Apache Parquet

Open, standard, efficient columnar storage

https://parquet.apache.org/

Who I am

- Apache Parquet PMC (project management committee) member
- Apache Arrow PMC member
- CPython core developer
- Free / open source software expert
- Working at QuantStack (https://quantstack.net/)
- https://github.com/pitrou
- Contact me at antoine@python.org

Antoine Pitrou
pitrou

Overview

Open, standard, efficient ?

- A community project, under the rules of the Apache Software Foundation
  - Open source specification
  - Several open source implementations
- *De facto* standard (not *de jure*)
  - No similar file format comes close in popularity
  - (but CSV is still ubiquitous!)
- Efficient
  - Storage footprint
  - Read performance
  - Efficient querying

Parquet

Columnar ?

- Traditional DB systems (but also CSV!) are row-oriented
  - Good for row-wise operations and mutations
- Modern analytics systems use column-oriented storage
  - Dataframes, analytics databases, data lakes…

Name	Weight	Vitamins	Months
strawberry	10	{"c": 67}	["Apr", "May", "Jun"]
grapefruit	400	{"a": 110, "c": 26}	["Dec", "Jan", "Feb", "Mar"]
fig	50		["Aug", "Sep"]
banana	150	{"a": 148}	

Columnar ? (#2)

- Column-oriented storage is good for
  - Compression efficiency
  - Reading a subset of columns
  - Computations over many rows

Name	Weight	Vitamins	Months
strawberry	10	{"c": 67}	["Apr", "May", "Jun"]
grapefruit	400	{"a": 110, "c": 26}	["Dec", "Jan", "Feb", "Mar"]
fig	50		["Aug", "Sep"]
banana	150	{"a": 148}	

Features: a high-level view

- Rich data model
  - Arbitrarily nested data with explicit schema
  - Data types that reflect common database-y data (numbers, temporals...)
  - Omitted (NULLs) / repeated values (lists, potentially nested)
- Single-pass sequential writing ({S3, GCS...}-friendly)
- Random access reading
  - Parallelizable across columns, pages…
  - Can selectively read columns
  - Can selectively read data (statistics, bloom filters)
- Optional flexible encryption
- CSV has almost nothing of all this!

Parquet

Anatomy of a Parquet file

Parquet data model: the nested schema

- All data in a Parquet file conforms to a single schema
- Arbitrarily **nested**
- Each node can be **required/optional/repeated**
- Only **leaf nodes** (columns) have physical data
- Each column has a specific data type

```
REQUIRED BYTE_ARRAY name (STRING)
REQUIRED DOUBLE weight
OPTIONAL GROUP vitamins {
  OPTIONAL DOUBLE a
  OPTIONAL DOUBLE c
}
OPTIONAL GROUP months (LIST) {
  REPEATED GROUP list {
    REQUIRED BYTE_ARRAY element (STRING)
  }
}
```

Parquet data model: physical types

- Each column has a mandatory **physical type**
  - BOOLEAN
  - INT32
  - INT64
  - ~~INT96~~ (deprecated)
  - FLOAT
  - DOUBLE
  - FIXED_LEN_BYTE_ARRAY aka FLBA (parametric)
  - BYTE_ARRAY

Parquet data model: logical types (#1)

- Columns can optionally be annotated with a logical type
- Numerical logical types:

Logical type	Supported physical types	Parameters
IntType	INT32, INT64	Bit width, is signed
DecimalType	INT32, INT64, FLBA(*n*), BYTE_ARRAY	Scale, precision
Float16Type	FLBA(2)	
DateType	INT32	
TimeType	INT32, INT64	Time unit (ms/µs/ns), is UTC
TimestampType	INT64	Time unit (ms/µs/ns), is UTC

Parquet data model: logical types (#2)

- String/binary logical types:

Logical type	Supported physical types	Parameters
StringType	BYTE_ARRAY	
EnumType	BYTE_ARRAY	
UUIDType	FLBA(16)	
JsonType	BYTE_ARRAY	
BsonType	BYTE_ARRAY	

- Misc logical types:

Logical type	Supported physical types	Parameters
NullType	any	
ListType	only on group nodes	
MapType	only on group nodes	

Data model: definition levels

- How are **optional** values represented?

Vitamins
{"c": 67}
{"a": 110, "c": 26}
{"a": 148}

`vitamins.a`

Values
110
148

Def levels
1
2
0
2

`vitamins.c`

Values
67
26

Def levels
2
2
0
1

Data model: repetition levels

- How are **repeated** values represented?

Months
["Apr", "May"]
["Dec", "Jan", "Feb"]
["Aug"]

`months`

Values
Apr
May
Dec
Jan
Feb
Aug

Def levels
2
2
2
2
2
2
0

Rep levels
0
1
0
1
1
0
0

Encodings

- How are physical values and levels **actually** represented?

Encoding	Physical types
PLAIN	all except levels
RLE	levels, BOOLEAN
DELTA_BINARY_PACKED	INT32, INT64
DELTA_LENGTH_BYTE_ARRAY	BYTE_ARRAY, FLBA
DELTA_BYTE_ARRAY	BYTE_ARRAY, FLBA
RLE_DICTIONARY	all except levels
BYTE_STREAM_SPLIT	INT32, INT64, FLOAT, DOUBLE, FLBA

(note: this table omits deprecated encodings)

Parquet

Encodings : focus on RLE_DICTIONARY

- In real-world data, columns often have a relatively small cardinality
- RLE_DICTIONARY encodes unique values in a dictionary
  - Indices use a hybrid of bit-packing and run-length-encoding (called “RLE encoding”)

color
blue
red
red
red
green
red

Dictionary
blue
red
green

PLAIN encoding

Indices
0
1
1
1
2
1

RLE encoding

Parquet

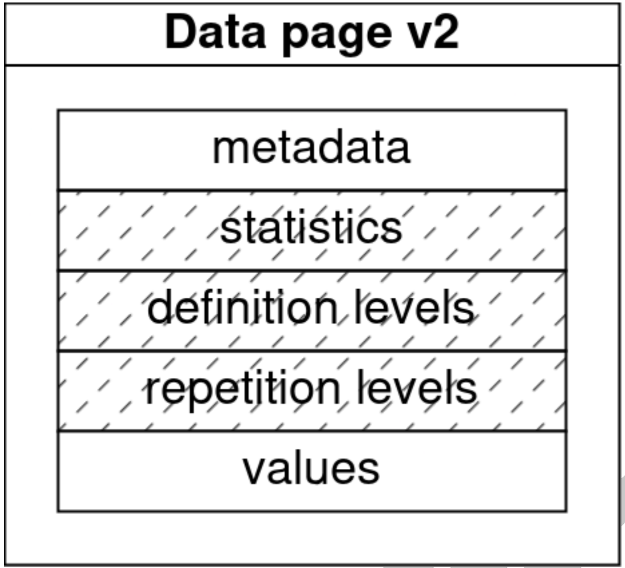
Compression

- Compression comes after encoding
  - Encoding step may improve compressibility (BYTE_STREAM_SPLIT)
- General-purpose compression codecs

Compression codec	Notes
UNCOMPRESSED	
GZIP	⚠ ubiquitous but under-performing
BROTLI	better than GZIP
SNAPPY	**widely used**, fast, moderately efficient
ZSTD	**state of the art**, fast and efficient
LZ4_RAW	**state of the art**, fastest
LZO	⚠ official library is GPL-licensed

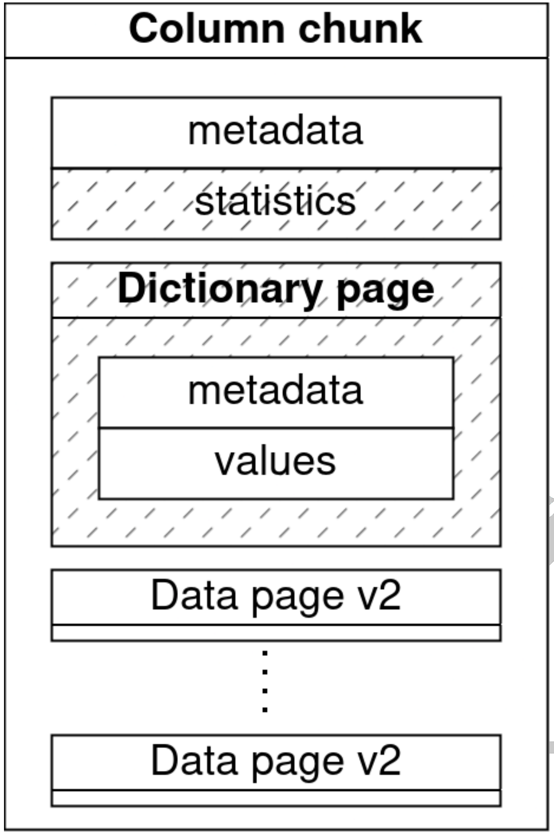
Anatomy of a file: data pages

- *Let's zoom out a bit...*
- Data pages are the smallest unit of work (encoding, compression)
- Actual size depends on data and writer configuration
- Typical data pages are both < 1MiB and < 20k rows



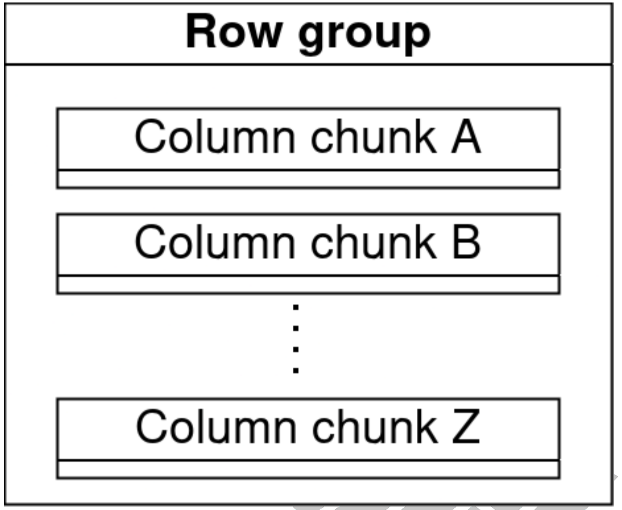
Anatomy of a file: column chunks

- A column chunk gathers many data pages of a given column
- Typical size is unbounded
- A single dictionary is shared at the column chunk level (for RLE_DICTIONARY)
- Data pages do not necessarily contain the same number of rows



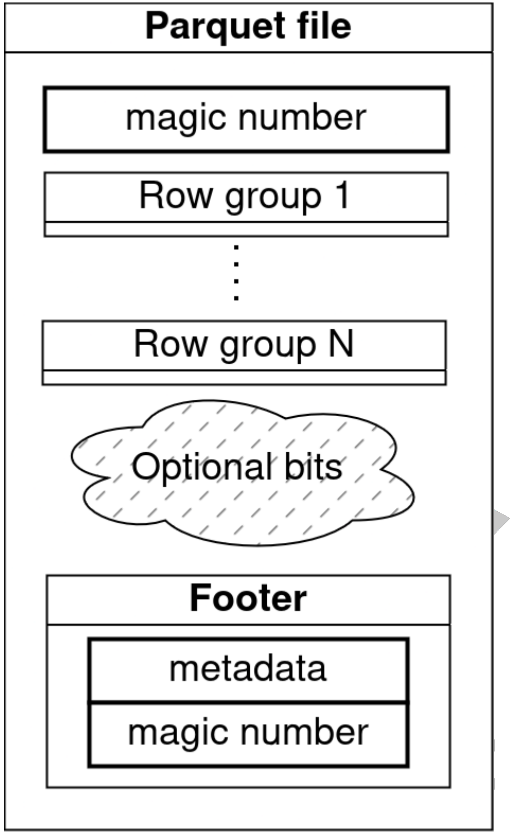
Anatomy of a file: row groups

- A row group contains **one** column chunk per physical column
- Typical size is unbounded (and can be very large)
- The number of row groups in a file varies from 1 to N (purely a writer decision)



Anatomy of a file: overall layout

- Writers can write this in a single sequential pass even if data is produced iteratively
  - No need to materialize all data at once in memory
  - Accumulate metadata and write it at the end
- Readers typically start by reading the footer
  - Reading footer metadata is on the critical path
  - Then random access into the file



Optional bits: the page index

- Goals:
  - Support page skipping / projection push-down given column predicate (`SELECT … WHERE 15 < weight < 30`)
  - Support indexed access (“give me row #10025”)
  - All while minimizing the number of I/Os (less seeking)
- Solution: two structures stored contiguously, per column, at the end of the file
  - The *offset index* allows direct navigation to data pages by row index
  - The *column index* stores statistics about data pages
    - Mainly min/max values (but also: null stats, def/rep levels histograms)
    - Efficiency is data-dependent (sortedness, clustering of values)
- “Speeding Up SELECT Queries with Parquet Page Indexes”, Zoltán Borók-Nagy and Gábor Szádovszky, Cloudera (https://chk.me/mOyDOeA)

Optional bits: Bloom filters

- Goal: allow data pruning for equality-based predicates
  (`SELECT … WHERE species = "cat"`)
- Solution: Bloom filters stored contiguously, column-wise, at the end of the file
  - One Bloom filter per column chunk (*not* data page)
  - A Bloom filter is a heuristic hash-based containment test
    - Two possible answers: "no" and "yes, perhaps"
  - Selectivity depends:
    1) Data cardinality: the more distinct values in a column chunk, the less selective
    2) Filter size: the larger the filter, the more selective
  - A well-known formula exists to choose filter size based on desired selectivity
- "Using Parquet's Bloom Filters", Trevor Hilton, InfluxData
  (https://chk.me/1UF79nd)

Encryption

- Optional whole-file encryption
  1) Ensure confidentiality
  2) Protect against tampering
- Individual file components (“modules”) are encrypted independently
  - Preserving full Parquet capabilities (random access, column selections, projection push-down...)
- Symmetric encryption only (AES GCM or AES CTR)
- Optional per-column keys, for more granular access control
- Key management is out of scope for the Parquet format
  - Implementations typically provide several strategies
- “Big data security in Apache projects”, Gidon Gershinsky (https://chk.me/FCKdUhF)

Ecosystem

Implementations

- Main open source implementations
  - Java (previously known as “parquet-mr”)
    - https://github.com/apache/parquet-java/
  - C++, a component of Arrow C++
    - Bindings to Python (PyArrow), Ruby, R…
    - https://arrow.apache.org/
  - Rust, a component of Arrow Rust
    - https://docs.rs/parquet/
- GPU implementation in cuDF
- An unknown number of proprietary / in-house implementations

Availability and support

- Parquet supported by a number of libraries, execution engines, services
  - Open source: DuckDB, Spark, Pandas, Dask, Iceberg...
    - “pg_parquet: An Extension to Connect Postgres and Parquet”, Craig Kerstiens, CrunchyData (https://chk.me/cjWw9OW)
  - Closed source: too many to name
- Domain-specific communities, such as GeoParquet

Present and future

Limitations

- Many features, not all of them supported by all implementations
  - LZ4_RAW, BYTE_STREAM_SPLIT, Bloom filters…
  - Writers are usually conservative
  - Enable features according to target user base when writing
- Metadata serialization (Thrift) inefficient with very wide schemas (thousands of columns)
- No random access *inside* data pages
  - Must decode/decompress whole page
- Not adapted to very large binary values (such as images)

Parquet

Alternatives

- Apache ORC
  - Similar characteristics as Parquet
  - Different technical choices, but efficiency roughly the same
  - Smaller ecosystem
- Lance v2
  - Innovative, extensible, but very young
  - Designed for the constraints of AI workloads
  - “Lance v2: A columnar container format for modern data”, Weston Pace, LanceDB (https://chk.me/JoJiMVF)
  - “Nimble and Lance: The Parquet Killers”, Chris Riccomini (https://chk.me/DKtvczc)

Parquet

Present and future

- Parquet is still being actively developed
  - Latest format spec release is 2.11.0 (November 2023)
- New Variant type (from Spark and Iceberg)
  - Efficient representation of semi-structured / dynamically typed data
- Discussions around a new metadata serialization format
  - Using Flatbuffers rather than Thrift
  - Much better efficiency on very wide schemas
  - Maintaining compatibility with older readers

Parquet

Discussion