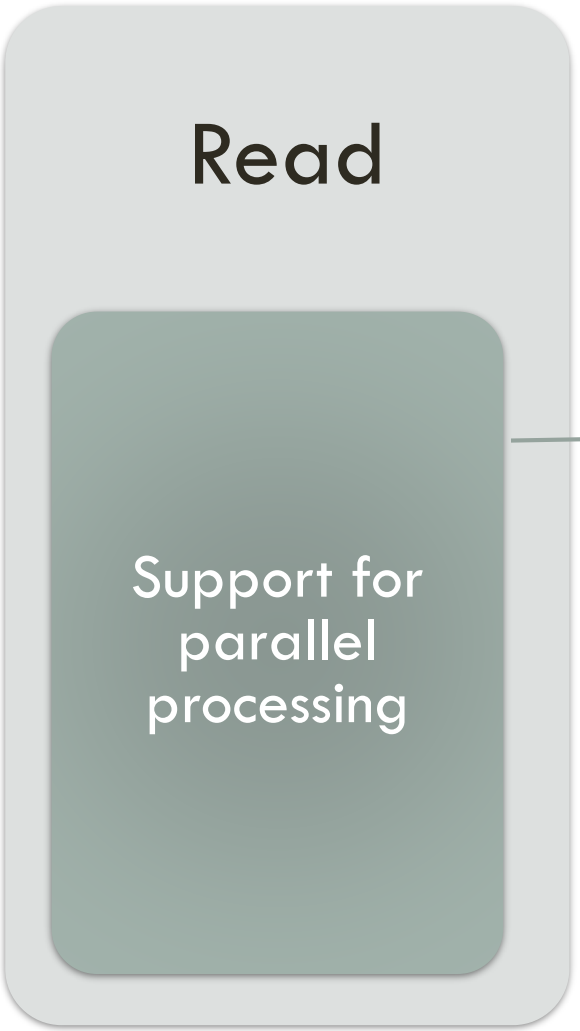
Write

Efficient appends of data

The only write operation is to add new pieces of data, so it must be easy and efficient to append a new set of data objects to the master dataset.

Scalable Storage

The batch layer stores the complete dataset – potentially terabytes or petabytes of data. It must therefore be easy to scale the storage as your dataset grows.



Constructing the batch views requires computing functions on the entire master dataset. The batch storage must consequently support parallel processing to handle large amounts of data in a scalable manner.

Both

- Tunable storage and processing costs
- Enforceable immutability

Storage costs money. You may choose to compress your data to help minimize your expenses, but decompressing your data during computations can affect performance. The batch layer should give you the flexibility to decide how to store and compress your data to suit your specific needs.

Both

- Tunable storage and processing costs
- Enforceable immutability

It's critical that you're able to enforce the immutability property on your masterdataset. Of course, computers by their very nature are mutable, so there will always be a way to mutate the data you're storing. The best you can do is put checks in place to disallow mutable operations. These checks should prevent bugs or other random errors from trampling over existing data.

CHOOSING A STORAGE SOLUTION FOR THE BATCH LAYER

With the requirements checklist in hand, you can now consider options for batch layer storage.

With such loose requirements (not even needing random access to the data) **you could use pretty much any distributed database** for the master dataset.

Ingeniería Industrial

FACULTAD DE CIENCIAS
FÍSICAS Y MATEMÁTICAS
UNIVERSIDAD DE CHILE

USING A KEY/VALUE STORE FOR THE MASTER DATASET

KEY/VALUE STORE

It's a data storage paradigm designed for storing, retrieving, and managing associative arrays, a data structure more commonly known today as a dictionary or hash.

Key	Value
K1	AAA , BBB , CCC
K2	AAA , BBB
K3	AAA , DDD
K4	AAA , 2 , 01/01/2015
K5	3 , ZZZ , 5623

KEY/VALUE STORE

We haven't discussed distributed key/value stores yet, but you can essentially think of them as giant persistent **hash maps** that are **distributed among many machines**.

You have to figure out…

- What the keys should be.
- What the values should be. A piece of data you want to store.

But what should a key be?

KEY/VALUE STORE

There's no natural key in the data model, nor is one necessary because the data is meant to be consumed in bulk. So you immediately hit an impedance mismatch between the data model and how key/value stores work.

The only really viable idea is to **generate a universally unique identifier** (UUID) to use as a key.

A UUID is simply a 128-bit value.

de305d54-75b4-431b-adb2-eb6b9e546014

KEY/VALUE STORE

This is only the start of the problems...

Because key/value stores need fine-grained access to key/value pairs to do random reads and writes, **you can't compress** multiple key/value pairs together.

You're severely limited in tuning the trade-off between storage costs and processing costs.

KEY/VALUE STORE

Key/value stores are meant to be used as **mutable stores**, which is a problem if enforcing immutability is so crucial for the master dataset.

Unless you modify the code of the key/value store you're using, you typically can't disable the ability to modify existing key/value pairs.

KEY/VALUE STORE

The biggest problem is that a **key/value store has a lot of things you don't need**: random reads, random writes, and all the machinery behind making those work.

This means the tool is enormously **more complex than it needs to be** to meet your requirements, making it much more likely you'll have a problem with it.

The key/value store indexes your data and provides unneeded services, which will increase your storage costs and lower your performance when reading and writing data.

Ingeniería Industrial
FACULTAD DE CIENCIAS
FÍSICAS Y MATEMÁTICAS
UNIVERSIDAD DE CHILE

USING A DISTRIBUTED FILESYSTEM

FILESYSTEMS

Files are **sequences of bytes**.

- The most efficient way to consume files is by scanning through them.
- They're stored sequentially on disk (sometimes they're split into blocks, but reading and writing is still essentially sequential).

FILESYSTEMS

- You have **full control** over the bytes of a file.
- You have the full **freedom to compress** them however you want.
- Unlike a key/value store, a filesystem **gives you exactly what you need and no more**, while also not limiting your ability to tune storage cost versus processing cost.
- Filesystems implement fine-grained permissions systems, which are perfect for **enforcing immutability**.

FILESYSTEMS

The **problem with a regular filesystem is that it exists on just a single machine**, so you can only scale to the storage limits and processing power of that one machine.

DISTRIBUTED FILESYSTEMS

Similar to regular filesystems, but they **spread their storage** across a cluster of computers.

- They scale by adding more machines to the cluster.
- Distributed filesystems are designed so that you have fault tolerance when a machine goes down, meaning that if you lose one machine, all your files and data will still be accessible.
- The **operations** you can do with a distributed filesystem are often **more limited** than you can do with a regular filesystem.

DISTRIBUTED FILESYSTEMS

Oftentimes having small files can be inefficient, so you want to make sure you keep your **file sizes relatively large to make use of the distributed filesystem properly** (the details depend on the tool, but 64 MB is a good rule of thumb).

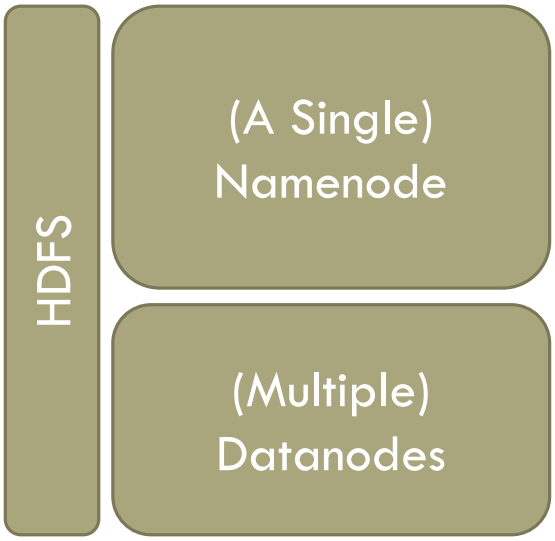
HOW DISTRIBUTED FILESYSTEMS WORK

It's tough to talk in the abstract about how any distributed filesystem works, so we'll ground our explanation with a specific tool: the **Hadoop Distributed File System** (HDFS).

HDFS and Hadoop MapReduce are the two prongs of the Hadoop project: a Java framework for distributed storage and distributed processing of large amounts of data.

HOW DISTRIBUTED FILESYSTEMS WORK

Hadoop is deployed across multiple servers, typically called a cluster, and HDFS is a **distributed and scalable filesystem that manages how data is stored across the cluster**.



Keeps the directory tree of all files

Tracks where across the cluster the file data is kept.

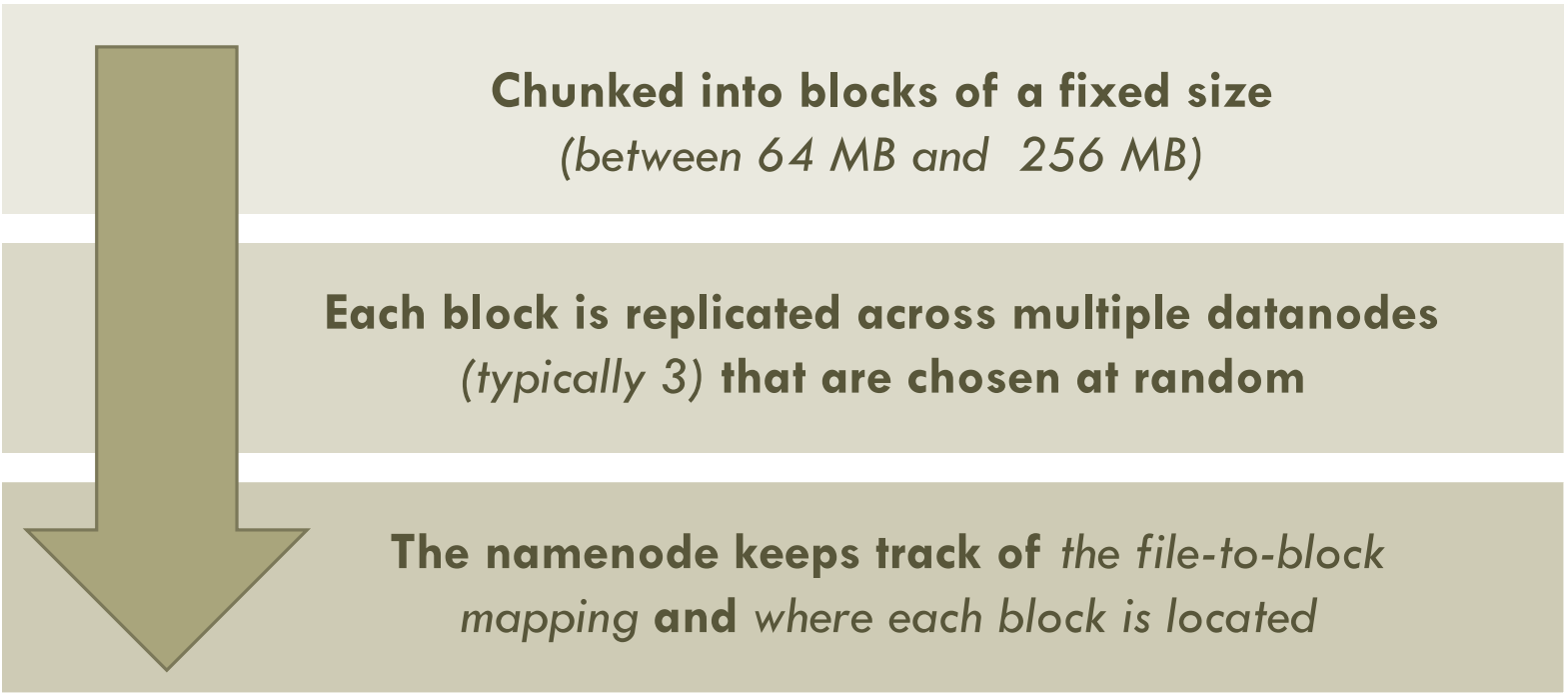
It does not store the data of these files itself.

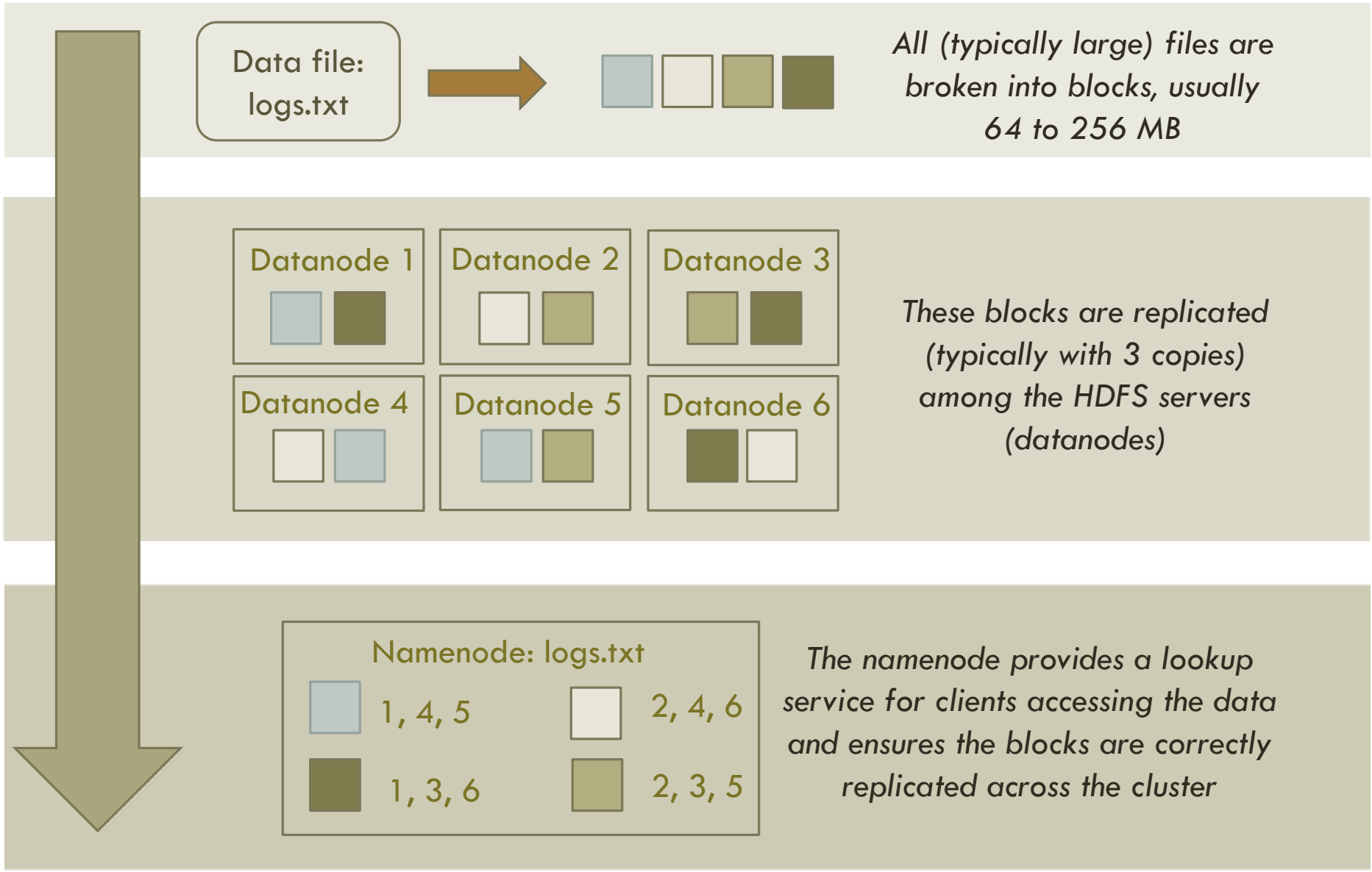
Stores data

Client applications can talk directly to a DataNode, once the NameNode has provided the location of the data.

HOW DISTRIBUTED FILESYSTEMS WORK

When you upload a file to HDF, the file…




Data file:
logs.txt
All (typically large) files are broken into blocks, usually 64 to 256 MB
Datanode 1
Datanode 2
Datanode 3
Datanode 4
Datanode 5
Datanode 6
These blocks are replicated (typically with 3 copies) among the HDFS servers (datanodes)
Namenode: logs.txt
1, 4, 5
2, 4, 6
1, 3, 6
2, 3, 5
The namenode provides a lookup service for clients accessing the data and ensures the blocks are correctly replicated across the cluster

HOW DISTRIBUTED FILESYSTEMS WORK

Distributing a file in this way across many nodes allows it to be **easily processed in parallel**.

When a program needs to access a file stored in HDFS, it contacts the namenode to determine which datanodes host the file contents.

HOW DISTRIBUTED FILESYSTEMS WORK

Additionally, with each block replicated across multiple nodes, **your data remains available even when individual nodes are offline**.

There are limits to this fault tolerance: if you have a replication factor of three, three nodes go down at once, and you're storing millions of blocks, chances are that some blocks happened to exist on exactly those three nodes and will be unavailable.

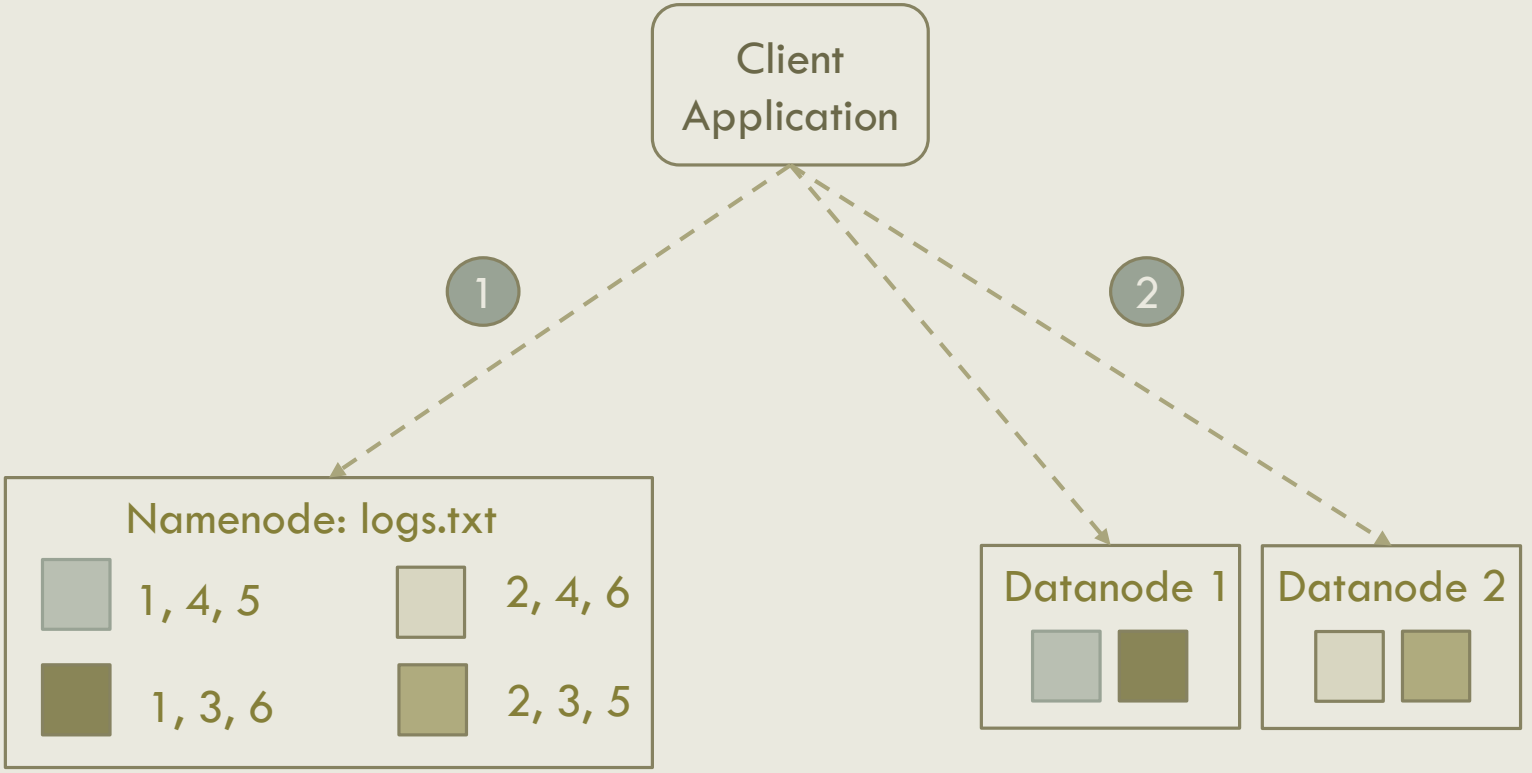
HOW DISTRIBUTED FILESYSTEMS WORK

Implementing a distributed filesystem is a difficult task, but you've now learned what's important from a user perspective.

Important things to know:

- Files are spread across multiple machines for **scalability** and also to enable **parallel processing**.
- File blocks are replicated across multiple nodes for **fault tolerance**.

Let's now explore how to store a master dataset using a distributed filesystem.



When an application processes a file stores in HDFS it first queries the namenode for the block locations.

Once the location are known, the application contacts the datanodes directly to Access the file contents.

STORING A MASTER DATASET WITH A DISTRIBUTED FILESYSTEM

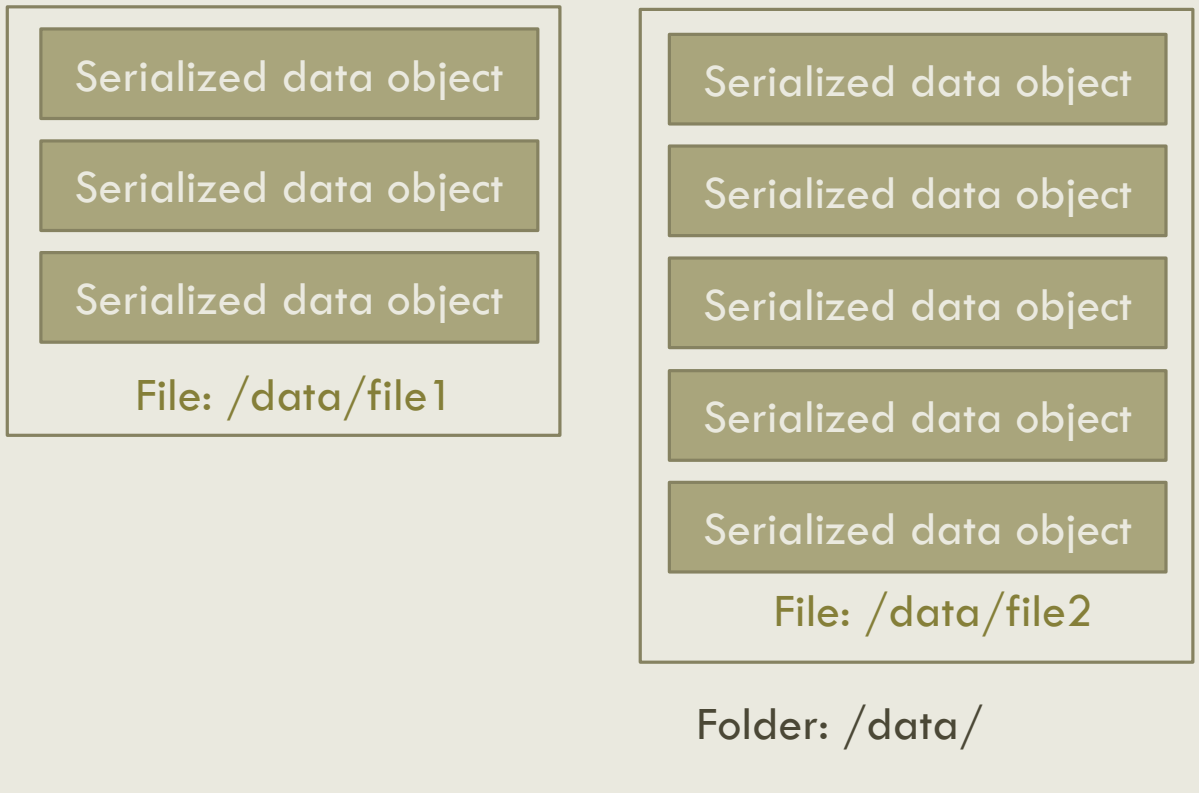
Distributed filesystems vary in the kinds of operations they permit.

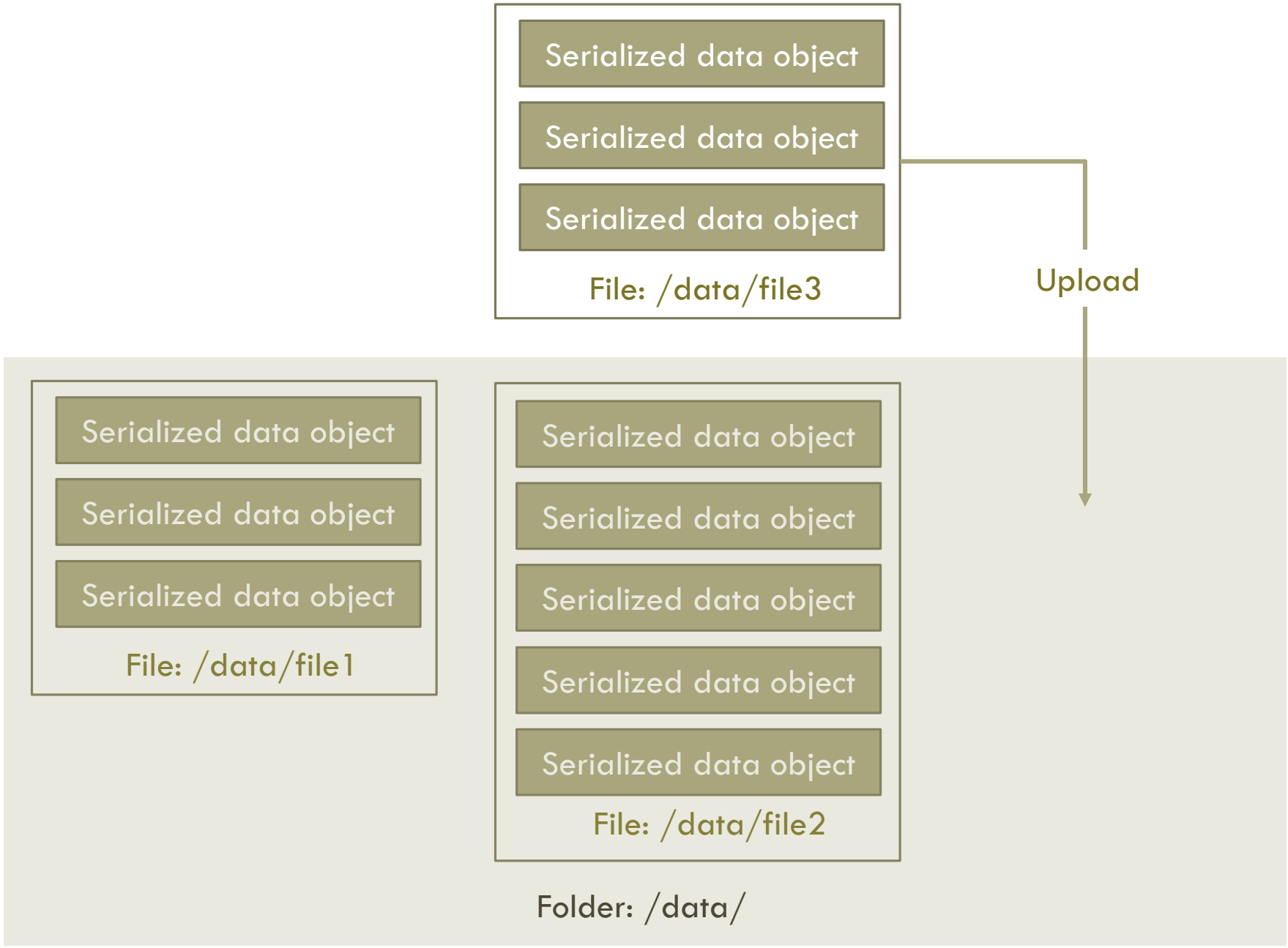
- Some distributed filesystems let you modify existing files, and others don't.
- Some allow you to append to existing files, and some don't have that feature.

STORING A MASTER DATASET WITH A DISTRIBUTED FILESYSTEM

We'll look at how you can store a master dataset on a distributed filesystem with only the most bare-boned of features, where a file can't be modified at all after being created.

With unmodifiable files you can't store the entire master dataset in a single file. What you can do instead is spread the master dataset among many files, and store all those files in the same folder. Each file would contain many serialized data objects.


Serialized data object
Serialized data object
Serialized data object
File: /data/file1
Serialized data object
Serialized data object
Serialized data object
Serialized data object
Serialized data object
File: /data/file2
Folder: /data/


Serialized data object
Serialized data object
Serialized data object
File: /data/file3
Upload
Serialized data object
Serialized data object
Serialized data object
File: /data/file1
Serialized data object
Serialized data object
Serialized data object
Serialized data object
Serialized data object
File: /data/file2
Folder: /data/

Write

- Efficient appends of data
- Scalable Storage

Read

- Support for parallel processing

Both

- Tunable storage and processing costs
- Enforceable immutability

Write

Efficient appends of data

Appending new data is as simple as adding a new file to the folder containing the master dataset.

Scalable Storage

Distributed filesystems evenly distribute the storage across a cluster of machines. You increase storage space and I/O throughput by adding more machines.

Read

Support for parallel processing

Distributed filesystems spread all data across many machines, making it possible to parallelize the processing across many machines. Distributed filesystems typically integrate with computation frameworks like MapReduce to make that processing easy to do.

Both

Tunable storage and processing costs

Enforceable immutability

Just like regultar filesystems, you have full control over how you store your data units within the files. You choose the file format for your data as well as the level of compression. You're free to do individual record compression, block-level compression, or neither.

Both

- Tunable storage and processing costs
- Enforceable immutability

Distributed filesystems typically have the same permissions systems you're used to using in regular filesystems. To enforce immutability, you can disable the ability to modify or delete files in the master dataset folder for the user with which your application runs. This redundant check will protect your previously existing data against bugs or other human mistakes.

STORING A MASTER DATASET WITH A DISTRIBUTED FILESYSTEM

At a high level, distributed filesystems are straightforward and a natural fit for the master dataset.

Like any tool they have their quirks. But it turns out that there's a little more you can exploit with the files and folders abstraction to improve storage of the master dataset.

VERTICAL PARTITIONING

Although the batch layer is built to run functions on the entire dataset, **many computations don't require looking at all the data**.

Ex.: You may have a computation that only requires information collected during the past two weeks.

The batch storage should allow you to **partition your data** so that a function **only accesses data relevant** to its computation. This process is called vertical partitioning.

VERTICAL PARTITIONING

Vertical partitioning can greatly contribute to making the batch layer **more efficient**.

While it's not strictly necessary for the batch layer (as it's capable of looking at all the data at once and filtering out what it doesn't need), vertical partitioning enables large performance gains, so it's important to know how to use the technique.

VERTICAL PARTITIONING

Vertically partitioning data on a distributed filesystem can be done by sorting your data into separate folders.

Suppose you're storing login information on a distributed filesystem. Each login contains a username, IP address, and timestamp.

VERTICAL PARTITIONING

- To vertically partition by day, you can create a separate folder for each day of data.
- Each day folder would have many files containing the logins for that day.
- Now if you only want to look at a particular subset of your dataset, you can just look at the files in those particular folders and ignore the other files.

VERTICAL PARTITIONING

Folder: /logins

Folder: /logins/2012-10-25

File: /logins/2012-10-25/logins-2012-10-25.txt

Alex 192.168.12.125 Thu Oct 25 22:33 – 22:46 (00:12)

Bob 192.168.8.251 Thu Oct 25 21:04 – 21:28 (00:24)

…

Folder: /logins/2012-10-26

File: /logins/2012-10-26/logins-2012-10-26-part1.txt

File: /logins/2012-10-26/logins-2012-10-26-part2.txt

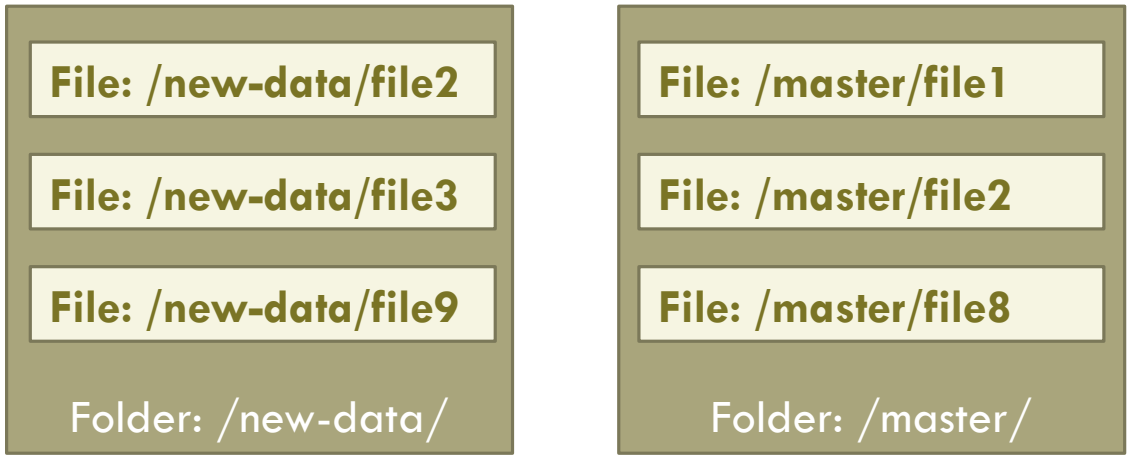
LOW-LEVEL NATURE OF DISTRIBUTED FILESYSTEMS

While distributed filesystems provide the storage and fault-tolerance properties you need for storing a master dataset, you'll find using their APIs directly too low-level for the tasks you need to run.

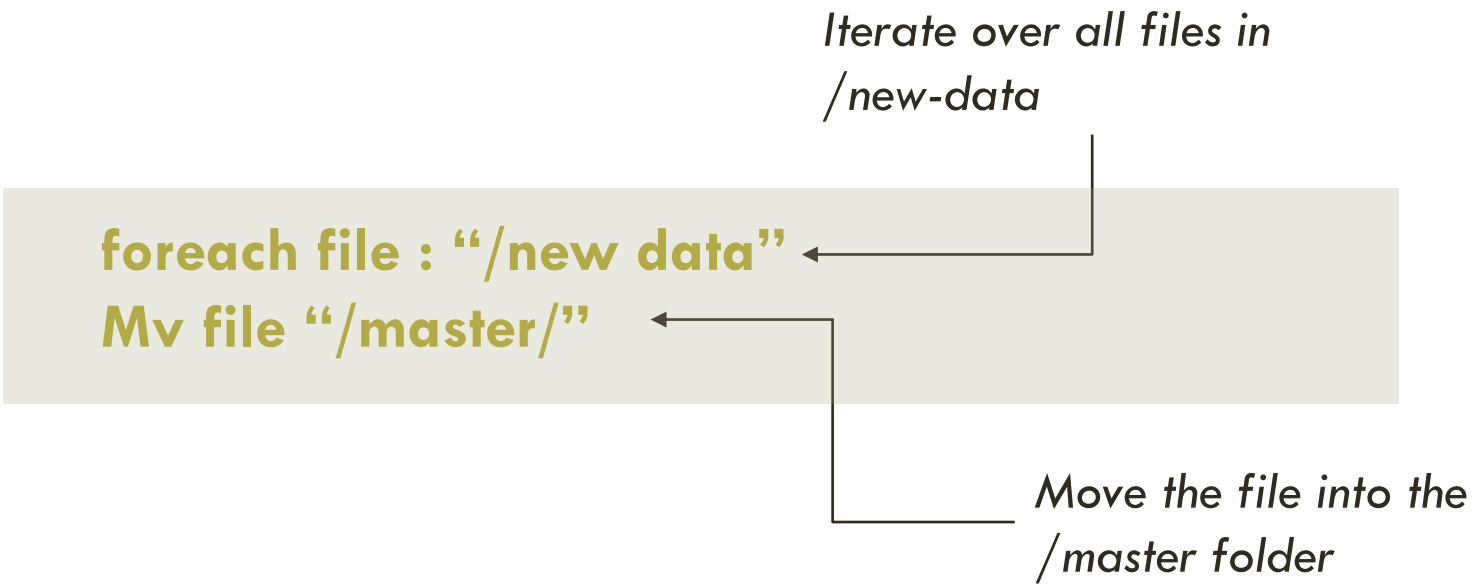
We'll illustrate this using regular Unix filesystem operations and show the difficulties you can get into when doing tasks like appending to a master dataset or vertically partitioning a master dataset.

LOW-LEVEL NATURE OF DISTRIBUTED FILESYSTEMS

Suppose your master dataset is in the folder /master and you have a folder of data in /new-data that you want to put inside your master dataset. Suppose the data in the folders is contained in files.



LOW-LEVEL NATURE OF DISTRIBUTED FILESYSTEMS



LOW-LEVEL NATURE OF DISTRIBUTED FILESYSTEMS

Unfortunately, this code has serious problems…

- If the master dataset folder contains any files of the same name, then the mv operation will fail. To do it correctly, you have to be sure you rename the file to a random filename and so avoid conflicts.

- One of the core requirements of storage for the master dataset is the ability to tune the trade-offs between storage costs and processing costs.

LOW-LEVEL NATURE OF DISTRIBUTED FILESYSTEMS

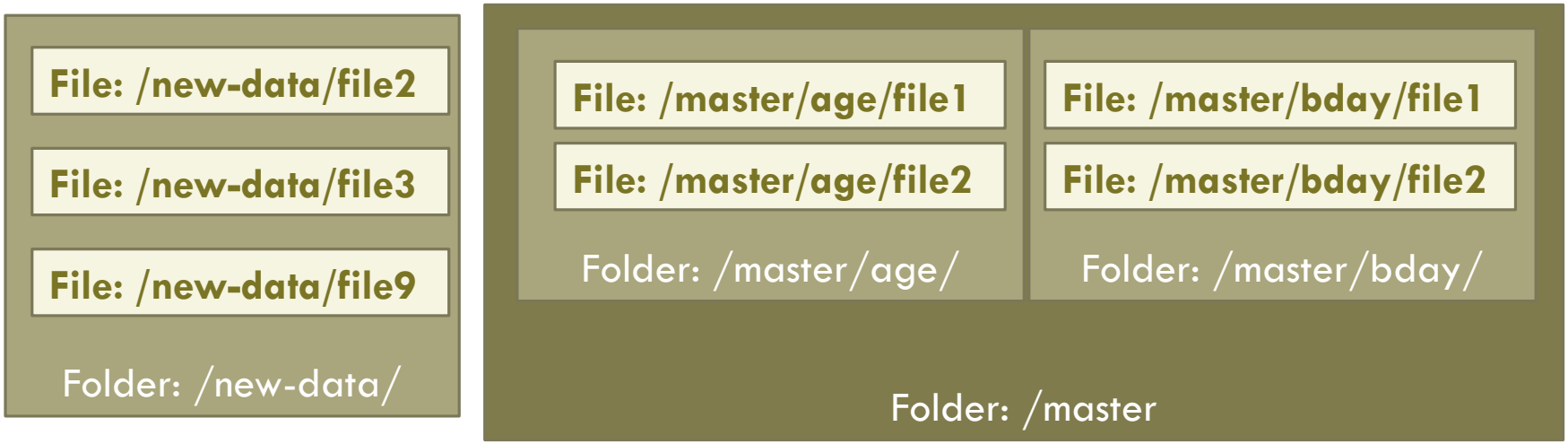
When storing a master dataset on a distributed filesystem, you choose a **file format** and **compression format** that makes the trade-off you desire.

What if the files in /new-data are of a different format than in /master? Then the mv operation won't work at all.

You instead need to copy the records out of /new-data and into a brand new file with the file format used in /master.

LOW-LEVEL NATURE OF DISTRIBUTED FILESYSTEMS

Let's now take a look at doing the same operation but with a vertically partitioned master dataset.



LOW-LEVEL NATURE OF DISTRIBUTED FILESYSTEMS

Just putting the files from /new-data into the root of /master is wrong because it wouldn't respect the vertical partitioning of /master.

Either the append operation should be disallowed, because /new-data isn't correctly vertically partitioned, or /new-data should be vertically partitioned as part of the append operation.

LOW-LEVEL NATURE OF DISTRIBUTED FILESYSTEMS

When you're just using a files-and-folders API directly, it's very **easy to make a mistake and break the vertical partitioning constraints** of a dataset.

All the operations and checks that need to happen to get these operations working correctly strongly indicate that files and folders are too low-level of an abstraction for manipulating datasets.

Ingeniería Industrial

FACULTAD DE CIENCIAS
FÍSICAS Y MATEMÁTICAS
UNIVERSIDAD DE CHILE

QUESTIONS?

SEE YOU ON THE NEXT CLASS!