Survey of columnar file formats and the techniques they use

Jim Pivarski

Princeton – DIANA

February 8, 2017

The ROOT file format is many things

- Generic: designed for any type of data.
- Key-value object store for histograms, lookup tables, etc.
- Big data storage: identically structured files with large TTrees. (Byte for byte, this is the most common use-case!)
- Binary and schemaed (TStreamerInfo) for efficient access.
- Hierarchical data, such as events containing jets containing tracks containing hits.
- Remotely accessible via the XRootD protocol.
- Record-oriented or columnar with a configurable splitting level.
- And now multilingual, with jsROOT, root4j, and soon go-root.

The ROOT file format is many things

- Generic: designed for any type of data.
- Key-value object store for histograms, lookup tables, etc.
- Big data storage: identically structured files with large TTrees. (Byte for byte, this is the most common use-case!)
- Binary and schemaed (TStreamerInfo) for efficient access.
- Hierarchical data, such as events containing jets containing tracks containing hits.
- Remotely accessible via the XRootD protocol.
- Record-oriented or columnar with a configurable splitting level.
- And now multilingual, with jsROOT, root4j, and soon go-root.

This talk will be about file formats that share these features and what we can learn from them.

The ROOT file format is many things

- Generic: designed for any type of data.
- Key-value object store for histograms, lookup tables, etc.
- Big data storage: identically structured files with large TTrees. (Byte for byte, this is the most common use-case!)
- Binary and schemaed (TStreamerInfo) for efficient access.
- Hierarchical data, such as events containing jets containing tracks containing hits.
- Remotely accessible via the XRootD protocol.
- Record-oriented or columnar with a configurable splitting level.
- And now multilingual, with jsROOT, root4j, and soon go-root.

This talk will be about file formats that share these features and what we can learn from them. But especially these.

Columnar data

Database storage formats resemble ROOT TNtuples: user usually only touches a few database columns, so it's important to be able to access them without being slowed down by the others.

Example: ORC file format for Hive (Hadoop as a database). Each data column is saved as a contiguous, equal-length array.

Generic, binary, non-columnar formats, such as ProtocolBuffers, Thrift, and Avro, are better suited to remote procedure calls (RPC) and streaming analytics ("live" data without storage).

Hierarchical data

Although SQL-99 introduced arrays and structures (nested data), its language support is underwhelming.

(For instance, how would you pick out the `px`, `py`, `pz` of the top two muons in an event and construct an invariant mass in SQL?)

ORC files store arrays and structures within a column "unsplit." If you want one subfield, you have to load or skip over all subfields.

Hierarchical data

Although SQL-99 introduced arrays and structures (nested data), its language support is underwhelming.

(For instance, how would you pick out the `px`, `py`, `pz` of the top two muons in an event and construct an invariant mass in SQL?)

ORC files store arrays and structures within a column "unsplit." If you want one subfield, you have to load or skip over all subfields.

Nevertheless, the industry is moving in this direction: a Google paper (link) described a hierarchical, columnar file format, used in-house since 2006.

This paper is the basis for Apache Parquet (file format), Apache Arrow (in-memory data representation), SparkSQL 2.0 optimizations, Ibis, Impala, Kudu, Drill query servers, and probably others.

They didn't know about ROOT!

diana hep

Dremel: Interactive Analysis of Web-Scale Datasets (2010)

Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, Theo Vassilakis

storage and reduce CPU cost due to cheaper compression. Column stores have been adopted for analyzing relational data [1] but to the best of our knowledge have not been extended to nested data models. The columnar storage format that we present is supported by many data processing tools at Google, including MR, Sawzall [20], and FlumeJava [7].

In this paper we make the following contributions:

- We describe a novel columnar storage format for nested data. We present algorithms for dissecting nested records into columns and reassembling them (Section 4).

Independently developed: this is xenobiology!

Translation dictionary

ROOT	Google Dremel and Apache Parquet
Streamer info	Schema: description of all data types
Splitting	Shredding: breaking objects into columns
Event cluster	Row group: group of columns (which may have different lengths) corresponding to a fixed number of rows
TBuffer	Column: contiguous data for one scalar leaf of the schema
TBasket	Page: fixed-size chunk of data for one compression pass

Repetition and definition levels

Schema of primitives that can be repeated, required, or optional.

```
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; }}
```

All data can be flattened with two extra fields: *r* and *d*.

r_1

```
DocId: 10
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'
```

r_2

```
DocId: 20
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

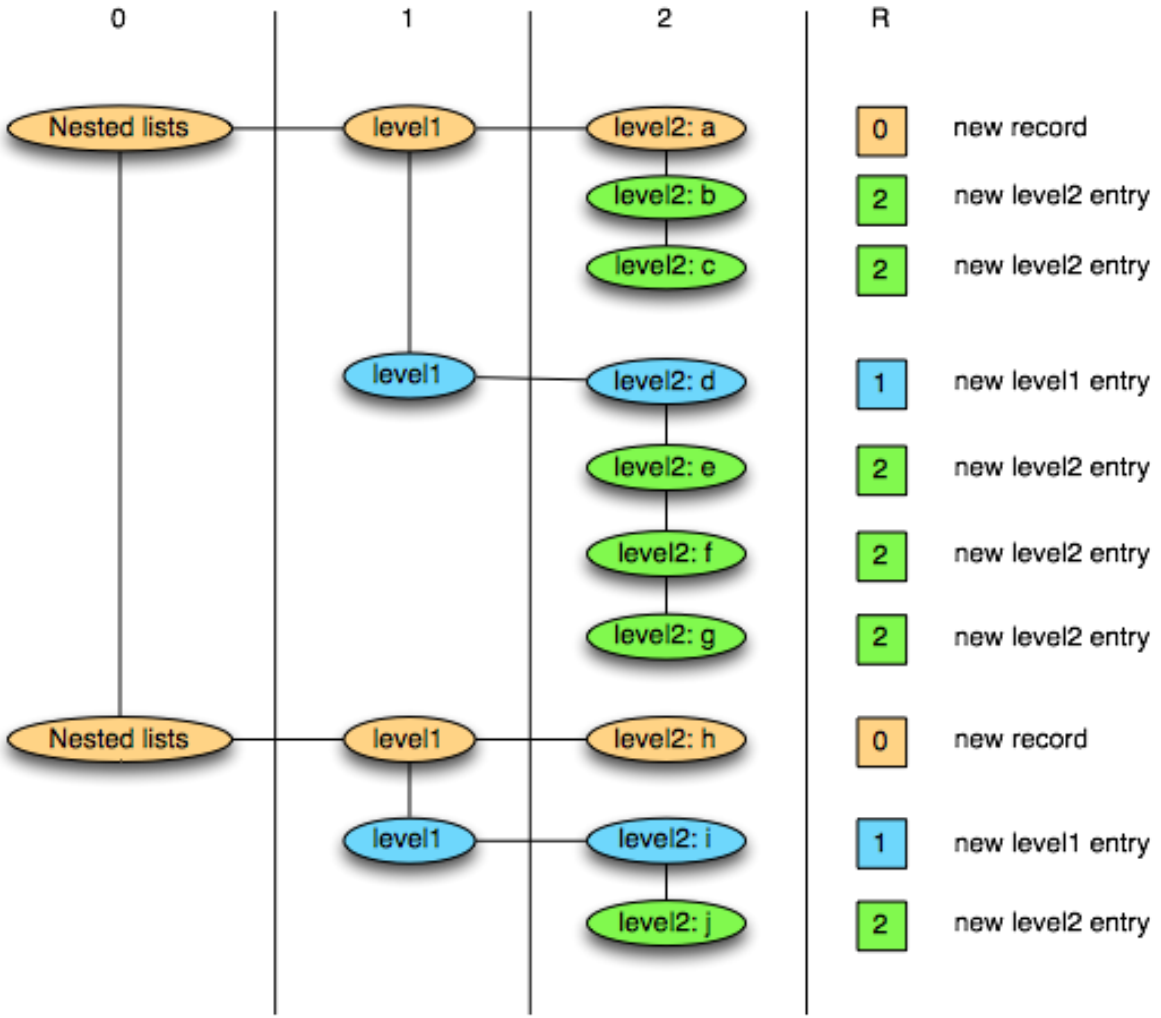
value	r	d
en-us	0	2
en	2	2
NULL	1	1
en-gb	1	2
NULL	0	1

Name.Language.Country

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

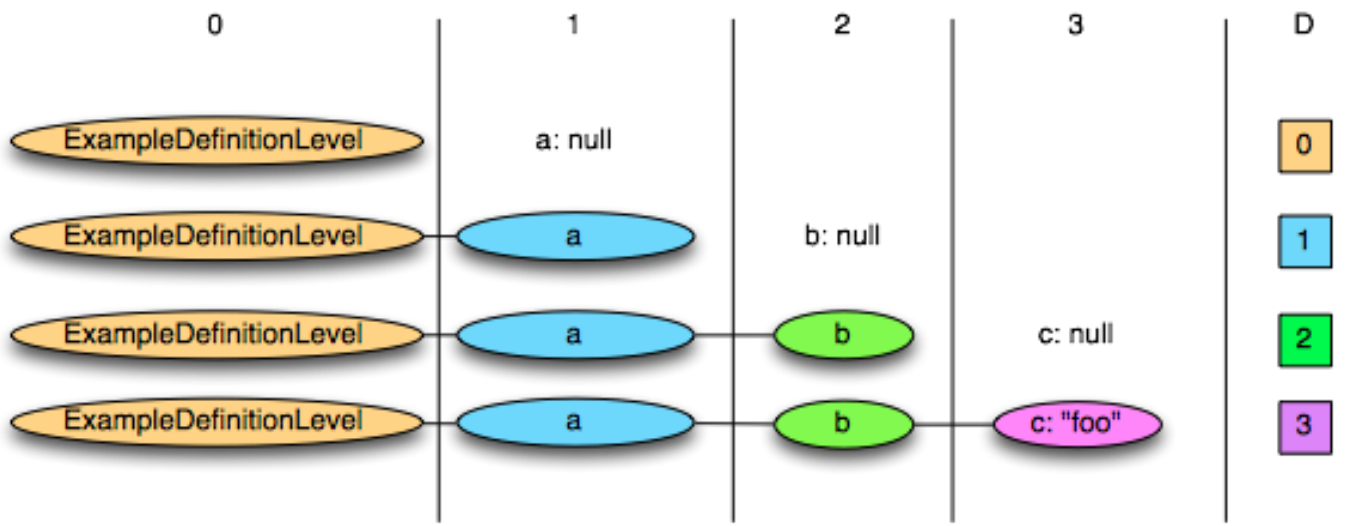
Detail on repetition level

Given:	[	[	a	b	c	]	[	d	e	f	g	]	]	[	[	h	]	[	i	j	]	]		
Data array:			a	b	c			d	e	f	g					h			i	j				
Repetition level:			0	2	2			1	2	2	2					0			1	2				



Detail on definition level

Value	Definition Level
`a: null`	0
`a: { b: null }`	1
`a: { b: { c: null } }`	2
`a: { b: { c: "foo" } }`	3 (actually defined)



Intended for nullable data (e.g. missing values), but also *required* to describe empty lists. Can't adopt repetition levels without definition levels.

Comparison to ROOT's counters

diana hep

Dremel/Parquet

- Repetition/definition levels describe arbitrarily deep nesting in one pair of arrays.
- Maximum possible r, d values determined by depth of nesting; tightly packable.
  (A depth-1 list of any length can be described by two bits *per element.*)
- Determining list size is a history-dependent calculation.

ROOT

- A separate counter branch would be needed for every level of depth.
- List length is bounded by the number of bits in the counter.
  (An 8-bit counter is more tightly packed for lists of length 4 through 255.)
- List size is readily available.

Best of both worlds

Can we keep counters but gain the ability to describe arbitrarily deep nesting in one array? Yes!

→ Fill the counter recursively.

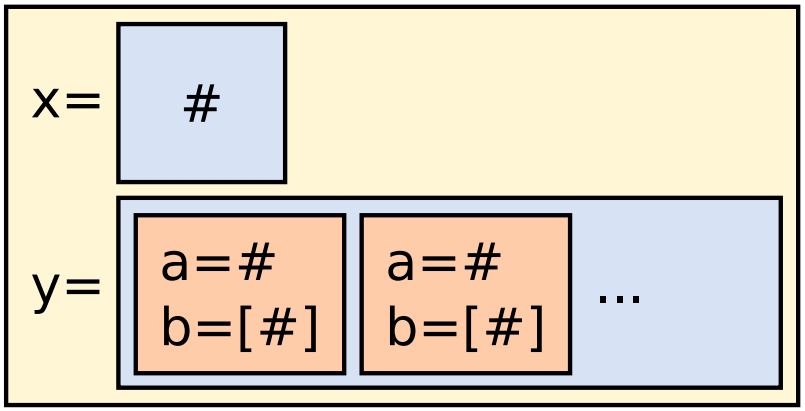
Given:	`[ [ a b c ] [ d e f g ] ] [ [ h ] [ i j ] ]`
Data array:	`a b c d e f g h i j`
Recursive counter:	2 3 4 2 1 2

- Lossless: we know to interpret the orange numbers as first-level and the blue numbers as second-level by reading it left to right and counting (history-dependent calculation).
- Backward compatible with ROOT's counters.
- Data array is now completely split; we can apply vectorized functions to it (e.g. GPU kernels) for any depth of nesting.

A more complex example

Consider a schema that mixes arrays and structs:

```
class outer {
  double x;
  vector<inner> y;
};
class inner {
  double a;
  vector<double> b;
};
```



Data like this:

```
outer(x=1, y=[inner(a=2, b=[ 3,  4]), inner(a=5, b=[])])
outer(x=6, y=[inner(a=9, b=[10, 11])                   ])
```

Only needs a counter for each leaf with different structure:

```
x:    [1, 6]
y.a: [2, 5, 9]          y.a@size: [2, 1]
y.b: [3, 4, 10, 11]     y.b@size: [2, 2, 0, 1, 2]
```

Another trick: variable length integers

Unsigned

Only use as many bytes as necessary for each individual integer (like UTF-8).

number	serialization
0	`00`
1	`01`
127	`7f`
128	`80 01`
129	`81 01`
16,383	`ff 7f`
16,384	`80 80 01`
16,385	`81 80 01`

Signed

Same trick as unsigned, but first transform so negative values near zero are short byte patterns.

number	transformed
0	0
−1	1
1	2
−2	3
2	4

- Simplifies encoding: same code for char, short, int, and long.
- Smaller file sizes. (Inserts the assumption that small integers are common, rather than making the compression algorithm discover that.)

Another consideration: abstract schemas

- ROOT's streamer info describes C++ objects. Hard to translate into objects in other languages (e.g. root4j).

Another consideration: abstract schemas

- ROOT's streamer info describes C++ objects. Hard to translate into objects in other languages (e.g. root4j).
- Many new schema systems are language-agnostic. They define types like "string," "integer," "record," and "list," which might be represented in different ways in different languages.

Another consideration: abstract schemas

- ROOT's streamer info describes C++ objects. Hard to translate into objects in other languages (e.g. root4j).
- Many new schema systems are language-agnostic. They define types like "string," "integer," "record," and "list," which might be represented in different ways in different languages.
- Parquet further distinguishes between "physical schema" (just enough information to encode/decode) and "logical schema" (more detail about how to represent for analysis).

physical	logical
integer	enumeration
list of pairs	hash table

Another consideration: abstract schemas

- ROOT's streamer info describes C++ objects. Hard to translate into objects in other languages (e.g. root4j).
- Many new schema systems are language-agnostic. They define types like "string," "integer," "record," and "list," which might be represented in different ways in different languages.
- Parquet further distinguishes between "physical schema" (just enough information to encode/decode) and "logical schema" (more detail about how to represent for analysis).

physical	logical
integer	enumeration
list of pairs	hash table

- Scientific Python community is developing a standard called Datashape (link) with C++ friendly features, including records, fixed and variable length arrays, and pointers.

Conclusions

With this talk, I mostly wanted to let you know what other developers are doing to solve similar problems.

If any of these could be construed as requests or recommendations, I'd like to find out if there's interest in using recursive counters to split data structures down to all levels of depth.

I'll have more to say about that in my next talk.