Chapter 4 Data storage on the batch layer

4.1 Storage requirements for the master dataset

To determine the requirements for data storage, you must consider how your data will be written and how it will be read. The role of the batch layer within th Lambda Architecture affects both values.

In chapter 2 we emphasized two key properties of data: data is immutable and eternally true. Consequently, 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 ad new data unit to your dataset. The storage solution must therefore be optimized to handle a large, constantly growing set of data.

The batch layer is also responsible for computing functions on the dataset to produce the batch views. This means the batch layer storage system needs to be good at reading lots of data at once. In particular, 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.

4.2 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 – it seems like you could use pretty much any distributed database for the master dataset.

4.2.1 Using a key/value store for the master dataset

We haven't discussed distributed key/value stores yet, but you can essentially think of them as giant persistent hashmaps that are distributed among many machines. If you're storing a master dataset on a key/value store, the first thing you have to figure out is what the keys should be and what the values should be.

What a value should be is obvious – it's a piece of data you want to store – but what should a key be? 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 UUID to use as a key.

But this is only the start of the problems with using key/value stores for a master dataset. Because key/value store need fine-grained access to key/value pairs to do random reads and writes, you can't compress multiple key/value pairs together. So you're severely limited in tuning the trade-off between storage costs and processing costs.

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.

The biggest problem, though, 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. In fact, most of the implementation of a key/value store is dedicated to these features you don't need at all. 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. Additionally, 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.

4.2.2 Distributed filesystems

It turns out there's a type of technology that you're already intimately familiar with that's a perfect fit for batch layer storage: filesystem.

Files are sequences of bytes, and the most efficient way to consume them 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). You have full control over the bytes of a file, and 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 abilit to tune storage cost versus processing cost. On top of that, filesystems implement fine-grained permission system, which are perfect for enforcing immutability.

The problem with a regular filesystems 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. But it turns out that there's a class of technologies called distributed filesystems that is quite is similar to the filesystems you're familiar with, except they spread their storage across a cluster of computers. They scale by adding more machines to the cluster. Distributed fielsystems 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.

There are some differences between distributed fielsystems and regular filesystems. The operations you can do with a distributed filesystem are often more limited than you can do with a regular filesystem. For instance, you may not be able to write to the middle of a file or even modify a file at all after creation. 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).

4.3 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). We feel the design of HDFS is sufficiently representative of how distributed filesystems work to demonstrate how such a tool can be used for the batch layer.

HDFS and Hadoop Map Reduce are the two prongs of the Hadoop project: a Java framework for distributed storage and distributed processing of large amounts of data. 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. Hadoop is a project of significant size and depth, so we'll only provide a high level description.

In an HDFS cluster, there are two types of nodes: a single namenode and multiple datanodes. When you upload a file to HDFS, the file is first chunked into blocks of a fixed size, typically between 64 MB and 256 MB. Each block is then replicated across multiple datanodes (typically three) that are chosen at random. The namenode keeps track of the file-to-block mapping and where each block is located. This design is 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.

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

4.4 Storing a master dataset with a distributed filesystem

Distributed filesystems vary in the kinds of operations they permit. Some distributed fielsystems let you modify existing files, and others don't. Some allow you to append to existing files, and some don't have the feature.

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

4.5 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. For example, 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 access data relevant to its computation. This process is called vertical partitioning, and it can greatly contribute to making the batch layer more efficient. While it's not strictly necessary for the batch layer, as the batch layer is capable of looking at all the data an once and filtering out what it doesn't need, vertical partitioning enables larges performance gains, so it's important to know how to use the technique.

Vertically partitioning data on a distributed filesystem can be down by sorting your data into separate folders. For example, suppose you're storing login information on a distributed filesystem. Each login contains a username, IP address, and timestamp. 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.

4.6 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.

Let's start with appending to a master dataset. 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. 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.

There's another problem. 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. 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.

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

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. But when you're just using a files-and-folders as part of the append operation. But when you're just using a files-and-folders API directly, it's very easy to make a mistake and break the vertical partitioning constrains to 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.

4.7 Storing the SuperWebAnalytics.com master dataset on a distributed filesystem

Let's now look at how you can make use of a distributed filesystem to store the master dataset for SuperWebAnalytics.com

When you last left this project, you had created a graph schema to represent the dataset. Every edge and property is represented via its own independent DataUnit.

A key observation is that a graph schema provides a natural vertical partitioning of the data. You can store all edge and property types in their own folders. Vertically partitioning the data this way lets you efficiently run computation that only look at certain properties and edges.