**Understanding Different BigData File Types**

With so many file types coming into play as the BigData world is ever expanding and the big players supporting certain file types, it is time to get a peek into the file type in a bit more details. This white paper tries to cover extensively used file types in BigData.

1. File Type Comparison

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **File Types** | **Performance** | **Block Compression** | **Splittable** |  |
| Text/csv | write optimized | No | Yes | readable and ubiquitously parsable. file structure is dependent on field order, new fields can only be appended at the end of records while existing fields can never be deleted. So CSV files have limited support for schema evolution. Only contains data no metadata |
| XML |  | No | No | To be avoided within Hadoop |
| JSON |  | No | No | Stores metadata with data |
| AVRO |  | Yes | Yes | can rename, add, delete and change the data types of fields by defining new independent schema |
| Sequence |  | Yes |  | Similar to csv, do not store metadata, store data in binary format, complexity in reading. Often only used for “in flight” data such as intermediate data storage used within a sequence of MapReduce jobs. |
| RC (Record Columnar) | Read optimized | Yes |  | significant compression and query performance benefits. writing an RC file requires more memory and computation than non-columnar file formats |
| ORC (Optimized RC) | Read optimized (Better than RC) | Yes (Better than RC) |  | invented to optimize performance in Hive and are primarily backed by HortonWorks. Cloudera Impala does not support ORC files. |
| Parquet | Read optimized | Yes |  | Columnar file format. new columns can be added at the end of the structure. At present, Hive and Impala are able to query newly added columns, but other tools in the ecosystem such as Hadoop Pig may face challenges. Parquet is supported by Cloudera and optimized for Cloudera Impala. Parquet column names are lowercase. If your Parquet file contains mixed case column names, Hive will not be able to read the column and will return queries on the column with null values and not log any errors. Unlike Hive, Impala handles mixed case column names |

* If you are storing intermediate data between MapReduce jobs, then Sequence files are preferred.
* If query performance against the data is most important, ORC (HortonWorks/Hive) or Parquet (Cloudera/Impala) are optimal --- but these files will take longer to write. (query performance improves when using Parquet with Spark SQL)
* Avro is great if schema is going to change over time, but query performance will be slower than ORC or Parquet.
* CSV files are excellent if going to extract data from Hadoop to bulk load into a database.

Commonly used compression technologies that enable efficient block storage and processing:

* Snappy
* LZO

To give an idea how file along compression impacts the size of file (below example takes a csv file):

* Uncompressed CSV - 1.8 GB
* Avro - 1.5 GB
* Avro w/ Snappy Compression - 750 MB
* Parquet w/ Snappy Compression - 300 MB

But the file compression comes with its own set of challenges. There are three types of performance to consider:

* Write performance -- how fast can the data be written
* Partial read performance -- how fast can you read individual columns within a file
* Full read performance -- how fast can you read every data element in a file

Any format that is not splittable should generally be avoided such as XML File and JSON File formats. Each of these formats contain a single document per file with an opening tag at the beginning and a closing tag at the end.

1. **JSON**
   1. **JSON – JavaScript Object Notation**

* a syntax for storing and exchanging data
* an easier-to-use alternative to XML
* a lightweight data-interchange format
* language independent (JSON format is text only, just like XML. Text can be read and used as a data format by any programming language)
* self-describing and easy to understand
  1. **JSON Example**

{"employees":[  
    {"firstName":"John", "lastName":"Doe"},  
    {"firstName":"Anna", "lastName":"Smith"},  
    {"firstName":"Peter", "lastName":"Jones"}  
]}

* 1. **XML (eXensible Markup Language) Example**

<employees>  
    <employee>  
        <firstName>John</firstName> <lastName>Doe</lastName>  
    </employee>  
    <employee>  
        <firstName>Anna</firstName> <lastName>Smith</lastName>  
    </employee>  
    <employee>  
        <firstName>Peter</firstName> <lastName>Jones</lastName>  
    </employee>  
</employees>

* 1. **Comparison XML and JSON**

|  |  |
| --- | --- |
| **Similarity** | **Difference** |
| "self describing" (human readable) | JSON doesn't use end tag |
| hierarchical (values within values) | JSON is shorter |
| can be parsed and used by lots of programming languages | JSON is quicker to read and write |
| can be fetched with an XMLHttpRequest | JSON can use arrays |
|  | XML has to be parsed with an XML parser, JSON can be parsed by a standard JavaScript function |

* 1. **JSON Syntax Rules**

JSON syntax is derived from JavaScript object notation syntax:

* Data is in name/value pairs - A name/value pair consists of a field name (in double quotes), followed by a colon, followed by a value: "firstName":"John"
* Data is separated by commas - "firstName":"John", "lastName":"Doe"
* Curly braces hold objects - {"firstName":"John", "lastName":"Doe"}
* Square brackets hold arrays
  1. **JSON Values**

JSON values can be:

* A number (integer or floating point)
* A string (in double quotes)
* A Boolean (true or false)
* An array (in square brackets)
* An object (in curly braces)
* Null
  1. **JSON Objects**

JSON objects are written inside curly braces. Just like JavaScript, JSON objects can contain multiple name/values pairs: {"firstName":"John", "lastName":"Doe"}

* 1. **JSON Arrays**

JSON arrays are written inside square brackets. Just like JavaScript, a JSON array can contain multiple objects:

"employees":[  
    {"firstName":"John", "lastName":"Doe"},   
    {"firstName":"Anna", "lastName":"Smith"},   
    {"firstName":"Peter","lastName":"Jones"}  
]

In the example above, the object "employees" is an array containing three objects. Each object is a record of a person (with a first name and a last name).

1. **AVRO**

Avro is a data serialization system. Serialization is the process of converting the state information of an object instance into a binary or textual form to persist into storage medium or transported over a network.

* 1. **Avro provides:**
  + Rich data structures.
  + A compact, fast, binary data format.
  + A container file, to store persistent data.
  + Remote procedure call (RPC).
  + Simple integration with dynamic languages. Code generation is not required to read or write data files nor to use or implement RPC protocols. Code generation as an optional optimization, only worth implementing for statically typed languages
  + Relies on schemas. When Avro data is stored in a file, its schema is stored with it, so that files may be processed later by any program.
  + When Avro data is read, the schema used when writing it is always present. This permits each datum to be written with no per-value overheads, making serialization both fast and small.

C’’

* 1. **Avro advantages:**
  + Schema evolution – When Avro is used in RPC (remote procedure call), the client and server exchange requires schemas in the connection handshake when data is written or read. Most interesting is that you can use different schemas for serialization and deserialization, and Avro will handle the missing/extra/modified fields. Since both client and server both have the other's full schema.
  + Untagged data – two ways to encode data when serializing with Avro: binary or JSON.
    - Providing a schema with binary data allows each datum be written without overhead. The result is more compact data encoding, and faster data processing.
    - Providing schemas with JSON facilitates implementation in languages that already have JSON libraries
  + Dynamic typing – This refers to serialization and deserialization without code generation. It complements the code generation, which is available in Avro for statically typed languages as an optional optimization.
  1. **Avro links:**

<http://avro.apache.org/docs/1.3.0/spec.html#preamble>

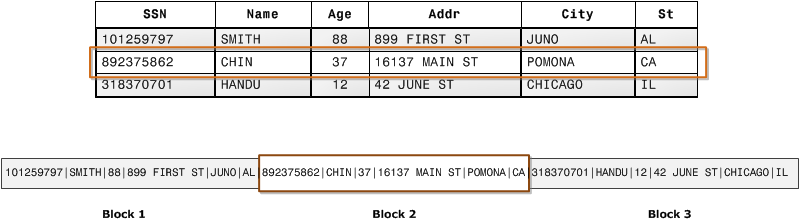
1. **Columnar Storage**

Columnar storage stores data in columns instead of rows. The goal of a columnar storage is to efficiently write and read data to and from hard disk storage in order to speed up the time it takes to return a query.

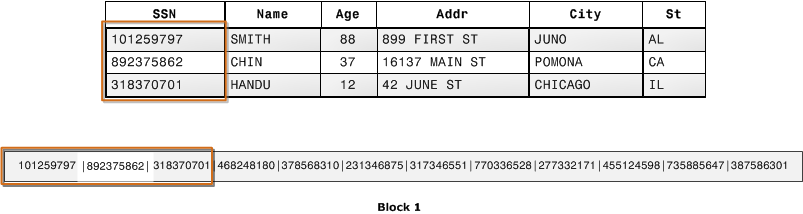
Benefits:

* data can be highly compressed. The compression permits columnar operations — like MIN, MAX, SUM, COUNT and AVG— to be performed very rapidly.
* a column-based format is self-indexing, it uses less disk space than a relational database management system (RDBMS) containing the same data.

Row storage:



Column storage:



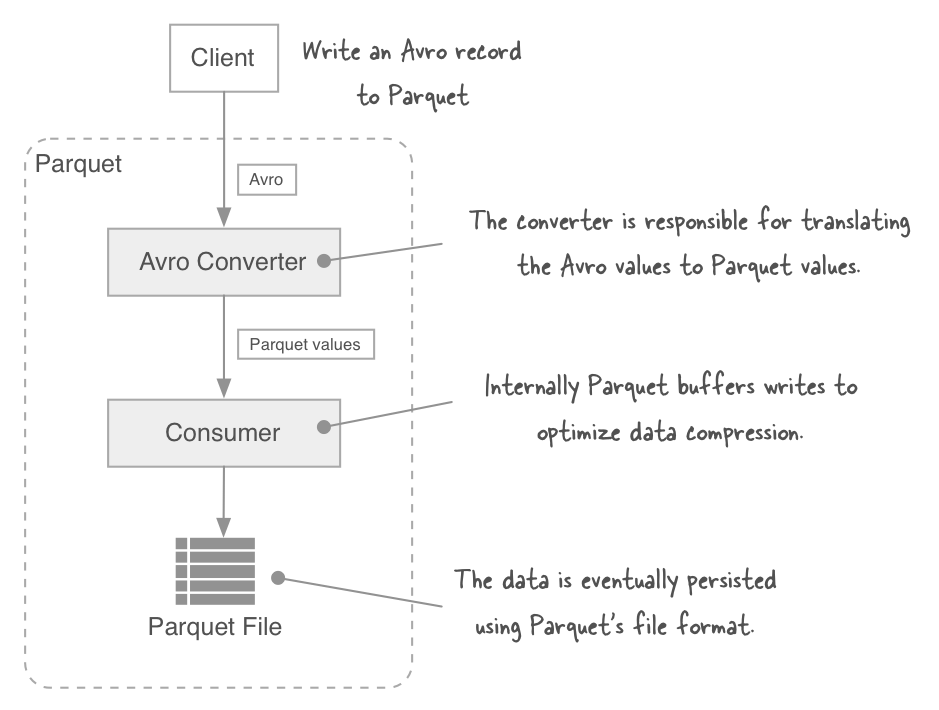
Further read:

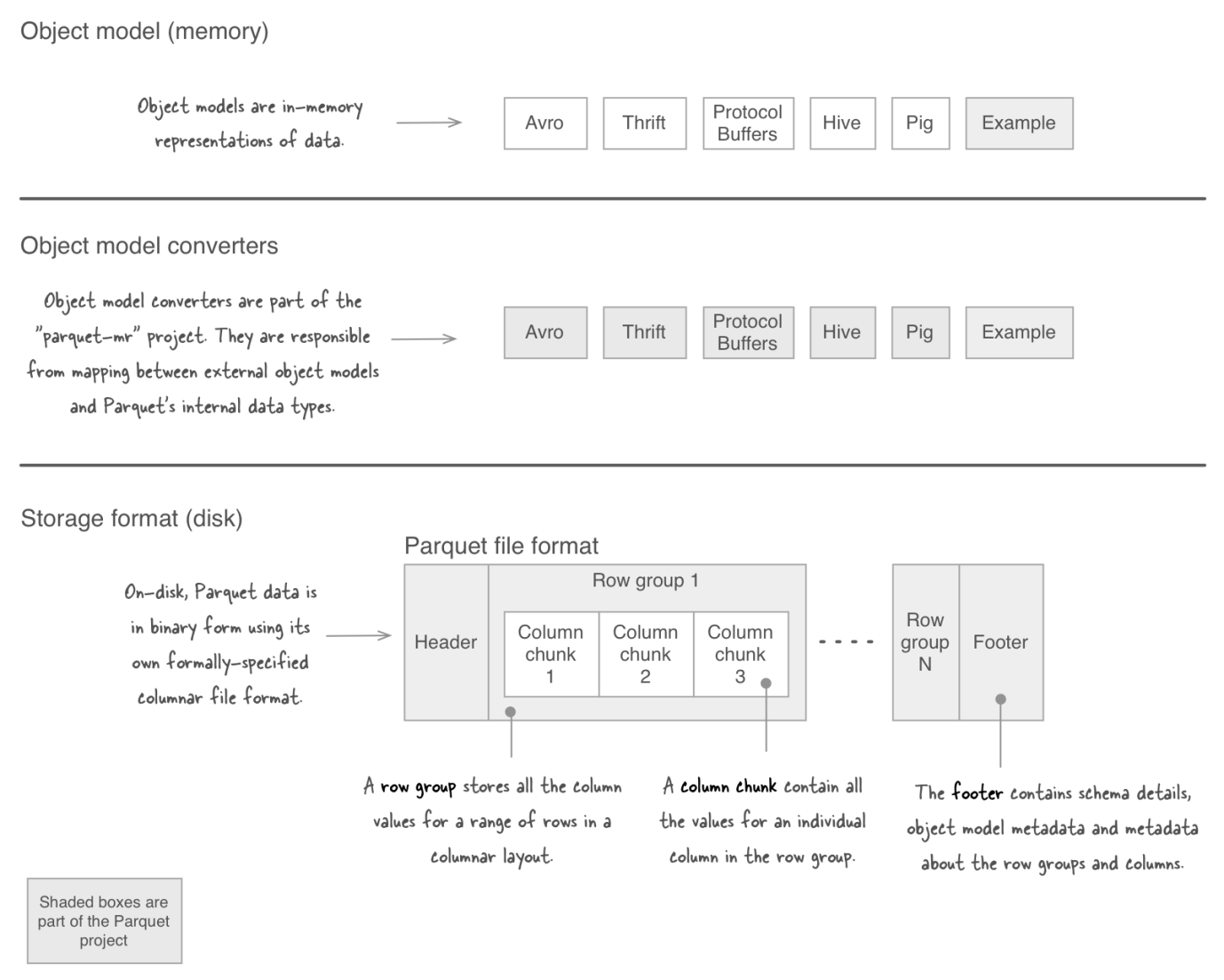
<http://db.csail.mit.edu/projects/cstore/abadi-sigmod08.pdf>

1. **Parquet**

Parquet is a columnar storage format that supports nested data. Parquet metadata is encoded using Apache Thrift. Parquet is built to support very efficient compression and encoding schemes. Parquet can work with MapReduce, Pig, Hive and Impala. It supports Avro, Thrift and Protocol Buffers. This is recommended file format in Cloudera bigdata stack.

* 1. **Parquet modules:**
* Parquet-format - contains format specifications and Thrift definitions of metadata required to properly read Parquet files
* Parquet-mr - contains multiple sub-modules, which implement the core components of reading and writing a nested, column-oriented data stream, map this core onto the parquet format, and provide Hadoop Input/Output Formats, Pig loaders, and other java-based utilities for interacting with Parquet
* Parquet-compatibility - contains compatibility tests that can be used to verify that implementations in different languages can read and write each other's files
  1. **Parquet Metadata:**
* file metadata, column (chunk) metadata and page header metadata. All thrift structures are serialized using the TCompactProtocol.
* Row group: A logical horizontal partitioning of the data into rows. There is no physical structure that is guaranteed for a row group. 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 live in a particular row group and is guaranteed to be contiguous in the file.
* Page: Column chunks are divided up into pages. A page is conceptually an indivisible unit (in terms of compression and encoding). There can be multiple page types which is interleaved in a column chunk.

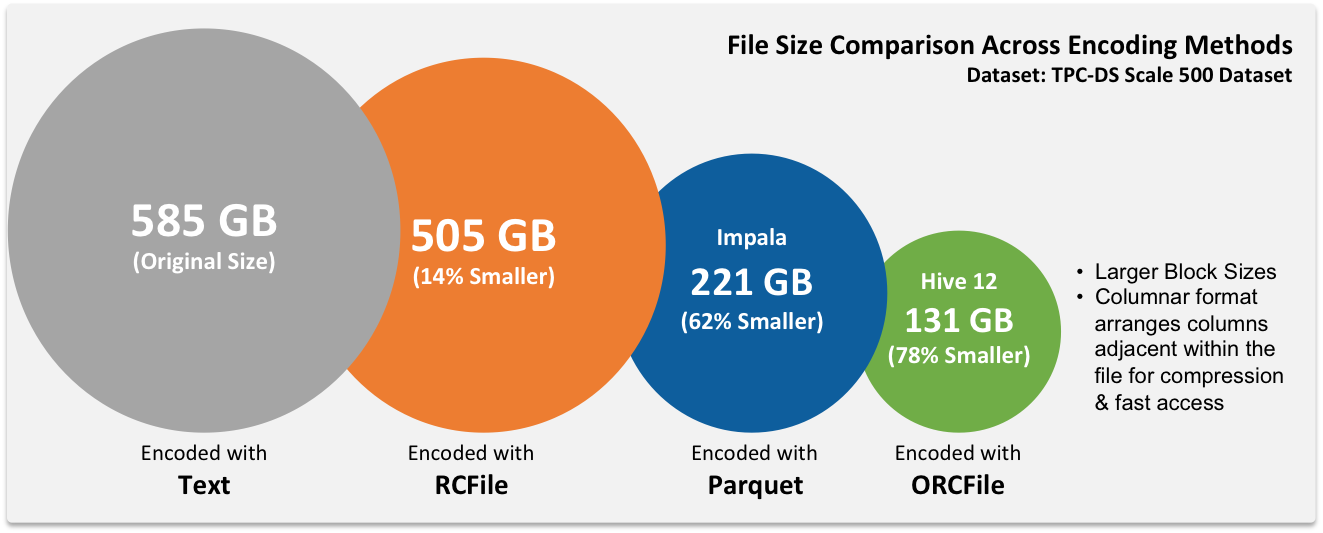




The Parquet file format incorporates several features that make it highly suited to data warehouse-style operations:

* Columnar storage layout. A query can examine and perform calculations on all values for a column while reading only a small fraction of the data from a data file or table.
* Flexible compression options. The data can be compressed with any of several codecs. Different data files can be compressed differently. The compression is transparent to applications that read the data files.
* Innovative encoding schemes. Sequences of identical, similar, or related data values can be represented in ways that save disk space and memory. The encoding schemes provide an extra level of space savings beyond the overall compression for each data file.
* Large file size. The layout of Parquet data files is optimized for queries that process large volumes of data, with individual files in the multi-megabyte or even gigabyte range.

Impala can create Parquet tables, insert data into them, convert data from other file formats to Parquet, and then perform SQL queries on the resulting data files. Parquet tables created by Impala can be accessed by Hive, and vice versa.



* 1. **Additional read:**

<https://github.com/Parquet/parquet-format>

Using the Parquet File Format with Impala Tables:

<http://www.cloudera.com/content/cloudera/en/documentation/cloudera-impala/v2-0-x/topics/impala_parquet.html>

<http://www.cloudera.com/content/cloudera/en/documentation/cdh5/v5-0-0/CDH5-Installation-Guide/cdh5ig_parquet.html>

Comparing sequential, ORC and Parquet files:

<http://appsintheopen.com/posts/43-comparing-sequence-files-orc-files-and-parquet-files>

Parquet compressed with Snappy and ORC compressed using gzip. Different compression techniques in Hadoop:

<http://comphadoop.weebly.com/>

1. **ORC – Optimized Row Columnar**

ORC file format provides a highly efficient way to store Hive data. It was designed to overcome limitations of the other Hive file formats. Using ORC files improves performance when Hive is reading, writing, and processing data. This is recommended file type in Hortonworks bigdata stack.

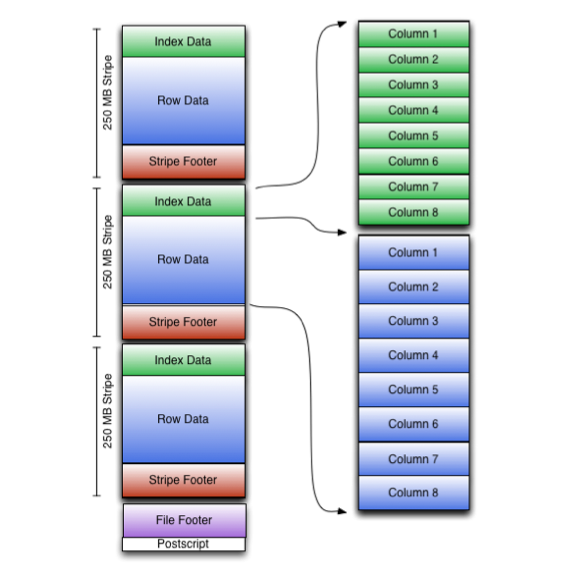
* 1. **ORC Structure:**

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.

This diagram illustrates the ORC file structure:



* 1. **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.

* 1. **Additional Read:**

<https://cwiki.apache.org/confluence/display/Hive/LanguageManual+ORC#LanguageManualORC-orc-spec>

1. **Checks of the files**

Given the different types of files, comes the challenge on how to ensure the file formats are getting translated into the required format as part of the development life cycle. The recommendation is to build a reusable parser to handle different files and convert into simpler formats that can then be used for validation. Also as part of BigData design cycle different file formats are used at different points. So different parsers would be needed at different check points for validations. These parsers needs to be parameterised so they can be easily configured for the project in hand and the file types used at a given stage of the project. The commonly used language that can be leveraged to build the parser – Java, Python, Pig, etc.