O'REILLY®

Compliments of pentaho®

PREVIEW EDITION

Hadoop Application Architectures

DESIGNING REAL WORLD BIG DATA APPLICATIONS

Mark Grover, Ted Malaska,
Jonathan Seidman & Gwen Shapira

pentaho

Bring Your Big Data to **Life**

Big Data Integration and Analytics

Optimize for Hadoop and more.
Learn how at pentaho.com

This Preview Edition of *Hadoop Application Architectures*, *Chapters 1 and 2*, is a work in progress. The final book is currently scheduled for release in April 2015 and will be available at *oreilly.com* and other retailers once it is published.

Hadoop Application Architectures

Mark Grover, Ted Malaska, Jonathan Seidman, and Gwen Shapira

Beijing • Cambridge • Farnham • Köln • 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://my.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: FIX ME!
Copyeditor: FIX ME!
Proofreader: FIX ME!
Indexer: FIX ME!
Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrator: Rebecca Demarest

April 2015: First Edition

Revision History for the First Edition:

2014-06-23: Early release revision 1

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

ISBN: 978-1-491-90008-6

[?]

Table of Contents

1. Data Modeling in Hadoop. 1
Data Storage Options 2
Standard File Formats 3
Hadoop File Types 5
Serialization Formats 7
Columnar formats 9
Compression 11
HDFS Schema Design 13
Location of HDFS files 14
Advanced HDFS Schema design 16
HBase Schema Design 19
Row Key 20
Timestamp 22
Hops 22
Tables and Regions 23
Using Columns 25
Using Column Families 26
Time-to-live (TTL) 27
Managing Metadata 27
What is metadata? 28
Why care about metadata? 28
Where to store metadata? 29
Examples of managing metadata 31
Limitations of Hive metastore and HCatalog 31
Other ways of storing metadata 32

2. Data Movement. 35
Data Ingestion Considerations 35
Timeliness of data ingestion 36

Incremental updates 38
Access patterns 39
Original source system and data structure 40
Transformations 43
Network Bottleneck 44
Network Security 44
Push or Pull 44
Build to Handle Failure 45
Level of complexity 46
Data Ingestion Options 46
File Transfers 47
Considerations for File Transfers vs. Other Ingest Methods 50
Sqoop 51
Flume 56
Kafka 65
Data Extraction 68
Summary 69

CHAPTER 1
Data Modeling in Hadoop

At its core, Hadoop is a distributed data store which provides a platform for implementing powerful parallel processing frameworks on it. The reliability of this data store when it comes to storing massive volumes of data couple with its flexibility related to running multiple processing frameworks makes it an ideal choice as the hub for all your data. This characteristic of Hadoop means that you can store any type of data as-is, and 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, un-processed 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 stored in it. This leads to lengthy cycles of analysis, data modeling, data transformation, loading, testing, etc. before data can be accessed. Furthermore, if a wrong decision is made, or requirements change, this cycle needs to 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. More and more 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 the ability of being able to store all of your raw data is a powerful feature, there are still many factors that should be taken into consideration before dumping your data into Hadoop. These considerations include:

- How the data is stored: There are a number of file formats and compression formats supported on Hadoop. Each of these have particular strengths which make them 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.
- Multi-tenancy: It's common for clusters to host multiple users, groups, and application types. Supporting multi-tenant clusters involves a number of important considerations when 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.
- Managing metadata: 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 cover them in the first chapter.

Another important factor when making storage decisions with Hadoop, but out of the scope of this book, is security and associated considerations. This includes decisions around authentication, fine-grained access control, and encryption - both for data on the wire and data at rest.

Data Storage Options

One of the most fundamental decisions to make when 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 file system, Hadoop allows for storage of data in any format, whether it's text, binary, images, etc. 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 is probably more straightforward than selecting file formats, but 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 un-compress, and not surprisingly require more CPU. The ability to split compressed files is also a very important consideration when working with data stored in Hadoop, and a topic we'll explore further.
- Data storage system: While all data in Hadoop rests in HDFS, there are decisions around what the underlying storage manager should be - i.e. 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 file system in this chapter and throughout the book, we'd be remiss in not mentioning work to enable alternate file systems with Hadoop. This includes open-source file systems such as GlusterFS and the Quantcast File Sytem, 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 file system 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 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 below 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 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 come in many other forms: CSV files, or unstructured data such as emails, etc. A primary consideration when storing text data in Hadoop is the organization of the files in the file system, which we'll discuss more in the section on schema design. Additionally, you'll want to select a compression format for the files, since text files can very quickly consume considerable space on your Hadoop cluster. Also, keep in mind that there is an overhead of type conversion associated with storing data in text format. For example, when storing *1234* in a text file and using it as an integer, requires a String to Integer conversion during read, and vice-versa during writing. This overhead adds up when you do a lot of such conversions.

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 provide the ability for 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 which makes 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 are 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 of these formats. 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[1]. For JSON, the Elephant Bird project[2] provides the `LzoJsonInputFormat`. For more details on processing these formats, the book Hadoop in Practice[3] provides several examples for processing XML and JSON files with MapReduce.

Binary Data

Although text is probably the most common source data format stored in Hadoop, Hadoop can also be used 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.

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 RCFiles 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 on compression, but note that the ability to split files can be a key consideration when storing data in Hadoop, since 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 the most commonly used file based format in Hadoop, but other file based formats are available such as MapFiles, SetFiles, ArrayFiles, and BloomMapFiles. Since these formats were specifically designed to work with MapReduce, they

1. *https://cwiki.apache.org/confluence/display/PIG/PiggyBank*
2. *https://github.com/kevinweil/elephant-bird/*
3. Holmes, Alex. Hadoop in Practice. Manning, 2012

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, since that's the format most commonly employed in implementing Hadoop jobs. For a more complete discussion of the other formats such as MapFiles and SetFiles, please refer to (insert reference to H:TDG)

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 I/O and take up more space on disk than the same data in compressed form.
- Record-compressed. Compresses each record as it's added to the file. Record-compression is not as efficient at compression as block-compression.
- Block compressed. 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 when using SequenceFiles. Also, the reference to *block* here is unrelated to the HDFS or filesystem *block*. A *block* in Block compression refers to a block 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 after every block in the file.

SequenceFiles are well supported within the Hadoop ecosystem, however their support outside of the ecosystem is limited. A common use case for SequenceFiles is as a container for smaller files. Since 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, please refer to (insert H:TDG reference).

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 sync to block boundaries.

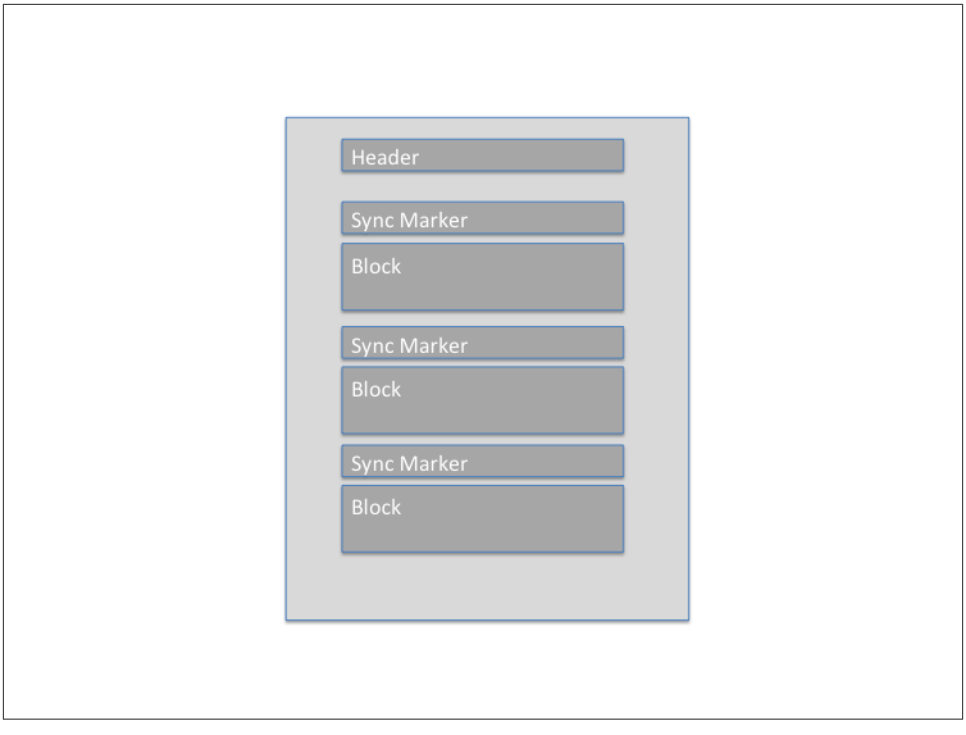


Figure 1-1. An example of a SequenceFile using block compression

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: inter-process 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, since it was specifically created as a replacement for 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 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 using 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 using 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 datafiles are compressible and splittable. Another important feature of Avro that makes it superior to SequenceFiles for Hadoop applications is support for schema evolution — 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 noted above, 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.

Avro defines a small number of primitive types such as boolean, int, float, string, and also supports 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 can also be more efficient when writing data, particularly if all columns of the record are available at write time since 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 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.
- Compression on columns is generally very efficient, particularly if the column has few distinct values.
- Columnar storage 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 ORC and Parquet.

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, highly efficient storage space utilization, and strong adaptivity to highly dynamic workload patterns. RCFiles are similar to SequenceFiles, except data is stored in a column-oriented fashion. 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

will likely replace the use of RCFile. RCFile is still a fairly common format used with Hive storage.

ORC

The Optimized Row Columnar (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:

- Light-weight, 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 as well as 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 underway 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. 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. Parquet provides the following benefits, many of which it shares with ORC:

- Similar to ORC files, 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.
- Designed to support complex nested data structures.
- Parquet stores full metadata at the end of files, so Parquet files are self-documenting.

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 effect 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 in order to improve performance of data processing. Since a major overhead in processing large amounts of data is disk 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 codecs supported on Hadoop are splittable. Since the MapReduce framework splits data for input to multiple tasks, having a non-splittable compression codec provides 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 codec, as well as file format. We'll discuss the various compression codecs 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. An important thing to note is 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, but its speed performance is not as good as Snappy. Gzip is also not splittable, so should be used with a container format. It should be noted that one reason Gzip is sometimes slower than Snappy for processing is that due to Gzip compressed files taking up fewer blocks, fewer tasks are required for processing the same data. For this reason, when using Gzip, using smaller blocks 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. For this reason, it's not an ideal codec for Hadoop storage, unless the primary need is for reducing the storage footprint. Like Snappy, bzip2 is not inherently splittable.

Compression Recommendations

- 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 like data is close together will provide better compression levels. For example, if you have stock ticks with the following 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 unique field such as time or price.
- Consider using a compact file format with support for splittable compression such as Avro. Figure 1-2 helps to illustrate how Avro or SequenceFiles support splittability with otherwise non-splittable compression formats, as well as illustrating the difference between HDFS blocks and Avro or SequenceFile blocks. Note that a single HFDS block can contain multiple Avro or SequenceFile blocks.

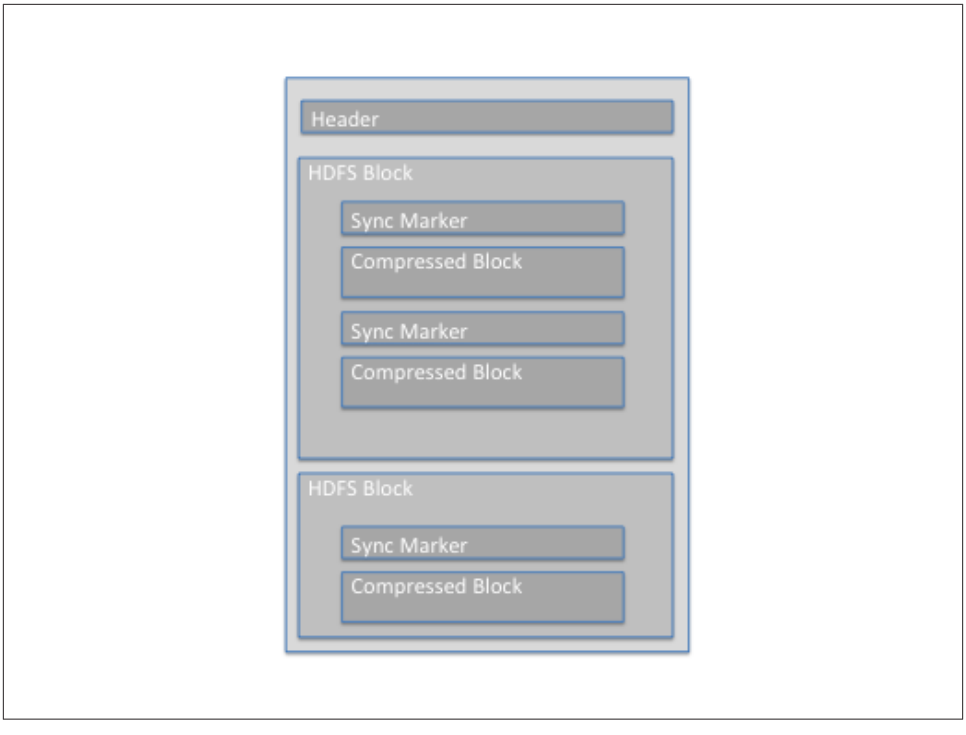


Figure 1-2. An example of compression with Avro

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 pointed out 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 ways (which we will discuss further in Chapter 2) without having the need to associate a schema or pre-process the data before processing it.

While many people use Hadoop for storing and processing unstructured data such as images, videos, emails or blog posts, or semi-structured data such as XML documents and log files, 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:

- Standard directory structure makes it easier to share data between teams working with the same data sets.
- Often times, you'd want to "stage" data in a separate location before all of it is ready to be processed. Conventions regarding staging data will help ensure that partially-loaded data will not get accidentally processed as if it was complete.
- Standardized organization of data will allow reusing code that processes it.
- Standardized locations also allow enforcing access and quota controls to prevent accidental deletion or corruption.
- Some tools in the Hadoop ecosystem sometimes make assumptions regarding the placement of data. It is often simpler to match those assumptions when 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. While unstructured and semi-structured 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. Knowing in advance the main use cases and data retrieval requirements will result in 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 thing to decide when designing an HDFS schema is the location of the files. Standard locations make it easier to find and share data between teams. Here's an example HDFS directory structure that we recommend:

- `/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.
- `/etl`: Data is in various stages of being processed by an ETL (extract, transform, and load) workflow.
- `/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.
- `/data`: Data sets that are shared across the organization. Since these are often critical data sources for analysis that drive business decisions, there are often controls around who can read and write this data. Very often user access is read only and data is written by automated (and audited) ETL processes
- `/app`: Everything required for Hadoop applications to run, except data. This includes: jar files, Oozie workflow definitions, Hive HQL files, etc.

The above directory structure simplifies the assignment of permissions to various groups and users. The directories under `/user` will typically only be readable and writable by the users who own them. `/etl` directory will be readable and writable by ETL processes (they typically run under their own user) and members of the ETL team. `/tmp` is typically readable and writable by everyone. Because 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 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 dataset in a directory named `/data/medication_orders`.

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 sub directory under the `/etl` directory. Within each of these application specific directory, you would have a directory for each ETL process or workflow for that application. Within the workflow directory, there are usually 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. The final structure will look similar to this: /etl/<group>/<application>/<process>/{input,processing,output,bad}. For example, if your Business Intelligence team has a click-

stream analysis application and one of its processes is to aggregate user preferences, the recommended name for the directory that contains the output data for this process would be: `/etl/BI/clickstream/aggregate-preferences/output`.

The application code directory `/app` is used for application artifacts such as jars for Oozie actions or Hive 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`.

Advanced HDFS Schema design

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

Partitioning

Partitioning of a data set is a very common technique used to reduce the amount of I/O required when processing the data set. When 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, however this means that every query will have to read the entire data set even when processing 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 sets, each such subset being called a partition. Each of these partitions would be present in a sub-directory of the directory containing the entire data set. This will allow queries to only read 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 last 3 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 last 3 months will only need to read 90 or so partitions and not the entire data set.

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

The above directory structure is understood by various tools like HCatalog, Hive, Impala, Pig, etc. 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 managable subsets. It is similar to hash-partitions used in many relational databases. In the example above, 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.

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 (also known as, 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.

An additional benefit of bucketing becomes apparent when joining two data sets. 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 the need 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 the cost of doing a reduce side join is $O(n^2)$ where n is the number of records in each data set. However, when 2 bucketed data sets are joined, there are b joins that need to be done on n/b records from both data sets which leads to b joins of $O(n^2/b^2)$ complexity. If these joins are done sequentially, that would be $O(n^2/b)$ time complexity which is b times faster than a non-bucketed join. Of course, if some or all of these joins can be done in parallel, the time savings would be even more. Moreover, since the

buckets are typically small enough to easily fit into memory, this means the entire join can be done in the map stage of a map-reduce job by loading the smaller of the buckets in memory. This is called map-side join and improves the join performance as compared to 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 merge join and not store the entire bucket in memory when joining. This is somewhat faster than 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.

You will note from the discussion above that 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, there is the option of storing the same data set multiple times, each with different physical organization. This is considered an anti-pattern in relational databases, but when using 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, cost of storage in Hadoop clusters is significantly lower, so there is less concern about wasted disk space. These attributes allow us to trade-off space for greater query speed, which is often desirable.

Denormalizing

Although we talked about joins in the previous subsections, another method of trading diskspace 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 normal form. This type of schema is designed to minimize duplication of data across tables 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 pre-joined data sets. The idea is to minimize the amount of work that will be done by queries by doing as much as possible of the required work 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 OLTP schema and a HDFS schema of a particular use-case, you will see that the Hadoop schema consolidates many of the small dimension tables into few larger dimensions by joining them during the ETL process. In 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 pre-processing of data 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 large number of queries, is worth doing once and reusing. In relational databases this pattern is often known as "Materialized Views". In the next chapter on Data Ingestion, we will discuss methods of keeping the Materialized Views updated.

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 designing successful design patterns for solving common problems with HBase.

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, you can associate values with keys and perform fast lookup of the values based on a given key.

There many details related to how regions and compactions work in HBase, various strategies for ingesting data into HBase, usage and understanding of block cache, etc. that we are glossing over when using the above simple analogy. However, it still serves as a very apt analogy, 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: . Put things in a hash table . Get things from a hash table . Iterate through a hash table (Note that HBase does give us more power with range scans here) . Increment the value of a hash table entry . Delete values from a hash table

It also makes sense to answer the question of why would you want to give up SQL for HBase. The value proposition of HBase lies in its scalability and flexibility. HBase finds its use in many applications, a popular one being Fraud Detection, which we will be discussing in more detail in Chapter 7. 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

In continuation of 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. Here are some reasons why row keys bear such a high importance in HBase schema design:

Getting

The row key is the key used when retrieving records from HBase.

Distribution

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 timestamp for row keys because it would make most of the `put` and `get` requests to be focussed on a single region and hence a single region server which somewhat defeats the purpose of having a distributed system. It's best to choose row keys so the load on the cluster is fairly distributed.

Block Cache

The block cache is a least recently used (LRU) cache that caches data blocks in memory. The idea behind the caching is that recently fetched records 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.

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 desired.

A bad choice in choosing the row key can lead to sub-optimal population of the block cache. For example, if you choose your row key to be a hash of some attribute, the block cache will be populated with random records which will have a very low likelihood of resulting in a cache hit. An alternative design in such a case would be to salt the first part of the key with something meaningful that allows records fetched together to be close in row key sort order.

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 8 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

In general, a shorter row key is better than a longer row key due to lower storage overhead and faster performance. However, longer row keys often lead to better `get`/`scan` properties. Therefore, there is always a tradeoff involved in choosing the right row key length.

Let's take a look at an example. Table 1-1 shows a table with 3 records in HBase.

Table 1-1. Example HBase table

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

The larger the size of 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 as well, so in general, it's also a good practice to keep the column names small.

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 alike 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.

Readability

While this is a very subjective point, we recommend that you 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

Keep in mind that row keys have to be unique. 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:

1. Timestamps determine which records are newer in case of a `put` request to modify the record.
2. Timestamps determine the order in which records are returned when multiple versions of a single record are requested.
3. Timestamps are also used to decide if a major compaction is going to remove the record in question because the time to live (TTL) when compared to the timestamp has elapsed.

By default, when 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 which 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 2 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 table shows a `courses` table with a student id and list of courses that the student is taking.

Table 1-3. Students table

Student Number	Student Name
11001	Bob
11002	David
11003	Charles

Table 1-4. Courses table

Student Number	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 2 hop request. The first hop will retrieve Charles' student number from the 'students` table and the second hop will retrieve the list of Charles' courses using the student number.

Like we alluded to in "Advanced HDFS Schema design" on page 16, examples like the above would be a good contender for denormalization since they would reduce the number of hops required to complete the request.

In general, while it's definitely possibly to have multi-hop requests with HBase, it's best to avoid them by better schema design (for example, by leveraging denormalization). This is because every hop is a round trip to HBase which costs 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 can lead to a significant imbalance in the distribution of load on one or more nodes of the cluster.

Figure 1-3 depicts a diagram that shows a topology of Region Servers, Regions and Tables in an example 3 node HBase cluster.

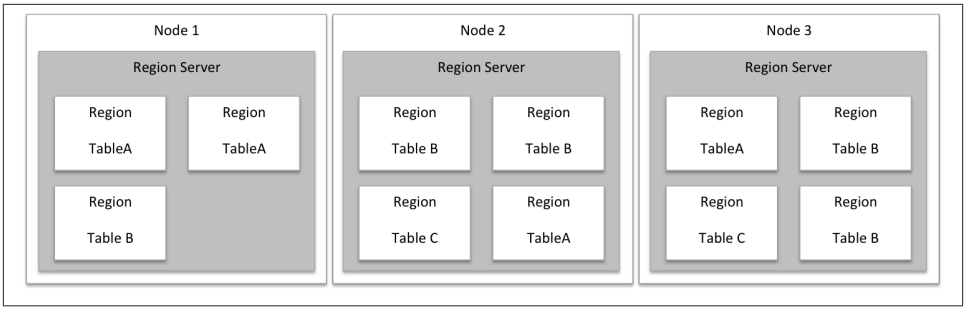


Figure 1-3. Topology of Region Servers, Regions and Tables

The important things to note here are:

- There is one region server per node.
- There many regions in a region server.
- At any given 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 can be used to select the right number of regions for a given table. These rules demonstrate a tradeoff 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. Therefore, the more regions that exist in a region server, the less memstore space available per region. This results in smaller flushes which in turn leads to more minor compactions.

Compaction time A larger region takes longer to compact. The empirical upper limit on the size of a region is around 20 GB.

The assignment of regions to an HBase table can be done in one of the two following ways:

1. The table is created with the default 1 region which then splits as data increases.
2. The table is created with a given number of regions and the region size is set to a high enough value (e.g. 100 GB per region) to avoid auto-splitting.

For most cases, we recommend to preselect the region count (Option #2) of the table to avoid performance impact of seeing random region splitting and sub-optimal region split ranges.

However, in some cases, automatic splitting (option #1) may make more sense. One such use case is a forever growing dataset where only the most recent data is updated. If the row key for this table is comprised 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 (except for TTL).

Using Columns

The concept of columns in an HBase table is very different than that in a traditional RDBMS. In HBase, unlike a 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 know the difference.

To understand the above a little better, 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 example below 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 2 columns named foo and bar. The benefits of doing so are:

1. We can get one column at a time independently of the other column
2. We can modify just each column independently of the other
3. 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. Using two separate columns for foo and bar shows the structure of such an HFile on disk:

Using two separate columns for foo and bar.

```
|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:

1. 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.
2. Both the column values would need to be updated together since they are stored as a single entity (column).

3. Both the columns would age out together based on the last update.

Using 1 column for both foo and bar shows the structure of the HFile in such a case:

Using 1 column for both foo and bar.

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

The amount of space consumed on disk plays a non-trivial role in deciding 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 WAL?
- How many records can fit into the Memstore flush?
- How long a compaction would take?

An important point to note in the above examples is the one-character column names. 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, hence the choice of single character column names is fairly common.

Using Column Families

While we just discussed the concept of columns above, there is also a concept of column families in HBase. A column family is essentially a container for columns. A table can have one or more column families. The important thing to note 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 column family is when the operations and/or their cadence being done on a subset of the columns of a table is significantly different from the other columns.

For example, let's take an HBase table with 2 columns. `column1` contains about 400 bytes per row and `column2` contains about 20 bytes. Now let us say that the value of `column1` gets set once and never changes but that of `column2` changes very often and that the access patterns call `get` requests on `column2` a lot more than on `column1`.

In such a case, two Column Families makes sense for the following reasons:

- **Lower Compaction Cost:** If we had 2 separate column families, the column family with `column 2` will have memstore flushes very frequently which will produce mi-

nor compactions. Since `column 2` is in its own column family, HBase will only need to compact 5% of the total records's worth of data, thereby making the compactions less impactful on performance.

- **Better Use of Block Cache:** As you saw earlier, when getting a record from HBase, the records close by to the requested record (in the same HBase block) are pulled into the block cache. If both `column 1` and `column 2` are in the same column family, the data for both `column 1` and `column 2` would be pulled into the block cache with each `get` request on `column 2`. This results in sub-optimal population of the block cache because the block cache would contain `column 1` data which will be used very infrequently since `column 1` receives very few `get` requests. Having `column 1` and `column 2` in separate column families would result in the block cache being populated with values only from `column 2` thereby increasing the number of cache hits on subsequent `get` requests on `column 2`.

Time-to-live (TTL)

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. With TTL, the idea is that if on a major compaction the timestamp is more than the specified TTL in the past, then 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. Let's say if you needed to remove all data past 7 years without using TTL, you would have to scan all 7 years of data every day and then insert a delete record for every record that is older than 7 years. Not only do you have to scan all 7 years, you have to create new delete records, which could be multi 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 a 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 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:

1. 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), schema associated with the data set [4], partitioning and sorting properties of the data set, if any, format of the data set, if applicable (e.g. CSV, TSV, SequenceFile, etc.). Such metadata is usually stored in a separate metadata repository.
2. Metadata about files on HDFS. This includes information like permissions and ownership of such files, location of various blocks of that file on datanodes, etc. Such information is usually stored and managed by Hadoop Namenode.
3. Metadata about tables in HBase. This includes information like table names, associated namespace, associated attributes (e.g. MAX_FILESIZE, READONLY, etc.), names of column families, etc. Such information is stored and managed by HBase itself.
4. Metadata about data ingest and transformations. This includes information like which user generated a given data set, where did the data set come from, how long did it take to generate it, how many records or alternatively, what was the size of the data loaded.
5. Metadata about data set statistics. This includes information like number of rows in a data set, number of unique values in each column, histogram of the distribution of data, 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 above - metadata about logical data sets. From here on, in this section, the word "metadata" would refer to metadata as it pertains to the first point above.

Why care about metadata?

There are two important reasons here:

1. 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.

4. 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 1 schema with the same data.

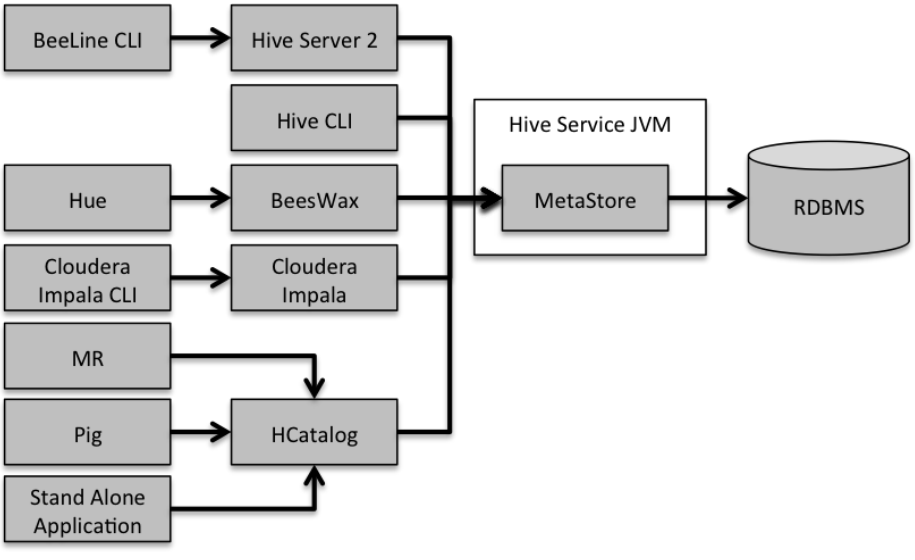
2. It allows you supply information about your data (e.g. partitioning or sorting properties) that can then be leveraged by various tools (written by you or someone else) to enforce them while populating data and leverage them when querying data.
3. It allows data management tools to "hook" into this 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 service in Hive that accesses the 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 collective logical system of Hive metastore service and database that provides metadata access to Hive and other systems.

As time progressed, 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 a 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 a accessibility veener around Hive metastore.



Now MapReduce, Pig, and stand alone applications could very well talk directly to the Hive metastore through it's APIs, but HCatalog allows easy access through it's WebHCat Rest APIs and it allows the cluster administrators to lock down access to the Hive Metastore for security concerns.

Note, 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 3 modes - embedded metastore, local metastore and remote metastore. While 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. 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. The exact answer to whether you should reuse the existing database instance instead of creating a new one depends on usage patterns of, existing load on, and other applications using the existing 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 - 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 da-

tabase 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 a High Availability for your Hive metastore database, it may make sense to use the existing HA database cluster for 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` commands creates metadata, `ALTER TABLE` command 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 Pig to store and retrieve metadata. See here for an example. If you are using a programming interface to querying data in Hadoop (e.g. MapReduce, Spark, Cascading, etc.), you can use HCatalog's Java API to store and retrieve metadata. For example, see here (Add something on github for MR, Crunch, Cascading, Spark, etc.) HCatalog also has a CLI and a REST API which can be used to store, update and retrieve metadata.

Limitations of Hive metastore and HCatalog

- Problems with high availability: In order to provide High Availability (HA) for Hive metastore, you have to provide high availability for the metastore database as well as the metastore service. 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 high availability to the Hive metastore database. As far as the metastore services goes, there is support concurrently to run multiple metastore on more than one node in the cluster. However, at the time of this writing, that has can lead to concurrency issues related to DDL statements and other queries being run at the same time [5]. The Hive community is working towards fixing these issues though.
- Fixed schema for metadata: While Hadoop provides a lot of flexibility on the type of data that can be stored, especially 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 meta-information 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

5. See HIVE-4759, for example

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 Hive metastore may not be the right tool for it.

- Another moving part: While not necessarily a limitation, you should keep in mind that 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 if/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 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 HDFS Schema Design section of this chapter, we recommend embedding some metadata in data set locations for organization and consistency. For example, in case of a partitioned data set, the directory structure would look like: `<data set name>/<partition_column_name=partition_column_value>/{files}`. Such a structure already contains the name of the data set the name of the partition column and the various values of the partitioning column the data set is partitioned on. Tools and applications can then leverage this metadata in file names and locations when processing. For example, to list all partition for a data set named `medication_orders`, would be a simple `ls` operation on `/data/medication_orders` directory of HDFS.

Storing the metadata in HDFS

You may decide to store the metadata on HDFS itself. One scheme 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 (see more details below). This is especially useful if you feel constrained by what metadata you can store in Hive metastore via HCatalog. Hive metastore and therefore HCatalog has a fixed schema of 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 [6] to store metadata. Moreover, Kite supports multiple Metadata providers which means while you can use Kite to store metadata in HDFS like described above, you can still use Kite to store data in HCatalog (and hence Hive metastore) via its integration with HCatalog. You can also easily transform metadata from one source (say HCatalog) to another (say .metadata directory in HDFS).

6. Kite is a set of libraries, tools, examples, and documentation focused on making it easier to build systems on top of the Hadoop ecosystem. You can read more at *http://kitesdk.org/* Kite allows users to create and manage metadata on HDFS like described above. Kite creates 2 files in the `.metadata` directory - `descriptor.properties` that the location and format of the data in the data set, and `schema.avsc`

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 one, are the architectural decisions on getting that data into your Hadoop cluster. Although Hadoop provides a file system 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 importing data into Hadoop from these different systems. As organizations move more and more data into Hadoop, these decisions will become even more critical. 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, semi-structured, un-structured 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 Chapter five, 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.
- Micro Batch – 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 100 millisecond range.

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

An important thing to note with these time classifications is 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 built-in 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 even processed with MapReduce and 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 like SequenceFile or Avro that supports splittable compression.

As the requirements move from batch processing to more frequent updates, tools like Flume or Kafka need to be considered. Sqoop and file transfers are not going to be a good selection as the delivery requirements get faster than 2 minutes. Further, as the requirements get to less than 2 minutes the storage layer may need to change to HBase or Solr for more granular insertions and read operations. As the requirements get to 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 are 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. We'll talk more about near real-time streaming with tools like Storm and Spark Streaming in chapter five. We'll also discuss how HBase and Solr fit into the near real-time and real-time category in chapter five. 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 dataset or modify it. If the requirements are for append only, 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 once 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 old data. Note that chapter one provides a full discussion of partitioning and organizing data in HDFS.

A very important thing to note when appending to directories with additional files is that HDFS is optimized for large files. If the requirements call for a 2 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. Longer consecutive scans on a file are faster than many seeks to load multiple files.

If the requirements are append and deltas, then additional work is required to land the data properly in HDFS. HDFS is read only — we can't update records in place as you would with a relational database. In this case a *compaction* job is required to handle deltas. A compaction job is one where the data is sorted by a primary key. If the row is found twice then the data from the newer 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 un-compacted 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

Note there are many ways to implement compaction with very different performance results. This will be discussed further in Chapter 3.

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 to take effect on completing an HBase `Put`, which typically occurs in milliseconds. There are many ways to ingest data into HBase with different levels of guarantees and timeliness attributes. At the end of this section we'll provide a summary that highlights these options along with their pros and cons.

Note that HBase requires additional administration and application development knowledge. Also, HBase has much different access patterns than HDFS that should be considered. Scan rates are an example of a difference between HDFS and HBase. HBase scan rates are about 8-10x slower then HDFS. Another difference is random access; HBase can access a single record in milliseconds while 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 then HDFS may not be the best fit, and HBase might be a better choice. Conversely, if scans and data transformations are required then 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 simplicity, best compression, and highest scan rate are called for, HDFS is the default selection. In addition, with newer versions of HDFS (Hadoop 2.3.0 and above) 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 Solr should be the considered.

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.
Crunch	Easy Java interface to do MapReduce or Spark processing interchangably	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.

Tool	Use Cases	Storage Device
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
Search Strings	Simple text-based interface to request data or free-form natural language search on text data	Solr

Original source system and data structure

When ingesting data from a file system, the following items should be considered:

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 often optimizing an ingestion pipeline requires looking at the system that the data is being pulled from. Hard drives can have read speeds of anywhere from 20MB/s to 100MB/s, 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. On a typical drive, 3 threads is normally required to maximize throughput.

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, lets take a CSV file. CSV is of course a very common format, and a simple CSV file can generally be easily imported into a Hive table for immediate access and processing. However, if the underlying storage of that CSV file were converted to a columnar format such as Parquet, it very likely could be processed much more efficiently, while taking up considerably less space on disk.

Another consideration here 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 left most 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 type is *car* then 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 wil 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, Spark, etc.

Compression

There is a pro and a con to compressing data on the original file system. 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 used with a splittable container format like SequenceFiles, RCFiles, Parquet Files, or Avro Files. 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 discuss compression considerations with HDFS in chapter one, and we'll look at concrete examples of ingestion and post-processing in the case studies later in the book.

Relational Database Managment Systems (RDBMS)

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 Sqoop. With simple SQL statements the user can define what data they wish to ingest or extract from Hadoop. Sqoop has many configuration options, with one of the most prominent being the ability to define the amount of resources used to pull the data.

Sqoop is a very rich tool with lots of options, but at the same time 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, 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 then it will likely be necessary 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.

A couple of important things to note with Sqoop:

- Sqoop input formats are normally not the file formats desired in the long run, so a batch job is most likely required at some point to convert the data.[1]
- With the Sqoop Architecture the data nodes and 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.

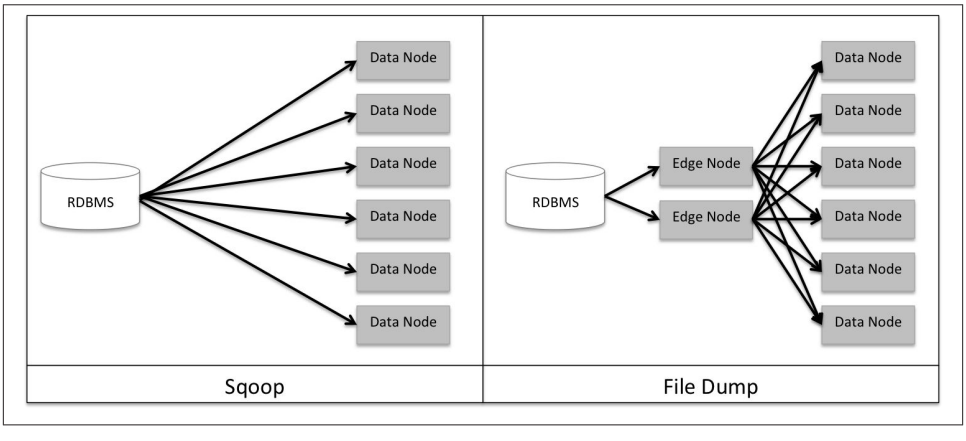


Figure 2-1. Sqoop vs RDBMS file export

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

Streaming data

Examples of streaming input data includes twitter feeds, a 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.

Log Files

Log files get their own section because they're a cross between file system and streaming input. An anti-pattern is to read the log files from disk as they are getting written, since this is almost impossible to implement without losing data. The correct way of doing

1. Note that this applies to Sqoop 1. Sqoop 2 Sqoop 2 holds the promise of fixing this problem, but at the time of writing is not yet feature complete.

this is to stream the logs directly to a tool like Flume or Kafka, which will write directly to Hadoop instead.

Transformations

In general, this decision point refers to the need to make 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: Converting XML or JSON 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, Java MapReduce or newer tools like Spark. The use of post-processing for this batch transformation step is preferred because of the check points for failure provided by this processing, including the check sum 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. This would be done by making MapReduce output failure records to a predefined failure directory. 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 usng interceptors and selectors. We'll cover Flume in more detail later on in this chapter, but we'll briefly cover the roles of interceptors and selectors here:

Interceptor

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. Note that special care must be taken in its implementation 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, there is a limitation on processing power because normally Flume is only installed on a sub-set of nodes in a cluster.

Selector

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.

Network Bottleneck

When 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 then it will be important to either add more network bandwidth or compress the data over the wire. This can be done 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.

Network Security

Sometimes there is a need to ingest data from sources that are only accessible 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. This can be done by simply encrypting the files before sending them or using a tool like Flume and applying encryption between agents.

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 use two common Hadoop tools covered in this chapter, Sqoop and Flume, to help clafify the distinction between push and pull in the context of data ingestion and extraction with Hadoop.

Sqoop

Sqoop is a pull solution which 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 data (pulling) 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.

Build to Handle Failure

Failure handling is a major consideration when 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, e.g. 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 3 times and has 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 file system directories that represent different bucketing in the life cycle of the file transfer process. Here is an example using this approach:

In this example there are four local file system 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 then move the file into the Failures directory
5. If the put was successful move the file into the Successful directory

It is important to 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 at 5 hours into a file transfer then the whole process has to start again.

File transfers are pretty simple, so lets 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 re-send 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 then try to commit them to the final destination, but if that commit fails then the batch of events need to be returned back to the channel.

The big difference between Flume and file transfers is that with Flume, in the case of a failure there is the chance for 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 duplicates 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.

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 a simple `hadoop fs -put` or using a mountable HDFS solution like FUSE or the new NFS gateway might provide a solution. We'll discuss these options below.

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. This ranges from simply loading files into HDFS, to 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 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 these reasons this should be considered as the first option when 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 parallize file transfers.
- File transfers are from a traditional file system 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 file system 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 using 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 file system isn't available to be mounted from the Hadoop cluster.

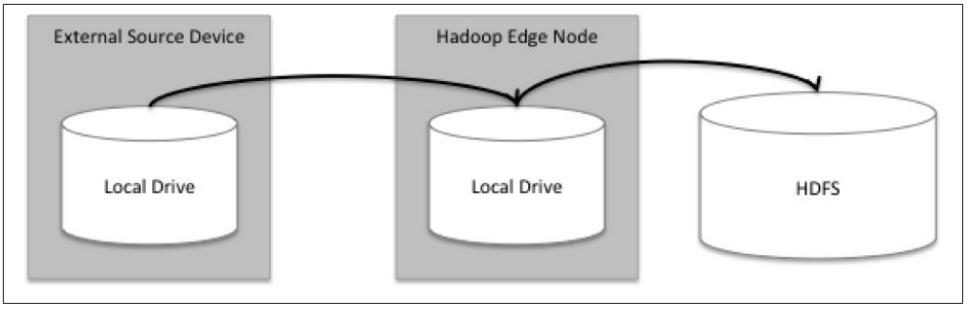


Figure 2-2. An example of double hop file uploads

The alternative is the single hop approach, 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.

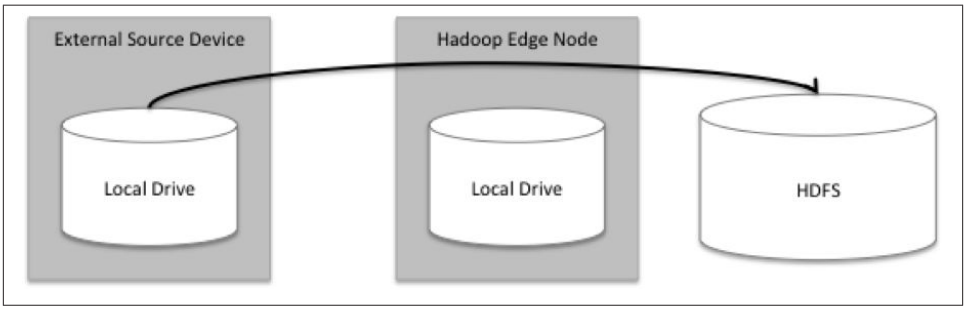


Figure 2-3. An example of single hop file uploads

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 file system. These options allow users to interact with HDFS using standard file system 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.
- Random writes are not supported; the underlying file system is still "write once, read many".

Another pitfall to mention with these solutions is the potential for misuse. Although it makes it easier for users to access HDFS and ingest files, this ease of access may en-

courage the ingestion of many small files. Since Hadoop is tuned for a relatively small number of large files, this scenario should be guarded against. It should be noted that "many 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 Chapter five.
- Use HBase to store the small files, using the path and file name as the key. We'll also discuss HBase in more detail in Chapter five.
- Use a container format such as SequenceFile to consolidate small files, as discussed in Chapter one.

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. The following provide some more details on these solutions:

Fuse-DFS

Fuse-DFS is built on the Filesystem in Userspace (FUSE) project, which was created to facilitate the creation of file systems on Unix/Linux systems without needing to modify the kernel. Fuse-DFS makes it easy to mount HDFS on a local file system, but as a user space module there are 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 it should be carefully considered before deploying as a production solution.

NFSv3

A more recent project adds support for the NFSv3 protocol to HDFS. This provides a scalable solution with a mimimal performance hit. The design involves an NFS gateway server that streams files to HDFS using the DFSClient.

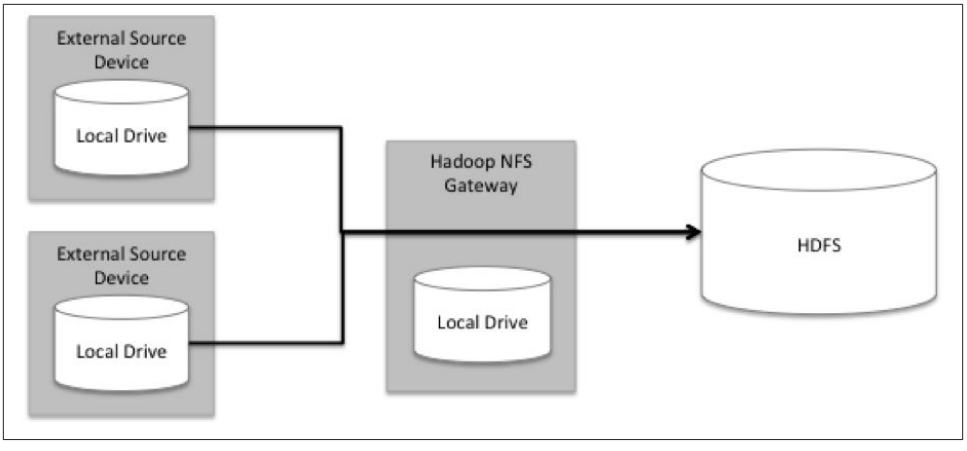


Figure 2-4. The HDFS NFSv3 Gateway

This solution is scalable with the addition of additional NFS gateway nodes. However the gateway nodes do not share a global cache, so understand that there are times where buffered and cache data may result in different results being returned, but there should be total consistency between NFS gateways for files that have been closed on HDFS.

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

Considerations for File Transfers vs. Other Ingest Methods

Simple file transfers are suitable in some cases, particularly when you have an existing set of files that need to be ingested into HDFS and keeping them in their source format is acceptable. Otherwise, the following are some considerations to take into account when trying to determine an appropriate ingestion method:

- 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 ingesting the files, so a tool like Flume is likely more suitable.
- Is reliability important? If so then consider using a tool like Flume, which provides durability of data in case of failure. Otherwise, with the file transfer methods an error mid-transfer will require a restart of the process.

- Is transformation of the data required before ingestion? In that case you'll definitely want to use a tool like Flume.

One option to consider if you have files to ingest is using the Flume spooling directory source[2]. This allows you 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 providing the option to perform transformations of the data in-flight if required.

Sqoop

Sqoop is a tool used to import data in bulk from a relational database management system (RDBMS) 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 JDBC driver, selects a portion of the table to be imported, and writes the data to HDFS. Sqoop is quite flexible and allows importing not just full tables but also add `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 the previous 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

2. *https://flume.apache.org/FlumeUserGuide.html#spooling-directory-source*

lets you control the number of mappers. As we note below, 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 by specifying 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.

Use 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 execite data import, while the Oracle connector will disable parallel queries to avoid bottlenecks on the query coordinator.

Goldilocks Method of Sqoop Performance Tuning[3]

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".

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[4]

There is a very common use case of having to ingest many tables from the same RDBMS. There are two different approaches to implementing this:

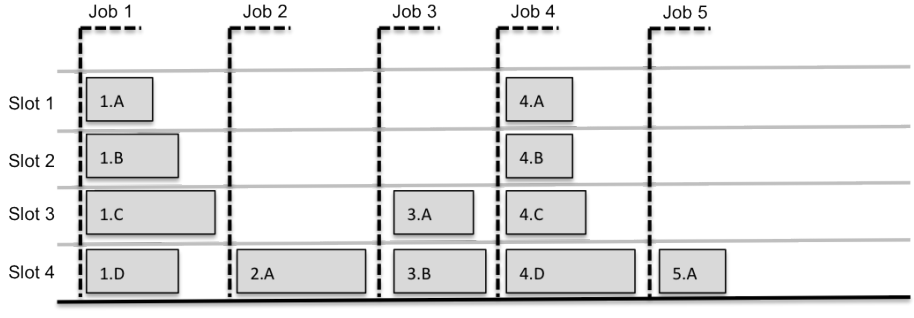
1. Load the Tables Sequentially One After the Other

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, lets imagine that we have

3. ***http://en.wikipedia.org/wiki/Goldilocks_principle — The*** **Goldilocks principle states that something must fall within certain margins, as opposed to reaching extremes.**

4. **Fair Schedulers are a method of sharing resources between jobs that allocates, 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, you can read Chapter seven, "Resource Management" in "Hadoop Operations" by Eric Sammer.**

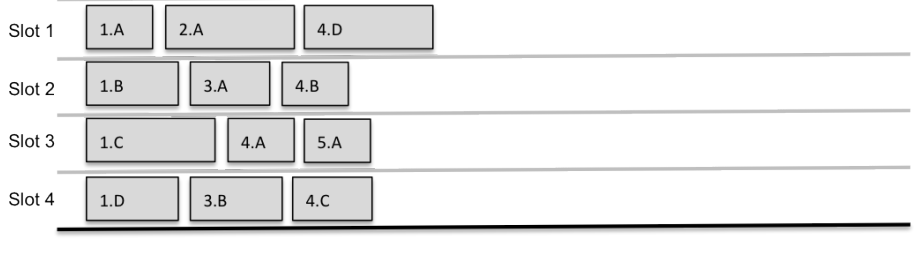
five tables. Each table is ingested using a different number of mappers since not all table sizes require allocating the full number of allowed mappers. The result of this example will look like this.



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

2. Load the tables in parallel

Running Sqoop jobs in parallel will allow you to use resources more effectively, but they come with the complexity of managing the over all total number of mappers being run against the RDBMS at the same time. The problem of managing the total number of mappers can be solved by using the FairScheduler 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 5 Jobs illustrated in the synchronized solution will look like the following:



Diagnozing 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 Hadoop cluster and the RDBMS is likely to be either 1GbE or 10GbE. This means that your ingest rate will be limited to around 120MB/s or 1.2GB/s 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 IO resources on the database server. If either of these resources are not available, adding more mappers is likely to make matters worse. If the query generated by the mappers is performing index reads instead of full table scans, or if multiple readers and writers are competing on access to the same data blocks, you will also see lower throughput. It is best to discuss Sqoop with your enterprise DBAs and have them monitor the database while importing data to Hadoop. Of course, it is best 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 to 250 and the second will import customers 251 to 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*: Since 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 IO, 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 file system 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 dataset 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`, it is possible to re-use 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 recom-

mend having Sqoop load the incremental update into a new directory. Once Sqoop finished loading the data (as will be indicated by the existance of a _SUCCESS file), the data can be cleaned and preprocessed before copying it 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 dataset 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 datasets 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 dataset 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. We will show how to do this in <insert appropriate case study name>.

Flume

Flume Overview

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 provide consideratons and recommendations for it's use. For more details on Flume see (insert reference to Flume book(s)).

Figure 2-5 shows the main components of Flume:

- External sources are any external system that generates events, this could be web servers, a social media feed such as Twitter, Java Messaging Service, machine logs, and so forth.
- Flume sources consume events from external sources and forward to channels. Flume sources are implemented to consume events from specific external sources, and many sources come bundled with Flume including AvroSource, SpoolDirectorySource, HTTPSource, JMSSource, and so on.
- Flume interceptors allow events to be intercepted and modified in flight. This can be transforming the event, enriching the event, etc. — basically anything that can be implmented 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. This can be used to send events down zero or more paths, so 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. Common channels include the memory channel, which stores events in memory. 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 it's 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, etc.

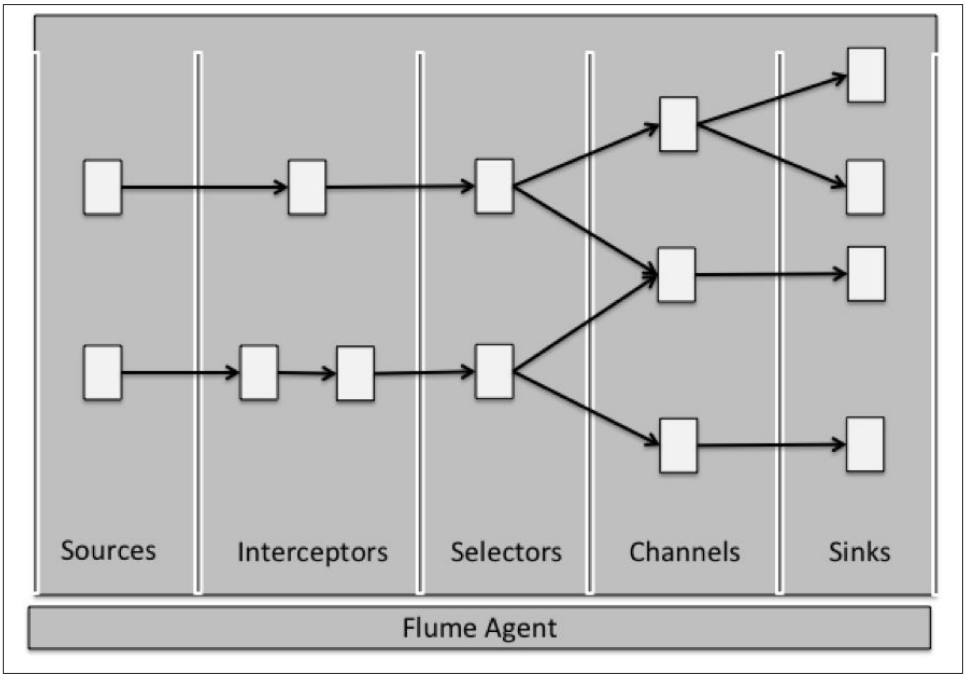


Figure 2-5. Flume components

Flume provides the following features:

- Reliability - Events are stored in channel until delivered to next stage.
- Recoverable - Events can be persisted to disk and recovered in the event of failure.
- Declarative - No coding required. Configuration specifies how components are wired together.

- Highly customizable - Although Flume includes a number of sources, sinks, etc. out of the box, Flume 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 of course 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 tradeoffs. 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 2-6 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 provide reliability: if one edge node goes down you don't lose events. It's also recommended to compress events before sending to reduce network traffic. SSL can also be used to encrypt data on the line if security is a concern.

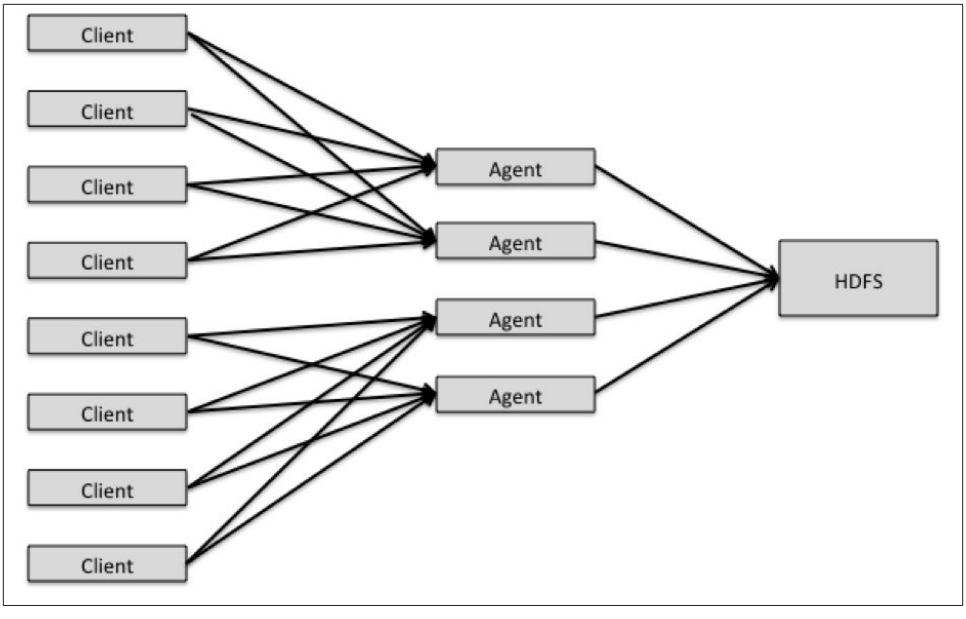


Figure 2-6. Flume fan-in architecture

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 the following diagram, 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 a 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 really requires a second Hadoop cluster.

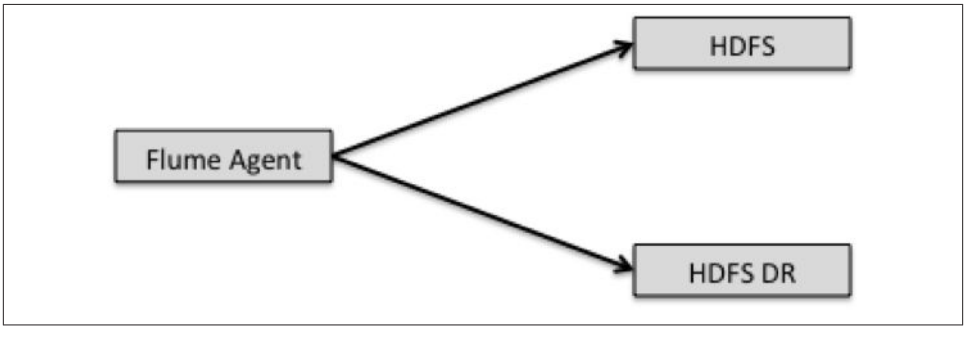


Figure 2-7. Flume event splitting architecture

Partitioning Data on Ingest

Flume can also be used to do partitioning of data as it's ingested. For example, the HDFS sink can partition events by timestamp, etc.

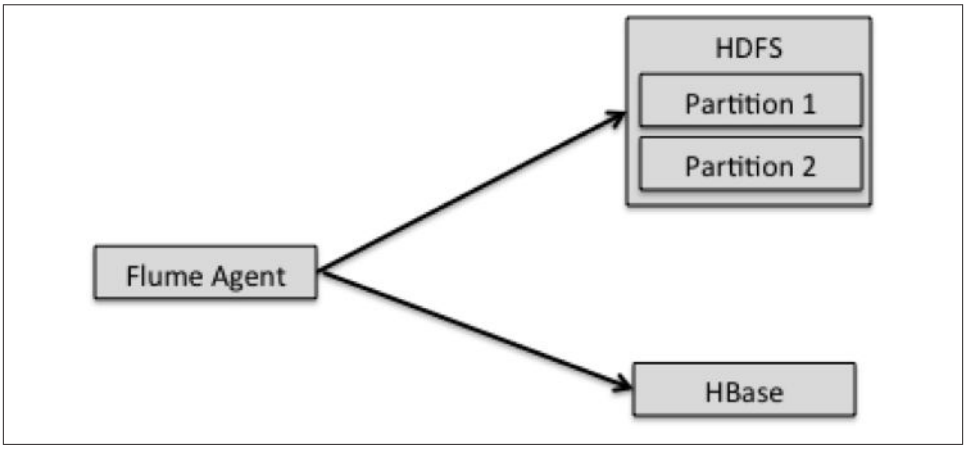


Figure 2-8. Flume partitioning architecture

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 that has to be done is to point a Flume Avro sink to Spark Streaming's Flume Stream.

File Formats

The following are some file format considerations when ingesting data into HDFS with Flume. We talk more about storage considerations in chapter one.

- 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 one. In general, when 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 click stream processing case study. Both Sequence and Avro files provide efficient compression while also being splittable. Avro is preferred, since 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 used with Flume 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; event serializers convert a Flume event into another format for output. In the case of text or sequence files, writing events to HDFS is straightforward since they're well supported by the Flume HDFS sink. As noted above, 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 have 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 things to think about 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 acknowledgement of success. This seems simple enough, but network latency can have a large impact here — as diagram x 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 1000, the time

it would take is 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 1000 events in a batch and adjust up or down from there based on performance.

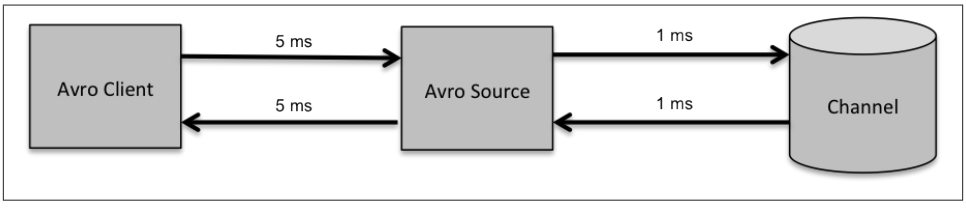


Figure 2-9. Effect of Latency on Flume Performance

- **Threads**: Understanding how Flume sources utilize threads will allow you to optimally take advantage of multi-threading with those sources. In general, more threads is good up to the point at which your network or CPUs are maxed out. Threading is different for different sources in Flume. Lets take the Avro source as an example; the Avro source is a netty server, and can have many connections to it at the same time, so to make the Avro source multi-threaded 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 from a pull source you need to configure more sources in your Flume agent.

Sinks

Remember, sinks are what gets your data to land in it's 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 30MB/s to a single disk; if you only have one sink writing to HDFS then all you're going to get is 30MB/s throughput with that sink. More sinks consuming from the same channel with 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 huge throughput value, and also remember for a flume sink to give guarantees 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, so if the sink is getting small batches then a lot

of time will be lost to syncing to disk instead of writing to disk. The only big down side to large batches with a sink is the risk of duplicate events.

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 MemoryChannels

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

Flume FileChannels

As noted before, since the file channel persists events to disk, it's more durable then MemoryChannel. It's important to note though that it's possible to make tradeoffs between durability and performance depending on how file channels are configured. The following provide some considerations and recommendations when using file channel:

- If you have multiple file channels on a single node, use distinct directories, and preferably separate disks, for each channel.
- Consider using RAID with striping for your data directories. Alternatively, if you have multiple disks, use a comma-separated list of locations for the data directory configuration. This will help protect against data loss in the event of a hard drive failure.
- In addition to the above, using an enterprise storage system such as network attached storage (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. This can be configured by setting `useDualCheckpoint` to `true`, and setting a directory value for `backupCheckpointDir`.

We should also note the much less commonly used JDBC channel, which persists events to any JDBC compliant datastore 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 MemoryChannel and FileChannel 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 FileChannel 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 MemoryChannel make it the better choice. We'll see an example of this when we discuss the click stream analysis case study.

Sizing Channels

When configuring a Flume agent there are often questions around whether to have more channels or less channels, 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 noe, the less memory 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 for the need to have more of the slow component to be 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 ether different, or it is 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, where 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, and not a storage location. It's main goal is to store data until it can be consumed by sinks. Normally, the buffer should only get big enough so that it's 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 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 exmaple 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 much so 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 things to review when 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 additional 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 then one source.
- Number of sinks: HDFS is made up of many drives. A single HDFSEventSink will only be writing to one spindle at a time. As we noted above, consider writing with more then one sink to increase performance.
- Channel: If using file channels understand the speed limitations. As we previously noted, a file channel can write to many drives which can help improve performance.
- Garbage Collection issues: When using the MemoryChannel it's possible to experience full GCs — this can happen when event objects live too long to make it out of Eden space.

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*. Kafka

keeps track of the messages as they are consumed and guarantees that messages will not get lost even if the consumer crashes, while allowing consumers to rewind the state of the queue and re-consume messages. The rewind feature is especially useful for debugging and troubleshooting.

Kafka can be used in place of a traditional message broker or message queue in an application architecture in order to decouple services. It was designed specifically for processing high rate activity streams such as clickstream or metrics collection. Kafka can run with a large number of message brokers, while recording its common state in ZooKeeper. As a distributed service, Kafka has significant scalability and reliability advantages over traditional queuing and messaging systems such as ActiveMQ.

Kafka is a generic message broker and was not written specifically for Hadoop. However, a common usage pattern is to store messages in HDFS for offline processing. For example, when tracking clickstream activity on a website Kafka can be used 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.

A common question is whether to use Kafka or Flume for ingest of log data or other streaming data sources into Hadoop. However, Kafka and Flume are really two separate solutions to two separate problems:

- Flume is an ingest solution - an easy and reliable way to store data streaming data in HDFS while solving common issues such as reliability, optimal file sizes, file formats, updating metadata and partitioning.
- Kafka, like all message brokers, is a solution to service decoupling and message queueing. It's a possible data source for Hadoop, but requires additional components in order to integrate with HDFS.

Due to the lack of integration between Kafka and Hadoop, we recommend deploying Kafka on a separate cluster than Hadoop, but in the same LAN. Kafka is a heavy disk I/O and memory user, and it can lead to resource contention with Hadoop tasks. At the same time it doesn't benefit from being co-located on DataNodes since MapReduce tasks are not aware of data locality in Kafka. It is possible and recommended to configure Kafka to use the Zookeeper nodes on the Hadoop cluster, so there will be no need to monitor and maintain two zookeeper clusters.

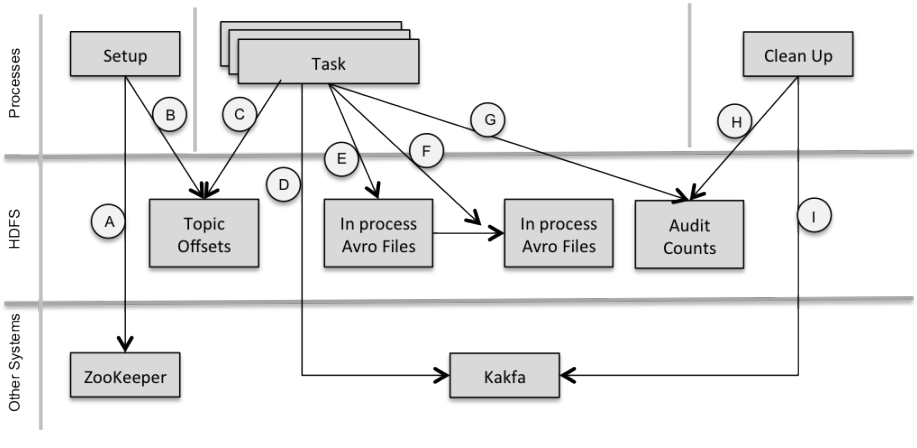
This leaves us with the question: How do we ingest data from Kafka to HDFS?

Kafka ships with a MapReduce job called Hadoop-Consumer that reads data from Kafka and writes it to HDFS. However this component is deprecated and not recommended for production use. It would make sense to use Flume to ingest messages from Kafka to Hadoop, as Flume was designed to ingest streaming messages. Unfortunately, at the

time this was written, Kafka source for Flume is not fully mature and not well maintained by the Flume or Kafka communities.

Instead we recommend using Camus[5], a separate open source project which allows 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.

Below is a high level diagram to help give an idea of how Camus works:



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 Kakfa.
- E - The tasks write data to a temp location in HDFS in the format defined by the user defined decoder, in this case Avro formatted files.

5. *https://github.com/linkedin/camus*

- F - The tasks move the data in the temp 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 Kakfa what has been persisted.

To use Camus, you may need to write your own decoder to convert Kafka messages to Avro. This is simalar 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 more time is generally spent 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 then using a simple file extract from Hadoop and then using a vendor specific ingest tool is an option. When using Sqoop it's important to use database-specific connectors when available. Regardless of the method used, it's very important to avoid overloading the target database — data volumes that are easily managed in Hadoop may easily overwhelm a traditional database. Do yourself and your database administrator a favor and ensure that you carefully consider load on the target system.

Another thing that should be noted 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` leverages MapReduce in order to perform parallel transfers of large volumes of data. It should be noted that `DistCp` is also suitable when either the source or target are a non-HDFS file system, for example an increasingly common need is to move data into a cloud-based system such as Amazon's Simple Storage System(S3).

Summary

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 things to consider 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.