Chapter 5. Hadoop I/O

Hadoop comes with a set of primitives for data I/O. Some of these are techniques that are more general than Hadoop, such as data integrity and compression, but deserve special consideration when dealing with multitterabyte datasets. Others are Hadoop tools or APIs that form the building blocks for developing distributed systems, such as serialization frameworks and on-disk data structures.

Data Integrity

Users of Hadoop rightly expect that no data will be lost or corrupted during storage or processing. However, because every I/O operation on the disk or network carries with it a small chance of introducing errors into the data that it is reading or writing, when the volumes of data flowing through the system are as large as the ones Hadoop is capable of handling, the chance of data corruption occurring is high.

The usual way of detecting corrupted data is by computing a *checksum* for the data when it first enters the system, and again whenever it is transmitted across a channel that is unreliable and hence capable of corrupting the data. The data is deemed to be corrupt if the newly generated checksum doesn't exactly match the original. This technique doesn't offer any way to fix the data—it is merely error detection. (And this is a reason for not using low-end hardware; in particular, be sure to use ECC memory.) Note that it is possible that it's the checksum that is corrupt, not the data, but this is very unlikely, because the checksum is much smaller than the data.

A commonly used error-detecting code is CRC-32 (32-bit cyclic redundancy check), which computes a 32-bit integer checksum for input of any size. CRC-32 is used for checksumming in Hadoop's `ChecksumFileSystem`, while HDFS uses a more efficient variant called CRC-32C.

Data Integrity in HDFS

HDFS transparently checksums all data written to it and by default verifies checksums when reading data. A separate checksum is created for every `dfs.bytes-per-checksum` bytes of data. The default is 512 bytes, and because a CRC-32C checksum is 4 bytes long, the storage overhead is less than 1%.

Datanodes are responsible for verifying the data they receive before storing the data and its checksum. This applies to data that they receive from clients and from other datanodes during replication. A client writing data sends it to a pipeline of datanodes (as explained in **Chapter 3**), and the last datanode in the pipeline verifies the checksum. If the datanode detects an error, the client receives a subclass of `IOException`, which it should handle in an application-specific manner (for example, by retrying the operation).

When clients read data from datanodes, they verify checksums as well, comparing them with the ones stored at the datanodes. Each datanode keeps a persistent log of checksum verifications, so it knows the last time each of its blocks was verified. When a client successfully verifies a block, it tells the datanode, which updates its log. Keeping statistics such as these is valuable in detecting bad disks.

In addition to block verification on client reads, each datanode runs a `DataBlockScanner` in a background thread that periodically verifies all the blocks stored on the datanode. This is to guard against corruption due to "bit rot" in the physical storage media. See **Datanode block scanner** for details on how to access the scanner reports.

Because HDFS stores replicas of blocks, it can "heal" corrupted blocks by copying one of the good replicas to produce a new, uncorrupt replica. The way this works is that if a client detects an error when reading a block, it reports the bad block and the datanode it was trying to read from to the namenode before throwing a `ChecksumException`. The namenode marks the block replica as corrupt so it doesn't direct any more clients to it or try to copy this replica to another datanode. It then schedules a copy of the block to be replicated on another datanode, so its replication factor is

back at the expected level. Once this has happened, the corrupt replica is deleted.

It is possible to disable verification of checksums by passing `false` to the `setVerifyChecksum()` method on `FileSystem` before using the `open()` method to read a file. The same effect is possible from the shell by using the `-ignoreCrc` option with the `-get` or the equivalent `-copyToLocal` command. This feature is useful if you have a corrupt file that you want to inspect so you can decide what to do with it. For example, you might want to see whether it can be salvaged before you delete it.

You can find a file's checksum with `hadoop fs -checksum`. This is useful to check whether two files in HDFS have the same contents—something that *distcp* does, for example (see **Parallel Copying with distcp**).

LocalFileSystem

The Hadoop `LocalFileSystem` performs client-side checksumming. This means that when you write a file called *filename*, the filesystem client transparently creates a hidden file, *.filename.crc*, in the same directory containing the checksums for each chunk of the file. The chunk size is controlled by the `file.bytes-per-checksum` property, which defaults to 512 bytes. The chunk size is stored as metadata in the *.crc* file, so the file can be read back correctly even if the setting for the chunk size has changed. Checksums are verified when the file is read, and if an error is detected, `LocalFileSystem` throws a `ChecksumException`.

Checksums are fairly cheap to compute (in Java, they are implemented in native code), typically adding a few percent overhead to the time to read or write a file. For most applications, this is an acceptable price to pay for data integrity. It is, however, possible to disable checksums, which is typically done when the underlying filesystem supports checksums natively. This is accomplished by using `RawLocalFileSystem` in place of `LocalFile System`. To do this globally in an application, it suffices to remap the implementation for `file` URIs by setting the property `fs.file.impl` to the value `org. apache. hadoop.fs.RawLocalFileSystem`. Alternatively, you can directly create a `RawLocal FileSystem` instance, which may be useful if you want to disable checksum verification for only some reads, for example:

```
Configuration conf = ...
FileSystem fs = new RawLocalFileSystem();
fs.initialize(null, conf);
```

ChecksumFileSystem

`LocalFileSystem` uses `ChecksumFileSystem` to do its work, and this class makes it easy to add checksumming to other (noncheck summed) filesystems, as `ChecksumFile System` is just a wrapper around `FileSystem`. The general idiom is as follows:

```
FileSystem rawFs = ...
FileSystem checksummedFs = new ChecksumFileSystem(rawFs);
```

The underlying filesystem is called the *raw* filesystem, and may be retrieved using the `getRawFileSystem()` method on `ChecksumFileSystem`. `ChecksumFileSystem` has a few more useful methods for working with checksums, such as `getChecksumFile()` for getting the path of a checksum file for any file. Check the documentation for the others.

If an error is detected by `ChecksumFileSystem` when reading a file, it will call its `reportChecksumFailure()` method. The default implementation does nothing, but `LocalFileSystem` moves the offending file and its checksum to a side directory on the same device called *bad_files*. Administrators should periodically check for these bad files and take action on them.

Compression

File compression brings two major benefits: it reduces the space needed to store files, and it speeds up data transfer across the network or to or from disk. When dealing with large volumes of data, both of these savings can be significant, so it pays to carefully consider how to use compression in Hadoop.

There are many different compression formats, tools, and algorithms, each with different characteristics. **Table 5-1** lists some of the more common ones that can be used with Hadoop.

Table 5-1. A summary of compression formats

Compression format	Tool	Algorithm	Filename extension	Splittable?
DEFLATE[a]	N/A	DEFLATE	*.deflate*	No
gzip	*gzip*	DEFLATE	*.gz*	No
bzip2	*bzip2*	bzip2	*.bz2*	Yes
LZO	*lzop*	LZO	*.lzo*	No[b]
LZ4	N/A	LZ4	*.lz4*	No
Snappy	N/A	Snappy	*.snappy*	No

[a] DEFLATE is a compression algorithm whose standard implementation is zlib. There is no commonly available command-line tool for producing files in DEFLATE format, as gzip is normally used. (Note that the gzip file format is DEFLATE with extra headers and a footer.) The *.deflate* filename extension is a Hadoop convention.

[b] However, LZO files are splittable if they have been indexed in a preprocessing step. See **Compression and Input Splits**.

All compression algorithms exhibit a space/time trade-off: faster compression and decompression speeds usually come at the expense of smaller space savings. The tools listed in **Table 5-1** typically give some control over this trade-off at compression time by offering nine different options: `-1` means optimize for speed, and `-9` means optimize for space. For example, the following command creates a compressed file *file.gz* using the fastest compression method:

```
% gzip -1 file
```

The different tools have very different compression characteristics. gzip is a general-purpose compressor and sits in the middle of the space/time trade-off. bzip2 compresses more effectively than gzip, but is slower. bzip2's decompression speed is faster than its compression speed, but it is still slower than the other formats. LZO, LZ4, and Snappy, on the other hand, all optimize for speed and are around an order of magnitude faster than gzip, but compress less effectively. Snappy and LZ4 are also significantly faster than LZO for decompression.[44]

The "Splittable" column in **Table 5-1** indicates whether the compression format supports splitting (that is, whether you can seek to any point in the stream and start reading from some point further on). Splittable compression formats are especially suitable for MapReduce; see **Compression and Input Splits** for further discussion.

Codecs

A *codec* is the implementation of a compression-decompression algorithm. In Hadoop, a codec is represented by an implementation of the `CompressionCodec` interface. So, for example, `GzipCodec` encapsulates the compression and decompression algorithm for gzip. **Table 5-2** lists the codecs that are available for Hadoop.

Table 5-2. Hadoop compression codecs

Compression format	Hadoop CompressionCodec
DEFLATE	`org.apache.hadoop.io.compress.DefaultCodec`
gzip	`org.apache.hadoop.io.compress.GzipCodec`
bzip2	`org.apache.hadoop.io.compress.BZip2Codec`
LZO	`com.hadoop.compression.lzo.LzopCodec`
LZ4	`org.apache.hadoop.io.compress.Lz4Codec`
Snappy	`org.apache.hadoop.io.compress.SnappyCodec`

The LZO libraries are GPL licensed and may not be included in Apache distributions, so for this reason the Hadoop codecs must be downloaded separately from **Google** (or **GitHub**, which includes bug fixes and more tools). The `LzopCodec` , which is compatible with the *lzop* tool, is essentially the LZO format with extra headers, and is the one you normally want. There is also an `LzoCodec` for the pure LZO format, which uses the *.lzo_deflate* filename extension (by analogy with DEFLATE, which is gzip without the headers).

Compressing and decompressing streams with CompressionCodec

`CompressionCodec` has two methods that allow you to easily compress or decompress data. To compress data being written to an output stream, use the `createOutputStream(OutputStream out)` method to create a `CompressionOutputStream` to which you write your uncompressed data to have it written in compressed form to the underlying stream.

Conversely, to decompress data being read from an input stream, call `createInputStream(InputStream in)` to obtain a `CompressionInputStream`, which allows you to read uncompressed data from the underlying stream.

`CompressionOutputStream` and `CompressionInputStream` are similar to `java.util. zip.DeflaterOutputStream` and `java.util.zip.DeflaterInputStream`, except that both of the former provide the ability to reset their underlying compressor or decompressor. This is important for applications that compress sections of the data stream as separate blocks, such as in a `SequenceFile`, described in **SequenceFile**.

Example 5-1 illustrates how to use the API to compress data read from standard input and write it to standard output.

Example 5-1. A program to compress data read from standard input and write it to standard output

```
public class StreamCompressor {

  public static void main(String[] args) throws Exception {
    String codecClassname = args[0];
    Class<?> codecClass = Class.forName(codecClassname);
    Configuration conf = new Configuration();
    CompressionCodec codec = (CompressionCodec)
      ReflectionUtils.newInstance(codecClass, conf);

    CompressionOutputStream out = codec.createOutputStream(System.out);
    IOUtils.copyBytes(System.in, out, 4096, false);
    out.finish();
  }
}
```

The application expects the fully qualified name of the `CompressionCodec` implementation as the first command-line argument. We use `ReflectionUtils` to construct a new instance of the codec, then obtain a compression wrapper around `System.out`. Then we call the utility method `copyBytes()` on `IOUtils` to copy the input to the output, which is compressed by the `CompressionOutputStream`. Finally, we call

`finish()` on `CompressionOutputStream`, which tells the compressor to finish writing to the compressed stream, but doesn't close the stream. We can try it out with the following command line, which compresses the string "Text" using the `StreamCompressor` program with the `GzipCodec`, then decompresses it from standard input using *gunzip*:

```
% echo "Text" | hadoop StreamCompressor org.apache.hadoop.io.compress.GzipCodec
  | gunzip -
Text
```

Inferring CompressionCodecs using CompressionCodecFactory

If you are reading a compressed file, normally you can infer which codec to use by looking at its filename extension. A file ending in *.gz* can be read with `GzipCodec`, and so on. The extensions for each compression format are listed in **Table 5-1**.

`CompressionCodecFactory` provides a way of mapping a filename extension to a `CompressionCodec` using its `getCodec()` method, which takes a `Path` object for the file in question. **Example 5-2** shows an application that uses this feature to decompress files.

Example 5-2. A program to decompress a compressed file using a codec inferred from the file's extension

```
public class FileDecompressor {

  public static void main(String[] args) throws Exception {
    String uri = args[0];
    Configuration conf = new Configuration();
    FileSystem fs = FileSystem.get(URI.create(uri), conf);

    Path inputPath = new Path(uri);
    CompressionCodecFactory factory = new CompressionCodecFactory(conf);
    CompressionCodec codec = factory.getCodec(inputPath);
    if (codec == null) {
      System.err.println("No codec found for " + uri);
      System.exit(1);
    }

    String outputUri =
        CompressionCodecFactory.removeSuffix(uri, codec.getDefaultExtension());
```

```
      InputStream in = null;
      OutputStream out = null;
      try {
        in = codec.createInputStream(fs.open(inputPath));
        out = fs.create(new Path(outputUri));
        IOUtils.copyBytes(in, out, conf);
      } finally {
        IOUtils.closeStream(in);
        IOUtils.closeStream(out);
      }
    }
  }
```

Once the codec has been found, it is used to strip off the file suffix to form the output filename (via the `removeSuffix()` static method of `CompressionCodecFactory`). In this way, a file named *file.gz* is decompressed to *file* by invoking the program as follows:

```
% hadoop FileDecompressor file.gz
```

`CompressionCodecFactory` loads all the codecs in **Table 5-2**, except LZO, as well as any listed in the `io.compression.codecs` configuration property (**Table 5-3**). By default, the property is empty; you would need to alter it only if you have a custom codec that you wish to register (such as the externally hosted LZO codecs). Each codec knows its default filename extension, thus permitting `CompressionCodecFactory` to search through the registered codecs to find a match for the given extension (if any).

Table 5-3. Compression codec properties

Property name	Type	Default value	Description
`io.compression.codecs`	Comma-separated `Class` names		A list of additional `CompressionCodec` classes for compression/decompression

Native libraries

For performance, it is preferable to use a native library for compression and decompression. For example, in one test, using the native gzip libraries reduced decompression times by up to 50% and compression times by around 10% (compared to the built-in Java implementation). **Table 5-4** shows the availability of Java and native implementations for each compression format. All formats have native implementations, but not all have a Java implementation (LZO, for example).

Table 5-4. Compression library implementations

Compression format	Java implementation?	Native implementation?
DEFLATE	Yes	Yes
gzip	Yes	Yes
bzip2	Yes	Yes
LZO	No	Yes
LZ4	No	Yes
Snappy	No	Yes

The Apache Hadoop binary tarball comes with prebuilt native compression binaries for 64-bit Linux, called *libhadoop.so*. For other platforms, you will need to compile the libraries yourself, following the *BUILDING.txt* instructions at the top level of the source tree.

The native libraries are picked up using the Java system property `java.library.path`. The *hadoop* script in the *etc/hadoop* directory sets this property for you, but if you don't use this script, you will need to set the property in your application.

By default, Hadoop looks for native libraries for the platform it is running on, and loads them automatically if they are found. This means you don't

have to change any configuration settings to use the native libraries. In some circumstances, however, you may wish to disable use of native libraries, such as when you are debugging a compression-related problem. You can do this by setting the property `io.native.lib.available` to `false`, which ensures that the built-in Java equivalents will be used (if they are available).

CodecPool

If you are using a native library and you are doing a lot of compression or decompression in your application, consider using `CodecPool`, which allows you to reuse compressors and decompressors, thereby amortizing the cost of creating these objects.

The code in **Example 5-3** shows the API, although in this program, which creates only a single `Compressor`, there is really no need to use a pool.

Example 5-3. A program to compress data read from standard input and write it to standard output using a pooled compressor

```
public class PooledStreamCompressor {

  public static void main(String[] args) throws Exception {
    String codecClassname = args[0];
    Class<?> codecClass = Class.forName(codecClassname);
    Configuration conf = new Configuration();
    CompressionCodec codec = (CompressionCodec)
        ReflectionUtils.newInstance(codecClass, conf);
    Compressor compressor = null;
    try {
      compressor = CodecPool.getCompressor(codec);
      CompressionOutputStream out =
          codec.createOutputStream(System.out, compressor);
      IOUtils.copyBytes(System.in, out, 4096, false);
      out.finish();
    } finally {
      CodecPool.returnCompressor(compressor);
    }
  }
}
```

We retrieve a `Compressor` instance from the pool for a given `CompressionCodec`, which we use in the codec's overloaded `createOutputStream()` method. By using a `finally` block, we ensure that the compressor is returned to the pool even if there is an `IOException` while copying the bytes between the streams.

Compression and Input Splits

When considering how to compress data that will be processed by MapReduce, it is important to understand whether the compression format supports splitting. Consider an uncompressed file stored in HDFS whose size is 1 GB. With an HDFS block size of 128 MB, the file will be stored as eight blocks, and a MapReduce job using this file as input will create eight input splits, each processed independently as input to a separate map task.

Imagine now that the file is a gzip-compressed file whose compressed size is 1 GB. As before, HDFS will store the file as eight blocks. However, creating a split for each block won't work, because it is impossible to start reading at an arbitrary point in the gzip stream and therefore impossible for a map task to read its split independently of the others. The gzip format uses DEFLATE to store the compressed data, and DEFLATE stores data as a series of compressed blocks. The problem is that the start of each block is not distinguished in any way that would allow a reader positioned at an arbitrary point in the stream to advance to the beginning of the next block, thereby synchronizing itself with the stream. For this reason, gzip does not support splitting.

In this case, MapReduce will do the right thing and not try to split the gzipped file, since it knows that the input is gzip-compressed (by looking at the filename extension) and that gzip does not support splitting. This will work, but at the expense of locality: a single map will process the eight HDFS blocks, most of which will not be local to the map. Also, with fewer maps, the job is less granular and so may take longer to run.

If the file in our hypothetical example were an LZO file, we would have the same problem because the underlying compression format does not provide a way for a reader to synchronize itself with the stream. However, it is possible to preprocess LZO files using an indexer tool that comes with the Hadoop LZO libraries, which you can obtain from the

Google and GitHub sites listed in **Codecs**. The tool builds an index of split points, effectively making them splittable when the appropriate MapReduce input format is used.

A bzip2 file, on the other hand, does provide a synchronization marker between blocks (a 48-bit approximation of pi), so it does support splitting. (**Table 5-1** lists whether each compression format supports splitting.)

WHICH COMPRESSION FORMAT SHOULD I USE?

Hadoop applications process large datasets, so you should strive to take advantage of compression. Which compression format you use depends on such considerations as file size, format, and the tools you are using for processing. Here are some suggestions, arranged roughly in order of most to least effective:

- Use a container file format such as sequence files (see the section), Avro datafiles (see the section), ORCFiles (see the section), or Parquet files (see the section), all of which support both compression and splitting. A fast compressor such as LZO, LZ4, or Snappy is generally a good choice.
- Use a compression format that supports splitting, such as bzip2 (although bzip2 is fairly slow), or one that can be indexed to support splitting, such as LZO.
- Split the file into chunks in the application, and compress each chunk separately using any supported compression format (it doesn't matter whether it is splittable). In this case, you should choose the chunk size so that the compressed chunks are approximately the size of an HDFS block.
- Store the files uncompressed.

For large files, you should not use a compression format that does not support splitting on the whole file, because you lose locality and make MapReduce applications very inefficient.

Using Compression in MapReduce

As described in **Inferring CompressionCodecs using CompressionCodecFactory**, if your input files are compressed, they will be decompressed automatically as they are read by MapReduce, using the filename extension to determine which codec to use.

In order to compress the output of a MapReduce job, in the job configuration, set the `mapreduce.output.fileoutputformat.compress` property to

`true` and set the `mapreduce.output.fileoutputformat.compress.codec` property to the classname of the compression codec you want to use. Alternatively, you can use the static convenience methods on `FileOutputFormat` to set these properties, as shown in **Example 5-4**.

Example 5-4. Application to run the maximum temperature job producing compressed output

```
public class MaxTemperatureWithCompression {

  public static void main(String[] args) throws Exception {
    if (args.length != 2) {
      System.err.println("Usage: MaxTemperatureWithCompression <input path> " +
          "<output path>");
      System.exit(-1);
    }

    Job job = new Job();
    job.setJarByClass(MaxTemperature.class);

    FileInputFormat.addInputPath(job, new Path(args[0]));
    FileOutputFormat.setOutputPath(job, new Path(args[1]));

    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);

    FileOutputFormat.setCompressOutput(job, true);
    FileOutputFormat.setOutputCompressorClass(job, GzipCodec.class);

    job.setMapperClass(MaxTemperatureMapper.class);
    job.setCombinerClass(MaxTemperatureReducer.class);
    job.setReducerClass(MaxTemperatureReducer.class);

    System.exit(job.waitForCompletion(true) ? 0 : 1);
  }
}
```

We run the program over compressed input (which doesn't have to use the same compression format as the output, although it does in this example) as follows:

```
% hadoop MaxTemperatureWithCompression input/ncdc/sample.txt.gz output
```

Each part of the final output is compressed; in this case, there is a single part:

```
% gunzip -c output/part-r-00000.gz
1949    111
1950    22
```

If you are emitting sequence files for your output, you can set the `mapreduce.output.fileoutputformat.compress.type` property to control the type of compression to use. The default is `RECORD`, which compresses individual records. Changing this to `BLOCK`, which compresses groups of records, is recommended because it compresses better (see **The SequenceFile format**).

There is also a static convenience method on `SequenceFileOutputFormat` called `setOutputCompressionType()` to set this property.

The configuration properties to set compression for MapReduce job outputs are summarized in **Table 5-5**. If your MapReduce driver uses the `Tool` interface (described in **GenericOptionsParser, Tool, and ToolRunner**), you can pass any of these properties to the program on the command line, which may be more convenient than modifying your program to hardcode the compression properties.

Table 5-5. MapReduce compression properties

Property name	Type	Default value	Description
`mapreduce.output.fileoutputformat.compress`	`boolean`	`false`	Whether to compress outputs
`mapreduce.output.fileoutputformat.compress.codec`	`Class name`	`org.apache.hadoop.io.compress.DefaultCodec`	The compression codec to use for outputs
`mapreduce.output.fileoutputformat.compress.type`	`String`	`RECORD`	The type of compression to use for sequence file outputs: `NONE`, `RECORD`, or `BLOCK`

Compressing map output

Even if your MapReduce application reads and writes uncompressed data, it may benefit from compressing the intermediate output of the map phase. The map output is written to disk and transferred across the network to the reducer nodes, so by using a fast compressor such as LZO, LZ4, or Snappy, you can get performance gains simply because the volume of data to transfer is reduced. The configuration properties to enable compression for map outputs and to set the compression format are shown in **Table 5-6**.

Table 5-6. Map output compression properties

Property name	Type	Default value	Description
`mapreduce.map.output.compress`	`boolean`	`false`	Whether to compress map outputs
`mapreduce.map.output.compress.codec`	`Class`	`org.apache.hadoop.io.compress.DefaultCodec`	The compression codec to use for map outputs

Here are the lines to add to enable gzip map output compression in your job (using the new API):

```
Configuration conf = new Configuration();
conf.setBoolean(Job.MAP_OUTPUT_COMPRESS, true);
conf.setClass(Job.MAP_OUTPUT_COMPRESS_CODEC, GzipCodec.class,
    CompressionCodec.class);
Job job = new Job(conf);
```

In the old API (see **Appendix D**), there are convenience methods on the `JobConf` object for doing the same thing:

```
conf.setCompressMapOutput(true);
conf.setMapOutputCompressorClass(GzipCodec.class);
```

Serialization

Serialization is the process of turning structured objects into a byte stream for transmission over a network or for writing to persistent storage. *Deserialization* is the reverse process of turning a byte stream back into a series of structured objects.

Serialization is used in two quite distinct areas of distributed data processing: for interprocess communication and for persistent storage.

In Hadoop, interprocess communication between nodes in the system is implemented using *remote procedure calls* (RPCs). The RPC protocol uses serialization to render the message into a binary stream to be sent to the remote node, which then deserializes the binary stream into the original message. In general, it is desirable that an RPC serialization format is:

Compact
: A compact format makes the best use of network bandwidth, which is the most scarce resource in a data center.

Fast
: Interprocess communication forms the backbone for a distributed system, so it is essential that there is as little performance overhead as possible for the serialization and deserialization process.

Extensible
: Protocols change over time to meet new requirements, so it should be straightforward to evolve the protocol in a controlled manner for clients and servers. For example, it should be possible to add a new argument to a method call and have the new servers accept messages in the old format (without the new argument) from old clients.

Interoperable
: For some systems, it is desirable to be able to support clients that are written in different languages to the server, so the format needs to be designed to make this possible.

On the face of it, the data format chosen for persistent storage would have different requirements from a serialization framework. After all, the lifespan of an RPC is less than a second, whereas persistent data may be read years after it was written. But it turns out, the four desirable properties of an RPC's serialization format are also crucial for a persistent storage format. We want the storage format to be compact (to make efficient use of storage space), fast (so the overhead in reading or writing terabytes of data is minimal), extensible (so we can transparently read data written in an older format), and interoperable (so we can read or write persistent data using different languages).

Hadoop uses its own serialization format, Writables, which is certainly compact and fast, but not so easy to extend or use from languages other than Java. Because Writables are central to Hadoop (most MapReduce programs use them for their key and value types), we look at them in some depth in the next three sections, before looking at some of the other serialization frameworks supported in Hadoop. Avro (a serialization system that was designed to overcome some of the limitations of Writables) is covered in **Chapter 12**.

The Writable Interface

The `Writable` interface defines two methods—one for writing its state to a `DataOutput` binary stream and one for reading its state from a `DataInput` binary stream:

```
package org.apache.hadoop.io;

import java.io.DataOutput;
import java.io.DataInput;
import java.io.IOException;

public interface Writable {
  void write(DataOutput out) throws IOException;
  void readFields(DataInput in) throws IOException;
}
```

Let's look at a particular `Writable` to see what we can do with it. We will use `IntWritable`, a wrapper for a Java `int`. We can create one and set its value using the `set()` method:

```
IntWritable writable = new IntWritable();
writable.set(163);
```

Equivalently, we can use the constructor that takes the integer value:

```
IntWritable writable = new IntWritable(163);
```

To examine the serialized form of the `IntWritable`, we write a small helper method that wraps a `java.io.ByteArrayOutputStream` in a

`java.io.DataOutputStream` (an implementation of `java.io.DataOutput`) to capture the bytes in the serialized stream:

```
public static byte[] serialize(Writable writable) throws IOException {
  ByteArrayOutputStream out = new ByteArrayOutputStream();
  DataOutputStream dataOut = new DataOutputStream(out);
  writable.write(dataOut);
  dataOut.close();
  return out.toByteArray();
}
```

An integer is written using four bytes (as we see using JUnit 4 assertions):

```
  byte[] bytes = serialize(writable);
  assertThat(bytes.length, is(4));
```

The bytes are written in big-endian order (so the most significant byte is written to the stream first, which is dictated by the `java.io.DataOutput` interface), and we can see their hexadecimal representation by using a method on Hadoop's `StringUtils`:

```
  assertThat(StringUtils.byteToHexString(bytes), is("000000a3"));
```

Let's try deserialization. Again, we create a helper method to read a `Writable` object from a byte array:

```
public static byte[] deserialize(Writable writable, byte[] bytes)
    throws IOException {
  ByteArrayInputStream in = new ByteArrayInputStream(bytes);
  DataInputStream dataIn = new DataInputStream(in);
  writable.readFields(dataIn);
  dataIn.close();
  return bytes;
}
```

We construct a new, value-less `IntWritable`, and then call `deserialize()` to read from the output data that we just wrote. Then we check that its value, retrieved using the `get()` method, is the original value, 163:

```
        IntWritable newWritable = new IntWritable();
        deserialize(newWritable, bytes);
        assertThat(newWritable.get(), is(163));
```

WritableComparable and comparators

`IntWritable` implements the `WritableComparable` interface, which is just a subinterface of the `Writable` and `java.lang.Comparable` interfaces:

```
package org.apache.hadoop.io;

public interface WritableComparable<T> extends Writable, Comparable<T> {
}
```

Comparison of types is crucial for MapReduce, where there is a sorting phase during which keys are compared with one another. One optimization that Hadoop provides is the `RawComparator` extension of Java's `Comparator`:

```
package org.apache.hadoop.io;

import java.util.Comparator;

public interface RawComparator<T> extends Comparator<T> {

  public int compare(byte[] b1, int s1, int l1, byte[] b2, int s2, int l2);

}
```

This interface permits implementors to compare records read from a stream without deserializing them into objects, thereby avoiding any overhead of object creation. For example, the comparator for `IntWritable`s implements the raw `compare()` method by reading an integer from each of the byte arrays `b1` and `b2` and comparing them directly from the given start positions (`s1` and `s2`) and lengths (`l1` and `l2`).

`WritableComparator` is a general-purpose implementation of `RawComparator` for `WritableComparable` classes. It provides two main functions. First, it provides a default implementation of the raw `compare()` method that deserializes the objects to be compared from the stream and invokes the object `compare()` method. Second, it acts as a factory for `RawComparator` instances (that `Writable` implementations have registered). For example, to obtain a comparator for `IntWritable`, we just use:

```
RawComparator<IntWritable> comparator =
    WritableComparator.get(IntWritable.class);
```

The comparator can be used to compare two `IntWritable` objects:

```
IntWritable w1 = new IntWritable(163);
IntWritable w2 = new IntWritable(67);
assertThat(comparator.compare(w1, w2), greaterThan(0));
```

or their serialized representations:

```
byte[] b1 = serialize(w1);
byte[] b2 = serialize(w2);
assertThat(comparator.compare(b1, 0, b1.length, b2, 0, b2.length),
    greaterThan(0));
```

Writable Classes

Hadoop comes with a large selection of `Writable` classes, which are available in the `org.apache.hadoop.io` package. They form the class hierarchy shown in **Figure 5-1**.

Writable wrappers for Java primitives

There are `Writable` wrappers for all the Java primitive types (see **Table 5-7**) except `char` (which can be stored in an `IntWritable`). All have a `get()` and `set()` method for retrieving and storing the wrapped value.

Table 5-7. Writable wrapper classes for Java primitives

Java primitive	Writable implementation	Serialized size (bytes)
`boolean`	`BooleanWritable`	1
`byte`	`ByteWritable`	1
`short`	`ShortWritable`	2
`int`	`IntWritable`	4
	`VIntWritable`	1–5
`float`	`FloatWritable`	4
`long`	`LongWritable`	8
	`VLongWritable`	1–9
`double`	`DoubleWritable`	8

When it comes to encoding integers, there is a choice between the fixed-length formats (`IntWritable` and `LongWritable`) and the variable-length formats (`VIntWritable` and `VLongWritable`). The variable-length formats use only a single byte to encode the value if it is small enough (between –112 and 127, inclusive); otherwise, they use the first byte to indicate whether the value is positive or negative, and how many bytes follow. For example, 163 requires two bytes:

```
byte[] data = serialize(new VIntWritable(163));
assertThat(StringUtils.byteToHexString(data), is("8fa3"));
```

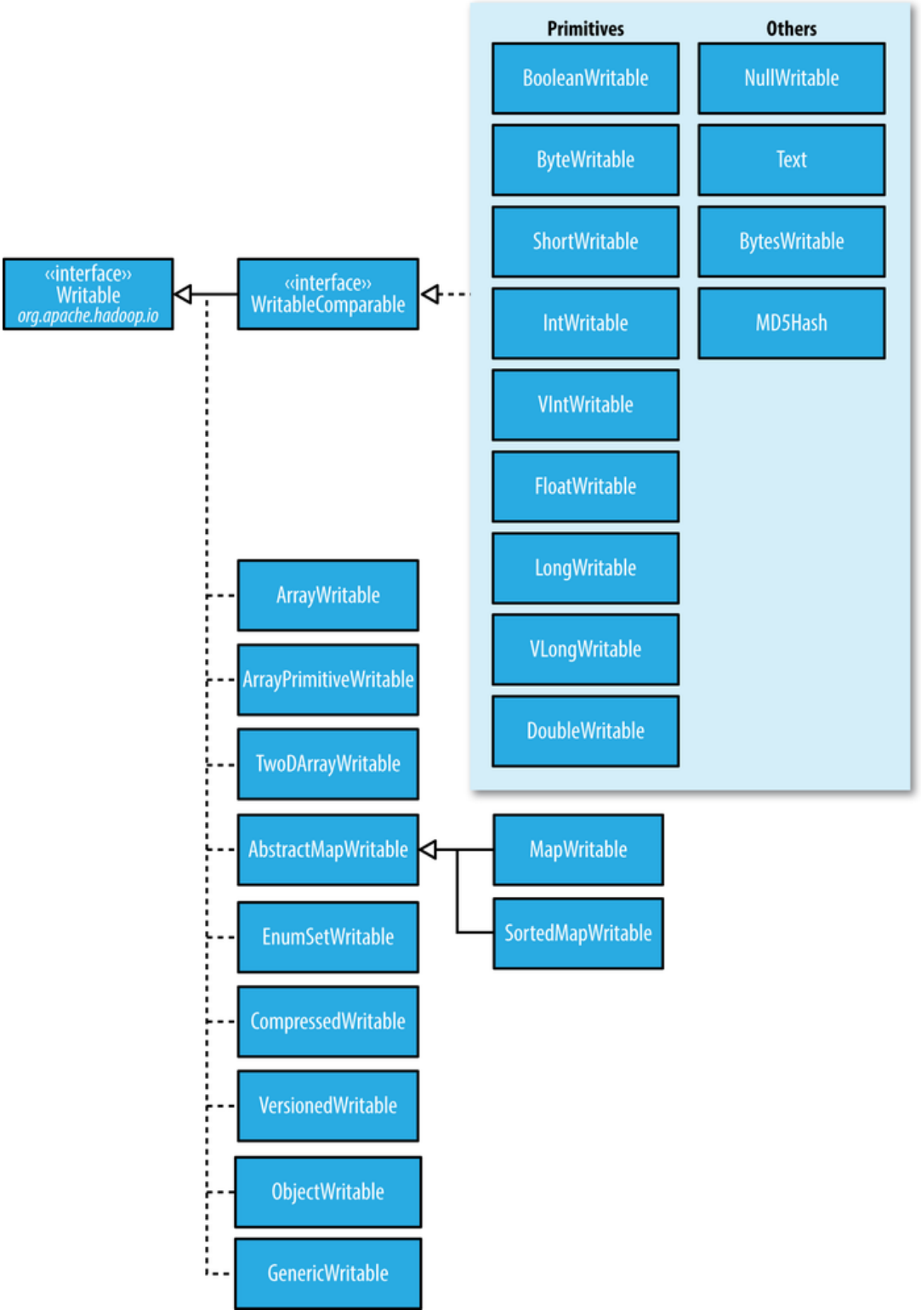

Figure 5-1. Writable class hierarchy

How do you choose between a fixed-length and a variable-length encoding? Fixed-length encodings are good when the distribution of values is fairly uniform across the whole value space, such as when using a (well-designed) hash function. Most numeric variables tend to have nonuniform distributions, though, and on average, the variable-length encoding will save space. Another advantage of variable-length encodings is that

you can switch from `VIntWritable` to `VLongWritable`, because their encodings are actually the same. So, by choosing a variable-length representation, you have room to grow without committing to an 8-byte `long` representation from the beginning.

Text

`Text` is a Writable for UTF-8 sequences. It can be thought of as the Writable equivalent of `java.lang.String`.

The `Text` class uses an `int` (with a variable-length encoding) to store the number of bytes in the string encoding, so the maximum value is 2 GB. Furthermore, `Text` uses standard UTF-8, which makes it potentially easier to interoperate with other tools that understand UTF-8.

Indexing

Because of its emphasis on using standard UTF-8, there are some differences between `Text` and the Java `String` class. Indexing for the `Text` class is in terms of position in the encoded byte sequence, not the Unicode character in the string or the Java `char` code unit (as it is for `String`). For ASCII strings, these three concepts of index position coincide. Here is an example to demonstrate the use of the `charAt()` method:

```
Text t = new Text("hadoop");
assertThat(t.getLength(), is(6));
assertThat(t.getBytes().length, is(6));

assertThat(t.charAt(2), is((int) 'D'));
assertThat("Out of bounds", t.charAt(100), is(-1));
```

Notice that `charAt()` returns an `int` representing a Unicode code point, unlike the `String` variant that returns a `char`. `Text` also has a `find()` method, which is analogous to `String`'s `indexOf()`:

```
Text t = new Text("hadoop");
assertThat("Find a substring", t.find("do"), is(2));
assertThat("Finds first 'o'", t.find("o"), is(3));
assertThat("Finds 'o' from position 4 or later", t.find("o", 4), is(4));
assertThat("No match", t.find("pig"), is(-1));
```

Unicode

When we start using characters that are encoded with more than a single byte, the differences between `Text` and `String` become clear. Consider the Unicode characters shown in **Table 5-8**.[45]

Table 5-8. Unicode characters

Unicode code point	U+0041	U+00DF	U+6771	U+10400
Name	LATIN CAPITAL LETTER A	LATIN SMALL LETTER SHARP S	N/A (a unified Han ideograph)	DESERET CAPITAL LETTER LONG I
UTF-8 code units	`41`	`c3 9f`	`e6 9d b1`	`f0 90 90 80`
Java representation	`\u0041`	`\u00DF`	`\u6771`	`\uD801\uDC00`

All but the last character in the table, U+10400, can be expressed using a single Java `char`. U+10400 is a supplementary character and is represented by two Java `char`s, known as a *surrogate pair*. The tests in **Example 5-5** show the differences between `String` and `Text` when processing a string of the four characters from **Table 5-8**.

Example 5-5. Tests showing the differences between the String and Text classes

```
public class StringTextComparisonTest {

  @Test
  public void string() throws UnsupportedEncodingException {

    String s = "\u0041\u00DF\u6771\uD801\uDC00";
    assertThat(s.length(), is(5));
    assertThat(s.getBytes("UTF-8").length, is(10));
```

```
        assertThat(s.indexOf("\u0041"), is(0));
        assertThat(s.indexOf("\u00DF"), is(1));
        assertThat(s.indexOf("\u6771"), is(2));
        assertThat(s.indexOf("\uD801\uDC00"), is(3));

        assertThat(s.charAt(0), is('\U0041'));
        assertThat(s.charAt(1), is('\U00DF'));
        assertThat(s.charAt(2), is('\U6771'));
        assertThat(s.charAt(3), is('\UD801'));
        assertThat(s.charAt(4), is('\UDC00'));

        assertThat(s.codePointAt(0), is(0x0041));
        assertThat(s.codePointAt(1), is(0x00DF));
        assertThat(s.codePointAt(2), is(0x6771));
        assertThat(s.codePointAt(3), is(0x10400));
    }

    @Test
    public void text() {

        Text t = new Text("\u0041\u00DF\u6771\uD801\uDC00");
        assertThat(t.getLength(), is(10));

        assertThat(t.find("\u0041"), is(0));
        assertThat(t.find("\u00DF"), is(1));
        assertThat(t.find("\u6771"), is(3));
        assertThat(t.find("\uD801\uDC00"), is(6));

        assertThat(t.charAt(0), is(0x0041));
        assertThat(t.charAt(1), is(0x00DF));
        assertThat(t.charAt(3), is(0x6771));
        assertThat(t.charAt(6), is(0x10400));
    }
}
```

The test confirms that the length of a `String` is the number of `char` code units it contains (five, made up of one from each of the first three characters in the string and a surrogate pair from the last), whereas the length of a `Text` object is the number of bytes in its UTF-8 encoding (10 = 1+2+3+4). Similarly, the `indexOf()` method in `String` returns an index in `char` code units, and `find()` for `Text` returns a byte offset.

The `charAt()` method in `String` returns the `char` code unit for the given index, which in the case of a surrogate pair will not represent a whole Unicode character. The `codePointAt()` method, indexed by `char` code unit, is needed to retrieve a single Unicode character represented as an `int`. In fact, the `charAt()` method in `Text` is more like the `codePointAt()` method than its namesake in `String`. The only difference is that it is indexed by byte offset.

Iteration

Iterating over the Unicode characters in `Text` is complicated by the use of byte offsets for indexing, since you can't just increment the index. The idiom for iteration is a little obscure (see **Example 5-6**): turn the `Text` object into a `java.nio.ByteBuffer`, then repeatedly call the `bytesToCodePoint()` static method on `Text` with the buffer. This method extracts the next code point as an `int` and updates the position in the buffer. The end of the string is detected when `bytesToCodePoint()` returns –1.

Example 5-6. Iterating over the characters in a Text object

```
public class TextIterator {

  public static void main(String[] args) {
    Text t = new Text("\u0041\u00DF\u6771\uD801\uDC00");

    ByteBuffer buf = ByteBuffer.wrap(t.getBytes(), 0, t.getLength());
    int cp;
    while (buf.hasRemaining() && (cp = Text.bytesToCodePoint(buf)) != -1) {
      System.out.println(Integer.toHexString(cp));
    }
  }
}
```

Running the program prints the code points for the four characters in the string:

```
% hadoop TextIterator
41
df
6771
10400
```

Mutability

Another difference from `String` is that `Text` is mutable (like all `Writable` implementations in Hadoop, except `NullWritable`, which is a singleton). You can reuse a `Text` instance by calling one of the `set()` methods on it. For example:

```
Text t = new Text("hadoop");
t.set("pig");
assertThat(t.getLength(), is(3));
assertThat(t.getBytes().length, is(3));
```

WARNING

In some situations, the byte array returned by the `getBytes()` method may be longer than the length returned by `getLength()`:

```
Text t = new Text("hadoop");
t.set(new Text("pig"));
assertThat(t.getLength(), is(3));
assertThat("Byte length not shortened", t.getBytes().length,
    is(6));
```

This shows why it is imperative that you always call `getLength()` when calling `getBytes()`, so you know how much of the byte array is valid data.

Resorting to String

`Text` doesn't have as rich an API for manipulating strings as `java.lang.String`, so in many cases, you need to convert the `Text` object to a `String`. This is done in the usual way, using the `toString()` method:

```
assertThat(new Text("hadoop").toString(), is("hadoop"));
```

BytesWritable

`BytesWritable` is a wrapper for an array of binary data. Its serialized format is a 4-byte integer field that specifies the number of bytes to follow, followed by the bytes themselves. For example, the byte array of

length 2 with values 3 and 5 is serialized as a 4-byte integer (`00000002`) followed by the two bytes from the array (`03` and `05`):

```
BytesWritable b = new BytesWritable(new byte[] { 3, 5 });
byte[] bytes = serialize(b);
assertThat(StringUtils.byteToHexString(bytes), is("000000020305"));
```

`BytesWritable` is mutable, and its value may be changed by calling its `set()` method. As with `Text`, the size of the byte array returned from the `getBytes()` method for `BytesWritable`—the capacity—may not reflect the actual size of the data stored in the `BytesWritable`. You can determine the size of the `BytesWritable` by calling `getLength()`. To demonstrate:

```
b.setCapacity(11);
assertThat(b.getLength(), is(2));
assertThat(b.getBytes().length, is(11));
```

NullWritable

`NullWritable` is a special type of Writable, as it has a zero-length serialization. No bytes are written to or read from the stream. It is used as a placeholder; for example, in MapReduce, a key or a value can be declared as a `NullWritable` when you don't need to use that position, effectively storing a constant empty value. `NullWritable` can also be useful as a key in a `SequenceFile` when you want to store a list of values, as opposed to key-value pairs. It is an immutable singleton, and the instance can be retrieved by calling `NullWritable.get()`.

ObjectWritable and GenericWritable

`ObjectWritable` is a general-purpose wrapper for the following: Java primitives, `String`, `enum`, `Writable`, `null`, or arrays of any of these types. It is used in Hadoop RPC to marshal and unmarshal method arguments and return types.

`ObjectWritable` is useful when a field can be of more than one type. For example, if the values in a `SequenceFile` have multiple types, you can declare the value type as an `ObjectWritable` and wrap each type in an `ObjectWritable`. Being a general-purpose mechanism, it wastes a fair

amount of space because it writes the classname of the wrapped type every time it is serialized. In cases where the number of types is small and known ahead of time, this can be improved by having a static array of types and using the index into the array as the serialized reference to the type. This is the approach that `GenericWritable` takes, and you have to subclass it to specify which types to support.

Writable collections

The `org.apache.hadoop.io` package includes six `Writable` collection types: `ArrayWritable`, `ArrayPrimitiveWritable`, `TwoDArrayWritable`, `MapWritable`, `SortedMap Writable`, and `EnumSetWritable`.

`ArrayWritable` and `TwoDArrayWritable` are `Writable` implementations for arrays and two-dimensional arrays (array of arrays) of `Writable` instances. All the elements of an `ArrayWritable` or a `TwoD ArrayWritable` must be instances of the same class, which is specified at construction as follows:

```
ArrayWritable writable = new ArrayWritable(Text.class);
```

In contexts where the `Writable` is defined by type, such as in `SequenceFile` keys or values or as input to MapReduce in general, you need to subclass `ArrayWritable` (or `TwoDArrayWritable`, as appropriate) to set the type statically. For example:

```
public class TextArrayWritable extends ArrayWritable {
  public TextArrayWritable() {
    super(Text.class);
  }
}
```

`ArrayWritable` and `TwoDArrayWritable` both have `get()` and `set()` methods, as well as a `toArray()` method, which creates a shallow copy of the array (or 2D array).

`ArrayPrimitiveWritable` is a wrapper for arrays of Java primitives. The component type is detected when you call `set()`, so there is no need to subclass to set the type.

`MapWritable` is an implementation of `java.util.Map<Writable, Writable>`, and `SortedMapWritable` is an implementation of `java.util.SortedMap<WritableComparable, Writable>`. The type of each key and value field is a part of the serialization format for that field. The type is stored as a single byte that acts as an index into an array of types. The array is populated with the standard types in the `org.apache.hadoop.io` package, but custom `Writable` types are accommodated, too, by writing a header that encodes the type array for nonstandard types. As they are implemented, `MapWritable` and `SortedMapWritable` use positive `byte` values for custom types, so a maximum of 127 distinct nonstandard `Writable` classes can be used in any particular `MapWritable` or `SortedMapWritable` instance. Here's a demonstration of using a `MapWritable` with different types for keys and values:

```
MapWritable src = new MapWritable();
src.put(new IntWritable(1), new Text("cat"));
src.put(new VIntWritable(2), new LongWritable(163));

MapWritable dest = new MapWritable();
WritableUtils.cloneInto(dest, src);
assertThat((Text) dest.get(new IntWritable(1)), is(new Text("cat")));
assertThat((LongWritable) dest.get(new VIntWritable(2)),
    is(new LongWritable(163)));
```

Conspicuous by their absence are `Writable` collection implementations for sets and lists. A general set can be emulated by using a `MapWritable` (or a `SortedMapWritable` for a sorted set) with `NullWritable` values. There is also `EnumSetWritable` for sets of enum types. For lists of a single type of `Writable`, `ArrayWritable` is adequate, but to store different types of `Writable` in a single list, you can use `GenericWritable` to wrap the elements in an `ArrayWritable`. Alternatively, you could write a general `ListWritable` using the ideas from `MapWritable`.

Implementing a Custom Writable

Hadoop comes with a useful set of `Writable` implementations that serve most purposes; however, on occasion, you may need to write your own custom implementation. With a custom `Writable`, you have full control over the binary representation and the sort order. Because `Writable`s

are at the heart of the MapReduce data path, tuning the binary representation can have a significant effect on performance. The stock `Writable` implementations that come with Hadoop are well tuned, but for more elaborate structures, it is often better to create a new `Writable` type rather than composing the stock types.

TIP

If you are considering writing a custom `Writable`, it may be worth trying another serialization framework, like Avro, that allows you to define custom types declaratively. See **Serialization Frameworks** and **Chapter 12**.

To demonstrate how to create a custom `Writable`, we shall write an implementation that represents a pair of strings, called `TextPair`. The basic implementation is shown in **Example 5-7**.

Example 5-7. A Writable implementation that stores a pair of Text objects

```
import java.io.*;

import org.apache.hadoop.io.*;

public class TextPair implements WritableComparable<TextPair> {

  private Text first;
  private Text second;

  public TextPair() {
    set(new Text(), new Text());
  }

  public TextPair(String first, String second) {
    set(new Text(first), new Text(second));
  }

  public TextPair(Text first, Text second) {
    set(first, second);
  }

  public void set(Text first, Text second) {
    this.first = first;
```

```
    this.second = second;
  }

  public Text getFirst() {
    return first;
  }

  public Text getSecond() {
    return second;
  }

  @Override
  public void write(DataOutput out) throws IOException {
    first.write(out);
    second.write(out);
  }

  @Override
  public void readFields(DataInput in) throws IOException {
    first.readFields(in);
    second.readFields(in);
  }

  @Override
  public int hashCode() {
    return first.hashCode() * 163 + second.hashCode();
  }

  @Override
  public boolean equals(Object o) {
    if (o instanceof TextPair) {
      TextPair tp = (TextPair) o;
      return first.equals(tp.first) && second.equals(tp.second);
    }
    return false;
  }

  @Override
  public String toString() {
    return first + "\t" + second;
  }

  @Override
  public int compareTo(TextPair tp) {
    int cmp = first.compareTo(tp.first);
```

```
      if (cmp != 0) {
        return cmp;
      }
      return second.compareTo(tp.second);
    }
  }
```

The first part of the implementation is straightforward: there are two `Text` instance variables, `first` and `second`, and associated constructors, getters, and setters. All `Writable` implementations must have a default constructor so that the MapReduce framework can instantiate them, then populate their fields by calling `readFields()`. `Writable` instances are mutable and often reused, so you should take care to avoid allocating objects in the `write()` or `readFields()` methods.

`TextPair`'s `write()` method serializes each `Text` object in turn to the output stream by delegating to the `Text` objects themselves. Similarly, `readFields()` deserializes the bytes from the input stream by delegating to each `Text` object. The `DataOutput` and `DataInput` interfaces have a rich set of methods for serializing and deserializing Java primitives, so, in general, you have complete control over the wire format of your `Writable` object.

Just as you would for any value object you write in Java, you should override the `hashCode()`, `equals()`, and `toString()` methods from `java.lang.Object`. The `hashCode()` method is used by the `HashPartitioner` (the default partitioner in MapReduce) to choose a reduce partition, so you should make sure that you write a good hash function that mixes well to ensure reduce partitions are of a similar size.

WARNING

If you plan to use your custom `Writable` with `TextOutputFormat`, you must implement its `toString()` method. `TextOutputFormat` calls `toString()` on keys and values for their output representation. For `TextPair`, we write the underlying `Text` objects as strings separated by a tab character.

`TextPair` is an implementation of `WritableComparable`, so it provides an implementation of the `compareTo()` method that imposes the order-

ing you would expect: it sorts by the first string followed by the second. Notice that, apart from the number of `Text` objects it can store, `TextPair` differs from `TextArrayWritable` (which we discussed in the previous section), since `TextArrayWritable` is only a `Writable`, not a `WritableComparable`.

Implementing a RawComparator for speed

The code for `TextPair` in **Example 5-7** will work as it stands; however, there is a further optimization we can make. As explained in **WritableComparable and comparators**, when `TextPair` is being used as a key in MapReduce, it will have to be deserialized into an object for the `compareTo()` method to be invoked. What if it were possible to compare two `TextPair` objects just by looking at their serialized representations?

It turns out that we can do this because `TextPair` is the concatenation of two `Text` objects, and the binary representation of a `Text` object is a variable-length integer containing the number of bytes in the UTF-8 representation of the string, followed by the UTF-8 bytes themselves. The trick is to read the initial length so we know how long the first `Text` object's byte representation is; then we can delegate to `Text`'s `RawComparator` and invoke it with the appropriate offsets for the first or second string. **Example 5-8** gives the details (note that this code is nested in the `TextPair` class).

Example 5-8. A RawComparator for comparing TextPair byte representations

```
public static class Comparator extends WritableComparator {

  private static final Text.Comparator TEXT_COMPARATOR = new Text.Comparator()

  public Comparator() {
    super(TextPair.class);
  }

  @Override
  public int compare(byte[] b1, int s1, int l1,
                     byte[] b2, int s2, int l2) {

    try {
```

```
            int firstL1 = WritableUtils.decodeVIntSize(b1[s1]) + readVInt(b1, s1);
            int firstL2 = WritableUtils.decodeVIntSize(b2[s2]) + readVInt(b2, s2);
            int cmp = TEXT_COMPARATOR.compare(b1, s1, firstL1, b2, s2, firstL2);
            if (cmp != 0) {
              return cmp;
            }
            return TEXT_COMPARATOR.compare(b1, s1 + firstL1, l1 - firstL1,
                                           b2, s2 + firstL2, l2 - firstL2);
          } catch (IOException e) {
            throw new IllegalArgumentException(e);
          }
        }
      }

      static {
        WritableComparator.define(TextPair.class, new Comparator());
      }
```

We actually subclass `WritableComparator` rather than implementing `RawComparator` directly, since it provides some convenience methods and default implementations. The subtle part of this code is calculating `firstL1` and `firstL2`, the lengths of the first `Text` field in each byte stream. Each is made up of the length of the variable-length integer (returned by `decodeVIntSize()` on `WritableUtils`) and the value it is encoding (returned by `readVInt()`).

The static block registers the raw comparator so that whenever MapReduce sees the `TextPair` class, it knows to use the raw comparator as its default comparator.

Custom comparators

As you can see with `TextPair`, writing raw comparators takes some care because you have to deal with details at the byte level. It is worth looking at some of the implementations of `Writable` in the `org.apache.hadoop.io` package for further ideas if you need to write your own. The utility methods on `WritableUtils` are very handy, too.

Custom comparators should also be written to be `RawComparator`s, if possible. These are comparators that implement a different sort order from the natural sort order defined by the default comparator. **Example 5-9** shows a comparator for `TextPair`, called `FirstComparator`, that consid-

ers only the first string of the pair. Note that we override the `compare()` method that takes objects so both `compare()` methods have the same semantics.

We will make use of this comparator in **Chapter 9**, when we look at joins and secondary sorting in MapReduce (see **Joins**).

Example 5-9. A custom RawComparator for comparing the first field of TextPair byte representations

```
public static class FirstComparator extends WritableComparator {

  private static final Text.Comparator TEXT_COMPARATOR = new Text.Comparator()

  public FirstComparator() {
    super(TextPair.class);
  }

  @Override
  public int compare(byte[] b1, int s1, int l1,
                     byte[] b2, int s2, int l2) {

    try {
      int firstL1 = WritableUtils.decodeVIntSize(b1[s1]) + readVInt(b1, s1);
      int firstL2 = WritableUtils.decodeVIntSize(b2[s2]) + readVInt(b2, s2);
      return TEXT_COMPARATOR.compare(b1, s1, firstL1, b2, s2, firstL2);
    } catch (IOException e) {
      throw new IllegalArgumentException(e);
    }
  }

  @Override
  public int compare(WritableComparable a, WritableComparable b) {
    if (a instanceof TextPair && b instanceof TextPair) {
      return ((TextPair) a).first.compareTo(((TextPair) b).first);
    }
    return super.compare(a, b);
  }
}
```

Serialization Frameworks

Although most MapReduce programs use `Writable` key and value types, this isn't mandated by the MapReduce API. In fact, any type can be used; the only requirement is a mechanism that translates to and from a binary representation of each type.

To support this, Hadoop has an API for pluggable serialization frameworks. A serialization framework is represented by an implementation of `Serialization` (in the `org.apache.hadoop.io.serializer` package). `WritableSerialization`, for example, is the implementation of `Serialization` for `Writable` types.

A `Serialization` defines a mapping from types to `Serializer` instances (for turning an object into a byte stream) and `Deserializer` instances (for turning a byte stream into an object).

Set the `io.serializations` property to a comma-separated list of classnames in order to register `Serialization` implementations. Its default value includes `org.apache.hadoop.io.serializer.WritableSerialization` and the Avro Specific and Reflect serializations (see **Avro Data Types and Schemas**), which means that only `Writable` or Avro objects can be serialized or deserialized out of the box.

Hadoop includes a class called `JavaSerialization` that uses Java Object Serialization. Although it makes it convenient to be able to use standard Java types such as `Integer` or `String` in MapReduce programs, Java Object Serialization is not as efficient as Writables, so it's not worth making this trade-off (see the following sidebar).

WHY NOT USE JAVA OBJECT SERIALIZATION?

Java comes with its own serialization mechanism, called Java Object Serialization (often referred to simply as "Java Serialization"), that is tightly integrated with the language, so it's natural to ask why this wasn't used in Hadoop. Here's what Doug Cutting said in response to that question:

> *Why didn't I use Serialization when we first started Hadoop? Because it looked big and hairy and I thought we needed something lean and mean, where we had precise control over exactly how objects are written and read, since that is central to Hadoop. With Serialization you can get some control, but you have to fight for it.*
>
> *The logic for not using RMI [Remote Method Invocation] was similar. Effective, high-performance inter-process communications are critical to Hadoop. I felt like we'd need to precisely control how things like connections, timeouts and buffers are handled, and RMI gives you little control over those.*

The problem is that Java Serialization doesn't meet the criteria for a serialization format listed earlier: compact, fast, extensible, and interoperable.

Serialization IDL

There are a number of other serialization frameworks that approach the problem in a different way: rather than defining types through code, you define them in a language-neutral, declarative fashion, using an *interface description language* (IDL). The system can then generate types for different languages, which is good for interoperability. They also typically define versioning schemes that make type evolution straightforward.

Apache Thrift and **Google Protocol Buffers** are both popular serialization frameworks, and both are commonly used as a format for persistent binary data. There is limited support for these as MapReduce formats;[46] however, they are used internally in parts of Hadoop for RPC and data exchange.

Avro is an IDL-based serialization framework designed to work well with large-scale data processing in Hadoop. It is covered in **Chapter 12**.

File-Based Data Structures

For some applications, you need a specialized data structure to hold your data. For doing MapReduce-based processing, putting each blob of binary data into its own file doesn't scale, so Hadoop developed a number of higher-level containers for these situations.

SequenceFile

Imagine a logfile where each log record is a new line of text. If you want to log binary types, plain text isn't a suitable format. Hadoop's `SequenceFile` class fits the bill in this situation, providing a persistent data structure for binary key-value pairs. To use it as a logfile format, you would choose a key, such as timestamp represented by a `LongWritable`, and the value would be a `Writable` that represents the quantity being logged.

`SequenceFile`s also work well as containers for smaller files. HDFS and MapReduce are optimized for large files, so packing files into a `SequenceFile` makes storing and processing the smaller files more efficient (**Processing a whole file as a record** contains a program to pack files into a `SequenceFile`).[47]

Writing a SequenceFile

To create a `SequenceFile`, use one of its `createWriter()` static methods, which return a `SequenceFile.Writer` instance. There are several overloaded versions, but they all require you to specify a stream to write to (either an `FSDataOutputStream` or a `FileSystem` and `Path` pairing), a `Configuration` object, and the key and value types. Optional arguments include the compression type and codec, a `Progressable` callback to be informed of write progress, and a `Metadata` instance to be stored in the `SequenceFile` header.

The keys and values stored in a `SequenceFile` do not necessarily need to be `Writable`s. Any types that can be serialized and deserialized by a `Serialization` may be used.

Once you have a `SequenceFile.Writer`, you then write key-value pairs using the `append()` method. When you've finished, you call the `close()` method (`SequenceFile.Writer` implements `java.io.Closeable`).

Example 5-10 shows a short program to write some key-value pairs to a `SequenceFile` using the API just described.

Example 5-10. Writing a SequenceFile

```
public class SequenceFileWriteDemo {

  private static final String[] DATA = {
    "One, two, buckle my shoe",
    "Three, four, shut the door",
    "Five, six, pick up sticks",
    "Seven, eight, lay them straight",
    "Nine, ten, a big fat hen"
  };

  public static void main(String[] args) throws IOException {
    String uri = args[0];
    Configuration conf = new Configuration();
    FileSystem fs = FileSystem.get(URI.create(uri), conf);
    Path path = new Path(uri);

    IntWritable key = new IntWritable();
    Text value = new Text();
    SequenceFile.Writer writer = null;
    try {
      writer = SequenceFile.createWriter(fs, conf, path,
          key.getClass(), value.getClass());

      for (int i = 0; i < 100; i++) {
        key.set(100 - i);
        value.set(DATA[i % DATA.length]);
        System.out.printf("[%s]\t%s\t%s\n", writer.getLength(), key, value);
        writer.append(key, value);
      }
    } finally {
      IOUtils.closeStream(writer);
    }
  }
}
```

The keys in the sequence file are integers counting down from 100 to 1, represented as `IntWritable` objects. The values are `Text` objects. Before each record is appended to the `SequenceFile.Writer`, we call the `getLength()` method to discover the current position in the file. (We will use this information about record boundaries in the next section, when we read the file nonsequentially.) We write the position out to the console, along with the key and value pairs. The result of running it is shown here:

```
% hadoop SequenceFileWriteDemo numbers.seq
[128]   100     One, two, buckle my shoe
[173]   99      Three, four, shut the door
[220]   98      Five, six, pick up sticks
[264]   97      Seven, eight, lay them straight
[314]   96      Nine, ten, a big fat hen
[359]   95      One, two, buckle my shoe
[404]   94      Three, four, shut the door
[451]   93      Five, six, pick up sticks
[495]   92      Seven, eight, lay them straight
[545]   91      Nine, ten, a big fat hen
...
[1976]  60      One, two, buckle my shoe
[2021]  59      Three, four, shut the door
[2088]  58      Five, six, pick up sticks
[2132]  57      Seven, eight, lay them straight
[2182]  56      Nine, ten, a big fat hen
...
[4557]  5       One, two, buckle my shoe
[4602]  4       Three, four, shut the door
[4649]  3       Five, six, pick up sticks
[4693]  2       Seven, eight, lay them straight
[4743]  1       Nine, ten, a big fat hen
```

Reading a SequenceFile

Reading sequence files from beginning to end is a matter of creating an instance of `SequenceFile.Reader` and iterating over records by repeatedly invoking one of the `next()` methods. Which one you use depends on the serialization framework you are using. If you are using `Writable` types, you can use the `next()` method that takes a key and a value argument and reads the next key and value in the stream into these variables:

```
public boolean next(Writable key, Writable val)
```

The return value is `true` if a key-value pair was read and `false` if the end of the file has been reached.

For other, non-Writable serialization frameworks (such as Apache Thrift), you should use these two methods:

```
public Object next(Object key) throws IOException
public Object getCurrentValue(Object val) throws IOException
```

In this case, you need to make sure that the serialization you want to use has been set in the `io.serializations` property; see **Serialization Frameworks**.

If the `next()` method returns a non-`null` object, a key-value pair was read from the stream, and the value can be retrieved using the `getCurrentValue()` method. Otherwise, if `next()` returns `null`, the end of the file has been reached.

The program in **Example 5-11** demonstrates how to read a sequence file that has `Writable` keys and values. Note how the types are discovered from the `SequenceFile.Reader` via calls to `getKeyClass()` and `getValueClass()`, and then `ReflectionUtils` is used to create an instance for the key and an instance for the value. This technique allows the program to be used with any sequence file that has `Writable` keys and values.

Example 5-11. Reading a SequenceFile

```
public class SequenceFileReadDemo {

  public static void main(String[] args) throws IOException {
    String uri = args[0];
    Configuration conf = new Configuration();
    FileSystem fs = FileSystem.get(URI.create(uri), conf);
    Path path = new Path(uri);

    SequenceFile.Reader reader = null;
    try {
```

```
        reader = new SequenceFile.Reader(fs, path, conf);
        Writable key = (Writable)
            ReflectionUtils.newInstance(reader.getKeyClass(), conf);
        Writable value = (Writable)
            ReflectionUtils.newInstance(reader.getValueClass(), conf);
        long position = reader.getPosition();
        while (reader.next(key, value)) {
          String syncSeen = reader.syncSeen() ? "*" : "";
          System.out.printf("[%s%s]\t%s\t%s\n", position, syncSeen, key, value);
          position = reader.getPosition(); // beginning of next record
        }
      } finally {
        IOUtils.closeStream(reader);
      }
    }
}
```

Another feature of the program is that it displays the positions of the *sync points* in the sequence file. A sync point is a point in the stream that can be used to resynchronize with a record boundary if the reader is "lost"—for example, after seeking to an arbitrary position in the stream. Sync points are recorded by `SequenceFile.Writer`, which inserts a special entry to mark the sync point every few records as a sequence file is being written. Such entries are small enough to incur only a modest storage overhead—less than 1%. Sync points always align with record boundaries.

Running the program in **Example 5-11** shows the sync points in the sequence file as asterisks. The first one occurs at position 2021 (the second one occurs at position 4075, but is not shown in the output):

```
% hadoop SequenceFileReadDemo numbers.seq
[128]   100     One, two, buckle my shoe
[173]   99      Three, four, shut the door
[220]   98      Five, six, pick up sticks
[264]   97      Seven, eight, lay them straight
[314]   96      Nine, ten, a big fat hen
[359]   95      One, two, buckle my shoe
[404]   94      Three, four, shut the door
[451]   93      Five, six, pick up sticks
[495]   92      Seven, eight, lay them straight
[545]   91      Nine, ten, a big fat hen
```

```
[590]   90      One, two, buckle my shoe
...
[1976]  60      One, two, buckle my shoe
[2021*] 59      Three, four, shut the door
[2088]  58      Five, six, pick up sticks
[2132]  57      Seven, eight, lay them straight
[2182]  56      Nine, ten, a big fat hen
...
[4557]  5       One, two, buckle my shoe
[4602]  4       Three, four, shut the door
[4649]  3       Five, six, pick up sticks
[4693]  2       Seven, eight, lay them straight
[4743]  1       Nine, ten, a big fat hen
```

There are two ways to seek to a given position in a sequence file. The first is the `seek()` method, which positions the reader at the given point in the file. For example, seeking to a record boundary works as expected:

```
reader.seek(359);
assertThat(reader.next(key, value), is(true));
assertThat(((IntWritable) key).get(), is(95));
```

But if the position in the file is not at a record boundary, the reader fails when the `next()` method is called:

```
reader.seek(360);
reader.next(key, value); // fails with IOException
```

The second way to find a record boundary makes use of sync points. The `sync(long position)` method on `SequenceFile.Reader` positions the reader at the next sync point after `position`. (If there are no sync points in the file after this position, then the reader will be positioned at the end of the file.) Thus, we can call `sync()` with any position in the stream—not necessarily a record boundary—and the reader will reestablish itself at the next sync point so reading can continue:

```
reader.sync(360);
assertThat(reader.getPosition(), is(2021L));
```

```
assertThat(reader.next(key, value), is(true));
assertThat(((IntWritable) key).get(), is(59));
```

WARNING

`SequenceFile.Writer` has a method called `sync()` for inserting a sync point at the current position in the stream. This is not to be confused with the `hsync()` method defined by the `Syncable` interface for synchronizing buffers to the underlying device (see **Coherency Model**).

Sync points come into their own when using sequence files as input to MapReduce, since they permit the files to be split and different portions to be processed independently by separate map tasks (see **SequenceFileInputFormat**).

Displaying a SequenceFile with the command-line interface

The `hadoop fs` command has a `-text` option to display sequence files in textual form. It looks at a file's magic number so that it can attempt to detect the type of the file and appropriately convert it to text. It can recognize gzipped files, sequence files, and Avro datafiles; otherwise, it assumes the input is plain text.

For sequence files, this command is really useful only if the keys and values have meaningful string representations (as defined by the `toString()` method). Also, if you have your own key or value classes, you will need to make sure they are on Hadoop's classpath.

Running it on the sequence file we created in the previous section gives the following output:

```
% hadoop fs -text numbers.seq | head
100     One, two, buckle my shoe
99      Three, four, shut the door
98      Five, six, pick up sticks
97      Seven, eight, lay them straight
96      Nine, ten, a big fat hen
95      One, two, buckle my shoe
94      Three, four, shut the door
93      Five, six, pick up sticks
```

```
92      Seven, eight, lay them straight
91      Nine, ten, a big fat hen
```

Sorting and merging SequenceFiles

The most powerful way of sorting (and merging) one or more sequence files is to use MapReduce. MapReduce is inherently parallel and will let you specify the number of reducers to use, which determines the number of output partitions. For example, by specifying one reducer, you get a single output file. We can use the sort example that comes with Hadoop by specifying that the input and output are sequence files and by setting the key and value types:

```
% hadoop jar \
  $HADOOP_HOME/share/hadoop/mapreduce/hadoop-mapreduce-examples-*.jar \
  sort -r 1 \
  -inFormat org.apache.hadoop.mapreduce.lib.input.SequenceFileInputFormat \
  -outFormat org.apache.hadoop.mapreduce.lib.output.SequenceFileOutputFormat \
  -outKey org.apache.hadoop.io.IntWritable \
  -outValue org.apache.hadoop.io.Text \
  numbers.seq sorted
% hadoop fs -text sorted/part-r-00000 | head
1       Nine, ten, a big fat hen
2       Seven, eight, lay them straight
3       Five, six, pick up sticks
4       Three, four, shut the door
5       One, two, buckle my shoe
6       Nine, ten, a big fat hen
7       Seven, eight, lay them straight
8       Five, six, pick up sticks
9       Three, four, shut the door
10      One, two, buckle my shoe
```

Sorting is covered in more detail in **Sorting**.

An alternative to using MapReduce for sort/merge is the `SequenceFile.Sorter` class, which has a number of `sort()` and `merge()` methods. These functions predate MapReduce and are lower-level functions than MapReduce (for example, to get parallelism, you need to partition your data manually), so in general MapReduce is the preferred approach to sort and merge sequence files.

The SequenceFile format

A sequence file consists of a header followed by one or more records (see Figure 5-2). The first three bytes of a sequence file are the bytes `SEQ`, which act as a magic number; these are followed by a single byte representing the version number. The header contains other fields, including the names of the key and value classes, compression details, user-defined metadata, and the sync marker.[48] Recall that the sync marker is used to allow a reader to synchronize to a record boundary from any position in the file. Each file has a randomly generated sync marker, whose value is stored in the header. Sync markers appear between records in the sequence file. They are designed to incur less than a 1% storage overhead, so they don't necessarily appear between every pair of records (such is the case for short records).

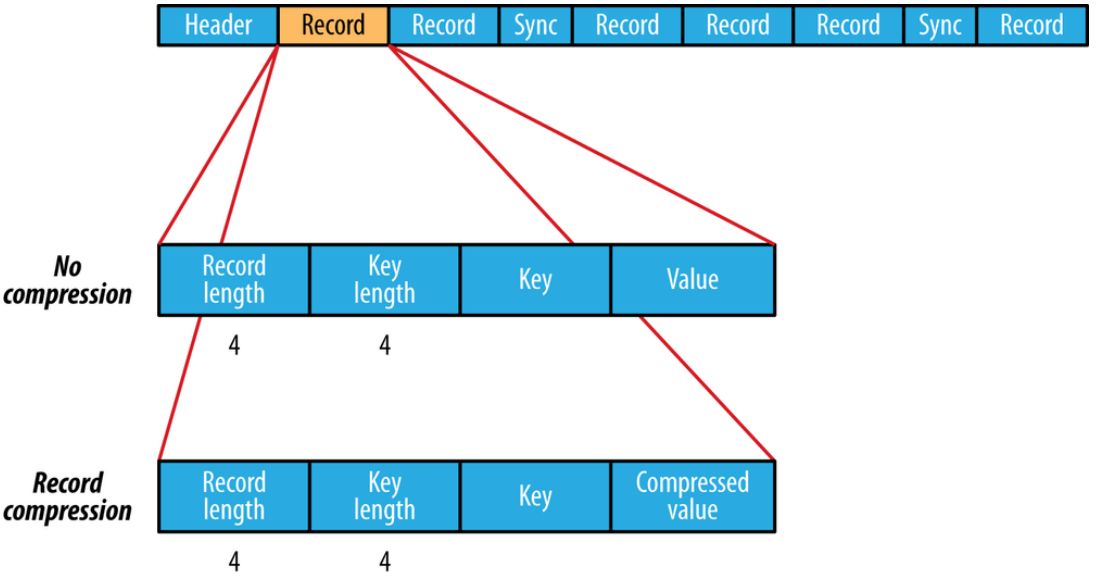

Figure 5-2. The internal structure of a sequence file with no compression and with record compression

The internal format of the records depends on whether compression is enabled, and if it is, whether it is record compression or block compression.

If no compression is enabled (the default), each record is made up of the record length (in bytes), the key length, the key, and then the value. The length fields are written as 4-byte integers adhering to the contract of the `writeInt()` method of `java.io.DataOutput`. Keys and values are serial-

ized using the `Serialization` defined for the class being written to the sequence file.

The format for record compression is almost identical to that for no compression, except the value bytes are compressed using the codec defined in the header. Note that keys are not compressed.

Block compression (**Figure 5-3**) compresses multiple records at once; it is therefore more compact than and should generally be preferred over record compression because it has the opportunity to take advantage of similarities between records. Records are added to a block until it reaches a minimum size in bytes, defined by the `io.seqfile.compress.blocksize` property; the default is one million bytes. A sync marker is written before the start of every block. The format of a block is a field indicating the number of records in the block, followed by four compressed fields: the key lengths, the keys, the value lengths, and the values.

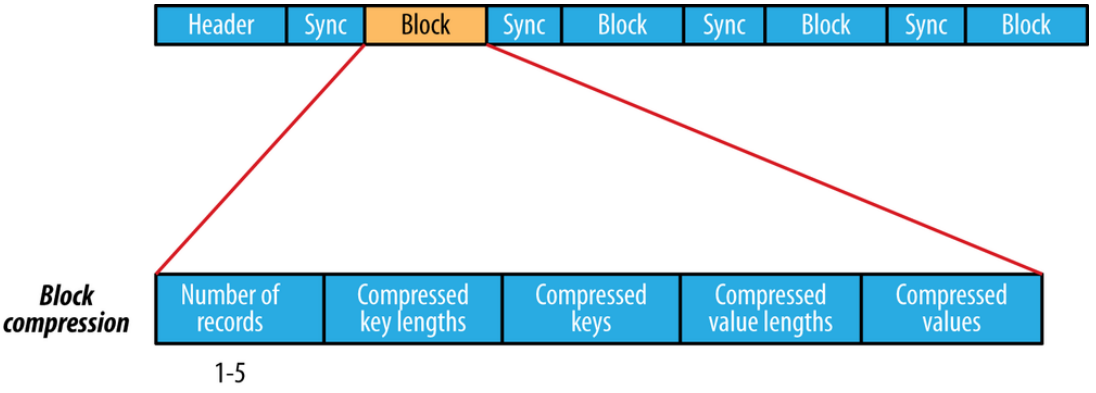

Figure 5-3. The internal structure of a sequence file with block compression

MapFile

A `MapFile` is a sorted `SequenceFile` with an index to permit lookups by key. The index is itself a `SequenceFile` that contains a fraction of the keys in the map (every 128th key, by default). The idea is that the index can be loaded into memory to provide fast lookups from the main data file, which is another `SequenceFile` containing all the map entries in sorted key order.

`MapFile` offers a very similar interface to `SequenceFile` for reading and writing—the main thing to be aware of is that when writing using

`MapFile.Writer`, map entries must be added in order, otherwise an `IOException` will be thrown.

MapFile variants

Hadoop comes with a few variants on the general key-value `MapFile` interface:

- `SetFile` is a specialization of `MapFile` for storing a set of `Writable` keys. The keys must be added in sorted order.
- `ArrayFile` is a `MapFile` where the key is an integer representing the index of the element in the array and the value is a `Writable` value.
- `BloomMapFile` is a `MapFile` that offers a fast version of the `get()` method, especially for sparsely populated files. The implementation uses a dynamic Bloom filter for testing whether a given key is in the map. The test is very fast because it is in-memory, and it has a non-zero probability of false positives. Only if the test passes (the key is present) is the regular `get()` method called.

Other File Formats and Column-Oriented Formats

While sequence files and map files are the oldest binary file formats in Hadoop, they are not the only ones, and in fact there are better alternatives that should be considered for new projects.

Avro datafiles (covered in **Avro Datafiles**) are like sequence files in that they are designed for large-scale data processing—they are compact and splittable—but they are portable across different programming languages. Objects stored in Avro datafiles are described by a schema, rather than in the Java code of the implementation of a `Writable` object (as is the case for sequence files), making them very Java-centric. Avro datafiles are widely supported across components in the Hadoop ecosystem, so they are a good default choice for a binary format.

Sequence files, map files, and Avro datafiles are all row-oriented file formats, which means that the values for each row are stored contiguously in the file. In a column-oriented format, the rows in a file (or, equivalently, a table in Hive) are broken up into row splits, then each split is stored in column-oriented fashion: the values for each row in the first col-

umn are stored first, followed by the values for each row in the second column, and so on. This is shown diagrammatically in **Figure 5-4**.

A column-oriented layout permits columns that are not accessed in a query to be skipped. Consider a query of the table in **Figure 5-4** that processes only column 2. With row-oriented storage, like a sequence file, the whole row (stored in a sequence file record) is loaded into memory, even though only the second column is actually read. Lazy deserialization saves some processing cycles by deserializing only the column fields that are accessed, but it can't avoid the cost of reading each row's bytes from disk.

With column-oriented storage, only the column 2 parts of the file (highlighted in the figure) need to be read into memory. In general, column-oriented formats work well when queries access only a small number of columns in the table. Conversely, row-oriented formats are appropriate when a large number of columns of a single row are needed for processing at the same time.

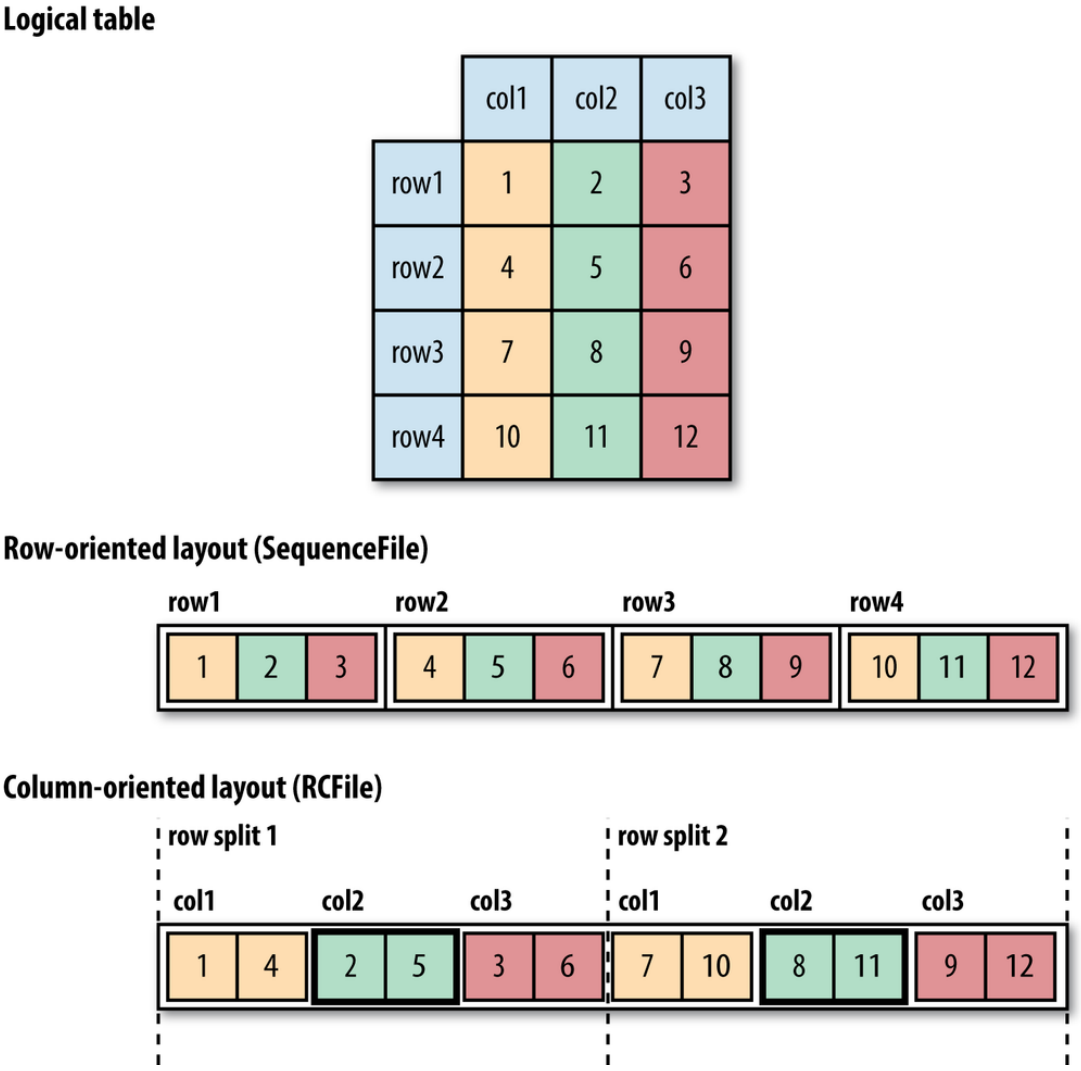

Figure 5-4. Row-oriented versus column-oriented storage

Column-oriented formats need more memory for reading and writing, since they have to buffer a row split in memory, rather than just a single row. Also, it's not usually possible to control when writes occur (via flush or sync operations), so column-oriented formats are not suited to streaming writes, as the current file cannot be recovered if the writer process fails. On the other hand, row-oriented formats like sequence files and Avro datafiles can be read up to the last sync point after a writer failure. It is for this reason that Flume (see **Chapter 14**) uses row-oriented formats.

The first column-oriented file format in Hadoop was Hive's *RCFile*, short for *Record Columnar File*. It has since been superseded by Hive's *ORCFile* (*Optimized Record Columnar File*), and *Parquet* (covered in **Chapter 13**). Parquet is a general-purpose column-oriented file format based on

Google's Dremel, and has wide support across Hadoop components. Avro also has a column-oriented format called *Trevni*.

---

[44] For a comprehensive set of compression benchmarks, ***jvm-compressor-benchmark*** is a good reference for JVM-compatible libraries (including some native libraries).

---

[45] This example is based on one from Norbert Lindenberg and Masayoshi Okutsu's **"Supplementary Characters in the Java Platform,"** May 2004.

---

[46] Twitter's **Elephant Bird project** includes tools for working with Thrift and Protocol Buffers in Hadoop.

---

[47] In a similar vein, the blog post **"A Million Little Files"** by Stuart Sierra includes code for converting a tar file into a `SequenceFile`.

---

[48] Full details of the format of these fields may be found in `SequenceFile`'s **documentation** and source code.