Parquet data format performance

Jim Pivarski

Princeton University – DIANA-HEP

February 21, 2018

What is Parquet?

1974	HBOOK	tabular	rowwise	FORTRAN	first ntuples in HEP
1983	ZEBRA	hierarchical	rowwise	FORTRAN	event records in HEP
1989	PAW CWN	tabular	columnar	FORTRAN	*faster* ntuples in HEP
1995	ROOT	hierarchical	columnar	C++	object persistence in HEP
2001	ProtoBuf	hierarchical	rowwise	many	Google's RPC protocol
2002	MonetDB	tabular	columnar	database	"first" columnar database
2005	C-Store	tabular	columnar	database	also early, became HP's Vertica
2007	Thrift	hierarchical	rowwise	many	Facebook's RPC protocol
2009	Avro	hierarchical	rowwise	many	Hadoop's object permanance and interchange format
2010	Dremel	hierarchical	columnar	C++, Java	Google's nested-object database (closed source), became BigQuery
2013	Parquet	hierarchical	columnar	many	open source object persistence, based on Google's Dremel paper
2016	Arrow	hierarchical	columnar	many	shared-memory object exchange

What is Parquet?

1974	HBOOK	tabular	rowwise	FORTRAN	first ntuples in HEP
1983	ZEBRA	hierarchical	rowwise	FORTRAN	event records in HEP
1989	PAW CWN	tabular	columnar	FORTRAN	*faster* ntuples in HEP
1995	ROOT	hierarchical	columnar	C++	object persistence in HEP
2001	ProtoBuf	hierarchical	rowwise	many	Google's RPC protocol
2002	MonetDB	tabular	columnar	database	"first" columnar database
2005	C-Store	tabular	columnar	database	also early, became HP's Vertica
2007	Thrift	hierarchical	rowwise	many	Facebook's RPC protocol
2009	Avro	hierarchical	rowwise	many	Hadoop's object permanance and interchange format
2010	Dremel	hierarchical	columnar	C++, Java	Google's nested-object database (closed source), became BigQuery
2013	Parquet	hierarchical	columnar	many	open source object persistence, based on Google's Dremel paper
2016	Arrow	hierarchical	columnar	many	shared-memory object exchange

Developed independently to do the same thing

Google Dremel authors claimed to be unaware of any precedents, so this is an example of convergent evolution.

wings are not limbs

wings are arms

wings are hands

Feature comparison: ROOT and Parquet

ROOT

- Store individual C++ objects rowwise in TDirectories and large collections of C++ objects (or simple tables) rowwise or columnar in TTrees.
- Can rewrite to the same file, like a database, but most users write once.
- Selective reading of columns (same).
- Cluster/basket structure (same).
- Plain encodings, one level of depth (deeper structures are rowwise).
- Compression codecs: gzip, lz4, lzma, zstd (under consideration)

Parquet

- Only store large collections of language-independent, columnar objects. The whole Parquet file is like a single "fully split" TTree.
- Write once, producing an immutable artifact.
- Selective reading of columns (same).
- Row group/page structure (same).
- Highly packed encodings, any level of depth (logarithmic scaling with depth).
- Compression codecs: snappy, gzip, lzo, brotli, lz4, zstd (version 2.3.2)

Implementation comparision: ROOT and Parquet (1)

ROOT

- Metadata and seeking through the file starts with a header.
- Header must be rewritten as objects (including baskets) accumulate.
- Failure is partially recoverable, but writing is non-sequential.
- Also facilitates rewriting (use as a database).

Parquet

- Metadata and seeking through the file starts with a footer.
- Data are written sequentially and seek points are only written at the end.
- Failure invalidates the whole file, but writing is sequential.
- Parquet files are supposed to be immutable artifacts.

Implementation comparision: ROOT and Parquet (2)

ROOT

- Layout of metadata and data structures are specified by streamers, which are saved to the same file.
- Streamer mechanism has built-in schema evolution.
- Data types are C++ types.
- Objects in TTrees are specified by the same streamers.

Parquet

- Layout of metadata and data structures are specified by Thrift, an external rowwise object specification.
- Thrift has schema evolution.
- Simple data types are described by a physical schema, related to external type systems by a logical schema.
- Thrift for metadata, schemas for data— no unification of data and metadata.

Implementation comparision: ROOT and Parquet (3)

ROOT

- Contiguous data array accompanied by:
  - navigation array: pointers to the start of each variable-sized object.
- Permits random access by entry index.

Parquet

- Contiguous data array accompanied by:
  - definition levels: integers indicating depth of first `null` in data; maximum for non-null data.
  - repetition levels: integers indicating depth of continuing sub-list, e.g. `0` means new top-level list.
- Definition levels even required for non-nullable data, to encode empty lists.
- Schema depth fixes maximum definition/repetition values, and therefore their number of bits.
- Must be unraveled for random access.

Implementation comparision: ROOT and Parquet (4)

ROOT

- Data are simply encoded, by streamers or as basic C++ types (e.g. `Char_t`, `Int64_t`, `float`, `double`).

Parquet

- Integers and booleans with known maxima are packed into the fewest possible bits.
- Other integers are encoded in variable-width formats, e.g. 1 byte up to 127, 2 bytes up to 16511, zig-zagging for signed integers.
- Dynamically switches between run length encoding and bit-packing.
- Optional "dictionary encoding," which replaces data with a dictionary of unique values and indexes into that dictionary (variable-width encoded).

Implementation comparision: ROOT and Parquet (5)

ROOT

- Granular unit of reading and decompression is a basket, which may be anywhere in the file (located by TKey).
- Entry numbers of baskets *may* line up in clusters (controlled by `AutoFlush`). Clusters are a convenient unit of parallelization.

Parquet

- Granular unit of reading and decompression is a page, which must be contiguous by column (similar to ROOT's `SortBasketsByBranch`).
- Entry numbers of columns (contiguous group of pages) *must* line up in row groups, which is the granular unit of parallelization.

File size comparisons

Parameters of the test (1)

Datasets: 13 different physics samples in CMS NanoAOD.

Non-trivial structure: variable length lists of numbers, but no objects.

ROOT: version 6.12/06 (latest release)

- Cluster sizes: 200, 1000, 5000, 20 000, 100 000 events
- Basket size: 32 000 bytes (1 basket per cluster in all but the largest)
- Freshly regenerated files in this ROOT version: `GetEntry` from the CMS originals and `Fill` into the datasets used in this study.

Parquet: C++ version 1.3.1 inside pyarrow-0.8.0 (latest release)

- Generated by ROOT → uproot → Numpy → pyarrow → Parquet, controlling array size so that Parquet row groups are identical to ROOT clusters, pages to baskets.
- Parquet files preserve the complete semantic information of the original; we can view the variable length lists of numbers in Pandas.

Parameters of the test (2)

The purpose of the ensemble of 13 physics samples is to vary probability distributions: e.g. Drell-Yan has a different muons-to-jets ratio than $t\bar{t}$.

However, these samples also differ in total content (number of events, number of particles), which is not relevant to performance.

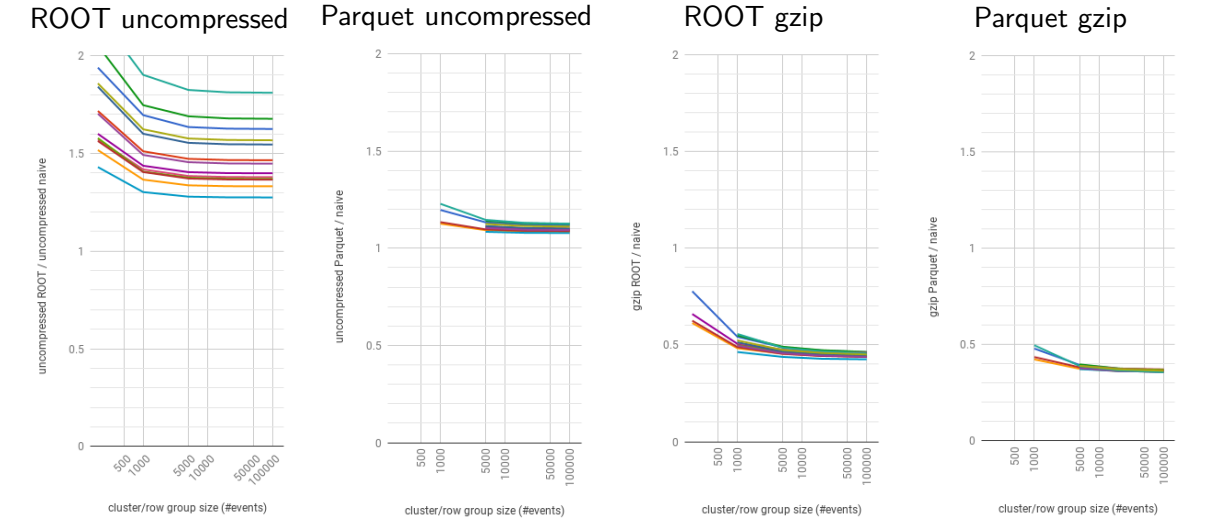
Each *sample* is divided by its "naïve size," obtained by saving as Numpy files:

- Each n byte number in memory becomes an n byte number on disk.
- Each boolean in memory becomes 1 bit on disk (packed).
- No compression, insignificant metadata ($<$1 kB per GB file).

Different conditions (cluster sizes, compression cases) and formats (ROOT, Parquet) have the *same* normalization factor for the *same* sample.

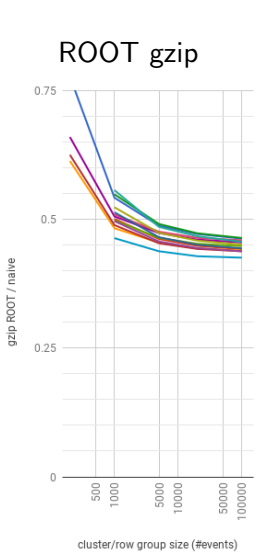
Normalized sizes above 1.0 are due to metadata and overhead; below 1.0 due to compression or packed encodings.

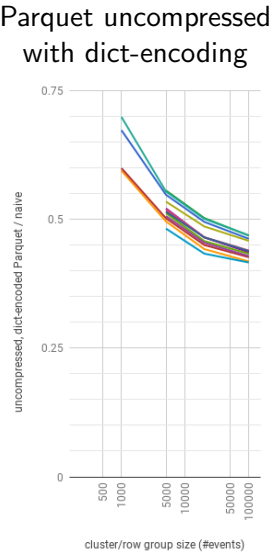
Normalized file sizes versus cluster/row group size

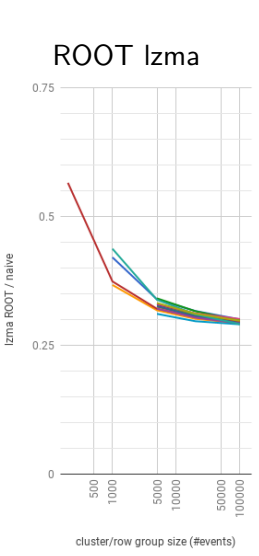


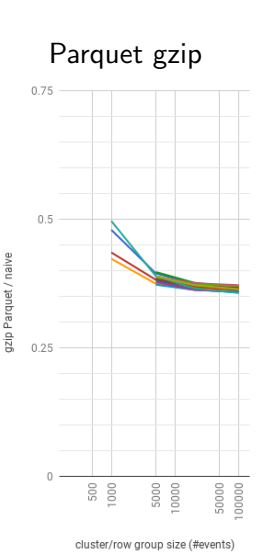
Uncompressed Parquet is smaller and less variable than ROOT, but gzip-compressed are similar.

Parquet's dictionary encoding is like a compression algorithm









Are NanoAOD's boolean branches favoring Parquet? (No.)

$601 + 21$ of NanoAOD's 955 branches are booleans (named `HLT_*` and `Flag_*`), which unnecessarily inflate the uncompressed ROOT size (8 bytes per boolean).

Highest cluster/row group (100 000 events), average and standard deviation sizes:

	all branches		without trigger	
	ROOT	Parquet	ROOT	Parquet
uncompressed	1.48 ± 0.16	1.10 ± 0.02	1.32 ± 0.11	$1.10 \pm 0.02^*$
dictionary encoding		0.44 ± 0.02		0.42 ± 0.01
lz4	0.60 ± 0.03		0.58 ± 0.03	
gzip	0.45 ± 0.01	0.36 ± 0.01	$0.45 \pm 0.01^*$	0.37 ± 0.01
lzma	0.30 ± 0.01		0.29 ± 0.01	

(*not copy-paste errors)

The triggers alone are not responsible for the large ROOT file sizes (and variance).

What about dropping the navigation arrays? (with TIOFeatures)

Last year, we predicted 10–30% improvements if we drop these arrays, depending on compression algorithm (lzma had the least to gain, lz4 the most).

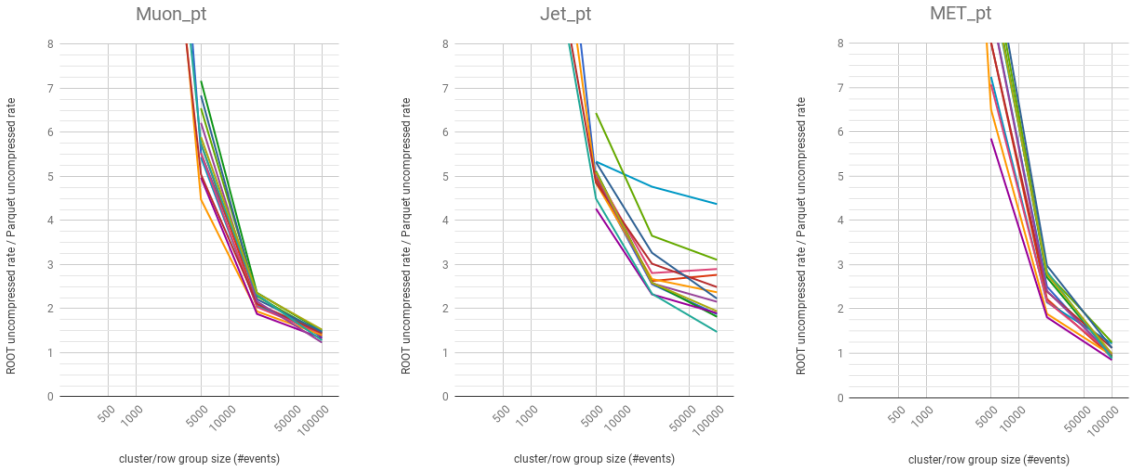
Highest cluster/row group (100 000 events), average and standard deviation sizes:

	ROOT default	ROOT no navigation	Parquet
uncompressed	1.48 ± 0.16	1.20 ± 0.07	1.10 ± 0.02
gzip	0.45 ± 0.01	0.35 ± 0.01	0.36 ± 0.01
lzma	0.30 ± 0.01	0.27 ± 0.01	
lz4	0.60 ± 0.03	0.41 ± 0.01	
dictionary encoding			0.44 ± 0.02

Dropping navigation arrays is a little better than predicted last year and brings ROOT-gzip exactly in line with Parquet-gzip. However, ROOT-lz4 is about the same as *uncompressed* Parquet with dictionary encoding.

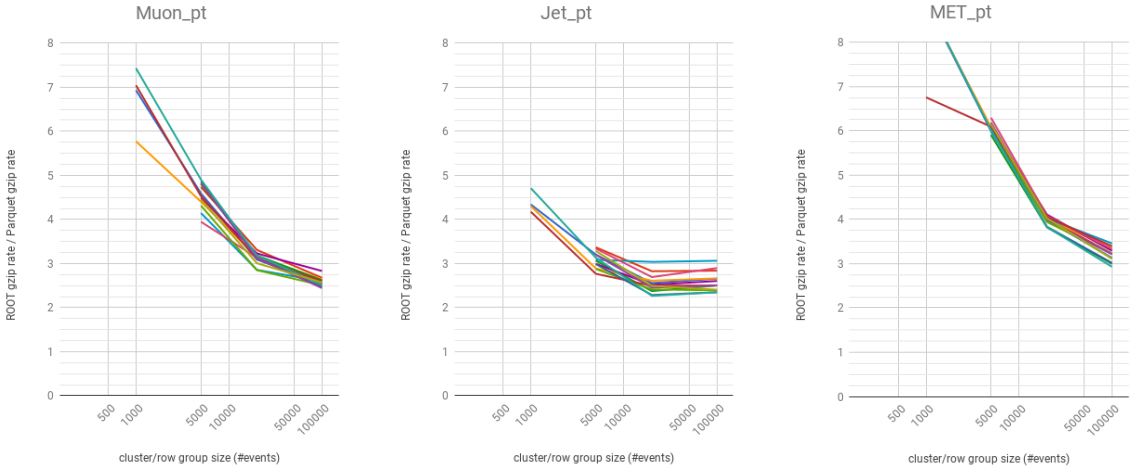
Throughput comparisons

ROOT reading rate / Parquet reading rate: uncompressed



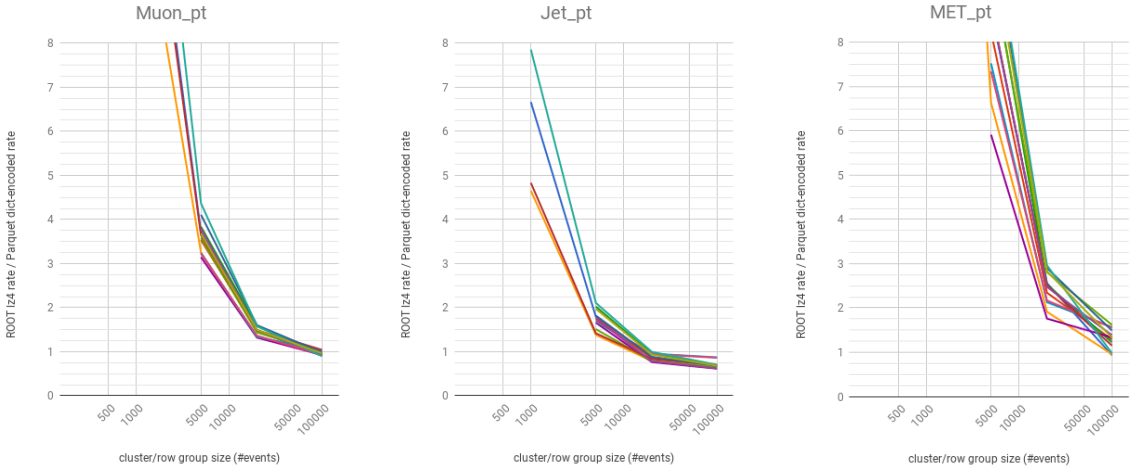
ROOT is much faster than Parquet-C++ for small cluster/row group sizes, but the difference levels out at high cluster/row group sizes.

ROOT reading rate / Parquet reading rate: both gzipped



The slope with respect to cluster/row group size is less pronounced when both are gzipped, but still ROOT is several times faster.

ROOT reading rate (lz4) / Parquet reading rate (dict-encoded)



Decoding lz4 in ROOT is about as fast as decoding Parquet's dictionary encoding (given large cluster/row group sizes), so there's no size vs speed advantage.

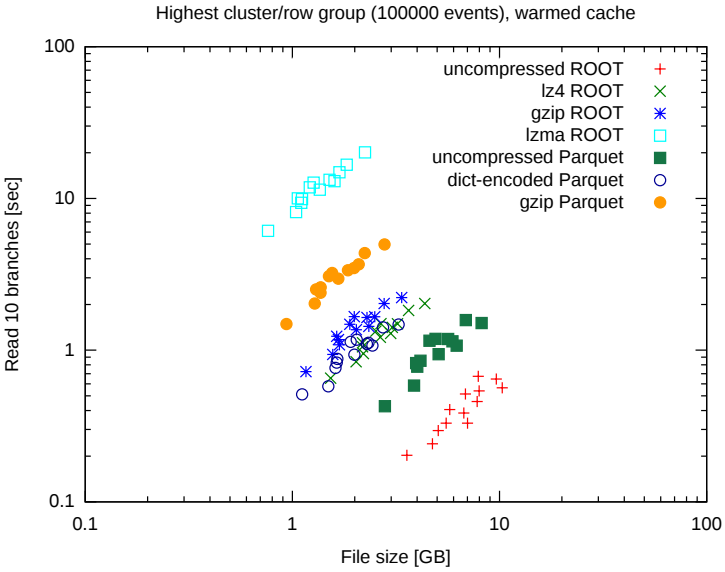
Throughput summary

Highest cluster/row group (100 000 events), average and standard deviation times to read 10 branches (seconds):

	ROOT		Parquet	
	cold cache	warm cache	cold cache	warm cache
uncompressed	0.6 ± 0.2	0.4 ± 0.2	1.1 ± 0.3	1.0 ± 0.3
gzip	1.6 ± 0.5	1.4 ± 0.4	3.1 ± 0.9	3.1 ± 0.9
lzma	12.2 ± 3.7	12.1 ± 3.7		
lz4	1.4 ± 0.4	1.3 ± 0.4		
dictionary encoding			1.0 ± 0.3	1.0 ± 0.3

For large cluster/row groups, ROOT 6.12/06 is twice as fast as Parquet-C++ 1.3.1.

Summary: Parquet's encodings yield smaller and slower files



. . . like compression