* **AVRO**

Apache Avrois a language-neutral data serialization system.Avro schemas are usually written in JSON, and data is usually encoded using a binary format, but there are other options, too. There is a higher-level language called Avro IDL for writing schemas in a C-like language that is more familiar to developers. There is also a JSON-based data encoder, which, being human readable, is useful for prototyping and debugging Avro data.Pig, Hive, Crunch, Spark can read and write Avro datafiles.Avro has rich schema resolution capabilities. Within certain carefully defined constraints, the schema used to read data need not be identical to the schema that was used to write the data. This is the mechanism by which Avro supports schema evolution. For example, a new, optional field may be added to a record by declaring it in the schema used to read the old data. New and old clients alike will be able to read the old data, while new clients can write new data that uses the new field. Conversely, if an old client sees newly encoded data, it will gracefully ignore the new field and carry on processing as it would have done with old data.Avro datafiles support compression and are splittable, which is crucial for a MapReduce data input format.A datafile has a header containing metadata, including the Avro schema and a sync marker, followed by a series of (optionally compressed) blocks containing the serialized

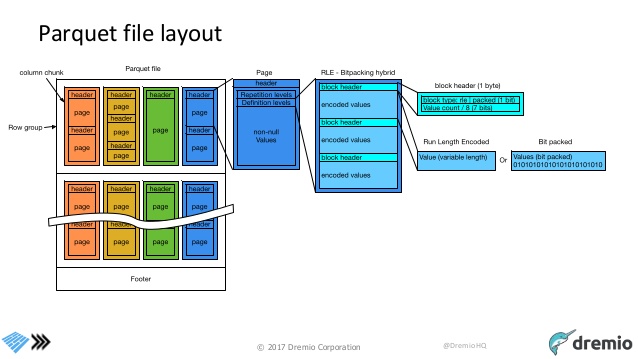
Avro objects. Blocks are separated by a sync marker that is unique to the file (the marker for a particular file is found in the header) and that permits rapid resynchronization with a block boundary after seeking to an arbitrary point in the file, such as an HDFS block boundary. Thus, Avro datafiles are splittable, which makes them amenable to efficient MapReduce processing.Avro defines a sort order for objects. All types except record have preordained rules for their sort order, as described in the Avro specification, that cannot be overridden by the user. For records, however, you can control the sort order by specifying the order attribute for a field. It takes one of three values: ascending (the default), descending (to reverse the order), or ignore (so the field is skipped for comparison purposes). **Avro implements efficient binary comparisons. That is to say, Avro does not have to deserialize binary data into objects to perform the comparison, because it can instead work directly on the byte streams.** In the case of the original StringPair schema (with no order attributes), for example, Avro implements the binary comparison as follows.The first field, left, is a UTF-8-encoded string, for which Avro can compare the bytes lexicographically. If they differ, the order is determined, and Avro can stop the comparison there. Otherwise, if the two byte sequences are the same, it compares the second two

(right) fields, again lexicographically at the byte level because the field is another UTF-8 string. Avro is best choice for RPC and masseging service like kafka ( kafka avro schema registry makes it easy to handle continious schema evolution)

to read avro data in human readable format.

% java -jar $AVRO\_HOME/avro-tools-\*.jar tojson pairs.avro

* **Parquet**



A Parquet file consists of a header followed by one or more blocks, terminated by a footer. The header contains only a 4-byte magic number, PAR1, that identifies the file as being in Parquet format, and all the file metadata is stored in the footer. The footer’s metadata includes the **format version**, the **schema**, any **extra key-value pairs**, and **metadata** for every block in the file. The final two fields in the footer are a 4-byte field encoding the length of the footer metadata, and the magic number again. The consequence of storing the metadata in the footer is that reading a Parquet file requires an initial seek to the end of the file (minus 8 bytes) to read the footer metadata length, then a second seek backward by that length to read the footer metadata. Unlike sequence files and Avro datafiles, where the metadata is stored in the header and sync markers are used to separate blocks, Parquet files don’t need sync markers since the block boundaries are stored in the footer metadata. Therefore, Parquet files are splittable, since the blocks can be located after reading the footer and can then be processed in parallel . **Each page contains values from the same column**, making a page a very good candidate for compression since the values are likely to be similar. The first level of compression is achieved through how the values are encoded. The simplest encoding is plain encoding, where values are written in full, but this doesn’t afford any compression in itself. Parquet also uses more compact encodings, including

**delta encoding** (the difference between values is stored),

**run-length** encoding (sequences of identical values are encoded as a single value and the count), and **dictionary encoding** (a dictionary of values is built and itself encoded, then values are encoded as integers representing the indexes in the dictionary).

In most cases, it also applies techniques such as **bit packing** to save space by storing several small values in a single byte. When writing files, Parquet will choose an appropriate encoding automatically, based on the **column type**. For example, Boolean values will be written using a combination of runlength encoding and bit packing. Most types are encoded using dictionary encoding by default; however, a plain encoding will be used as a fallback if the dictionary becomes too large. The threshold size at which this happens is referred to as the dictionary page size and is the same as the page size by default. Note that the encoding that is actually used is stored in the file metadata to ensure that readers use the correct encoding. In addition to the encoding, a second level of compression can be applied using a standard compression algorithm on the encoded page bytes. By default, no compression is applied, but Snappy, gzip, and LZO compressors are all supported. For nested data, each page will also store the definition and repetition levels for all the values in the page. Since levels are small integers, they can be very efficiently encoded using a bit-packed run-length encoding. A page is the smallest unit of storage in a Parquet file, so retrieving an arbitrary row requires that the page containing the row be decompressed and decoded. Thus, for single-row lookups, it is more efficient to have smaller pages, so there are fewer values to read through before reaching the target value. However, smaller pages incur a higher storage and processing overhead, due to the extra metadata (offsets, dictionaries) resulting from more pages. The default page size is 1 MB.Using Parqet **a reduction in size by one third on our large datasets was experienced**. **Scan times were also reduced to a fraction of the original** in the common case of needing only a subset of the columns. For each column . Storing definition levels and repetition levels efficiently In regards to storage, this effectively boils down to creating three sub columns for each primitive type. However, the overhead for storing these sub columns is low thanks to the columnar representation. That’s because levels are bound by the depth of the schema and can be stored efficiently using only a few bits per value. The levels will always have zero as a lower bound and the depth of the column as an upper bound. Even better, **fields that are not repeated do not need a repetition level** and **required fields do not need a definition level**, bringing down the upper bound. **In the special case of a flat schema with all fields, the repetition levels and definition levels are omitted completely and we only store the values of the columns.** This is effectively the same representation we would choose if we had to support only flat tables.

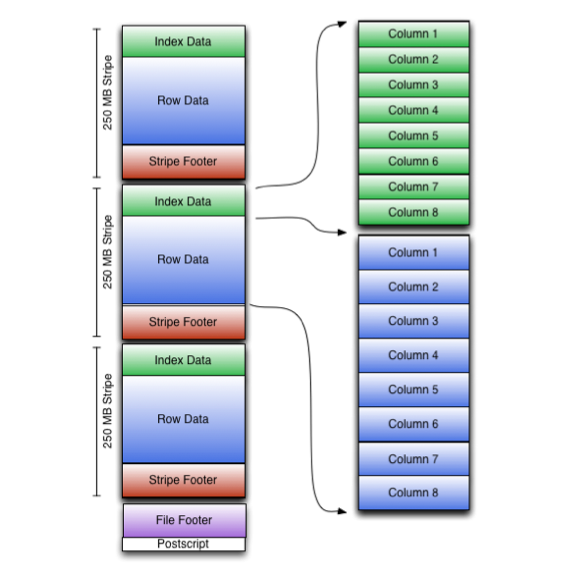
<https://blog.twitter.com/engineering/en_us/a/2013/dremel-made-simple-with-parquet.html>

[https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/36632.pdf](https://static.googleusercontent.com/media/research.google.com/en/pubs/archive/36632.pdf)

* **ORC**

Compared with RCFile format, for example, 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 datetime, decimal, and the complex types
* light-weight indexes stored within the file
* skip row groups that don’t pass predicate filtering
* seek to a given row
* block-mode compression based on data type
* run-length encoding for integer columns
* dictionary encoding for string columns
* concurrent reads of the same file using separate RecordReaders
* ability to split files without scanning for markers
* bound the amount of memory needed for reading or writing
* metadata stored using Protocol Buffers, which allows addition and removal of fields



### **File Structure** An ORC file contains groups of row data called **stripes**, along with 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, and each column's data type. It also contains column-level aggregates count, min, max, and sum.

### **Stripe Structure** As shown in the diagram, each stripe in an ORC file holds index data, row data, and a stripe footer.The **stripe footer** contains a directory of stream locations. **Row data** is used in table scans.**Index data** includes min and max values for each column and the row positions within each column. (A bit field or bloom filter could also be included.) Row index entries provide offsets that enable seeking to the right compression block and byte within a decompressed block.  Note that ORC indexes are used only for the selection of stripes and row groups and not for answering queries.Having relatively frequent row index entries enables row-skipping within a stripe for rapid reads, despite large stripe sizes. By default every 10,000 rows can be skipped.With the ability to skip large sets of rows based on filter predicates, you can sort a table on its secondary keys to achieve a big reduction in execution time. For example, if the primary partition is transaction date, the table can be sorted on state, zip code, and last name. Then looking for records in one state will skip the records of all other states.

ORC provides three level of indexes within each file:

* file level - statistics about the values in each column across the entire file
* stripe level - statistics about the values in each column for each stripe
* row level - statistics about the values in each column for each set of 10,000 rows within a stripe

The file and stripe level column statistics are in the file footer so that they are easy to access to determine if the rest of the file needs to be read at all. Row level indexes include both the column statistics for each row group and the position for seeking to the start of the row group.