Parquet performance tuning: The missing guide

Ryan Blue
Strata + Hadoop World  NY 2016
Contents.

- Big data at Netflix
- Parquet format background
- Optimization basics
- Stats and dictionary filtering
- Format 2 and compression
- Future work
Big data at Netflix.
Big data at Netflix.

- 600B Events
- 40+ PB DW
- Read 3PB
- Write 300TB
Strata San Jose results.
Metrics dataset.

Based on Atlas, Netflix’s telemetry platform.

- Performance monitoring backend and UI

Example metrics data.

- Partitioned by **day**, and **cluster**
- Columns include metric **time**, **name**, **value**, and **host**
- Measurements for each minute are stored in a Parquet table
Parquet format background.
Parquet data layout.

ROW GROUPS.
- Data needed for a group of rows to be reassembled
- Smallest task or input split size
- Made of COLUMN CHUNKS

COLUMN CHUNKS.
- Contiguous data for a single column
- Made of DATA PAGES and an optional DICTIONARY PAGE

DATA PAGES.
- Encoded and compressed runs of values
Row groups.
Column chunks and pages.
Read less data.

**Columnar organization.**
- Encoding: make the data smaller
- Column projection: read only the columns you need

**Row group filtering.**
- Use *footer stats* to eliminate row groups
- Use *dictionary pages* to eliminate row groups

**Page filtering.**
- Use *page stats* to eliminate pages
Basics.
Setup.

**Parquet writes:**
- Version 1.8.1 or later – includes fix for incorrect statistics, PARQUET-251
- 1.9.0 due in October

**Reads:**
- Presto: Used 0.139
- Spark: Used version 1.6.1 reading from Hive
- Pig: Used parquet-pig 1.9.0 for predicate push-down
Pig configuration.

-- enable pushdown/filtering
set parquet.pig.predicate.pushdown.enable true;

-- enables stats and dictionary filtering
set parquet.filter.statistics.enabled true;
set parquet.filter.dictionary.enabled true;
Spark configuration.

// turn on Parquet push-down, stats filtering, and dictionary filtering
sqlContext.setConf("parquet.filter.statistics.enabled", "true")
sqlContext.setConf("parquet.filter.dictionary.enabled", "true")
sqlContext.setConf("spark.sql.parquet.filterPushdown", "true")

// use the non-Hive read path
sqlContext.setConf("spark.sql.hive.convertMetastoreParquet", "true")

// turn off schema merging, which turns off push-down
sqlContext.setConf("spark.sql.parquet.mergeSchema", "false")
sqlContext.setConf("spark.sql.hive.convertMetastoreParquet.mergeSchema", "false")
Writing the data.

**Spark:**
```
sqlContext
  .table("raw_metrics")
  .write.insertInto("metrics")
```

**Pig:**
```
metricsData = LOAD 'raw_metrics'
  USING SomeLoader;
STORE metricsData INTO 'metrics'
  USING ParquetStorer;
```
Writing the data.

Spark:
sqlContext
  .table("raw_metrics")
  .write.insertInto("metrics")

Pig:
metricsData = LOAD 'raw_metrics'
  USING SomeLoader;
STORE metricsData INTO 'metrics'
  USING ParquetStorer;

OutOfMemoryError
or
ParquetRuntimeException
Writing too many files.

Data doesn’t match partitioning.
- Tasks write a file per partition

Symptoms:
- OutOfMemoryError
- ParquetRuntimeException: New Memory allocation 1047284 bytes is smaller than the minimum allocation size of 1048576 bytes.
- Successfully write lots of small files, slow split planning
Account for partitioning.

Spark.
sqlContext
  .table("raw_metrics")
  .sort("day", "cluster")
  .write.insertInto("metrics")

Pig.
metrics = LOAD 'raw_metrics'
  USING SomeLoader;
metricsSorted = ORDER metrics
  BY day, cluster;
STORE metricsSorted INTO 'metrics'
  USING ParquetStorer;
Filter to select partitions.

**Spark.**

```scala
val partition = sqlContext
  .table("metrics")
  .filter("day = 20160929")
  .filter("cluster = 'emr_adhoc'")
```

**Pig.**

```pig
metricsData = LOAD 'metrics'
  USING ParquetLoader;
partition = FILTER metricsData BY
  date == 20160929 AND
  cluster == 'emr_adhoc'
```
Stats filters.
Sample query.

**Spark.**

```scala
val low_cpu_count = partition
  .filter("name = 'system.cpu.utilization'")
  .filter("value < 0.8")
  .count()
```

**Pig.**

```pig
low_cpu = FILTER partition BY
    name == 'system.cpu.utilization' AND
    value < 0.8;
low_cpu_count = FOREACH
    (GROUP low_cpu ALL) GENERATE
    COUNT(name);
```
My job was 5 minutes faster!
Did it work?

- Success metrics: S3 bytes read, CPU time spent

  S3N: Number of bytes read: 1,366,228,942,336
  CPU time spent (ms): 280,218,780

- Filter didn’t work. Bytes read shows the entire partition was read.

- What happened?
Inspect the file.

- Stats show what happened:

  Row group 0: count: 84756 845.42 B records
  type encodings count avg size nulls min / max
  name   BINARY  G _  84756 61.52 B 0 "A..." / "z..."
  ...  

  Row group 1: count: 84756 845.42 B records
  type encodings count avg size nulls min / max
  name   BINARY  G _  85579 61.52 B 0 "A..." / "z..."

- Every row group matched the query
Add query columns to the sort.

Spark.
sqlContext
  .table("raw_metrics")
  .sort("day", "cluster", "name")
  .write.insertInto("metrics")

Pig.
metrics = LOAD 'raw_metrics'
  USING SomeLoader;
metricsSorted = ORDER metrics
  BY day, cluster, name;
STORE metricsSorted INTO 'metrics'
  USING ParquetStorer;
Inspect the file, again.

- Stats are fixed:

  **Row group 0:** count: 84756  845.42 B records
    - type    encodings count     avg size   nulls min / max
    - name    BINARY G _     84756     61.52 B    0   "A..." / "F..."

  ...  

  **Row group 1:** count: 85579  845.42 B records
    - type    encodings count     avg size   nulls min / max
    - name    BINARY G _     85579     61.52 B    0   "F..." / "N..."

  ...  

  **Row group 2:** count: 86712  845.42 B records
    - type    encodings count     avg size   nulls min / max
    - name    BINARY G _     86712     61.52 B    0   "N..." / "b..."
Dictionary filters.
Dictionary filtering.

Dictionary is a compact list of all the values.

- Search term missing? Skip the row group
- Like a bloom filter without false positives

When dictionary filtering helps:

- When a column is sorted in each file, not globally sorted – one row group matches
- When filtering an unsorted column
Dictionary filtering overhead.

Read overhead.
- Extra seeks
- Extra page reads

Not a problem in practice.
- Reading both dictionary and row group resulted in < 1% penalty
- Stats filtering prevents unnecessary dictionary reads
Works out of the box, right?

Nope.

- Only works when columns are completely dictionary-encoded
- Plain-encoded pages can contain any value, dictionary is no help
- All pages in a chunk must use the dictionary

Dictionary fallback rules:

- If dictionary + references > plain encoding, fall back
- If dictionary size is too large, fall back (default threshold: 1 MB)
Fallback to plain encoding.

```
parquet-tools dump -d
utc_timestamp_ms TV=142990 RL=0 DL=1 DS: 833491 DE:PLAIN_DICTIONARY

<table>
<thead>
<tr>
<th>Page</th>
<th>Encoding</th>
<th>Values</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>DLE:RLE RLE:BIT_PACKED V:RLE</td>
<td></td>
<td>72912</td>
</tr>
<tr>
<td>1</td>
<td>DLE:RLE RLE:BIT_PACKED V:RLE</td>
<td></td>
<td>135022</td>
</tr>
<tr>
<td>2</td>
<td>DLE:RLE RLE:BIT_PACKED V:PLAIN</td>
<td></td>
<td>1048607</td>
</tr>
<tr>
<td>3</td>
<td>DLE:RLE RLE:BIT_PACKED V:PLAIN</td>
<td></td>
<td>1048607</td>
</tr>
<tr>
<td>4</td>
<td>DLE:RLE RLE:BIT_PACKED V:PLAIN</td>
<td></td>
<td>714941</td>
</tr>
</tbody>
</table>
```

What’s happening:

- Values repeat, but change over time
- Dictionary gets too large, falls back to plain encoding
- Dictionary encoding is a size win!
Avoid encoding fallback.

**Increase max dictionary size.**
- 2-3 MB usually worked
- `parquet.dictionary.page.size`

**Decrease row group size.**
- 24, 32, or 64 MB
- `parquet.block.size`
- New dictionary for each row group
- Also lowers memory consumption!

**Run several tests to find the right configuration (per table).**
Row group size.

Other reasons to decrease row group size:

- Reduce memory consumption – but **not** to avoid write-side OOM
- Increase number of tasks / parallelism
Results!
Results (from Pig).

**CPU and wall time dropped.**
- Initial: CPU Time: 280,218,780 ms  Wall Time: 15m 27s
- Filtered: CPU Time: 120,275,590 ms  Wall Time: 9m 51s
- Final: CPU Time: 9,593,700 ms  Wall Time: 6m 47s

**Bytes read is much better.**
- Initial: S3 bytes read: 1,366,228,942,336  (1.24 TB)
- Filtered: S3 bytes read: 49,195,996,736  (45.82 GB)
Filtered vs. final time.

Row group filtering is parallel.
- Split planning is independent of stats (or else is a bottleneck)
- Lots of very small tasks: read footer, read dictionary, stop processing

Combine splits in Pig/MR for better time.
- 1 GB splits tend to work well
Other work.
Format version 2.

What’s included:

- New encodings: delta-integer, prefix-binary
- New page format to enable page-level filtering

New encodings didn’t help with Netflix data.

- Delta-integer didn’t help significantly, even with timestamps (high overhead?)
- Not large enough prefixes in URL and JSON data

Page filtering isn’t implemented (yet).
Brotli compression.

- New compression library, from Google
- Based on LZ77, with compatible license

**Faster compression, smaller files, or both.**

- brotli-5: **19.7% smaller, 2.7% slower** – 1 day of data from Kafka
- brotli-4: **14.8% smaller, 12.5% faster** – 1 hour, 4 largest Parquet tables
- brotli-1: **8.1% smaller, 28.3% faster** – JSON-heavy dataset
Brotli compression. (continued)
Future work.
Future work.

Short term:
- Release Parquet 1.9.0
- Test Zstd compression
- Convert embedded JSON to Avro – good preliminary results

Long-term:
- New encodings: Zig-zag RLE, patching, and floating point decomposition
- Page-level filtering
Thank you!

Questions?

https://jobs.netflix.com/

rblue@netflix.com