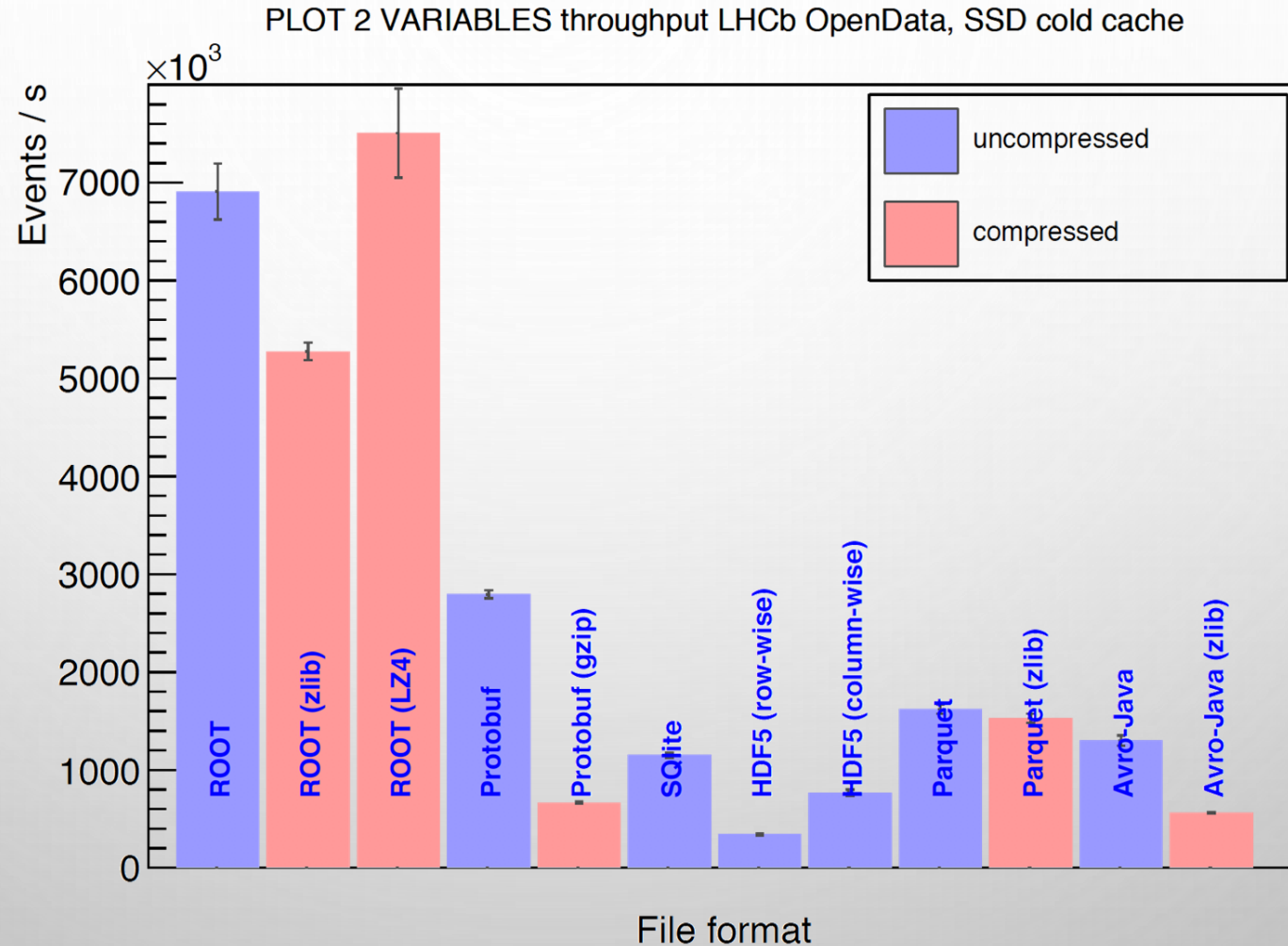


The background of the slide is a light gray gradient with several realistic water droplets of various sizes scattered across it. The droplets have highlights and shadows, giving them a three-dimensional appearance. The main title is centered in the upper half of the slide.

ROOT I/O PAST, PRESENT AND FUTURE

Philippe Canal for the ROOT Team.

COMPARISON WITH OTHER I/O SYSTEMS



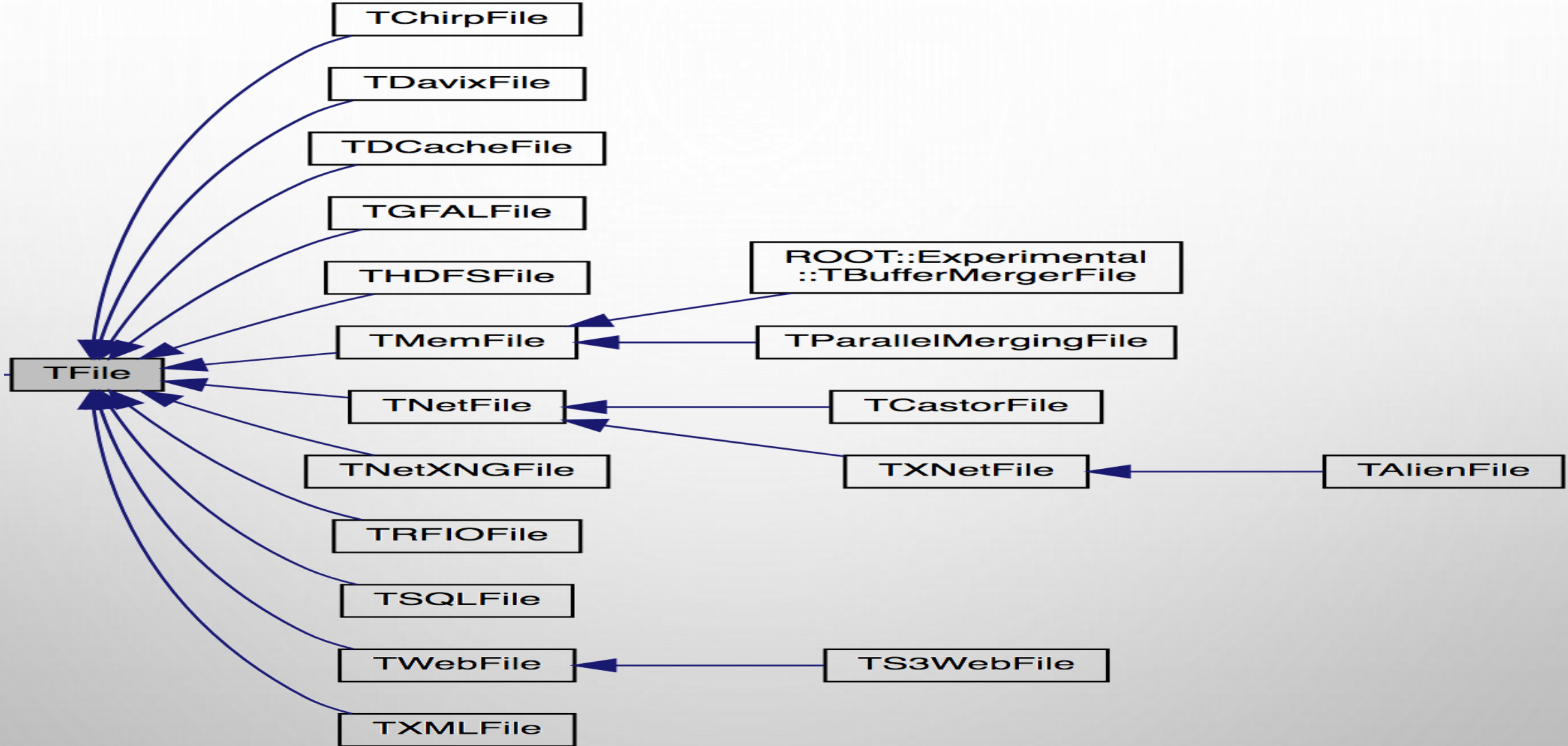
COLLABORATORS

- Ph Canal, FNAL/CMS
 - B. Bockelman, O. Shadura, J. Pivarski, Z. Zhang, DIANA
 - D. Piparo, J. Blomer, G. Amadio, CERN
 - Peter Van Gemmeren ANL/ATLAS
 - Amit Bashyal ANL/Summer
 - Sergey Linev GSI
- + **RDataFrame** Developers including Enrico Guiraud, Javier Cervantes Villanueva

THE ROOT FILE

- **TFiles** are *binary* and have:
 - a *header, records* and can be compressed (transparently for the user)
- **TFiles** have a logical “file system like” structure
 - e.g. directory hierarchy
- **TFiles** are **self-descriptive**:
 - Can be read without the code of the objects streamed into them
 - E.g. can be read from **JavaScript** and also in **Java, Go** and ... **Rust**

FLAVOUR OF TFILES



HOW DOES IT WORK IN A NUTSHELL?

- **C++ does not support native I/O** of its objects
- Key ingredient: reflection information - **Provided by ROOT**
 - What are the data members of the class of which this object is instance? I.e. How does the object look in memory?
- The steps, from memory to disk:
 1. Serialisation: from an object in memory to a blob of bytes
 2. Compression: use an algorithm to reduce size of the blob (e.g. zip, lzma, lz4)
 3. Writing to the physical resource (disk) via OS primitives

SERIALISATION: NOT A TRIVIAL TASK

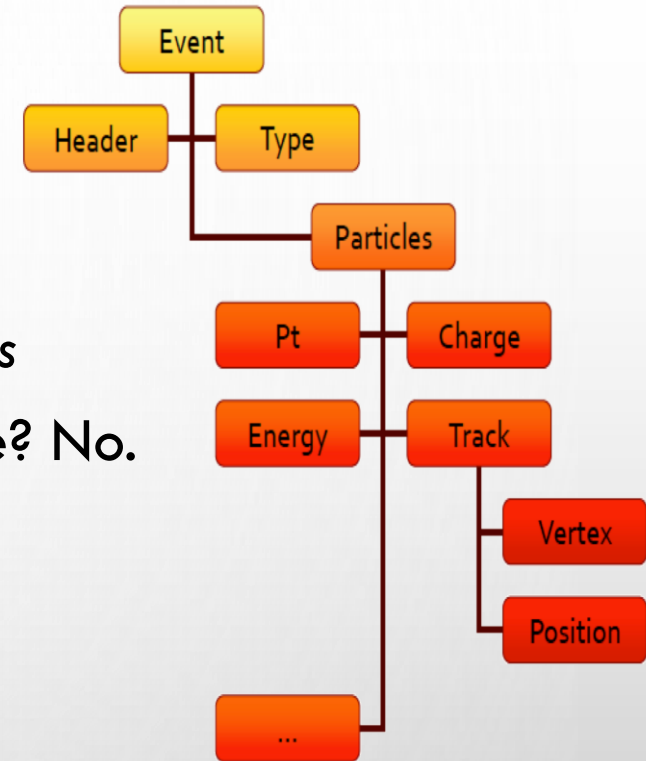
For example:

- Must be platform independent: e.g. 32bits, 64bits
 - Remove padding if present, little endian/big endian
- Must follow pointers correctly
 - And avoid loops ;)
- Must treat C++ standard constructs (STL, smart pointers)
- Support for custom serialization of numerical type
 - For example floating point that are double precision in memory stored in only 4 bytes
- Support for schema evolution
 - Object shape different on file and on disk.
- Must take into account customizations by the user
 - E.g. skip “transient data members”
 - I/O customization rule (transformation of data)

TTrees

TTREE

- High Energy Physics: many statistically independent *collision events*
- Create an event class, serialise and write out N instances on a file? No. Very inefficient!
- Organise the dataset in **columns**

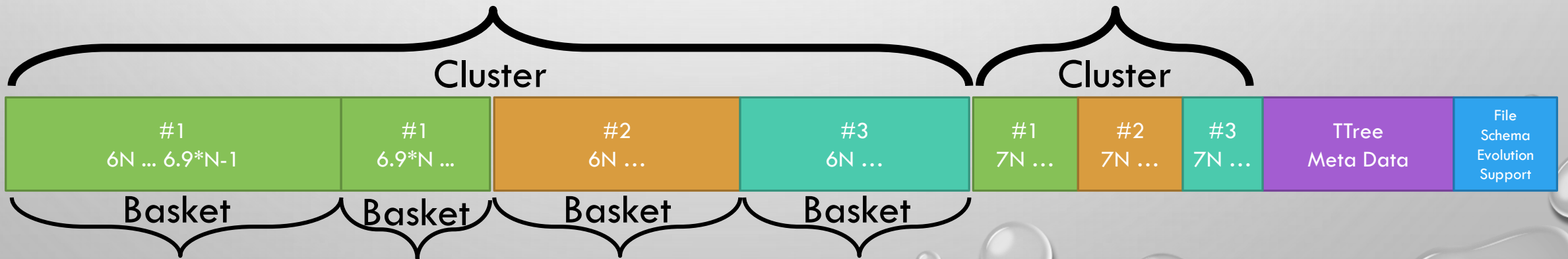
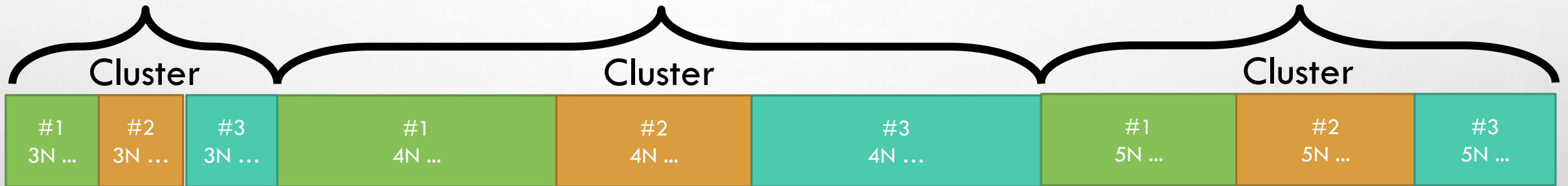
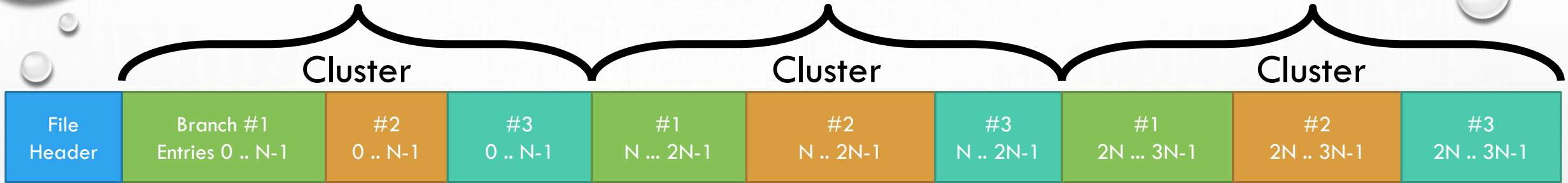


A columnar dataset in ROOT is represented by **TTree**:

- Also called *tree*, columns also called *branches*
- An object type per column, **any type of object**
- One row per *entry* (or, in collider physics, *event*)

pt_x	pt_y	pt_z	theta
entries			
			Branches Any kind of objects.

ANATOMY OF A FILE



OPTIMAL RUNTIME AND STORAGE USAGE

Runtime:

- Can decide what columns to read
- Prefetching, read-ahead optimisations

Storage Usage:

- Run-length Encoding (RLE). Compression of individual columns values is very efficient
 - Physics values: potentially all “similar”, e.g. within a few orders of magnitude - position, momentum, charge, index

COMPARISON WITH OTHER I/O SYSTEMS

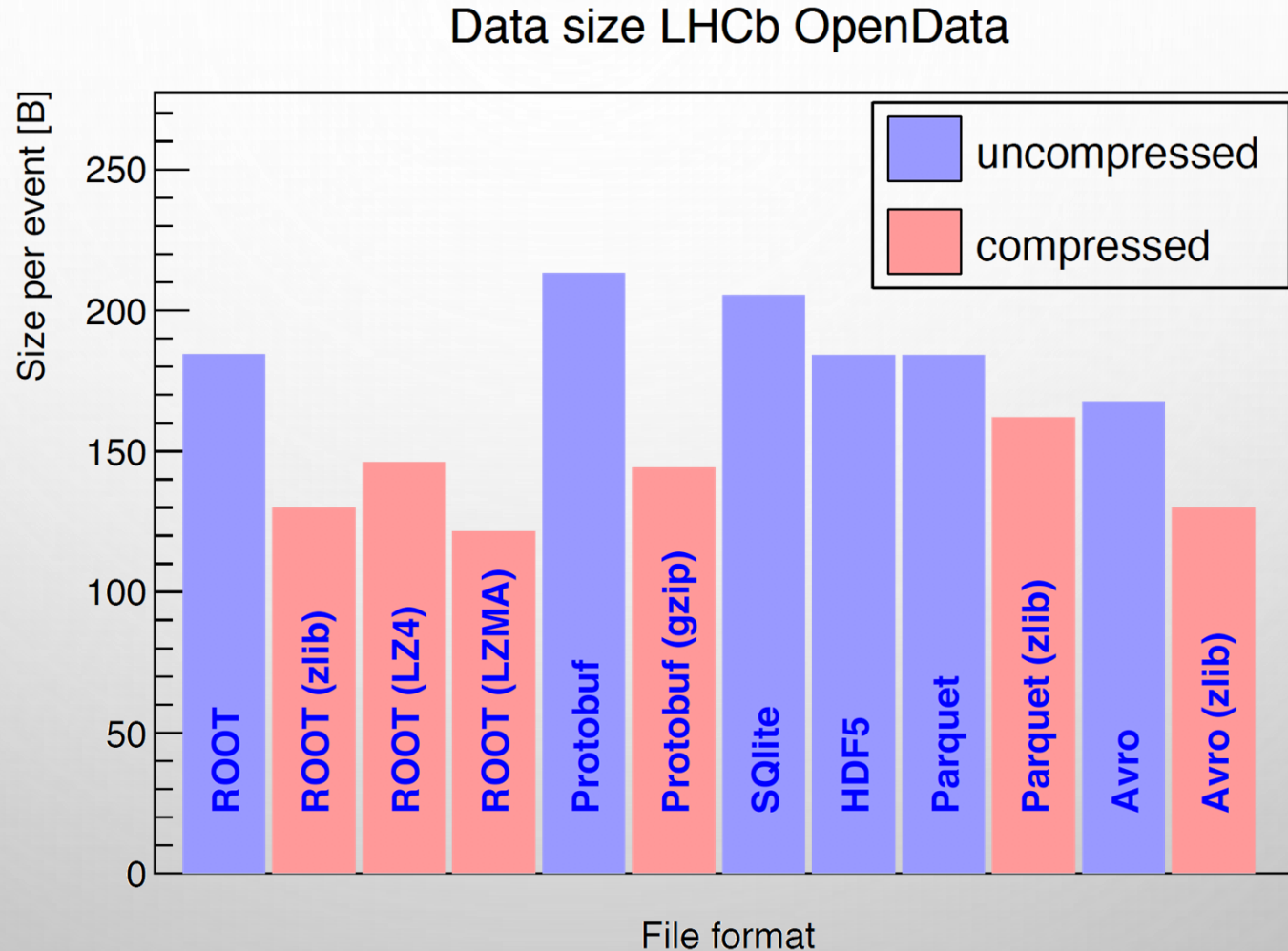
	ROOT	PB	SQLite	HDF5	Parquet	Avro
Well-defined encoding	✓	✓	✓	✓	✓	✓
C/C++ Library	✓	✓	✓	✓	✓	✓
Self-describing	✓	✘	✓	✓	✓	✓
Nested types	✓	✓	?	?	✓	✓
Columnar layout	✓	✘	✘	✓	✓	✘
Compression	✓	✓	✘	?	✓	✓
Schema evolution	✓	✘	✓	✘	?	?

✓ = supported

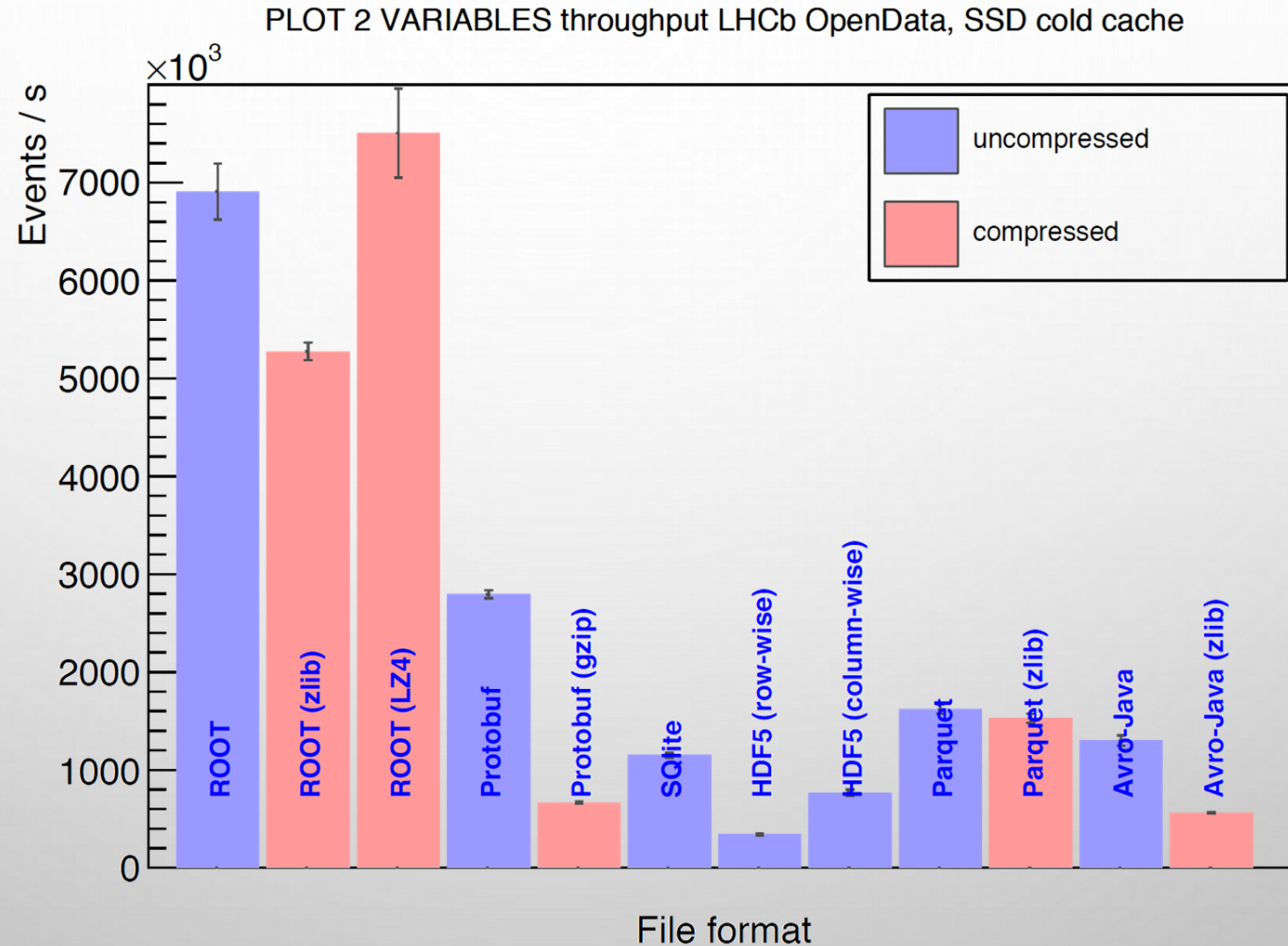
✘ = unsupported

? = difficult / unclear

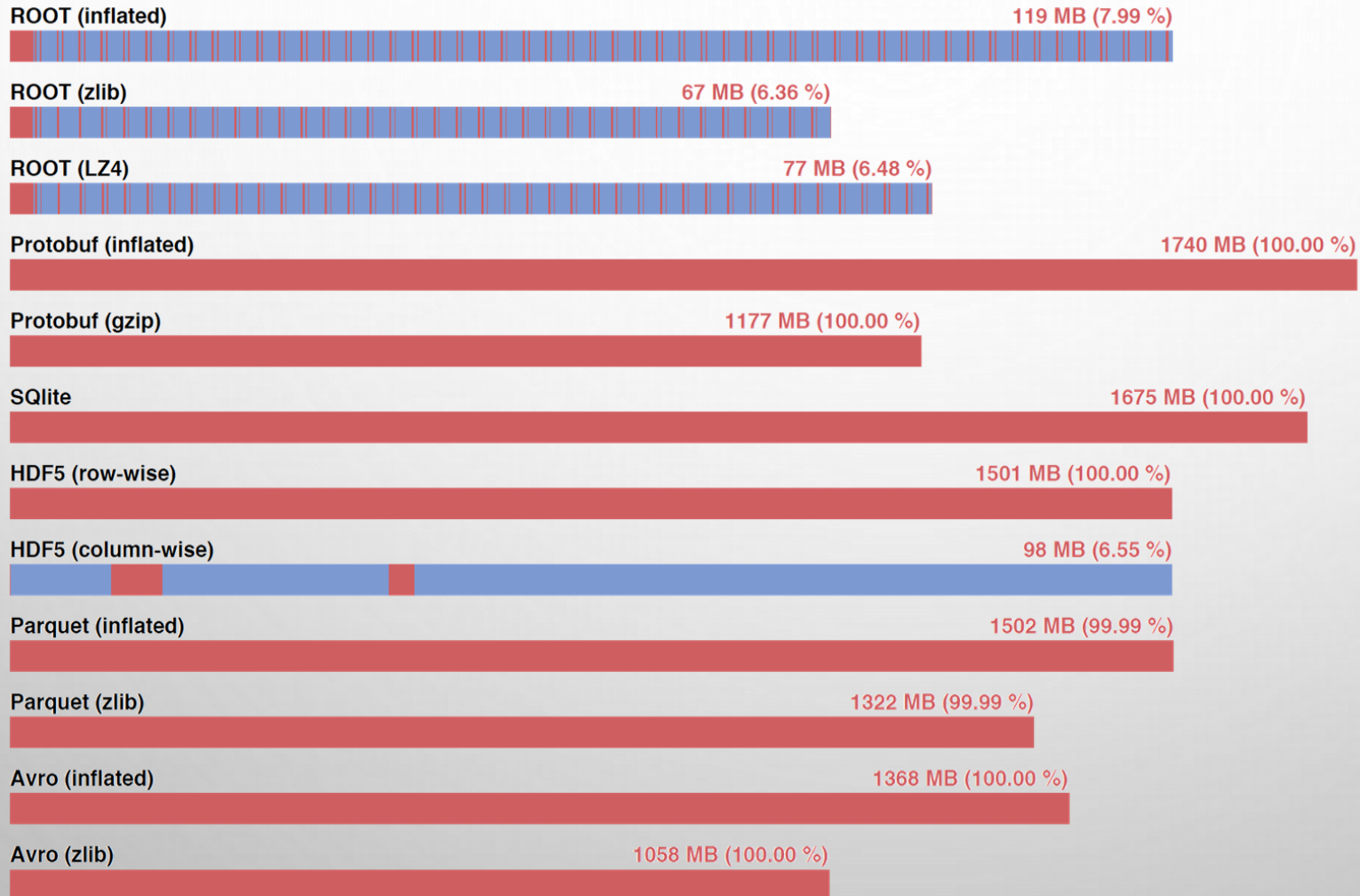
COMPARISON WITH OTHER I/O SYSTEMS



COMPARISON WITH OTHER I/O SYSTEMS




I/O PATTERNS



The less you read (red sections), the faster

Lastest Improvements

TFILE IMPROVEMENTS

- **StreamerInfo record no longer processed if (hash) already seen**
 - hash is sha256
 - save time and global lock contention (3x runtime 8 logical cores with small files, ~1 MB, added to a TChain)
- LZ4 compression will soon be the default and can be read by:
 - v5.34/38
 - v6.08/06 [not yet released]
 - v6.10/08, v6.12/02, v6.14/00

See Oksana's presentation next!
- Significant improvement in **JSON/XML** support

TTREE IMPROVEMENTS

- Improvement in *TTreeCache FillBuffer* and extension of *TTreePerfStats*
- Revised treatment of tree name in *TChain* URLs
 - Limited and Deprecated: `root://servername/filename.root/treename`
- Added an experimental feature that allows the IO libraries to skip writing out redundant information for some split classes, resulting in disk space savings. This is disabled by default and may be enabled by setting:

```
ROOT::TIOFeatures features;  
features.Set(ROOT::Experimental::EIOFeatures::kGenerateOffsetMap);  
ttree_ref.SetIOFeatures(features);
```

Saving: 4 bytes per entries per branch (Split collections)



DATA FRAME: READING ARBITRARY DATASETS WITH DATA SOURCE

- **RDataSource** interface available to the user to read arbitrary datasets
- Potentially any columnar dataset can be read into R
 - Current examples: **CSV**, **Arrow**, in-flight data, **Atlas XAOD** (demonstrator)
- Same analysis, different source
- Can always convert to **T** (full “out of capability” of **RDataFrame**!)

See RDataFrame Presentation
By Enrico G.
at 14:30 on Monday

INCREASED CONCURRENCY

- ***RDataFrame*** can write same tree from many threads
- Introduced ***ROOT::TReentrantRWLock*** [re-entrant for both reading and writing and read to write transition]
 - Reentrant read-write lock with a configurable internal mutex/lock and a condition variable to synchronize readers and writers when necessary. Optimize for fast ‘reading’ while still fair to ‘writers’.
- Some ***TCollection*** now have a thread-safe mode (***THashList***, ***THashTable***, ***TList*** and ***TObjArray***)
- ***TROOT::RecursiveRemove*** is now thread safe and more scalable
 - User overload of ***TObject::RecursiveRemove*** needs update to be thread-safe

TBRANCH AND TTREE

TBranch::BackFill to be used instead of ***TBranch::Fill*** to create a well clustered file when catching up with the number of entries already in the ***TTree***:

```
for(auto e = 0; e < tree->GetEntries(); ++e) { // loop over entries.  
    for(auto branch : branchCollection) {  
        ... Make change to the data associated with the branch ...  
        branch->BackFill();  
    }  
}
```

TTREE 2 NUMPY

- Convert in PyROOT a TTree to a numpy array: TTree.AsMatrix

```
myTree # Contains branches x and y of type float

# Convert to numpy array and apply numpy methods
myArray = myTree.AsMatrix()
m = np.mean(myArray, axis = 0)

# Read only specific branches, specify type
xAsInts = myTree.AsMatrix(columns = ['x'], dtype = 'int')
```

SECONDARY TTREE CACHE MISS

- Off by default.
- Tracks all branches used **after** the learning phase (cache misses).
- On cache miss, prefetch one basket for each of those branches

- Helps in the case when infrequently accessed branches are accessed together.
 - For example: select and skim

- NOTE - when this mode is enabled, the memory dedicated to the cache will up to double if there cache misses.

PRELOADING AND RETAINING UNCOMPRESSED CLUSTERS

- Can prevent additional reads from occurring when reading events out of sequence.
- To decompress a full cluster at time
 - Use `TTree::SetClusterPrefetch`.
- Retain previous cluster by setting `MaxVirtualSize` to a negative value
 - Absolute value indicates how many additional clusters will be kept in memory

Future ...

2018 PRIORITIES

- Tweaks to compression algorithm choice, e.g. **LZ4**, studying **Zstd**, **preconditioning**
- Increasing C++ standard support: *std::shared_ptr*, *std::variant*
- Increase internal parallelism: **unzipping**, **branch reading**
- Increase read performance, e.g. with “bulk I/O”
- Reduce file size: **work on redundant info**, **re-use compression dictionaries**
- Parallel *TTree/TFile* Merging over *MPI* on-HPC (Summer work @ ANL).
- Increasing connection to ML world (via NumPy Arrays, RTensor)

TTREE/TFILE FOR V7: “RFOREST” PROTOTYPES

- Why is it needed:
 - provide a thread-safe I/O
 - offer modern C++ iterator/cursor interfaces to **ROOT** data sets
 - offer a well-defined bulk I/O interface
 - allow for arbitrarily nested split collections
 - open the door to (more easily) store time series data in **ROOT**
- Who can use it:
 - other **ROOT** code and **ROOT** users
- Why now:
 - I/O capabilities and interfaces are the foundation of other tasks, e.g. parallel histogram filling
 - New column-wise storage formats/libraries are rising (e.g. **Parquet**, **Arrow**), **ROOT** should not fall behind in terms of performance/features.

FURTHER DOWN THE LINE

➤ Pass-through I/O

- Reduce copy to a minimal especially for ‘very high speed device’
- Challenges: compression, platform independence, padding and alignment rules
- Enables also: faster inter-process communication.

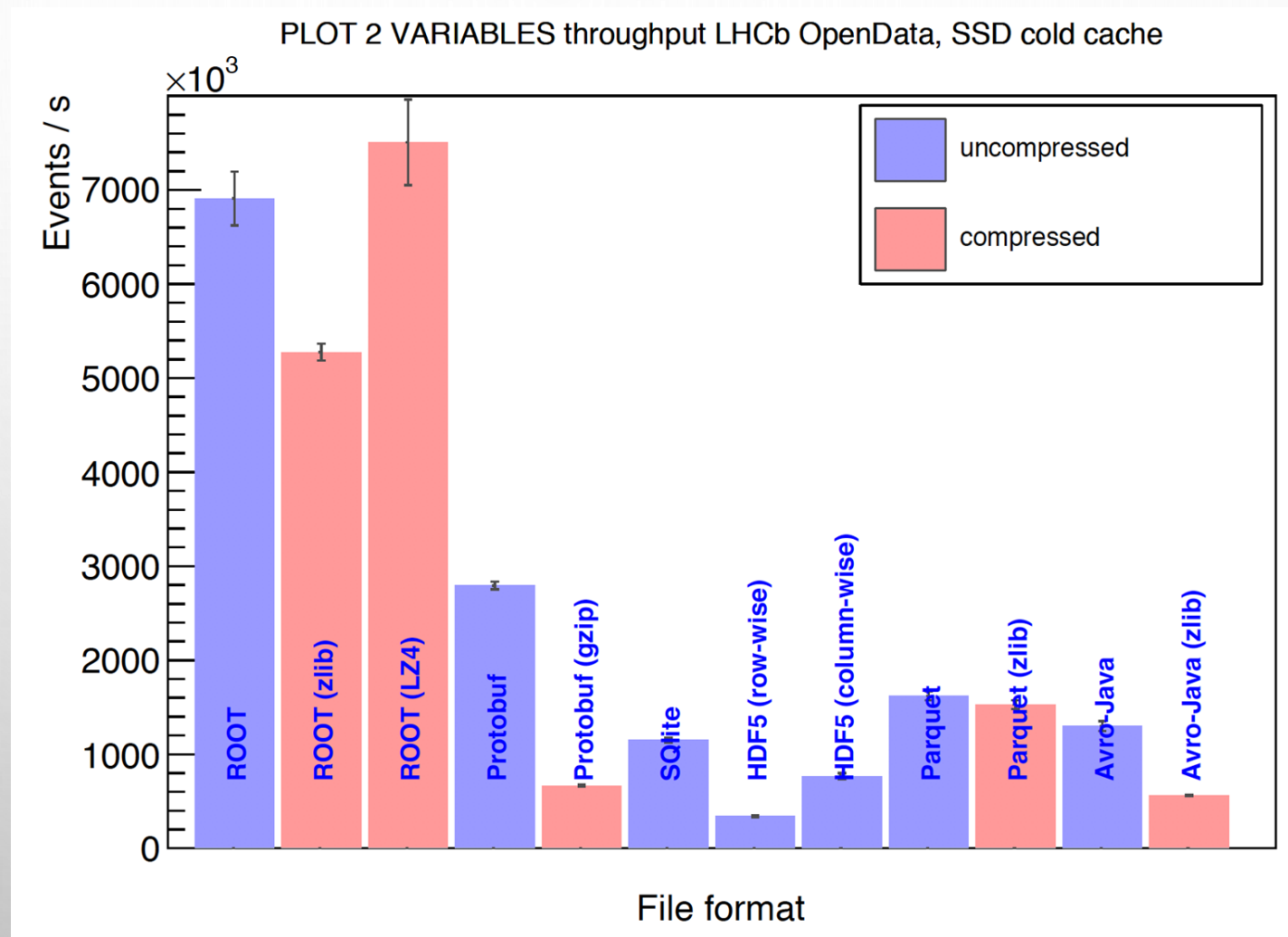
➤ Object Stores

- Main advantages: data de-duplication and “cache” sharing (and maybe indexing)

➤ Improve efficiency for many-core/many-processor architecture

- i.e. when all processes should not (can not) be writing to the (same) disk(s)

COMPARISON WITH OTHER I/O SYSTEMS



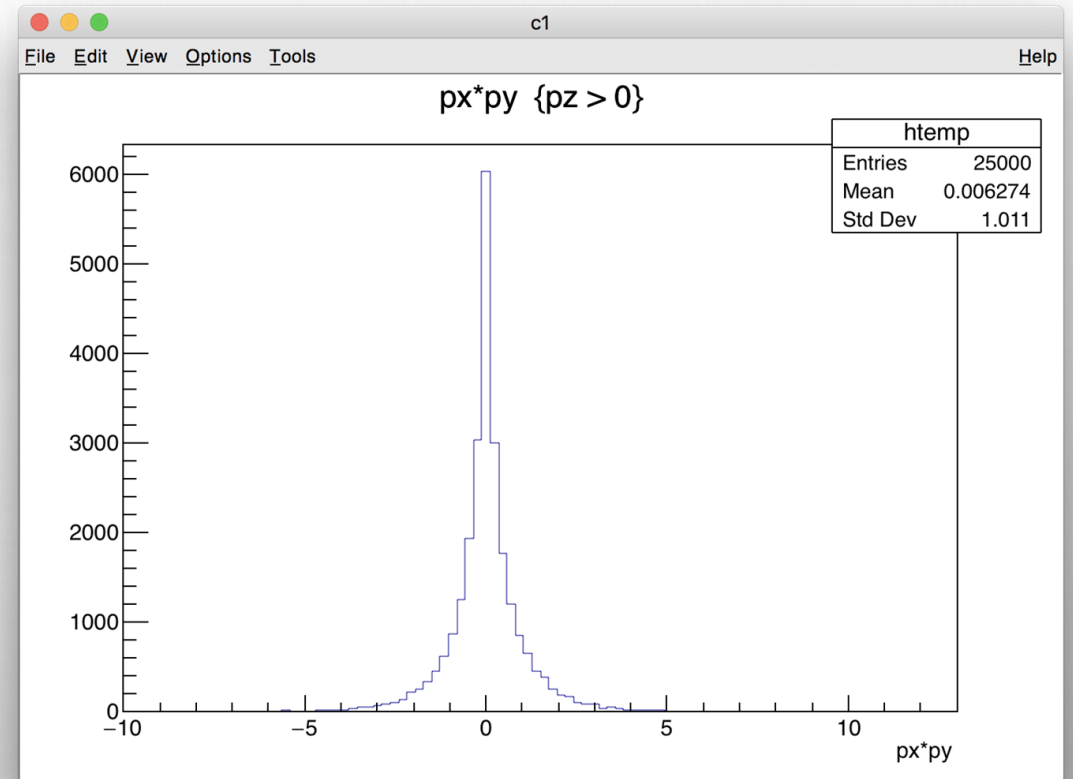
Backups Slides

ONE LINE ANALYSIS

Works for all types of
columns, not only numbers!

```
TFile f(filename);  
TTree *mytree;  
f.GetObject("tree", mytree);  
mytree->Draw("px * py", "pz > 0");
```

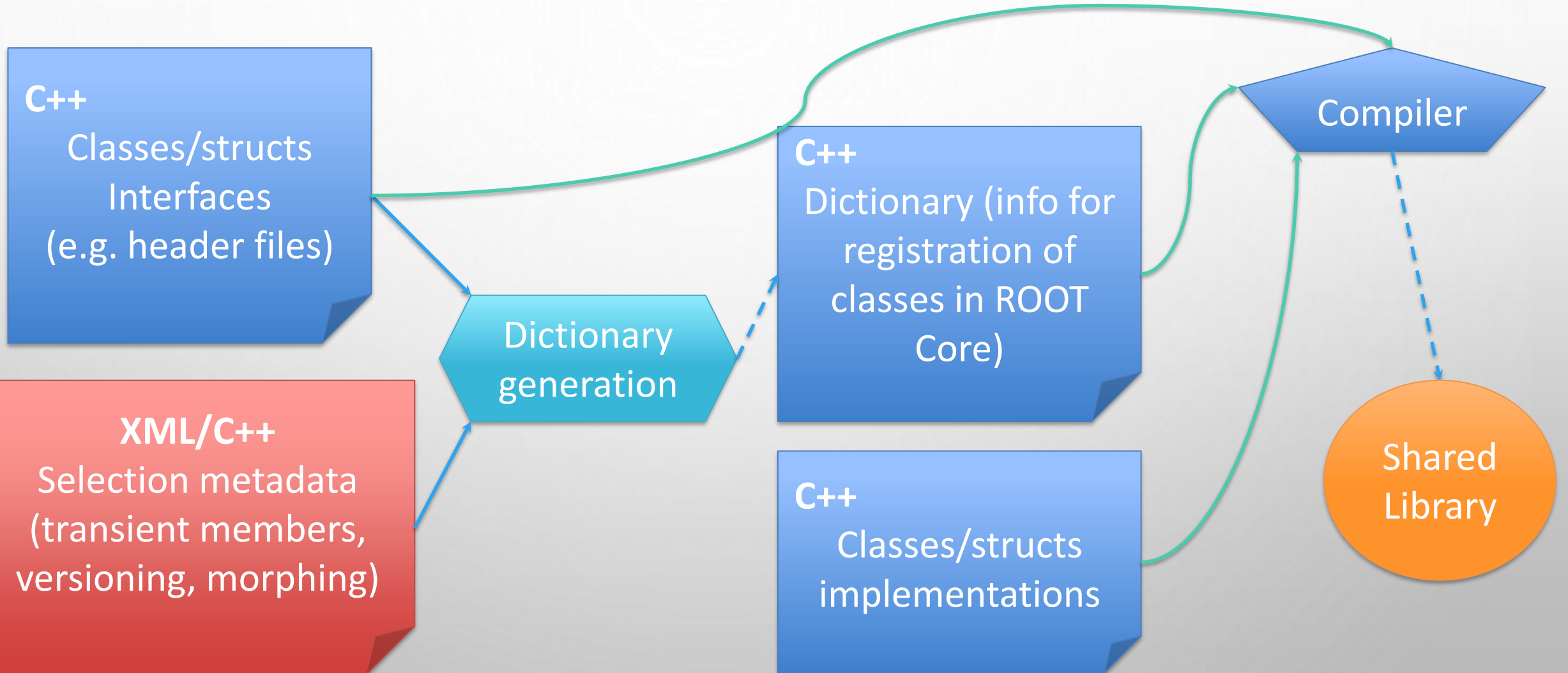
```
RDataFrame d(*mytree);  
auto h2 = d.Filter("pz > 0").Histo1D("px * py");  
h2.Draw();
```



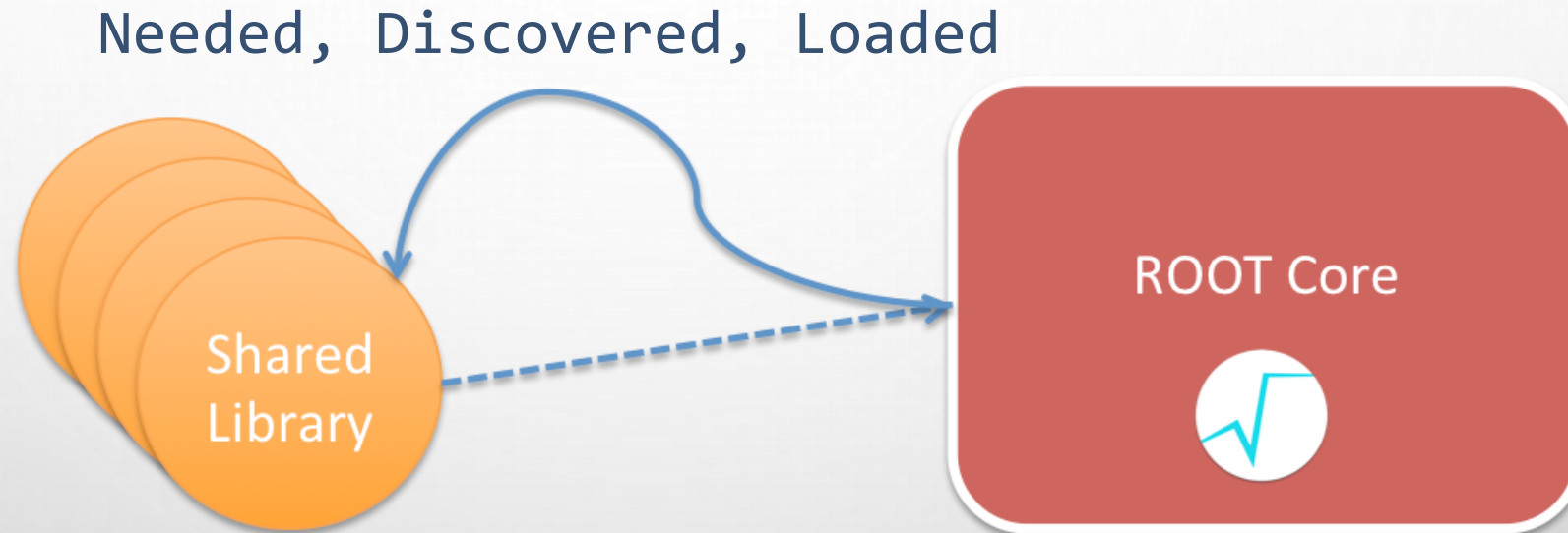
A WELL DOCUMENTED FILE FORMAT

Byte Range	Record Name	Description
1->4	"root"	Root file identifier
5->8	fVersion	File format version
9->12	fBEGIN	Pointer to first data record
13->16 [13->20]	fEND	Pointer to first free word at the EOF
17->20 [21->28]	fSeekFree	Pointer to FREE data record
21->24 [29->32]	fNbytesFree	Number of bytes in FREE data record
25->28 [33->36]	nfree	Number of free data records
29->32 [37->40]	fNbytesName	Number of bytes in TNamed at creation time
33->33 [41->41]	fUnits	Number of bytes for file pointers
34->37 [42->45]	fCompress	Compression level and algorithm
38->41 [46->53]	fSeekInfo	Pointer to TStreamerInfo record
42->45 [54->57]	fNbytesInfo	Number of bytes in TStreamerInfo record
46->63 [58->75]	fUUID	Universal Unique ID

PERSISTENCY



INJECTION OF REFLECTION INFORMATION



Now ROOT “knows” how to serialise the instances implemented in the library (series of data members, type, transiency) and to write them on disk in row or column format.

TFILE IN ACTION

```
TH1F* myHist = nullptr;  
TFile f("myfile.root");  
f.GetObject("h", myHist);  
myHist->Draw();
```

RECENT IMPROVEMENTS

- **StreamerInfo record no longer processed if (hash) already seen**
 - hash is sha256
 - save time and global lock contention (3x runtime 8 logical cores with small files, ~1 MB, added to a TChain)
- TBufferFile has been split in TBufferText and TBufferIO
 - Will allow to remove the lower level virtual function call
- StreamerInfo record now compressed with same level (or lack thereof) as the rest of the file.
- Allow writing temporary objects (with same address) in the same TBuffer(s).
 - new parameter cacheReuse added to TBuffer*::WriteObject
- Switch (and fix) Parallel unzipping to use TBB tasks.
 - mutually exclusive with TTree::GetEntry parallelised with TBB
- Added TKey::ReadObject<typeName>

MORE CHANGES

See Oksana's presentation next!

- LZ4 compression will soon be the default and can be read by:
 - v5.34/38
 - v6.08/06 [not yet released]
 - v6.10/08
 - v6.12/02
 - v6.14/00
- Revised treatment of tree name in TChain URLs
 - Support for the ill-defined way to pass the tree name in the url as '/tree_name' is limited to cases where the substring '.root' is contained in the file name but not in the tree name.
 - This syntax is now deprecated.

JSON / XML

- Implemented reading of objects data from JSON
- Provide `TBufferJSON::ToJSON()` and `TBufferJSON::FromJSON()` methods
- Provide `TBufferXML::ToXML()` and `TBufferXML::FromXML()` methods
- Converts NaN and Infinity values into null in JSON, there are no other direct equivalent
- Compaction of arrays within the JSON output

ROOT GLOBAL LOCK: NOW A READ-WRITE LOCK

- Resolved the race conditions inherent to the use of the **RecursiveRemove** mechanism.
- Introduced `ROOT::TReentrantRWLock`,
 - Reentrant read-write lock with a configurable internal mutex/lock and a condition variable to synchronize readers and writers when necessary.
- Allows a single reader to take the write lock without releasing the reader lock. It also allows the writer to take a read lock. In other words, the lock is re-entrant for both reading and writing.
- Tries to make faster the scenario when readers come and go but there is no writer. In that case, readers will not pay the price of taking the internal mutex.
- Tries to be fair with writers, giving them the possibility to claim the lock and wait for only the remaining readers, thus preventing starvation.

ROOT GLOBAL LOCK: NOW A READ-WRITE LOCK

- Switched the ROOT global to be a `ROOT::TReentrantRWLock` and renamed it `ROOT::gCoreMutex`. The old name `gROOTMutex` and `gInterpreterMutex` are deprecated and may be removed in future releases.
- Added `TReadLockGuard`, `TWriteLockGuard`, `R__READ_LOCKGUARD` and `R__WRITE_LOCKGUARD` to take advantage of the new lock. The legacy `TLockGuard` and `R__LOCKGUARD` use the write lock.
- Improved scaling of `TROOT::RecursiveRemove` in the case of large collection.
- Added a thread safe mode for the following ROOT collections: `THashList`, `THashTable`, `TList` and `TObjArray`. When ROOT's thread safe mode is enabled and the collection is set to use internal locks by:
 - `collection->UseRWLock();`
- all operations on the collection will take the read or write lock when needed, currently they shared the global lock (`ROOT::gCoreMutex`).

SIGNIFICANT REVAMP OF TTREECACHE::FILLBUFFER

The new scheme insures a much more stable and efficient behavior in case of low memory given by the user compared to the size of the buffers or 'odd' basket layout.

The basket collection is now done in 4 phases:

1. One basket per branch, basket must contain the requested entry and is not yet loaded or used,
2. Even out by adding baskets so that all branches reach the same entry (or close)
3. Add the remaining branches from the current cluster.
4. Add the basket from the beginning of the cluster to the current entry (if any)

then repeat the 4 steps for the next cluster.

SIGNIFICANT REVAMP OF TTREECACHE::FILLBUFFER

The iteration is stopped as soon as the cache is 'full' as defined by these rules:

- During step 1 of the first cluster, continue up to 4 times the user requested cache size
- During steps 2 to 4 of the first cluster, continue up to 2 times the user requested cache size
- During steps 2 to 4, the 'first' basket of a branch is accepted up to 4 times the user requested cache size (i.e as if it had been selected during the 1st step)
- During the other clusters, continue up to the user requested cache size
- A basket is rejected/skipped if its individual size is larger than the user requested cache size

In addition, upon seeing a cache miss, `FillBuffer` now detects if all the baskets in the cache have already been used (read from the cache) in which case we can discard them and load the next set of baskets.

TTREE PERF STATS

As a side effect, we now keep a record of which baskets are in the cache and which of those baskets have been used. The `TTreePerfStats` now keeps a complete log of all the baskets that are:

- loaded in the main cache (and how many times)
- loaded in the 'miss' cache (and how many times)
- used
- read directly (complete cache miss)

This will be helpful in understanding situation of over-read or slow operations.

I/O OF TEMPORARY OBJECTS.

- Add the ability to store the 'same' object several time (assumably with different data) in a single buffer. Instead of

```
while(...) { TObjArray arr; ... update the content of "arr" buffer << arr; }
```

- which would only really stream the array at the first iteration because it will be detected has having the same address and thus assumed to be the same object. We can now do:

```
while(...) { TObjArray arr; ... update the content of "arr" buffer.WriteObject(&arr, kFALSE); }
```

where the last argument of `WriteObject` tells the buffer do *not* remember this object's address and to always stream it. This feature is also available via `WriteObjectAny`.

I/O FORWARD INCOMPATIBLE FEATURES.

- Added a new mechanism for providing clean forward-compatibility breaks in a TTree (i.e., a newer version of ROOT writes a TTree an older version cannot read). When future versions of ROOT utilize an IO feature that this version does not support, ROOT will provide a clear error message instead of crashing or returning garbage data. In future ROOT6 releases, forward-compatibility breaks will only be allowed if a non-default feature is enabled via the ROOT::Experimental namespace; it is expected ROOT7 will enable forward-compatibility breaks by default.
- When a file using an unsupported file format feature is encountered, the error message will be similar to the following:

```
Error in <TBasket::Streamer>: The value of fIOBits (00000000000000000000000000001111110) contains unknown flags (supported flags are 00000000000000000000000000000001)
```

indicating this was written with a newer version of ROOT utilizing critical IO features this version of ROOT does not support. Refusing to deserialize.

- When an older version of ROOT, without this logic, encounters the file, the error message will be similar to the following:

```
Error in <TBasket::Streamer>: The value of fNevBufSize is incorrect (-72) ; trying to recover by setting it to zero
```

- Added an experimental feature that allows the IO libraries to skip writing out redundant information for some split classes, resulting in disk space savings. This is disabled by default and may be enabled by setting:

```
ROOT::TIOFeatures features;  
features.Set(ROOT::Experimental::EIOFeatures::kGenerateOffsetMap);  
ttree_ref.SetIOFeatures(features);
```

ROOT I/O PAST, PRESENT AND FUTURE

- What is ROOT I/O? What are its strengths? What are its weakness? How does it compare to alternatives? This presentation will address these questions and present performance contrasts with other solutions. We will then review the latest additions, fixes and features. Finally, we will give our vision of where we can take ROOT I/O to make it even better and more relevant to today's environment and challenges.

VERY RECENT ADDITIONS

- [\[TTree\] Create event clusters when TTree is flushed.](#)
- IO Features
- Rule scheduling improvements

ROOT FILE DESCRIPTION

