1. Data Modeling in Hadoop

At its core, Hadoop is a distributed data store that provides a platform for implementing powerful parallel processing frameworks. The reliability of this data store when it comes to storing massive volumes of data, coupled with its flexibility in running multiple processing frameworks makes it an ideal choice for your data hub. This characteristic of Hadoop means that you can store any type of data as is, without placing any constraints on how that data is processed.

A common term one hears in the context of Hadoop is *Schema-on-Read*. This simply refers to the fact that raw, unprocessed data can be loaded into Hadoop, with the structure imposed at processing time based on the requirements of the processing application.

This is different from *Schema-on-Write*, which is generally used with traditional data management systems. Such systems require the schema of the data store to be defined before the data can be loaded. This leads to lengthy cycles of analysis, data modeling, data transformation, loading, testing, and so on before data can be accessed. Furthermore, if a wrong decision is made or requirements change, this cycle must start again. When the application or structure of data is not as well understood, the agility provided by the Schema-on-Read pattern can provide invaluable insights on data not previously accessible.

Relational databases and data warehouses are often a good fit for well-understood and frequently accessed queries and reports on high-value data. Increasingly, though, Hadoop is taking on many of these workloads, particularly for queries that need to operate on volumes of data that are not economically or technically practical to process with traditional systems.

Although being able to store all of your raw data is a powerful feature, there are still many factors that you should take into consideration before dumping your data into Hadoop. These considerations include:

Data storage formats

There are a number of file formats and compression formats supported on Hadoop. Each has particular strengths that make it better suited to specific applications. Additionally, although Hadoop provides the Hadoop Distributed File System (HDFS) for storing data, there are several commonly used systems implemented on top of HDFS, such as HBase for additional data access functionality and Hive for additional data management functionality. Such systems need to be taken into consideration as well.

Multitenancy
It's common for clusters to host multiple users, groups, and application types. Supporting multitenant clusters involves a number of important considerations when you are planning how data will be stored and managed.

Schema design
Despite the schema-less nature of Hadoop, there are still important considerations to take into account around the structure of data stored in Hadoop. This includes directory structures for data loaded into HDFS as well as the output of data processing and analysis. This also includes the schemas of objects stored in systems such as HBase and Hive.

Metadata management
As with any data management system, metadata related to the stored data is often as important as the data itself. Understanding and making decisions related to metadata management are critical.

We'll discuss these items in this chapter. Note that these considerations are fundamental to architecting applications on Hadoop, which is why we're covering them early in the book.

Another important factor when you're making storage decisions with Hadoop, but one that's beyond the scope of this book, is security and its associated considerations. This includes decisions around authentication, fine-grained access control, and encryption—both for data on the wire and data at rest. For a comprehensive discussion of security with Hadoop, see *Hadoop Security* by Ben Spivey and Joey Echeverria (O'Reilly).

Data Storage Options

One of the most fundamental decisions to make when you are architecting a solution on Hadoop is determining how data will be stored in Hadoop. There is no such thing as a standard data storage format in Hadoop. Just as with a standard filesystem, Hadoop allows for storage of data in any format, whether it's text, binary, images, or something else. Hadoop also provides built-in support for a number of formats optimized for Hadoop storage and processing. This means users have complete control and a number of options for how data is stored in Hadoop. This applies to not just the raw data being ingested, but also intermediate data generated during data processing and derived data that's the result of data processing. This, of course, also means that there are a number of decisions involved in determining how to optimally store your data. Major considerations for Hadoop data storage include:

File format
There are multiple formats that are suitable for data stored in Hadoop. These include plain text or Hadoop-specific formats such as SequenceFile. There are also more complex but more functionally rich options, such as Avro and Parquet. These different formats have different strengths that make them more or less suitable depending on the application and source-data types. It's possible to create your own custom file format in Hadoop, as well.

Compression
This will usually be a more straightforward task than selecting file formats, but it's still an important factor to consider. Compression codecs commonly used with Hadoop have different characteristics; for example, some codecs compress and uncompress faster but don't compress as aggressively, while other codecs create smaller files but take longer to compress and uncompress, and not surprisingly require more CPU. The ability to split compressed files is also a very important consideration when you're working with data stored in Hadoop—we'll discuss splittability considerations further later in the chapter.

Data storage system

While all data in Hadoop rests in HDFS, there are decisions around what the underlying storage manager should be—for example, whether you should use HBase or HDFS directly to store the data. Additionally, tools such as Hive and Impala allow you to define additional structure around your data in Hadoop.

Before beginning a discussion on data storage options for Hadoop, we should note a couple of things:

- We'll cover different storage options in this chapter, but more in-depth discussions on best practices for data storage are deferred to later chapters. For example, when we talk about ingesting data into Hadoop we'll talk more about considerations for storing that data.
- Although we focus on HDFS as the Hadoop filesystem in this chapter and throughout the book, we'd be remiss in not mentioning work to enable alternate filesystems with Hadoop. This includes open source filesystems such as GlusterFS and the Quantcast File System, and commercial alternatives such as Isilon OneFS and NetApp. Cloud-based storage systems such as Amazon's Simple Storage System (S3) are also becoming common. The filesystem might become yet another architectural consideration in a Hadoop deployment. This should not, however, have a large impact on the underlying considerations that we're discussing here.

Standard File Formats

We'll start with a discussion on storing standard file formats in Hadoop—for example, text files (such as comma-separated value [CSV] or XML) or binary file types (such as images). In general, it's preferable to use one of the Hadoop-specific container formats discussed next for storing data in Hadoop, but in many cases you'll want to store source data in its raw form. As noted before, one of the most powerful features of Hadoop is the ability to store all of your data regardless of format. Having online access to data in its raw, source form—"full fidelity" data—means it will always be possible to perform new processing and analytics with the data as requirements change. The following discussion provides some considerations for storing standard file formats in Hadoop.

Text data

A very common use of Hadoop is the storage and analysis of logs such as web logs and server logs. Such text data, of course, also comes in many other forms: CSV files, or unstructured data such as emails. A primary consideration when you are storing text data in Hadoop is the organization of the files in the filesystem, which we'll discuss more in the section "HDFS Schema Design". Additionally, you'll want to select a compression format for the files, since text files can very quickly consume considerable space on your Hadoop cluster. Also, keep in mind that there is an overhead of type conversion associated with storing data in text format. For example, storing 1234 in a text file and using it as an integer requires a string-to-integer conversion during reading, and vice versa during writing. It also takes up more space to store *1234* as text than as an integer. This overhead adds up when you do many such conversions and store large amounts of data.

Selection of compression format will be influenced by how the data will be used. For archival purposes you may choose the most compact compression available, but if the data will be used in processing jobs such as MapReduce, you'll likely want to select a splittable format. Splittable formats enable Hadoop to split files into chunks for processing, which is critical to efficient parallel processing. We'll discuss compression types and considerations, including the concept of splittability, later in this chapter.

Note also that in many, if not most cases, the use of a container format such as SequenceFiles or Avro will provide advantages that make it a preferred format for most file types, including text; among other things, these container formats provide functionality to support splittable compression. We'll also be covering these container formats later in this chapter.

Structured text data

A more specialized form of text files is structured formats such as XML and JSON. These types of formats can present special challenges with Hadoop since splitting XML and JSON files for processing is tricky, and Hadoop does not provide a built-in InputFormat for either. JSON presents even greater challenges than XML, since there are no tokens to mark the beginning or end of a record. In the case of these formats, you have a couple of options:

- Use a container format such as Avro. Transforming the data into Avro can provide a compact and efficient way to store and process the data.
- Use a library designed for processing XML or JSON files. Examples of this for XML include XMLLoader in the PiggyBank library for Pig. For JSON, the Elephant Bird project provides the LzoJsonInputFormat. For more details on processing these formats, see the book *Hadoop in Practice* by Alex Holmes (Manning), which provides several examples for processing XML and JSON files with MapReduce.

Binary data

Although text is typically the most common source data format stored in Hadoop, you can also use Hadoop to process binary files such as images. For most cases of storing and processing binary files in Hadoop, using a container format such as SequenceFile is preferred. If the splittable unit of binary data is larger than 64 MB, you may consider putting the data in its own file, without using a container format.

Hadoop File Types

There are several Hadoop-specific file formats that were specifically created to work well with MapReduce. These Hadoop-specific file formats include file-based data structures such as sequence files, serialization formats like Avro, and columnar formats such as RCFile and Parquet. These file formats have differing strengths and weaknesses, but all share the following characteristics that are important for Hadoop applications:

Splittable compression
These formats support common compression formats and are also splittable. We'll discuss splittability more in the section "Compression", but note that the ability to split files can be a key consideration for storing data in Hadoop because it allows large files to be split for input to MapReduce and other types of jobs. The ability to split a file for processing by multiple tasks is of course a fundamental part of parallel processing, and is also key to leveraging Hadoop's data locality feature.

Agnostic compression
The file can be compressed with any compression codec, without readers having to know the codec. This is possible because the codec is stored in the header metadata of the file format.

We'll discuss the file-based data structures in this section, and subsequent sections will cover serialization formats and columnar formats.

File-based data structures

The *SequenceFile* format is one of the most commonly used file-based formats in Hadoop, but other file-based formats are available, such as MapFiles, SetFiles, ArrayFiles, and BloomMapFiles. Because these formats were specifically designed to work with MapReduce, they offer a high level of integration for all forms of MapReduce jobs, including those run via Pig and Hive. We'll cover the SequenceFile format here, because that's the format most commonly employed in implementing Hadoop jobs. For a more complete discussion of the other formats, refer to *Hadoop: The Definitive Guide*.

SequenceFiles store data as binary key-value pairs. There are three formats available for records stored within SequenceFiles:

Uncompressed
For the most part, uncompressed SequenceFiles don't provide any advantages over their compressed alternatives, since they're less efficient for input/output (I/O) and take up more space on disk than the same data in compressed form.

Record-compressed
This format compresses each record as it’s added to the file.

Block-compressed
This format waits until data reaches block size to compress, rather than as each record is added. Block compression provides better compression ratios compared to record-compressed SequenceFiles, and is generally the preferred compression option for SequenceFiles. Also, the reference to *block* here is unrelated to the HDFS or filesystem block. A *block* in block compression refers to a group of records that are compressed together within a single HDFS block.

Regardless of format, every SequenceFile uses a common header format containing basic metadata about the file, such as the compression codec used, key and value class names, user-defined metadata, and a randomly generated sync marker. This sync marker is also written into the body of the file to allow for seeking to random points in the file, and is key to facilitating splittability. For example, in the case of block compression, this sync marker will be written before every block in the file.

SequenceFiles are well supported within the Hadoop ecosystem, however their support outside of the ecosystem is limited. They are also only supported in Java. A common use case for SequenceFiles is as a container for smaller files. Storing a large number of small files in Hadoop can cause a couple of issues. One is excessive memory use for the NameNode, because metadata for each file stored in HDFS is held in memory. Another potential issue is in processing data in these files—many small files can lead to many processing tasks, causing excessive overhead in processing. Because Hadoop is optimized for large files, packing smaller files into a SequenceFile makes the storage and processing of these files much more efficient. For a more complete discussion of the small files problem with Hadoop and how SequenceFiles provide a solution, refer to *Hadoop: The Definitive Guide*.

Figure 1-1 shows an example of the file layout for a SequenceFile using block compression. An important thing to note in this diagram is the inclusion of the sync marker before each block of data, which allows readers of the file to seek to block boundaries.

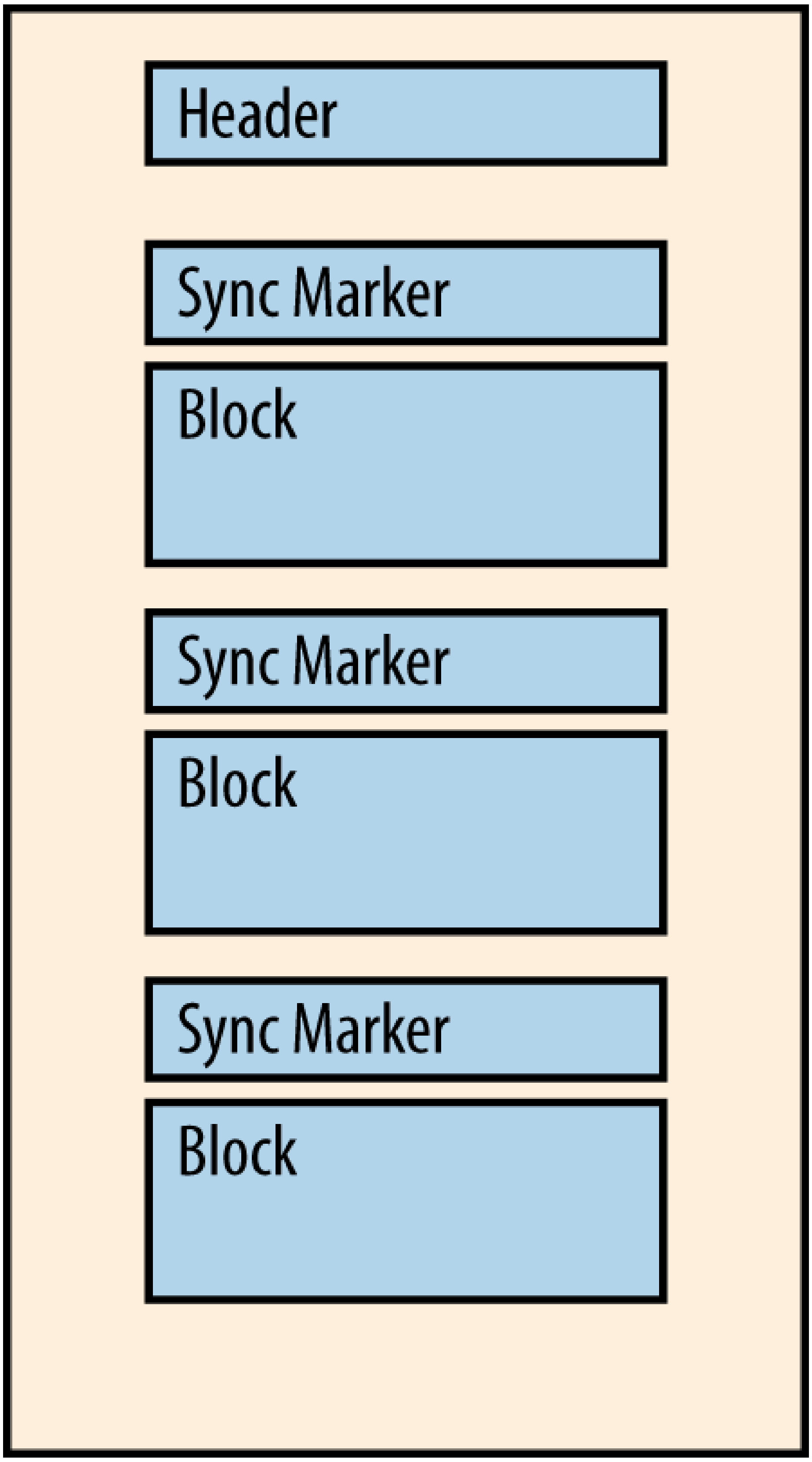

Figure 1-1. An example of a SequenceFile using block compression

Serialization Formats

Serialization refers to the process of turning data structures into byte streams either for storage or transmission over a network. Conversely, *deserialization* is the process of converting a byte stream back into data structures. Serialization is core to a distributed processing system such as Hadoop, since it allows data to be converted into a format that can be efficiently stored as well as transferred across a network connection. Serialization is commonly associated with two aspects of data processing in distributed systems: interprocess communication (remote procedure calls, or RPC) and data storage. For purposes of this discussion we're not concerned with RPC, so we'll focus on the data storage aspect in this section.

The main serialization format utilized by Hadoop is Writables. Writables are compact and fast, but not easy to extend or use from languages other than Java. There are, however, other serialization frameworks seeing increased use within the Hadoop ecosystem, including Thrift, Protocol Buffers, and Avro. Of these, Avro is the best suited, because it was specifically created to address limitations of Hadoop Writables. We'll examine Avro in more detail, but let's first briefly cover Thrift and Protocol Buffers.

Thrift

Thrift was developed at Facebook as a framework for implementing cross-language interfaces to services. Thrift uses an Interface Definition Language (IDL) to define interfaces, and uses an IDL file to generate stub code to be used in implementing RPC clients and servers that can be used across languages. Using Thrift allows us to implement a single interface that can be used with different languages to access different underlying systems. The Thrift RPC layer is very robust, but for this chapter, we're only concerned with Thrift as a serialization framework. Although sometimes used for data serialization with Hadoop, Thrift has several drawbacks: it does not support internal compression of records, it's not splittable, and it lacks native MapReduce support. Note that there are externally available libraries such as the Elephant Bird project to address these drawbacks, but Hadoop does not provide native support for Thrift as a data storage format.

Protocol Buffers

The Protocol Buffer (protobuf) format was developed at Google to facilitate data exchange between services written in different languages. Like Thrift, protobuf structures are defined via an IDL, which is used to generate stub code for multiple languages. Also like Thrift, Protocol Buffers do not support internal compression of records, are not splittable, and have no native MapReduce support. But also like Thrift, the Elephant Bird project can be used to encode protobuf records, providing support for MapReduce, compression, and splittability.

Avro

Avro is a language-neutral data serialization system designed to address the major downside of Hadoop Writables: lack of language portability. Like Thrift and Protocol Buffers, Avro data is described through a language-independent schema. Unlike Thrift and Protocol Buffers, code generation is optional with Avro. Since Avro stores the schema in the header of each file, it’s self-describing and Avro files can easily be read later, even from a different language than the one used to write the file. Avro also provides better native support for MapReduce since Avro data files are compressible and splittable. Another important feature of Avro that makes it superior to SequenceFiles for Hadoop applications is support for *schema evolution*; that is, the schema used to read a file does not need to match the schema used to write the file. This makes it possible to add new fields to a schema as requirements change.

Avro schemas are usually written in JSON, but may also be written in Avro IDL, which is a C-like language. As just noted, the schema is stored as part of the file metadata in the file header. In addition to metadata, the file header contains a unique sync marker. Just as with SequenceFiles, this sync marker is used to separate blocks in the file, allowing Avro files to be splittable. Following the header, an Avro file contains a series of blocks containing serialized Avro objects. These blocks can optionally be compressed, and within those blocks, types are stored in their native format, providing an additional boost to compression. At the time of writing, Avro supports Snappy and Deflate compression.

While Avro defines a small number of primitive types such as Boolean, int, float, and string, it also supports complex types such as array, map, and enum.

Columnar Formats

Until relatively recently, most database systems stored records in a row-oriented fashion. This is efficient for cases where many columns of the record need to be fetched. For example, if your analysis heavily relied on fetching all fields for records that belonged to a particular time range, row-oriented storage would make sense. This option can also be more efficient when you're writing data, particularly if all columns of the record are available at write time because the record can be written with a single disk seek. More recently, a number of databases have introduced columnar storage, which provides several benefits over earlier row-oriented systems:

- Skips I/O and decompression (if applicable) on columns that are not a part of the query.
- Works well for queries that only access a small subset of columns. If many columns are being accessed, then row-oriented is generally preferable.
- Is generally very efficient in terms of compression on columns because entropy within a column is lower than entropy within a block of rows. In other words, data is more similar within the same column, than it is in a block of rows. This can make a huge difference especially when the column has few distinct values.
- Is often well suited for data-warehousing-type applications where users want to aggregate certain columns over a large collection of records.

Not surprisingly, columnar file formats are also being utilized for Hadoop applications. Columnar file formats supported on Hadoop include the RCFile format, which has been popular for some time as a Hive format, as well as newer formats such as the Optimized Row Columnar (ORC) and Parquet, which are described next.

RCFile

The RCFile format was developed specifically to provide efficient processing for MapReduce applications, although in practice it's only seen use as a Hive storage format. The RCFile format was developed to provide fast data loading, fast query processing, and highly efficient storage space utilization. The RCFile format breaks files into row splits, then within each split uses column-oriented storage.

Although the RCFile format provides advantages in terms of query and compression performance compared to SequenceFiles, it also has some deficiencies that prevent optimal performance for query times and compression. Newer columnar formats such as ORC and Parquet address many of these deficiencies, and for most newer applications, they will likely replace the use of RCFile. RCFile is still a fairly common format used with Hive storage.

ORC

The ORC format was created to address some of the shortcomings with the RCFile format, specifically around query performance and storage efficiency. The ORC format provides the following features and benefits, many of which are distinct improvements over RCFile:

- Provides lightweight, always-on compression provided by type-specific readers and writers. ORC also supports the use of zlib, LZO, or Snappy to provide further compression.
- Allows predicates to be pushed down to the storage layer so that only required data is brought back in queries.
- Supports the Hive type model, including new primitives such as decimal and complex types.

- Is a splittable storage format.

A drawback of ORC as of this writing is that it was designed specifically for Hive, and so is not a general-purpose storage format that can be used with non-Hive MapReduce interfaces such as Pig or Java, or other query engines such as Impala. Work is under way to address these shortcomings, though.

Parquet

Parquet shares many of the same design goals as ORC, but is intended to be a general-purpose storage format for Hadoop. In fact, ORC came after Parquet, so some could say that ORC is a Parquet wannabe. As such, the goal is to create a format that's suitable for different MapReduce interfaces such as Java, Hive, and Pig, and also suitable for other processing engines such as Impala and Spark. Parquet provides the following benefits, many of which it shares with ORC:

- Similar to ORC files, Parquet allows for returning only required data fields, thereby reducing I/O and increasing performance.
- Provides efficient compression; compression can be specified on a per-column level.
- Is designed to support complex nested data structures.
- Stores full metadata at the end of files, so Parquet files are self-documenting.
- Fully supports being able to read and write to with Avro and Thrift APIs.
- Uses efficient and extensible encoding schemas—for example, bit-packaging/run length encoding (RLE).

Avro and Parquet

Over time, we have learned that there is great value in having a single interface to all the files in your Hadoop cluster. And if you are going to pick one file format, you will want to pick one with a schema because, in the end, most data in Hadoop will be structured or semistructured data.

So if you need a schema, Avro and Parquet are great options. However, we don't want to have to worry about making an Avro version of the schema and a Parquet version. Thankfully, this isn't an issue because Parquet can be read and written to with Avro APIs and Avro schemas.

This means we can have our cake and eat it too. We can meet our goal of having one interface to interact with our Avro and Parquet files, and we can have a block and columnar options for storing our data.

Compression

Compression is another important consideration for storing data in Hadoop, not just in terms of reducing storage requirements, but also to improve data processing performance. Because a major overhead in processing large amounts of data is disk and network I/O, reducing the amount of data that needs to be read and written to disk can significantly decrease overall processing time. This includes compression of source data, but also the intermediate data generated as part of data processing (e.g., MapReduce jobs). Although compression adds CPU load, for most cases this is more than offset by the savings in I/O.

Although compression can greatly optimize processing performance, not all compression formats supported on Hadoop are splittable. Because the MapReduce framework splits data for input to multiple tasks, having a nonsplittable compression format is an impediment to efficient processing. If files cannot be split, that means the entire file needs to be passed to a single MapReduce task, eliminating the advantages of parallelism and data locality that Hadoop provides. For this reason, splittability is a major consideration in choosing a compression format as well as file format. We'll discuss the various compression formats available for Hadoop, and some considerations in choosing between them.

Snappy

Snappy is a compression codec developed at Google for high compression speeds with reasonable compression. Although Snappy doesn’t offer the best compression sizes, it does provide a good trade-off between speed and size. Processing performance with Snappy can be significantly better than other compression formats. It’s important to note that Snappy is intended to be used with a container format like SequenceFiles or Avro, since it’s not inherently splittable.

LZO

LZO is similar to Snappy in that it’s optimized for speed as opposed to size. Unlike Snappy, LZO compressed files are splittable, but this requires an additional indexing step. This makes LZO a good choice for things like plain-text files that are not being stored as part of a container format. It should also be noted that LZO’s license prevents it from being distributed with Hadoop and requires a separate install, unlike Snappy, which can be distributed with Hadoop.

Gzip

Gzip provides very good compression performance (on average, about 2.5 times the compression that’d be offered by Snappy), but its write speed performance is not as good as Snappy’s (on average, it’s about half of Snappy’s). Gzip usually performs almost as well as Snappy in terms of read performance. Gzip is also not splittable, so it should be used with a container format. Note that one reason Gzip is sometimes slower than Snappy for processing is that Gzip compressed files take up fewer blocks, so fewer tasks are required for processing the same data. For this reason, using smaller blocks with Gzip can lead to better performance.

bzip2

bzip2 provides excellent compression performance, but can be significantly slower than other compression codecs such as Snappy in terms of processing performance. Unlike Snappy and Gzip, bzip2 is inherently splittable. In the examples we have seen, bzip2 will normally compress around 9% better than GZip, in terms of storage space. However, this extra compression comes with a significant read/write performance cost. This performance difference will vary with different machines, but in general bzip2 is about 10 times slower than GZip. For this reason, it’s not an ideal codec for Hadoop storage, unless your primary need is reducing the storage footprint. One example of such a use case would be using Hadoop mainly for active archival purposes.

Compression recommendations

In general, any compression format can be made splittable when used with container file formats (Avro, SequenceFiles, etc.) that compress blocks of records or each record individually. If you are doing compression on the entire file without using a container file format, then you have to use a compression format that inherently supports splitting (e.g., bzip2, which inserts synchronization markers between blocks).

Here are some recommendations on compression in Hadoop:

- Enable compression of MapReduce intermediate output. This will improve performance by decreasing the amount of intermediate data that needs to be read and written to and from disk.
- Pay attention to how data is ordered. Often, ordering data so that like data is close together will provide better compression levels. Remember, data in Hadoop file formats is compressed in chunks, and it is the entropy of those chunks that will determine the final compression. For example, if you have stock ticks with the columns timestamp, stock ticker, and stock price, then ordering the data by a repeated field, such as stock ticker, will provide better compression than ordering by a unique field, such as time or stock price.
- Consider using a compact file format with support for splittable compression, such as Avro. Figure 1-2 illustrates how Avro or SequenceFiles support splittability with otherwise nonsplittable compression formats. A single HDFS block can contain multiple Avro or SequenceFile blocks. Each of the Avro or SequenceFile blocks can be compressed and decompressed individually and independently

of any other Avro/SequenceFile blocks. This, in turn, means that each of the HDFS blocks can be compressed and decompressed individually, thereby making the data splittable.

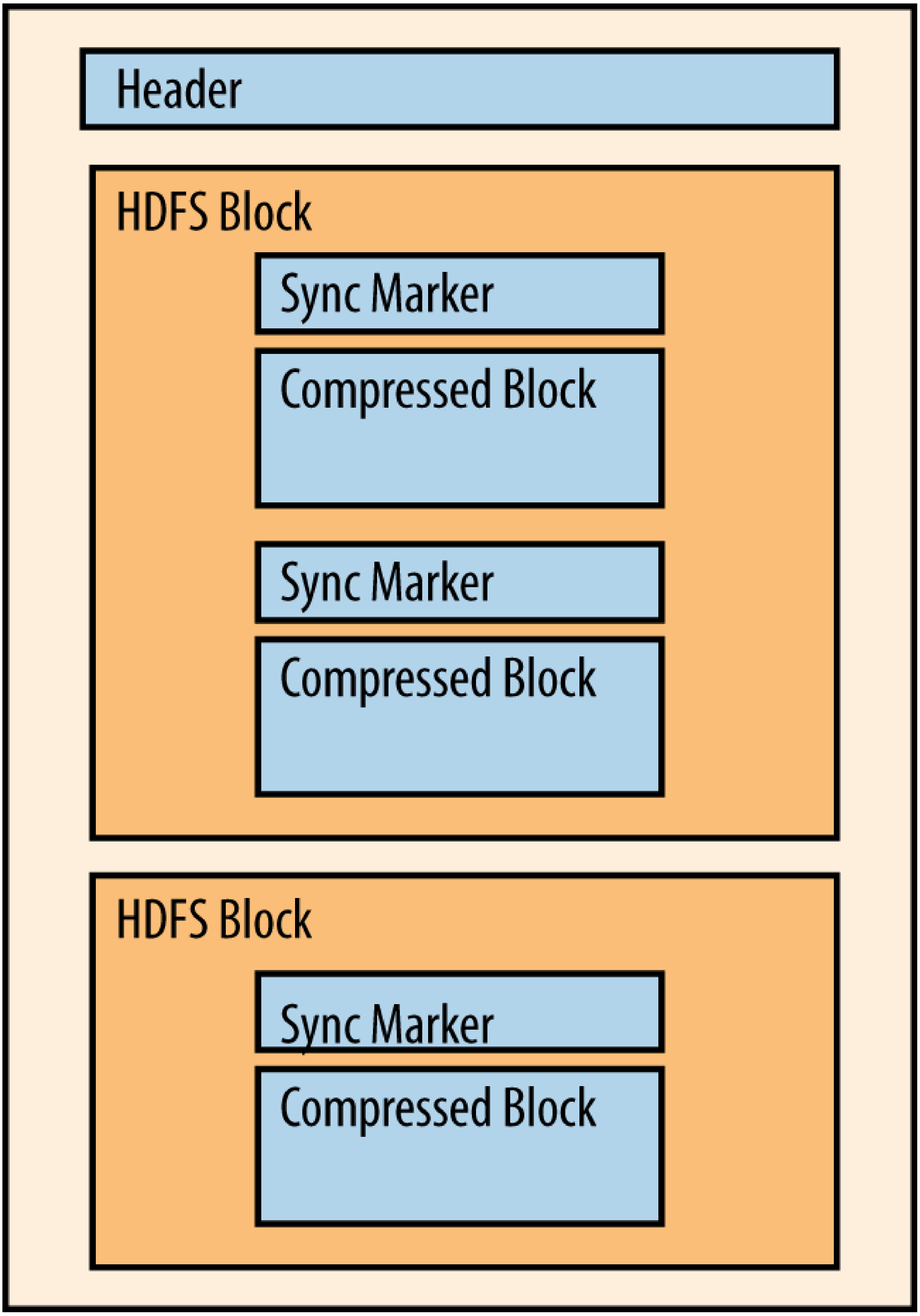

Figure 1-2. An example of compression with Avro

HDFS Schema Design

As pointed out in the previous section, HDFS and HBase are two very commonly used storage managers. Depending on your use case, you can store your data in HDFS or HBase (which internally stores it on HDFS).

In this section, we will describe the considerations for good schema design for data that you decide to store in HDFS directly. As mentioned earlier, Hadoop's Schema-on-Read model does not impose any requirements when loading data into Hadoop. Data can be simply ingested into HDFS by one of many methods (which we will discuss further in Chapter 2) without our having to associate a schema or preprocess the data.

While many people use Hadoop for storing and processing unstructured data (such as images, videos, emails, or blog posts) or semistructured data (such as XML documents and logfiles), some order is still desirable. This is especially true since Hadoop often serves as a data hub for the entire organization, and the data stored in HDFS is intended to be shared among many departments and teams. Creating a carefully structured and organized repository of your data will provide many benefits. To list a few:

- A standard directory structure makes it easier to share data between teams working with the same data sets.
- It also allows for enforcing access and quota controls to prevent accidental deletion or corruption.
- Oftentimes, you'd "stage" data in a separate location before all of it was ready to be processed. Conventions regarding staging data will help ensure that partially loaded data will not get accidentally processed as if it were complete.
- Standardized organization of data allows for reuse of some code that may process the data.

- Some tools in the Hadoop ecosystem sometimes make assumptions regarding the placement of data. It is often simpler to match those assumptions when you are initially loading data into Hadoop.

The details of the data model will be highly dependent on the specific use case. For example, data warehouse implementations and other event stores are likely to use a schema similar to the traditional star schema, including structured fact and dimension tables. Unstructured and semistructured data, on the other hand, are likely to focus more on directory placement and metadata management.

The important points to keep in mind when designing the schema, regardless of the project specifics, are:

- Develop standard practices and enforce them, especially when multiple teams are sharing the data.
- Make sure your design will work well with the tools you are planning to use. For example, the version of Hive you are planning to use may only support table partitions on directories that are named a certain way. This will impact the schema design in general and how you name your table subdirectories, in particular.
- Keep usage patterns in mind when designing a schema. Different data processing and querying patterns work better with different schema designs. Understanding the main use cases and data retrieval requirements will result in a schema that will be easier to maintain and support in the long term as well as improve data processing performance.

Location of HDFS Files

To talk in more concrete terms, the first decisions to make when you’re designing an HDFS schema is the location of the files. Standard locations make it easier to find and share data between teams. The following is an example HDFS directory structure that we recommend. This directory structure simplifies the assignment of permissions to various groups and users:

/user/<username>
Data, JARs, and configuration files that belong only to a specific user. This is usually scratch type data that the user is currently experimenting with but is not part of a business process. The directories under */user* will typically only be readable and writable by the users who own them.

/etl
Data in various stages of being processed by an ETL (extract, transform, and load) workflow. The */etl* directory will be readable and writable by ETL processes (they typically run under their own user) and members of the ETL team. The */etl* directory tree will have subdirectories for the various groups that own the ETL processes, such as business analytics, fraud detection, and marketing. The ETL workflows are typically part of a larger application, such as clickstream analysis or recommendation engines, and each application should have its own subdirectory under the */etl* directory. Within each application-specific directory, you would have a directory for each ETL process or workflow for that application. Within the workflow directory, there are subdirectories for each of the data sets. For example, if your Business Intelligence (BI) team has a clickstream analysis application and one of its processes is to aggregate user preferences, the recommended name for the directory that contains the data would be */etl/BI/clickstream/aggregate_preferences*. In some cases, you may want to go one level further and have directories for each stage of the process: *input* for the landing zone where the data arrives, *processing* for the intermediate stages (there may be more than one *processing* directory), *output* for the final result, and *bad* where records or files that are rejected by the ETL process land for manual troubleshooting. In such cases, the final structure will look similar to */etl/<group>/<application>/<process>/{input,processing,output,bad}*

/tmp
Temporary data generated by tools or shared between users. This directory is typically cleaned by an automated process and does not store long-term data. This directory is typically readable and writable by everyone.

/data
Data sets that have been processed and are shared across the organization. Because these are often critical data sources for analysis that drive business decisions, there are often controls around who can read and write this data. Very often user access is read-only, and data is written by automated (and audited) ETL processes. Since data in */data* is typically business-critical, only automated ETL processes are typically allowed to write them—so changes are controlled and audited. Different business groups will have read access to different directories under */data*, depending on their reporting and processing needs. Since */data* serves as the location for shared processed data sets, it will contain subdirectories for each data set. For example, if you were storing all orders of a pharmacy in a table called *medication_orders*, we recommend that you store this data set in a directory named */data/medication_orders*.

/app
Includes everything required for Hadoop applications to run, except data. This includes JAR files, Oozie workflow definitions, Hive HQL files, and more. The application code directory */app* is used for application artifacts such as JARs for Oozie actions or Hive user-defined functions (UDFs). It is not always necessary to store such application artifacts in HDFS, but some Hadoop applications such as Oozie and Hive require storing shared code and configuration on HDFS so it can be used by code executing on any node of the cluster. This directory should have a subdirectory for each group and application, similar to the structure used in */etl*. For a given application (say, Oozie), you would need a directory for each version of the artifacts you decide to store in HDFS, possibly tagging, via a symlink in HDFS, the latest artifact as *latest* and the currently used one as *current*. The directories containing the binary artifacts would be present under these versioned directories. This will look similar to: */app/<group>/<application>/<version>/<artifact directory>/<artifact>*. To continue our previous example, the JAR for the latest build of our aggregate preferences process would be in a directory structure like */app/BI/clickstream/latest/aggregate_preferences/uber-aggregate-preferences.jar*.

/metadata
Stores metadata. While most table metadata is stored in the Hive metastore, as described later in the "Managing Metadata", some extra metadata (for example, Avro schema files) may need to be stored in HDFS. This directory would be the best location for storing such metadata. This directory is typically readable by ETL jobs but writable by the user used for ingesting data into Hadoop (e.g., Sqoop user). For example, the Avro schema file for a data set called `movie` may exist at a location like this: */metadata/movielens/movie/movie.avsc*. We will discuss this particular example in more detail in Chapter 10.

Advanced HDFS Schema Design

Once the broad directory structure has been decided, the next important decision is how data will be organized into files. While we have already talked about how the format of the ingested data may not be the most optimal format for storing it, it's important to note that the default organization of ingested data may not be optimal either. There are a few strategies to best organize your data. We will talk about partitioning, bucketing, and denormalizing here.

Partitioning

Partitioning a data set is a very common technique used to reduce the amount of I/O required to process the data set. When you're dealing with large amounts of data, the savings brought by reducing I/O can be quite significant. Unlike traditional data warehouses, however, HDFS doesn't store indexes on the data. This lack of indexes plays a large role in speeding up data ingest, but it also means that every query will have to read the entire data set even when you're processing only a small subset of the data (a pattern called *full table scan*). When the data sets grow very big, and queries only require access to subsets of data, a very good solution is to break up the data set into smaller subsets, or partitions. Each of these partitions would be present in a subdirectory of the directory containing the entire data set. This will allow queries to read only the specific partitions (i.e., subdirectories) they require, reducing the amount of I/O and improving query times significantly.

For example, say you have a data set that stores all the orders for various pharmacies in a data set called *medication_orders*, and you'd like to check order history for just one physician over the past three months. Without partitioning, you'd need to read the entire data set and filter out all the records that don't pertain to the query.

However, if we were to partition the entire orders data set so each partition included only a single day's worth of data, a query looking for information from the past three months will only need to read 90 or so partitions and not the entire data set.

When placing the data in the filesystem, you should use the following directory format for partitions: *<data set name>/<partition_column_name=partition_column_value>/{files}*. In our example, this translates to: *medication_orders/date=20131101/{order1.csv, order2.csv}*

This directory structure is understood by various tools, like HCatalog, Hive, Impala, and Pig, which can leverage partitioning to reduce the amount of I/O required during processing.

Bucketing

Bucketing is another technique for decomposing large data sets into more manageable subsets. It is similar to the hash partitions used in many relational databases. In the preceding example, we could partition the orders data set by date because there are a large number of orders done daily and the partitions will contain large enough files, which is what HDFS is optimized for. However, if we tried to partition the data by physician to optimize for queries looking for specific physicians, the resulting number of partitions may be too large and resulting files may be too small in size. This leads to what's called the *small files problem*. As detailed in "File-based data structures", storing a large number of small files in Hadoop can lead to excessive memory use for the NameNode, since metadata for each file stored in HDFS is held in memory. Also, many small files can lead to many processing tasks, causing excessive overhead in processing.

The solution is to *bucket* by *physician*, which will use a hashing function to map physicians into a specified number of buckets. This way, you can control the size of the data subsets (i.e., buckets) and optimize for query speed. Files should not be so small that you'll need to read and manage a huge number of them, but also not so large that each query will be slowed down by having to scan through huge amounts of data. A good average bucket size is a few multiples of the HDFS block size. Having an even distribution of data when hashed on the bucketing column is important because it leads to consistent bucketing. Also, having the number of buckets as a power of two is quite common.

An additional benefit of bucketing becomes apparent when you're joining two data sets. The word *join* here is used to represent the general idea of combining two data sets to retrieve a result. Joins can be done via SQL-on-Hadoop systems but also in MapReduce, or Spark, or other programming interfaces to Hadoop.

When both the data sets being joined are bucketed on the join key and the number of buckets of one data set is a multiple of the other, it is enough to join corresponding buckets individually without having to join the entire data sets. This significantly reduces the time complexity of doing a reduce-side join of the two data sets. This is because doing a reduce-side join is computationally expensive. However, when two bucketed data sets are joined, instead of joining the entire data sets together, you can join just the corresponding buckets with each other, thereby reducing the cost of doing a join. Of course, the buckets from both the tables can be joined in parallel. Moreover, because the buckets are typically small enough to easily fit into memory, you can do the entire join in the map stage of a Map-Reduce job by loading the smaller of the buckets in memory. This is called a *map-side join*, and it improves the join performance as compared to a reduce-side join even further. If you are using Hive for data analysis, it should automatically recognize that tables are bucketed and apply this optimization.

If the data in the buckets is *sorted*, it is also possible to use a merge join and not store the entire bucket in memory when joining. This is somewhat faster than a simple bucket join and requires much less memory. Hive supports this optimization as well. Note that it is possible to bucket any table, even when there are no logical partition keys. It is recommended to use both sorting and bucketing on all large tables that are frequently joined together, using the join key for bucketing.

As you can tell from the preceding discussion, the schema design is highly dependent on the way the data will be queried. You will need to know which columns will be used for joining and filtering before deciding on partitioning and bucketing of the data. In cases when there are multiple common query patterns and it is challenging to decide on one partitioning key, you have the option of storing the same data set multiple times, each with different physical organization. This is considered an anti-pattern in relational databases, but with Hadoop, this solution can make sense. For one thing, in Hadoop data is typically write-once, and few updates are expected. Therefore, the usual overhead of keeping duplicated data sets in sync is greatly reduced. In addition, the cost of storage in Hadoop clusters is significantly lower, so there is less concern about wasted disk space. These attributes allow us to trade space for greater query speed, which is often desirable.

Denormalizing

Although we talked about joins in the previous subsections, another method of trading disk space for query performance is denormalizing data sets so there is less of a need to join data sets. In relational databases, data is often stored in *third normal form*. Such a schema is designed to minimize redundancy and provide data integrity by splitting data into smaller tables, each holding a very specific entity. This means that most queries will require joining a large number of tables together to produce final result sets.

In Hadoop, however, joins are often the slowest operations and consume the most resources from the cluster. Reduce-side joins, in particular, require sending entire tables over the network. As we’ve already seen, it is important to optimize the schema to avoid these expensive operations as much as possible. While bucketing and sorting do help there, another solution is to create data sets that are prejoined—in other words, preaggregated. The idea is to minimize the amount of work queries have to do by doing as much as possible in advance, especially for queries or subqueries that are expected to execute frequently. Instead of running the join operations every time a user tries to look at the data, we can join the data once and store it in this form.

Looking at the difference between a typical Online Transaction Processing (OLTP) schema and an HDFS schema of a particular use case, you will see that the Hadoop schema consolidates many of the small dimension tables into a few larger dimensions by joining them during the ETL process. In the case of our pharmacy example, we consolidate frequency, class, admin route, and units into the medications data set, to avoid repeated joining.

Other types of data preprocessing, like aggregation or data type conversion, can be done to speed up processes as well. Since data duplication is a lesser concern, almost any type of processing that occurs frequently in a large number of queries is worth doing once and reusing. In relational databases, this pattern is often known as *Materialized Views*. In Hadoop, you instead have to create a new data set that contains the same data in its aggregated form.

HDFS Schema Design Summary

To recap, in this section we saw how we can use partitioning to reduce the I/O overhead of processing by selectively reading and writing data in particular partitions. We also saw how we can use bucketing to speed up queries that involve joins or sampling, again by reducing I/O. And, finally, we saw how denormalization plays an important role in speeding up Hadoop jobs. Now that we have gone through HDFS schema design, we will go through the schema design concepts for HBase.

HBase Schema Design

In this section, we will describe the considerations for good schema design for data stored in HBase. While HBase is a complex topic with multiple books written about its usage and optimization, this chapter takes a higher-level approach and focuses on leveraging successful design patterns for solving common problems with HBase. For an introduction to HBase, see *HBase: The Definitive Guide*, or *HBase in Action*.

The first thing to understand here is that HBase is not a relational database management system (RDBMS). In fact, in order to gain a better appreciation of how HBase works, it's best to think of it as a huge hash table. Just like a hash table, HBase allows you to associate values with keys and perform fast lookup of the values based on a given key.

There are many details related to how regions and compactions work in HBase, various strategies for ingesting data into HBase, using and understanding block cache, and more that we are glossing over when using the hash table analogy. However, it's still a very apt comparison, primarily because it makes you think of HBase as more of a distributed key-value store instead of an RDBMS. It makes you think in terms of `get`, `put`, `scan`, `increment`, and `delete` requests instead of SQL queries.

Before we focus on the operations that can be done in HBase, let's recap the operations supported by hash tables. We can:

1. Put things in a hash table
2. Get things from a hash table
3. Iterate through a hash table (note that HBase gives us even more power here with *range scans*, which allow specifying a start and end row when scanning)
4. Increment the value of a hash table entry
5. Delete values from a hash table

It also makes sense to answer the question of why you would want to give up SQL for HBase. The value proposition of HBase lies in its scalability and flexibility. HBase is useful in many applications, a popular one being fraud detection, which we will be discussing in more detail in Chapter 9. In general, HBase works for problems that can be solved in a few `get` and `put` requests.

Now that we have a good analogy and background of HBase, let's talk about various architectural considerations that go into designing a good HBase schema.

Row Key

To continue our hash table analogy, a row key in HBase is like the key in a hash table. One of the most important factors in having a well-architected HBase schema is good selection of a row key. Following are some of the ways row keys are used in HBase and how they drive the choice of a row key.

Record retrieval

The row key is the key used when you're retrieving records from HBase. HBase records can have an unlimited number of columns, but only a single row key. This is different from relational databases, where the primary key can be a composite of multiple columns. This means that in order to create a unique row key for records, you may need to combine multiple pieces of information in a single key. An example would be a key of the type `customer_id,order_id,timestamp` as the row key for a row describing an order. In a relational database `customer_id`, `order_id`, and `timestamp` would be three separate columns, but in HBase they need to be combined into a single unique identifier.

Another thing to keep in mind when choosing a row key is that a `get` operation of a single record is the fastest operation in HBase. Therefore, designing the HBase schema in such a way that most common uses of the data will result in a single `get` operation will improve performance. This may mean putting a lot of information into a single record—more than you would do in a relational database. This type of design is called *denormalized*, as distinct from the *normalized* design common in relational databases. For example, in a relational database you will probably store customers in one table, their contact details in another, their orders in a third table, and the order details in yet another table. In HBase you may choose a very wide design where each order record contains all the order details, the customer, and his contact details. All of this data will be retrieved with a single `get`.

Distribution

The row key determines how records for a given table are scattered throughout various regions of the HBase cluster. In HBase, all the row keys are sorted, and each region stores a range of these sorted row keys. Each region is pinned to a region server (i.e., a node in the cluster).

A well-known anti-pattern is to use a timestamp for row keys because it would make most of the `put` and `get` requests focused on a single region and hence a single region server, which somewhat defeats the purpose of having a distributed system. It's usually best to choose row keys so the load on the cluster is fairly distributed. As we will see later in this chapter, one of the ways to resolve this problem is to *salt* the keys. In particular, the combination of device ID and timestamp or reverse timestamp is commonly used to salt the key in machine data.

Block cache

The block cache is a least recently used (LRU) cache that caches data blocks in memory. By default, HBase reads records in chunks of 64 KB from the disk. Each of these chunks is called an *HBase block*. When an HBase block is read from the disk, it will be put into the block cache. However, this insertion into the block cache can be bypassed if you desire. The idea behind the caching is that recently fetched records (and those in the same HBase block as them) have a high likelihood of being requested again in the near future. However, the size of block cache is limited, so it's important to use it wisely.

A poor choice of row key can lead to suboptimal population of the block cache. For example, if you choose your row key to be a hash of some attribute, the HBase block would have records that aren't necessarily *close* to each other in terms of relevance. Consequently, the block cache will be populated with these unrelated records, which will have a very low likelihood of resulting in a cache hit. In such a case, an alternative design would be to salt the first part of the row key with something meaningful that allows records fetched together in the same HBase block to be *close* to each other in the row key sort order. A salt is normally a hash mod on the original key or a part thereof, so it can be generated solely from the original key. We show you an example of salting keys in Chapter 9.

Ability to scan

A wise selection of row key can be used to co-locate related records in the same region. This is very beneficial in range scans since HBase will have to scan only a limited number of regions to obtain the results. On the other hand, if the row key is chosen poorly, range scans may need to scan multiple region servers for the data and subsequently filter the unnecessary records, thereby increasing the I/O requirements for the same request. Also, keep in mind that HBase scan rates are about eight times slower than HDFS scan rates. Therefore, reducing I/O requirements has a significant performance advantage, even more so compared to data stored in HDFS.

Size

The size of your row key will determine the performance of your workload. In general, a shorter row key is better than a longer row key due to lower storage overhead and faster read/write performance. However, longer row keys often lead to better `get`/`scan` properties. Therefore, there is always a trade-off involved in choosing the right row key length.

Let's take a look at an example. Table 1-1 shows a table with three records in HBase.

RowKeyA	Timestamp	ColumnA	ValueA
RowKeyB	Timestamp	ColumnB	ValueB
RowKeyC	Timestamp	ColumnC	ValueC

Table 1-1. Example HBase table

The longer the row key, the more I/O the compression codec has to do in order to store it. The same logic also applies to column names, so in general it’s a good practice to keep the column names short.

Note

HBase can be configured to compress the row keys with Snappy. Since row keys are stored in a sorted order, having row keys that are close to each other when sorted will compress well. This is yet another reason why using a hash of some attribute as a row key is usually not a good idea since the sort order of row keys would be completely arbitrary.

Readability

While this is a very subjective point, given that the row key is so commonly used, its readability is important. It is usually recommended to start with something human-readable for your row keys, even more so if you are new to HBase. It makes it easier to identify and debug issues, and makes it much easier to use the HBase console as well.

Uniqueness

Ensuring that row keys are unique is important, since a row key is equivalent to a key in our hash table analogy. If your selection of row keys is based on a non-unique attribute, your application should handle such cases, and only put your data in HBase with a unique row key.

Timestamp

The second most important consideration for good HBase schema design is understanding and using the timestamp correctly. In HBase, timestamps serve a few important purposes:

- Timestamps determine which records are newer in case of a `put` request to modify the record.
- Timestamps determine the order in which records are returned when multiple versions of a single record are requested.
- Timestamps are also used to decide if a major compaction is going to remove the out-of-date record in question because the time-to-

live (TTL) when compared to the timestamp has elapsed. “Out-of-date” means that the record value has either been overwritten by another `put` or deleted.

By default, when you are writing or updating a record, the timestamp on the cluster node at that time of write/update is used. In most cases, this is also the right choice. However, in some cases it’s not. For example, there may be a delay of hours or days between when a transaction actually happens in the physical world and when it gets recorded in HBase. In such cases, it is common to set the timestamp to when the transaction actually took place.

Hops

The term *hops* refers to the number of synchronized `get` requests required to retrieve the requested information from HBase.

Let’s take an example of a graph of relationships between people, represented in an HBase table. Table 1-2 shows a persons table that contains name, list of friends, and address for each person.

David	Barack, Stephen	10 Downing Street
Barack	Michelle	The White House
Stephen	Barack	24 Sussex Drive

Table 1-2. Persons table

Now, thinking again of our hash table analogy, if you were to find out the address of all of David’s friends, you’d have to do a two-hop request. In the first hop, you’d retrieve a list of all of David’s friends, and in the second hop you’d retrieve the addresses of David’s friends.

Let’s take another example. Table 1-3 shows a students table with an ID and student name. Table 1-4 shows a courses table with a student ID and list of courses that the student is taking.

11001	Bob
11002	David
11003	Charles

Table 1-3. Students table

11001	Chemistry, Physics
11002	Math
11003	History

Table 1-4. Courses table

Now, if you were to find out the list of courses that Charles was taking, you’d have to do a two-hop request. The first hop will retrieve Charles’s student ID from the students table and the second hop will retrieve the list of Charles’ courses using the student ID.

As we alluded to in “Advanced HDFS Schema Design”, examples like the preceding would be a good contender for denormalization because they would reduce the number of hops required to complete the request.

In general, although it’s possible to have multihop requests with HBase, it’s best to avoid them through better schema design (for example, by leveraging denormalization). This is because every hop is a round-trip to HBase that incurs a significant performance overhead.

Tables and Regions

Another factor that can impact performance and distribution of data is the number of tables and number of regions per table in HBase. If not done right, this factor can lead to a significant imbalance in the distribution of load on one or more nodes of the cluster.

Figure 1-3 shows a topology of region servers, regions, and tables in an example three-node HBase cluster.

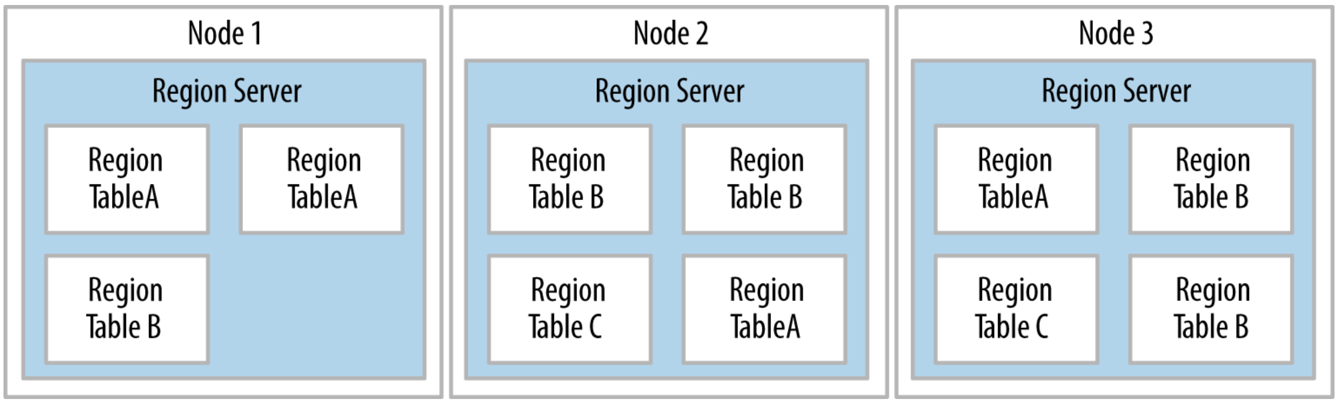

Figure 1-3. Topology of region servers, regions, and tables

The important things to note here are:

- There is one region server per node.
- There are many regions in a region server.
- At any time, a given region is pinned to a particular region server.
- Tables are split into regions and are scattered across region servers. A table must have at least one region.

There are two basic rules of thumb that you can use to select the right number of regions for a given table. These rules demonstrate a trade-off between performance of the `put` requests versus the time it takes to run a compaction on the region.

Put performance

All regions in a region server receiving `put` requests will have to share the region server's memstore. A *memstore* is a cache structure present on every HBase region server. It caches the writes being sent to that region server and sorts them in before it flushes them when certain memory thresholds are reached. Therefore, the more regions that exist in a region server, the less memstore space is available per region. This may result in smaller flushes, which in turn may lead to smaller and more HFiles and more minor compactions, which is less performant. The default configuration will set the ideal flush size to 100 MB; if you take the size of your memstore and divide that by 100 MB, the result should be the maximum number of regions you can reasonably put on that region server.

Compaction time

A larger region takes longer to compact. The empirical upper limit on the size of a region is around 20 GB, but there are very successful HBase clusters with upward of 120 GB regions.

You can assign regions to an HBase table in one of the two following ways:

- Create the table with a single default region, which then autosplits as data increases.
- Create the table with a given number of regions and set the region size to a high enough value (e.g., 100 GB per region) to avoid autosplitting.

For preselecting, you should make sure that you have the correct split policy selected. You will most likely want ConstantSizeRegionSplitPolicy or DisabledRegionSplitPolicy.

For most cases, we recommend preselecting the region count (the second option) of the table to avoid the performance impact of seeing random region splitting and sub-optimal region split ranges.

However, in some cases, automatic splitting (the first option) may make more sense. One such use case is a forever-growing data set where only the most recent data is updated. If the row key for this table is composed of `{Salt}{SeqID}`, it is possible to control the distribution of the writes to a fixed set of regions. As the regions split, older regions will no longer need to be compacted (barring the periodic TTL-based compaction).

Using Columns

The concept of columns in an HBase table is very different than those in a traditional RDBMS. In HBase, unlike in an RDBMS, a record can have a million columns and the next record can have a million completely different columns. This isn’t recommended, but it’s definitely possible and it’s important to understand the difference.

To illustrate, let’s look into how HBase stores a record. HBase stores data in a format called HFile. Each column value gets its own row in HFile. This row has fields like row key, timestamp, column names, and values. The HFile format also provides a lot of other functionality, like versioning and sparse column storage, but we are eliminating that from the next example for increased clarity.

For example, if you have a table with two logical columns, foo and bar, your first logical choice is to create an HBase table with two columns named foo and bar. The benefits of doing so are:

- We can get one column at a time independently of the other column.
- We can modify each column independently of the other.
- Each column will age out with a TTL independently.

However, these benefits come with a cost. Each logical record in the HBase table will have two rows in the HBase HFile format. Here is the structure of such an HFile on disk:

```
|RowKey |TimeStamp  |Column  |Value
|101    |1395531114 |F       |A1
|101    |1395531114 |B       |B1
```

The alternative choice is to have both the values from foo and bar in the same HBase column. This would apply to all records of the table and bears the following characteristics:

- Both the columns would be retrieved at the same time. You may choose to disregard the value of the other column if you don’t need it.
- Both the column values would need to be updated together since they are stored as a single entity (column).
- Both the columns would age out together based on the last update.

Here is the structure of the HFile in such a case:

```
|RowKey |TimeStamp  |Column  |Value
|101    |1395531114 |X       |A1|B1
```

The amount of space consumed on disk plays a nontrivial role in your decision on how to structure your HBase schema, in particular the number of columns. It determines:

- How many records can fit in the block cache
- How much data can fit through the Write-Ahead-Log maintained by HBase
- How many records can fit into the memstore flush
- How long a compaction would take

Note

Notice the one-character column names in the previous examples. In HBase, the column and row key names, as you can see, take up space in the HFile. It is recommended to not waste that space as much as possible, so the use of single-character column names is fairly common.

Using Column Families

In addition to columns, HBase also includes the concept of *column families*. A column family is essentially a container for columns. A table can have one or more column families. The takeaway here is that each column family has its own set of HFiles and gets compacted independently of other column families in the same table.

For many use cases, there is no need for more than one column family per table. The main reason to use more than one is when the operations being done and/or the rate of change on a subset of the columns of a table is significantly different from the other columns.

For example, let’s consider an HBase table with two columns: column1 contains about 400 bytes per row and column2 contains about 20 bytes. Now let’s say that the value of column1 gets set once and never changes, but that of column2 changes very often. In addition, the access patterns call `get` requests on column2 a lot more than on column1.

In such a case, having two column families makes sense for the following reasons:

Lower compaction cost
If we had two separate column families, the column family with column2 will have memstore flushes very frequently, which will produce minor compactions. Because column2 is in its own column family, HBase will only need to compact 5% of the total records' worth of data, thereby making the compactions less impactful on performance.

Better use of block cache
As you saw earlier, when a record is retrieved from HBase, the records near the requested record (in the same HBase block) are pulled into the block cache. If both column1 and column2 are in the same column family, the data for both columns would be pulled into the block cache with each `get` request on column2. This results in suboptimal population of the block cache because the block cache would contain column1 data, which will be used very infrequently since `column 1` receives very few `get` requests. Having column1 and column2 in separate column families would result in the block cache being populated with values only from column2, thereby increasing the number of cache hits on subsequent `get` requests on column 2.

Time-to-Live

TTL is a built-in feature of HBase that ages out data based on its timestamp. This idea comes in handy in use cases where data needs to be held only for a certain duration of time. So, if on a major compaction the timestamp is older than the specified TTL in the past, the record in question doesn't get put in the HFile being generated by the major compaction; that is, the older records are removed as a part of the normal upkeep of the table.

If TTL is not used and an aging requirement is still needed, then a much more I/O-intensive operation would need to be done. For example, if you needed to remove all data older than seven years without using TTL, you would have to scan all seven years of data every day and then insert a delete record for every record that is older than seven years. Not only do you have to scan all seven years, but you also have to create new delete records, which could be multiple terabytes in size themselves. Then, finally, you still have to run a major compaction to finally remove those records from disk.

One last thing to mention about TTL is that it's based on the timestamp of an HFile record. As mentioned earlier, this timestamp defaults to the time the record was added to HBase.

Managing Metadata

Up until now, we have talked about data and the best way to structure and store it in Hadoop. However, just as important as the data is metadata about it. In this section, we will talk about the various forms of metadata available in the Hadoop ecosystem and how you can make the most out of it.

What Is Metadata?

Metadata, in general, refers to data about the data. In the Hadoop ecosystem, this can mean one of many things. To list a few, metadata can refer to:

Metadata about logical data sets
This includes information like the location of a data set (e.g., directory in HDFS or the HBase table name), the schema associated with the data set,[1] the partitioning and sorting properties of the data set, if any, and the format of the data set, if applicable (e.g., CSV, TSV, SequenceFile, etc.). Such metadata is usually stored in a separate metadata repository.

Metadata about files on HDFS
This includes information like permissions and ownership of such files and the location of various blocks of that file on data nodes. Such information is usually stored and managed by Hadoop NameNode.

Metadata about tables in HBase
This includes information like table names, associated namespace, associated attributes (e.g., `MAX_FILESIZE`, `READONLY`, etc.), and the names of column families. Such information is stored and managed by HBase itself.

Metadata about data ingest and transformations
This includes information like which user generated a given data set, where the data set came from, how long it took to generate it, and how many records there are or the size of the data loaded.

Metadata about data set statistics
This includes information like the number of rows in a data set, the number of unique values in each column, a histogram of the distribution of data, and maximum and minimum values. Such metadata is useful for various tools that can leverage it for optimizing their execution plans but also for data analysts, who can do quick analysis based on it.

In this section, we will be talking about the first point in the preceding list: metadata about logical data sets. From here on in this section, the word *metadata* will refer to metadata in that context.

Why Care About Metadata?

There are three important reasons to care about metadata:

- It allows you to interact with your data through the higher-level logical abstraction of a table rather than as a mere collection of files on HDFS or a table in HBase. This means that the users don't need to be concerned about where or how the data is stored.
- It allows you to supply information about your data (e.g., partitioning or sorting properties) that can then be leveraged by various tools (written by you or someone else) while populating and querying data.
- It allows data management tools to "hook" into this metadata and allow you to perform data discovery (discover what data is available and how you can use it) and lineage (trace back where a given data

set came from or originated) analysis.

Where to Store Metadata?

The first project in the Hadoop ecosystem that started storing, managing, and leveraging metadata was Apache Hive. Hive stores this metadata in a relational database called the *Hive metastore*. Note that Hive also includes a service called the *Hive metastore service* that interfaces with the Hive metastore database. In order to avoid confusion between the database and the Hive service that accesses this database, we will call the former *Hive metastore database* and the latter *Hive metastore service*. When we refer to something as *Hive metastore* in this book, we are referring to the collective logical system comprising both the service and the database.

Over time, more projects wanted to use the same metadata that was in the Hive metastore. To enable the usage of Hive metastore outside of Hive, a separate project called HCatalog was started. Today, HCatalog is a part of Hive and serves the very important purpose of allowing other tools (like Pig and MapReduce) to integrate with the Hive metastore. It also opens up access to the Hive metastore to a broader ecosystem by exposing a REST API to the Hive metastore via the WebHCat server.

You can think of HCatalog as an accessibility veneer around the Hive metastore. See Figure 1-4 for an illustration.

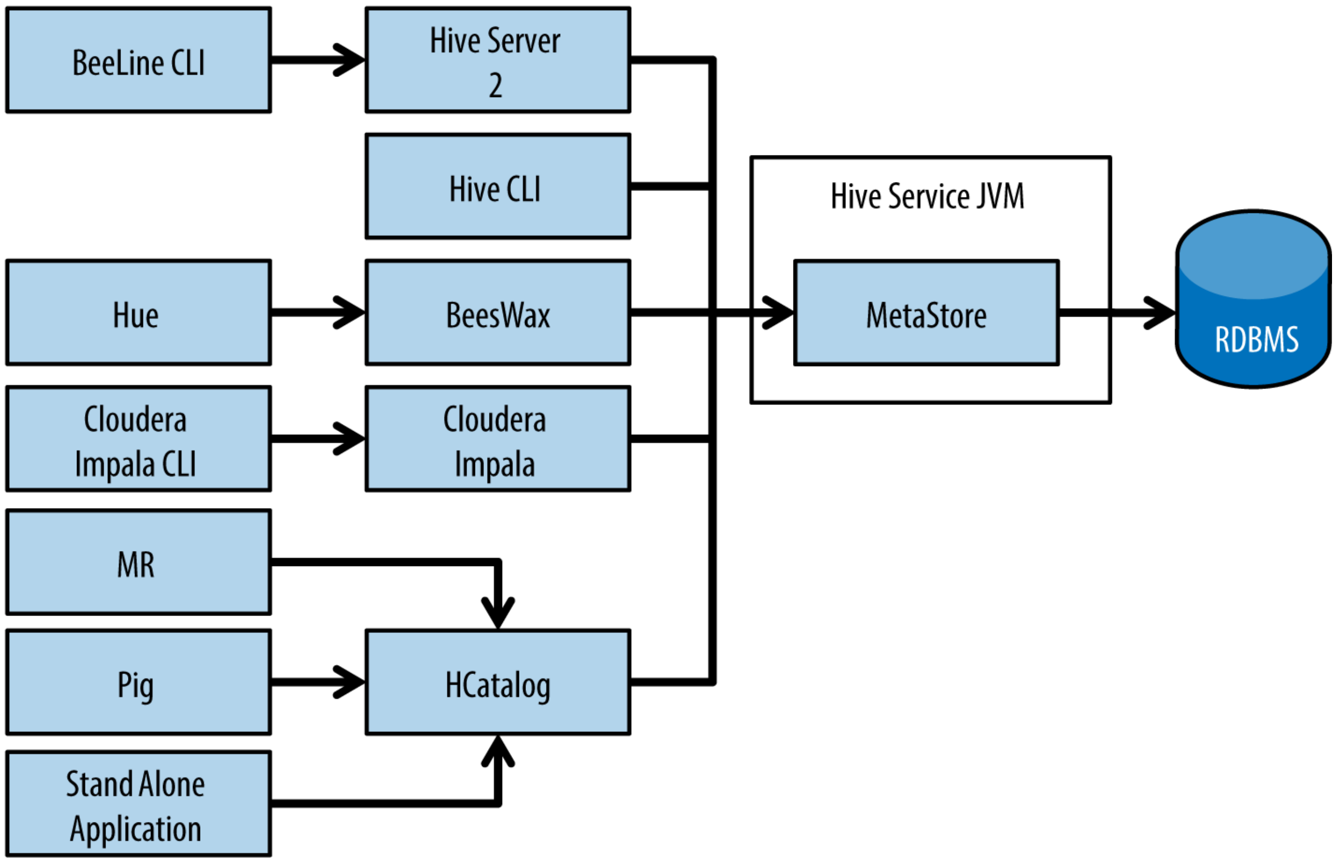

Figure 1-4. HCatalog acts as an accessibility veneer around the Hive metastore

Now MapReduce, Pig, and standalone applications could very well talk directly to the Hive metastore through its APIs, but HCatalog allows easy access through its WebHCat Rest APIs and it allows the cluster administrators to lock down access to the Hive metastore to address security concerns.

Note that you don't have to use Hive in order to use HCatalog and the Hive metastore. HCatalog just relies on some infrastructure pieces from Hive to store and access metadata about your data.

Hive metastore can be deployed in three modes: embedded metastore, local metastore, and remote metastore. Although we won't be able to do justice to the details of each of these modes here, we recommend that you use the Hive metastore in remote mode, which is a requirement for using HCatalog on top of the Hive metastore. A few of the popular databases that are supported as Hive metastore databases are MySQL, PostgreSQL, Derby, and Oracle. MySQL is by far the most commonly used in industry. You can, of course, run a new database instance, create a user for Hive and assign it the right permissions, and use this database as your metastore. If you already have a relational database instance in the organization that you can utilize, you have the option of using it as the Hive metastore database. You can simply create a new schema for Hive along with a new user and associated privileges on the existing instance.

Whether you should reuse the existing database instance instead of creating a new one depends on usage patterns of the database instance, existing load on the database instance, and other applications using the database instance. On one hand, it's good from an operational perspective to not have a new database instance for every new application (in this case the Hive metastore service, which handles metadata in the Hadoop ecosystem) but on the other hand, it makes sense to not have your Hadoop infrastructure cross-depend on a rather uncoupled database instance. Other considerations may matter as well. For example, if you already have an existing highly available database cluster in your organization, and want to use it to have high availability for your Hive metastore database, it may make sense to use the existing HA database cluster for the Hive metastore database.

Examples of Managing Metadata

If you are using Hive or Impala, you don't have to do anything special to create or retrieve metadata. These systems integrate directly with the Hive metastore, which means a simple `CREATE TABLE` command creates metadata, `ALTER TABLE` alters metadata, and your queries on the tables retrieve the stored metadata.

If you are using Pig, you can rely on HCatalog's integration with it to store and retrieve metadata. If you are using a programming interface to query data in Hadoop (e.g., MapReduce, Spark, or Cascading), you can use HCatalog's Java API to store and retrieve metadata. HCatalog also has a command-line interface (CLI) and a REST API that can be used to store, update, and retrieve metadata.

Limitations of the Hive Metastore and HCatalog

There are some downsides to using the Hive metastore and HCatalog, some of which are outlined here:

Problems with high availability
To provide HA for the Hive metastore, you have to provide HA for the metastore database as well as the metastore service. The metastore database is a database at the end of the day and the HA story for databases is a solved problem. You can use one of many HA database cluster solutions to bring HA to the Hive metastore database. As far as the metastore services goes, there is support concurrently to run multiple metastores on more than one node in the cluster. However, at the time of this writing, that can lead to concurrency issues related to data definition language (DDL) statements and other queries being run at the same time.[2] The Hive community is working toward fixing these issues.

Fixed schema for metadata
While Hadoop provides a lot of flexibility on the type of data that can be stored, especially because of the Schema-on-Read concept, the Hive metastore, since it's backed by a relational backend, provides a fixed schema for the metadata itself. Now, this is not as bad as it sounds since the schema is fairly diverse, allowing you to store information all the way from columns and their types to the sorting properties of the data. If you have a rare use case where you want to store and retrieve metainformation about your data that currently can't be stored in the metastore schema, you may have to choose a different system for metadata. Moreover, the metastore is intended to provide a tabular abstraction for the data sets. If it doesn't make sense to represent your data set as a table, say if you have image or video data, you may still have a need for storing and retrieving metadata, but the Hive metastore may not be the right tool for it.

Another moving part
Although not necessarily a limitation, you should keep in mind that the Hive metastore service is yet one more moving part in your infrastructure. Consequently you have to worry about keeping the

metastore service up and securing it like you secure the rest of your Hadoop infrastructure.

Other Ways of Storing Metadata

Using Hive metastore and HCatalog is the most common method of storing and managing table metadata in the Hadoop ecosystem. However, there are some cases (likely one of the limitations listed in the previous section) in which users prefer to store metadata outside of the Hive metastore and HCatalog. In this section, we describe a few of those methods.

Embedding metadata in file paths and names

As you may have already noticed in the section “HDFS Schema Design”, we recommend embedding some metadata in data set locations for organization and consistency. For example, in case of a partitioned data set, the directory structure would look like:

<data set name>/<partition_column_name=partition_column_value>/{files}

Such a structure already contains the name of the data set, the name of the partition column, and the various values of the partitioning column the data set is partitioned on. Tools and applications can then leverage this metadata in filenames and locations when processing. For example, listing all partitions for a data set named medication_orders would be a simple `ls` operation on the */data/medication_orders* directory of HDFS.

Storing the metadata in HDFS

You may decide to store the metadata on HDFS itself. One option to store such metadata is to create a hidden directory, say *.metadata*, inside the directory containing the data in HDFS. You may decide to store the schema of the data in an Avro schema file. This is especially useful if you feel constrained by what metadata you can store in the Hive metastore via HCatalog. The Hive metastore, and therefore HCatalog, has a fixed schema dictating what metadata you can store in it. If the metadata you’d like to store doesn’t fit in that realm, managing your own metadata may be a reasonable option. For example, this is what your directory structure in HDFS would look like:

```
/data/event_log
/data/event_log/file1.avro
/data/event_log/.metadata
```

The important thing to note here is that if you plan to go this route, you will have to create, maintain, and manage your own metadata. You may, however, choose to use something like Kite SDK[3] to store metadata. Moreover, Kite supports multiple metadata providers, which means that although you can use it to store metadata in HDFS as just described, you can also use it to store data in HCatalog (and hence the Hive metastore) via its integration with HCatalog. You can also easily transform metadata from one source (say HCatalog) to another (say the *.metadata* directory in HDFS).

Conclusion

Although data modeling in any system is a challenging task, it’s especially challenging in the Hadoop ecosystem due to the vast expanse of options available. The larger number of options exists partly due to Hadoop’s increased flexibility. Even though Hadoop is still Schema-on-Read, choosing the right model for storing your data provides a lot of benefits like reducing storage footprint, improving processing times, making authorization and permission management easier, and allowing for easier metadata management.

As we discussed in this chapter, you have a choice of storage managers, HDFS and HBase being the most common ones. If you’re using HDFS, a number of file formats exist for storing data, with Avro being a very popular choice for row-oriented formats, and ORC and Parquet being a popular choice for column-oriented ones. A good choice of compression codecs to use for HDFS data exists as well, Snappy being one of the more popular ones. If you’re storing data in HBase, the choice of row key is arguably the most important design decision from a modeling perspective.

The next important modeling choice relates to managing metadata. Although *metadata* can refer to many things, in this chapter we focused on schema-related metadata and types of fields associated with the data. The Hive metastore has become the de-facto standard for storing and managing metadata, but there are cases when users manage their own.

Choosing the right model for your data is one of the most important decisions you will make in your application, and we hope that you spend the appropriate amount of time and effort to get it right the first time.

[1] Note that Hadoop is Schema-on-Read. Associating a schema doesn't take that away, it just implies that it is one of the ways to interpret the data in the data set. You can associate more than one schema with the same data.

[2] See HIVE-4759, for example.

[3] Kite is a set of libraries, tools, examples, and documentation focused on making it easier to build systems on top of the Hadoop ecosystem. Kite allows users to create and manage metadata on HDFS as described in this section.