ROOT I/O

Philippe Canal for the ROOT Team

ROOT
Data Analysis Framework
https://root.cern

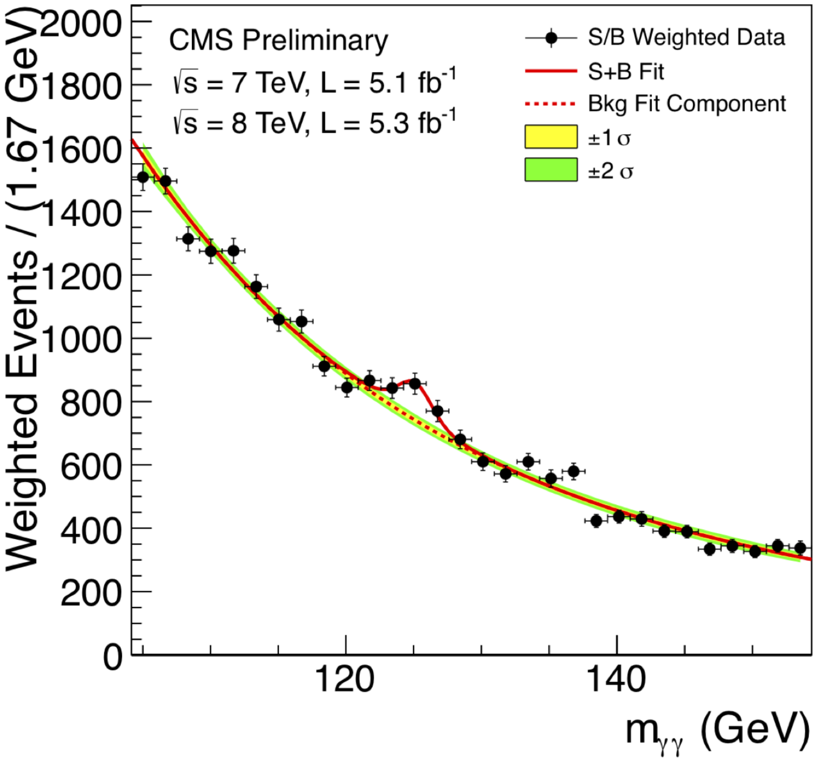
Resources

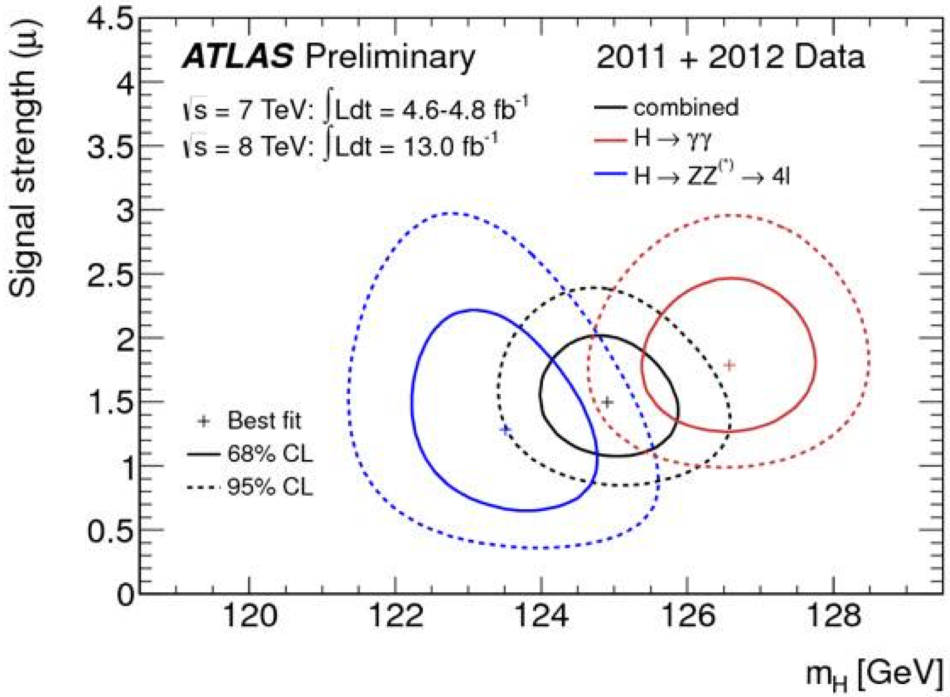
- ROOT Website: https://root.cern
- Introduction material: https://root.cern/getting-started
  - Includes a booklet for beginners: **the “ROOT Primer”**
- Reference Guide: https://root.cern/doc/master/index.html
- Training material: https://github.com/root-project/training
- Forum: https://root-forum.cern.ch

Introduction


LHCb
ATLAS
CERN Meyrin
CERN Prévessin
SPS 7 km
SUISSE
FRANCE
ALICE
CMS
LHC 27 km

What can you do with ROOT?





ROOT in a Nutshell

- ROOT is a software framework with building blocks for:
  - Data processing
  - Data analysis
  - Data visualisation
  - Data storage

An Open Source Project
We are on github
github.com/root-project
All contributions are warmly welcome!

- ROOT is written mainly in C++ (newer code in C++11/17 standard)
  - Bindings for Python available as well
- Adopted in High Energy Physics and other sciences (but also industry)
  - More than 1 EB of data in ROOT format
  - Fits and parameters' estimations for discoveries (e.g. the Higgs)
  - Thousands of ROOT plots in scientific publications
- Started in **1995**

ROOT in a Nutshell

ROOT can be seen as a collection of building blocks for various activities, like:

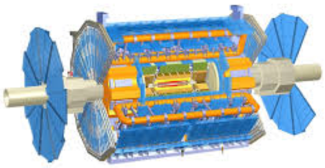
- **Data analysis: histograms, graphs, functions**
- **I/O: row-wise, column-wise** storage of any C++ object
- **Statistical tools** (RooFit/RooStats): rich modeling and statistical inference
- Math: **non trivial functions** (e.g. Erf, Bessel), optimised math functions
- **C++ interpretation**: full language compliance
- **Multivariate Analysis** (TMVA): e.g. Boosted decision trees, NN
- **Advanced graphics** (2D, 3D, event display)
- **Declarative Parallel Analysis**: RDataFrame
- And more: HTTP servering, JavaScript visualisation

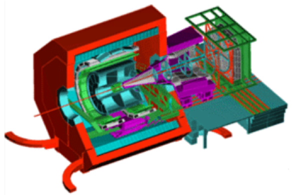
Unstar 595 Fork 433

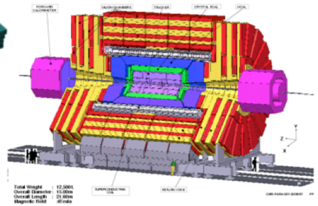
176 contributors

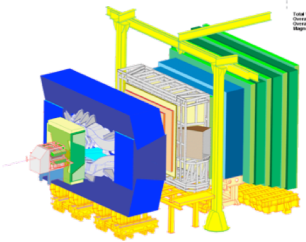
ROOT Application Domains

A selection of the experiments adopting ROOT









Analysis

Offline Processing

Event Selection, statistical treatment …

Further processing, skimming

Reconstruction

Data

Raw

Reco

…

Analysis Formats

Images

Event Filtering

Data Storage: Local, Network

Interpreter

- ROOT has a built-in interpreter : Cling
  - C++ interpretation: highly non trivial and not foreseen by the language!
  - One of its kind: Just In Time (JIT) compilation
  - A C++ interactive shell

```
$ root
root[0] 3 * 3
(const int) 9
```

- Can interpret “macros” (non compiled programs)
  - Rapid prototyping possible

- ROOT provides also Python bindings
  - Can use Python interpreter directly after a simple *import ROOT*
  - Possible to “mix” the two languages (see more later)

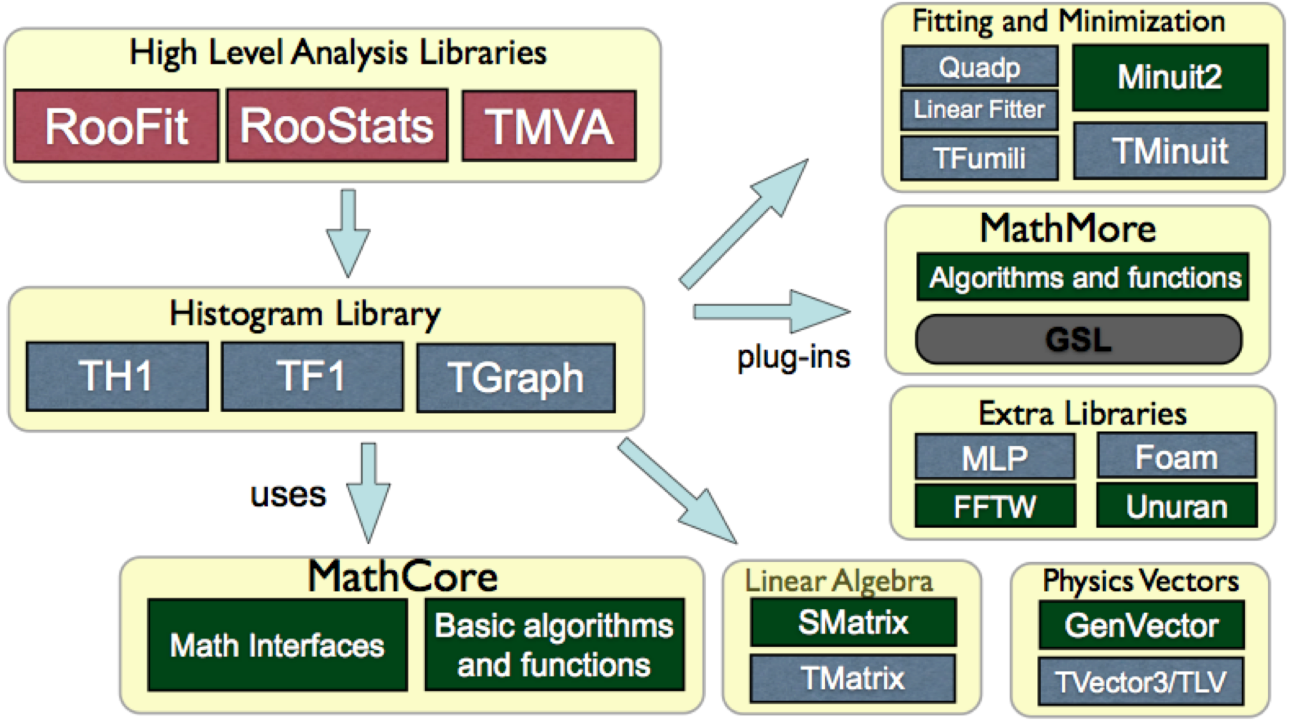
Persistency or Input/Output (I/O)

- ROOT offers the possibility to write C++ objects into files
  - This is impossible with C++ alone
  - Used the LHC detectors to write several petabytes per year
- Achieved with serialization of the objects using the reflection capabilities, ultimately provided by the interpreter
  - Raw and column-wise streaming
  - **No explicit** instrumentation needed in most cases.
- As simple as this for ROOT objects: one method - *TDirectoryFile::WriteObject*

Cornerstone for storage of experimental data

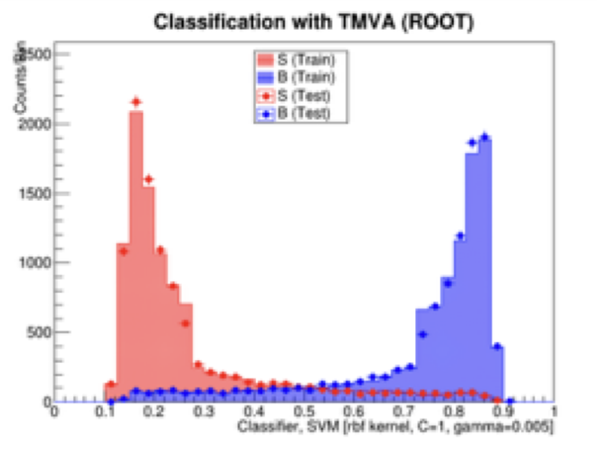
Mathematics and Statistics

- ROOT provides a rich set of mathematical libraries and tools for sophisticated statistical data analysis



TMVA : Toolkit for Multi-Variate data Analysis in ROOT

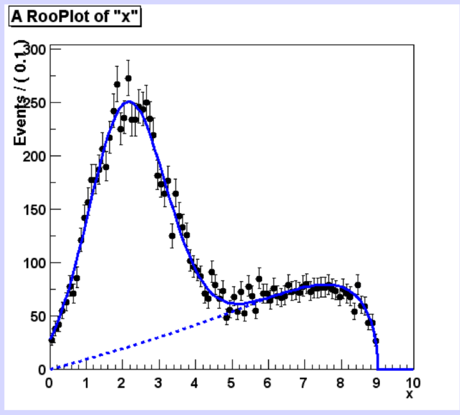
- provides several built-in ML methods including:
  - Boosted Decision Trees
  - Deep Neural Networks
  - Support Vector Machines



- and interfaces to external ML tools
  - scikit-learn, Keras (Theano/Tensorflow), R

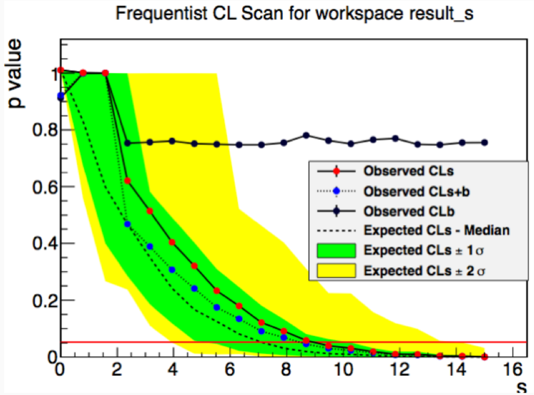
RooFit: Toolkit for Data Modeling and Fitting

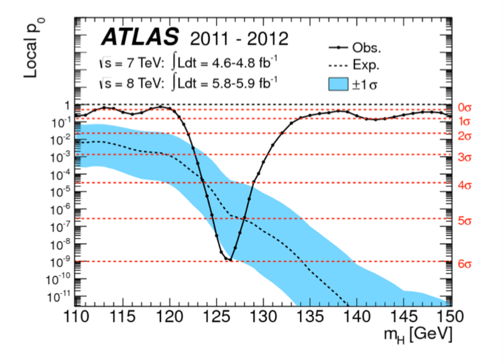
- functionality for building models: probability density functions (p.d.f.)
  - distribution of observables in terms of parameters $P(x;p)$
- complex model building from standard components
  - e.g. composition, addition, convolution,...
- RooFit models have functionality for
  - maximum likelihood fitting for parameter estimation
  - toy MC generation
  - visualization
  - sharing and storing (workspace)



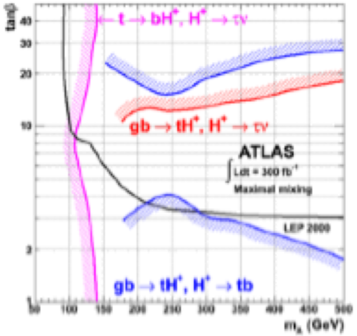
RooStats

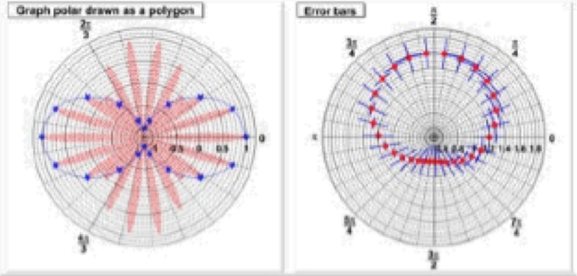
- Advanced Statistical Tools for HEP analysis. Used for :
  - estimation of Confidence/Credible intervals
  - hypotheses Tests
    - e.g. Estimation of Discovery significance
- Provides both Frequentist and Bayesian tools
- Facilitate combination of results

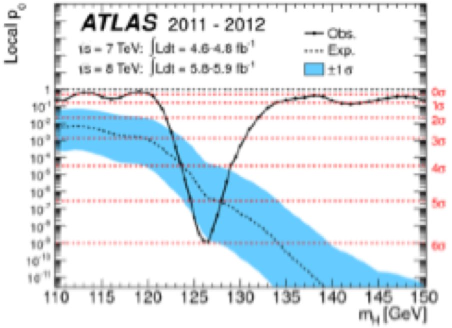


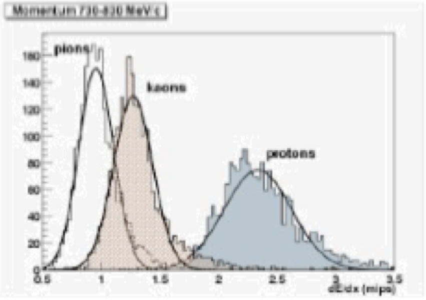


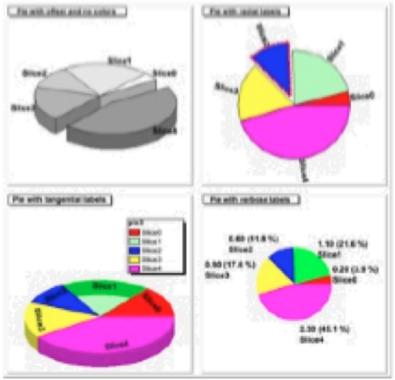
Many formats for data analysis, and not only, plots

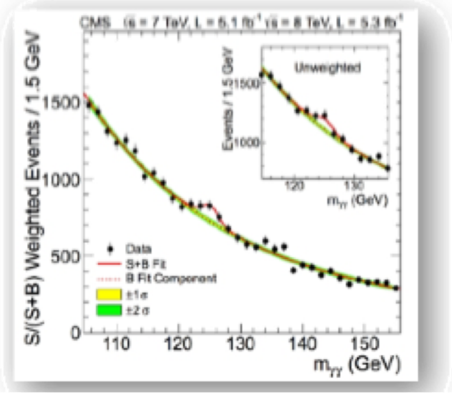




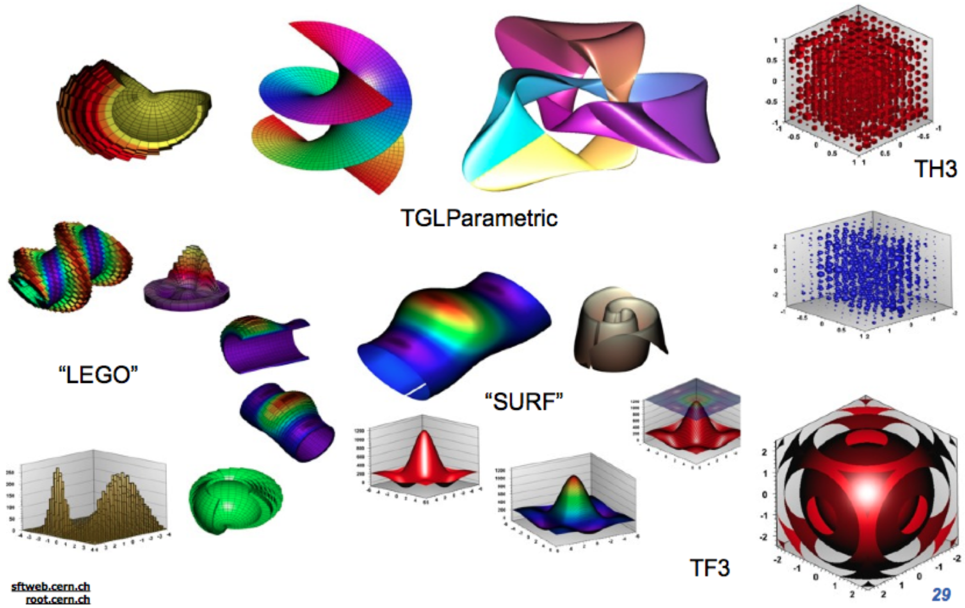


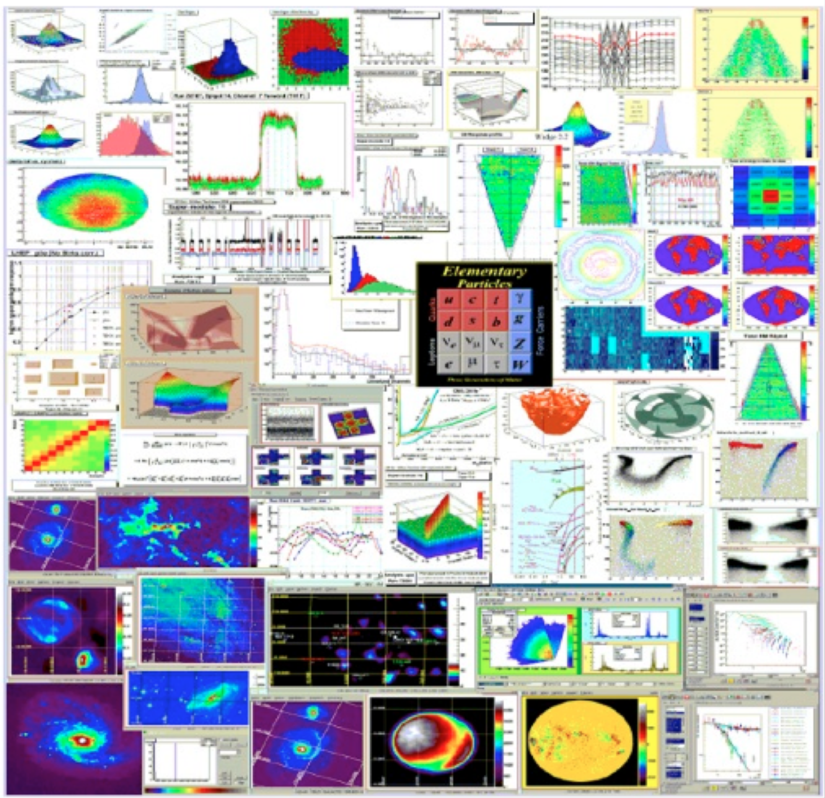






2D and 3D Graphics

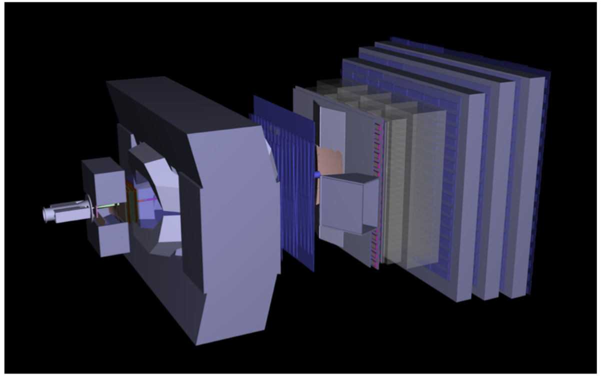




Can save graphics in many formats:
ps, pdf, svg, jpeg, LaTex, png, c, root …

JSROOT

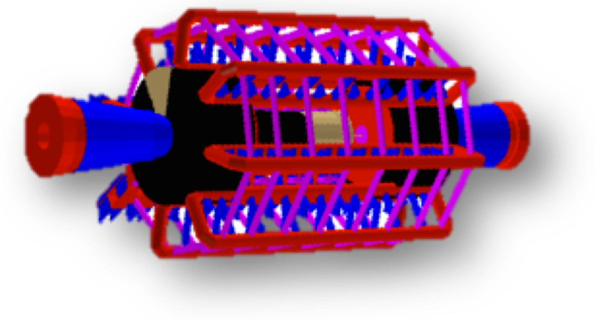
- JSROOT: a JavaScript version of ROOT graphics and I/O
- Complements traditional graphics
- Visualisation on the web or embedded in notebooks
- Basic functionality for exploring data in ROOT format

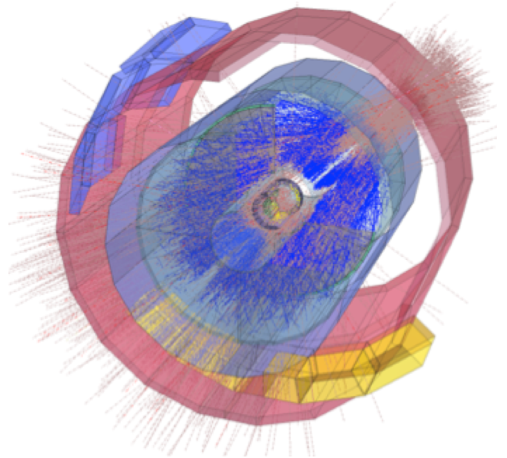


Parallelism

- Many ongoing efforts to provide means for parallelisation in ROOT
- Explicit parallelism
  - **TThreadExecutor** and **TProcessExecutor**
  - Protection of resources
- Implicit parallelism
  - **RDataFrame**: Declarative Parallel analysis
  - TTreeProcessor: process tree events in parallel
  - TTree::GetEntry: process of tree branches in parallel
- Parallelism is a prerequisite element for tackling data analysis during LHC Run III and HL-LHC

Many More Features!

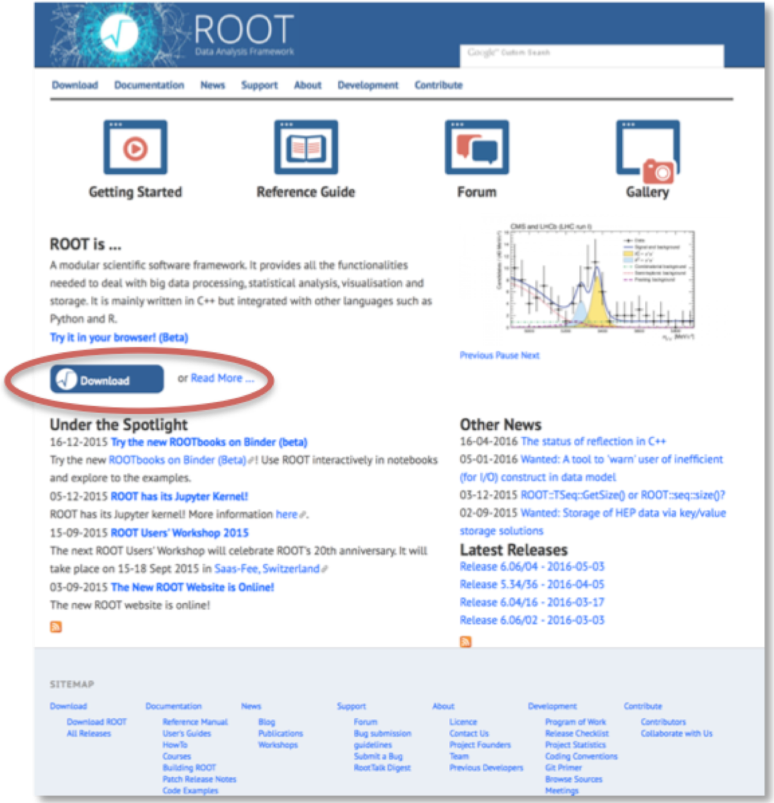




- Geometry Toolkit
  - Represent geometries as complex as LHC detectors
- Event Display (EVE)
  - Visualise particle collisions within detectors

https://root.cern

- ROOT web site: **the** source of information and help for ROOT users
  - For beginners and experts
  - Downloads, installation instructions
  - Documentation of all ROOT classes
  - Manuals, tutorials, presentations
  - Forum
  - …



The ROOT Prompt and Macros

The ROOT Prompt

- C++ is a compiled language
  - A compiler is used to translate source code into machine instructions
- ROOT provides a C++ **interpreter**
  - Interactive C++, without the need of a separate compiler, like Python, Ruby, Haskell …
    - Code is **Just-in-Time compiled!**
  - Allows reflection (inspect at runtime layout of classes)
  - Is started with the command:

```
root
```

  - The interactive shell is also called “ROOT prompt” or “ROOT interactive prompt”

ROOT As a Calculator

$$\frac{1}{1-x} = 1 + x + x^2 + x^3 + x^4 + \dots$$
$$= \sum_{n=0}^{\infty} x^n$$

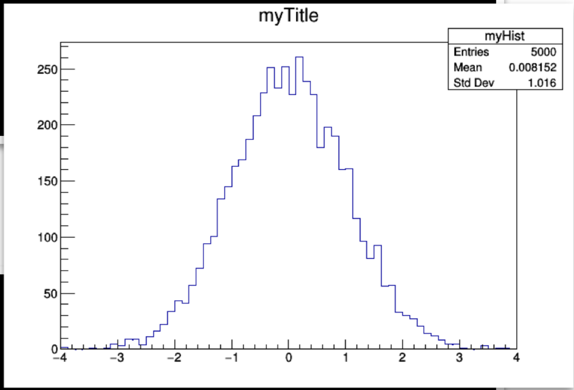
```
root [0] double x=.5
(double) 0.5
root [1] int N=30
(int) 30
root [2] double gs=0;
```

Here we make a step forward.
We declare **variables** and use a ***for*** control structure.

```
root [3] for (int i=0;i<N;++i) gs += pow(x,i)
root [4] std::abs(gs - (1/(1-x)))
(Double_t) 1.86265e-09
```

Example: C++ to Python

```
> root
root [0] TH1F h("myHist", "myTitle", 64, -4, 4)
root [1] h.FillRandom("gaus")
root [2] h.Draw()
```




```
> python
>>> from ROOT import TH1F
>>> h = TH1F("myHist", "myTitle", 64, -4, 4)
>>> h.FillRandom("gaus")
>>> h.Draw()
```

Dynamic C++ (JITting)

```
import ROOT
cpp_code = """
int f(int i) { return i*i; }
class A {
public:
  A() { cout << "Hello PyROOT!" << endl; }
};
"""
# Inject the code in the ROOT interpreter
ROOT.gInterpreter.ProcessLine(cpp_code)

# We find all the C++ entities in Python!
a = ROOT.A()   # this prints Hello PyROOT!
x = ROOT.f(3)  # x = 9
```

C++ code we want to invoke from Python

Dynamic C++ (JITting)

my_cpp_library.h

```cpp
int f(int i) { return i*i; }

class A {
public:
   A() { cout << "Hello PyROOT!" << endl; }
};
```

my_python_module.py

```python
# Make the header known to the interpreter
ROOT.gInterpreter.ProcessLine('#include "my_cpp_library.h"')

# We find all the C++ entities in Python!
a = ROOT.A()    # this prints Hello PyROOT!
x = ROOT.f(3)   # x = 9
```

Dynamic Library Loading

```cpp
int f(int i);
class A {
public:
  A();
};
```

my_cpp_library.h

```cpp
#include "my_cpp_library.h"
int f(int i) { return i*i; }
A::A() { cout << "Hello PyROOT!" << endl; }
```

my_cpp_library.cpp

my_cpp_library.so

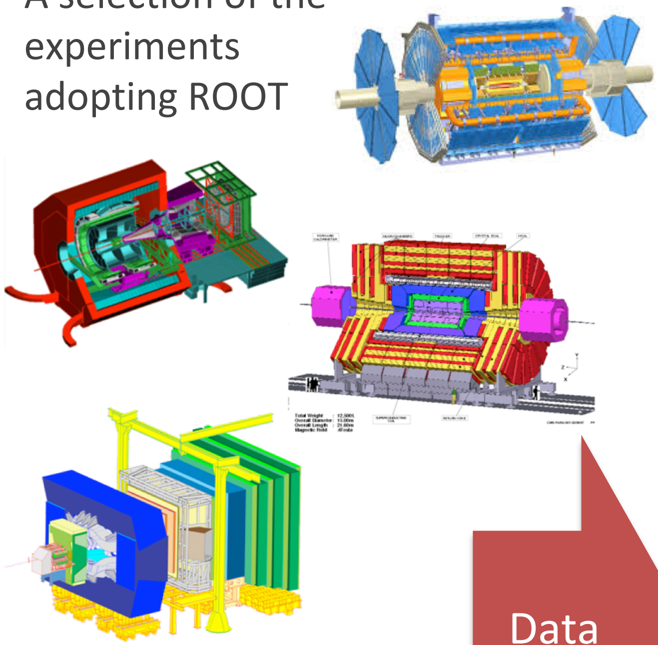
my_python_module.py

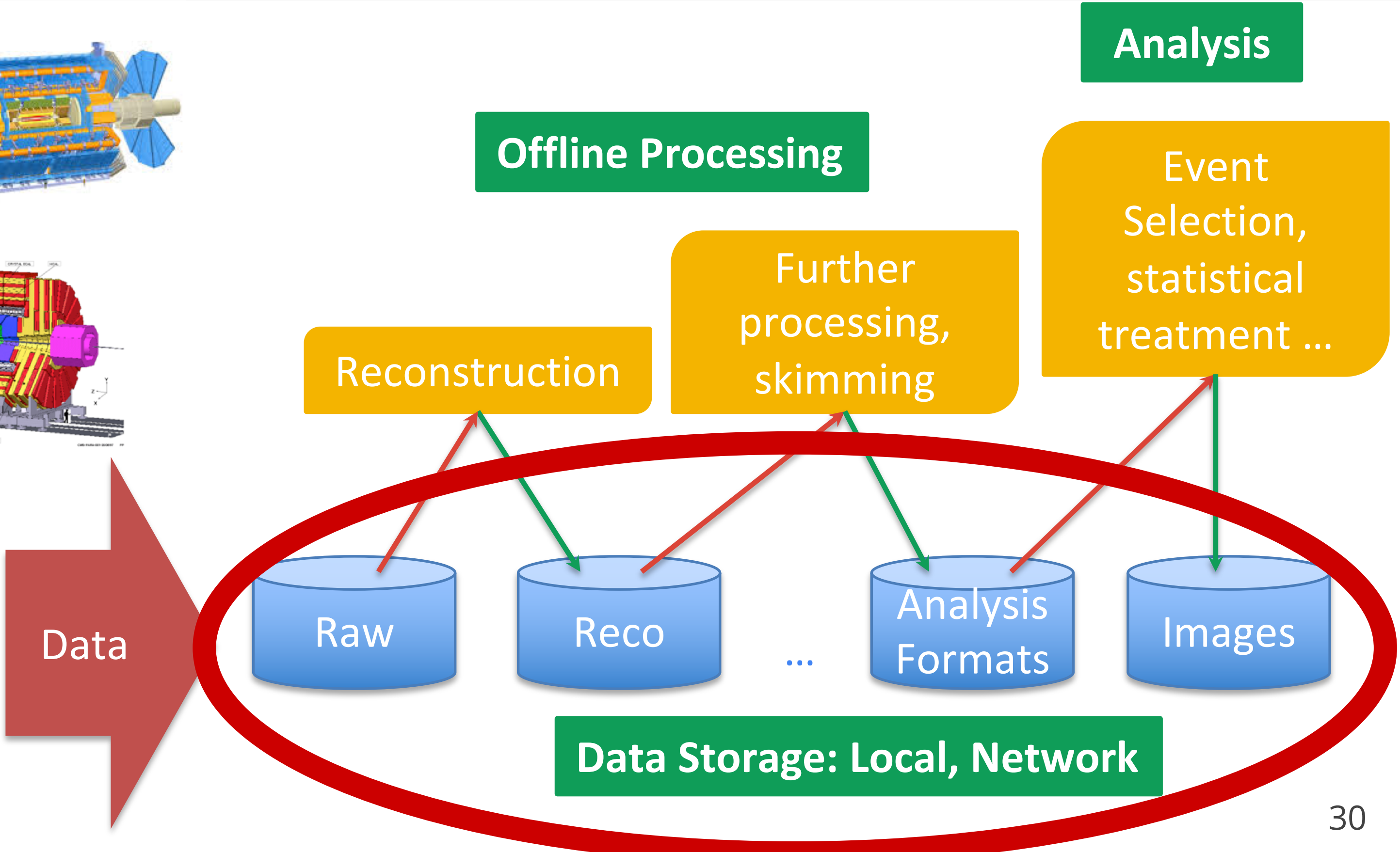
```python
# Load a C++ library
ROOT.gInterpreter.ProcessLine('#include "my_cpp_library.h"')
ROOT.gSystem.Load('./my_cpp_library.so')

# We find all the C++ entities in Python!
a = ROOT.A()   # this prints Hello PyROOT!
x = ROOT.f(3)  # x = 9
```

Reading and Writing Data

I/O at LHC: an Example

A selection of the experiments adopting ROOT



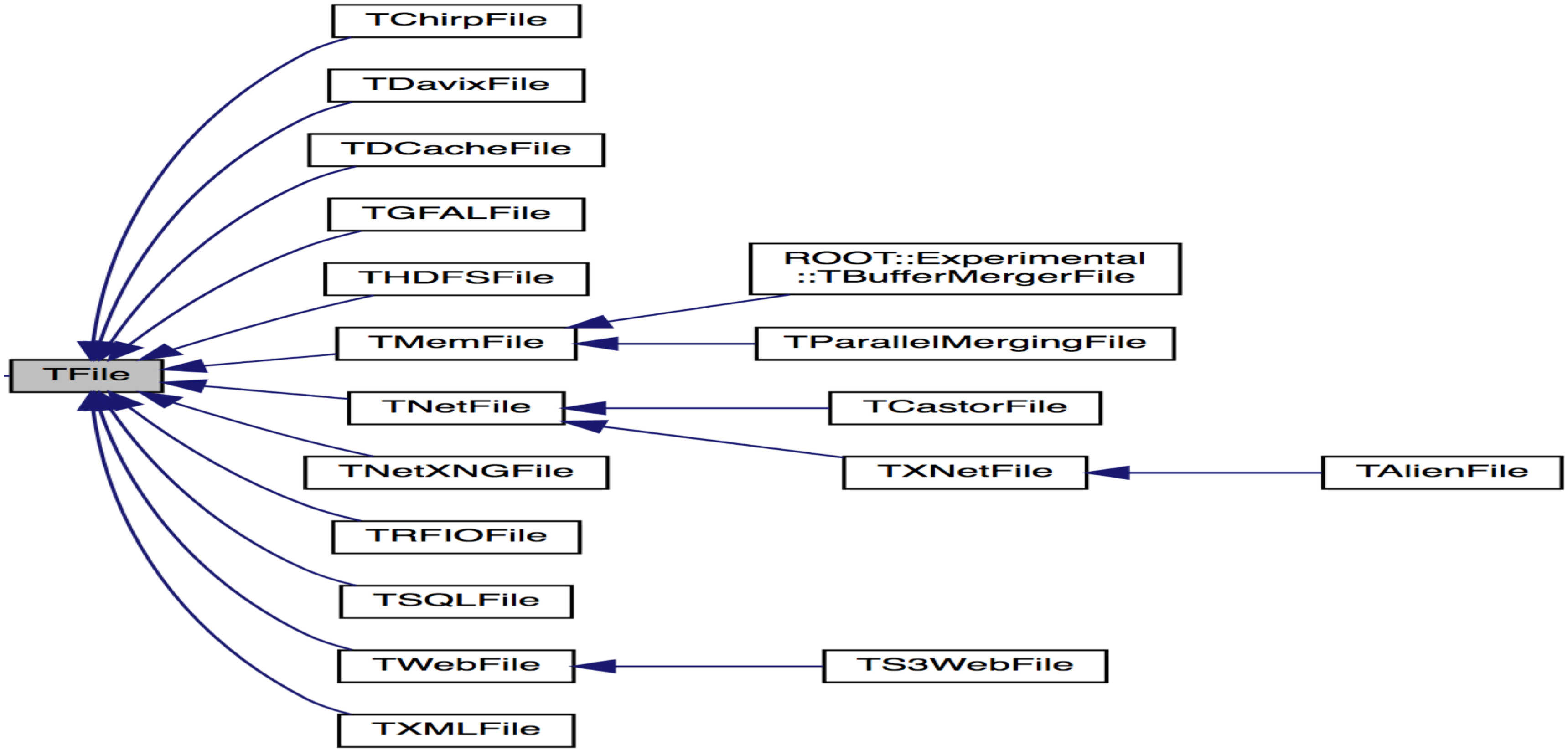


The ROOT File

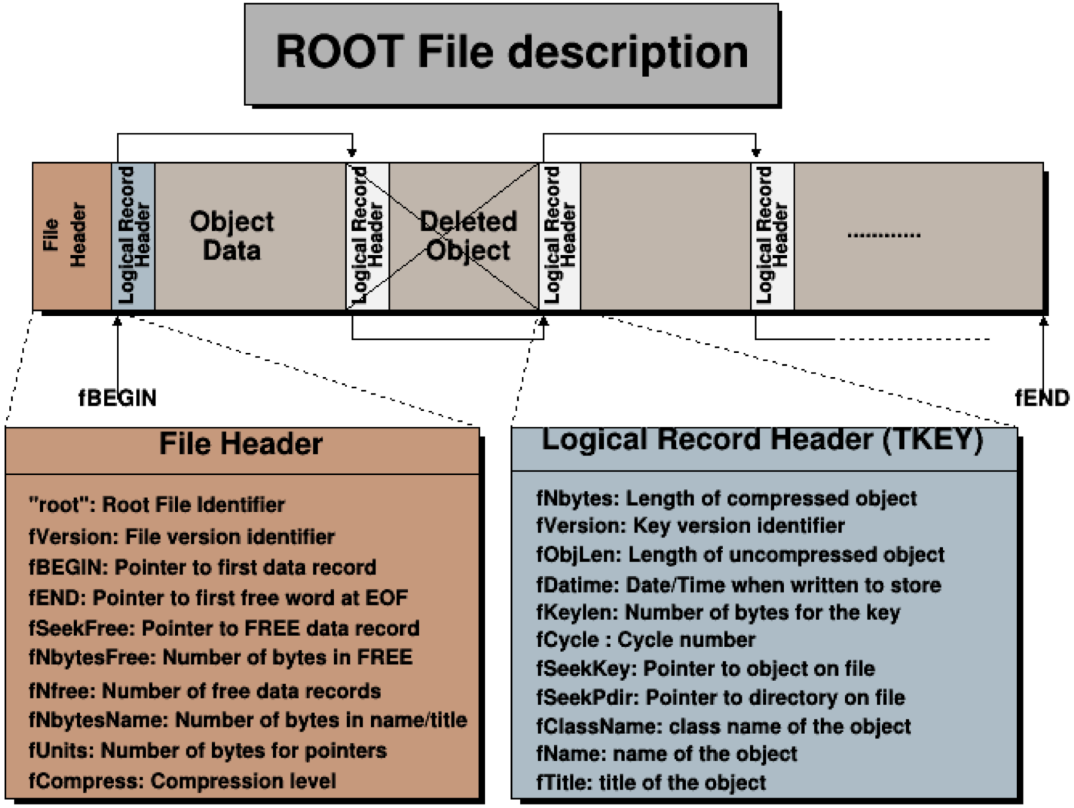
- In ROOT, objects are written in files*
- ROOT provides its file class: the **TFile**
- TFiles are *binary* and have: a *header*, *records* and can be compressed (transparently for the user)
- TFiles have a logical “file system like” structure
  - e.g. directory hierarchy
- **TFiles are self-descriptive**: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

* this is an understatement - we’ll not go into the details.

Flavour of TFiles



ROOT File Description



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

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:

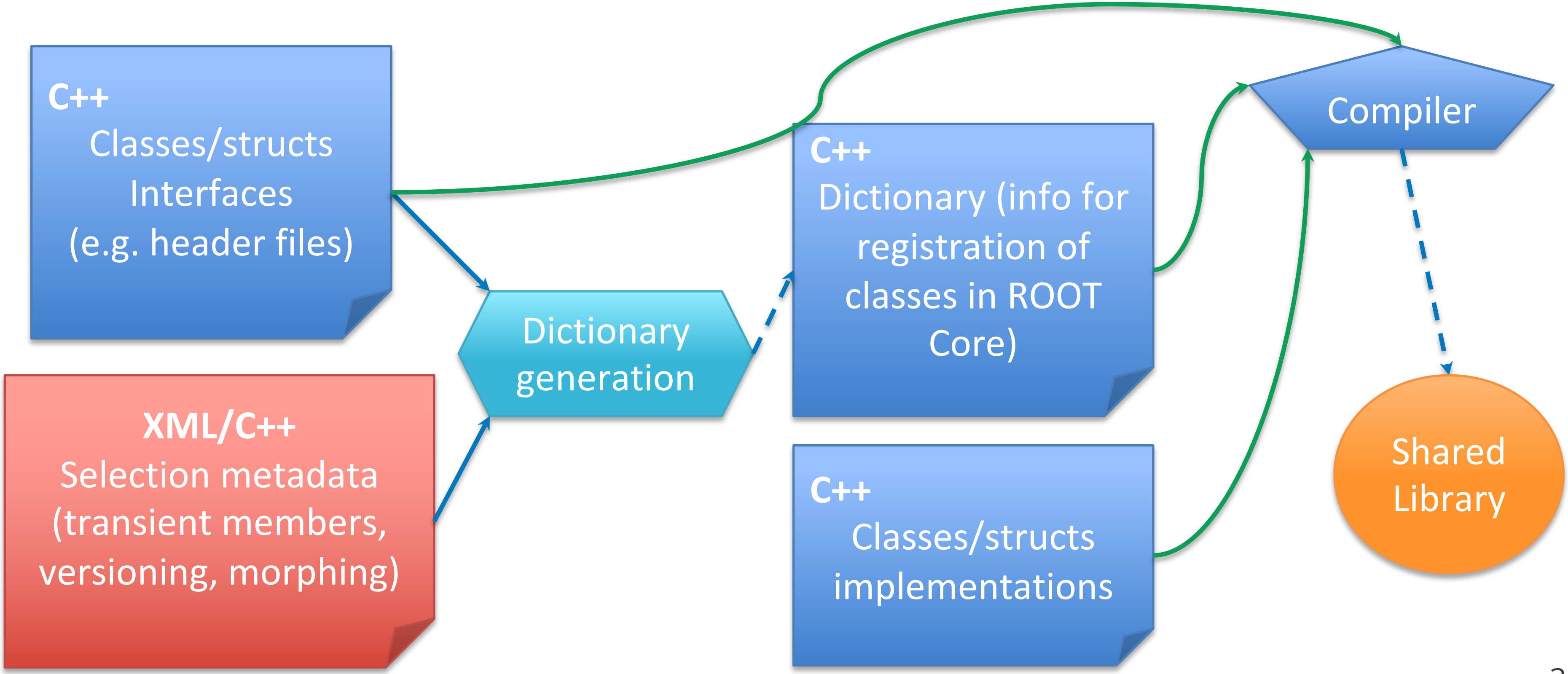
1. Serialisation: from an object in memory to a blob of bytes
2. Compression: use an algorithm to reduce size of the blob (e.g. zip, lzma, lz4)
3. Writing to the physical resource (disk) via OS primitives

Serialisation: not a trivial task

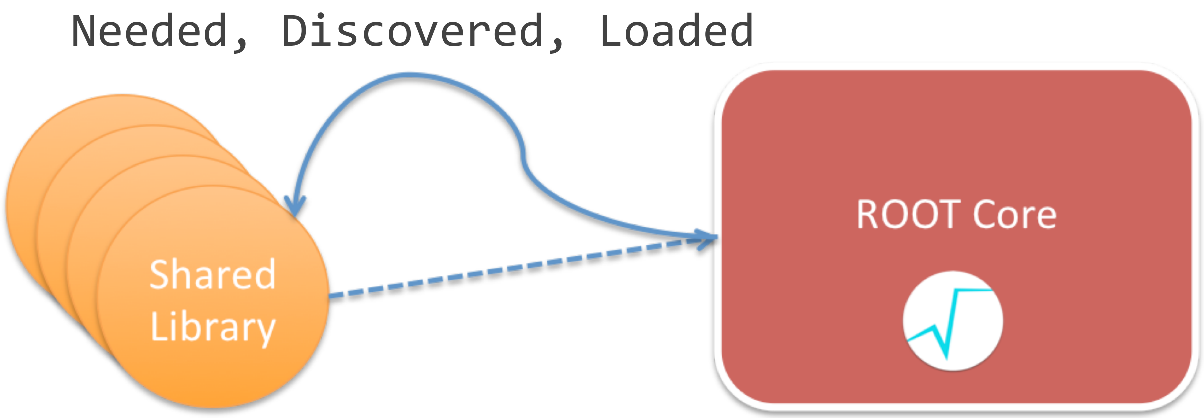
For example:

- 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 stl constructs
- 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 customisations by the user
  - E.g. skip “transient data members”
  - I/O customization rule (transformation of data)

Persistency



Injection of Reflection Information

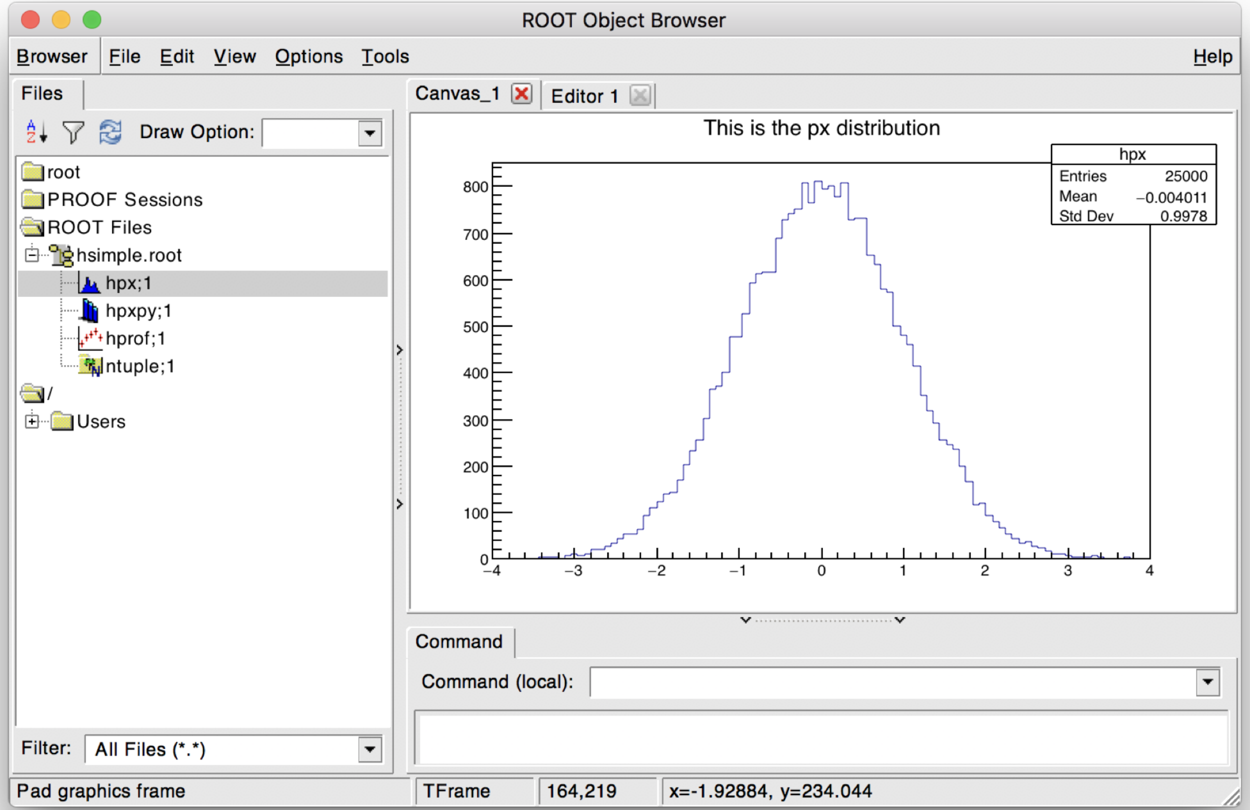


Now ROOT "knows" how to serialise the instances implemented in the library (series of data members, type, transiency) and to write them on disk in row or column format.

TFile in Action

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

TBrowser



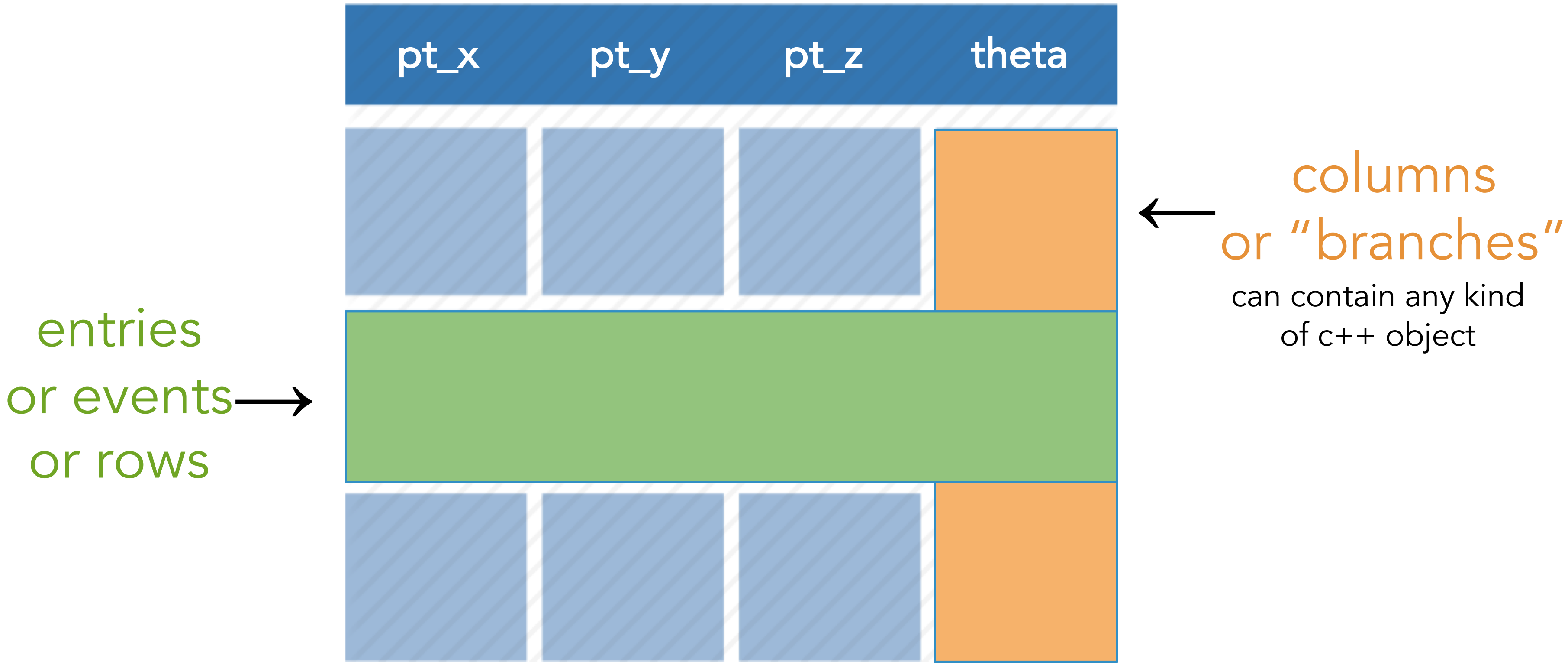
ROOT Object Browser
Browser File Edit View Options Tools Help
Files
Draw Option:
root
PROOF Sessions
ROOT Files
hsimple.root
hpx;1
hpxpy;1
hprof;1
ntuple;1
/
Users
Filter: All Files (*.*)
Canvas_1
Editor 1
This is the px distribution
hpx
Entries 25000
Mean −0.004011
Std Dev 0.9978
Command
Command (local):
Pad graphics frame
TFrame
164,219
x=-1.92884, y=234.044

The ROOT Columnar Format

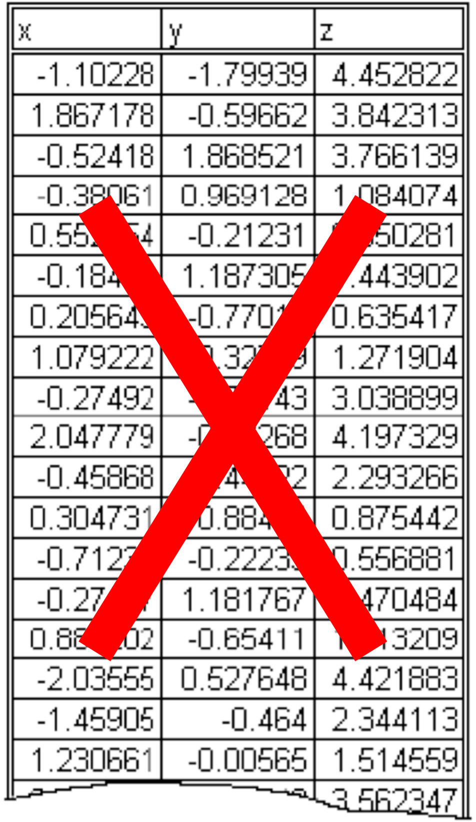
Columns and Rows

- 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**

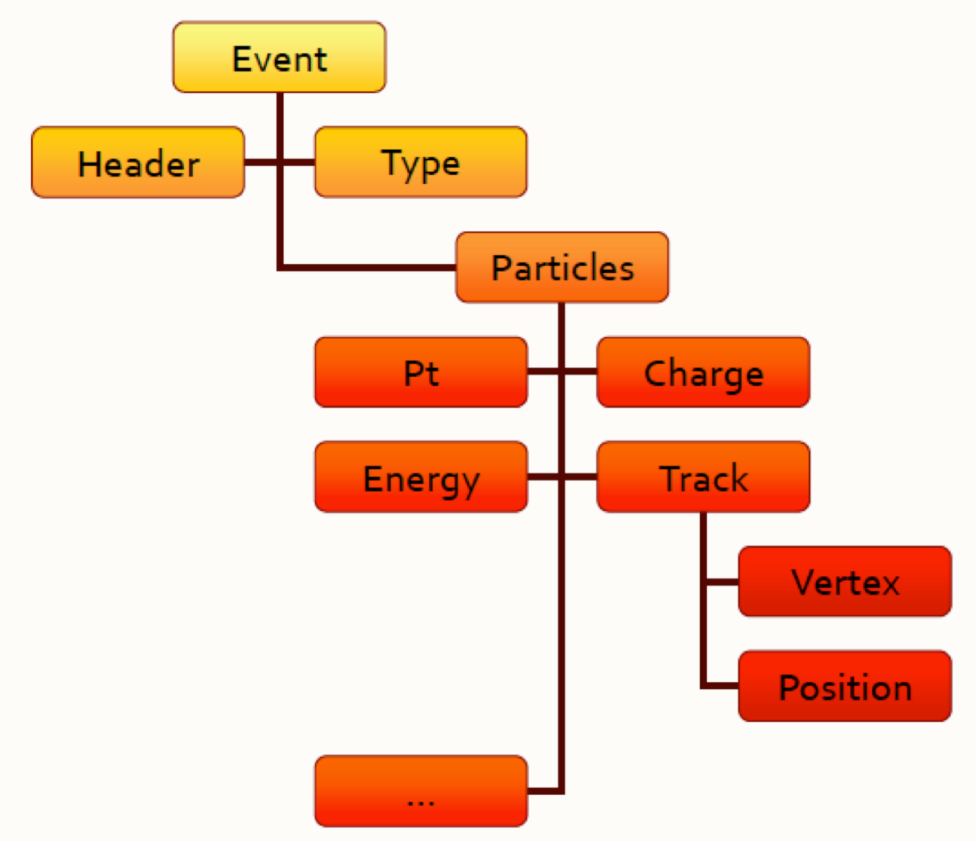
Columnar Representation



Relations Among Columns



x	y	z
-1.10228	-1.79939	4.452822
1.867178	-0.59662	3.842313
-0.52418	1.868521	3.766139
-0.38061	0.969128	1.084074
[illegible]	-0.21231	[illegible]
[illegible]	1.187305	[illegible]
[illegible]	[illegible]	0.635417
1.079222	[illegible]	1.271904
-0.27492	[illegible]	3.038899
2.047779	[illegible]	4.197329
-0.45868	[illegible]	2.293266
0.304731	[illegible]	0.875442
[illegible]	[illegible]	0.556881
[illegible]	1.181767	[illegible]
[illegible]	-0.65411	[illegible]
-2.03555	0.527648	4.421883
-1.45905	-0.464	2.344113
1.230661	-0.00565	1.514559
[illegible]	[illegible]	3.562347

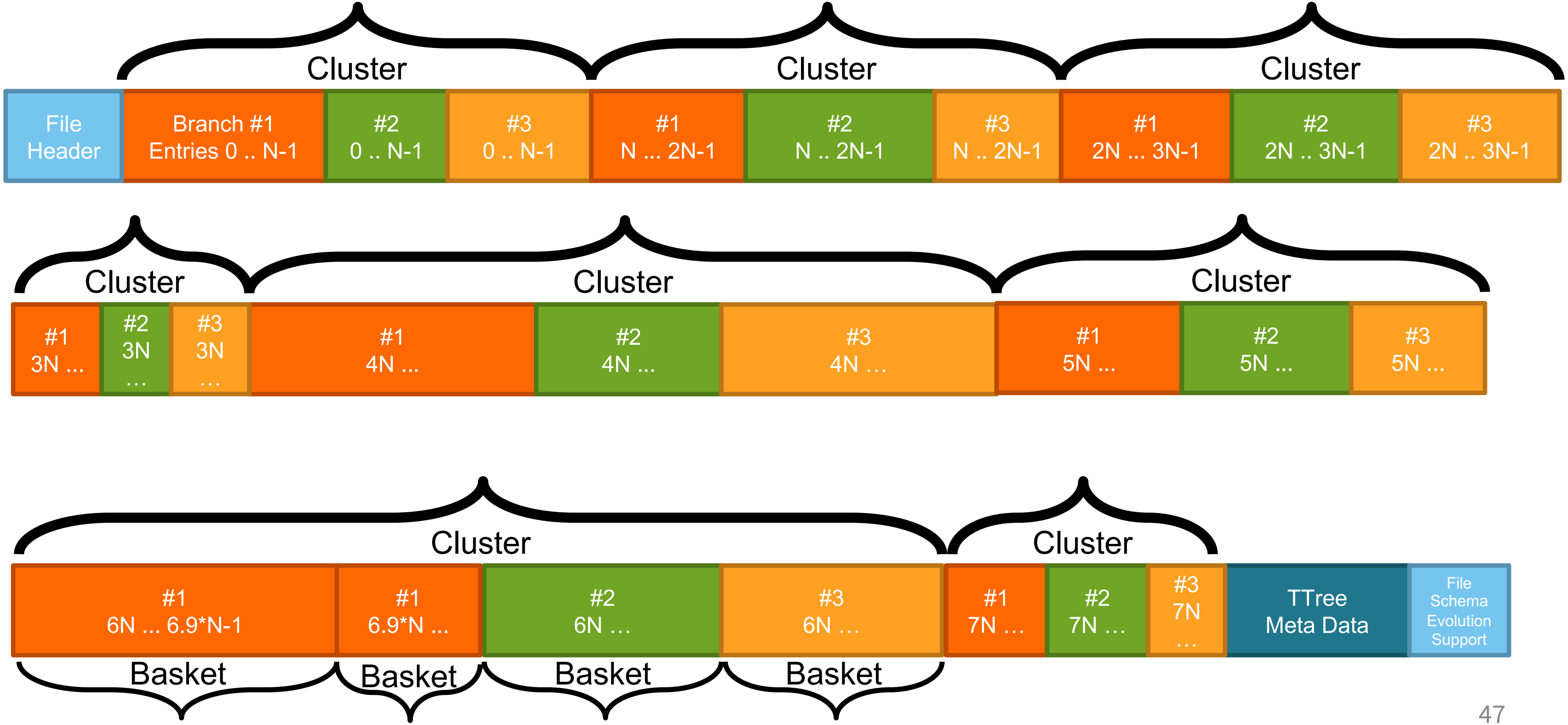


TTree

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*)

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

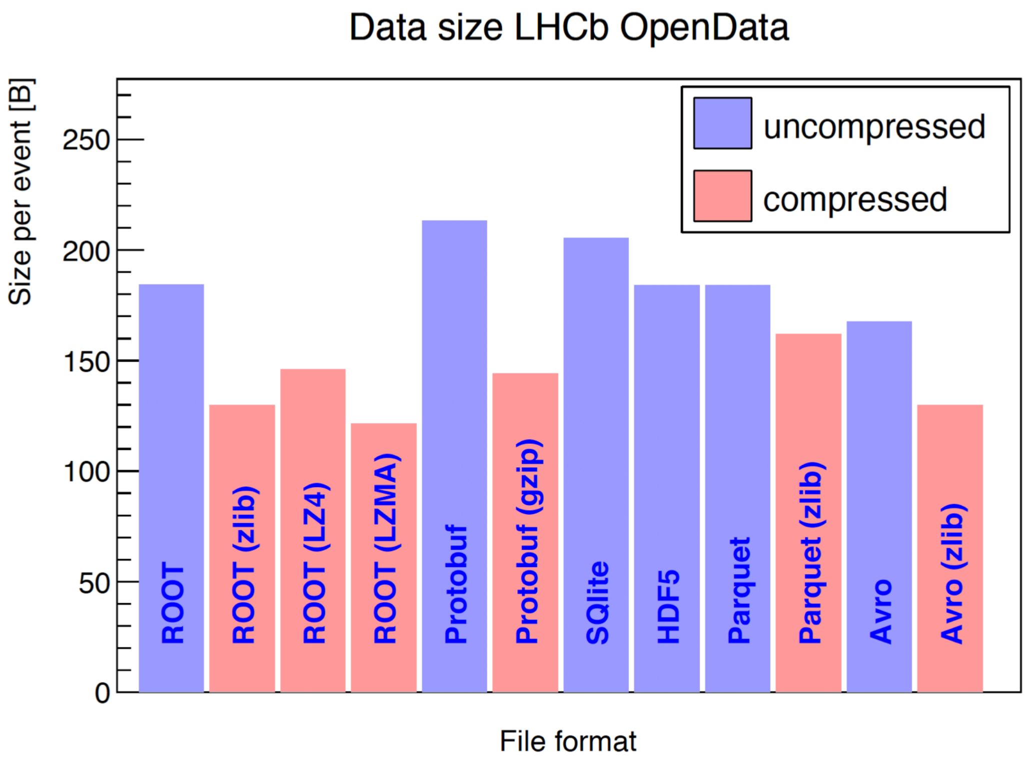
Comparison With Other I/O Systems

	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

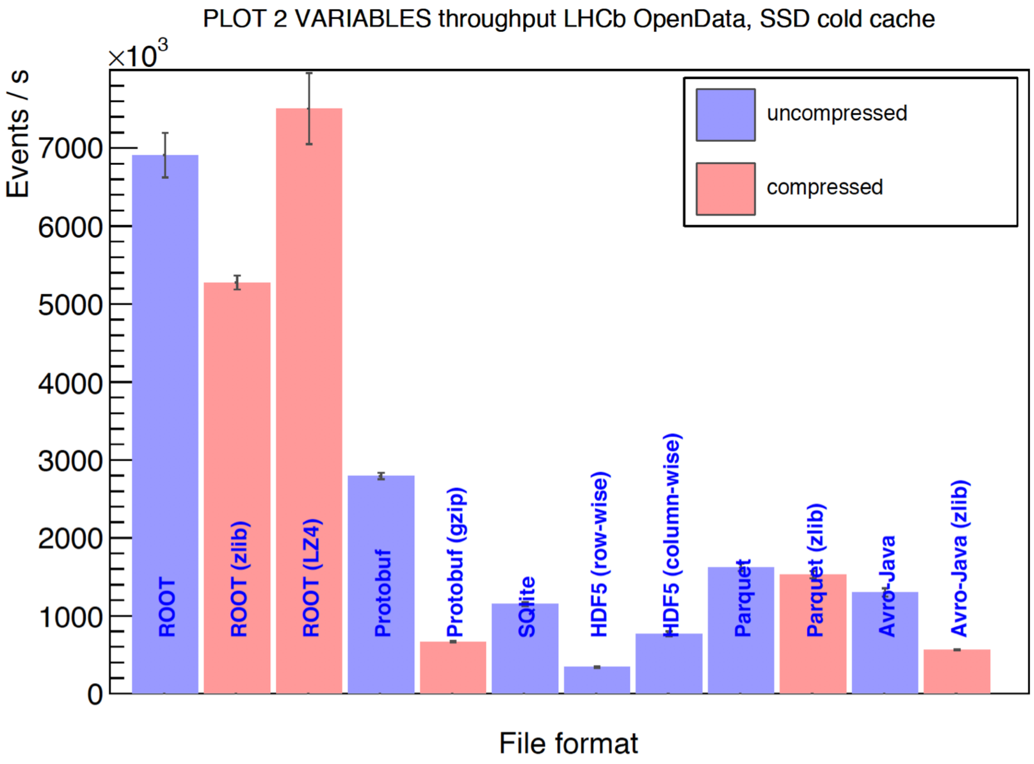
J. Blomer, A quantitative review of data formats for HEP analyses ACAT 2017

Comparison With Other I/O Systems



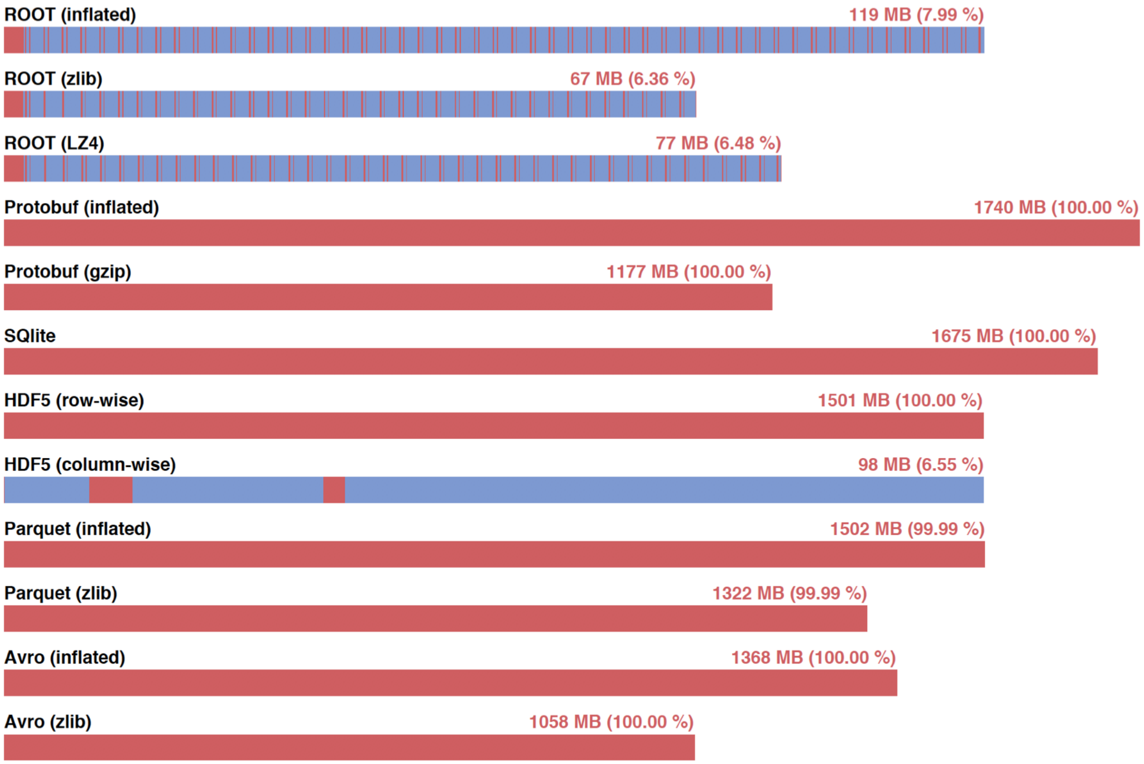
J. Blomer, A quantitative review of data formats for HEP analyses ACAT 2017

Comparison With Other I/O Systems



J. Blomer, A quantitative review of data formats for HEP analyses ACAT 2017

I/O Patterns

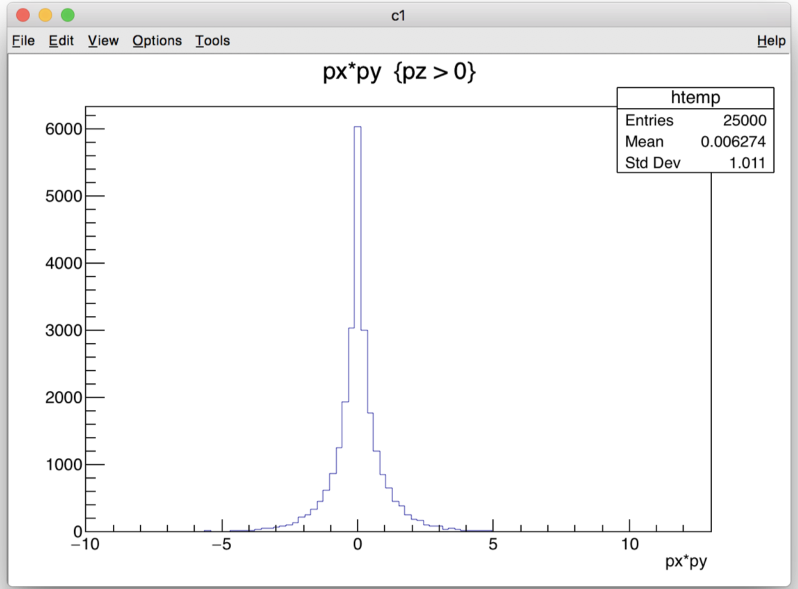


The less you read (red sections), the faster

J. Blomer, [A quantitative review of data formats for HEP analyses](#) ACAT 2017

One Line Analysis

Works for all types of columns, not only numbers!




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

One Line Analysis

- It is possible to produce simple plots described as strings
- **TTree::Draw()** method
- Good for quick looks, does not scale
  - E.g. one loop on all events per plot
- See backup slide for newer functional analysis framework (**RDataFrame**)

Filling a Tree with Numbers

```
TFile f("SimpleTree.root","RECREATE"); // Create file first. The TTree will be associated to it
TTree data("tree","Example TTree");    // No need to specify column names

double x, y, z, t;
data.Branch(”px",&x,"x/D");      // Associate variable pointer to column and specify its type, double
data.Branch(”py",&y,"y/D");
data.Branch(”pz",&z,"z/D");
data.Branch("t",&t,"t/D");

for (int i = 0; i<128; ++i) {
   x = gRandom->Uniform(-10,10);
   y = gRandom->Gaus(0,5);
   z = gRandom->Exp(10);
   t = gRandom->Landau(0,2);
   data.Fill();                 // Make sure the values of the variables are recorded
}
data.Write();                   // Dump on the file
f.Close();
```

Filling a Tree with Objects

```
TRandom3 R;
using trivial4Vectors =
std::vector<std::vector<double>>;

TFile f("vectorCollection.root",
        "RECREATE");
TTree t("t","Tree with pseudo particles");

trivial4Vectors  parts;
auto partsPtr = &parts;

t1.Branch("tracks", &partsPtr);
// pi+/pi- mass
constexpr double M = 0.13957;
```

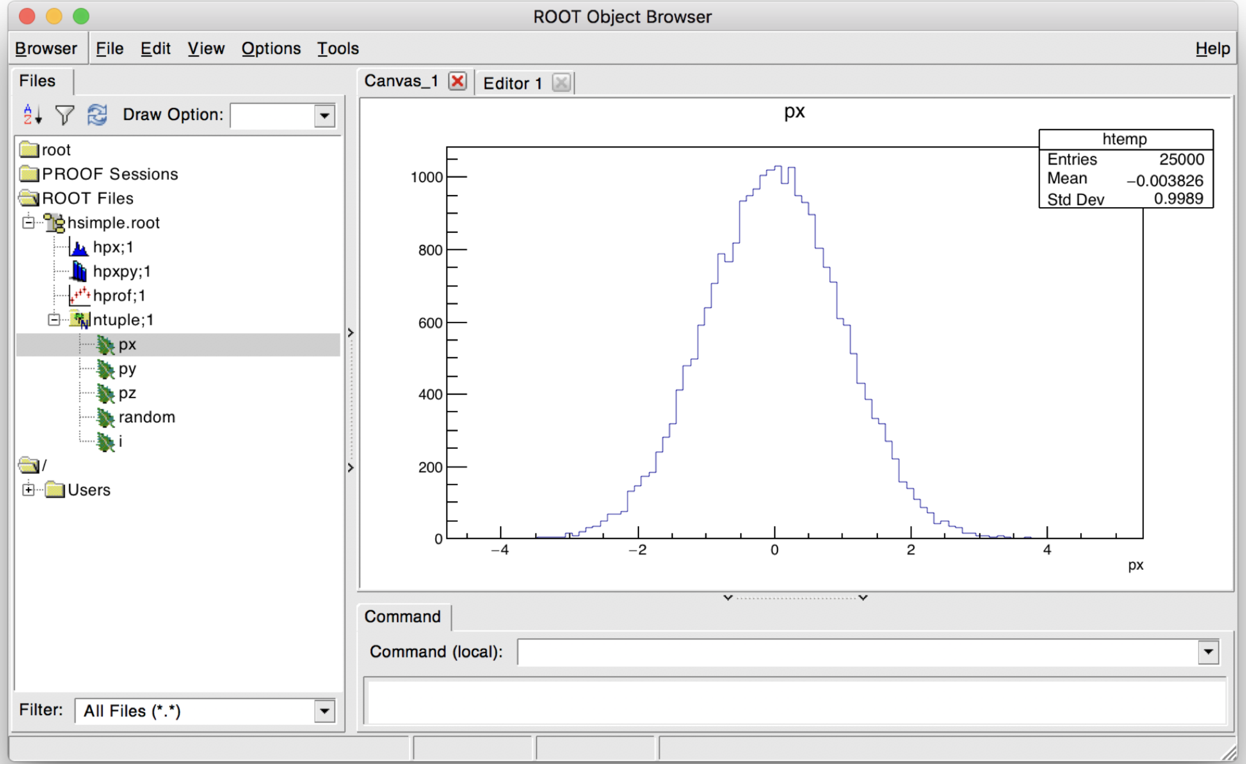
```
for (int i = 0; i < 128; ++i) {
   auto nPart = R.Poisson(20);
   particles.clear(); parts.reserve(nPart);
   for (int j = 0; j < nPart; ++j) {
      auto pt = R.Exp(10);
      auto eta = R.Uniform(-3,3);
      auto phi = R.Uniform(0, 2*TMath::Pi() );
      parts.emplace_back({pt, eta, phi, M});
    }
    t.Fill();
  }
  t.Write();
}
```

Reading Objects from a TTree

```
{

using trivial4Vector =
std::vector<double>;
using trivial4Vectors =
std::vector<trivial4Vector>;

TFile f("parts.root");
TTreeReader myReader("t", &f);
TTreeReaderValue<trivial4Vectors>
partsRV(myReader, "parts");

TH1F h("pt","Particles Transverse
Momentum;P_{T} [GeV];#", 64, 0, 10);
```

```
while (myReader.Next()) {
   for (auto &p : *partsRV ) {
         auto pt = p[0];
         h.Fill(pt);
   }
}
h.Draw();

}
```

Reading a TNtuple with TBrowser

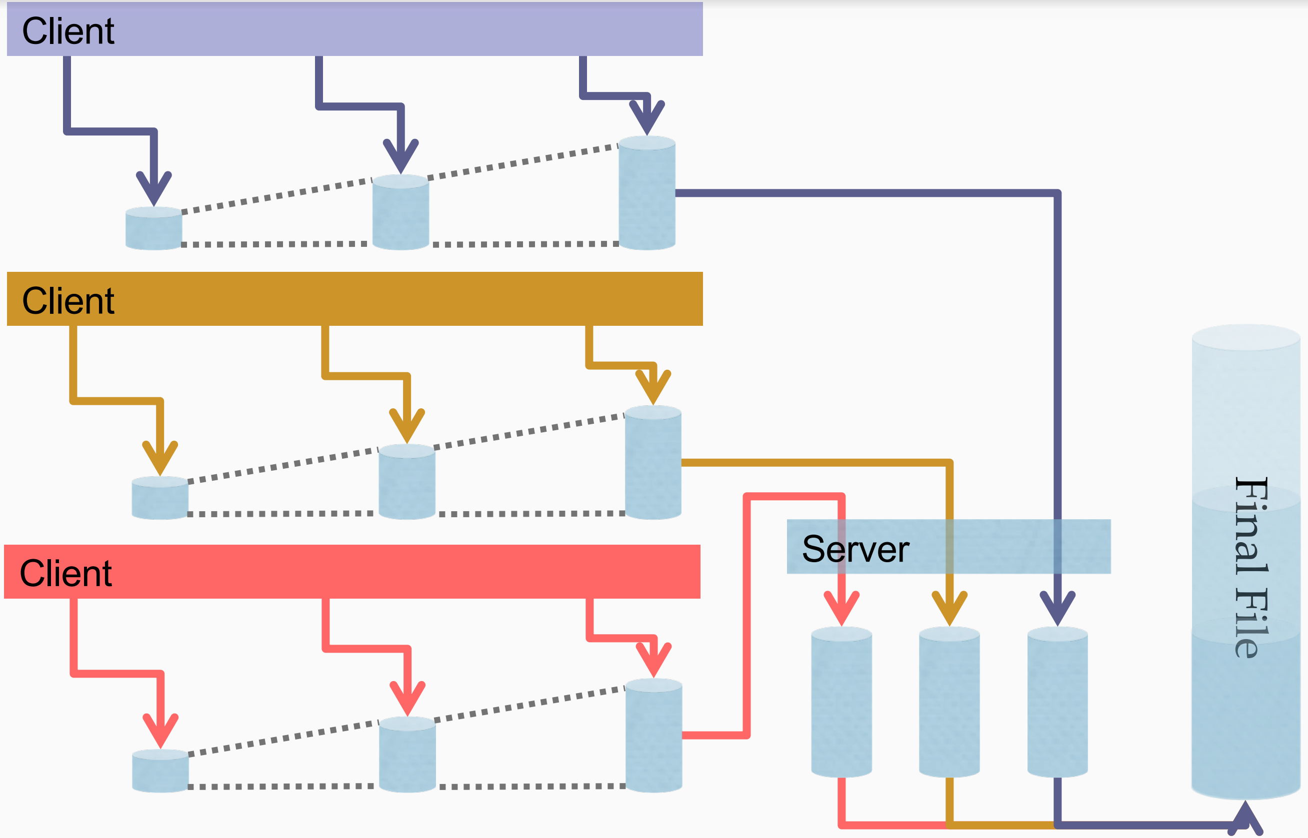


Many writers?

ROOT Files vs Multiple Writers

- ROOT Files inherently deals with variable size records
  - Data frequently contains variable size collection
  - Compression done inline
  - For each branch/column data store in ‘bunch’ of several entries/row, named a ‘Basket’; this is the unit of compression.
- Pre-reservation of file space not an option

Old Fashion Arrangement



Fast Merging

- ROOT Files can be ‘fast’ merged by ‘only’
  - Copying/appending the compressed data (baskets)
  - Updating the meta data (TTree object)
  - In first approximation we reach disk bandwith
    - Actually … half … since we read then write.

- Leverage this capability and use in-memory file to add support for multiple writers to the same file
  - Multi-thread in production
  - MPI prototype

With Parallel Merging

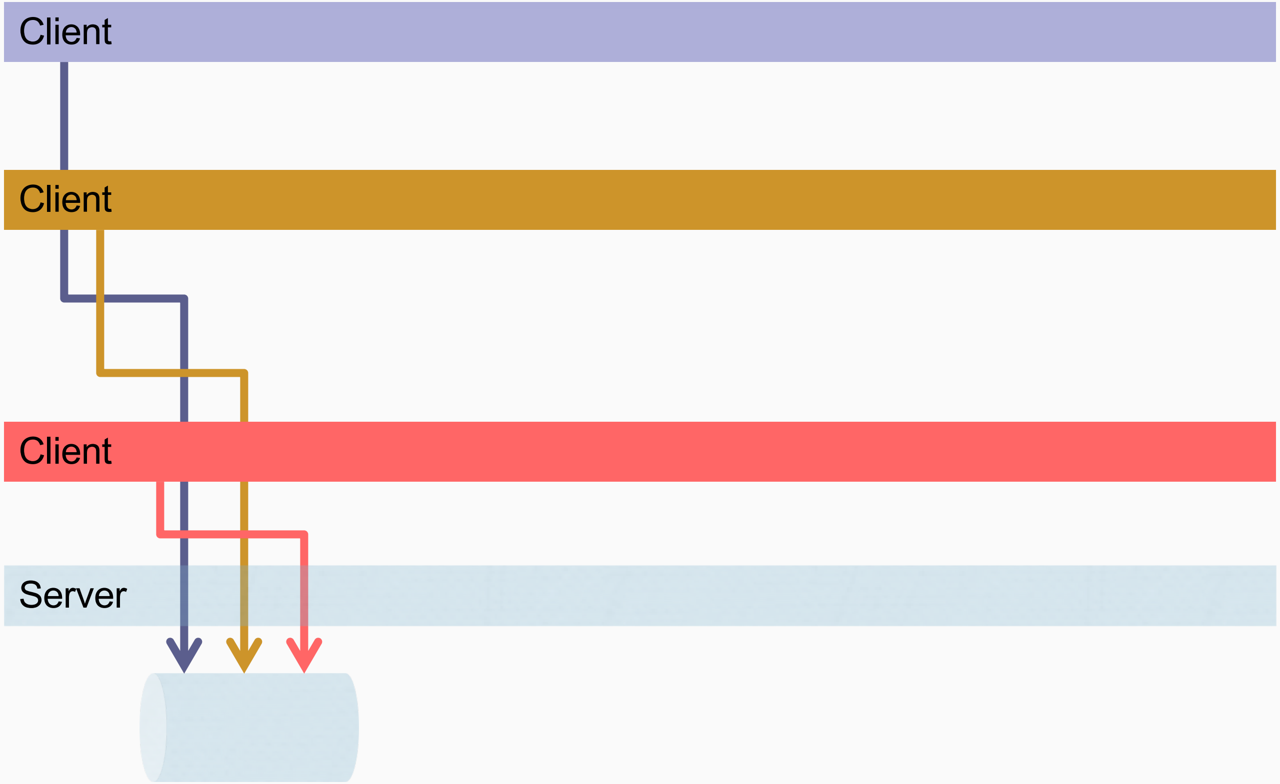
Client

Client

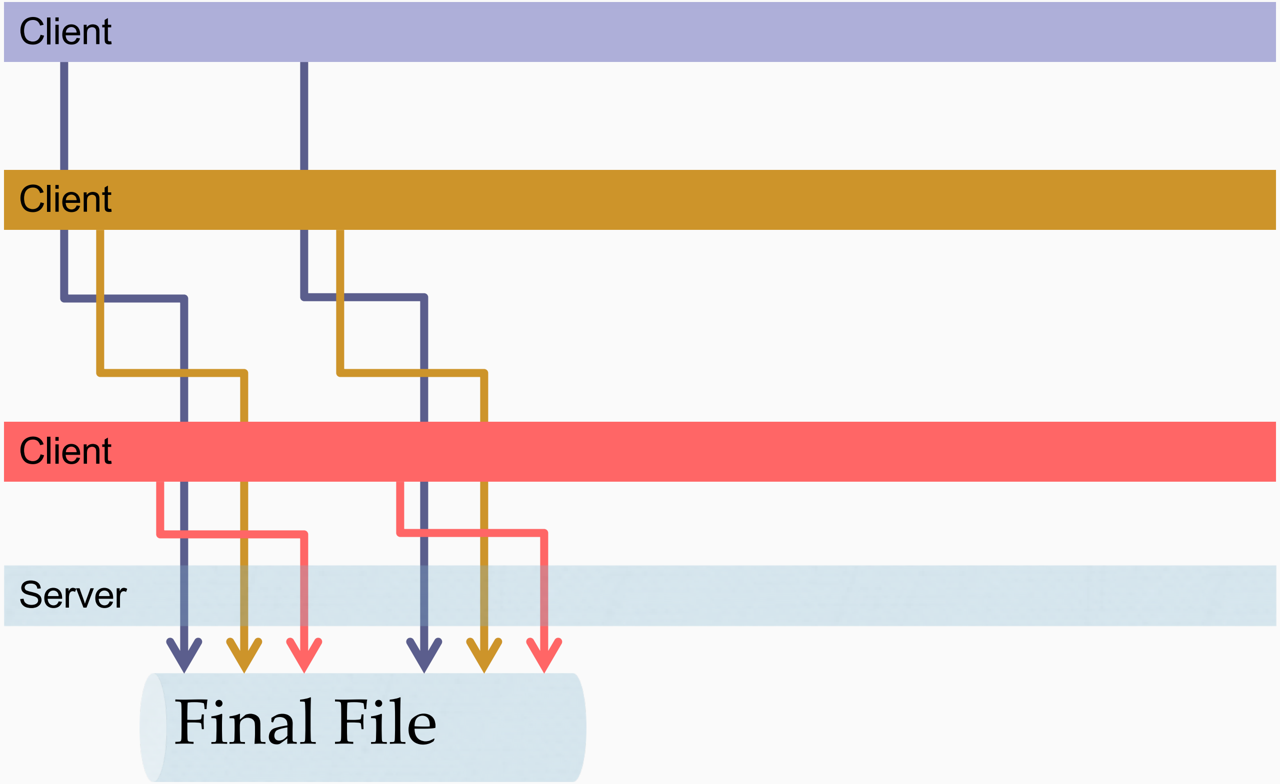
Client

Server

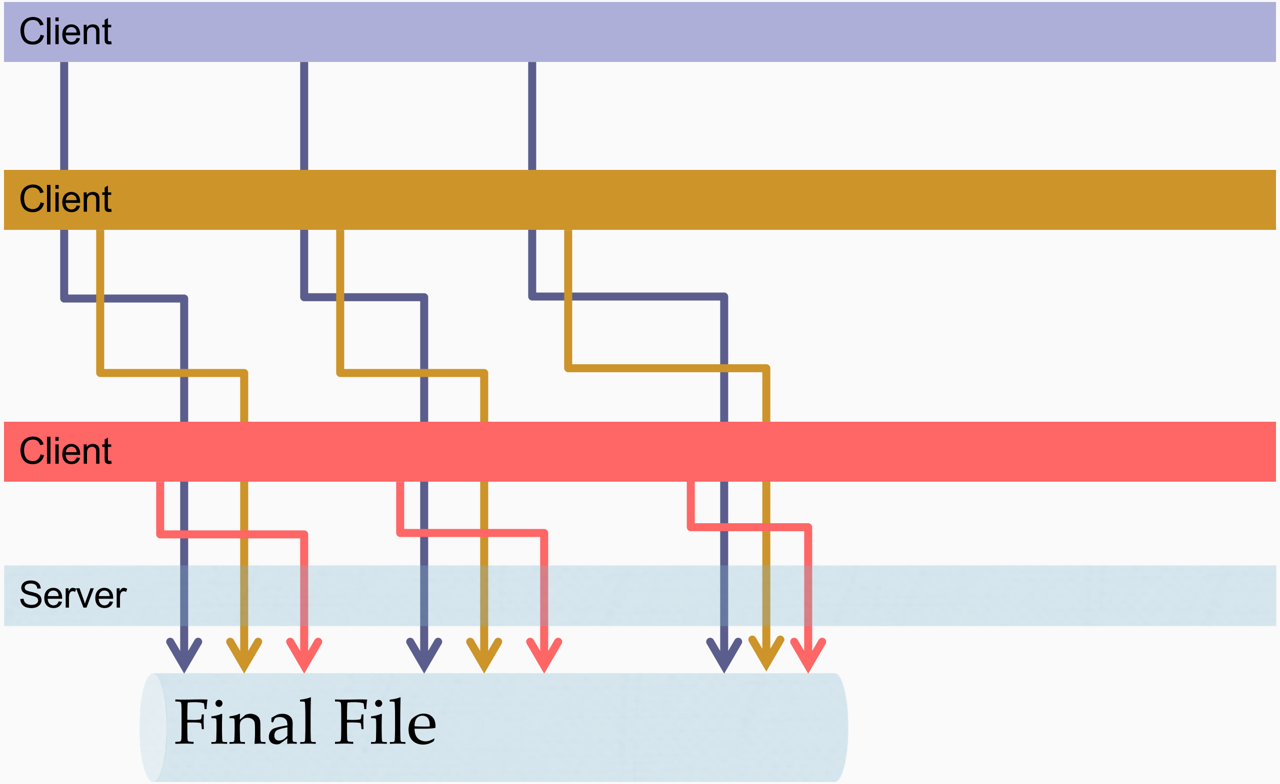
With Parallel Merging



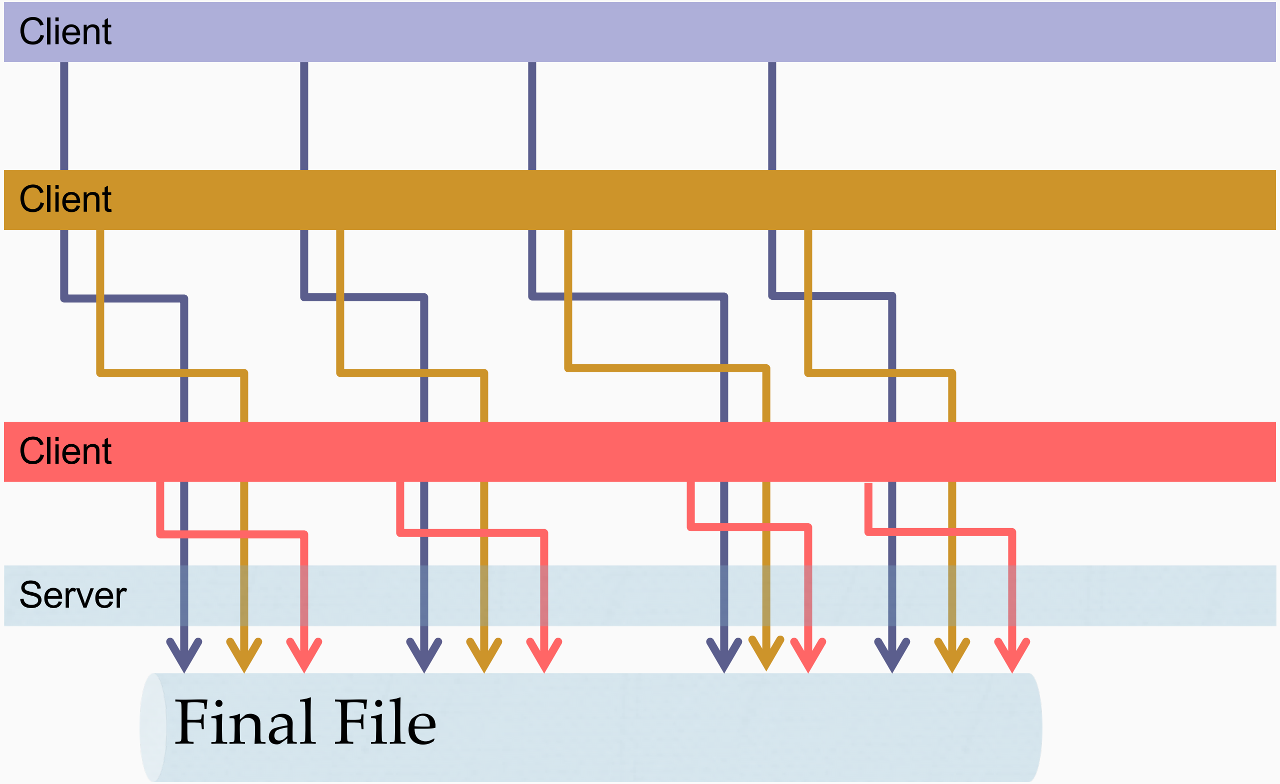
With Parallel Merging



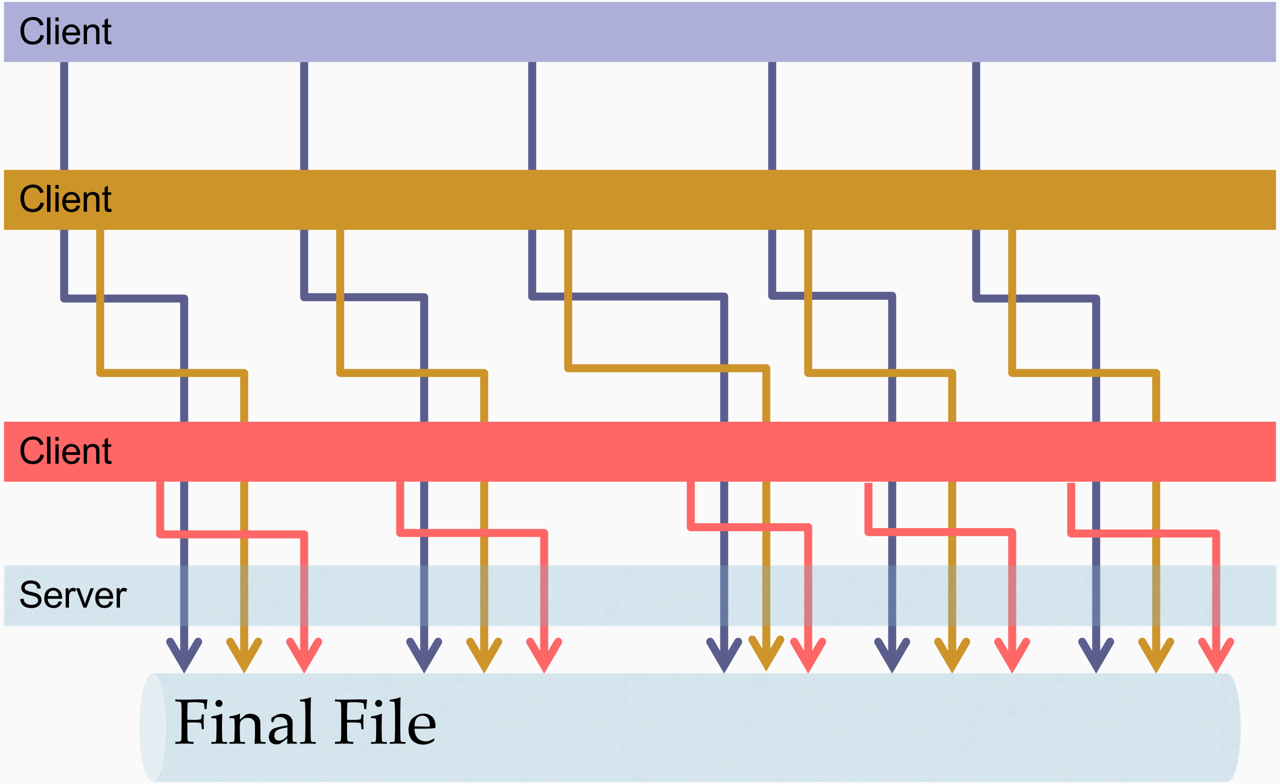
With Parallel Merging



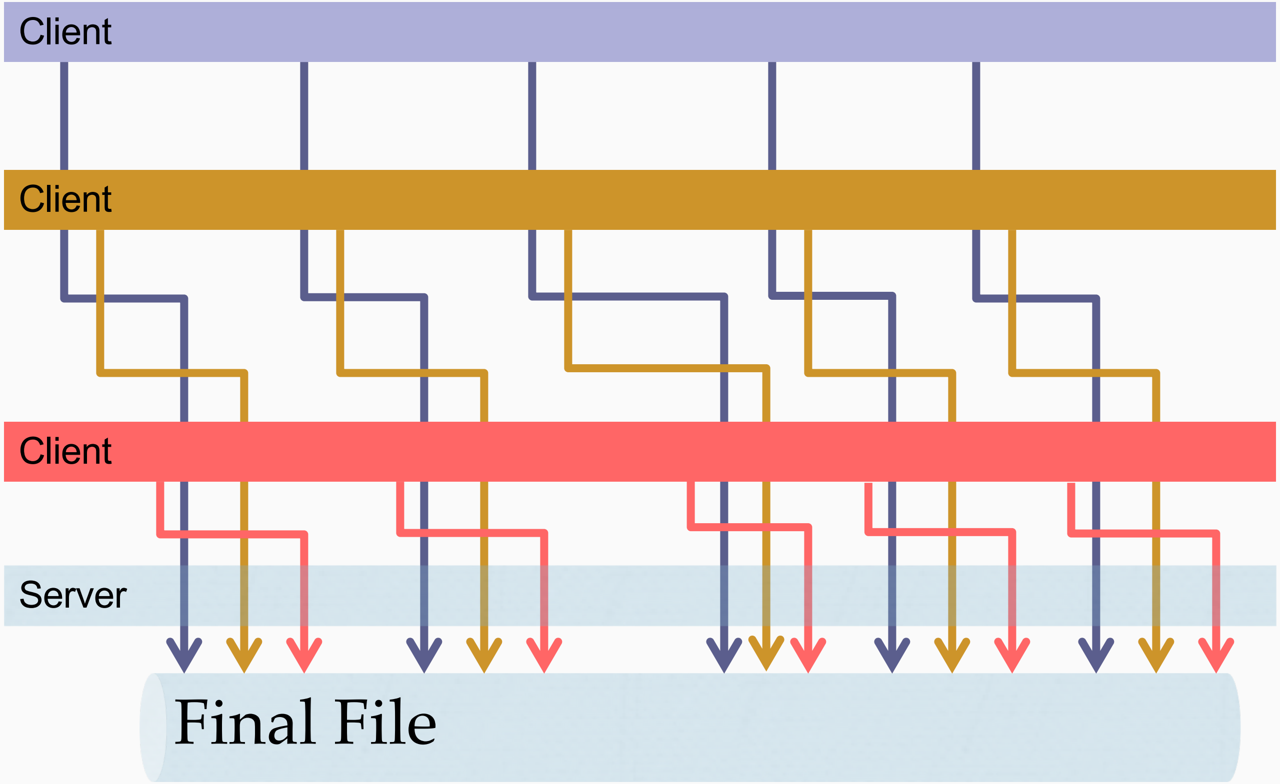
With Parallel Merging



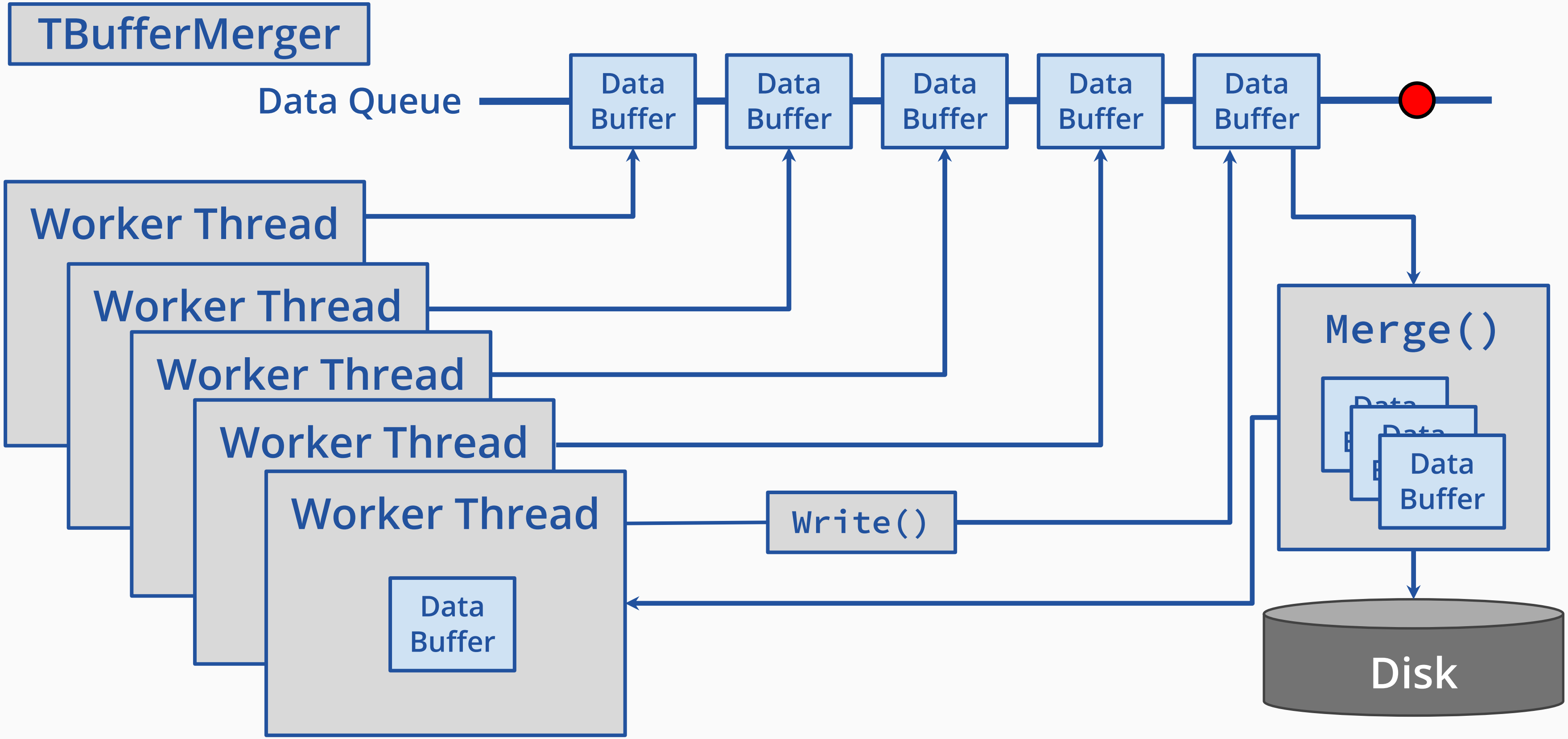
With Parallel Merging



With Parallel Merging



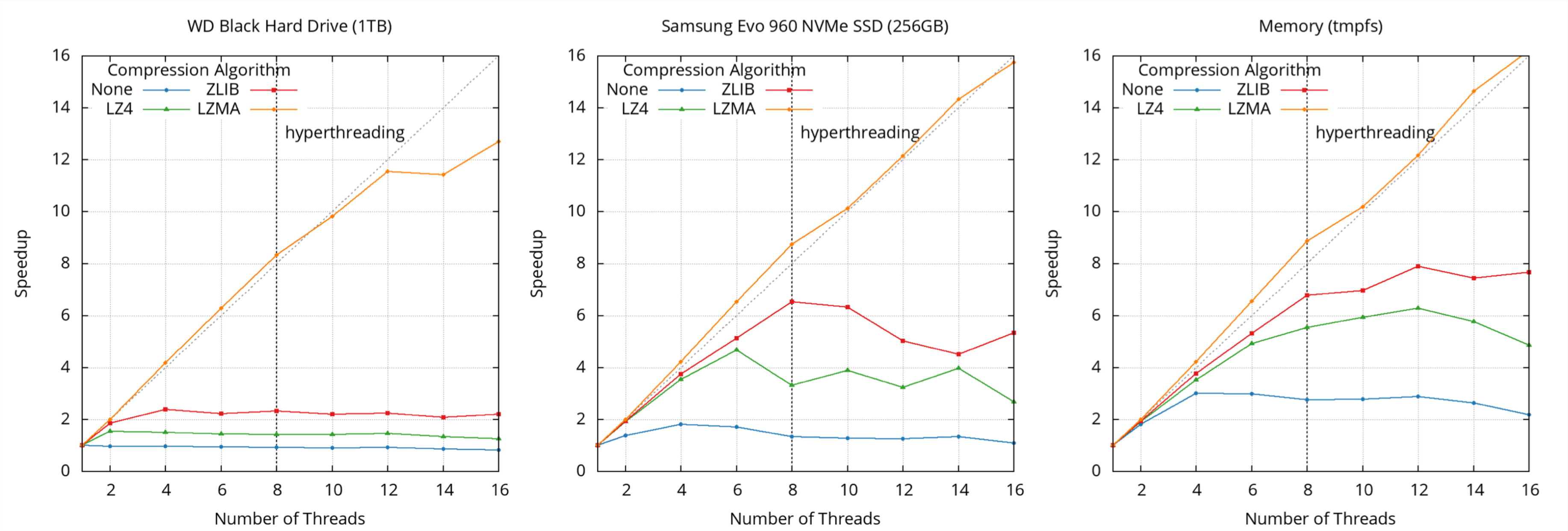
TBufferMerger



TBufferMerger Single Branch Benchmark

- Create ~1GB of **simple** data and write out to different media using different compression algorithms
- Measured time to flush disk cache is negligible compared to runtime
- Synthetic benchmark that exacerbates the role of I/O by doing light amount of work (generating a random number)
- Test environment
  - Intel® CoreTM i7-7820X Processor (8 cores, 11M Cache, up to 4.30 GHz)
  - Write out data to HDD, NVMe SSD, DRAM
  - Compare compression algorithms: LZ4, ZLIB, LZMA, no compression
  - GCC 8.1.0, C++17, -O3 -march=native (skylake-avx512), release build

Single Branch Benchmark: Speedup



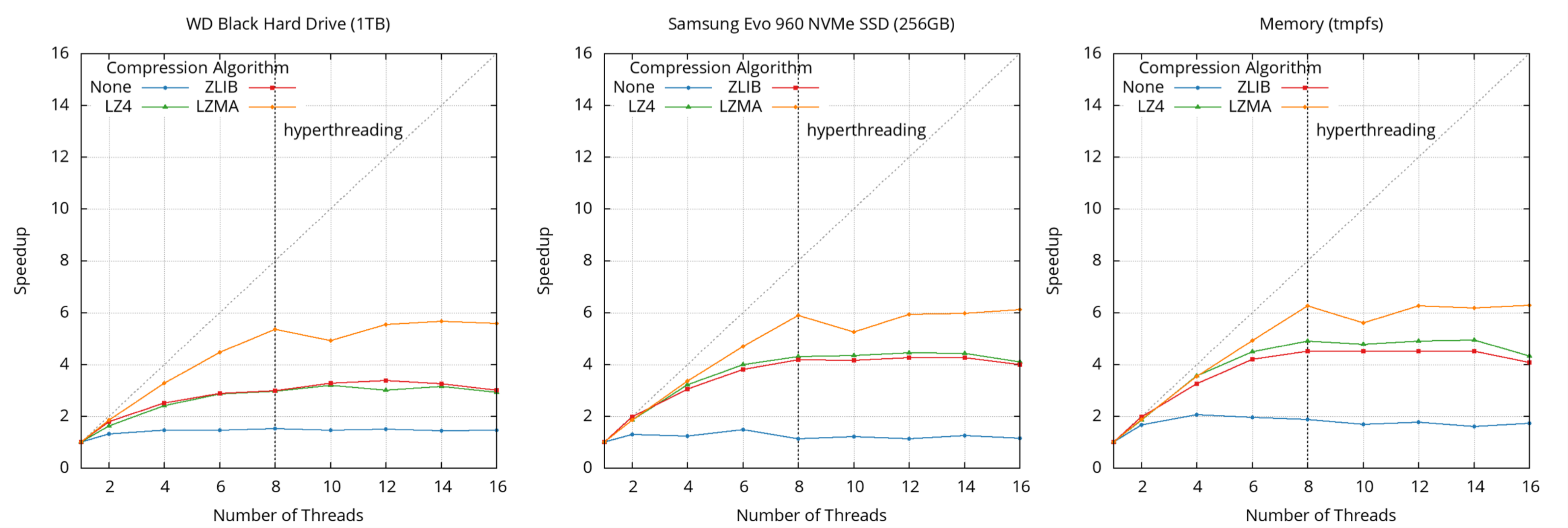
All figures using ROOT master branch

TBufferMerger Multi Branch Benchmark

- Create 1GB of **complex** data and write out to different media using different compression algorithms
- Synthetic benchmark to investigate what changes with added data complexity vs previous benchmark, IMT disabled but speedups are similar
- 1 branch = std::vector<Event> (3x Vector3D, 3x double, 3x int)
- Data compresses better, so uncompressed is writing more output
- Test environment
  - Intel® CoreTM i7-7820X Processor (8 cores, 11M Cache, up to 4.30 GHz)
  - Write out data to HDD, NVMe SSD, DRAM
  - Compare compression algorithms: LZ4, ZLIB, LZMA, no compression
  - GCC 8.1.0, C++17, -O3 -march=native (skylake-avx512), release build 9

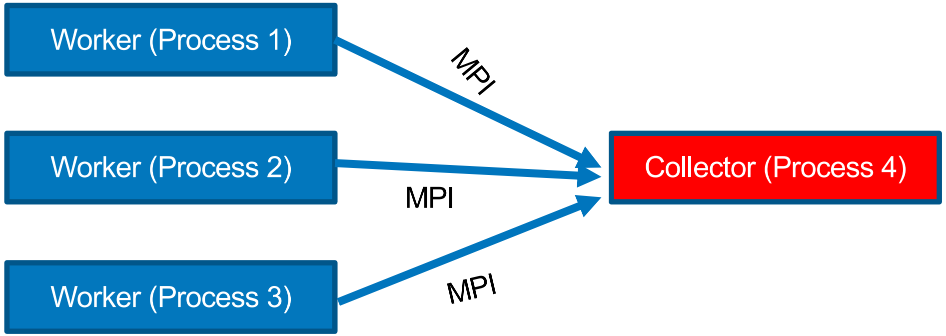
Multi Branch Benchmark: Speedup

Test creates 10 branches, each with a vector of 10 Events



All figures using ROOT master branch

MPI Prototype: Basic Structure



Communication is done via MPI functionalities

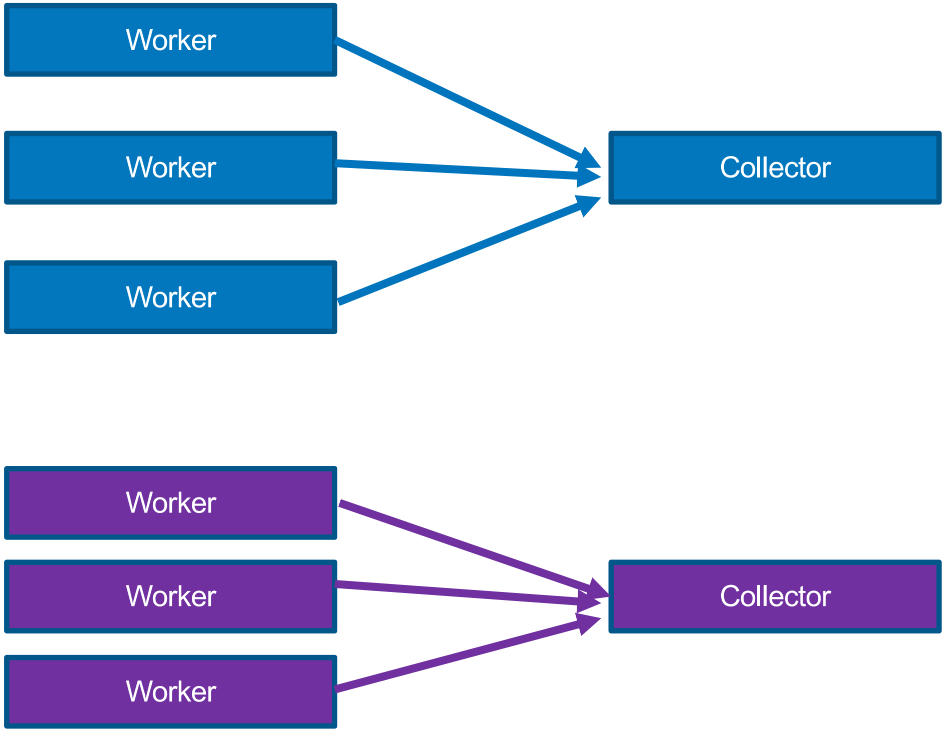
Reading/Writing into buffer is done using TMemFile functionalities

<u>Each of the workers and collectors is one unique MPI Process or Rank.</u>

Workers:
- Process Events (Populate TTrees or TH1D’s)
- Send Processed Events to Collector Using MPI functionalities

Collectors:
- Receive Processed Events from Workers
- Merge them
- Write into disk

MPI Prototype: Basic Structure



Processes can be divided into many worker/processor sub groups and do multiple parallel merging.

Wrap up

ROOT I/O

- Write almost any C++ objects/data into files
  - Used the LHC detectors to write several petabytes per year
- Leverage Cling C++ reflection capabilities
- Object-wise and column-wise streaming
- Very efficient in space and run-time
- Multiple writers support
  - Multi-thread in production
  - Multi-process (via MPI) in prototype
- Multiple language support, ROOT files can be read in:
  - C++, Python, JavaScript
  - Java, Go, even Rust (Contributions)

Backup slides

CREDITS

E. Tejedor, D. Piparo, G. Amadio, A Bashyal and the rest of the ROOT Team

ROOT
Data Analysis Framework

https://root.cern

RDataFrame Basics

Can we do Better?

simple yet powerful way to analyse data with modern C++

provide high-level features, e.g.
less typing, better expressivity, abstraction of complex operations

allow transparent optimisations, e.g.
multi-thread parallelisation and caching

Improved Interfaces

what we write

```
TTreeReader reader(data);
TTreeReaderValue<A> x(reader,"x");
TTreeReaderValue<B> y(reader,"y");
TTreeReaderValue<C> z(reader,"z");
while (reader.Next()) {
    if (IsGoodEntry(*x, *y, *z))
        h->Fill(*x);
}
```

what we *mean*

- full control over the event loop
- requires some boilerplate
- users implement common tasks again and again
- parallelisation is not trivial

RDataFrame: declarative analyses

```
RDataFrame d(data);
auto h = d.Filter(IsGoodEntry, {"x","y","z"})
          .Histo1D("x");
```

- full control over *the analysis*
- no boilerplate
- common tasks are already implemented

? parallelization is not trivial?

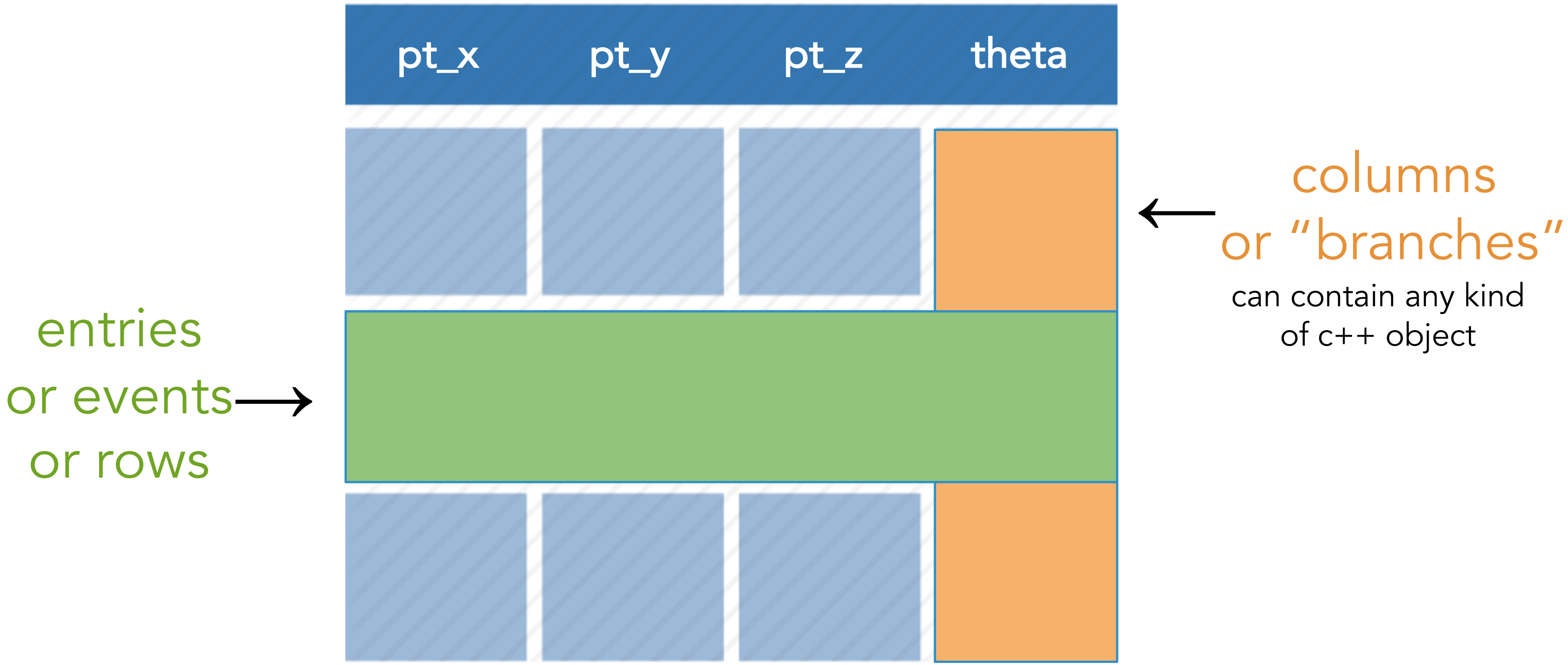
RDataFrame: declarative analyses

```
ROOT::EnableImplicitMT();
RDataFrame d(data);
auto h = d.Filter(IsGoodEntry, {"x","y","z"})
          .Histo1D("x");
```

- full control over *the analysis*
- no boilerplate
- common tasks are already implemented

? parallelization is not trivial?

Columnar Representation



RDataFrame: quick how-to

1. build a data-frame object by specifying your data-set
2. apply a series of transformations to your data
   - filter (e.g. apply some cuts) or
   - define new columns
3. apply actions to the transformed data to produce results (e.g. fill a histogram)

Creating a RDataFrame - 1 file

```
RDataFrame d1("treename", "file.root");

auto filePtr = TFile::Open("file.root");
RDataFrame d2("treename", filePtr);

TTree *treePtr = nullptr;
filePtr->GetObject("treename", treePtr);
RDataFrame d3(*treePtr); // by reference!
```

Three ways to create a RDataFrame that reads tree "treename" from file "file.root"

Creating a RDataFrame - more files

```
RDataFrame d1("treename", "file*.root");
RDataFrame d2("treename", {"file1.root","file2.root"});

std::vector<std::string> files = {"file1.root","file2.root"};
RDataFrame d3("treename", files);

TChain chain("treename");
chain.Add("file1.root); chain.Add("file2.root);
RDataFrame d4(chain); // passed by reference, not pointer!
```

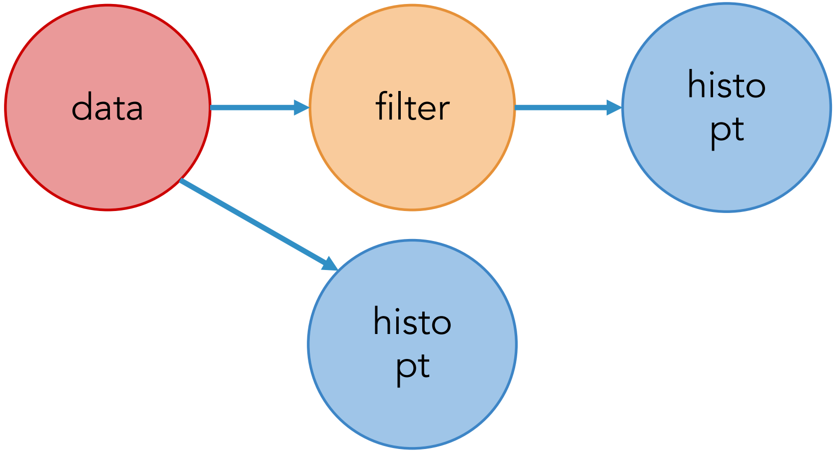
Here RDataFrame reads tree “treename” from files “file1.root” and “file2.root”

Cut on theta, fill histogram with pt

```
RDataFrame d("t", "f.root");
auto h = d.Filter("theta > 0").Histo1D("pt");
h->Draw(); // event loop is run here, when you access a result
          // for the first time
```

event-loop is run *lazily*, upon first access to the results

Think of your analysis as data-flow




```
auto h2 = d.Filter("theta > 0").Histo1D("pt");
auto h1 = d.Histo1D("pt");
```

Using callables instead of strings

```
// define a c++11 lambda - an inline function - that checks “x>0”
auto IsPos = [](double x) { return x > 0.; };
// pass it to the filter together with a list of branch names
auto h = d.Filter(IsPos, {"theta"}).Histo1D("pt");
h->Draw();
```

any callable (function, lambda, functor class) can be used as a filter, as long as it returns a boolean

Filling multiple histograms

```
auto h1 = d.Filter("theta > 0").Histo1D("pt");
auto h2 = d.Filter("theta < 0").Histo1D("pt");
h1->Draw();         // event loop is run once here h2-
>Draw("SAME"); // no need to run loop again here
```

Book all your actions upfront. The first time a result is accessed, RDataFrame will fill all booked results.

Define a new column

```
double m = d.Filter("x > y")
            .Define("z", "sqrt(x*x + y*y)")
            .Mean("z");
```

`Define` takes the name of the new column and its expression. Later you can use the new column as if it was present in your data.

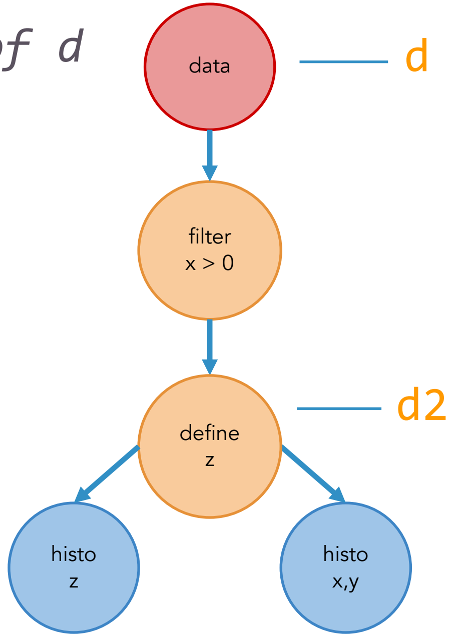
Define a new column

```
double SqrtSumSq(double, double) { return … ; }
double m = d.Filter("x > y")
            .Define("z", SqrtSumSq, {"x","y"})
            .Mean("z");
```

Just like `Filter`, `Define` accepts any callable object (function, lambda, functor class…)

Think of your analysis as data-flow

```
// d2 is a new data-frame, a transformed version of d
auto d2 = d.Filter("x > 0")
           .Define("z", "x*x + y*y");

// make multiple histograms out of it
auto hz = d2.Histo1D("z");
auto hxy = d2.Histo2D("x","y");
```



You can store transformed data-frames in variables, then use them as you would use a RDataFrame.

Cutflow reports

```
d.Filter("x > 0", "xcut")
 .Filter("y < 2", "ycut");
d.Report();
```

```
// output
xcut       : pass=49        all=100       --   49.000 %
ycut       : pass=22        all=49        --   44.898 %
```

When called on the main TDF object, `Report` prints statistics for all filters *with a name*

Running on a range of entries #1

```
// stop after 100 entries have been processed
auto hz = d.Range(100).Histo1D("x");

// skip the first 10 entries, then process one every two until the end
auto hz = d.Range(10, 0, 2).Histo1D("x");
```

Ranges are only available in single-thread executions.
They are useful for quick initial data explorations.

Running on a range of entries #2

```
// ranges can be concatenated with other transformations
auto c = d.Filter("x > 0")
          .Range(100)
          .Count();
```

This `Range` will process the first 100 entries *that pass the filter*

Saving data to file

```
auto new_df = df.Filter("x > 0")
                 .Define("z", "sqrt(x*x + y*y)")
                 .Snapshot("tree",
"newfile.root");
```

We filter the data, add a new column, and then save everything to file. No boilerplate code at all.

Creating a new data-set

```
RDataFrame d(100);
auto new_d = d.Define("x", []() { return double(rand()) / RAND_MAX; })
              .Define("y", []() { return rand() % 10; })
              .Snapshot("tree", "newfile.root");
```

We create a special TDF with 100 (empty) entries, define some columns, save it to file

N.B. `rand()` is generally [not a good way] to produce uniformly distributed random numbers

Not Only ROOT Datasets

- TDataSource: Plug *any columnar* format in RDataFrame
- Keep the programming model identical!
- ROOT provides CSV data source
- More to come
  - TDataSource is a programmable interface!
  - E.g. https://github.com/bluehood/mdfds LHCb raw format - not in the ROOT repo

Not Only ROOT Datasets

```
auto fileName = "tdf014_CsvDataSource_MuRun2010B.csv";
auto tdf = ROOT::Experimental::TDF::MakeCsvDataFrame(fileName);

auto filteredEvents =
tdf.Filter("Q1 * Q2 == -1")
.Define("m", "sqrt(pow(E1 + E2, 2) - (pow(px1 + px2, 2) + pow(py1 + py2, 2) + pow(pz1 + pz2, 2)))");

auto invMass =
filteredEvents.Histo1D({"invMass", "CMS Opendata: #mu#mu mass;mass [GeV];Events", 512, 2, 110}, "m");
```

tdf014_CsvDataSource_MuRun2010B.csv:

```
Run,Event,Type1,E1,px1,py1,pz1,pt1,eta1,phi1,Q1,Type2,E2,px2,py2,pz2,pt2,eta2,phi2,Q2,M
146436,90830792,G,19.1712,3.81713,9.04323,-16.4673,9.81583,-1.28942,1.17139,1,T,5.43984,-0.362592,2.62699,-
4.74849,2.65189,-1.34587,1.70796,1,2.73205
146436,90862225,G,12.9435,5.12579,-3.98369,-11.1973,6.4918,-1.31335,-0.660674,-1,G,11.8636,4.78984,-6.26222,-
8.86434,7.88403,-0.966622,-0.917841,1,3.10256
```

RDataFrame
Extra features

Caching

```
RDataFrame d("mytree", "myFile.root");
auto cached_d = d.Cache();
```

All the content of the TDF is now in (contiguous) memory.
Analysis as fast as it can be (vectorisation possible too).

N.B. It is always possible to selectively cache columns to save some memory!

Creating a new data-set - parallel

```
ROOT::EnableImplicitMT();
RDataFrame d(100);
auto new_d = d.Define("x", []() { return double(rand()) / RAND_MAX; })
              .Define("y", []() { return rand() % 10; })
              .Snapshot("tree", "newfile.root");
```

We create a special TDF with 100 (empty) entries, define some columns, save it to file -- in parallel

N.B. `rand()` is generally not a good way to produce uniformly distributed random numbers

More on histograms #1

```
auto h = d.Histo1D("x","w");
```

TDF can produce *weighted* TH1D, TH2D and TH3D.
Just pass the extra column name.

More on histograms #2

```
auto h = d.Histo1D({"h","h",10,0.,1.},"x", "w");
```

You can specify a model histogram with a set axis range, a name and a title (optional for TH1D, mandatory for TH2D and TH3D)

Filling histograms with arrays

```
auto h = d.Histo1D("pt_array", "x_array");
```

If `pt_array` and `x_array` are an array or an STL container (e.g. std::vector), TDF fills histograms with all of their elements. `pt_array` and `x_array` are required to have equal size for each event.

C++ / JIT / PyROOT

Pure C++

```
d.Filter([](double t) { return t > 0.; }, {"th”})
 .Snapshot<vector<float>>("t","f.root",{"pt_x"});
```

C++ and JIT-ing with CLING

```
d.Filter("th > 0").Snapshot("t","f.root","pt*");
```

pyROOT -- just leave out the ;

```
d.Filter("th > 0").Snapshot("t","f.root","pt*")
```