O'REILLY®

Hadoop Application Architectures

DESIGNING REAL-WORLD BIG DATA APPLICATIONS

Mark Grover, Ted Malaska,
Jonathan Seidman & Gwen Shapira

BEDROCK

Award-winning platform for integrated data lake managment

DON'T GO IN THE LAKE WITHOUT US

LEARN MORE

www.zaloni.com

ZALONI

This Excerpt contains Chapters 1 and 2 of the book *Hadoop Application Architectures*. The complete book is available at oreilly.com and through other retailers.

Hadoop Application Architectures

Mark Grover, Ted Malaska,
Jonathan Seidman & Gwen Shapira

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Hadoop Application Architectures
by Mark Grover, Ted Malaska, Jonathan Seidman, and Gwen Shapira

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editors: Ann Spencer and Brian Anderson
Production Editor: Nicole Shelby
Copyeditor: Rachel Monaghan
Proofreader: Elise Morrison
Indexer: Ellen Troutman
Interior Designer: David Futato
Cover Designer: Ellie Volckhausen
Illustrator: Rebecca Demarest

July 2015: First Edition

Revision History for the First Edition
2015-06-26: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781491900086* for release details.

978-1-491-90008-6

[LSI]

Table of Contents

CHAPTER 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 Glus-

terFS 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" on page 14. 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" on page 12, 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.

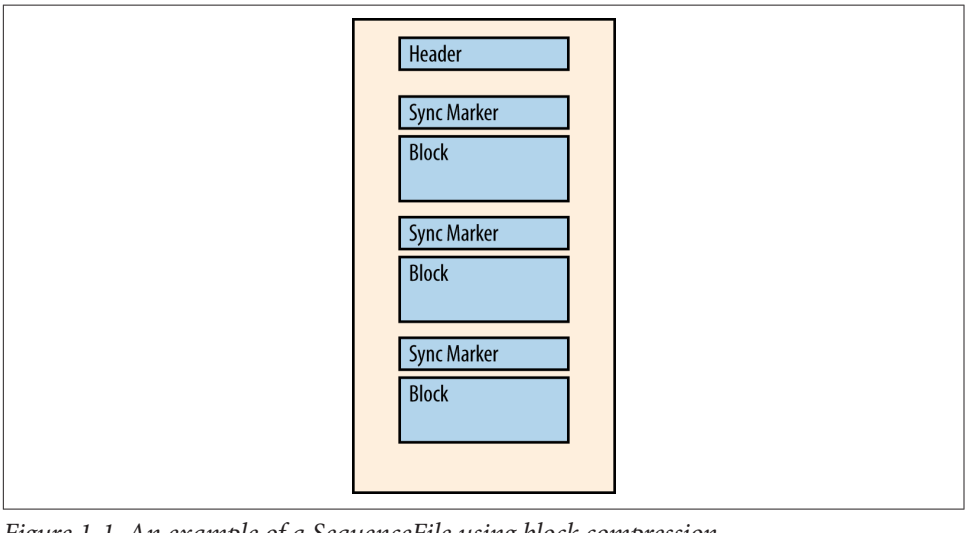

Figure 1-1. An example of a SequenceFile using block compression

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.

Comparing Failure Behavior for Different File Formats

An important aspect of the various file formats is failure handling; some formats handle corruption better than others:

- Columnar formats, while often efficient, do not work well in the event of failure, since this can lead to incomplete rows.
- Sequence files will be readable to the first failed row, but will not be recoverable after that row.

- Avro provides the best failure handling; in the event of a bad record, the read will continue at the next sync point, so failures only affect a portion of a file.

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.

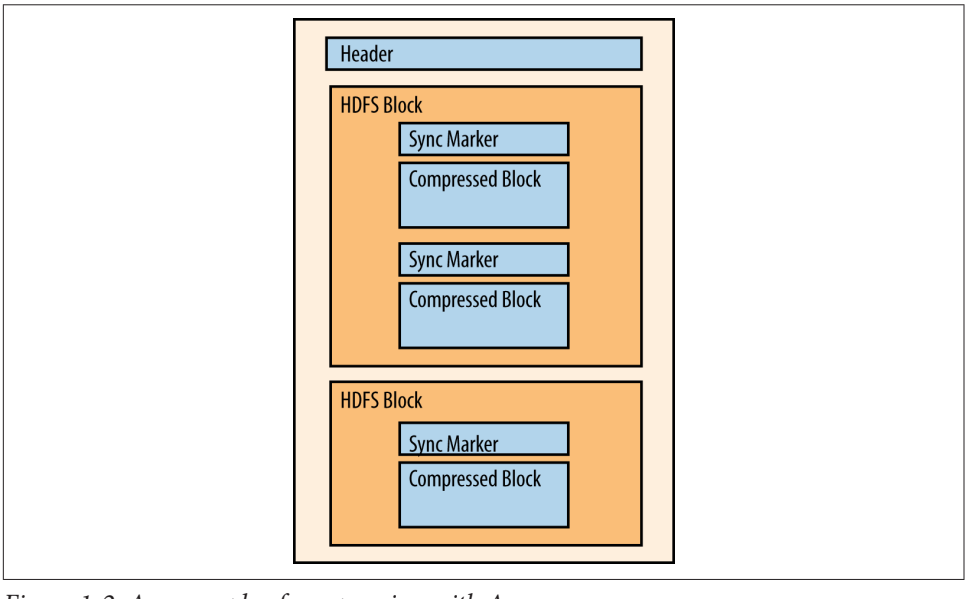

Figure 1-2. An example of compression with Avro

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 docu-

ments 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 deci-

sions, 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" on page 31, 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 ???.

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 for-

mat 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" on page 6, 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 antipattern 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 *Materi-*

alized 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 ???. 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 ???.

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.

Table 1-1. Example HBase table

Row key	Timestamp	Column	Value
RowKeyA	Timestamp	ColumnA	ValueA
RowKeyB	Timestamp	ColumnB	ValueB
RowKeyC	Timestamp	ColumnC	ValueC

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.

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.

Table 1-2. Persons table

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

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.

Table 1-3. Students table

Student ID	Student name
11001	Bob
11002	David
11003	Charles

Table 1-4. Courses table

Student ID	Courses
11001	Chemistry, Physics
11002	Math
11003	History

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" on page 17, 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.

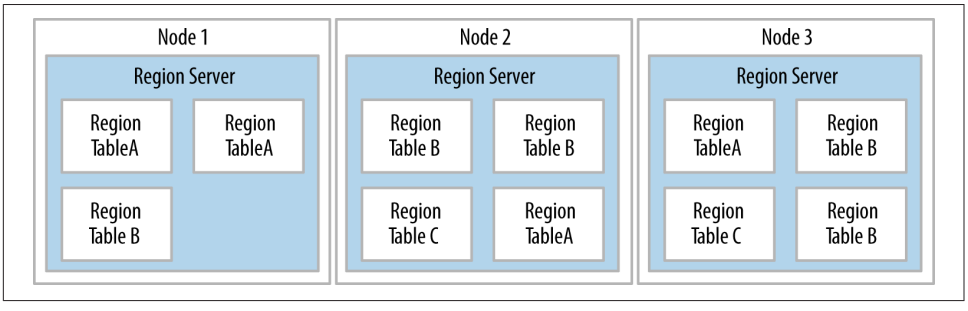

Figure 1-3. Topology of region servers, regions, and tables

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 suboptimal 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

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.

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.

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.

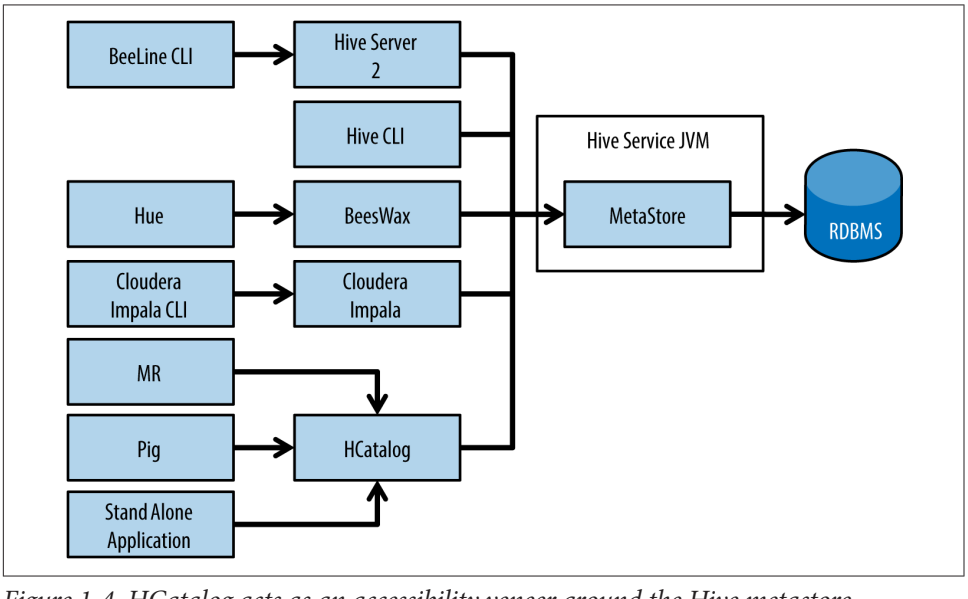

Figure 1-4. HCatalog acts as an accessibility veneer around the Hive metastore

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 meta-

store 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" on page 14, 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}
```

2 See HIVE-4759, for example.

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.

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.

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.

CHAPTER 2

Data Movement

Now that we've discussed considerations around storing and modeling data in Hadoop, we'll move to the equally important subject of moving data between external systems and Hadoop. This includes ingesting data into Hadoop from systems such as relational databases or logs, and extracting data from Hadoop for ingestion into external systems. We'll spend a good part of this chapter talking about considerations and best practices around data ingestion into Hadoop, and then dive more deeply into specific tools for data ingestion, such as Flume and Sqoop. We'll then discuss considerations and recommendations for extracting data from Hadoop.

Data Ingestion Considerations

Just as important as decisions around how to store data in Hadoop, which we discussed in Chapter 1, are the architectural decisions on getting that data into your Hadoop cluster. Although Hadoop provides a filesystem client that makes it easy to copy files in and out of Hadoop, most applications implemented on Hadoop involve ingestion of disparate data types from multiple sources and with differing requirements for frequency of ingestion. Common data sources for Hadoop include:

- Traditional data management systems such as relational databases and mainframes
- Logs, machine-generated data, and other forms of event data
- Files being imported from existing enterprise data storage systems

There are a number of factors to take into consideration when you're importing data into Hadoop from these different systems. As organizations move more and more data into Hadoop, these decisions will become even more critical. Here are the considerations we'll cover in this chapter:

Timeliness of data ingestion and accessibility
: What are the requirements around how often data needs to be ingested? How soon does data need to be available to downstream processing?

Incremental updates
: How will new data be added? Does it need to be appended to existing data? Or overwrite existing data?

Data access and processing
: Will the data be used in processing? If so, will it be used in batch processing jobs? Or is random access to the data required?

Source system and data structure
: Where is the data coming from? A relational database? Logs? Is it structured, semistructured, or unstructured data?

Partitioning and splitting of data
: How should data be partitioned after ingest? Does the data need to be ingested into multiple target systems (e.g., HDFS and HBase)?

Storage format
: What format will the data be stored in?

Data transformation
: Does the data need to be transformed in flight?

We'll start by talking about the first consideration: the timeliness requirements for the data being ingested.

Timeliness of Data Ingestion

When we talk about timeliness of data ingestion, we're referring to the time lag from when data is available for ingestion to when it's accessible to tools in the Hadoop ecosystem. The time classification of an ingestion architecture will have a large impact on the storage medium and on the method of ingestion. For purposes of this discussion we're not concerned with streaming processing or analytics, which we'll discuss separately in ???, but only with when the data is stored and available for processing in Hadoop.

In general, it's recommended to use one of the following classifications before designing the ingestion architecture for an application:

Macro batch
: This is normally anything over 15 minutes to hours, or even a daily job.

Microbatch
: This is normally something that is fired off every 2 minutes or so, but no more than 15 minutes in total.

Near-Real-Time Decision Support
: This is considered to be "immediately actionable" by the recipient of the information, with data delivered in less than 2 minutes but greater than 2 seconds.

Near-Real-Time Event Processing
: This is considered under 2 seconds, and can be as fast as a 100-millisecond range.

Real Time
: This term tends to be very overused, and has many definitions. For purposes of this discussion, it will be anything under 100 milliseconds.

It's important to note that as the implementation moves toward real time, the complexity and cost of the implementation increases substantially. Starting off at batch (e.g., using simple file transfers) is always a good idea; start simple before moving to more complex ingestion methods.

With more lenient timeliness requirements, HDFS will likely be the preferred source as the primary storage location, and a simple file transfer or Sqoop jobs will often be suitable tools to ingest data. We'll talk more about these options later, but these ingestion methods are well suited for batch ingestion because of their simple implementations and the validation checks that they provide out of the box. For example, the `hadoop fs -put` command will copy a file over and do a full checksum to confirm that the data is copied over correctly.

One consideration when using the `hdfs fs -put` command or Sqoop is that the data will land on HDFS in a format that might not be optimal for long-term storage and processing, so using these tools might require an additional batch process to get the data into the desired format. An example of where such an additional batch process would be required is loading Gzip files into HDFS. Although Gzip files can easily be stored in HDFS and processed with MapReduce or other processing frameworks on Hadoop, as we discussed in the previous chapter Gzip files are not splittable. This will greatly impact the efficiency of processing these files, particularly as the files get larger. In this case a good solution is to store the files in Hadoop using a container format that supports splittable compression, such as SequenceFiles or Avro.

As your requirements move from batch processing to more frequent updates, you should consider tools like Flume or Kafka. Sqoop and file transfers are not going to be a good selection, as the delivery requirements get shorter than two minutes. Further, as the requirements become shorter than two minutes, the storage layer may need to change to HBase or Solr for more granular insertions and read operations. As the requirements move toward the real-time category, we need to think about memory first and permanent storage second. All of the parallelism in the world isn't going

to help for response requirements under 500 milliseconds as long as hard drives remain in the process. At this point, we start to enter the realm of stream processing, with tools like Storm or Spark Streaming. It should be emphasized that these tools are actually focused on data processing, as opposed to data ingest like Flume or Sqoop. Again, we'll talk more about near-real-time streaming with tools like Storm and Spark Streaming in ???. Flume and Kafka will be discussed further in this chapter.

Incremental Updates

This decision point focuses on whether the new data is data that will append an existing data set or modify it. If the requirements are for append only—for example, when you're logging user activity on a website—then HDFS will work well for the vast majority of implementations. HDFS has high read and write rates because of its ability to parallelize I/O to multiple drives. The downside to HDFS is the inability to do appends or random writes to files after they're created. Within the requirements of an "append only" use case, this is not an issue. In this case it's possible to "append" data by adding it as new files or partitions, which will enable MapReduce and other processing jobs to access the new data along with the old data.

Note that Chapter 1 provides a full discussion of partitioning and organizing data stored in HDFS.

A very important thing to note when you're appending to directories with additional files is that HDFS is optimized for large files. If the requirements call for a two-minute append process that ends up producing lots of small files, then a periodic process to combine smaller files will be required to get the benefits from larger files. There are a number of reasons to prefer large files, one of the major reasons being how data is read from disk. Using a long consecutive scan to read a single file is faster than performing many seeks to read the same information from multiple files.

We briefly discuss methods to manage small files later in this chapter, but a full discussion is beyond the scope of this book. See *Hadoop: The Definitive Guide* or *Hadoop in Practice* for detailed discussions on techniques to manage small files.

If the requirements also include modifying existing data, then additional work is required to land the data properly in HDFS. HDFS is read only—you can't update records in place as you would with a relational database. In this case we first write a "delta" file that includes the changes that need to be made to the existing file. A *compaction job* is required to handle the modifications. In a compaction job, the data is

sorted by a primary key. If the row is found twice, then the data from the newer delta file is kept and the data from the older file is not. The results of the compaction process are written to disk, and when the process is complete the resulting compaction data will replace the older, uncompacted data. Note that this compaction job is a batch process—for example, a MapReduce job that's executed as part of a job flow. It may take several minutes depending on the data size, so it will only work for multi-minute timeliness intervals.

Table 2-1 is an example of how a compaction job works. Note in this example that the first column will be considered the primary key for the data.

Table 2-1. Compaction

Original data	New data	Resulting data
A,Blue,5	B,Yellow,3	A,Blue,5
B,Red,6	D,Gray,0	B,Yellow,3
C,Green,2		C,Green,2
		D,Gray,0

There are many ways to implement compaction with very different performance results. We'll provide some examples of implementing this in ???.

Another option to consider beyond HDFS and file compactions is to use HBase instead. HBase handles compactions in the background and has the architecture to support deltas that take effect upon the completion of an HBase `put`, which typically occurs in milliseconds.

Note that HBase requires additional administration and application development knowledge. Also, HBase has much different access patterns than HDFS that should be considered—for example, scan rates. HBase scan rates are about 8–10 times slower than HDFS. Another difference is random access; HBase can access a single record in milliseconds, whereas HDFS doesn't support random access other than file seeking, which is expensive and often complex.

Access Patterns

This decision point requires deep understanding of the underlying requirements for information delivery. How is the data going to be used once it is in Hadoop? For example: if the requirements call for random row access, HDFS may not be the best fit, and HBase might be a better choice. Conversely, if scans and data transformations are required, HBase may not be a good selection. Even though there can be many variables to consider, we suggest this basic guiding principle: for cases where simplic-

ity, best compression, and highest scan rate are called for, HDFS is the default selection. In addition, in newer versions of HDFS (Hadoop 2.3.0 and later) caching data into memory is supported. This allows tools to read data directly from memory for files loaded into the cache. This moves Hadoop towards a massively parallel in-memory database accessible to tools in the Hadoop ecosystem. When random access is of primary importance, HBase should be the default, and for search processing you should consider Solr.

For a more detailed look, Table 2-2 includes common access patterns for these storage options.

Table 2-2. Access patterns for Hadoop storage options

Tool	Use cases	Storage device
MapReduce	Large batch processes	HDFS is preferred. HBase can be used but is less preferred.
Hive	Batch processing with SQL-like language	HDFS is preferred. HBase can be used but is less preferred.
Pig	Batch processing with a data flow language	HDFS is preferred. HBase can be used but is less preferred.
Spark	Fast interactive processing	HDFS is preferred. HBase can be used but is less preferred.
Giraph	Batch graph processing	HDFS is preferred. HBase can be used but is less preferred.
Impala	MPP style SQL	HDFS is preferred for most cases, but HBase will make sense for some use cases even through the scan rates are slower, namely near-real-time access of newly updated data and very fast access to records by primary key.
HBase API	Atomic `puts`, `gets`, and `deletes` on record-level data	HBase

Original Source System and Data Structure

When ingesting data from a filesystem, you should consider the following items:

Read speed of the devices on source systems

Disk I/O is often a major bottleneck in any processing pipeline. It may not be obvious, but optimizing an ingestion pipeline often requires looking at the system from which the data is being retrieved. Generally, with Hadoop we'll see read speeds of anywhere from 20 MBps to 100 MBps, and there are limitations on the motherboard or controller for reading from all the disks on the system. To maximize read speeds, make sure to take advantage of as many disks as possible on the source system. On some network attached storage (NAS) systems, additional mount points can increase throughput. Also note that a single reading thread may not be able to maximize the read speed of a drive or device. Based on our experience, on a typical drive three threads is normally required to maximize throughput, although this number will vary.

Original file type

Data can come in any format: delimited, XML, JSON, Avro, fixed length, variable length, copybooks, and many more. Hadoop can accept any file format, but not all formats are optimal for particular use cases. For example, consider a CSV file. CSV is a very common format, and a simple CSV file can generally be easily imported into a Hive table for immediate access and processing. However, many tasks converting the underlying storage of this CSV file may provide more optimal processing; for example, many analytical workloads using Parquet as a storage format may provide much more efficient processing while also reducing the storage size of the file.

Another consideration is that not all file formats can work with all tools in the Hadoop ecosystem. An example of this would be variable-length files. Variable-length files are similar to flat files in that columns are defined with a fixed length. The difference between a fixed-length file and a variable-length file is that in the variable-length file one of the leftmost columns can decide the rules to read the rest of the file. An example of this is if the first two columns are an 8-byte ID followed by a 3-byte type. The ID is just a global identifier and reads very much like a fixed-length file. The type column, however, will set the rules for the rest of the record. If the value of the type column is *car*, the record might contain columns like max speed, mileage, and color; however, if the value is *pet*, then there might be columns in the record such as size and breed. These different columns will have different lengths, hence the name "variable length." With this understanding we can see that a variable-length file may not be a good fit for Hive, but can still be effectively processed by one of the processing frameworks available for Hadoop, such as a Java MapReduce job, Crunch, Pig, or Spark.

Compression

There is a pro and a con to compressing data on the original filesystem. The pro is that transferring a compressed file over the network requires less I/O and network

bandwidth. The con is that most compression codecs applied outside of Hadoop are not splittable (e.g., Gzip), although most of these compression codecs are splittable in Hadoop if you use them with a splittable container format like SequenceFiles, Parquet files, or Avro files as we discussed in Chapter 1. Normally the way to do this is to copy the compressed file to Hadoop and convert the files in a post-processing step. It's also possible to do the conversion as the data is streamed to Hadoop, but normally it makes more sense to use the distributed processing power of the Hadoop cluster to convert files, rather than just the edge nodes that are normally involved in moving data to the cluster.

We discussed compression considerations with HDFS in Chapter 1, and we'll look at concrete examples of ingestion and post-processing in the case studies later in the book.

Relational database management systems

It is common for Hadoop applications to integrate data from RDBMS vendors like Oracle, Netezza, Greenplum, Vertica, Teradata, Microsoft, and others. The tool of choice here is almost always Apache Sqoop. Sqoop is a very rich tool with lots of options, but at the same time it is simple and easy to learn compared to many other Hadoop ecosystem projects. These options will control which data is retrieved from the RDBMS, how the data is retrieved, which connector to use, how many map tasks to use, split patterns, and final file formats.

Sqoop is a batch process, so if the timeliness of the data load into the cluster needs to be faster than batch, you'll likely have to find an alternate method. One alternative for certain use cases is to split data on ingestion, with one pipeline landing data in the RDBMS, and one landing data in HDFS. This can be enabled with tools like Flume or Kafka, but this is a complex implementation that requires code at the application layer.

Note that with the Sqoop architecture the DataNodes, not just the edge nodes, are connecting to the RDBMS. In some network configurations this is not possible, and Sqoop will not be an option. Examples of network issues are bottlenecks between devices and firewalls. In these cases, the best alternative to Sqoop is an RDBMS file dump and import into HDFS. Most relational databases support creating a delimited file dump, which can then be ingested into Hadoop via a simple file transfer. Figure 2-1 shows the difference between a file export with Sqoop versus RMDBS.

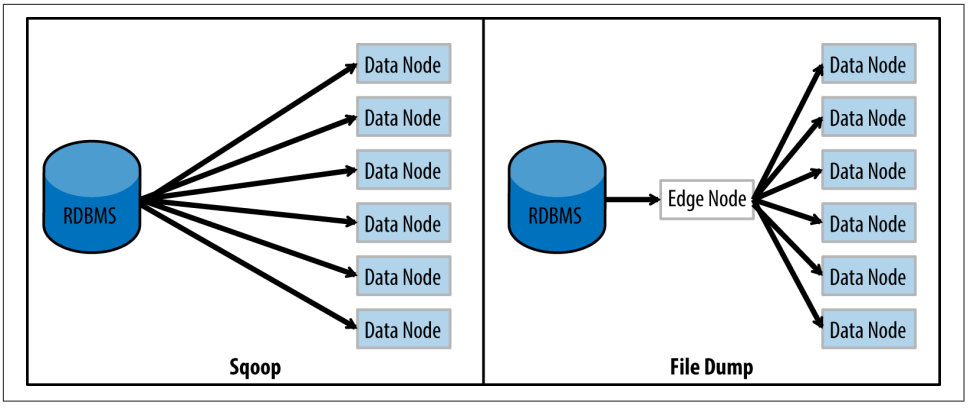

Figure 2-1. Sqoop versus RDBMS file export

We'll discuss more considerations and best practices for using Sqoop later in this chapter.

Streaming data

Examples of streaming input data include Twitter feeds, a Java Message Service (JMS) queue, or events firing from a web application server. In this situation a tool like Flume or Kafka is highly recommended. Both systems offer levels of guarantees and provide similar functionality, with some important differences. Later in this chapter we will drill down deeper into both Flume and Kafka.

Logfiles

Logfiles get their own section because they're a cross between filesystem and streaming input. An anti-pattern is to read the logfiles from disk as they are written because this is almost impossible to implement without losing data. The correct way of ingesting logfiles is to stream the logs directly to a tool like Flume or Kafka, which will write directly to Hadoop instead.

Transformations

In general, *transformation* refers to making modifications on incoming data, distributing the data into partitions or buckets, or sending the data to more than one store or location. Here are some simple examples of each option:

Transformation
: XML or JSON is converted to delimited data.

Partitioning
: Incoming data is stock trade data and partitioning by ticker is required.

Splitting
: The data needs to land in HDFS and HBase for different access patterns.

Making a decision on how to transform data will depend on the timeliness of the requirements. If the timeliness of the data is batch, then the preferred solution will most likely be a file transfer followed by a batch transformation. Note that these transformations can be accomplished with tools like Hive, Pig, or Java MapReduce, or newer tools like Spark. The use of post-processing for this batch transformation step is preferred because of the checkpoints for failure provided by this processing, including the checksum on the file transfer and the all-or-nothing success/failure behavior of MapReduce. An important thing to note: MapReduce can be configured to transfer, partition, and split data beyond all-or-nothing processing. You would do so by configuring MapReduce with two output directories: one for records that were processed successfully and one for failures. In some cases, the timeliness of the data requirements will not allow for the simplicity of a file transfer followed by MapReduce processing. Sometimes the work has to be done as the data is in flight to the cluster with a stream ingestion tool like Flume. When using Flume, this can be done using interceptors and selectors. We'll cover Flume in more detail later in this chapter, but we'll briefly cover the roles of interceptors and selectors here.

Interceptors

A Flume interceptor is a Java class that allows for in-flight enrichment of event data. Since it's implemented in Java it provides great flexibility in the functionality that can be implemented. The interceptor has the capability to transform an event or a batch of events, including functionality to add or remove events from a batch. You must take special care when implementing it because it is part of a streaming framework, so the implementation should not cause the pipe to be slowed by things like calls to external services or garbage collection issues. Also remember that when transforming data with interceptors, you have a limit on processing power because normally Flume is only installed on a subset of nodes in a cluster.

Selectors

In Flume a selector object acts as a "fork in the road." It will decide which of the roads (if any) an event will go down. We'll provide an example of this in the following discussion of Flume patterns.

Network Bottlenecks

When you're ingesting into Hadoop the bottleneck is almost always the source system or the network between the source system and Hadoop. If the network is the bottleneck, it will be important to either add more network bandwidth or compress the data over the wire. You can do this by compressing the files before sending them over

the wire or using Flume to compress the data between agents on different sides of the network bottleneck.

The easiest way to know if the network is the bottleneck is to look at your throughput numbers and at the network configuration: if you are using 1 Gb ethernet, the expected throughput is around 100 MBps. If this is the throughput you are seeing with Flume, than indeed you maximized the network capacity and need to consider compressing the data. For a more accurate diagnostic, you can use a network monitoring tool to determine your network utilization.

Network Security

Sometimes you need to ingest data from sources that you can access only by going outside the company's firewalls. Depending on the data, it may be important to encrypt the data while it goes over the wire. You can do so by simply encrypting the files before sending them, for example using OpenSSL. Alternatively, Flume includes support for encrypting data sent between Flume agents. Note that Kafka does not currently support encryption of data within a Kafka data flow, so you'll have to do additional work to encrypt and decrypt the data outside of Kafka.

Push or Pull

All the tools discussed in this chapter can be classified as either pushing or pulling tools. The important thing to note is the actor in the architecture because in the end that actor will have additional requirements to consider, such as:

- Keeping track of what has been sent
- Handling retries or failover options in case of failure
- Being respectful of the source system that data is being ingested from
- Access and security

We'll cover these requirements in more detail in this chapter, but first we'll discuss two common Hadoop tools—Sqoop and Flume—to help clarify the distinction between push and pull in the context of data ingestion and extraction with Hadoop.

Sqoop

Sqoop is a pull solution that is used to extract data from an external storage system such as a relational database and move that data into Hadoop, or extract data from Hadoop and move it into an external system. It must be provided various parameters about the source and target systems, such as connection information for the source database, one or more tables to extract, and so on. Given these parameters, Sqoop will run a set of jobs to move the requested data.

In the example where Sqoop is extracting (pulling) data from a relational database, we have considerations such as ensuring that we extract data from the source database at a defined rate, because Hadoop can easily consume too many resources on the source system. We also need to ensure that Sqoop jobs are scheduled to not interfere with the source system's peak load time. We'll provide further details and considerations for Sqoop later in this chapter and provide specific examples in the case studies later in the book.

Flume

Note that Flume can be bucketed in both descriptions depending on the source used. In the case of the commonly used Log4J appender, Flume is pushing events through a pipeline. There are also several Flume sources, such as the spooling directory source or the JMS source, where events are being pulled. Here again, we'll get into more detail on Flume in this section and in our case studies later in the book.

Failure Handling

Failure handling is a major consideration when you're designing an ingestion pipeline; how to recover in case of failure is a critical question. With large distributed systems, failure is not a question of *if*, but of *when*. We can put in many layers of protection when designing an ingestion flow (sometimes at great cost to performance), but there is no silver bullet that can prevent all possible failures. Failure scenarios need to be documented, including failure delay expectations and how data loss will be handled.

The simplest example of a failure scenario is with file transfers—for example, performing a `hadoop fs -put <filename>` command. When the `put` command has finished, the command will have validated that the file is in HDFS, replicated three times, and passed a checksum check. But what if the `put` command fails? The normal way of handling the file transfer failure is to have multiple local filesystem directories that represent different bucketing in the life cycle of the file transfer process. Let's look at an example using this approach.

In this example, there are four local filesystem directories: *ToLoad*, *InProgress*, *Failure*, and *Successful*. The workflow in this case is as simple as the following:

1. Move the file into *ToLoad*.
2. When the `put` command is ready to be called, move the file into *InProgress*.
3. Call the `put` command.
4. If the `put` command fails, move the file into the *Failures* directory.
5. If the `put` was successful, move the file into the *Successful* directory.

Note that a failure in the middle of a file `put` will require a complete resend. Consider this carefully when doing a `hadoop fs -put` with very large files. If a failure on the network happens five hours into a file transfer, the entire process has to start again.

File transfers are pretty simple, so let's take an example with streaming ingestion and Flume. In Flume, there are many areas for failure, so to keep things simple, let's just focus on the following three:

- A "pull" source such as the spooling directory source could fail to load events to the channel because the channel is full. In this case, the source must pause before retrying the channel, as well as retain the events.
- An event receiving source could fail to load the event to the channel because the channel is full. In this case, the source will tell the Flume client it was unsuccessful in accepting the last batch of events and then the Flume client is free to resend to this source or another source.
- The sink could fail to load the event into the final destination. A sink will take a number of events from the channel and then try to commit them to the final destination, but if that commit fails then the batch of events needs to be returned to the channel.

The big difference between Flume and file transfers is that with Flume, in the case of a failure there is a chance of duplicate records getting created in Hadoop. This is because in the case of failure the event is always returned to the last step, so if the batch was half successful we will get duplicates. There are methods that try to address this issue with Flume and other streaming solutions like Kafka, but there is a heavy performance cost in trying to remove duplicates at the time of streaming. We'll see examples of processing to handle deduplication later in the book.

Level of Complexity

The last design factor that needs to be considered is complexity for the user. A simple example of this is if users need to move data into Hadoop manually. For this use case, using a simple `hadoop fs -put` or a mountable HDFS solution, like FUSE (Filesystem in Userspace) or the new NFS (Network File System) gateway, might provide a solution. We'll discuss these options next.

Data Ingestion Options

Now that we've discussed considerations around designing data ingestion pipelines, we'll dive more deeply into specific tools and methods for moving data into Hadoop. These range from simply loading files into HDFS to using more powerful tools such as Flume and Sqoop. As we've noted before, we're not attempting to provide in-depth introductions to these tools, but rather to provide specific information on effectively

leveraging these tools as part of an application architecture. The suggested references will provide more comprehensive and in-depth overviews for these tools.

We'll start by discussing basic file transfers in the next section, and then move on to discussing Flume, Sqoop, and Kafka as components in your Hadoop architectures.

File Transfers

The simplest and sometimes fastest way to get data into (and out of) Hadoop is by doing file transfers—in other words, using the `hadoop fs -put` and `hadoop fs -get` commands. For this reason file transfers should be considered as the first option when you are designing a new data processing pipeline with Hadoop.

Before going into more detail, let's review the characteristics of file transfers with Hadoop:

- It's an all-or-nothing batch processing approach, so if an error occurs during file transfer no data will be written or read. This should be contrasted with ingestion methods such as Flume or Kafka, which provide some level of failure handling and guaranteed delivery.
- By default file transfers are single-threaded; it's not possible to parallelize file transfers.
- File transfers are from a traditional filesystem to HDFS.
- Applying transformations to the data is not supported; data is ingested into HDFS as is. Any processing of the data needs to be done after it lands in HDFS, as opposed to in-flight transformations that are supported with a system like Flume.
- It is a byte-by-byte load, so any types of file can be transferred (text, binary, images, etc.).

HDFS client commands

As noted already, the `hadoop fs -put` command is a byte-by-byte copy of a file from a filesystem into HDFS. When the `put` job is completed, the file will be on HDFS as one or more blocks with each block replicated across different Hadoop DataNodes. The number of replicas is configurable, but the normal default is to replicate three times. To make sure that the blocks don't get corrupted, a checksum file accompanies each block.

When you use the `put` command there are normally two approaches: the *double-hop* and *single-hop*. The double-hop, shown in Figure 2-2, is the slower option because it involves an additional write and read to and from the disks on the Hadoop edge

node. Sometimes, though, this is the only option because the external source filesystem isn't available to be mounted from the Hadoop cluster.

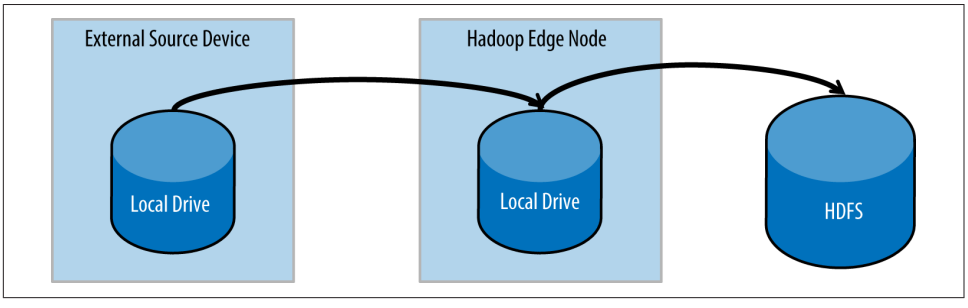

Figure 2-2. Double-hop file uploads

The alternative is the single-hop approach as shown in Figure 2-3, which requires that the source device is mountable—for example, a NAS or SAN. The external source can be mounted, and the `put` command can read directly from the device and write the file directly to HDFS. This approach has the benefits of improved performance. It also lessens the requirement to have edge nodes with large local drives.

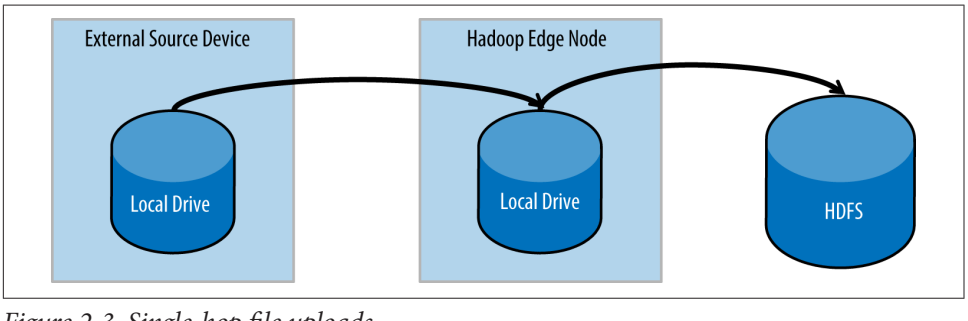

Figure 2-3. Single-hop file uploads

Mountable HDFS

In addition to using the commands available as part of the Hadoop client, there are several options available that allow HDFS to be mounted as a standard filesystem. These options allow users to interact with HDFS using standard filesystem commands such as `ls`, `cp`, `mv`, and so forth. Although these options can facilitate access to HDFS for users, they're of course still subject to the same limitations as HDFS:

- Just as with HDFS, none of these options offer full POSIX semantics.[1]
- Random writes are not supported; the underlying filesystem is still "write once, read many."

Another pitfall with mountable HDFS is the potential for misuse. Although it makes it easier for users to access HDFS and ingest files, this ease of access may encourage the ingestion of many small files. Since Hadoop is tuned for a relatively small number of large files, this scenario should be guarded against. Note that "many small files" could mean millions of files in a reasonably sized cluster. Still, for storage and processing efficiency it's better to store fewer large files in Hadoop. If there is a need to ingest many small files into Hadoop, there are several approaches that can be used to mitigate this:

- Use Solr for storing and indexing the small files. We'll discuss Solr in more detail in ???.
- Use HBase to store the small files, using the path and filename as the key. We'll also discuss HBase in more detail in ???.
- Use a container format such as SequenceFiles or Avro to consolidate small files.

There are several projects providing a mountable interface to Hadoop, but we'll focus on two of the more common choices, Fuse-DFS and NFS:

Fuse-DFS
: Fuse-DFS is built on the FUSE project,which was designed to facilitate the creation of filesystems on UNIX/Linux systems without the need to modify the kernel. Fuse-DFS makes it easy to mount HDFS on a local filesystem, but as a user space module, it involves a number of hops between client applications and HDFS, which can significantly impact performance. Fuse-DFS also has a poor consistency model. For these reasons you should carefully consider it before deploying it as a production solution.

NFSv3
: A more recent project adds support for the NFSv3 protocol to HDFS (see Figure 3-4). This provides a scalable solution with a minimal performance hit. The design involves an NFS gateway server that streams files to HDFS using the DFSClient. You can scale this solution by adding multiple NFS gateway nodes.

1 POSIX refers to a set of standards that operating systems should adhere to in order to support portability, including filesystem specifications. While HDFS is "POSIX-like," it does not fully support all the features expected of a POSIX-compliant filesystem.

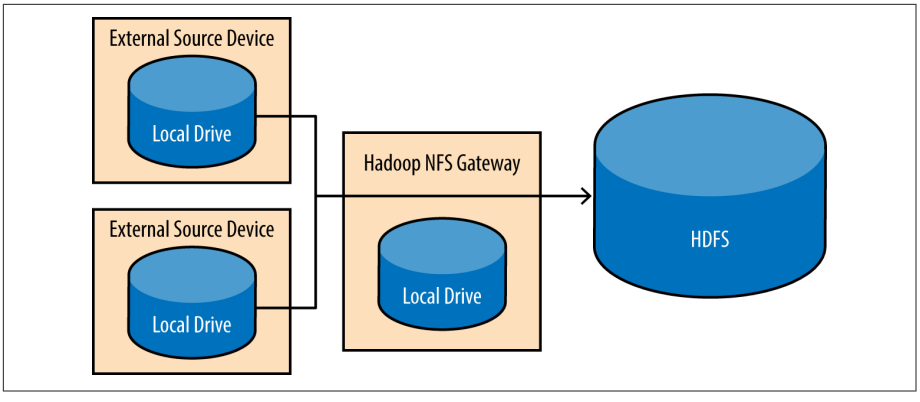

Figure 2-4. The HDFS NFSv3 gateway

Note that with the NFS gateway, writes are sent by the kernel out of order. This requires the NFS server to buffer and reorder the writes before sending them to HDFS, which can have an impact with high data volumes, both in performance and disk consumption. Before you deploy any mountable HDFS solution, it's highly recommended that you perform testing with sufficient data volumes to ensure that the solution will provide suitable performance. In general, these options are recommended for only small, manual data transfers, not for ongoing movement of data in and out of your Hadoop cluster.

Considerations for File Transfers versus Other Ingest Methods

Simple file transfers are suitable in some cases, particularly when you have an existing set of files that needs to be ingested into HDFS, and keeping the files in their source format is acceptable. Otherwise, the following are some considerations to take into account when you're trying to determine whether a file transfer is acceptable, or whether you should use a tool such as Flume:

- Do you need to ingest data into multiple locations? For example, do you need to ingest data into both HDFS and Solr, or into HDFS and HBase? In this case using a file transfer will require additional work after the files are ingested, and using Flume is likely more suitable.
- Is reliability important? If so, remember that an error mid-transfer will require a restart of the file transfer process. Here again, Flume is likely a more appropriate solution.
- Is transformation of the data required before ingestion? In that case, Flume is almost certainly the correct tool.

One option to consider if you have files to ingest is using the Flume spooling directory source. This allows you to ingest files by simply placing them into a specific

directory on disk. This will provide a simple and reliable way to ingest files as well as the option to perform in flight transformations of the data if required.

Sqoop: Batch Transfer Between Hadoop and Relational Databases

We've already discussed Sqoop quite a bit in this chapter, so let's get into more detail on considerations and best practices for it. As we've discussed, Sqoop is a tool used to import data in bulk from a relational database management system to Hadoop and vice versa. When used for importing data into Hadoop, Sqoop generates map-only MapReduce jobs where each mapper connects to the database using a Java database connectivity (JDBC) driver, selects a portion of the table to be imported, and writes the data to HDFS. Sqoop is quite flexible and allows not just importing full tables, but also adding `where` clauses to filter the data we import and even supply our own query.

For example, here is how you can import a single table:

```
sqoop import --connect jdbc:oracle:thin:@localhost:1521/oracle \
--username scott --password tiger \
--table  HR.employees --target-dir /etl/HR/landing/employee \
--input-fields-terminated-by "\t" \
--compression-codec org.apache.hadoop.io.compress.SnappyCodec --compress
```

And here is how to import the result of a join:

```
sqoop import \
  --connect jdbc:oracle:thin:@localhost:1521/oracle \
--username scott  --password tiger \
--query 'SELECT a.*, b.* FROM a JOIN b on (a.id == b.id) WHERE $CONDITIONS' \
--split-by a.id --target-dir /user/foo/joinresults
```

Note that in this example, `$CONDITIONS` is a literal value to type in as part of the command line. The `$CONDITIONS` placeholder is used by Sqoop to control the parallelism of the job, and will be replaced by Sqoop with generated conditions when the command is run.

In this section we'll outline some patterns of using Sqoop as a data ingestion method.

Choosing a split-by column

When importing data, Sqoop will use multiple mappers to parallelize the data ingest and increase throughput. By default, Sqoop will use four mappers and will split work between them by taking the minimum and maximum values of the primary key column and dividing the range equally among the mappers. The `split-by` parameter lets you specify a different column for splitting the table between mappers, and `num-mappers` lets you control the number of mappers. As we'll discuss shortly, one reason to specify a parameter for `split-by` is to avoid data skew. Note that each mapper will have its own connection to the database, and each will retrieve its portion of the table when we specify its portion limits in a `where` clause. It is important to choose a split

column that has an index or is a partition key to avoid each mapper having to scan the entire table. If no such key exists, specifying only one mapper is the preferred solution.

Using database-specific connectors whenever available

Different RDBMSs support different dialects of SQL language. In addition they each have their own implementation and their own methods of optimizing data transfers. Sqoop ships with a generic JDBC connector that will work with any database that supports JDBC, but there are also vendor-specific connectors that translate across different dialects and optimize data transfer. For example, the Teradata connector will use Teradata's FastExport utility to optimally execute data import, while the Oracle connector will disable parallel queries to avoid bottlenecks on the query coordinator.

Using the Goldilocks method of Sqoop performance tuning

In most cases, Hadoop cluster capacity will vastly exceed that of the RDBMS. If Sqoop uses too many mappers, Hadoop will effectively run a denial-of-service attack against your database. If we use too few mappers, data ingest rates may be too slow to meet requirements. It is important to tune the Sqoop job to use a number of mappers that is "just right"—that is, one that adheres to the Goldilocks principle.[2]

Since the risk of overloading the database is much greater than the risk of a slower ingest, we typically start with a very low number of mappers and gradually increase it to achieve a balance between Sqoop ingest rates and keeping the database and network responsive to all users.

Loading many tables in parallel with fair scheduler throttling

There is a very common use case of having to ingest many tables from the same RDBMS. There are two different approaches to implementing fair scheduler throttling:[3]

Load the tables sequentially
: This solution is by far the simplest but has the drawback of not optimizing the bandwidth between the RDBMS and the Hadoop cluster. To illustrate this, imagine that we have five tables. Each table is ingested using a different number of mappers because not all table sizes require allocating the full number of allowed mappers. The result of this example will look like Figure 2-5.

2 The Goldilocks principle states that something must fall within certain margins, as opposed to reaching extremes.

3 Fair schedulers are a method of sharing resources between jobs that allocate, on average, an equal share of the resources to each job. To read more on the Hadoop fair scheduler and how it can be used, see Chapter 7, "Resource Management" in Hadoop Operations by Eric Sammer (O'Reilly).

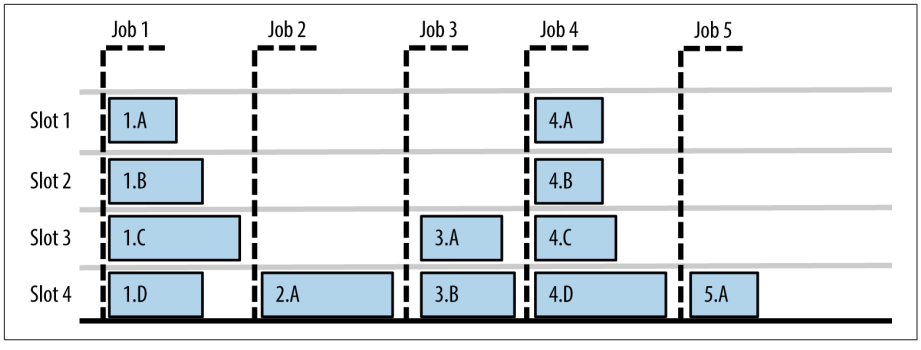

Figure 2-5. Sqoop tasks executing sequentially

As you can see from the figure, there are mappers left idle during certain jobs. This leads to the idea of running these jobs in parallel so we can optimally leverage the available mappers and decrease the processing and network time.

Load the tables in parallel

Running Sqoop jobs in parallel will allow you to use resources more effectively, but it adds the complexity of managing the total number of mappers being run against the RDBMS at the same time. You can solve the problem of managing the total number of mappers by using the fair scheduler to make a pool that has the maximum mappers set to the maximum number of mappers Sqoop should be allowed to use to interact with the RDBMS. If done correctly, the execution of the five jobs illustrated in the synchronized solution will look like Figure 2-6.

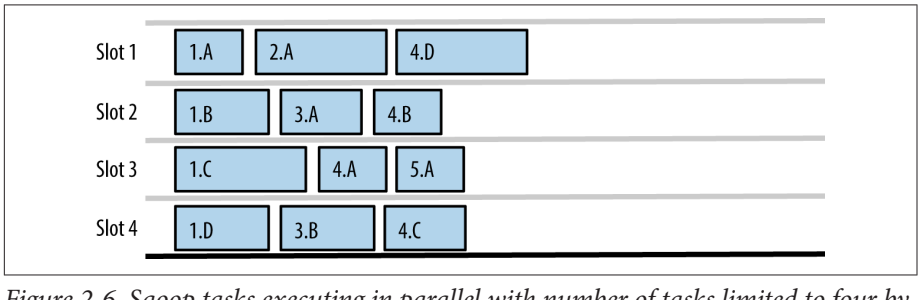

Figure 2-6. Sqoop tasks executing in parallel with number of tasks limited to four by the fair scheduler

Diagnosing bottlenecks

Sometimes we discover that as we increase the number of mappers, the ingest rate does not increase proportionally. Ideally, adding 50% more mappers will result in 50% more rows ingested per minute. While this is not always realistic, if adding more mappers has little impact on ingest rates, there is a bottleneck somewhere in the pipe.

Here are a few likely bottlenecks:

Network bandwidth
: The network between the Hadoop cluster and the RDBMS is likely to be either 1 GbE or 10 GbE. This means that your ingest rate will be limited to around 120 MBps or 1.2 GBps, respectively. If you are close to these rates, adding more mappers will increase load on the database, but will not improve ingest rates.

RDBMS
: Reading data from the database requires CPU and disk I/O resources on the database server. If either of these resources is not available, adding more mappers is likely to make matters worse. Check the query generated by the mappers. When using Sqoop in incremental mode, we expect it to use indexes. When Sqoop is used to ingest an entire table, full table scans are typically preferred. If multiple mappers are competing for access to the same data blocks, you will also see lower throughput. It is best to discuss Sqoop with your enterprise database administrators and have them monitor the database while importing data to Hadoop. Of course, it is also ideal to schedule Sqoop execution times when the database utilization is otherwise low.

Data skew
: When using multiple mappers to import an entire table, Sqoop will use the primary key to divide the table between the mappers. It will look for the highest and lowest values of the key and divide the range equally between the mappers. For example, if we are using two mappers, `customer_id` is the primary key of the table, and `select min(customer_id),max(customer_id) from customers` returns 1 and 500, then the first mapper will import customers 1—250 and the second will import customers 251—500. This means that if the data is skewed, and parts of the range actually have less data due to missing values, some mappers will have more work than others. You can use `--split-by` to choose a better split column, or `--boundary-query` to use your own query to determine the way rows will be split between mappers.

Connector
: Not using an RDBMS-specific connector, or using an old version of the connector, can lead to using slower import methods and therefore lower ingest rates.

Hadoop
: Because Sqoop will likely use very few mappers compared to the total capacity of the Hadoop cluster, this is an unlikely bottleneck. However, it is best to verify that Sqoop's mappers are not waiting for task slots to become available and to check disk I/O, CPU utilization, and swapping on the DataNodes where the mappers are running.

Inefficient access path
: When a primary key does not exist, or when importing the result of a query, you will need to specify your own split column. It is incredibly important that this column is either the partition key or has an index. Having multiple mappers run full-table scans against the same table will result in significant resource contention on the RDBMS. If no such split column can be found, you will need to use only one mapper for the import. In this case, one mapper will actually run faster than multiple mappers with a bad split column.

Keeping Hadoop updated

Ingesting data from an RDBMS to Hadoop is rarely a single event. Over time the tables in the RDBMS will change and the data in Hadoop will require updating. In this context it is important to remember that HDFS is a read-only filesystem and the data files cannot be updated (with the exception of HBase tables, where appends are supported). If we wish to update data, we need to either replace the data set, add partitions, or create a new data set by merging changes.

In cases where the table is relatively small and ingesting the entire table takes very little time, there is no point in trying to keep track of modifications or additions to the table. When we need to refresh the data in Hadoop, we simply re-run the original Sqoop import command, ingest the entire table including all the modifications, and then replace the old version with a newer one.

When the table is larger and takes a long time to ingest, we prefer to ingest only the modifications made to the table since the last time we ingested it. This requires the ability to identify such modifications. Sqoop supports two methods for identifying new or updated rows:

Sequence ID
: When each row has a unique ID, and newer rows have higher IDs than older ones, Sqoop can track the last ID it wrote to HDFS. Then the next time we run Sqoop it will only import rows with IDs higher than the last one. This method is useful for fact tables where new data is added, but there are no updates of existing rows.

  For example, first create a job:

```
sqoop job --create movie_id --import  --connect \
jdbc:mysql://localhost/movielens  \
--username training --password training \
--table movie --check-column id --incremental append
```

  Then, to get an incremental update of the RDBMS data, execute the job:

```
sqoop job --exec movie_id
```

Timestamp
: When each row has a timestamp indicating when it was created or when it was last updated, Sqoop can store the last timestamp it wrote to HDFS. In the next execution of the job, it will only import rows with higher timestamps. This method is useful for slowly changing dimensions where rows are both added and updated.

When running Sqoop with the `--incremental flag`, you can reuse the same directory name, so the new data will be loaded as additional files in the same directory. This means that jobs running on the data directory will see all incoming new data without any extra effort. The downside of this design is that you lose data consistency—jobs that start while Sqoop is running may process partially loaded data. We recommend having Sqoop load the incremental update into a new directory. Once Sqoop finishes loading the data (as will be indicated by the existence of a *_SUCCESS* file), the data can be cleaned and preprocessed before it is copied into the data directory. The new directory can also be added as a new partition to an existing Hive table.

When the incremental ingest contains not only new rows but also updates to existing rows, we need to merge the new data set with the existing one. Sqoop supports this with the command `sqoop-merge`. It takes the merge key (typically the primary key of the table), the old and new directories, and a target location as parameters. Sqoop will read both data sets and when two rows exist for the same key, it will keep the latest version. `sqoop-merge` code is fairly generic and will work with any data set created by Sqoop. If the data sets are known to be sorted and partitioned, the merge process can be optimized to take advantage of these facts and work as a more efficient map-only job.

Flume: Event-Based Data Collection and Processing

Flume is a distributed, reliable, and available system for the efficient collection, aggregation, and movement of streaming data. Flume is commonly used for moving log data, but can be used whenever there's a need to move massive quantities of event data such as social media feeds, message queue events, or network traffic data. We'll provide a brief overview of Flume here and then cover considerations and recommendations for its use. For more details on Flume, refer to the Flume documentation or *Using Flume* by Hari Shreedharan (O'Reilly).

Flume architecture

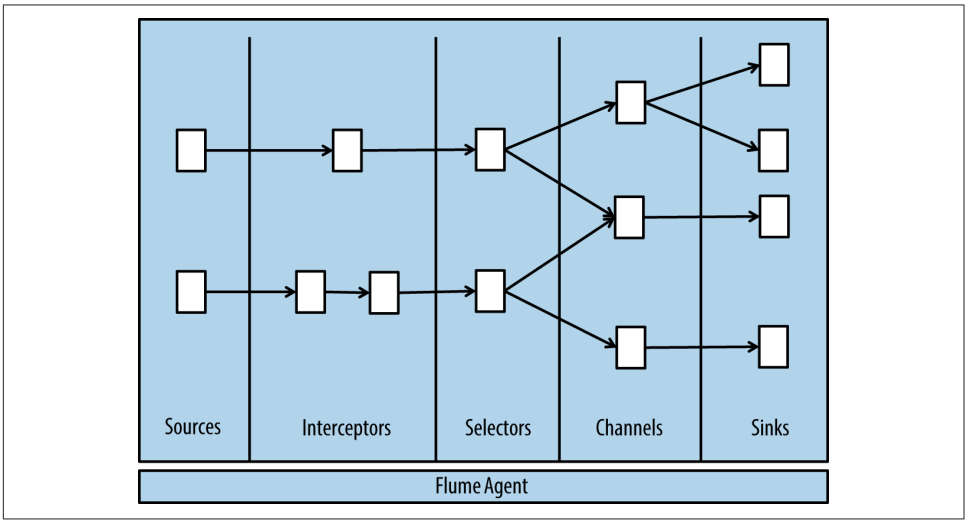

Figure 2-7. Flume components

Figure 2-7 shows the main components of Flume:

- Flume sources consume events from external sources and forward to channels. These external sources could be any system that generates events, such as a social media feed like Twitter, machine logs, or message queues. Flume sources are implemented to consume events from specific external sources, and many sources come bundled with Flume, including AvroSource, SpoolDirectorySource, HTTPSource, and JMSSource.
- Flume interceptors allow events to be intercepted and modified in flight. This can be transforming the event, enriching the event, or basically anything that can be implemented in a Java class. Some common uses of interceptors are formatting, partitioning, filtering, splitting, validating, or applying metadata to events.
- Selectors provide routing for events. Selectors can be used to send events down zero or more paths, so they are useful if you need to fork to multiple channels, or send to a specific channel based on the event.
- Flume channels store events until they're consumed by a sink. The most commonly used channels are the memory channel and the file channel. The memory channel stores events in memory, which provides the best performance among the channels, but also makes it the least reliable, because events will be lost if the process or host goes down. More commonly used is the disk channel, which provides more durable storage of events by persisting to disk. Choosing the right channel is an important architectural decision that requires balancing performance with durability.

- Sinks remove events from a channel and deliver to a destination. The destination could be the final target system for events, or it could feed into further Flume processing. An example of a common Flume sink is the HDFS sink, which, as its name implies, writes events into HDFS files.
- The Flume agent is a container for these components. This is a JVM process hosting a set of Flume sources, sinks, channels, and so on.

Flume provides the following features:

Reliability
: Events are stored in the channel until delivered to the next stage.

Recoverable
: Events can be persisted to disk and recovered in the event of failure.

Declarative
: No coding is required. The configuration specifies how components are wired together.

Highly customizable
: Although Flume includes a number of sources, sinks, and more out of the box, it provides a highly pluggable architecture that allows for the implementation of custom components based on requirements.

A Note About Guarantees

Although Flume provides "guaranteed" delivery, in the real world there's no way to provide a 100% guarantee. Failures can happen at many levels: memory can fail, disks can fail, even entire nodes can fail. Even though Flume can't provide 100% guarantees of event delivery, the level of guarantee can be tuned through configuration. Higher levels of guarantee of course come with a price in performance. It will be the responsibility of the architect to decide the level of guarantee required, keeping in mind the performance trade-offs. We'll discuss some of these considerations further in this section.

Flume patterns

The following patterns illustrate some common applications of Flume for data ingestion:

Fan-in
: Figure 2-8 shows an example of a fan-in architecture, probably the most common Flume architecture. In this example, there's a Flume agent on each source system, say web servers, which send events to agents on Hadoop edge nodes.

These edge nodes should be on the same network as the Hadoop cluster. Using multiple edge nodes provides reliability: if one edge node goes down, you don't lose events. It's also recommended to compress events before you send them to reduce network traffic. SSL can also be used to encrypt data on the wire if security is a concern. We'll see a full example of a Flume fan-in configuration in the case study on clickstream processing.

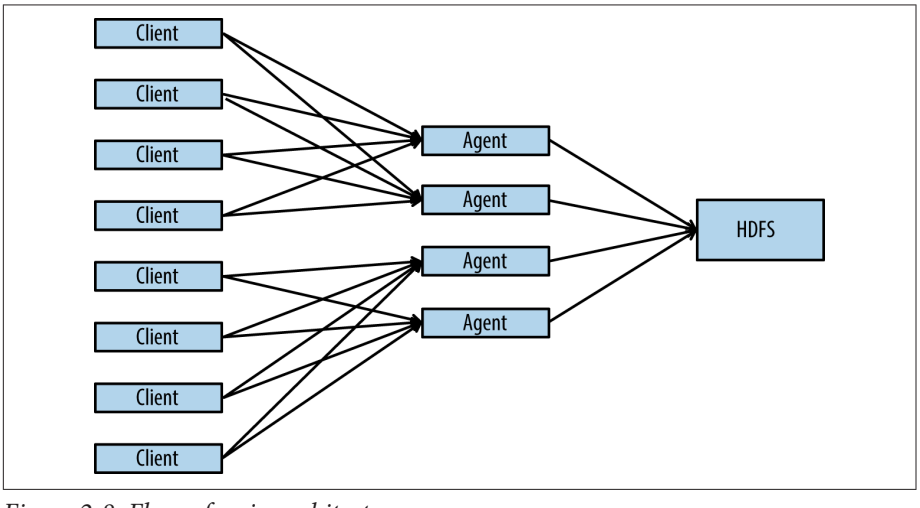

Figure 2-8. Flume fan-in architecture

Splitting data on ingest

Another very common pattern is to split events for ingestion into multiple targets. A common use of this is to send events to a primary cluster and a backup cluster intended for disaster recovery (DR). In Figure 2-9, we're using Flume to write the same events into our primary Hadoop cluster as well as a backup cluster. This DR cluster is intended to be a failover cluster in the event that the primary cluster is unavailable. This cluster also acts as a data backup, since effectively backing up Hadoop-sized data sets usually requires a second Hadoop cluster.

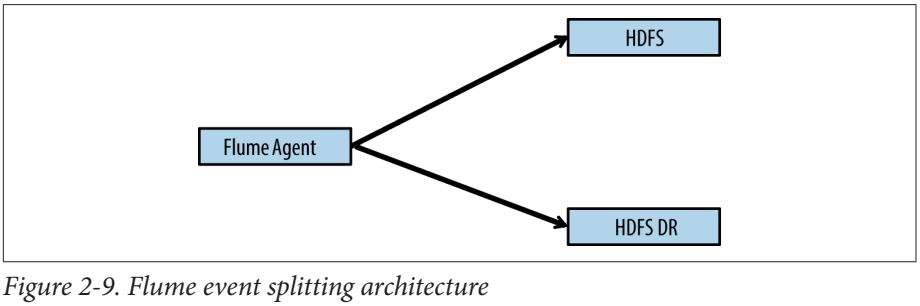

Figure 2-9. Flume event splitting architecture

Partitioning data on ingest

Flume can also be used to do partitioning of data as it's ingested, as shown in Figure 2-10. For example, the HDFS sink can partition events by timestamp.

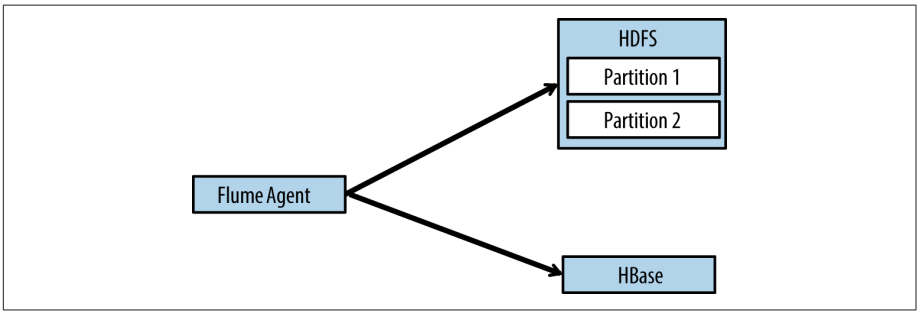

Figure 2-10. Flume partitioning architecture

Splitting events for streaming analytics

Up until now we have only talked about a persistent layer as final targets for Flume, but another common use case is sending to a streaming analytics engine such as Storm or Spark Streaming where real-time counts, windowing, and summaries can be made. In the case of Spark Streaming, the integration is simple since Spark Streaming implements the interface for the Flume Avro source, so all we have to do is point a Flume Avro sink to Spark Streaming's Flume Stream.

File formats

The following are some file format considerations when you're ingesting data into HDFS with Flume (we talked more about storage considerations in Chapter 1).

Text files

Text is a very common format for events processed through Flume, but is not an optimal format for HDFS files for the reasons discussed in Chapter 1. In general, when you're ingesting data through Flume it's recommended to either save to SequenceFiles, which is the default for the HDFS sink, or save as Avro. Note that saving files as Avro will likely require some extra work to convert the data, either on ingestion or as post-processing step. We'll provide an example of Avro conversion in the clickstream processing case study. Both SequenceFiles and Avro files provide efficient compression while also being splittable. Avro is preferred because it stores schema as part of the file, and also compresses more efficiently. Avro also checkpoints, providing better failure handling if there's an issue writing to a file.

Columnar formats

Columnar file formats such as RCFile, ORC, or Parquet are also not well suited for Flume. These formats compress more efficiently, but when you use them with

Flume, they require batching events, which means you can lose more data if there's a problem. Additionally, Parquet requires writing schema at the end of files, so if there's an error the entire file is lost.

Writing to these different formats is done through Flume *event serializers*, which convert a Flume event into another format for output. In the case of text or sequence files, writing events to HDFS is straightforward because they're well supported by the Flume HDFS sink. As noted earlier, ingesting data as Avro will likely require additional work, either on ingest or as a post-processing step. Note that there is an Avro event serializer available for the HDFS sink, but it will use the event schema for creating the Avro data, which will most likely not be the preferred schema for storing the data. It's possible, though, to override the EventSerializer interface to apply your own logic and create a custom output format. Creating custom event serializers is a common task, since it's often required to make the format of the persisted data look different from the Flume event that was sent over the wire. Note that interceptors can also be used for some of this formatting, but event serializers are closer to the final output to disk, while the interceptor's output still has to go through a channel and a sink before getting to the serialization stage.

Recommendations

The following are some recommendations and best practices for using Flume.

Flume sources. A Flume source is of course the origin for a Flume pipeline. This is where data is either pushed to Flume (say, in the case of the Avro source), or where Flume is pulling from another system for data (for example, in the case of the JMS source). There are two main considerations in configuring Flume sources that are critical to optimal Flume performance:

Batch size
: The number of events in a Flume batch can have dramatic effects on performance. Let's take the Avro source as an example. In this example, a client sends a batch of events to the Avro source, and then must wait until those events are in the channel and the Avro source has responded back to the client with an acknowledgment of success. This seems simple enough, but network latency can have a large impact here; as Figure 2-11 shows, even a couple of milliseconds of delay in a ping could add up to a significant delay in processing events. In this diagram there is a 12-millisecond delay to get a batch through. Now, if you are loading a million events with a batch size of 1 versus a batch size of 1,000, it would take 3.33 hours for the first batch, versus 12 seconds for the second batch. As you can see, setting an appropriate batch size when configuring Flume is important to optimal performance. Start with 1,000 events in a batch and adjust up or down from there based on performance.

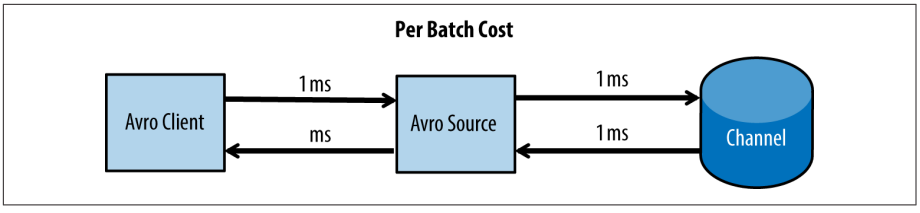

Figure 2-11. Effect of latency on flume performance

Threads
: Understanding how Flume sources utilize threads will allow you to optimally take advantage of multithreading with those sources. In general, more threads are good up to the point at which your network or CPUs are maxed out. Threading is different for different sources in Flume. Let's take the Avro source as an example; the Avro source is a Netty server[4] and can have many connections to it at the same time, so to make the Avro source multithreaded you simply add more clients or client threads. A JMS source is very different; this source is a pull source, so to get more threads you need to configure more sources in your Flume agent.

Flume sinks. Remember, sinks are what cause your data to land in its final destination. Here are some considerations for configuring Flume sinks in your Flume agents:

Number of sinks
: A sink can only fetch data from a single channel, but many sinks can fetch data from that same channel. A sink runs in a single thread, which has huge limitations on a single sink—for example, throughput to disk. Assume with HDFS you get 30 MBps to a single disk; if you only have one sink writing to HDFS then all you're going to get is 30 MBps throughput with that sink. More sinks consuming from the same channel will resolve this bottleneck. The limitation with more sinks should be the network or the CPU. Unless you have a really small cluster, HDFS should never be your bottleneck.

Batch Sizes
: Remember with all disk operations, buffering adds significant benefits in terms of throughput. Also remember for a Flume sink to guarantee to the channel that an event is committed, that event needs to be on disk. This means that the output stream to disk needs to be flushed, which requires the overhead of an `fsync` system call. If the sink is getting small batches, then a lot of time will be lost to executing these system calls. The only big downside to large batches with a sink is an

4 Netty is a Java-based software framework that's designed to support simplified development of client-server networking applications.

increased risk of duplicate events. For example, if a Flume process dies after some events have been written to HDFS but not acknowledged, those events might get rewritten when the process restarts. It's therefore important to carefully tune batch sizes to provide a balance between throughput and potential duplicates. However, as we noted earlier, it's relatively easy to add a post-processing step to remove duplicate records, and we'll show an example later in the book.

Flume interceptors. Interceptors can be a very useful component in Flume. The capability to take an event or group of events and modify them, filter them, or split them provides powerful functionality, but this comes with a warning. A custom interceptor is of course custom code, and custom code comes with risk of issues like memory leaks or consuming excessive CPU. Just remember with great power comes greater risk of introducing bugs in production, so it's very important to carefully review and test any custom interceptor code.

Flume memory channels. If performance is your primary consideration, and data loss is not an issue, then memory channel is the best option. Keep in mind that with the memory channel, if a node goes down or the Java process hosting the channel is killed, the events still in the memory channel are lost forever. Further, the number of events that can be stored is limited by available RAM, so the memory channel's ability to buffer events in the event of downstream failure is limited.

Flume file channels. As noted before, since the file channel persists events to disk, it's more durable than the memory channel. It's important to note, though, that it's possible to make trade-offs between durability and performance depending on how file channels are configured.

Before we proceed with recommendations for file channels, note that in addition to maintaining events on disk, the file channel also writes events to periodic checkpoints. These checkpoints facilitate restart and recovery of file channels, and should be taken into consideration when you're configuring file channels to ensure reliability.

Here are some considerations and recommendations for using file channels:

- Configuring a file channel to use multiple disks will help to improve performance because the file channel will round-robin writes between the different disks. Further, if you have multiple file channels on a single node, use distinct directories, and preferably separate disks, for each channel.
- In addition to the preceding point, using an enterprise storage system such as NAS can provide guarantees against data loss, although at the expense of performance and cost.
- Use dual checkpoint directories. A backup checkpoint directory should be used to provide higher guarantees against data loss. You can configure this by setting

`useDualCheckpoint` to `true`, and setting a directory value for `backupCheckpointDir`.

We should also note the much less commonly used JDBC channel that persists events to any JDBC-compliant data store, such as a relational database. This is the most durable channel, but also the least performant due to the overhead of writing to a database. The JDBC channel is generally not recommended because of performance, and should only be used when an application requires maximum reliability.

A Note About Channels

Although we've been discussing the difference between the memory channel and file channel and why you might prefer one over the other, there are times when you'll want to use both in the same architecture. For example, a common pattern is to split events between a persistent sink to HDFS, and a streaming sink, which sends events to a system such as Storm or Spark Streaming for streaming analytics. The persistent sink will likely use a file channel for durability, but for the streaming sink the primary requirements will likely be maximum throughput. Additionally, with the streaming data flow dropping events is likely acceptable, so the best-effort and high-performance characteristics of the memory channel make it the better choice.

Sizing Channels. When configuring a Flume agent, you'll often face questions around whether to have more channels or fewer, or how to size the capacity of those channels. Here are some simple pointers:

Memory channels
: Consider limiting the number of memory channels on a single node. Needless to say, the more memory channels you configure on a single node, the less memory is available to each of those channels. Note that a memory channel can be fed by multiple sources and can be fetched from by multiple sinks. It's common for a sink to be slower than the corresponding source, or vice versa, making it necessary to have more of the slower component connected to a single channel. You would want to consider more memory channels on a node in the case where you have different pipelines, meaning the data flowing through the pipeline is either different or going to a different place.

File channels
: When the file channel only supported writing to one drive, it made sense to configure multiple file channels to take advantage of more disks. Now that the file channel can support writing to multiple drives or even to multiple files on the same drive, there's no longer as much of a performance benefit to using multiple

file channels. Here again, the main reason to configure multiple file channels on a node is the case where you have different pipelines. Also, whereas with the memory channel you need to consider the amount of memory available on a node when configuring your channels, with the file channel you of course need to consider the amount of disk space available.

Channel size
: Remember the channel is a buffer, not a storage location. Its main goal is to store data until it can be consumed by sinks. Normally, the buffer should only get big enough so that its sinks can grab the configured batch sizes of data from the channel. If your channel is reaching max capacity, you most likely need more sinks or need to distribute the writing across more nodes. Larger channel capacities are not going to help you if your sinks are already maxed out. In fact, a larger channel may hurt you; for example, a very large memory channel could have garbage collection activity that could slow down the whole agent.

Finding Flume bottlenecks

There are many options to configure Flume, so many that it may be overwhelming at first. However, these options are there because Flume pipelines are meant to be built on top of a variety of networks and devices. Here is a list of areas to review when you're trying to identify performance issues with Flume:

Latency between nodes
: Every client-to-source batch commit requires a full round-trip over the network. A high network latency will hurt performance if the batch size is low. Using many threads or larger batch sizes can help mitigate this issue.

Throughput between nodes
: There are limits to how much data can be pushed through a network. Short of adding more network capacity, consider using compression with Flume to increase throughput.

Number of threads
: Taking advantage of multiple threads can help improve performance in some cases with Flume. Some Flume sources support multiple threads, and in other cases an agent can be configured to have more than one source.

Number of sinks
: HDFS is made up of many drives. A single HDFS sink will only be writing to one spindle at a time. As we just noted, consider writing with more than one sink to increase performance.

Channel
: If you're using file channels, understand the performance limitations of writing to disk. As we previously noted, a file channel can write to many drives, which can help improve performance.

Garbage collection issues
: When using the memory channel, you might experience full garbage collections; this can happen when event objects are stored in the channel for too long.

Kafka

Apache Kafka is a distributed publish-subscribe messaging system. It maintains a feed of messages categorized into topics. Applications can publish messages on Kafka, or they can subscribe to topics and receive messages that are published on those specific topics. Messages are published by *producers* and are pulled and processed by *consumers*. As a distributed system Kafka runs in a cluster, and each node is called a *broker*.

For each topic, Kafka maintains a *partitioned log*. Each partition is an ordered subset of the messages published to the topic. Partitions of the same topic can be spread across many brokers in the cluster, as a way to scale the capacity and throughput of a topic beyond what a single machine can provide. Messages are ordered within a partition and each message has a unique *offset*. A combination of topic, partition, and offset will uniquely identify a message in Kafka. Producers can choose which partition each message will be written to based on the message key, or they can simply distribute the messages between partitions in a round-robin fashion.

Consumers are registered in *consumer groups*; each group contains one or more consumers, and each consumer reads one or more partitions of a topic. Each message will be delivered to only one consumer within a group. However, if multiple groups are subscribed to the same topic, each group will get all messages. Having multiple consumers in a group is used for load balancing (supporting higher throughput than a single consumer can handle) and high availability (if one consumer fails, the partitions it was reading will be reassigned to other consumers in the group).

As you know from the preceding discussion, a topic is the main unit of application-level data separation. A consumer or consumer group will read all data in each topic it is subscribed to, so if only a subset of the data is of interest for each application, it should be placed in two different topics. If sets of messages are always read and processed together, they belong in the same topic.

A partition, on the other hand, is the main unit of parallelism. Each partition has to fit in a single server, but a topic can be as large as the sum of its partitions. In addition, each partition can be consumed by at most one consumer from the same group, so while you can add more consumers to increase read throughput, you are effectively limited by the number of partitions available in a topic. Therefore, we recom-

mend at least as many partitions per node as there are servers in the cluster, possibly planning for a few years of growth. There are no real downsides to having a few hundred partitions per topic.

Kafka stores all messages for a preconfigured length of time (typically weeks or months) and only keeps track of the offset of the last message consumed by each consumer. This allows consumers to recover from crashes by rereading the topic partitions starting from the last good offset. Consumers can also rewind the state of the queue and reconsume messages. The rewind feature is especially useful for debugging and troubleshooting. By storing all messages for a period of time and not tracking acknowledgments for individual consumers and messages, Kafka can scale to more than 10,000 consumers, support consumers that read data in infrequent batches (MapReduce jobs, for example), and maintain low latency even with very high messaging throughput. Traditional message brokers that track individual acknowledgments typically need to store all unacknowledged messages in memory. With large number of consumers or consumers that read data in batches, this often causes swapping and severe degradation of performance as a result.

The following are some common use cases for Kafka:

- Kafka can be used in place of a traditional message broker or message queue in an application architecture in order to decouple services.
- Kafka's most common use case is probably high rate activity streams, such as website clickstream, metrics, and logging.
- Another common use case is stream data processing, where Kafka can be used as both the source of the information stream and the target where a streaming job records its results for consumption by other systems. We'll see examples of this in ???.

Kafka fault tolerance

Each topic partition can be replicated to multiple brokers, to allow continued service and no data loss in case of broker crash. For each partition, one replica is designated as a *leader* and the rest as followers. All reads and writes are done from the leader. The followers serve as insurance: if the leader crashes, a new leader will be elected from the list of synchronized replicas. A follower is considered synchronized if it did not fall too far behind the leader in replication.

When writing data, the producer can choose whether the write will be acknowledged by all synchronized replicas or just the leader, or to not wait for acknowledgments at all. The more acknowledgments the producer waits for, the more it can be sure the data written will not be lost in a crash. However, if reliability is not a concern, then waiting for fewer acknowledgements will allow for higher throughput. Administra-

tors can also specify a minimum threshold for the number of synchronized replicas—if the number of synchronized replicas falls below this limit, writes that require acknowledgment from all synchronized replicas will be rejected. This provides extra assurance that messages will not be accidentally written to just a single broker, where they may be lost if this broker crashes.

Note that if a network failure occurs after a message was sent but before it was acknowledged, the producer has no way to check if it was written to a broker. It is up to the producer to decide whether to resend the message and risk duplicates or to skip it and risk missing data.

When reading data, the consumer will only read *committed* messages—that is, messages that were written to all synchronized replicas. This means that consumers don't need to worry about accidentally reading data that will later disappear when a broker crashes.

Kafka guarantees that all messages sent by a producer to a specific topic partition will be written to the partition in the order they were sent. The consumers will see those messages in the order they were sent. And if a partition was defined with replication factor of *N*, a failure of *N*—1 nodes will not cause any message loss.

Kafka can support *at least once*, *at most once*, and *exactly once* delivery semantics, depending on the configuration of the consumer:

- If the consumer advances the offset in Kafka after processing messages (default settings), it is possible that a consumer will crash after processing the data but before advancing the offset. This means that a new consumer will reread messages since the last offset; messages will be processed *at least once*, but there may be duplicates.
- If the consumer first advances the offset and then processes the messages, it may crash after the offset was advanced but before data was processed. In this case, the new consumer will read data from the new offset, skipping messages that were read but not processed by the crashed consumer. Messages will be processed *at most once*, but there may be data loss.
- If the consumer uses a two-phase commit to process data and advance the offset at the same time, messages will be processed *exactly once*. Since two-phase commits have high overhead, this is best done in batch consumers and not streaming processes. Camus, a Hadoop consumer that we'll discuss later, has *exactly once* semantics—both data and offset are getting written to HDFS by the mapper, and both will be removed if a mapper fails.

To complete the review of Kafka's high availability guarantees, we'll mention that Kafka can be deployed in multiple data centers. In this setup, one cluster serves as a

consumer of another cluster, and as such replicates all messages with the same HA guarantees available to all consumers.

Kafka and Hadoop

Kafka is a generic message broker and was not written specifically for Hadoop. However, there are several use cases that combine both systems. One use case involves consuming messages from Kafka and storing them in HDFS for offline processing. For example, when tracking clickstream activity on a website, you can use Kafka to publish site activities such as page views and searches. These messages can then be consumed by real-time dashboards or real-time monitors, as well as stored in HDFS for offline analysis and reporting.

Another common use case is real-time stream processing. Kafka provides a reliable stream of messages as the data source for Spark Streaming or Storm. This use case will be covered in detail in ???.

A common question is whether to use Kafka or Flume for ingest of log data or other streaming data sources into Hadoop. As usual, this depends on requirements:

- Flume is a more complete Hadoop ingest solution; it has very good support for writing data to Hadoop, including HDFS, HBase, and Solr. It is configuration-based, so Hadoop administrators can deploy and use Flume without writing any code. Flume handles many common issues in writing data to HDFS, such as reliability, optimal file sizes, file formats, updating metadata, and partitioning.
- Kafka is a good fit as a reliable, highly available, high performance data source. If the requirements involve fault-tolerant message delivery, message replay, or a large number of potential consumers, Kafka is an excellent choice. However, this means you will have to develop your own producers and consumers rather than rely on existing sources and sinks.

Looking at how Flume and Kafka compare, it seems that they are complementary—Kafka provides a flexible, high-throughput, and reliable message bus, whereas Flume provides an easy-to-use collection of sources, interceptors, and Hadoop integration. It therefore can make a lot of sense to use both.

Flume includes a Kafka source, Kafka sink, and Kafka channel. These allow for sending events from Kafka to Flume sinks (such as HDFS, HBase, and Solr), or from Flume sources (such as Log4j and Netcat) to Kafka, and they even use Kafka to enhance Flume with a reliable channel.

Flume's Kafka source is a Kafka consumer, which reads data from Kafka and sends it to the Flume channel where it can continue flowing through the Flume topology. Configuring the source is as simple as setting a topic, the ZooKeeper server used by Kafka, and the channel. The source allows for tuning the batch sizes that will be sent

from the source to the channel. Use smaller batches for low latency and larger batches for higher throughput and reduced CPU utilization.

By default, Flume uses the groupId `flume` when reading from Kafka. Adding multiple Flume sources with the same groupId will mean that each Flume agent will get a subset of the messages and can increase throughput. In addition, if one of the sources fails, the remaining sources will rebalance so that they can continue consuming all messages. Flume's Kafka source is reliable and will not lose messages if a source, channel, sink, or agent fails.

Flume's Kafka sink is a Kafka producer, which sends data from a Flume channel to Kafka. Configuring a Kafka sink requires setting a topic and a list of Kafka brokers. Similar to the Kafka source, it is also possible to tune the batch size on the Kafka sink for improved throughput or reduced latency.

Flume's Kafka channel combines a producer and a consumer. When a Flume source sends messages to the Kafka channel, these events will be sent to a Kafka topic. Each batch will be sent to a separate Kafka partition, so the writes will be load-balanced. When the Kafka sink reads from the channel, the events will be consumed from Kafka. The Kafka channel is highly reliable. When used with a properly configured Kafka server, it can guarantee that messages will not be lost, or even significantly delayed, in the event of server crashes.

Streaming data is not always appropriate. Batching typically supports better throughputs, and is a better fit for compressed columnar storage such as Parquet. For batch loads from Kafka, we recommend using Camus, a separate open source project that allows for ingesting data from Kafka to HDFS. Camus is flexible and relatively well maintained. It features automatic discovery of Kafka topics from ZooKeeper, conversion of messages to Avro and Avro schema management, and automatic partitioning. In the setup stage of the job, it fetches the list of topics and the ID of the first messages to consume from ZooKeeper, and then splits the topics among the map tasks. Each map task pulls messages from Kafka and writes them to HDFS directories based on timestamps. Camus map tasks will only move the messages to the final output location when the task finishes successfully. This allows the use of Hadoop's speculative execution without the risk of writing duplicate messages in HDFS.

Figure 2-12 is a high-level diagram to help give you an idea of how Camus works.

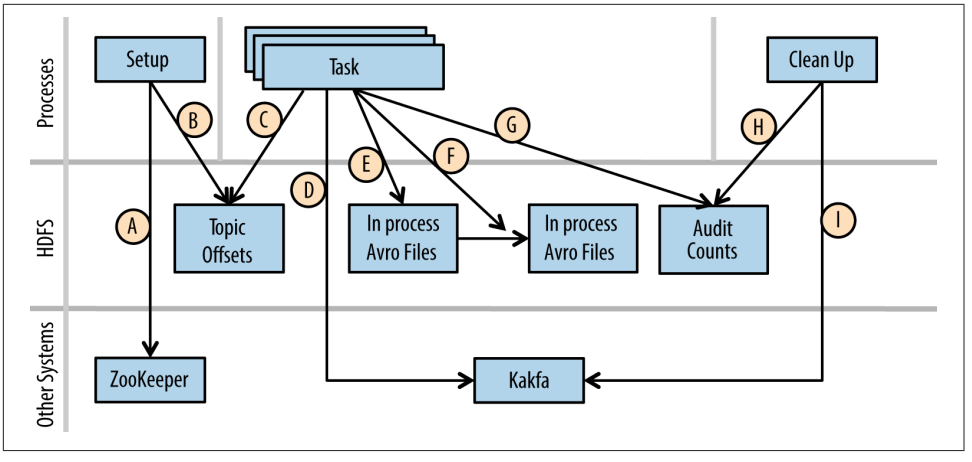

Figure 2-12. Camus execution flow

This diagram details the following series of steps:

- A: The setup stage fetches broker URLs and topic information from ZooKeeper.
- B: The setup stage persists information about topics and offsets in HDFS for the tasks to read.
- C: The tasks read the persisted information from the setup stage.
- D: The tasks get events from Kafka.
- E: The tasks write data to a temporary location in HDFS in the format defined by the user-defined decoder, in this case Avro-formatted files.
- F: The tasks move the data in the temporary location to a final location when the task is cleaning up.
- G: The task writes out audit counts on its activities.
- H: A clean-up stage reads all the audit counts from all the tasks.
- I: The clean-up stage reports back to Kafka what has been persisted.

To use Camus, you may need to write your own decoder to convert Kafka messages to Avro. This is similar to a serializer in Flume's HDFSEventSink.

Data Extraction

The bulk of this chapter has been concerned with getting data into Hadoop, which is where you generally spend more time when designing applications on Hadoop. Getting data out of Hadoop is, of course, also an important consideration though, and many of the considerations around getting data into Hadoop also apply to getting it

out. The following are some common scenarios and considerations for extracting data from Hadoop:

Moving data from Hadoop to an RDBMS or data warehouse
: A common pattern is to use Hadoop for transforming data for ingestion into a data warehouse—in other words, using Hadoop for ETL. In most cases, Sqoop will be the appropriate choice for ingesting the transformed data into the target database. However, if Sqoop is not an option, using a simple file extract from Hadoop and then using a vendor-specific ingest tool is an alternative. When you're using Sqoop it's important to use database-specific connectors when they're available. Regardless of the method used, avoid overloading the target database—data volumes that are easily managed in Hadoop may overwhelm a traditional database. Do yourself and your database administrator a favor and carefully consider load on the target system.

  Another consideration is that as Hadoop matures and closes the gap in capabilities with traditional data management systems, we'll see more workloads being moved to Hadoop from these systems, reducing the need to move data out of Hadoop.

Exporting for analysis by external applications
: Related to the previous item, Hadoop is a powerful tool for processing and summarizing data for input to external analytical systems and applications. For these cases a simple file transfer is probably suitable—for example, using the `Hadoop fs -get` command or one of the mountable HDFS options.

Moving data between Hadoop clusters
: Transferring data between Hadoop clusters is also common—for example, for disaster recovery purposes or moving data between multiple clusters. In this case the solution is DistCp, which provides an easy and efficient way to transfer data between Hadoop clusters. DistCp uses MapReduce to perform parallel transfers of large volumes of data. DistCp is also suitable when either the source or target is a non-HDFS filesystem—for example, an increasingly common need is to move data into a cloud-based system, such as Amazon's Simple Storage System (S3).

Conclusion

There's a seemingly overwhelming array of tools and mechanisms to move data into and out of Hadoop, but as we discussed in this chapter, if you carefully consider your requirements it's not difficult to arrive at a correct solution. Some of these considerations are:

- The source systems for data ingestion into Hadoop, or the target systems in the case of data extraction from Hadoop
- How often data needs to be ingested or extracted
- The type of data being ingested or extracted
- How the data will be processed and accessed

Understanding the capabilities and limitations of available tools and the requirements for your particular solution will enable you to design a robust and scalable architecture.

About the Authors

Mark Grover is a committer on Apache Bigtop and a committer and PMC member on Apache Sentry (incubating) and a contributor to Apache Hadoop, Apache Hive, Apache Sqoop, and Apache Flume projects. He is also a section author of O'Reilly's book on Apache Hive, *Programming Hive*.

Ted Malaska is a Senior Solutions Architect at Cloudera helping clients be successful with Hadoop and the Hadoop ecosystem. Previously, he was a Lead Architect at the Financial Industry Regulatory Authority (FINRA), helping build out a number of solutions from web applications and service-oriented architectures to big data applications. He has also contributed code to Apache Flume, Apache Avro, Yarn, and Apache Pig.

Jonathan Seidman is a Solutions Architect at Cloudera working with partners to integrate their solutions with Cloudera's software stack. Previously, he was a technical lead on the big data team at Orbitz Worldwide, helping to manage the Hadoop clusters for one of the most heavily trafficked sites on the Internet. He's also a co-founder of the Chicago Hadoop User Group and Chicago Big Data, technical editor for Hadoop in Practice, and he has spoken at a number of industry conferences on Hadoop and big data.

Gwen Shapira is a Solutions Architect at Cloudera. She has 15 years of experience working with customers to design scalable data architectures. She was formerly a senior consultant at Pythian, Oracle ACE Director, and board member at NoCOUG. Gwen is a frequent speaker at industry conferences and maintains a popular blog.

Colophon

The animal on the cover of *Hadoop Application Architectures* is a manatee (family Trichechidae), of which there are three extant species: the Amazonian, the West Indian, and the West African.

Manatees are fully aquatic mammals that can weigh up to 1,300 pounds. The name *manatee* is derived from the Taíno word *manatí*, meaning "breast." They are thought to have evolved from four-legged land mammals over 60 million years ago; their closest living relatives are elephants and hyraxes.

Though they live exclusively underwater, manatees often have coarse hair and whiskers. They also have thick wrinkled skin and a prehensile upper lip that is used to gather food. Manatees are herbivores and spend half the day grazing on fresh- or saltwater plants. In particular, they prefer the floating hyacinth, pickerel weeds, water lettuce, and mangrove leaves. The upper lip is split into left and right sides that can

move independently of one another, and the lip can even help in tearing apart plants to aid in chewing.

Manatees tend to be solitary animals except when searching for a mate or nursing their young. They emit a wide range of sounds for communication, and are similar to dolphins and seals in how they retain knowledge, engage in complex associative learning, and master simple tasks.

In the wild, manatees have no natural predators. Instead, the greatest threats to manatees come from humans—boat strikes, water pollution, and habitat destruction. The problem of boat strikes is so pervasive in the manatee population of the middle Atlantic that in 2008, boats were responsible for a quarter of manatee deaths. Additionally, a large portion of that population had severe scarring and sometimes mutilation from encounters with watercraft. All three species of manatee are listed as vulnerable to extinction by the World Conservation Union.

Many of the animals on O'Reilly covers are endangered; all of them are important to the world. To learn more about how you can help, go to *animals.oreilly.com*.

The cover image is from the *Brockhaus Lexicon*. The cover fonts are URW Typewriter and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.