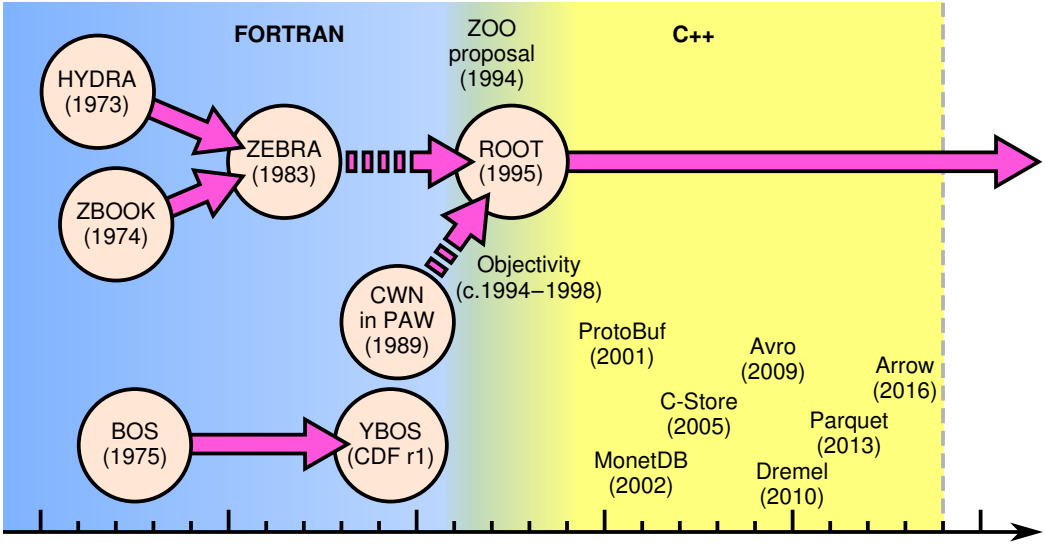
Overview of Serialization Technologies

Jim Pivarski

Princeton University – DIANA-HEP

March 28, 2018

45 years of serialization formats in (and out of) HEP



The two most significant features, setting HEP apart

Hierarchically nested structures

For example: event contains jets,
jets contain tracks,
tracks contain hits...

It's important that the nested objects have variable length, since structs of structs of integers are not *really* nested: they compile to constant offsets, just like flat data.

Fortran (pre-90) didn't have this feature, so physicists had to add it.

Columnar representation

For example: all values of muon p_T are contiguous in serialized form, followed by all values of muon η, all values of muon ϕ, and all #muons per event.

Thus, you can read muon p_T without reading jet p_T.

Easy for flat data: it's just a transpose.

There are several techniques for solving it in general (hot CS topic in early 2000's).

20 questions to ask about any file format

Expressiveness

- **Hierarchically nested or flat tables?**
- Has schema (strongly typed) or dynamic?
- Schema evolution, if applicable?
- Language agnostic or specific?

Performance

- **Rowwise or columnar?**
- Compressed/compressible?
- Robust against bit errors?
- Serialized/runtime unity?

2^{20} yes/no combinations = 1 million formats

Accessibility

- Human readable, binary, or both?
- Immutable/append only/full database?
- Partial-write recovery?
- Parallel readable?
- Parallel writable?
- Streamable?
- Random accessible?
- Database-style indexing?
- RPC protocol?

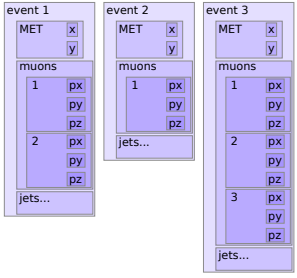
Community

- Has specification?
- Independent implementations?
- Size of user base?

Expressiveness: Hierarchically nested or flat tables?

Hierarchical nesting could be seen as a special case of graph data, but it's an important special case because hierarchical types may have strongly typed schemas and special contiguity properties, such as rowwise vs. columnar.

Hierarchical



Flat table

	column 1	column 2	column 3
row 1			
row 2			
row 3			
row 4			
row 5			
row 6			

(conversion to a flat table is *lossy!*)

Haves: ROOT, Parquet, Avro, JSON, XML, and many others...

Have nots: CSV, SQLite, Numpy, *high-performance* HDF5...

Expressiveness: Has schema (strongly typed) or dynamic?

Same issue as in programming languages: can we express the data type once for all elements of a collection or do we have to express it separately for each element?

Haves: ROOT, Parquet, Avro, and many others...

Have nots: JSON, BSON (binary JSON), MessagePack, and many others...

Just as in programming languages, there are arguments for and against schemas, and they're helpful in some situations, harmful in others.

In HEP, we know the schema in advance for reasonably large blocks of event data and want the performance advantages of schemas. Without a schema, every object must be accompanied by type metadata (even if it's just a reference). Repeated field names account for most of JSON and BSON's bloat (it's *not* because JSON is text!).

Expressiveness: Schema evolution, if applicable?

If a schema is used to compile runtime objects, *then* a mismatch between an old schema and a new schema can render old data unreadable. Schema evolution is a set of rules to automatically translate data schemas into container objects.

Haves: ROOT, ProtoBuf, Thrift, Avro, all in very different ways

Have nots: Objectivity, Java serialization (without manual effort)...

Inapplicables: Any schemaless format (e.g. JSON) and any format without fixed containers: Google FlatBuffers (runtime indirection), OAMap (JIT)

Schema evolution isn't a well-defined dichotomy, but a spectrum of techniques filling the continuum between static typing (assert types at the earliest possible time) and dynamic typing (assert types at the latest possible time). ROOT's typing is not strictly "static" because it uses a JIT-compiler to compile Streamer Rules.

Generally, we'd like to delay "hard compiling" types until *after* we've seen the data schema and *before* we run over a million events. JIT is a good solution.

Expressiveness: Language agnostic or specific?

A serialization format may be specialized to a given language and (eventually) handle every data type in that language. This is true of ROOT, which covers almost all of C++'s types. Good for optimizing data types to the code, rather than serialization.

Alternately, it could define a type system typical of programming languages but not aligned with any one language. These types are then mapped onto each language without a full bijection (e.g. all ProtoBuf types can be expressed in C++, but not all C++ types can be expressed in ProtoBuf). Good for cross-language interoperability.

Agnostic: ProtoBuf, Thrift, Avro, Parquet, Cap'n Proto, OAMap, SQL schemas

Specific: ROOT, Pickle (Python), Kryo (Java, for Spark to send data to jobs)

Language-specific formats are actually pretty rare: it was hard to find examples!

Performance: Rowwise or columnar?

Columnar representation allows more efficient access to a subset of attributes (the only ones used in an analysis, for instance) because network → computer, disk → memory, memory → CPU, and memory → co-processor pipelines all cache *contiguous bytes*.

Hierarchically nested example: `vector<vector<pair<char, int>>>`

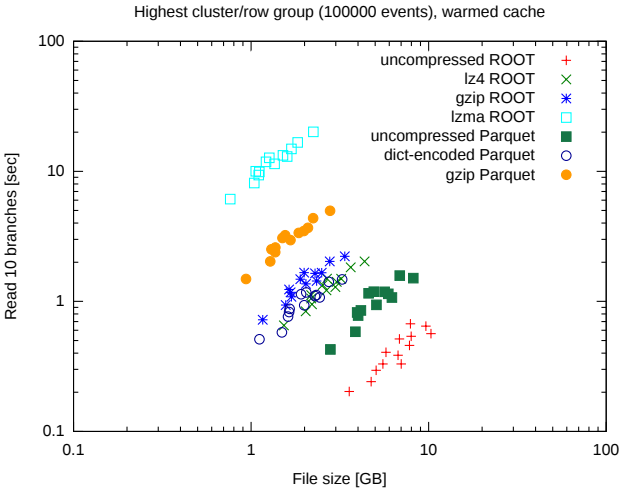
logical data	`[[(a,1), (b,2), (c,3), (d,4)], [], [(e,5), (f,6)]], [], [[(g,7)]]`
outer stops	`[ 3, 3, 4]`
inner stops	`[ 4, 4, 6, 7 ]`
1st attribute	`[ a, b, c, d, e, f, g ]`
2nd attribute	`[ 1, 2, 3, 4, 5, 6, 7 ]`

The "stops" column is a running total number of entries at each closing bracket *of that level of hierarchy*. The attribute data (leaves of the tree) are stored without boundaries.

Haves: ROOT (only one level deep), Parquet, ORC (database file), OAMap...
Have nots: JSON, ProtoBuf, Thrift, Avro, Google FlatBuffers, and many others...

Performance: Compressed/compressible?

Of course you can take any file and compress it, but it sure is convenient if compression is a built-in option. Compression trades read and write throughput for serialized size.



Some packed encodings skirt the boundary between compressed and uncompressed. Parquet's tightly packed data is smaller and slower *without* explicit compression than ROOT *with* lz4 compression.

Haves: ROOT, ProtoBuf, Thrift, Avro, Parquet, HDF5, and many others...

Have nots: JSON, CSV...

Performance: Robust aginst bit errors?

In large datasets, bit errors can occur. Robustness consists of three questions:

- Can a bit error be detected, for instance using a CRC?
- Can a bit error be repaired with replication in the file? (I've never heard of this.)
- If bad data has to be skipped, how much has to be skipped? A page or a whole file?

Since compression increases sensitivity to bit errors (can scramble an entire compression frame), compression formats often have CRC features built in. The question is whether the file format saves CRCs and whether readers check them.

Haves: ROOT, HDF5, Avro, Parquet, mostly through using the compressor's CRCs. As far as I know, these do not include robustness in object headers, just bulk data.

Have nots: JSON, CSV... Hard to be sure a format with compression doesn't use the CRC.

Performance: Serialized/runtime unity?

This is new; increasingly relevant as storage class memory becomes mainstream:

> Is the byte-representation of serialized data equal to the byte-representation of runtime objects? That is, are data structures zero-copy, zero-conversion?

It allows, for instance, a memory-mapped file to be used directly as runtime data. `mmap` (1 system call) is often faster than `fread` (many system calls) and deserializing data can be a throughput bottleneck, depending on what you do with it afterward.

Links: SNIA SSSI, DAX direct access, PMEM development kit, MMU mapping.

Haves: This is a design goal of the Apache Arrow project, particularly for Spark, Pandas, and R DataFrames. "Feather" is memory-mapped Arrow on disk. Apache Drill (SQL) highlights their lack of "row materialization." OAMap (raw data or Arrow backend). Cap'n Proto, Google Flatbuffers.

Have nots: Most data formats, including ROOT.

Compression always spoils serialized/runtime unity. It's not for archival data.

Accessibility: Human readable, binary, or both?

From *ASDF: A new data format for astronomy* (P. Greenfield, M. Droettboom, E. Bray):

> [HDF5] is an entirely binary format. FITS headers are easily human-readable. Nothing like that is the case for HDF5 files. All inspection of HDF5 files must be done through HDF5 software. With FITS, it is quite possible to inspect the header with very primitive tools. The consequence of this for HDF5 is that the HDF5 toolset must be installed, and its software must be used if one wants to inspect the contents of the HDF5 file in any way.

I find it fascinating that human-readability is such an important consideration in astronomy that it's an argument against HDF5, but HEP rarely considers it.

Human readable numbers with fewer than 4 or 8 characters may be viewed as a kind of lossy compression, especially when combined with standard compression...

Binary: ROOT, BSON, and most formats mentioned in this talk

Human: JSON, CSV, YAML (basis of ASDF)

Both: FITS, ASDF, YAML (through `!!binary`), XML (through `CDATA`)

Accessibility: Immutable/append only/full database?

Can you add data to an existing file? It doesn't count if you have to overwrite or rewrite large sections of the file to "make room" for the new data.

Most file formats are intended as immutable artifacts: if you want a new version of the data, you write a new file. In HEP, we often *use* data that way.

Append-only is an interesting case between immutability and full database-like access: you can add records to the end, but not in the middle.

Closely related to the question of partial-write recovery: appendable formats can usually be recovered up to the last good record.

Immutable:	Parquet (all metadata in footer), JSON and most text formats...
Appendable:	CSV, JSONLines (JSON `\n` JSON...), Avro, OAMap (raw data backend)
Database:	ROOT (header describes seek keys; can add new keys, invalidate old ones), HDF5, SQLite

Partial-write recovery?

If a writing process does not finish writing data, is the partially written data readable?

(Particularly important for DAQ systems that could crash and lose data forever.)

Haves: ROOT (with special-case handling in the reader), most rowwise appendable formats like CSV, JSONLines, and Avro, OAMap because it fills columns corresponding to innermost structures first

Have nots: Parquet (all metadata in footer; footer must be written), HDF5 (!!!)

Accessibility: Parallel readable?

Does the file format define chunks that can be understood independently of one another? Could a writing thread spoil that independence?

Haves: The ROOT format, if treated as an immutable object, is parallel readable at the basket level. uproot takes advantage of this, but ROOT's Implicit Multi-Threading (IMT) introduces synchronization points to coordinate mutable state (protect against writing threads?).

Blosc, a meta-compression framework, decompresses data in CPU cache-sized blocks faster than `memcpy` can copy uncompressed data.

HDF5 has parallel reading, as well as memory-mapped arrays in Numpy.

Have nots: Any variable-length data without seek keys: Avro, JSON...

Incidentally, parallel reading and writing are unsung features of memory-mapping: no mutable state is introduced by a file pointer. Many thread-local pointers read from different parts of the virtual address space.

Accessibility: Parallel writable?

Different from parallel reading: reading and writing are not symmetric!

Can a file format be *written* by independent threads? The positions of pages must be determined before writing, before you know how large the data to save will be (because it's compressed). This may lead to overestimates and therefore "gaps" in the output file, inflating its size.

Haves: Blosc, HDF5, memory-mapped arrays in Numpy...

Have nots: ROOT can compress baskets in parallel, but it must write them sequentially.

Accessibility: streamable?

Can a program make sense of data *before the file is fully read* and *without seeking?*

Some file access methods, like HTTP, have no ability to seek. For local disks, sequential reads are much faster than random reads.

Haves: JSONLines, CSV, Avro

Have nots: ROOT (seek keys in header), Parquet (seek keys in footer)

Accessibility: Random accessible?

Can a program jump to a desired element of a large collection by entry number?

It doesn't count if you have to deserialize the first $N - 1$ elements to find the location of element N.

Haves: ROOT (lookup index included for variable-length data). Arrow and OAMap (offset columns act as indexes to nested data). However, if data are compressed, the entire basket/page/partition must be decompressed to access a single element.

Have nots: ProtoBuf, Thrift, Avro, and any rowwise format without a lookup index. JSON, CSV, XML and any text-based format without a lookup index.

Uncompressed Parquet is *not* random accessible: its tightly packed encoding prevents this. You have to decode an entire data page to find an element by entry number (just as you would if it were compressed).

Accessibility: Database-style indexing?

Random accessibility allows you to jump to a desired *entry number;* database-style indexing allows you to jump to or quickly select data *by value.*

It still counts if the search is not constant time but $\mathcal{O}(\log N)$ (search through a tree). It does not count if the search is $\mathcal{O}(N)$.

Haves: Parquet has metadata to indicate whether the data are sorted and what the minimum/maximum values are in a batch ("zone maps").

Have nots: ROOT, Avro, JSON... formats that aren't actually used in databases.

Accessibility: RPC protocol?

Some formats were designed for archival files, some for Remote Procedure Calls (RPC), but many function as both.

An important consideration in RPC is to be able to send messages as soon as they're ready, which usually *favors rowwise formats over columnar formats*. Spark-Streaming is criticized because Spark's columnar data representation forces it to send data in "micro batches" (100 events per batch, for example).

RPC: JSON, ProtoBuf, Thrift, Avro, Cap'n Proto, and MessagePack were all designed for RPC, but work as file formats.

Archival: ROOT, HDF5, Parquet, and Google Flatbuffers were designed to be file formats. Objects defined by ROOT streamers (rowwise) could be messages. Google Flatbuffers was retrofitted to support messaging.

Apache Arrow was designed to be an in-memory format, shared among processes. This is mostly for large, immutable buffers of data, but it also has a messaging format for sending batches of columnar data using Google Flatbuffers as the "envelope."

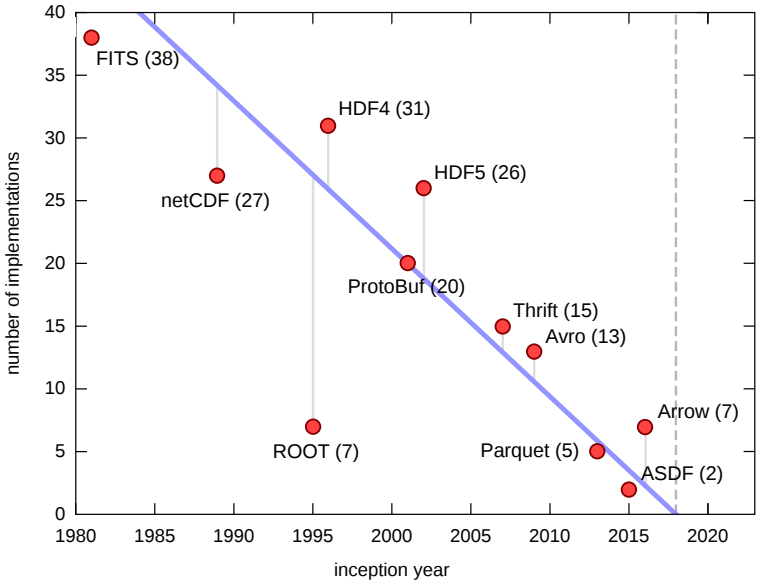
Community: Has specification?

Most data formats have specifications, human-readable documents describing what every byte means. This allows for independent implementations and a chance to clarify what the code is *supposed* to do (to determine what "wrong" means).

FITS	https://fits.gsfc.nasa.gov/standard30/fits_standard30aa.pdf
ProtoBuf	https://developers.google.com/protocol-buffers/docs/encoding
Thrift	**UNOFFICIAL:** https://erikvanoosten.github.io/thrift-missing-specification
Avro	http://avro.apache.org/docs/current/spec.html
Parquet	http://parquet.apache.org/documentation/latest
Arrow	https://arrow.apache.org/docs/memory_layout.html
HDF5	https://support.hdfgroup.org/HDF5/doc/H5.format.html
ROOT	some class headers like TFile and TKey, but not nearly enough info to read data

The ROOT file format has no specification, by choice (to allow for rapid change).

Community: Independent implementations?

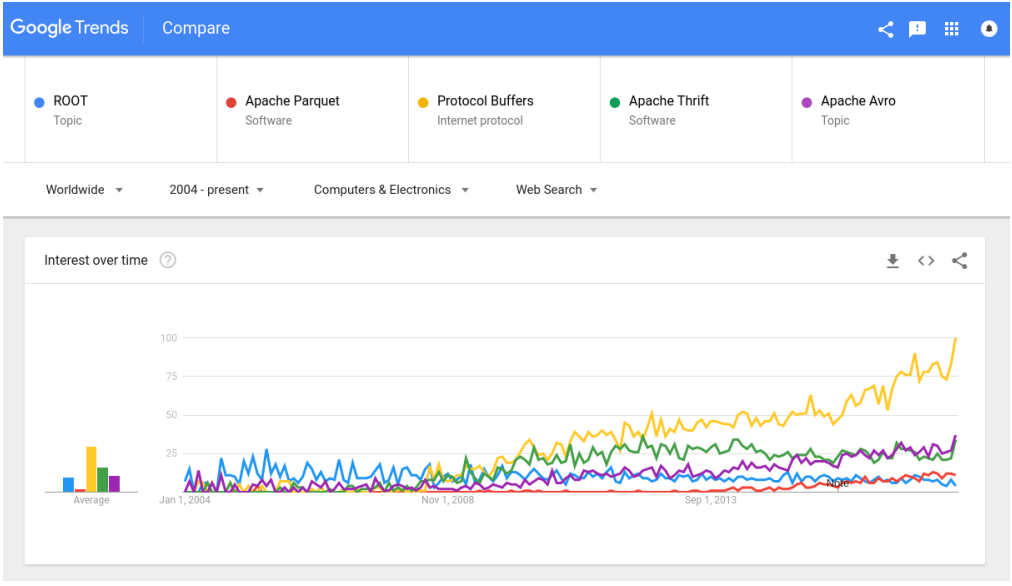


Contrary to a HEP attitude against "reinventing the wheel," it's common to reimplement I/O for popular file formats in different contexts.

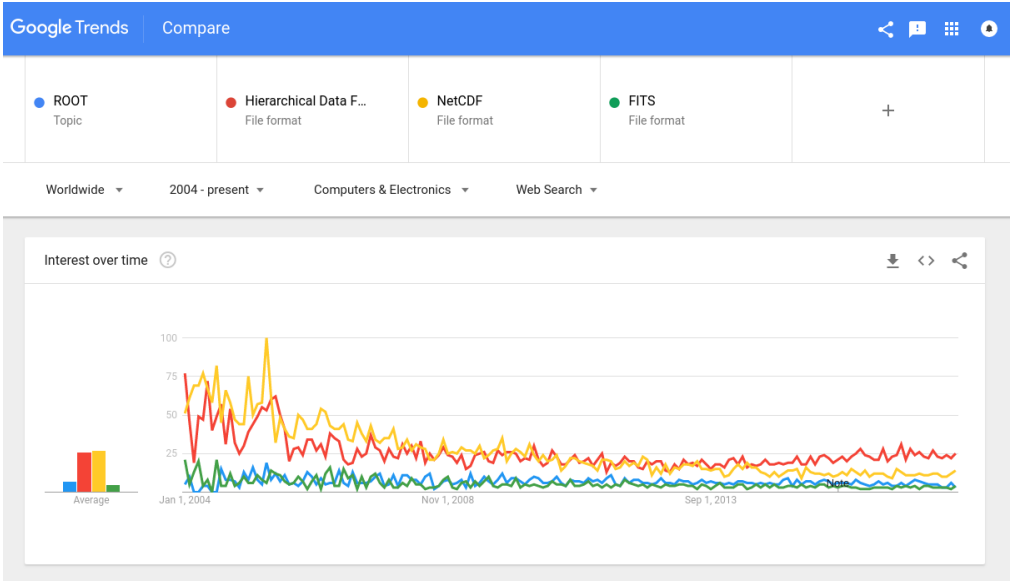
ROOT has 7: two C++ implementations, one in each of Java, Javascript, Go, Python, and Rust.

(Few are well-developed.)

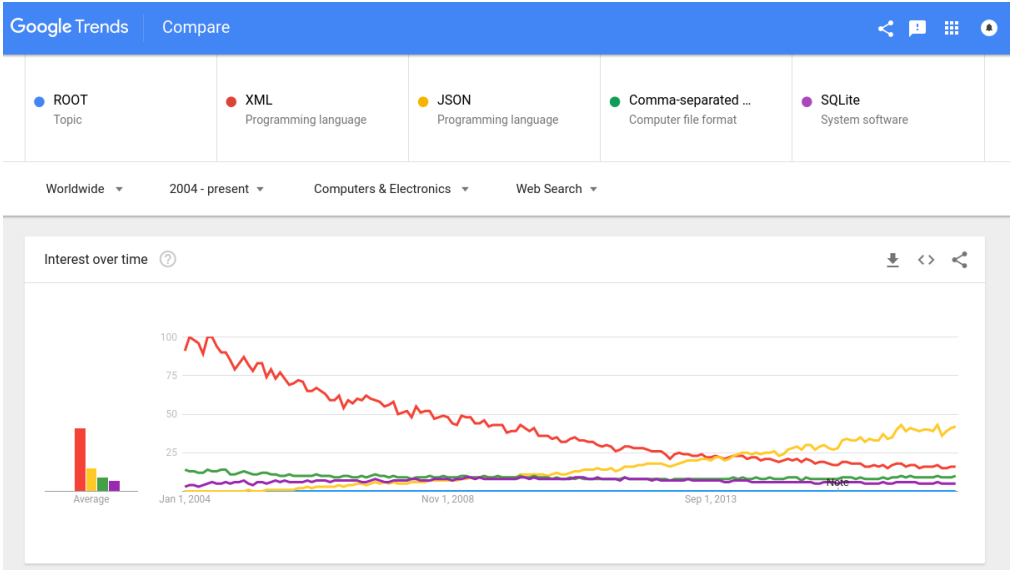
Community: Size of user base?



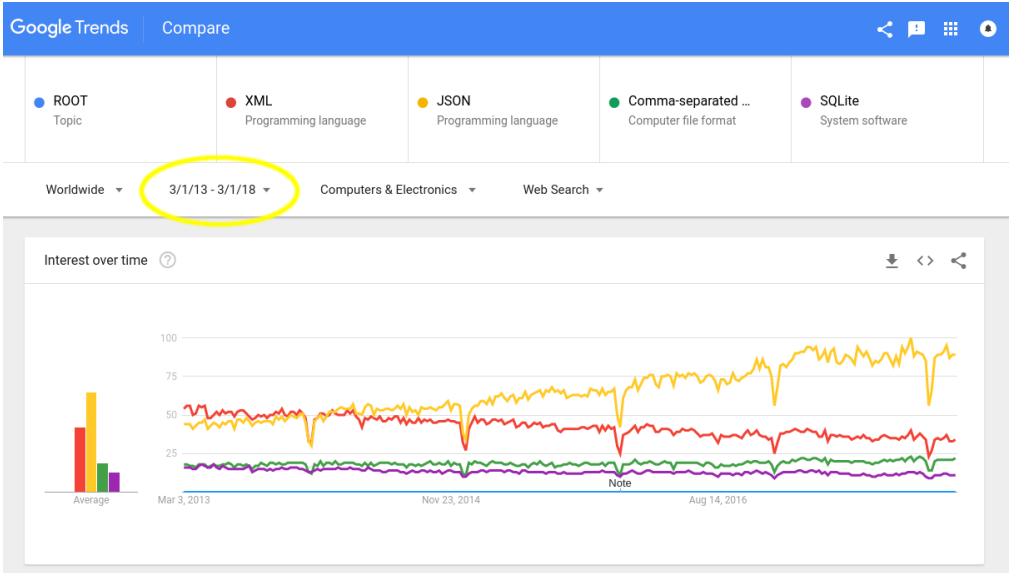
Community: Size of user base?



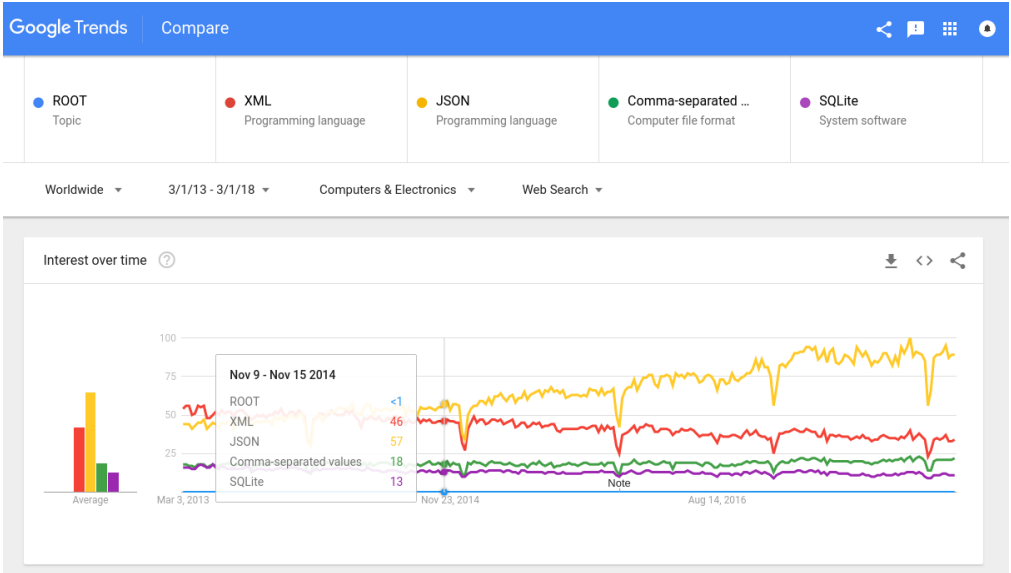
Community: Size of user base?



Community: Size of user base?



Community: Size of user base?



Community: Size of user base?