**Data Storage**

When you are architecting a solution to optimize (minimum space, fast processing) data storage. Main consideration includes

* File formats
* Compressions

**File-formats in Big Data**

Why use different file formats:

* To save storage.
* Fast processing
* Less time for I/o operations

Choices available on file formats:

1. Faster read
2. Faster writes
3. **Splitable (In big data we consider only splitable file formats only)**
   * Ability to split a file for processing multiple tasks to leverage the multiprocessing
4. Schema evolution support
5. Agnostic compression (file can be compressed with any compression)
6. Most for to compatible to platforms. Eg ORC is compatible with but not with others. Parquet is genialized file format which is compatible with many platforms like Spark etc.

**File Formats**

**Row based (**writing new records is easy)

* Faster writes
* Slower Reads (whole row of data from disk has be read in memory even if requirement is to be read only few columns)
* Less compressions

**Column Based** (for faster reads)

(Entire file is divided in several columns)

* Slower writes
* Efficient reads
* Good compression (bez data of same type is stored in a single column )

**Text File Formats (Human Readable files formats)**

CSV, XML, JSON

* As files are stored in plain text format. So, can’t be splittable. So, we can’t use these text file formats in big data.
* Everything is stored as a text. Even integer is also stored as a text. Which will cause huge space storage.

Ex. if integer 4569087 this will be stored in text format. It will take app **14 bytes** bez it is stored in Unicode format and same number stored in **integer** will take just **4 bytes.**

* Bez file size will be large. So, i/o operations will also be slowed.
* As everything is stored in text format. So, data calculations will be slower bez every time text has to convert into number formats for calculations.

**Specialized Big data file formats**

* All file formats are splitable
* All could be serialized
* Any compression could be used to all of them

(Compression code is kept in meta data of this file. So, a reader will get to know about this code from meta data.)

* Avro
* Orc
* Parquet

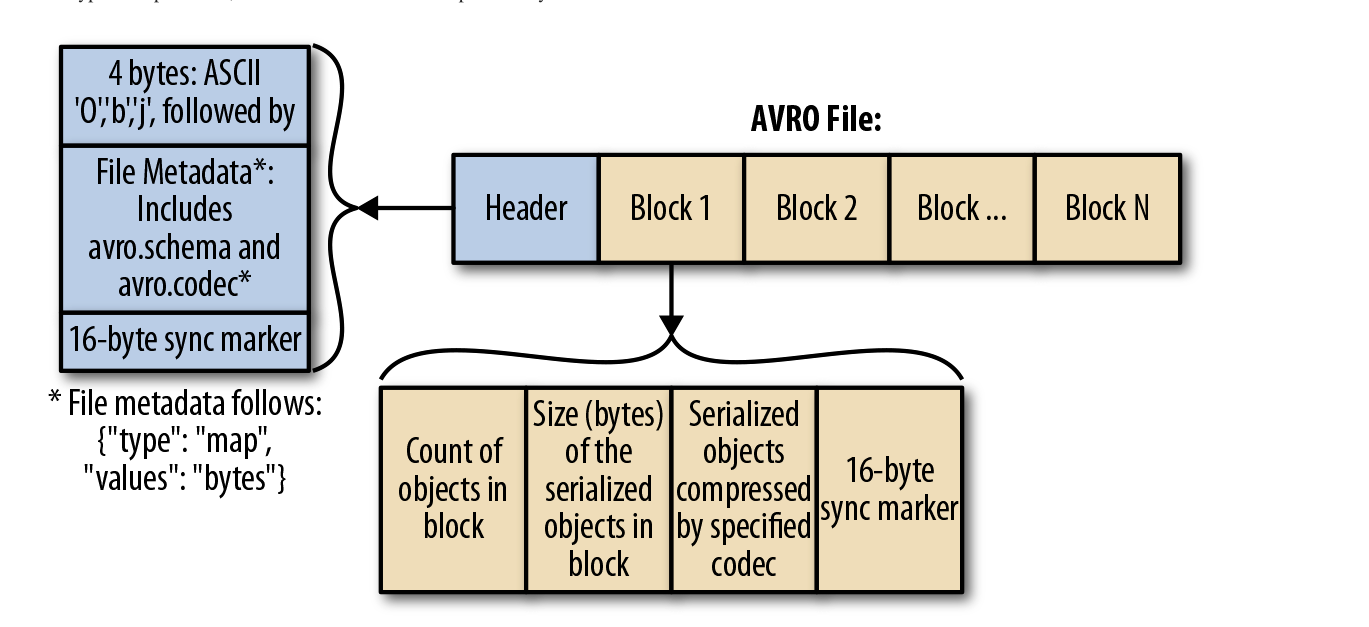
**Avro File format :**

**Best fit for :**

* Write heavy workloads
* When there is lot of schema change
* When you need to read entire record
* Row based file format which is widely used as a serialization platform.
* Avro stores the schema in json format making it easy to read and interpret by any program.
* Data stored in binary format making it compact and efficient.
* Language-neutral data serialization system. It can process many programming languages ex . Python, Java, C++, C ,C# etc.
* Best file format for schema evolution. Ex. Missing fields, add new fields , updates in existing fields.

**Avro is ideal for storing data in data lake landing zone because**

* + Data from landing zone is usually read as a whole for further processing by downstream systems (**So, row-based format is more efficient here in this case)**
  + Downstream system can easily retrieve table schemas form file (there is no need to store schemas separately in external meta data)
  + Any source schema change is easily handled. (schema evolution)
* Avro is choice for write-heavy workloads given it’s easy to append rows here.
* Avro stores the schema in the header, so data is self-describing.



**ORC(Optimized row columnar)**

ORC is row columnar (ORC) file format is a column based file format provides highly efficient way to store data in a compact form.

* Orc is the best file format for Hive,
* ORC supports all the data types including complex one too
* Orc support schema evolution too but not as good as Avro. Metadata in case of ORC is stored in protocol buffers which allow to add or delete fields.
* Drawback of orc file that it was designed for Hive , not general purpose storage format such as Java. However efforts taking place to make general purpose.

ORC provides lightweight, always on compression provided by type specific readers and writers such as below mentioned, which result in smaller files:

* Dictionary encoding
* Bit packing
* Delta encoding
* Run length encoding

ORC also supports these below compression to provide further storage efficiency.

* Zlib
* Lzo
* Snappy

**Dictionary encoding**

When you have multiple repeated values in a column. Ex if we have a table like this. Then a Key value pairs of repeated values will be created. And replace by values.

|  |  |  |
| --- | --- | --- |
| **Id** | **City** | **State** |
| 1 | Agra | UP |
| 2 | Lucknow | UP |
| 3 | Noida | UP |
| 4 | Bhopal | MP |

**States = {UP:1,MP:2}**

After this dictionary when encoding will applied. Then this is how data will look like. Which save lot of space

|  |  |  |
| --- | --- | --- |
| **Id** | **City** | **State** |
| 1 | Agra | 1 |
| 2 | Lucknow | 1 |
| 3 | Noida | 1 |
| 4 | Bhopal | 2 |

**Bit Packing-**

If take only that space which is required to store that specific data. Ex. a int data type takes 4 bytes but we to build numbers which are lesser that 128 only 1 byte. So, it will take just 1 byte only to store.

**Delta Encoding**

If we have incremental data then

Ex.1

|  |
| --- |
| Time |
| 12:40:00 00:00:00 |
| 1 |
| 2 |
| 1 |
| 3 |

After delta encoding

|  |
| --- |
| Time |
| 12:40:00 00:00:00 |
| 12:40:01 00:00:00 |
| 12:40:03 00:00:00 |
| 12:40:04 00:00:00 |
| 12:40:07 00:00:00 |

Ex.2

|  |
| --- |
| 68900 |
| 1 |
| 2 |
| 8 |
| 2 |

After delta encoding

|  |
| --- |
| 68900 |
| 98991 |
| 68993 |
| 68998 |
| 98700 |

**Run length Encoding –**

Wwwwssssgggtt run length encoding will convert this 1W3w4s3g2t

**Predicate Pushdown**

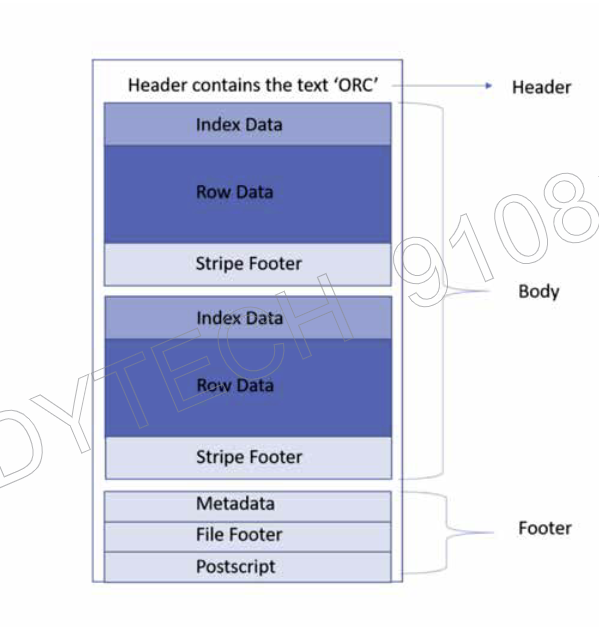
ORC allows predicate push down to the storage layer. So, that only required data is brought back to queries.

Select \* From employee id = 122345

Inside ‘Where’ clause whatever condition we mention are called predicates.

Pushdown means not all the record will be scanned to find employee instead only a very few records will be scanned.

Header: it explains that what type of file is

****

**Orc File**

stripe

body:

* Default size of stripe is 250MB. Stripe enables efficient reads.
* Stripe footer: contains the encoding of column.

Postscript hold compression parameters and provides a way to interpret rest of the file.

File Footer

* List of strips
* Number of rows in a stripe
* Each column data type
* Column level aggregate. Counts, min, max & sum.

Meta Data: (contains file level and strip level)

File level min :1 , max: 100,00,000

Stripe1 min:1, max:10,000

Stripe2 min:10,001, max:20,000

………………………………………………….

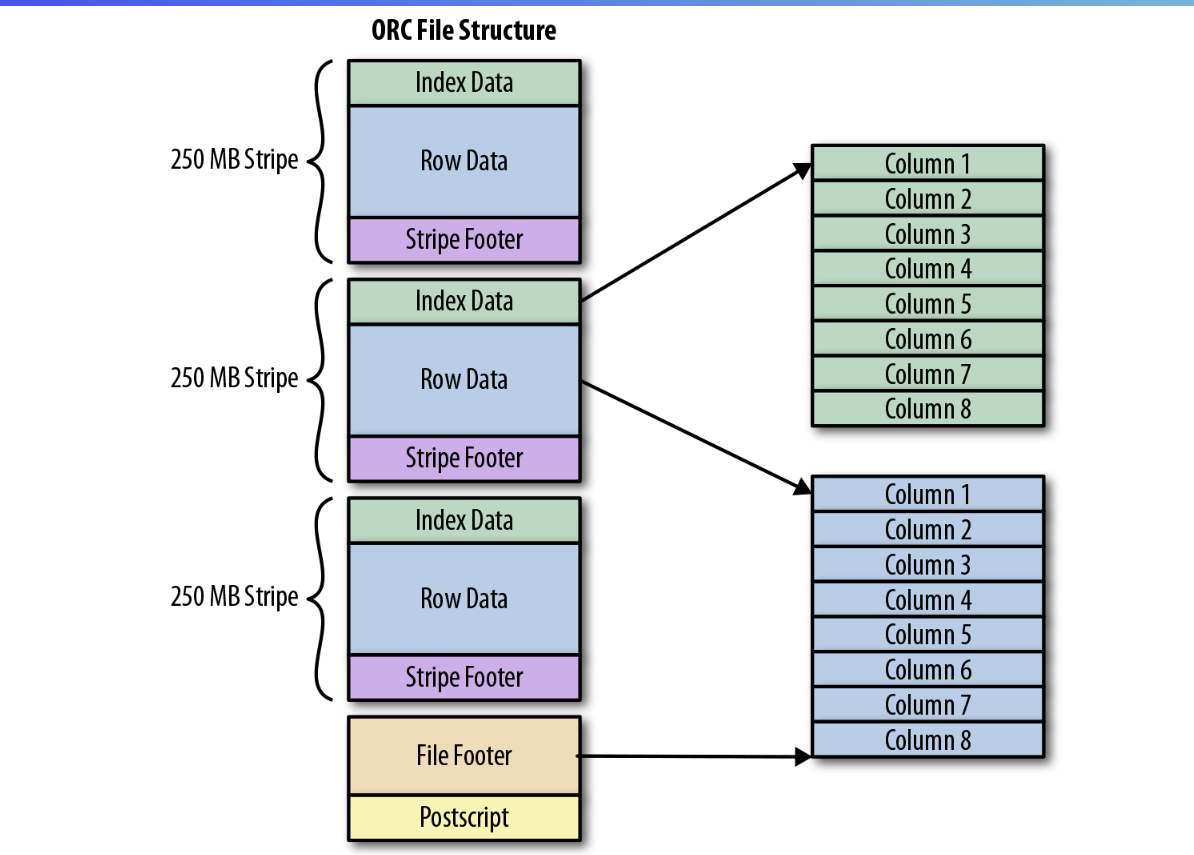
…………………………………………………..

Stripe10 min:91,00,001, max:10,00,000

stripe

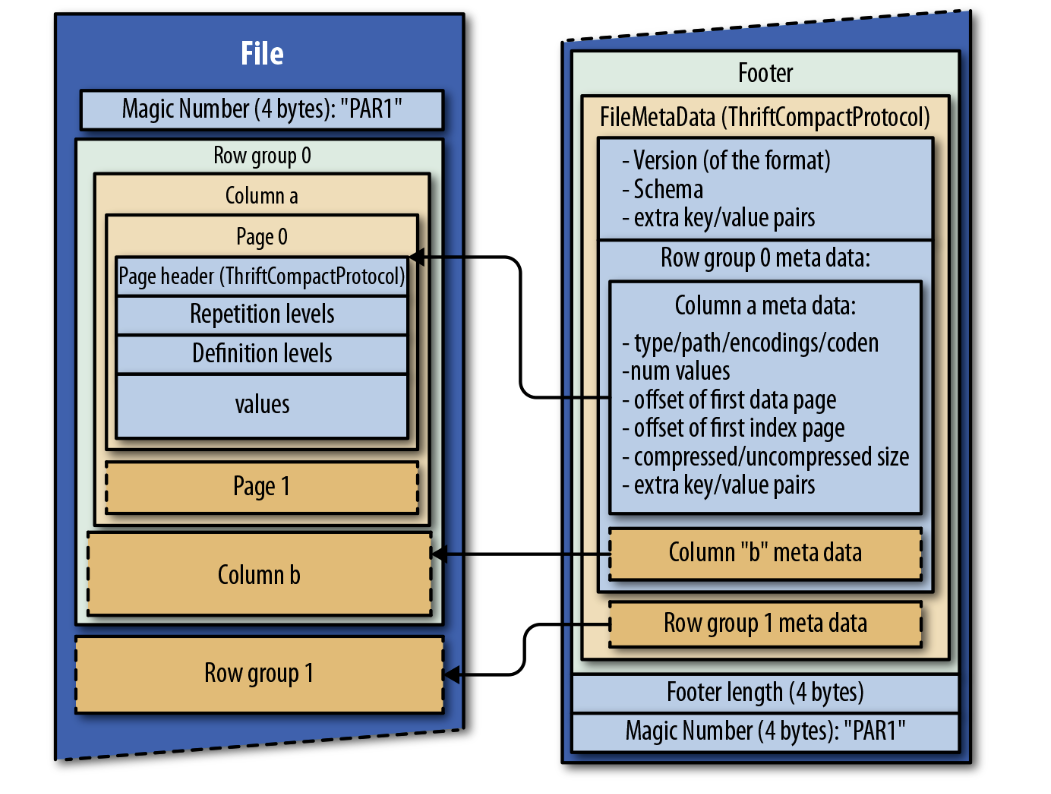
**Indexes:** ORC provides 3 level of indexes within each file:

* File level: Statistics of values in each column across the entire file
* Stripe level : Statistics of values in each column across the entire stripe.
* Row level: statistics about values in each column for each set of 10,000 rows within a stripe.

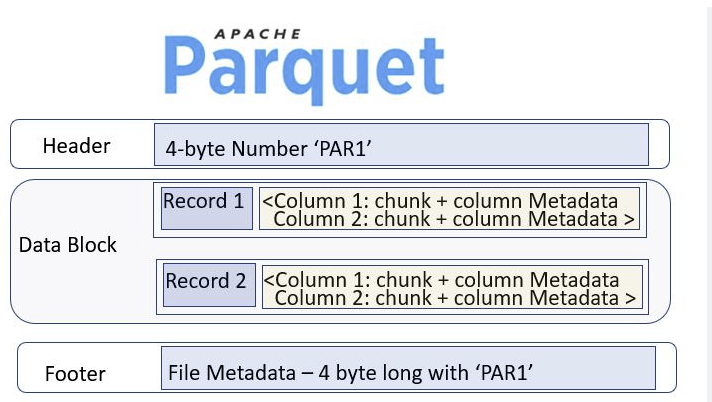


**Paraquet file formats**

* Very good for handling nested data
* Paraquet shares many design goals with ORC, but it is more general purpose. Compatible with many other platforms.
* Support meta data at the end of files. So, paraquet files are self-documenting.
* Use efficient and extensible encoding schemes like bit-packaging, run length etc.



**Parquet file format**



Chart, bubble chart

Description automatically generated

Best for nested data

Support Acid