Apache Parquet

a column-oriented file format

Sebastian Nagel

OSCAR – Common Crawl – Collab
2022-04-22

A columnar file format?

CSV, XML, JSON

- verbose
- interchangeable
- slow reading (parsing)

Database

- binary, efficient
- fast reading (querying)
- “captive” / silo-dependent

➥ interchangeable but efficient format

- binary, fast (de)serialization
- fully specified and documented, defined semantics
- system and language independent API
- optimized for common access patterns

Row-major vs. Column-major

- choice by access patterns
  - read/write entire row at once
  - need to process just a few columns
  - SQL-like aggregations
- row-major file or serialization formats
  - CSV, XML, JSON
  - protobuf, Thrift – interface definition and compiler
  - Avro – API to read/write data using external schema, container file format
- column-major formats
  - Parquet, ORC (file formats)
  - Arrow (in-memory columnar)

a columnar storage and file format,
started 2013 by Twitter and Cloudera

- format specification (using Thrift)
- guaranteed backward / defined forward compatibility
- integrated schema (self-describing)
- hierarchical (nested) schema
- API for C++ and Java
  (used as base for more language bindings)

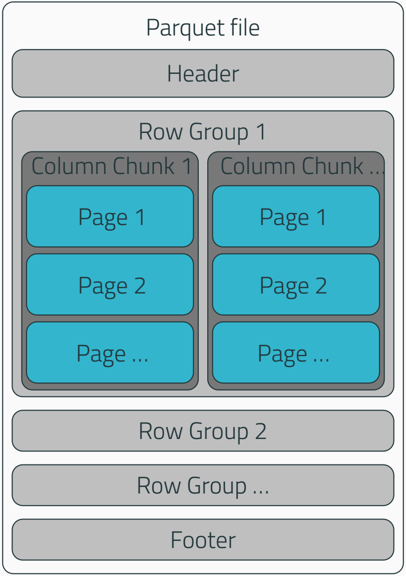
Parquet – Columns

- binary types: boolean, int32/64, float/double, byte array
- more logical types: signed/unsigned, date and time, ...
- field values: required, optional, repeated
- per-column compression

  snappy, gzip, lzo, brotli, lz4, zstd
- encoding

  plain, dictionary, run-length/bit-packed, delta

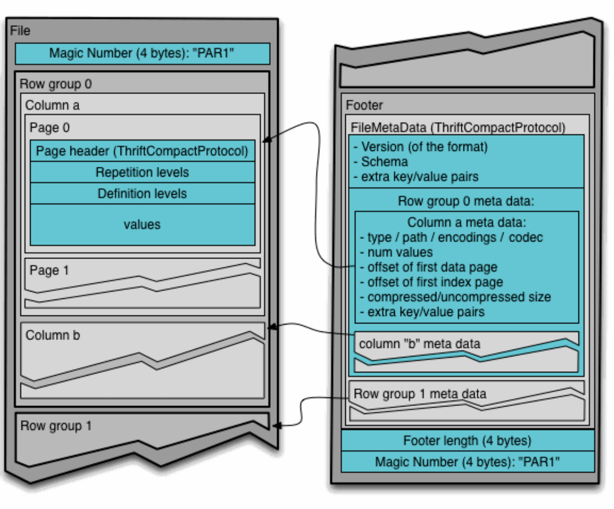
Parquet – File Layout



A column chunk is split into “pages”. Pages store the column values using a suitable encoding (plain, dictionary, run-length/bit-packed, delta). With dictionary encoding all values are hold in a separate dictionary page to speed up look-ups and filtering.

Pages are optionally compressed (gzip, zstd, etc.). Page-level compression (same as per-record WARC compression) allows to read only the requested pages.

Parquet – File Metadata



Parquet – Nested Encoding

```
DocId: 10                    r1
Links
  Forward: 20
  Forward: 40
  Forward: 60
Name
  Language
    Code: 'en-us'
    Country: 'us'
  Language
    Code: 'en'
  Url: 'http://A'
Name
  Url: 'http://B'
Name
  Language
    Code: 'en-gb'
    Country: 'gb'
```

```
message Document {
  required int64 DocId;
  optional group Links {
    repeated int64 Backward;
    repeated int64 Forward; }
  repeated group Name {
    repeated group Language {
      required string Code;
      optional string Country; }
    optional string Url; }}
```

```
DocId: 20                    r2
Links
  Backward: 10
  Backward: 30
  Forward:  80
Name
  Url: 'http://C'
```

DocId

value	r	d
10	0	0
20	0	0

Name.Url

value	r	d
http://A	0	2
http://B	1	2
NULL	1	1
http://C	0	2

Links.Forward

value	r	d
20	0	2
40	1	2
60	1	2
80	0	2

Links.Backward

value	r	d
NULL	0	1
10	0	2
30	1	2

Name.Language.Code

value	r	d
en-us	0	2
en	2	2
NULL	1	1
en-gb	1	2
NULL	0	1

Name.Language.Country

value	r	d
us	0	3
NULL	2	2
NULL	1	1
gb	1	3
NULL	0	1

repetition and definition levels used to represent

- optional or repeated values
- and nested columns

cf. Dremel paper (2010) and Dremel Made Simple with Parquet

Caveat: tools may not fully support nested columns

Parquet – Parallelization

levels of parallelization

- pages – compression and encoding
- column chunks – I/O
- row groups and files – task (MapReduce, Spark, etc.)
- partitions (not part of the Parquet spec)

```
data/
  year=2017/
    month=01/2017-01_file1.parquet
             2017-01_file2.parquet
             ...
    month=02/2017-02_file1.parquet
```

  - cheap selection on partition "column"
  - allows to add data continuously

Parquet – Performance Optimization

- read only what you need
- partitions
- columns (projection push down, column pruning)
- column chunks fitting filter conditions (predicate push down)
  - use column stats (per file or column chunk) for filtering (requires that are sorted or partially homogeneous)
  - (Cloud) apply filters on the storage nodes

Apache ORC

ORC (Optimized Row Columnar) format, started 2013 by Hortonworks and Facebook as part of the Hive project, since 2015 an independent Apache project

- very similar to Parquet
- different community (maybe smaller)
- may introduce new features first
  - bloom filters
  - column encryption
  - (both later added to Parquet)

Apache Arrow

in-memory columnar data

- cross-language – Java and C++: Python, Ruby, Rust, Go, Node.js
- API to read Parquet (also ORC) files into memory and process the data
- memory layout optimized for SIMD operations
- utilized by Spark, Pandas, and other projects

Columnar URL Index for Common Crawl

- motivation: existing URL index (wayback machine) was heavily loaded
  - optimized to fetch by URL or domain
  - post-filtering eg. by MIME type
- columnar storage format much better for analytical queries and aggregations
- efficient querying with SparkSQL and Presto / AWS Athena

Columnar URL Index for Common Crawl

- existing index files (JSON) are converted to Parquet and enriched by SparkSQL
- crawls 2013 – Jan 2022
  - 250 billion rows (web page captures)
  - 60,000 Parquet files (total 20 TiB) on AWS S3
- benchmarked storage size and performance on Athena (in 2018)
  - flat schema wins over nested (querying)
  - gzip over snappy (storage size and querying)
  - Parquet and ORC are very close but differ significantly from query to query

Example 1: Querying Columnar URL Index with Athena / Presto

```sql
--      at least one language code in the URL path
--
-- The idea was taken from
--   - Resnik/Smith 2003: The Web as a Parallel Corpus,
--     http://www.aclweb.org/anthology/J03-3002.pdf
--   - Buck 2015: Corpus Acquisition from the Interwebs,
--     http://mt-class.org/jhu-2015/slides/lecture-crawling.pdf
--
SELECT url_host_registered_domain AS domain,
       COUNT(DISTINCT(url_path_lang)) as n_lang,
       COUNT(*) as n_pages,
       histogram(url_path_lang) as lang_counts
FROM "ccindex"."ccindex",
  UNNEST(regexp_extract_all(url_path, '(?<=/)(?:[a-z][a-z])(?=/)')) AS t (url_path_lang)
WHERE crawl = 'CC-MAIN-2018-05'
  AND subset = 'warc'
  AND url_host_registry_suffix = 'va'
GROUP BY url_host_registered_domain
HAVING COUNT(*) >= 100
   AND COUNT(DISTINCT(url_path_lang)) >= 1
ORDER BY n_pages DESC;
```

Run Query | Save As | Format query | New Query (Run time: 5.79 seconds, Data scanned: 16.07MB)

Results

	domain	n_lang	n_pages	lang_counts
1	vatican.va	40	42795	{de=3147, fi=3, ru=20, be=1, pt=4036, bg=11, lt=1, hr=395, fr=5677, hu=79, uc=2, u
2	iubilaeummisericordiae.va	7	2916	{de=445, pt=273, en=454, it=542, pl=168, fr=422, es=612}

Example 2: WARC Storage Occupied per MIME Type

Common Crawl tries to crawl only HTML pages without page dependencies (images, CSS, JavaScript). However, a small percentage of non-HTML content is accepted to obtain a broad sample of document formats used on the web.

The issue with PDF documents, images and other non-HTML formats is that they tend to occupy more storage in WARC archives. But which formats at which scale?

```sql
-- average length and occupied storage of WARC records by MIME type
SELECT COUNT(*)                                                AS pages,
       round(COUNT(*)*100.0/SUM(COUNT(*)) OVER(), 3) AS perc_pages,
       round(AVG(warc_record_length)/power(2,10), 0) AS avg_rec_kB,
       round(SUM(warc_record_length)/power(2,40), 3) AS storage_TB,
       round(SUM(warc_record_length) * 100.0
          / SUM(SUM(warc_record_length)) OVER(), 3) AS perc_storage,
       content_mime_detected
FROM "ccindex"
WHERE crawl = 'CC-MAIN-2019-22' -- May 2019
  AND subset = 'warc'           -- only successful fetches
GROUP BY content_mime_detected
ORDER BY storage_TB DESC, n_pages DESC;
```

Example 2: WARC Storage Occupied per MIME Type

The SQL query above aggregates the WARC record length by the detected MIME type and calculates average and total sum. The result is sorted by the amount of occupied storage:

pages	%	avg.rec. kiB	storage TiB	%	MIME type
2033659795	75.890	17	32.012	65.019	text/html
605403020	22.592	15	8.290	16.837	application/xhtml+xml
19423997	0.725	388	7.014	14.246	application/pdf
4158147	0.155	257	0.997	2.024	image/jpeg
166558	0.006	885	0.137	0.279	audio/mpeg
633587	0.024	225	0.133	0.270	image/png
181213	0.007	484	0.082	0.166	application/zip
3944276	0.147	10	0.036	0.074	application/rss+xml
43070	0.002	847	0.034	0.069	video/mp4
42868	0.002	802	0.032	0.065	audio/mp4
38406	0.001	902	0.032	0.066	appl./vnd.android.package-archive
54795	0.002	499	0.025	0.052	application/epub+zip

Although the May 2019 dataset includes only 0.7% PDF files, these account for 7 TiB or 14% of the total storage. To minimize the storage usage we decided to increase the revisit frequency for storage-intensive formats.

Columnar URL Index – SparkSQL Query Plan

```
18/06/27 13:20:57 INFO CCIndexExport: Executing query SELECT COUNT(*) as n_pages,
       COUNT(*) * 100.0 / SUM(COUNT(*)) OVER() as perc_pages,
       AVG(warc_record_length) as avg_warc_record_length,
       SUM(warc_record_length) as sum_warc_record_length,
       SUM(warc_record_length) * 100.0 / SUM(SUM(warc_record_length)) OVER() as perc_warc_storage,
       content_mime_detected
FROM ccindex
WHERE subset = 'warc'
GROUP BY content_mime_detected
ORDER BY n_pages DESC:
18/06/27 13:20:57 INFO FileSourceStrategy: Pruning directories with: isnotnull(subset#26),(subset#26 = warc)
18/06/27 13:20:57 INFO FileSourceStrategy: Post-Scan Filters:
18/06/27 13:20:57 INFO FileSourceStrategy: Output Data Schema: struct<content_mime_detected: string, warc_record_leng
18/06/27 13:20:57 INFO FileSourceScanExec: Pushed Filters:
18/06/27 13:20:57 INFO InMemoryFileIndex: Selected 1 partitions out of 3, pruned 66.66666666666667% partitions.
18/06/27 13:20:57 WARN WindowExec: No Partition Defined for Window operation! Moving all data to a single partition,
== Physical Plan ==
*Sort [n_pages#56L DESC NULLS LAST], true, 0
+- *Project [n_pages#56L, CheckOverflow((CheckOverflow((cast(cast(_w0#93L as decimal(20,0)) as decimal(21,1)) * 100.0
   +- Window [sum(_w1#94L) windowspecdefinition(ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) AS _we0#97L
      +- Exchange SinglePartition
         +- *HashAggregate(keys=[content_mime_detected#20], functions=[count(1), avg(cast(warc_record_length#23 as bi
            +- Exchange hashpartitioning(content_mime_detected#20, 200)
               +- *HashAggregate(keys=[content_mime_detected#20], functions=[partial_count(1), partial_avg(cast(warc_
                  +- *Project [content_mime_detected#20, warc_record_length#23]
                     +- *FileScan parquet [content_mime_detected#20,warc_record_length#23,crawl#25,subset#26] Batched
```

Summary and Conclusion

- columnar data formats (Parquet or ORC) a good option if
  - big data
  - column-major access patterns
  - (not too) frequent reads, infrequent writes
- Parquet and Arrow good example how to build reusable software components
  - format specification > API (C++ and Java)
  - standardized data representation
    on disk (Parquet) > in memory (Arrow)

Apache Parquet Documentation

2017 dataengineeringpodcast.com – Data Serialization Formats with Doug Cutting and Julien Le Dem

2018 Julien Le Dem, The columnar roadmap: Apache Parquet and Apache Arrow

2018 Owen O'Malley, Fast Access To Your Complex Data - Avro, JSON, ORC, and Parquet [video]

2022 Cost Efficiency @ Scale in Big Data File Format

Questions?