Hadoop I/O :

Data Integrity - Data Integrity in HDFS – Local File System – Checksum File System - Compression and Input Splits - Using Compression in Map Reduce - Serialization - The Writable Interface - Writable Classes - Implementing a Custom Writable - Serialization Frameworks - File-Based Data Structures – Sequence File - MapFile - Other File Formats and Column-Oriented Formats.

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 multi tera byte datasets.

- Hadoop tools or APIs that form the building blocks for developing distributed system, such as serialization frameworks and on-disk data structures.
- Hadoop Map Reduce job, input files are read from HDFS, Data are usually compressed to reduce the file sizes, after decompression, serialized bytes are transformed into Java objects before being passed to a user-defined map() function.
- Output records are serialized, compressed, and eventually pushed back to HDFS in conversely.
- Hadoop's Sequence File provides a persistent data structure for binary key-value pairs.

Hadoop I/O consists of

1. Data Integrity
2. Data Compression
3. Data Serialization
4. File-Based Data Structures

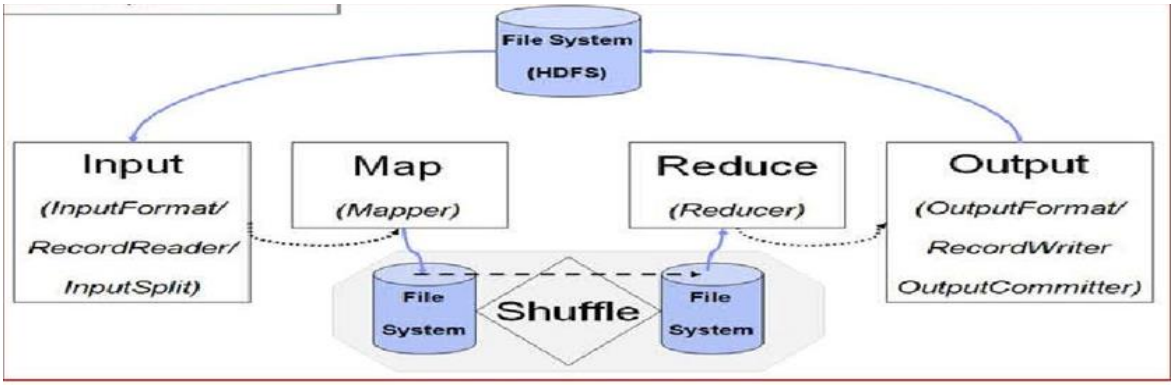


Fig: Basic flow of Hadoop I/O

1) Data Integrity:

Hadoop I/O is a technique doesn't offer any way to fix the data, just only error detection.

- Hadoop 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.

Data Integrity consists of

a) Data Integrity in HDFS

b) Local File System

c) Checksum File System

a) 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 io.bytes.per.checksum.
- HDFS stores replicas of blocks, it can "heal" corrupted blocks by copying one of the good replicas to produce a new, uncorrupt replica the datanode it was trying to read from to the namenode before throwing a Checksum Exception

b) Local File System:

The Hadoop Local File System performs client-side checksumming, this means that when you write a file a called filename,the file system client transparently creates a hidden file,.filename.crc, inthe same directory containing the checksums for each chunk of the file.

- Like HDFS, the chunk size is controlled by the io.bytes.per.check 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 fairly cheap to compute, typically adding a few percent overhead to the time to read or write a file.

c) Checksum File System:

Local File System uses Checksum File Systemto do its work, and class makes it easy to add checksumming to other (nonchecksummed) filesystems, as Checksum File System is just a wrapper around FileSystem.

- LocalFileSystem uses ChecksumFileSystem to do its work, and thisclass makes it easy to add checksumming to other filesystems, asChecksumFileSystem 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 onChecksumFileSystem.30

2. Data Compression:

File compression brings two major benefits as 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.
- It consists of
  - Codecs
  - Compression and Input Splits
  - Using Compression in MapReduce

a) Codecs: A codec is the implementation of a compression-decompression algorithm. In Hadoop, a codec is represented by an implementation of the Compression Codec interface

Table 4-2. 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

- Hadoop codecs must be downloaded separately fromhttp://code.google.com/p/hadoop-gpl-compression/

b) 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 64 MB, the file will be stored as 16 blocks, and a MapReduce job using this file as input will create 16 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 16 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.

c) Using Compression in MapReduce: When considering how to compress data that will be processed by Map Reduce, it is important to understand whether the compression format supports splitting. If your input files are compressed, they will be automatically decompressed as they are read by Map Reduce, using the filename extension to determine the codec to use.

```
JobConf conf = new JobConf(MaxTemperatureWithCompression.class);
conf.setJobName("Max temperature with output compression");

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

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

conf.setBoolean("mapred.output.compress", true);
conf.setClass("mapred.output.compression.codec", GzipCodec.class,
    CompressionCodec.class);

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

JobClient.runJob(conf);
```

2) Data 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 process of turning a byte stream back into a series of structured objects.

In Hadoop, inter process communication between nodes in the system is implemented using remote procedure calls (RPCs).In general, it is desirable that an RPC serialization format is:

- **Compact:** A compact format makes the best use of network bandwidth
- **Fast:** Inter process communication forms the backbone for a distributedsystem, 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 itshould be straightforward to evolve the protocol in a controlled manner forclients and servers.

- **Interoperable:** For some systems, it is desirable to be able to support clients that are written in different languages to the server.

It consists of

a) The Writable Interface

b) Writable Classes

c) Implementing a Custom Writable

d) Serialization Frameworks

e) Avro

a) The Writable Interface: The Writable interface defines two methods are one for writing its state to a Data Output binary stream, and one for reading its state from a Data Input binary stream

The bytes are written in big-endian order (so the most significant byte is written to the stream first, this 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;
```

b) Writable Classes: Hadoop comes with a large selection of Writable classes in the org.apache.hadoop.iopack

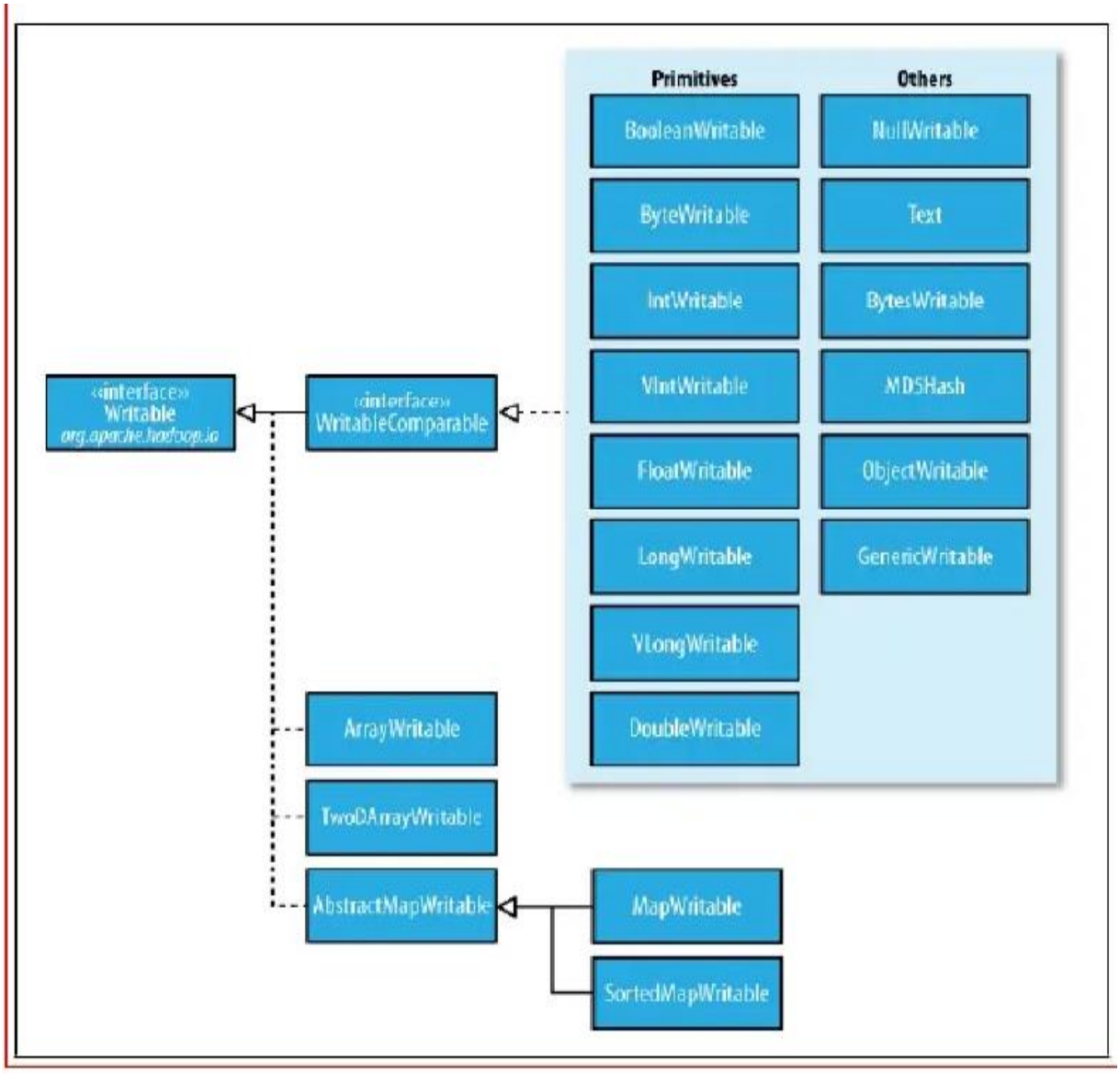


Fig: Writable wrappers for Java primitives

c) 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 Serialization | 105 a custom Writable, you have full control over the binary representation and the sort order.

d) Serialization Frameworks: 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 Writable Serialization, 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)

e) Avro: Apache Avro4 is a language-neutral data serialization system. The project was created by Doug Cutting (the creator of Hadoop) to address the major downside of Hadoop Writables: lack of language portability. The Avro specification precisely defines the binary format that all implementations must support.

4) File-Based Data Structures:

Apache Hadoop's Sequence File provides a persistent data structure for binary key-value pairs. In contrast with other persistent key-value data structures like B-Trees, you can't seek to a specified key editing, adding or removing it. This file is append-only.

- Map Reduce-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.
- It consists of
    - a)Sequence file
    - b)Map file

a) Sequence File: Imagine a log file, where each log record is a newline of text. If you want to log binary types, plain text isn't a suitable format. Hadoop's Sequence File class fits the bill in this :Hadoop I/O situation, providing a persistent data structure for binary key-value pairs

- **Writing a Sequence File:**To create a Sequence File, use one of its create Writer () static methods, which returns a Sequence File. Writer instance.
- **Reading a Sequence File:**Reading sequence files from beginning to end is a matter of creating an instance of Sequence File.
- **The Sequence File format:** A sequence file consists of a header followed by one or more records.

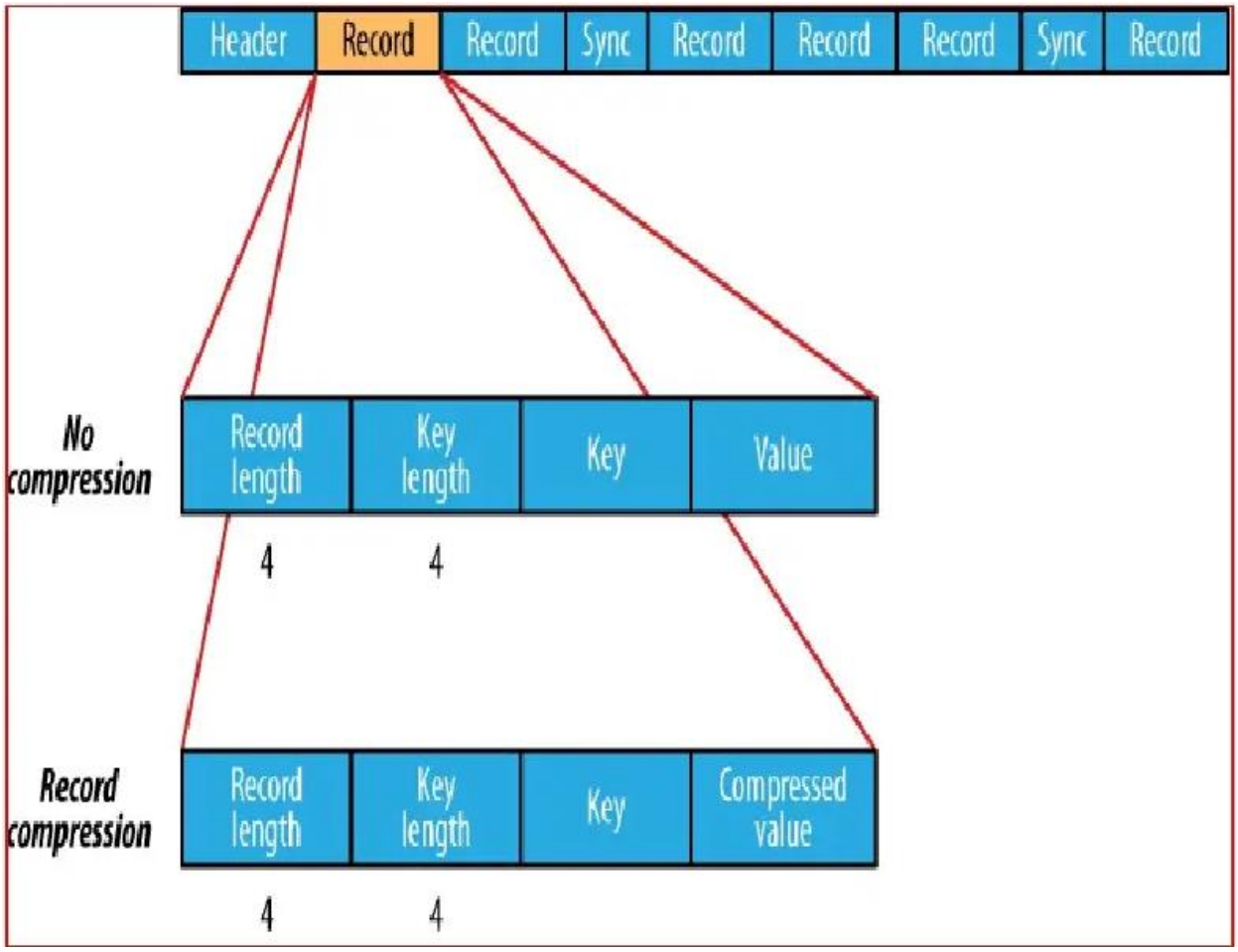


Fig: The internal structure of a sequence file with no compression and record compression

b) Map File: A Map File is a sorted Sequence File with an index to permit lookups by key. MapFile can be thought of as a persistent form of java.util.Map, which is able to grow beyond the size of a Map that is kept in memory.

- **Writing a MapFile:** Writing a MapFile is similar to writing a Sequence File: you create an instance of MapFile.
- **Reading a MapFile**: Iterating through the entries in order in a MapFile is similar to the procedure for a Sequence File: you create a MapFile.
- **Converting a Sequence File to a MapFile:** A MapFile is as an indexed and sorted Sequence File. So it's quite natural to want to be able to convert a Sequence File into a MapFile.

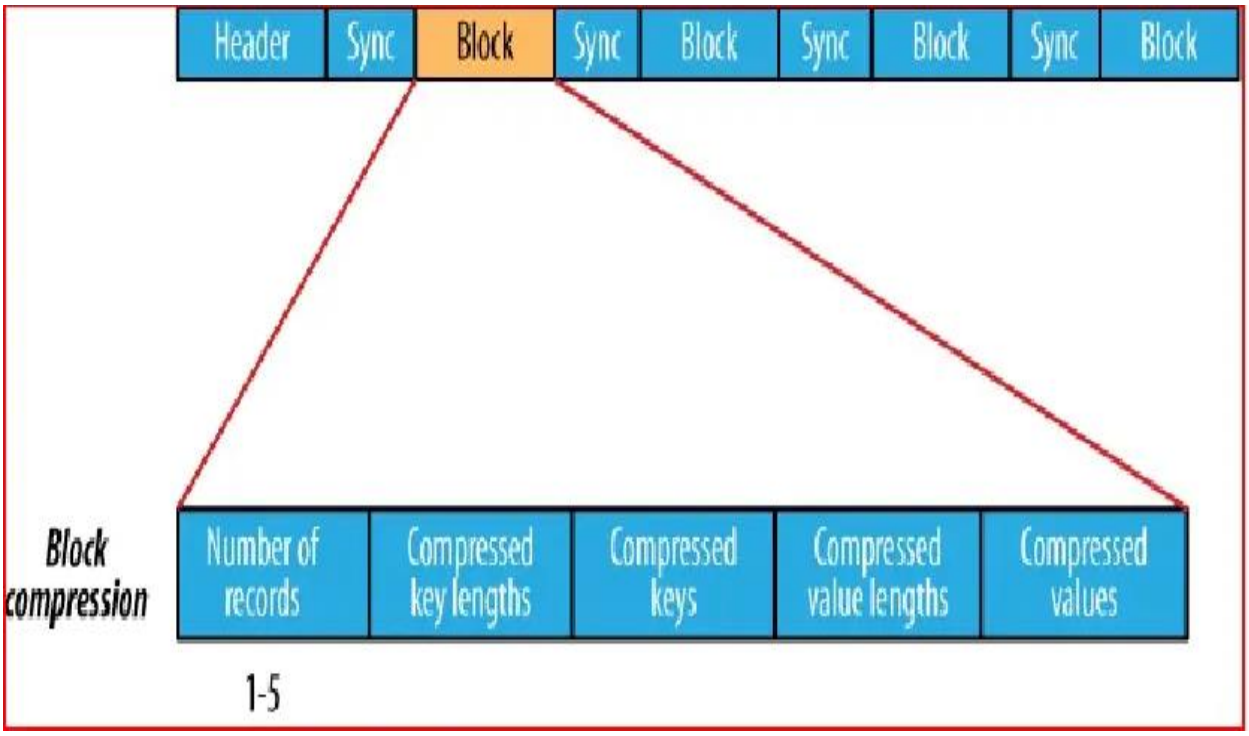


Fig: The internal structure of a sequence file with block compression

Other File Formats and Column-Oriented Formats.

The various Hadoop file formats have evolved in data engineering solutions to ease these issues across a number of use cases.

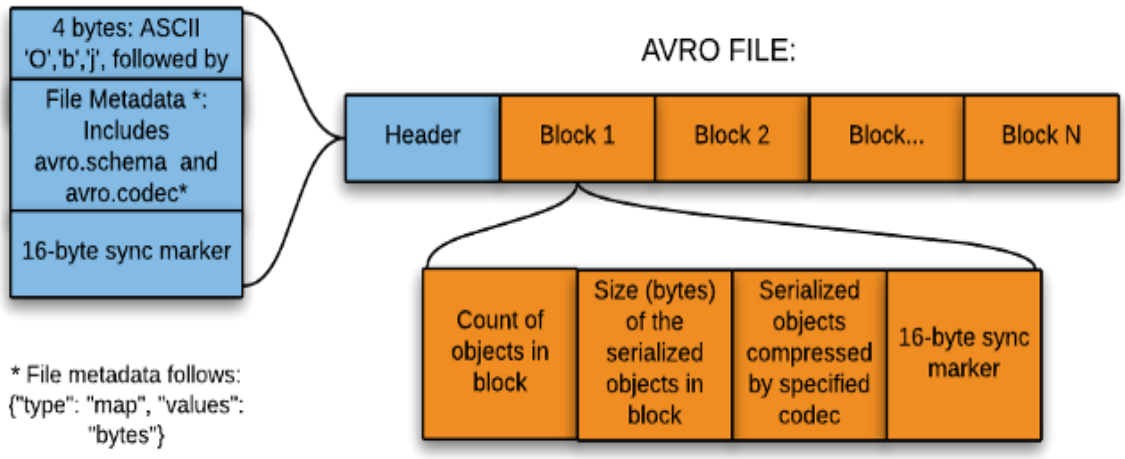
Choosing an appropriate file format can have some significant benefits:

1. Faster read times
2. Faster write times
3. Splittable files
4. Schema evolution support
5. Advanced compression support

Some file formats are designed for general use, others are designed for more specific use cases, and some are designed with specific data characteristics in mind. So there is quite a lot of choice.

AVRO File Format

- Avro format is a row-based storage format for Hadoop, which is widely used as a serialization platform.
- Avro format stores the schema in JSON format, making it easy to read and interpret by any program.
- The data itself is stored in a binary format making it compact and efficient in Avro files.
- Avro format is a language-neutral data serialization system. It can be processed by many languages (currently C, C++, C#, Java, Python, and Ruby).
- A key feature of Avro format is the robust support for data schemas that changes over time, i.e., schema evolution. Avro handles schema changes like missing fields, added fields, and changed fields.
- Avro format provides rich data structures. For example, you can create a record that contains an array, an enumerated type, and a sub-record.



The Avro format is the ideal candidate for storing data in a data lake landing zone because:

1. Data from the landing zone is usually read as a whole for further processing by downstream systems (the row-based format is more efficient in this case).
2. Downstream systems can easily retrieve table schemas from Avro files (there is no need to store the schemas separately in an external meta store).
3. Any source schema change is easily handled (schema evolution).

PARQUET File Format

Parquet, an open-source file format for Hadoop, stores **nested data structures** in a flat **columnar format**.

- Compared to a traditional approach where data is stored in a row-oriented approach, Parquet file format is more efficient in terms of storage and performance.
- It is especially good for queries that read particular columns from a "wide" (with many columns) table since only needed columns are read, and IO is minimized.

Columnar storage format

In order to understand the Parquet file format in Hadoop better, first, let's see what a columnar format is. In a column-oriented format, the values of each column of the same type in the records are stored together.

For exampleFor example, if there is a record comprising ID, employee Name, and Department, then all the values for the ID column will be stored together, values for the Name column together, and so on. If we take the same record schema as mentioned above, having three fields ID (int), NAME (varchar), and Department (varchar), the table will look something like this:

ID	Name	Department
1	emp1	d1
2	emp2	d2
3	emp3	d3

For this table, the data in a row-wise storage format will be stored as follows:

1	emp1	d1	2	emp2	d2	3	emp3	d3

Whereas, the same data in a Column-oriented storage format will look like this:

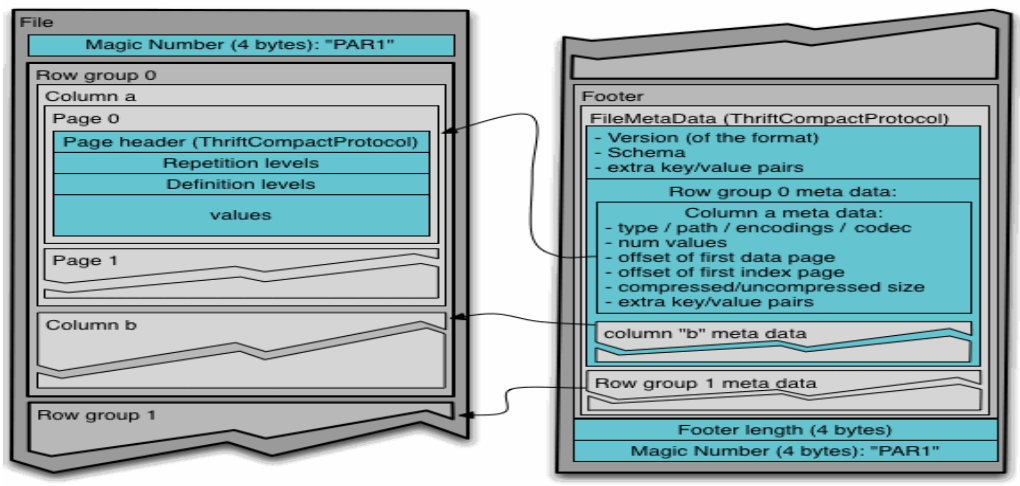
1	2	3	emp1	emp2	emp3	d1	d2	d3

The columnar storage format is more efficient when you need to query a few columns from a table. It will read only the required columns since they are adjacent, thus minimizing IO.

For example, let's say you want only the NAME column. In a row storage format, each record in the dataset has to be loaded, parsed into fields, and extracted the data for Name. The column-oriented format can directly go to the Name column as all the values for that column are stored together. It doesn't need to go through the whole record.

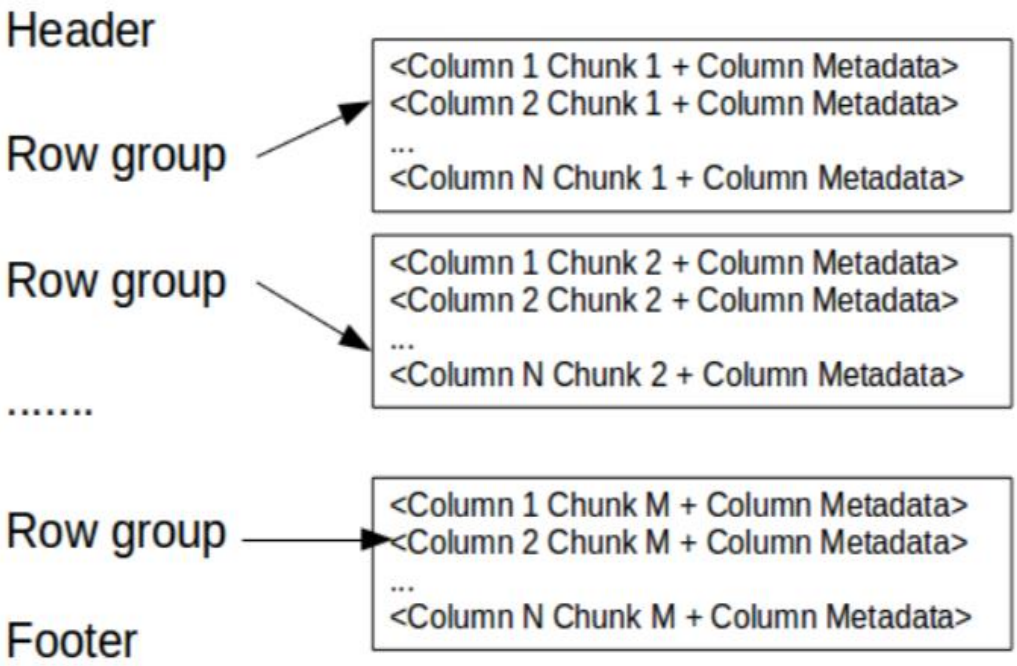
So, the column-oriented format increases the query performance as less seek time is required to go to the required columns, and less IO is required as it needs to read only the columns whose data are required.

One of the unique features of Parquet is that it can store data with nested structures in a columnar fashion too. This means that in a Parquet file format, even the nested fields can be read individually without reading all the fields in the nested structure. Parquet format uses the record shredding and assembly algorithm for storing nested structures in a columnar fashion.



To understand the Parquet file format in Hadoop, you should be aware of the following terms-

- **Row group**: A logical horizontal partitioning of the data into rows. A row group consists of a column chunk for each column in the dataset.
- **Column chunk**: A chunk of the data for a particular column. These column chunks live in a particular row group and are guaranteed to be contiguous in the file.
- **Page**: Column chunks are divided up into pages written back to back. The pages share a standard header and readers can skip the page they are not interested in.



Here, the Header contains a magic number “PAR1” (4-byte) that identifies the file as a Parquet format file.

Footer contains the following-

- File metadata- The file metadata contains the locations of all the column metadata start locations. It also includes the format version, the schema, and any extra key-value pairs. Readers are expected first to read the file metadata to find all the column chunks they are interested in. The column chunks should then be read sequentially.
- length of file metadata (4-byte)
- magic number “PAR1” (4-byte)

ORC File Format

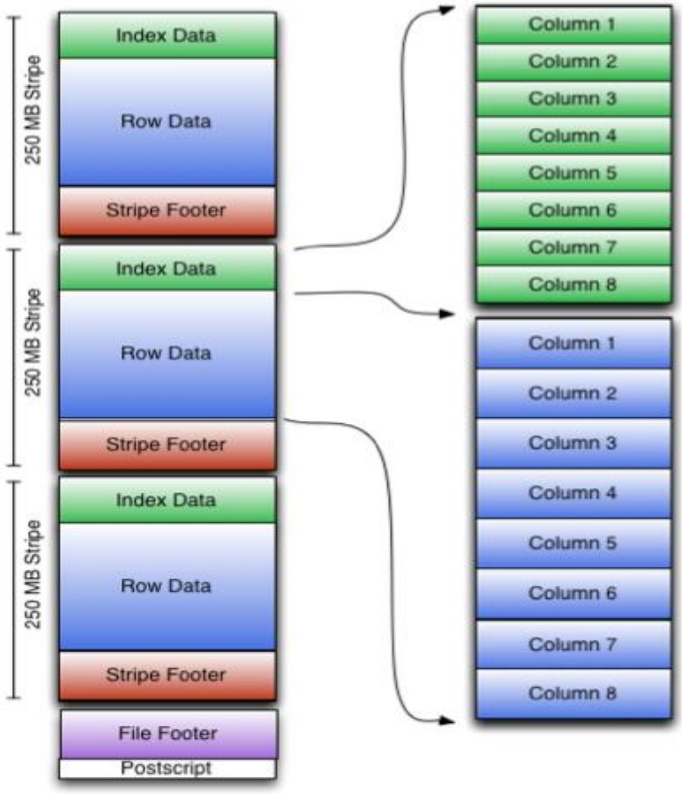
ORC File Format

The *Optimized Row Columnar* (ORC) file format provides a highly efficient way to store data. It was designed to overcome the limitations of other file formats. ORC file format ideally stores data compact and enables skipping over irrelevant parts without the need for large, complex, or manually maintained indices. The ORC file format addresses all of these issues.

ORC file format has many advantages such as:

- A single file as the output of each task, which reduces the NameNode’s load
- Hive type support including Date Time, decimal, and the complex types (struct, list, map, and union)
- Concurrent reads of the same file using separate Record Readers
- ORC file format has the ability to split files without scanning for markers

- Estimate an upper bound on heap memory allocation by the Reader/Writer based on the information in the file footer
- Metadata stored using Protocol Buffers, which allows the addition and removal of fields



ORC file format stores collections of rows in one file and within the collection the row data is stored in a columnar format.

An ORC file contains groups of row data called **stripes** and auxiliary information in a file **footer**. At the end of the file a **postscript** holds compression parameters and the size of the compressed footer.

The default stripe size is **250 MB**. Large stripe sizes enable large, efficient reads from HDFS.

The file footer contains:

- A list of stripes in the file.
- The number of rows per stripe.
- Each column's data type.

Stripe footer contains a directory of stream locations. It also contains column-level aggregates count, min, max, and sum.

Stripe footer contains a directory of stream locations.

Row data is used in table scans.

Index data include min and max values for each column and the row's positions within each column. ORC indexes are used only for the selection of stripes and row groups and not for answering queries.

Comparisons Between Different File Formats

AVRO vs. PARQUET

1. AVRO is a row-based storage format, whereas PARQUET is a columnar-based storage format.
2. PARQUET is much better for analytical querying, i.e., reads and querying are much more efficient than writing.
3. Writiing operations in AVRO are better than in PARQUET.
4. AVRO is much matured than PARQUET when it comes to schema evolution. PARQUET only supports schema append, whereas AVRO supports a much-featured schema evolution, i.e., adding or modifying columns.
5. PARQUET is ideal for querying a subset of columns in a multi-column table. AVRO is ideal in the case of ETL operations, where we need to query all the columns.

ORC vs. PARQUET

1. PARQUET is more capable of storing nested data.
2. ORC is more capable of Predicate Pushdown.
3. ORC supports ACID properties.
4. ORC is more compression efficient.