Avro File Format in Hadoop

Apache Avro is a data serialization system native to Hadoop which is also language independent. Apache Avro project was created by Doug Cutting, creator of Hadoop to increase data interoperability in Hadoop. Avro implementations for C, C++, C#, Java, PHP, Python, and Ruby are available making it easier to interchange data among various platforms.

#### What is data serialization

Just to make it clear here Data serialization is a mechanism to convert data (class objects, data structures) into a stream of bytes (binary form) in order to send it across network or store it persistently in a file or DB.

**Table of contents**

1. Avro in Hadoop
2. Avro file format
3. Schema Declaration in Avro
4. Primitive Types in Avro
5. Complex Types in Avro

#### Avro in Hadoop

Main features of Avro in Hadoop are-

* Avro is language independent
* It is schema based

To define structure for Avro data, language-independent schema is used. Avro schemas are defined using JSON that helps in data interoperability.

Some of the benefits of using schema in Avro are-

1. For language interoperability, since schema is defined using JSON.
2. You can save Avro schema in a separate file with **.avsc** extension.
3. It allows for evolution of schema. You can add or remove a column.
4. Using Avro you can perform serialization and deserialization without code generation. Since data in Avro is always stored with its corresponding schema, you can always read a serialized item regardless of whether you know the schema ahead of time.

#### Avro file format

Avro includes a simple object container file format. A file has a schema, and all objects stored in the file must be written according to that schema, using binary encoding. Objects are stored in blocks that may be compressed. Synchronization markers are used between blocks to permit efficient splitting of files for MapReduce processing.

Avro file consists of:

* A file header
* One or more file data blocks.

|  |  |  |  |
| --- | --- | --- | --- |
| Header | Data block | Data block | ……. |

A file header consists of:

* Four bytes, ASCII ‘O’, ‘b’, ‘j’, followed by 1.
* file metadata which includes the schema. Also contains information about the compression codec used to compress blocks.
* The 16-byte, randomly-generated sync marker for this file.

A file data block consists of:

* A long indicating the count of objects in this block.
* A long indicating the size in bytes of the serialized objects in the current block, after any codec is applied.
* The serialized objects. If a codec is specified, this is compressed by that codec.
* The file’s 16-byte sync marker.

#### Schema Declaration in Avro

A Schema is represented in JSON by one of:

* A JSON string, naming a defined type.
* A JSON object, of the form:{“type”: “typeName” …attributes…}  
  where typeName is either a primitive or derived type name, as defined below. Attributes not defined in this document are permitted as metadata, but must not affect the format of serialized data.
* A JSON array, representing a union of embedded types.

#### Primitive Types in Avro

The set of primitive type names is:

null: no value  
boolean: a binary value  
int: 32-bit signed integer  
long: 64-bit signed integer  
float: single precision (32-bit) IEEE 754 floating-point number  
double: double precision (64-bit) IEEE 754 floating-point number  
bytes: sequence of 8-bit unsigned bytes  
string: unicode character sequence

Primitive types have no specified attributes.

Primitive type names are also defined type names. Thus, for example, the schema “string” is equivalent to:

{“type”: “string”}

#### Complex Types in Avro

Avro supports six kinds of complex types: records, enums, arrays, maps, unions and fixed.

**Records**– Records use the type name “record” and support following attributes:

name: a JSON string providing the name of the record (required).  
namespace, a JSON string that qualifies the name;  
doc: a JSON string providing documentation to the user of this schema (optional).  
aliases: a JSON array of strings, providing alternate names for this record (optional).  
fields: a JSON array, listing fields (required). Each field is a JSON object with the following attributes:

For example, schema for employee record:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | {           "type":      "record",           "name":      "EmployeeRecord",           "doc":      "Employee Record",           "fields":      [                   {"name":     "empId",      "type":     "int"},                   {"name":     "empName",      "type":     "string"},                   {"name":     "age",          "type":     "int"}           ]  } |

**Enums**– Enums use the type name “enum” and support the following attributes:

name: a JSON string providing the name of the enum (required).  
namespace, a JSON string that qualifies the name;  
aliases: a JSON array of strings, providing alternate names for this enum (optional).  
doc: a JSON string providing documentation to the user of this schema (optional).  
symbols: a JSON array, listing symbols, as JSON strings (required). All symbols in an enum must be unique; duplicates are prohibited. Every symbol must match the regular expression [A-Za-z\_][A-Za-z0-9\_]\* (the same requirement as for names).

For example declaring days of week using an Enum:

|  |  |
| --- | --- |
| 1  2  3  4 | { "type": "enum",    "name": "WeekDays",    "symbols" : ["MONDAY", "TUESDAY", "WEDNESDAY", "THURSDAY", "FRIDAY", "SATURDAY", "SUNDAY"]  } |

**Arrays**– Arrays use the type name “array” and support a single attribute:

items: the schema of the array’s items.

For example declaring an array of strings:

|  |  |
| --- | --- |
| 1 | {"type": "array", "items": "string"} |

**Maps**– Maps use the type name “map” and support one attribute:  
values: the schema of the map’s values.  
Map keys are assumed to be strings.  
For example, a map from string to long is declared with:

|  |  |
| --- | --- |
| 1 | {"type": "map", "values": "long"} |

**union**– A union is represented using JSON array and each element in the array is a schema. For example, [“null”, “string”] declares a schema which may be either a null or string. Data confirming to the union schema must match one of the schema in the union.

**Fixed**– Fixed uses the type name “fixed” and supports two attributes:

name: a string naming this fixed (required).  
namespace, a string that qualifies the name;  
aliases: a JSON array of strings, providing alternate names for this enum (optional).  
size: an integer, specifying the number of bytes per value (required).

For example, declaring a 16-byte quantity:

|  |  |
| --- | --- |
| 1 | {"type": "fixed", "size": 16, "name": "md5"} |

That’s all for the topic **Avro File Format in Hadoop**. If something is missing or you have something to share about the topic please write a comment.

# Parquet File Format in Hadoop

Apache Parquet is a columnar storage format used in the Apache Hadoop eco system.

**Table of contents**

1. What is a column oriented format
2. Benefits of using Columnar Storage format
3. Parquet file format
4. Parquet file format Structure
5. Types in Parquet format
6. Logical Types in Parquet format

#### What is a column oriented format

Before going into Parquet file format in Hadoop let’s first understand what is column oriented file format and what benefit does it provide.

In a column oriented storage format, values are stored columns wise i.e. values of each row in the same column are stored rather than storing the data row wise as in the traditional row type data format.

**As example** if there is a table with 3 columns ID (int), NAME (varchar) and AGE (int)

|  |  |  |
| --- | --- | --- |
| **ID** | **NAME** | **AGE** |
| 1 | N1 | 35 |
| 2 | N2 | 45 |
| 3 | N3 | 55 |

Then in a row wise storage format the data will be stored as follows-

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | N1 | 35 | 2 | N2 | 45 | 3 | N3 | 55 |

In columnar format same data will be stored column-wise as follows-

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | N1 | N2 | N3 | 35 | 45 | 55 |

#### Benefits of using Columnar Storage format

As you can see from the layout in the above example, even if you query only the Name column, in the row oriented format whole row will be loaded into the memory. With the column oriented format if the Name is queried, only the Name column will be read into memory. That way query performance is improved as less I/O is required to read the same data.

Also you can notice from the layout that the data of the same data type is residing adjacent to each other. That helps is compressing the data better so less storage is required.

#### Parquet file format

Parquet file format being the columnar oriented format brings the same benefit in terms of-

1. Less storage
2. Increased query performance

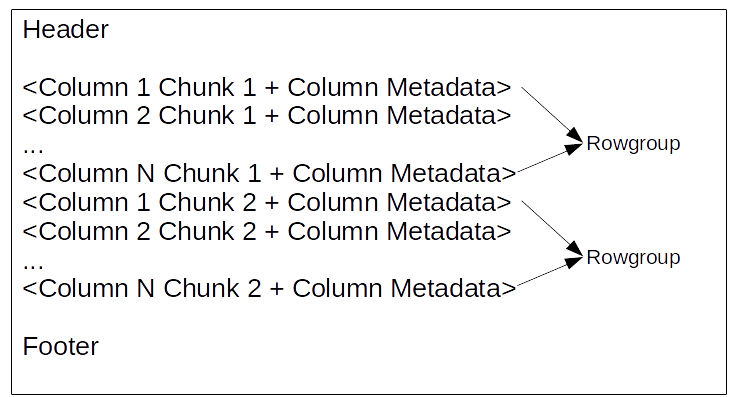
Apart from that Parquet format also has a feature to store even the nested structures in the columnar oriented format. Other columnar formats tend to store nested structures by flattening it and storing only the top level in columnar format.

Parquet file format can be used with any Hadoop ecosystem like Hive, Impala , Pig, and Spark.

#### Parquet file format Structure

A parquet file consists of Header, Row groups and Footer.

The format is as follows-



***Parquet file format***

**Header**– The header contains a 4-byte magic number “PAR1” which means the file is a Parquet format file.

**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.

**Page**– Column chunks are divided up into pages.

**Footer**– Contains the file metadata which includes the version of the format, schema, extra key/value pairs and the locations of all the column metadata start locations. Readers are expected to first read the file metadata to find all the column chunks they are interested in. The columns chunks should then be read sequentially.

#### Types in Parquet format

The types supported by the parquet file format are intended to be as minimal as possible, with a focus on how the types effect on disk storage.The types are:

BOOLEAN: 1 bit boolean  
INT32: 32 bit signed ints  
INT64: 64 bit signed ints  
INT96: 96 bit signed ints  
FLOAT: IEEE 32-bit floating point values  
DOUBLE: IEEE 64-bit floating point values  
BYTE\_ARRAY: arbitrarily long byte arrays.

#### Logical Types in Parquet format

Logical types are used to extend the types that parquet can be used to store, by specifying how the primitive types should be interpreted. This keeps the set of primitive types to a minimum and reuses parquet’s efficient encodings.

Full list of logical types can be accessed here – <https://github.com/apache/parquet-format/blob/master/LogicalTypes.md>

That’s all for the topic **Parquet File Format in Hadoop**. If something is missing or you have something to share about the topic please write a comment.

# Sequence File Format in Hadoop

Sequence files in Hadoop are flat files that store data in the form of **serialized key/value pairs**. Sequence file format is one of the binary file format supported by Hadoop and it integrates very well with MapReduce (also Hive and PIG).

Some of the features of the Sequence files in Hadoop are as follows –

1. Stores data in binary form so works well in scenarios where you want to store images in HDFS, model complex data structures as (key, value) pair.
2. Sequence files in Hadoop support both compression and splitting. When you compress a sequence file whole file is not compressed as a single unit but the records or the block of records are compressed with in the sequence file. Because of that sequence file can support splitting even if the compressor used is not splittable like Snappy, Lz4 or Gzip.
3. Sequence file can also be used as a container for storing a large number of small files. Since Hadoop works best with large files so storing large number of small files with in a sequence file makes processing more efficient and also requires less NameNodememory as it has to store metadata about one sequence file rather than many small files.
4. Since data is stored in (key, value) pair in Sequence file, internally the temporary outputs of maps are stored using SequenceFile.

**Table of contents**

1. SequenceFile Compression types
2. Sync points in sequence file
3. SequenceFile Formats in Hadoop
4. SequenceFile classes

#### SequenceFile Compression types

For sequence files in Hadoop there are three choices for compression.

1. **NONE** – Both key/value are uncompressed.
2. **RECORD** – If sequence file compression type is RECORD then only values are compressed.
3. **BLOCK** – If sequence file compression type is BLOCK then both keys and values are compressed. Both keys and values are collected in ‘blocks’ separately and compressed. The size of the ‘block’ is configurable. You will have to modify the following property in core-site.xml.  
   **io.seqfile.compress.blocksize** – The minimum block size for compression in block compressed SequenceFiles. Default is 1000000 bytes (1 million bytes).

#### Sync points in sequence file

In Sequence file sync-markers are recorded every few 100 bytes. Because of these sync points sequence file is splittale and can be used as input to MapReduce.

#### SequenceFile Formats in Hadoop

There are three different sequence file formats depending on the selected compression type. Note that the header format remains same across all.

**SequenceFile Header format**

**Version** – 3 bytes of magic header SEQ, followed by 1 byte of actual version number (e.g. SEQ4 or SEQ6)  
**KeyClassName** – key class  
**ValueClassName** – value class  
**Compression** – A boolean which specifies if compression is turned on for keys/values in this file.  
**BlockCompression** – A boolean which specifies if block-compression is turned on for keys/values in this file.  
**Compression codec** – CompressionCodec class which is used for compression of keys and/or values (if compression is enabled).  
**Metadata** – SequenceFile.Metadata for this file.  
**Sync** – A sync marker to denote end of the header.

**Uncompressed SequenceFile Format**

* + Header
  + Record
    - Record length
    - Key length
    - Key
    - Value
  + A sync-marker every few 100 bytes or so.

**Record-Compressed SequenceFile Format**

* Header
* Record
  + Record length
  + Key length
  + Key
  + Compressed Value
* A sync-marker every few 100 bytes or so.

Sequence files in hadoop

***Record SequenceFile Format***

**Block-Compressed SequenceFile Format**

* Header
* Record Block
  + Uncompressed number of records in the block
  + Compressed key-lengths block-size
  + Compressed key-lengths block
  + Compressed keys block-size
  + Compressed keys block
  + Compressed value-lengths block-size
  + Compressed value-lengths block
  + Compressed values block-size
  + Compressed values block
* A sync-marker every block.

Sequence file block format

***Sequence file block format***

#### SequenceFile classes

SequenceFile provides **SequenceFile.Writer**, **SequenceFile.Reader** and **SequenceFile.Sorter** classes for writing, reading and sorting respectively.

There are three SequenceFile Writers based on the SequenceFile.CompressionType used to compress key/value pairs:

* **Writer** : Uncompressed records.
* **RecordCompressWriter** : Record-compressed files, only compress values.
* **BlockCompressWriter** : Block-compressed files, both keys & values are compressed.

The recommended way is to use the static **createWriter methods** provided by the SequenceFile to chose the preferred format.

The SequenceFile.Reader can read any of the above SequenceFile formats.

That’s all for the topic **Sequence File Format in Hadoop**. If something is missing or you have something to share about the topic please write a comment.