LOAD BIG DATA EFFICIENTLY

PART 2: HOW DIFFERENT DATA FORMATS WORK

- *Recap performance results*
- *Row vs columnar formats*
- *How selected formats work*

{json}

Size, write and loading times

10 Million row dataset has been used

	Size	Monthly Costs
JSON	100 000 GB	2.000 €
AVRO	6 000 GB	120 €

Format	Size	Write time VS Code	Load time VS Code
JSON	1208 MB	6.6 s	11.7 s
CSV	593 MB	6.2 s	5.3 s
CSV with infer schema	593 MB	6.2 s	23.5 s
PARQUET	81.5 MB	5.4 s	1.5 s
AVRO	69.2 MB	2.5 s	2.3 s

{json}

vs.

AVRO™

- 94 % size
- 62 % write time
- 80 % load time

CSV

vs.

Parquet

- 86 % size
- 13 % write time
- 72 % load time

Smaller size and faster reads and writes save you also compute and storage costs

Load time for meta data and data load

Format	Load time Spark UI	of which meta data	of which actual load
JSON	11 s	4 s	7 s
CSV	5 s	45 ms	5 s
CSV with infer schema	23 s	20 ms + 11 s	12 s
PARQUET	1.1 s	98 ms	1 s
AVRO	2 s	0 s	2 s

Observations

- Avro and Parquet seem to be highly compressed and significant smaller than CSV and JSON
- Despite compression writes are up to 50 % faster, reads also significantly faster than CSV and JSON. Avro writes faster than parquet thow
- Parquet and Avro seem to provide the schema correctly. CSV and Json don‘t. Json seem not to understand timestamps.
- Parquet seem to contain meta data due to scanning activity in SQL and also a read job ahead
- CSV and JSON have a preliminary job interfering the schema
- Avro provides schema correctly without any preliminary job

Data format types

Unstructured

text, e.g. txt

videos, e.g. mp4

sound, e.g. mp3

pictures, e.g. png

Unorganised and unformatted information and data

Semi-structured

{json}

xml

Contains tags or markers that create some structure or semantic but not fully enforced

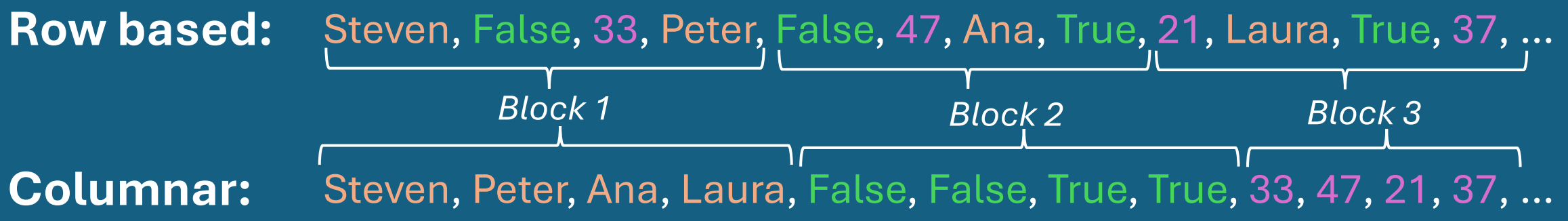
Structured

Enforce schema and data type rules, tabular format

Row-based vs columnar storage formats

Name	Female	Age
Steven	False	33
Peter	False	47
Ana	True	21
Laura	True	37

Saved on disk:



Row-based vs columnar storage formats

Name	female	Age
	False	
	False	
	True	
	True	

Block: Minimum amount of data on disk or a filesystem which can be written or read

Fragmented data is more difficult to read than physically close to each other

Row based: Steven, False, 33, Peter, False, 47, Ana, True, 21, Laura, True, 37, ...

Block 1 *Block 2* *Block 3*

Columnar: Steven, Peter, Ana, Laura, False, False, True, True, 33, 47, 21, 37, ...

Benefits and Usage

Row based: Steven, False, 33, Peter, False, 47, Ana, True, 21, Laura, True, 37, ...

- Write-heavy transactional workflows
- Loading all data per row
- Doing row based filtering

Columnar: Steven, Peter, Ana, Laura, False, False, True, True, 33, 47, 21, 37, ...

- Read-heavy (analytical) workflows
- Reading only one or part of the columns
- Aggregations or calculations like sum, max, min etc.

– Comma Separated Value

```
Age,Female,Name
33,false,Steven
47,false,Peter
21,true,Ana
37,true,Laura
```

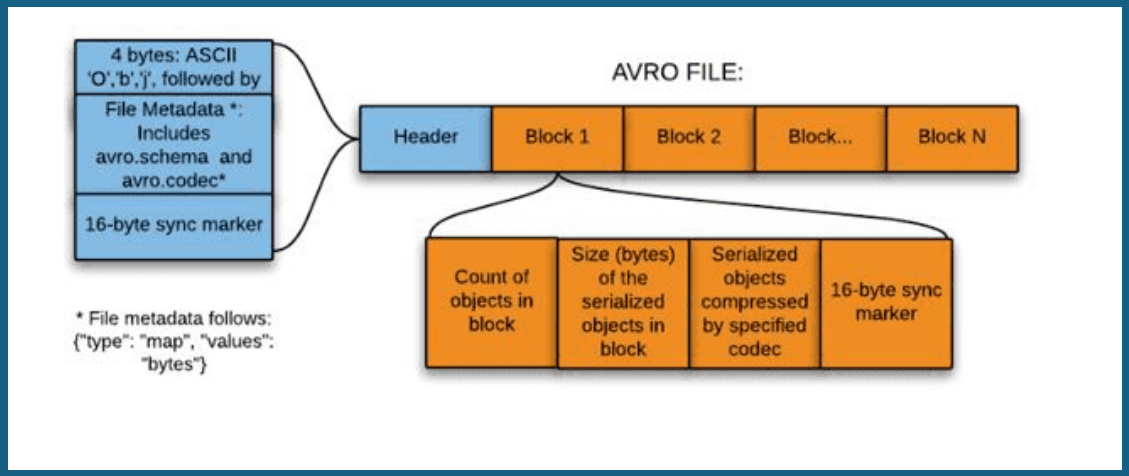
- Developed by IBM in 1972
- Row-based -> Fast when writing
- Human-Readable
- Flexible: e.g. separator or header
- No schema and non self describing
- Has compression options like gzip, snappy, etc. (not human readable anymore)
- Splitable when raw or depending compression mode
- High support e.g. even Excel

{json} – JavaScript Object Notation)

- Specified first in early 2000s
- Row-based -> Fast when writing
- Human Readable

```
{"Age":33,"Female":false,"Name":"Steven"}
{"Age":47,"Female":false,"Name":"Peter"}
{"Age":21,"Female":true,"Name":"Ana"}
{"Age":37,"Female":true,"Name":"Laura"}
```

- Self-Describing: contains information describing the data
  - Supports structures like strings, integers, objects, Booleans, nulls, arrays and describe the schema itself
  - Dates are not supported
  - Big size due to duplicate column names
- Has compression options like gzip, snappy, etc. (not human readable anymore)
- Splitable if raw and multiline option = false
- Often used as default response from REST APIs

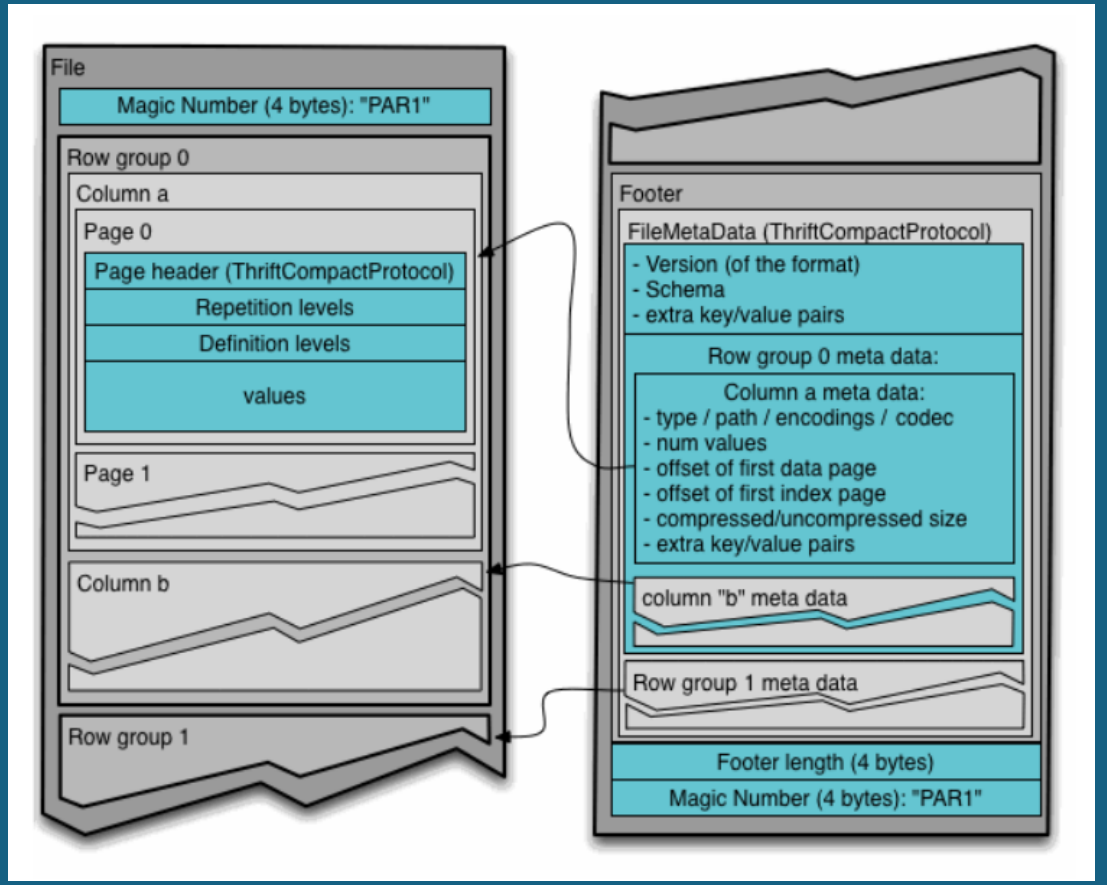


- Available since around 2009
- Row-based -> Fast when writing
- Compressible, Snappy as default and aiming high writing speed
- Binary (not human readable), data blocks can also be saved in JSON but slower and bigger size
- Allows complex data types
- Focus on minimal size and high write efficiency
- Schema (column names, type and null value) saved inside the file as JSON -> Thus self-describing and efficient for serialisation
- Schema evolution
- Splitable for parallel reads
- Often used for streaming e.g. with Apache Kafka
- Not always supported

- Launched 2013, developed by Cloudera and Twitter
- Allows complex data types
- Hybrid approach columnar but also row-level via so called row groups. Focus on efficient data queries using predicate pushdown
- Compressible, Snappy as default with the aim of high speed
- Optimized columnar storage and efficient compression and encoding makes it also fast in writing data and saving significant amount of size
- Schema (column names, type and null value) and other meta data are saved in the footer. Thus self-describing and efficient for serialisation
- Schema evolution
- Updates require a recreation
- Splitable for parallel reads

PARQUET

- Row Groups are a logical division on row level of a parquet defaulting to 128 MB
- Column part relates to column chunk of row groups
- Pages are invisible units where the encoding and compression happens
- Footer containing file metadata which can be used for predicate pushdown:
  - File level: num rows/ columns, schema
  - Row group: num rows/ columns
  - Column level: min, max, null count, distinct value counts



Summary

- CSV and JSON are both human readable, row level formats allowing fast writes and easy to open and process
- Compression is possible but not as fast as Avro and Parquet. It also limits the splitability
- CSV is further easy coruptable
- Avro is very efficient to save big amounts of data fast
- Parquet is very efficient in loading data including predicate pushdown but also writing data
- Avro and Parquet are not human readable and support is more limited especially for Avro