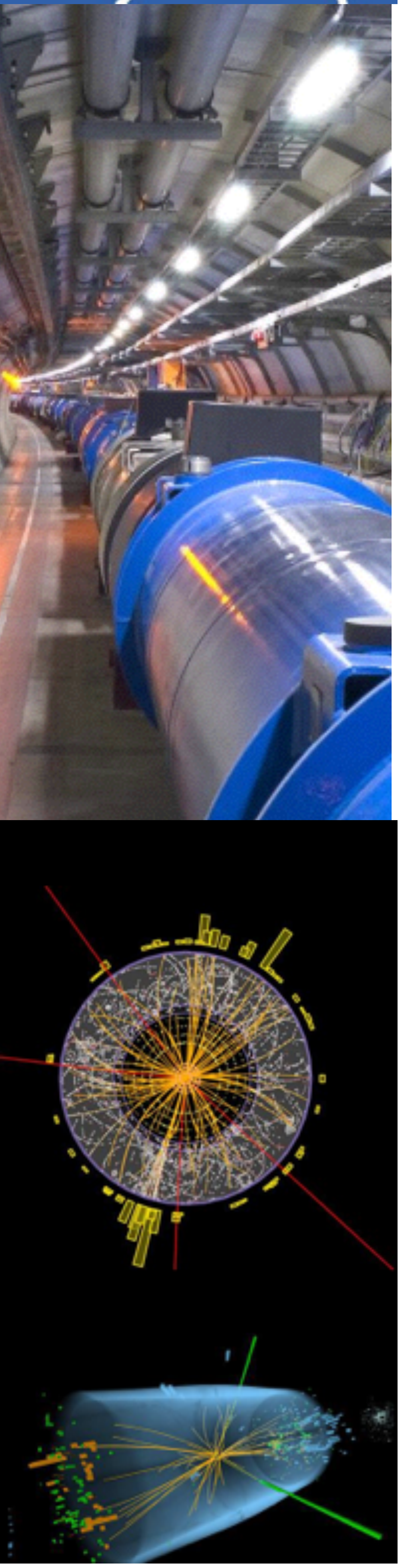
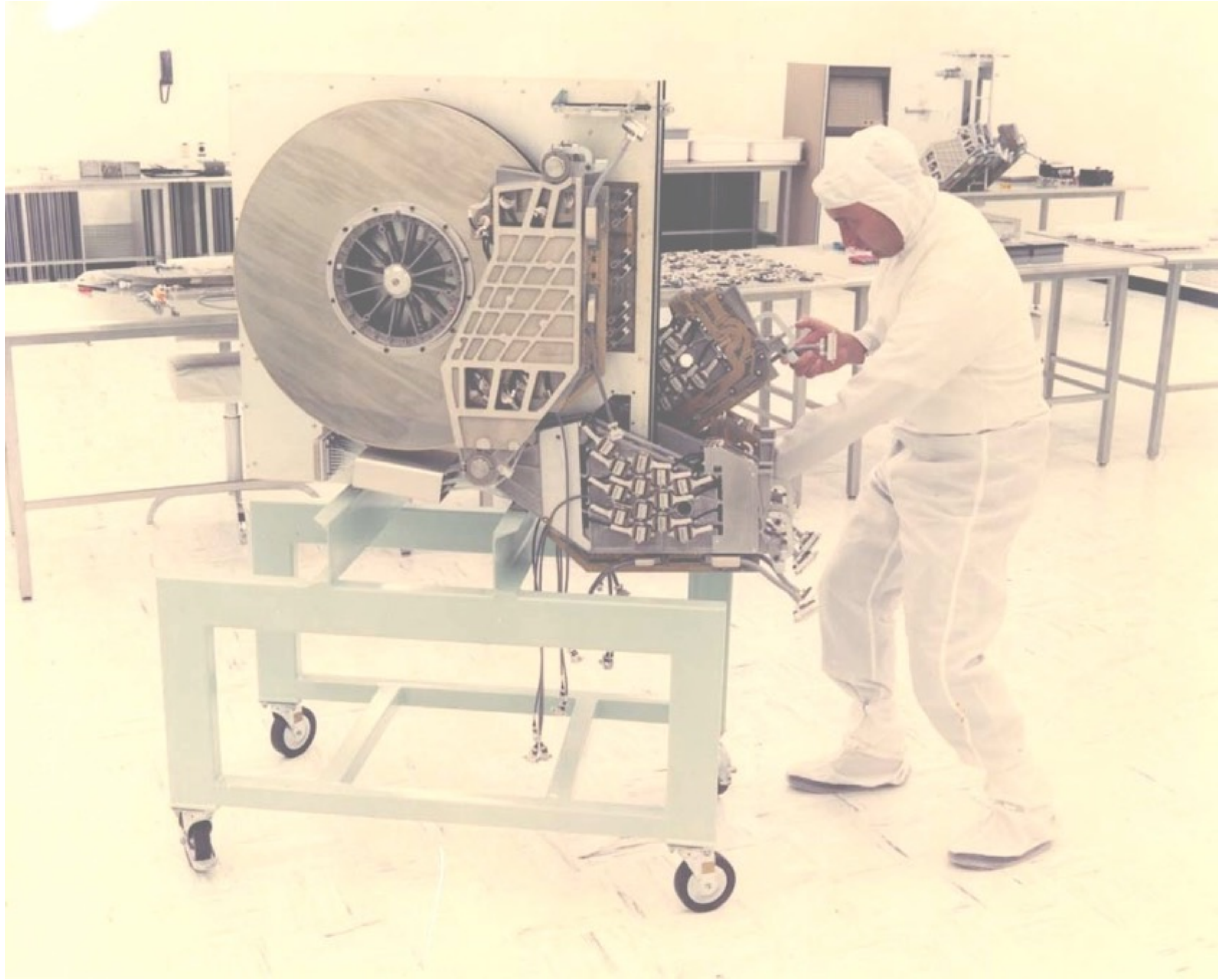
Introduction to ROOT I/O





- Introduction to I/O in ROOT
  - how to save ROOT objects in a file
    - reflection
  - example: saving an histogram

Dr Lorenzo Moneta
CERN PH-SFT
CH-1211 Geneva 23
sftweb.cern.ch
root.cern.ch

Saving Objects

Cannot do in C++:

```
TNamed* o = new TNamed("name","title");
std::write("file.bin", "obj1", o);
TNamed* p = std::read("file.bin", "obj1");
p->GetName();
```

E.g. LHC experiments use C++ to manage data

Need to write C++ objects and read them back

- `std::cout` not an option: 15 PetaBytes / year of processed data (i.e. data that will be read)

Saving Objects – Saving Types

What's needed?

```
TNamed* o = new TNamed("name","title");
std::write("file.bin", "obj1", o);
TNamed* p = std::read("file.bin", "obj1");
p->GetName();
```

Cannot do in C++

Store *data members* of `TNamed`; need to know:

1) type of object
2) data members for the type
3) where data members are in memory
4) read their values from memory, write to disk

Reflection

Need type description (aka *reflection*)

1. types, sizes, members

`TMyClass` is a class.

```
class TMyClass {
  float fFloat;
  Long64_t fLong;
};
```

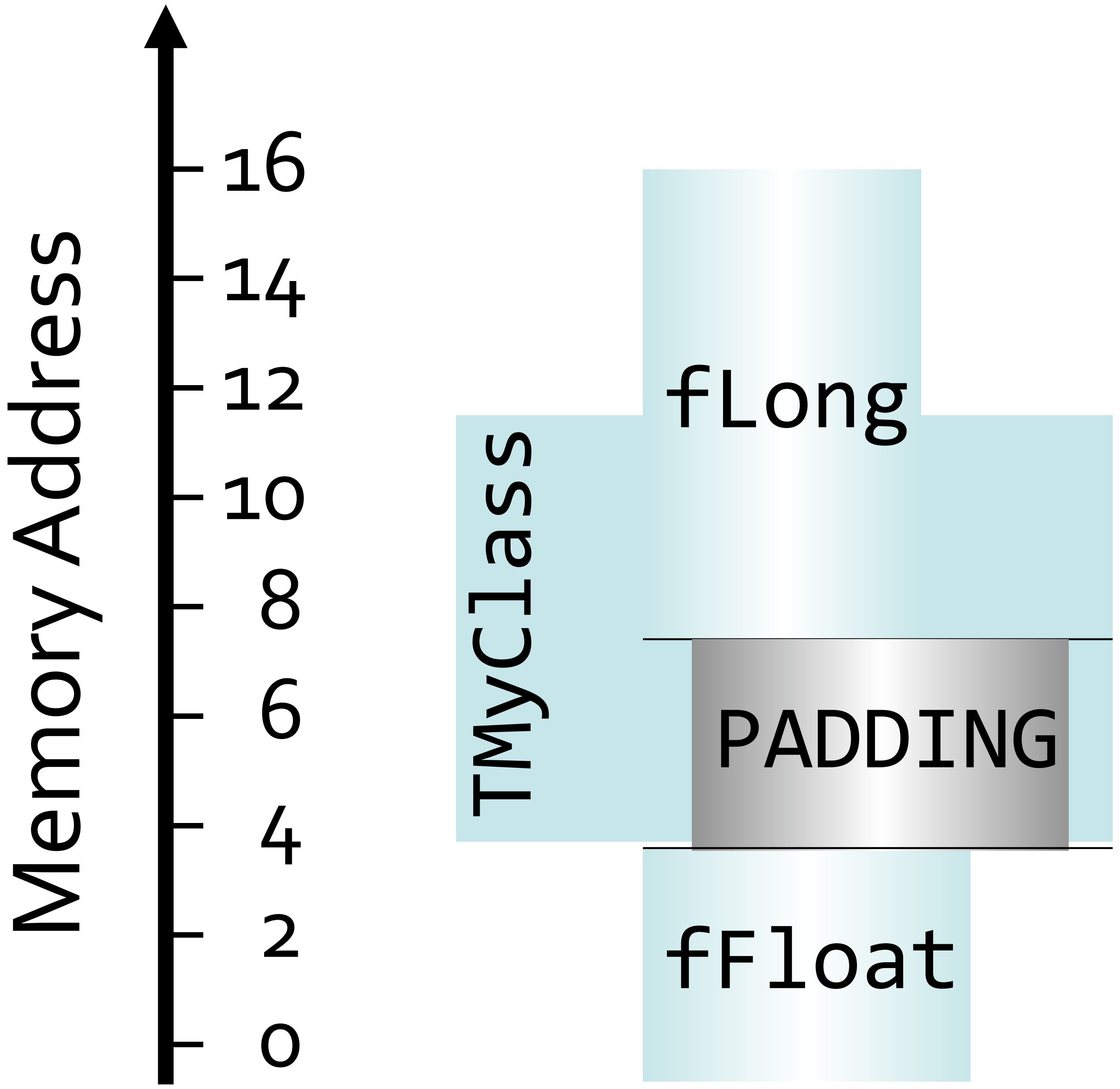
Members:

- "`fFloat`", type `float`, size 4 bytes
- "`fLong`", type `Long64_t`, size 8 bytes

Need type description (platform and compiler dependent)

1. types, sizes, members
2. offsets in memory

```
class TMyClass {
  float fFloat;
  Long64_t fLong;
};
```



"fFloat" is at offset 0

"fLong" is at offset 8

ROOT And Reflection

- How to generate the reflection information (dictionary) in ROOT ?
  - Simply use ACLiC:

```
.L MyCode.cxx+
```

  - Creates dictionary library of all types defined in MyCode.cxx!

- Dictionaries are needed also for interpreter
- ROOT has already dictionaries for all its types

Saving Objects in ROOT

- Use the `TFile` class
  - we need first to open the file by creating a TFile object

```
TFile* f = TFile::Open("file.root","NEW");
```

use option "`RECREATE`" if the file already exists

- Write an object deriving from `TObject`:

```
object->Write("optionalName")
```

if the optionalName is not given the object will be written in the file with its original name (`object->GetName()`)

- For any object (but with dictionary)

```
f->WriteObject(object, "name");
```

TFile Class

- ROOT stores objects in TFiles:

```
TFile* f = TFile::Open("file.root", "NEW");
```

- TFile behaves like file system:

```
f->mkdir("dir");
```

- TFile has a current directory:

```
f->cd("dir");
```

- You can browse the content:

```
f->ls();
 TFile**            file.root
 TFile*             file.root
  TDirectoryFile*         dir    dir
  KEY: TDirectoryFile     dir;1  dir
```

Saving Histogram in a File

- How to save objects in a file

```
TFile* f = TFile::Open("myfile.root",”NEW”);
TH1D* h1 = new TH1D(“h1”,”h1”,100,-5.,5.);

h1->FillRandom(“gaus”);  // fill histogram with random data

h1->Write();

delete f;
```

- TFile compresses data using ZIP

```
h1->Write();
f->GetCompressionFactor()
(Float_t)1.685546875000000e+00
```

Where is My Histogram ?

- All histograms and trees are owned by `TFile` which acts like a scope
- After closing the file (i.e when the file object is deleted) also the histogram, trees and graphs objects are deleted
- This code will crash ROOT:

```
TFile* f = TFile::Open("myfile.root",”RECREATE”);

TH1D* h1 = new TH1D(“h1”,”h1”,100,-5.,5.);

delete f;

h1->Draw(); // will crash - DO NOT DO IT!!!

 *** Break *** segmentation violation
```

- Other objects (e.g graphs) will be still there and can be accessed afterwards
- This can be changed with

```
TH1::AddDirectory(false);
```

Reading a File

- Reading is simple:

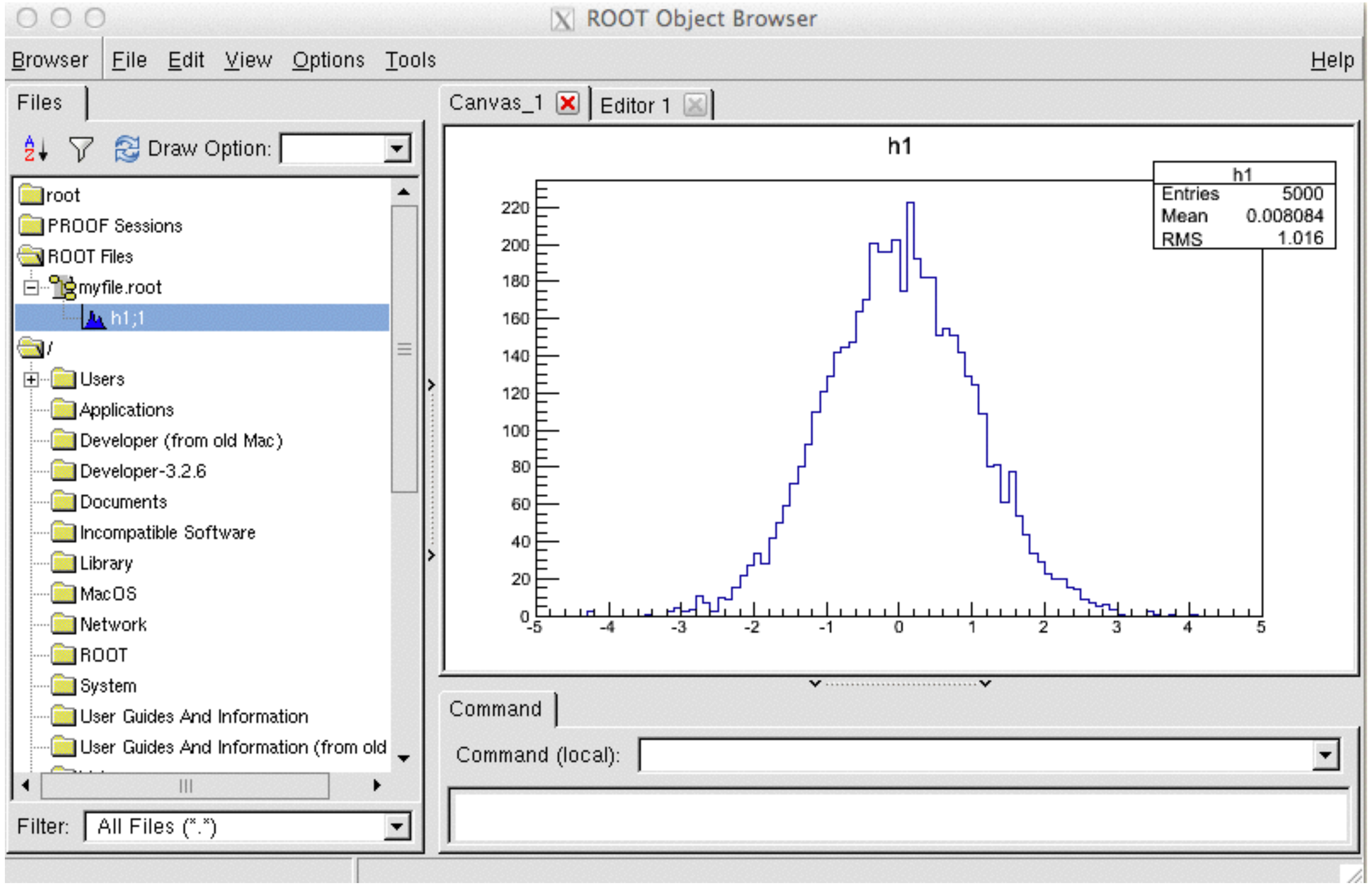
```
TFile* f = TFile::Open("myfile.root");
TH1* h = 0;
f->GetObject("h", h);
h->Draw();
delete f;
```

- Can also use
  - `TH1 * h = (TH1*) f->Get("h1");`
  - `TH1 * h = (TH1*) f->GetObjectChecked("h1",”TH1”);`
    - which returns a null pointer if the read object is not of the right type
- Remember:
  - TFile owns the histogram
  - the histogram is gone when the file is closed

TBrowser

- GUI for browsing ROOT objects written in a file

```
root [0] new TBrowser();
```



File Merging

- ROOT file containing the same data objects (e.g. histograms, Trees, etc...) can be merged using the command line tool $ROOTSYS/bin/hadd

```
$>  hadd  fileOut.root file1.root file2.root file3.root
```

```
$>  hadd  -h
Usage: hadd [-f[0-9]] [-k] [-T] [-O] [-n maxopenedfiles] [-v verbosity] targetfile
source1 [source2 source3 ...]
This program will add histograms from a list of root files and write them
to a target root file. The target file is newly created and must not
exist, or if -f ("force") is given, must not be one of the source files.
Supply at least two source files for this to make sense... ;-)
```

- hadd use functionality of TObject::Merge to merge the contained ROOT

Time for Exercises!

Put in practice the concepts to which you were just exposed: read the instructions and solve a simple exercises on ROOT I/O

Exercise: Writing and Reading Histograms from a file

ROOT Trees

- ROOT Trees:
  - `TNtuple` class (a simple Tree)
  - `TTree` class
- How to create a Tree and to write in a file
- How to read a Tree and query variables
- How to analyze the Tree
- Merging of Trees: `TChain`
- Using Tree Friends

Ntuple

- Ntuple class:
  - `TNtuple`
    - for storing tabular data
    - e.g. various rows with numbers

E	px	py	pz	pt	eta	phi
51.238284	25.706396	-0.238289	44.322524	25.7075	1.31298	-0.009269
61.68896	-21.06428	21.67916	53.775813	30.2273	1.34022	2.34181
69.232896	29.087514	21.827313	58.912468	36.3664	1.25952	0.643758
55.435615	-19.07243	-32.10133	40.973823	37.3397	0.948547	-2.10689
44.081717	-21.76992	18.725977	33.445573	28.7157	0.993186	2.43122
46.970421	-4.27962	-41.94243	-20.70599	42.1602	-0.473261	-1.67248
109.01623	14.823946	-24.63801	105.15587	28.7538	2.00801	-1.02915
81.517424	35.058621	-9.93301	-72.91995	36.4386	-1.44416	-0.27609
59.76741	-18.99337	-21.60084	52.390826	28.7636	1.3608	-2.29205
59.123105	-11.88863	56.536229	12.564104	57.7727	0.215796	1.77806
125.44969	30.001331	11.727174	-121.2436	32.2119	-2.03581	0.372627
44.246096	24.595778	-7.317789	36.044621	25.6613	1.14067	-0.289182
42.820008	29.287483	5.4862204	30.752201	29.7969	0.903863	0.185177
28.761648	-28.45137	-3.030433	-2.927228	28.6123	-0.102129	-3.03548
44.427016	34.281963	-11.5006	25.811686	36.1596	0.663957	-0.323673

ROOT Ntuple class

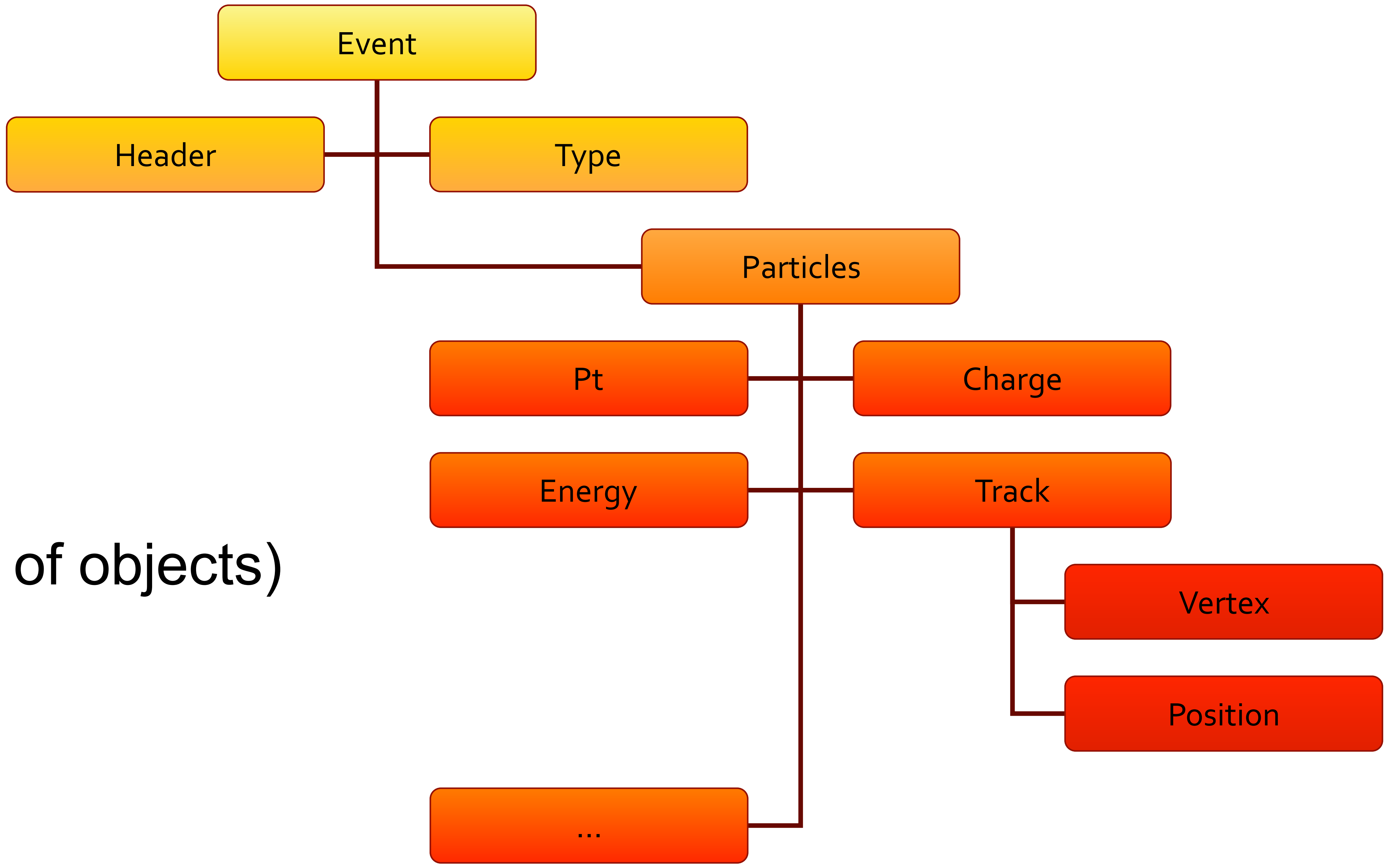
- ROOT N-tuple can store only floating point variables
  - limitation that all variables must be of the same type

```
TNtuple data("ntuple","Example N-tuple","x:y:z:t");

// fill it with random data
for (int i = 0; i<10000; ++i) {
   float x = gRandom->Uniform(-10,10);
   float y = gRandom->Uniform(-10,10);
   float z = gRandom->Gaus(0,5);
   float t = gRandom->Exp(10);

   data.Fill(x,y,z,t);
}
// write in a file
TFile f("ntuple_data.root","RECREATE");
data.Write();
f.Close();
```

  - not really useful for storing complex analysis data

ROOT Trees

- Ntuple class:
  - `TNtuple`
    - for storing tabular data
    - e.g. Excel Table with numbers



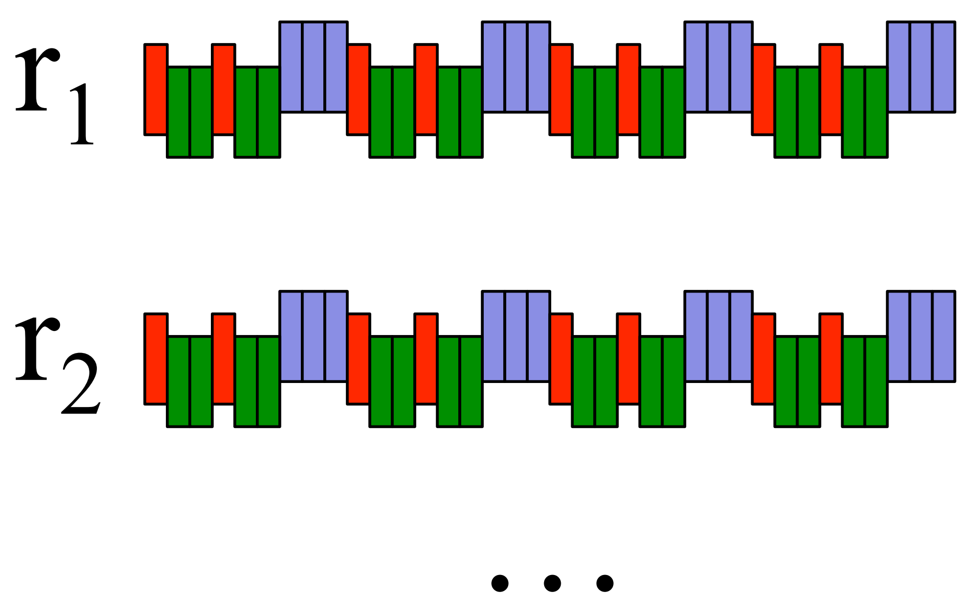
- Tree class in ROOT
  - `TTree`
    - for storing complex data types (any type of objects)

Why Trees ?

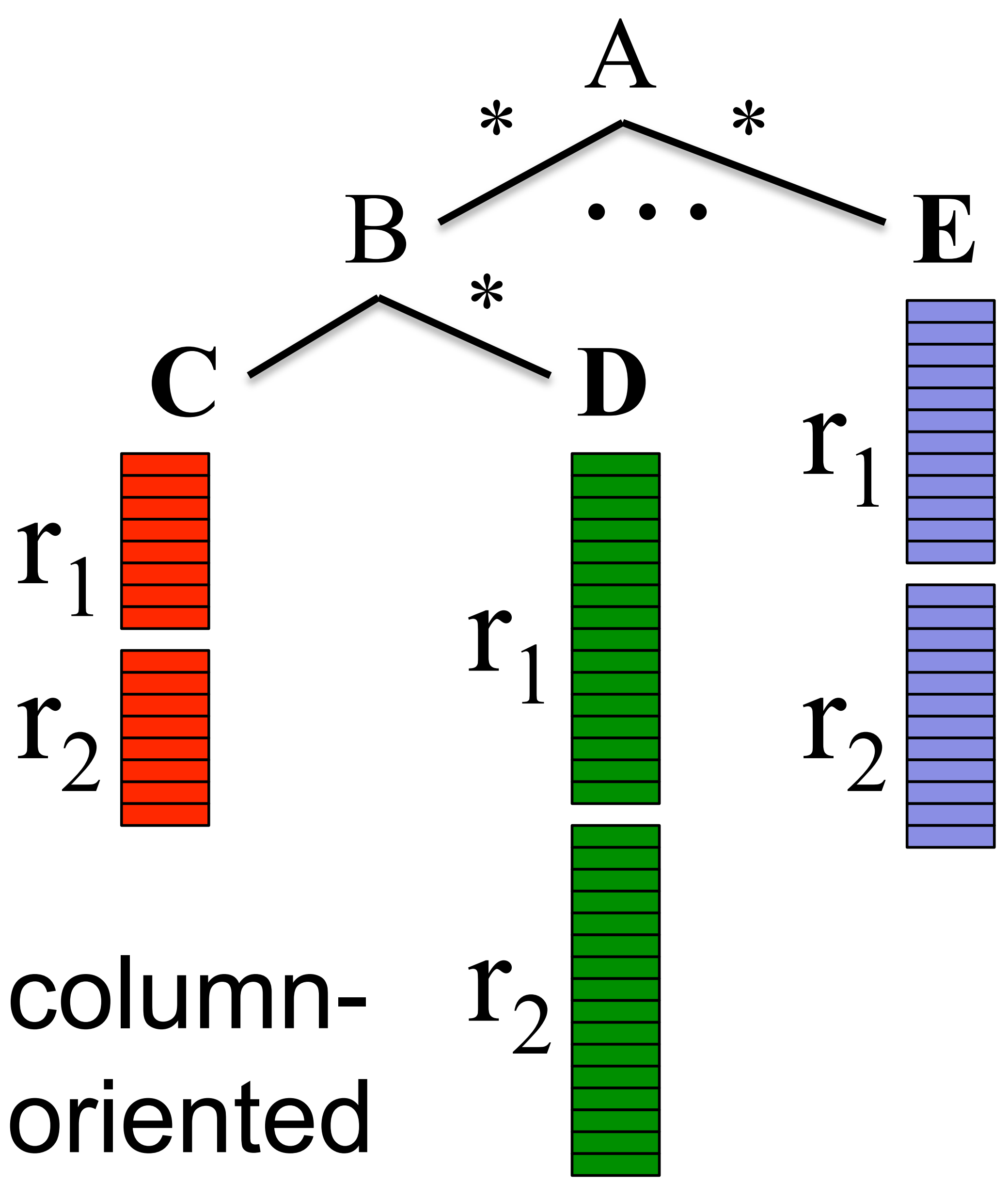
- `object.Write()` is convenient for simple objects like histograms, but inappropriate for saving collections of events containing complex objects
- When reading a collection (e.g. a TObjArray)
  - read all elements (all events) in memory
- With trees:
  - only a part of it (less I/O)
- Trees buffered to disk (TFile);
  - I/O is integral part of TTree concept
- Trees can read only a sub-set of all events
  - only the selected columns
  - Trees have a column oriented storage

Tree Access

- Databases have typically row wise access/storage
  - Can only access the full object (e.g. full event)
- ROOT trees have column wise access
  - Designed to access the object or a subset of the object attributes (e.g. only particles' energy)



record-oriented



column-oriented

Column vs Row Wise Access

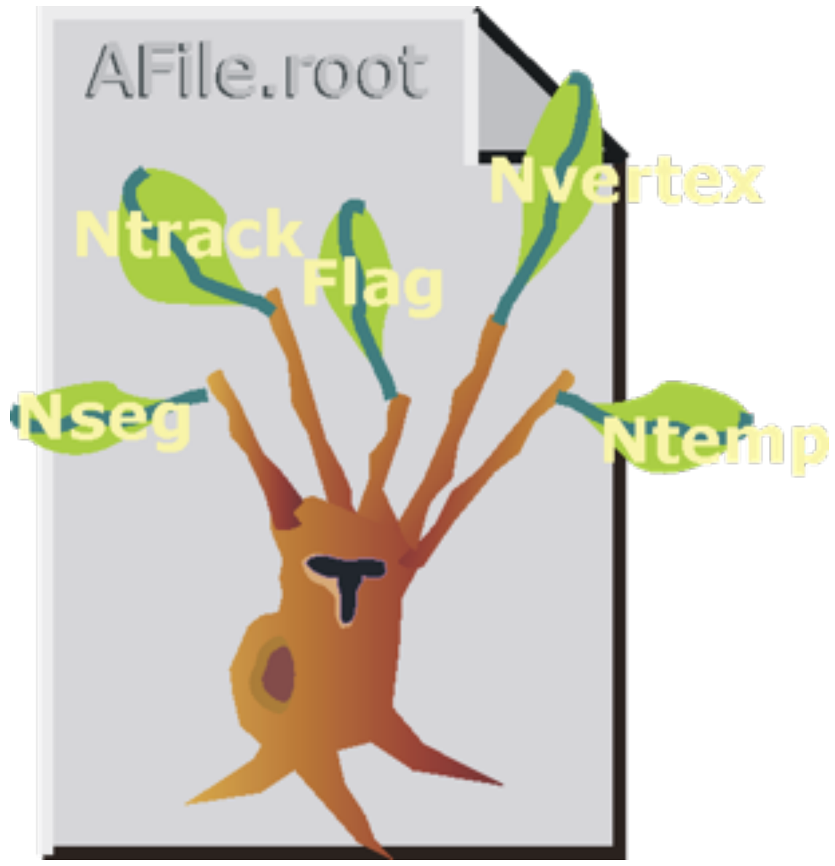
- Advantage of column wise representation:
  - can read only interest part of event, e.g. read only the event muon candidates
    - Less I/O operation: → faster to read
  - same members consecutive, e.g. for object with position in X, Y, Z, and energy E, all X are consecutive, then come all Y, then Z, then E.
    - much higher compression efficiency: → less space on disk

- Disadvantage:
  - more expensive to write
    - adding new events to an existing tree
- ROOT Trees are designed to write once and to read many times

ROOT Tree Structure: Branches

- A ROOT Tree is composed of Branches
  - a Branch (TBranch) can hold a simple variable, a list of variables, an object or even a collection of objects
    - no splitting: the whole object is written in the branch
    - splitting: the object member are assigned to separate branches

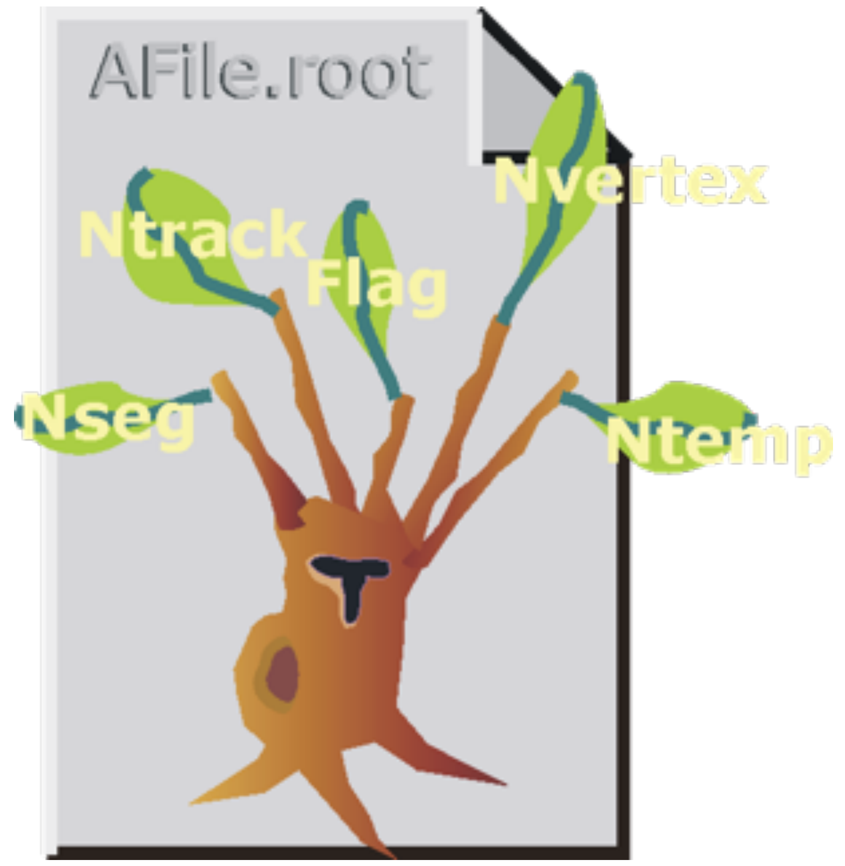
no splitting



splitting

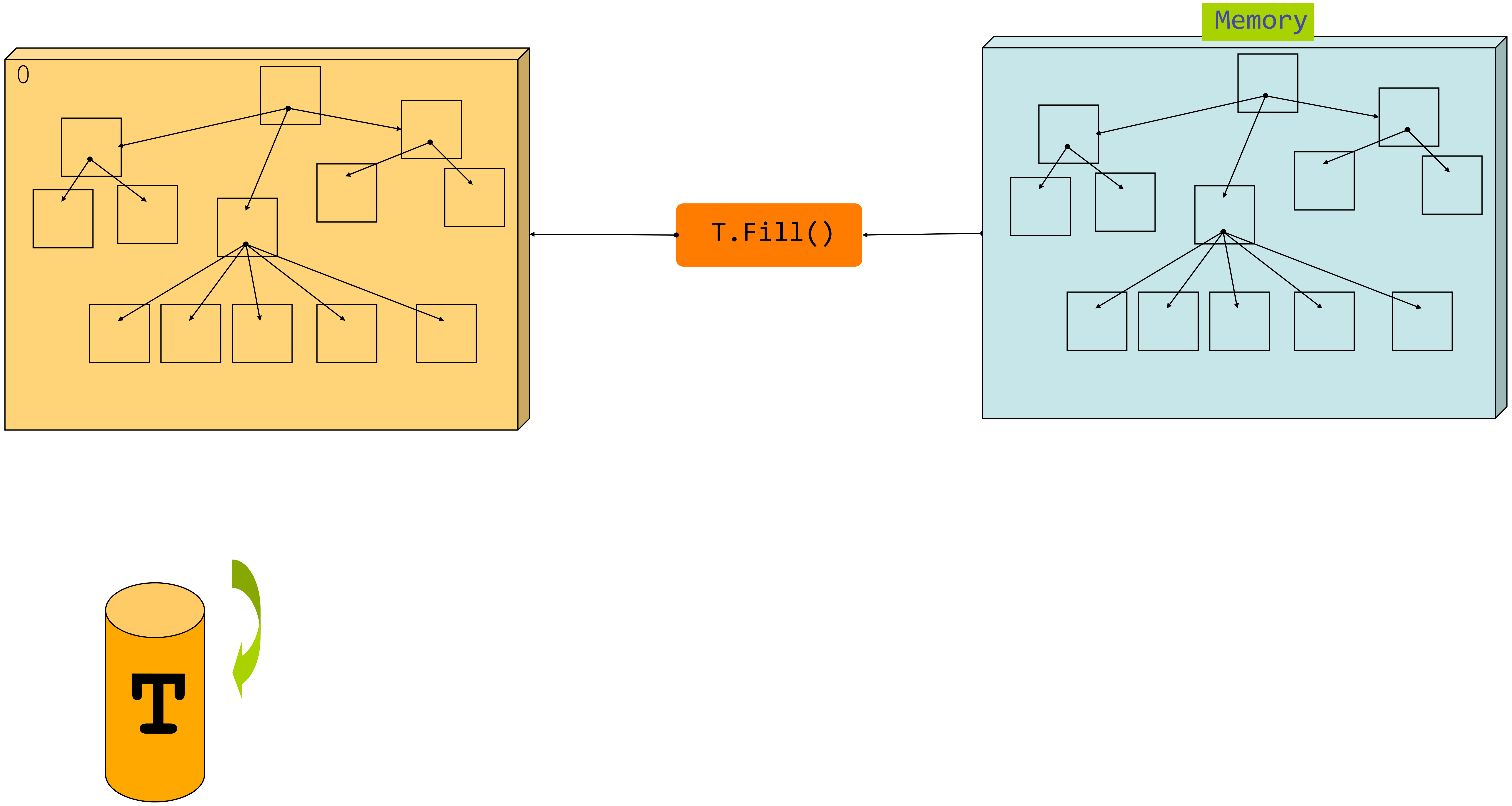
Branches and Leafs

- The leaves (TLeaf) are the data containers of the branch
- It is possible to read only a sub-set of all the branches in a tree
  - variables or object known to be used together should be put in the same branch
    - faster read-access
- Branches of the same tree can also be written to separate files



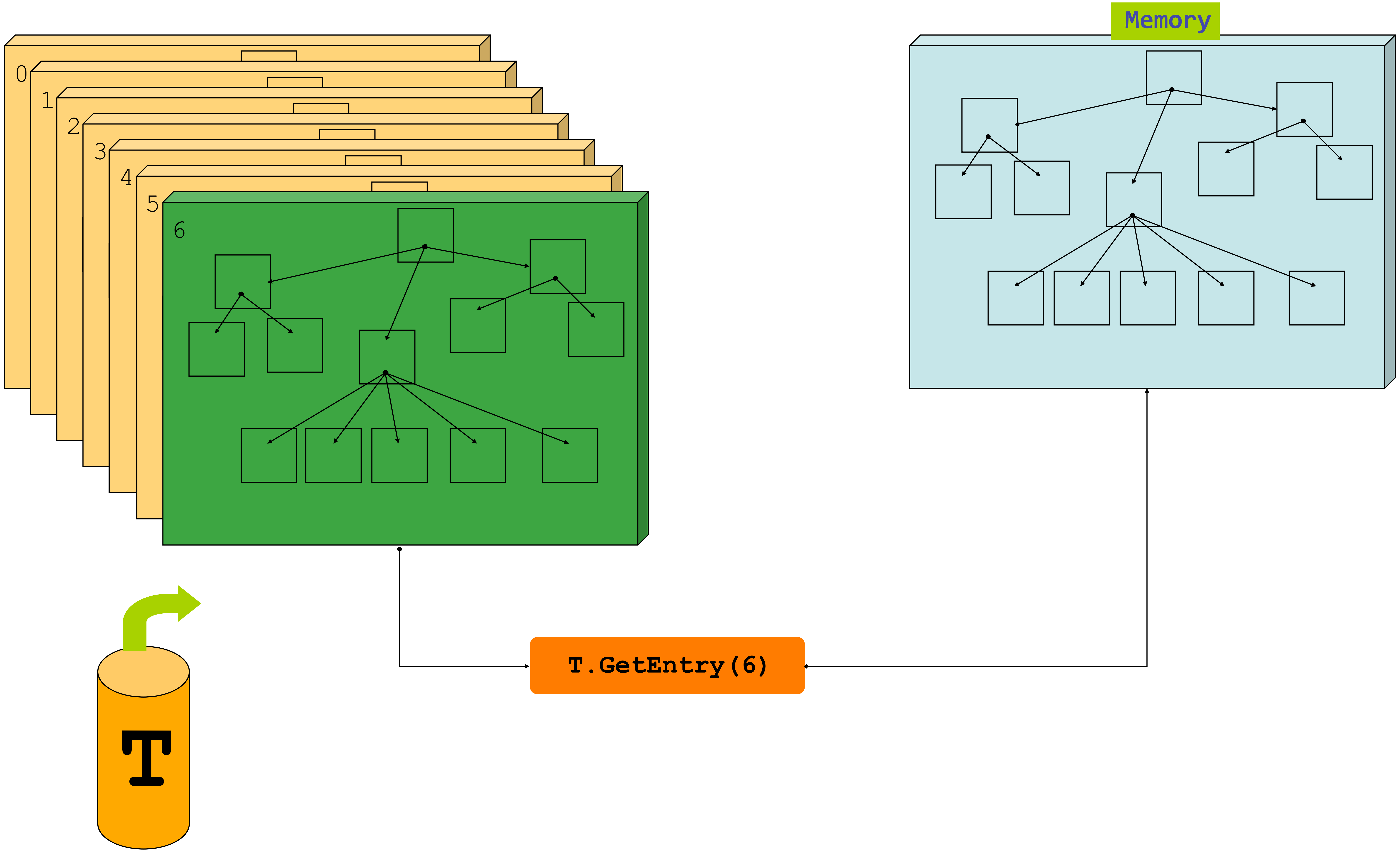
Memory ↔ Tree

- Each Node is a branch in the Tree



Memory ↔ Tree

- Each Node is a branch in the Tree



5 Steps to Build a Tree

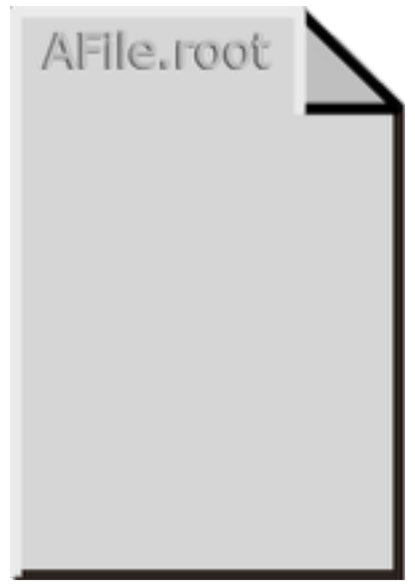
- Steps:

1. Create a TFile
2. Create a TTree
3. Add a TBranch to a TTree
4. Fill the Tree
5. Write the file

Building a ROOT Tree (1 and 2)

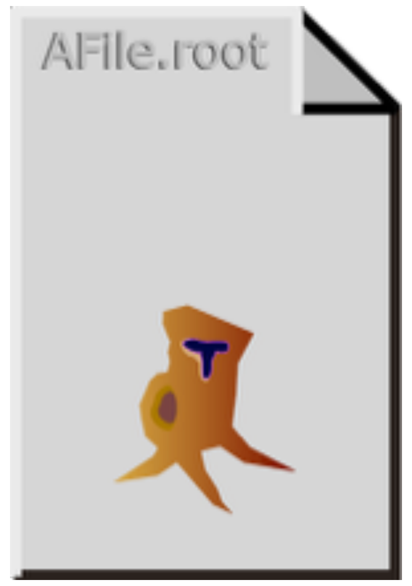
- Step 1:
  - Create a TFile class
    - Tree can be huge → need file for swapping filled entries

```
TFile *hfile = TFile::Open("AFile.root","RECREATE");
```



- Step 2:
  - Create a TTree class

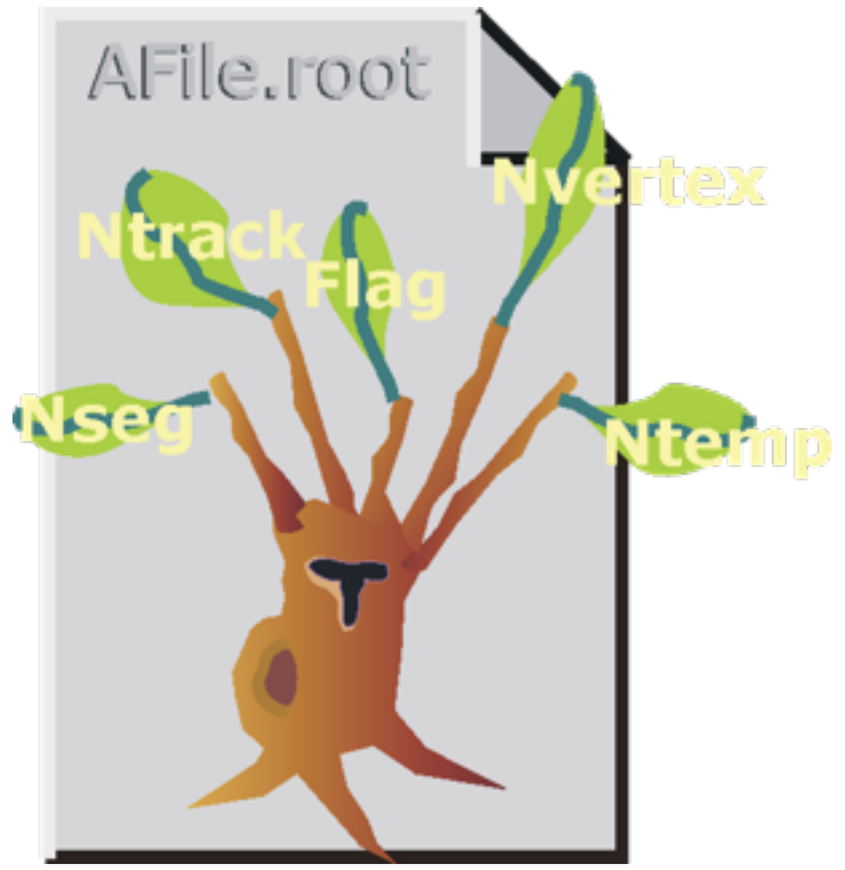
```
TTree *tree = new TTree(“myTree”,”A Tree”);
```



Adding a Branch to the Tree

- Step 3: adding a branch. We need:
  - –Name of the Branch (e.g. “eBranch”)
  - –Address of the pointer to the object we want to store (e.g. Event **)
    - optionally we can specify also:
      - –branch buffer size (default is 32000)
      - –split level (default is 99, max splitting)

```
Event *myEvent = new Event();
myTree->Branch("eBranch", &myEvent);
```

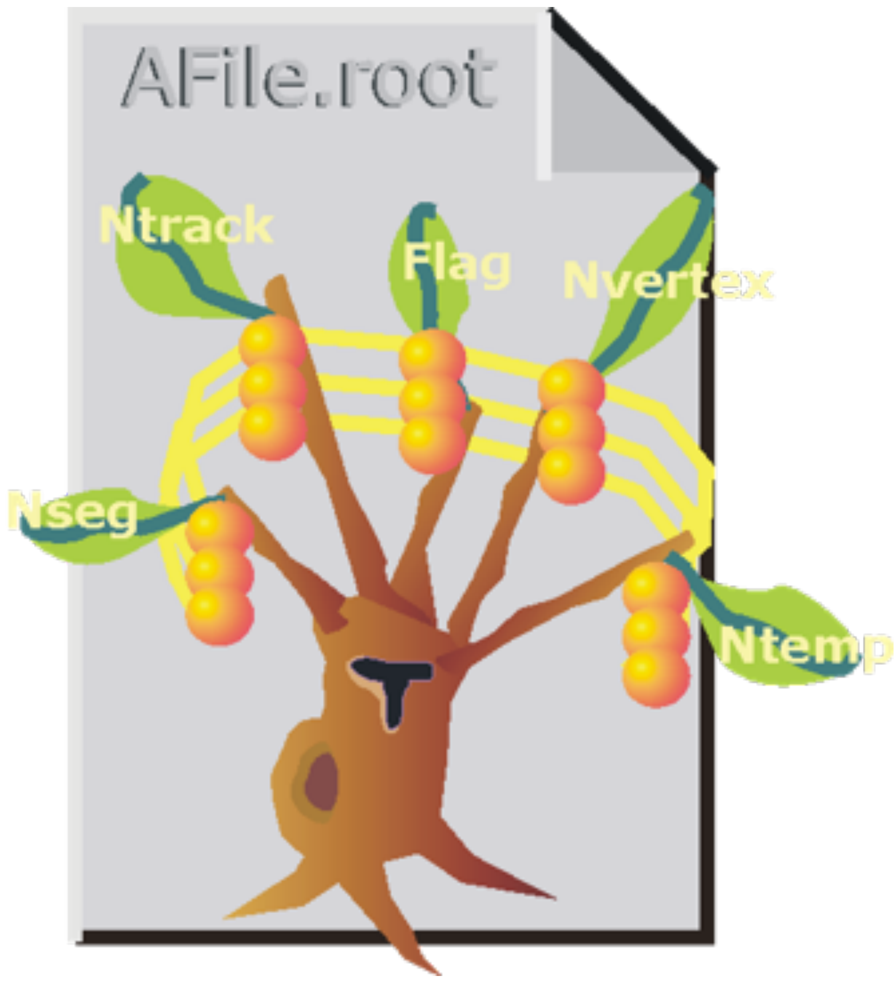


`myEvent` is an hypothetical object of type `Event` we want to store in the tree

Note that we need to have generated the ROOT dictionary for the object we want to store

Fill the Tree

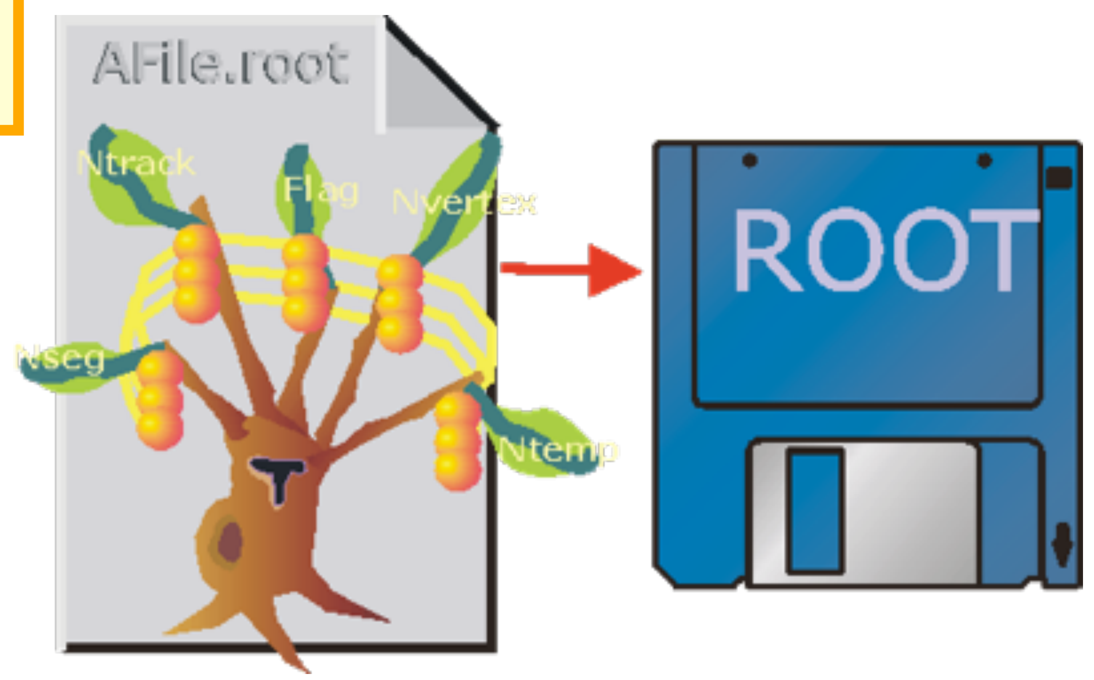
- Loop on the tree
- assign values to the object we want to store
  - e.g. by calling `myEvent->Generate`
- call `TTree::Fill()` creates a new entry in the tree:
  - snapshot of values of branches' objects




```
for (int e=0;e<100000;++e) {
   myEvent->Generate(e); // fill event
   myTree->Fill();       // fill the tree
}
```

- After, write Tree to file:

```
myTree->Write();
```



Example Macro

- Example on how to create a TTree with the object “Event”, fill with 10000 events and write to the file

```
void WriteTree()
{
   Event *myEvent = new Event();
   TFile f("AFile.root", "RECREATE");
   TTree *t = new TTree("myTree","A Tree");
   t->Branch("eBranch",&myEvent, 32000, 99);
   for (int e=0;e<100000;++e) {
      myEvent->Generate();  // hypothetical
      t->Fill();
   }
   t->Write();
}
```

Note: Event is an hypothetical class provided by the user
In TTree::Branch you can specify buffer size (32000) and split level (99)

Tree's with list of variables

- In case of a Tree containing a simple list of variables or array of variable, a variant exists:

```
void WriteTree()
{
   Int_t ntrack;
   Double_t p[100];
    TFile f("AFile.root", "RECREATE");
    TTree *t = new TTree("simpleTree","A Simple Tree");
    t->Branch("ntrack",&ntrack,"ntrack/I");
    t->Branch("p",p,"p[ntrack]/F");
    for (int e=0;e<100000;++e) {
        ntrack=...
       for (int i = 0; i < ntrack; ++i) p[i]=....
        t->Fill();
    }
    t->Write();
}
```

Time for Exercises!

Put in practice the concepts to which you were just exposed: read the instructions and solve the exercises on creating a Tree

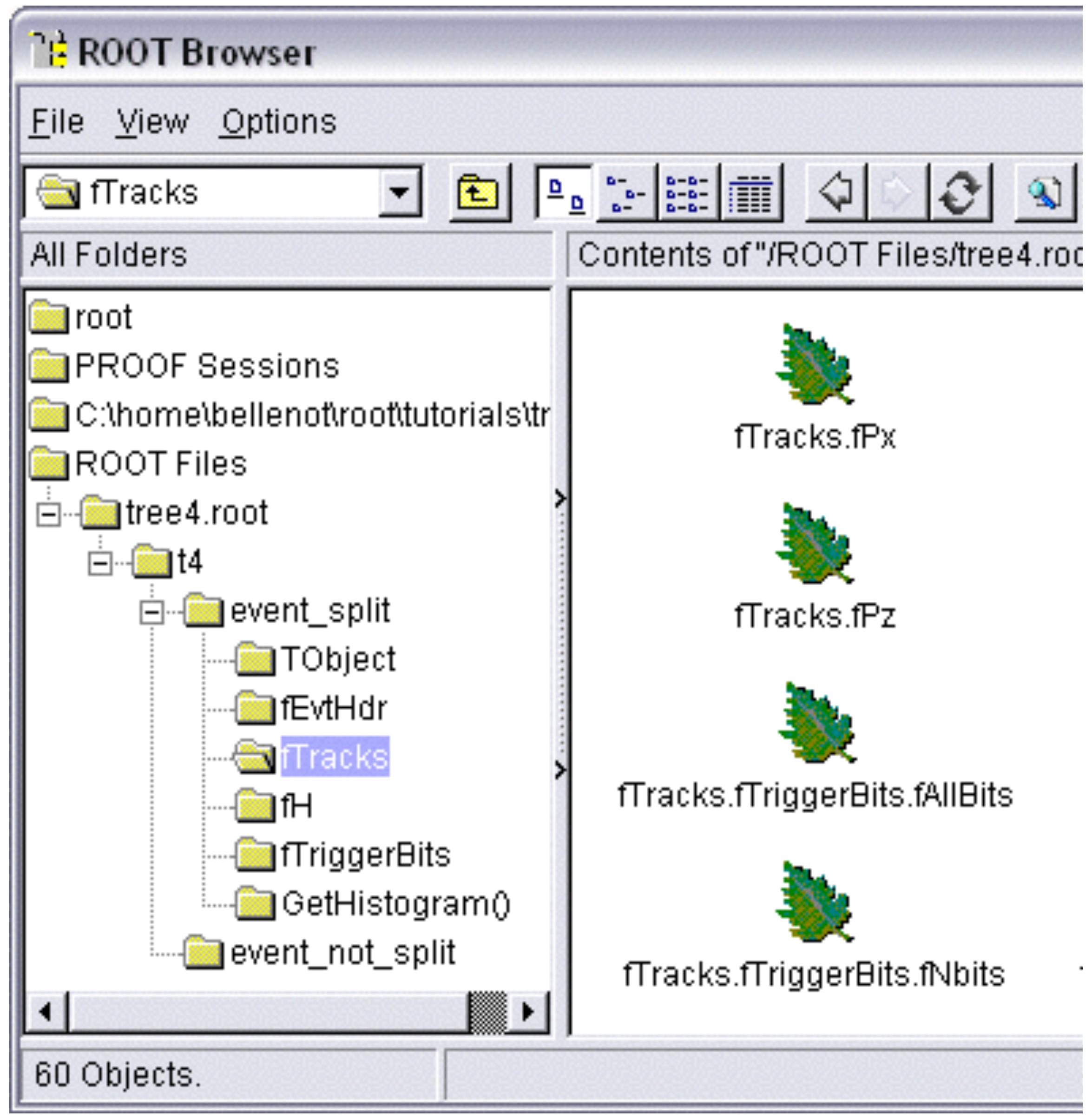
Exercise: Creating a ROOT Tree

Reading a Tree

- Open the file and get the TTree object from the file

```
TFile f("AFile.root");
TTree *myTree = 0;
f.GetObject("myTree",myTree);
```

- Or browse the TTree using the TBrowser



- TTree::Print() shows the data layout
  – list of branches

- TTree::Draw("expression","selection")

Scanning a Tree

- Syntax for querying a tree
  - Print the first 8 variables of the tree:

```
MyTree->Scan();
```

  - Prints all the variables of the tree:

```
MyTree->Scan("*");
```

  - Prints the values of var1, var2 and var3.

```
MyTree->Scan("var1:var2:var3");
```

  - A selection can be applied in the second argument:
  - Prints the values of var1, var2 and var3 for the entries where var1 is greater than 0

```
MyTree->Scan("var1:var2:var3", "var1>0");
```

- Use the same syntax as `TTree::Draw()`

Looking at the Tree

- More on scanning the Tree

```
root [] myTree->Scan("fEvtHdr.fDate:fNtrack:fPx:fPy","",
                 "colsize=13 precision=3 col=13:7::15.10");

******************************************************************************
* Row * Instance * fEvtHdr.fDate * fNtrack *           fPx *             fPy *
******************************************************************************
*   0 *        0 *        960312 *     594 *          2.07 *     1.459911346 *
*   0 *        1 *        960312 *     594 *         0.903 *   -0.4093382061 *
*   0 *        2 *        960312 *     594 *         0.696 *    0.3913401663 *
*   0 *        3 *        960312 *     594 *        -0.638 *     1.244356871 *
*   0 *        4 *        960312 *     594 *        -0.556 *   -0.7361358404 *
*   0 *        5 *        960312 *     594 *         -1.57 *   -0.3049036264 *
*   0 *        6 *        960312 *     594 *        0.0425 *    -1.006743073 *
*   0 *        7 *        960312 *     594 *          -0.6 *    -1.895804524 *
```

Looking at the Tree

- `TTree::Show(entry_number)` shows values for one entry

```
root [] myTree->Show(0);

======> EVENT:0
eBranch          = NULL
fUniqueID        = 0
fBits            = 50331648
[...]
fNtrack          = 594
fNseg            = 5964
[...]
fEvtHdr.fRun     = 200
[...]
fTracks.fPx      = 2.066806, 0.903484, 0.695610,-0.637773,…
fTracks.fPy      = 1.459911, -0.409338, 0.391340, 1.244357,…
```

Analysis of TTree

- Different ways of analyzing trees:
  - inspection of variables:
    - use TTree::Draw() which can be extended using a function defined in a C macro (see TTree::MakeProxy)
  - write your own C++ code
    - require declaring and setting address for branches
    - ROOT provides a facility for creating some skeleton analysis code to read and loop a Tree
      - TTree::MakeClass
    - User still control iterations on TTree
- Using TSelector
  - ROOT controls iterations
    - can be parallelized using PROOF

TTree in Analysis (TTree::Draw)

- TTree::Draw for interactive queries of a Tree
  - suppose we have a tree with a branch "tracks" containing a `std::vector<ROOT::Math::XYZTVector>`

```
******************************************************************************
*Br    0 :tracks    : Int_t tracks_                                           *
*Entries :    10000 : Total  Size=     103261 bytes  File Size  =      28261 *
*Baskets :        5 : Basket Size=      32000 bytes  Compression=   2.84     *
*............................................................................*
*Br    1 :tracks.fCoordinates.fX : Double_t fX[tracks_]                      *
*Entries :    10000 : Total  Size=    8079269 bytes  File Size  =    7819412 *
*Baskets :      249 : Basket Size=    3990016 bytes  Compression=   1.03     *
*............................................................................*
*Br    2 :tracks.fCoordinates.fY : Double_t fY[tracks_]                      *
*Entries :    10000 : Total  Size=    8079269 bytes  File Size  =    7819897 *
*Baskets :      249 : Basket Size=    3990016 bytes  Compression=   1.03     *
*............................................................................*
*Br    3 :tracks.fCoordinates.fZ : Double_t fZ[tracks_]                      *
*Entries :    10000 : Total  Size=    8079269 bytes  File Size  =    7786816 *
*Baskets :      249 : Basket Size=    3990016 bytes  Compression=   1.04     *
*............................................................................*
*Br    4 :tracks.fCoordinates.fT : Double_t fT[tracks_]                      *
*Entries :    10000 : Total  Size=    8079269 bytes  File Size  =    7663469 *
*Baskets :      249 : Basket Size=    3990016 bytes  Compression=   1.05     *
*............................................................................*
```

TTree::Draw syntax

`TTree::Draw`("expression", "selection(weight)")

- draw X component of all tracks

```
tree->Draw("tracks.fX");
```

- draw Eta of all tracks

```
tree->Draw("tracks.Eta()");
```

- draw Eta of tracks with pt > 5

```
tree->Draw("tracks.Eta()","tracks.Pt()> 5");
```

- draw number of tracks

```
tree->Draw("@tracks.size()");
```

  – note special symbol "@" to access collection object

TTree::Draw syntax (2)

- draw Pt of first track

```
tree->Draw("tracks[0].Pt");
tree->Draw("@tracks.front().Pt()""
```

- draw Px vs Py for all tracks

```
tree->Draw("tracks.X():tracks.Y()","","colz");
```

  - note we passed a graphics option colz for the histogram

- draw P vs Eta in a TProfile plot with 30 bins [-3,3]

```
tree->Draw("tracks.Pt():tracks.Eta() >> ph(30,-3,3)",
           "","prof");
```

- see more in TTree::Draw documentation

Special TTree::Draw functions

- These functions can be used to build `TTree::Draw` expressions:
  - Entry$ return the tree entry number

```
tree->Draw("Entry$");
```

  - Length$(formula) : return the total number of element

```
tree->Draw("Length$(tracks)");
```

  - Sum$(formula) : return the total sum of element

```
tree->Draw("Sum$(tracks)");
```

- More functions are available, see `TTree::Draw` documentation

TTree::Draw

- `TTree::Draw` is powerful and can make queries on variable of a tree and function of variables
- can call simple member functions of objects
  - member functions with void arguments or taking values
- cannot call member functions having objects as arguments
  - e.g. this does not work !

```
tree->Draw("(tracks[0]+tracks[1]).M()");
```

- Solution for more complex interactive analysis:
  - write your own function in C++ code

Time for Exercises!

Put in practice the concepts to which you were just exposed: read the instructions and solve the exercises on reading and analyzing the Tree

[Exercise: Read a ROOT Tree]

TTReeReader

- New functionality to read TTree in ROOT 6

```
void TreeReaderSimple() {
   TH1F *myHist = new TH1F("h1","ntuple",100,-4,4);

   TFile *myFile = TFile::Open("hsimple.root");
   TTreeReader myReader("ntuple", myFile);

   TTreeReaderValue<Float_t> myPx(myReader, "px");
   TTreeReaderValue<Float_t> myPy(myReader, "py");

   while (myReader.Next()) {
      myHist->Fill(*myPx + *myPy);
   }

   myHist->Draw();
}
```

– bind Tree branches to `TTreeReaderValue` objects
  - type safety by using templated objects
  - possible only with Cling, when JIT compilation is available

How To Read a Tree in C++

- Create a variable pointing to the data (a pointer to data object)

```
Event * myEvent = 0;
```

- Associate a branch with the variable

```
myTree->SetBranchAddress("eBranch",&myEvent);
```

- Read ith-entry in the Tree

```
myTree->GetEntry(i);
```

- now variable points to data object for the i-th event

```
myEvent->GetTracks()->First()->Dump();
==> Dumping object at: 0x0763aad0, name=Track, class=Track
fPx            0.651241    X component of the momentum
fPy            1.02466     Y component of the momentum
fPz            1.2141      Z component of the momentum
[...]
```

How To Read a Tree

- Example macro

```
void ReadTree() {
  TFile f("AFile.root");
  TTree *tree = (TTree*)f->Get("myTree");
  Event *myEvent = 0;
  TBranch* brEvent = 0;
  tree->SetBranchAddress("eBranch", &myEvent, &brEvent);
  Long64_t nent = tree->GetEntries();
  for (Long64_t i = 0; i < nbent; ++i) {
     tree->GetEntry(i);
     //brEvent->GetEntry(i); // to read only the branch
     myEvent->Analyze();
  }
}
```

- Data pointers (e.g. myEvent) MUST be set to 0
- SetBranchAddress requires address of pointers to event object and TBranch (i.e. Event**, TBranch **)

Accessing Tree Branches

- If we are interested in only some branches of a Tree:
  - Use `TTree::SetBranchStatus()` or just `TBranch::GetEntry()` to select the branches to be read
  - by default all branches are read when calling `TTree::GetEntry(event_number)`
  - Speed up considerably the reading phase
  - Example: reading only a branch with an array of muons

```
TClonesArray* myMuons = 0;
// disable all branches
tree->SetBranchStatus("*", 0);
// re-enable the "muon" branches
tree->SetBranchStatus("muon*", 1);
tree->SetBranchAddress("muon", &myMuons);
// now read (access) only the "muon" branches
for (Long64_t i = 0; i < myTree->GetEntries(); ++i) {
     tree->GetEntry(i);
```

Using a Proxy function for Tree queries

- Create a macro `proxy.C`

```
TH1F *h1;

void proxy_Begin(TTree*) {
  h1 = new TH1F("h1","Invariant Mass",100,0,100);
}

double proxy() {
   h1->Fill ( (tracks[0] + tracks[1]).M() );
   return 0;
}

void proxy_Terminate() {
  h1->Draw();
}
```

- use macro in `TTree::Draw`

```
tree->Draw("proxy.C");
```

Using Make Class

```
root[1] tree->MakeClass("MyClass");
```

- will generate a MyClass.h and MyClass.C files with the skeleton code for doing analysis
  - declarations for all tree branches
  - setting the corresponding branch address
- After having filled the functions MyClass::Loop with the needed analysis code, run on the tree data:

```
root[2] .L MyClass.C
root[3] MyClass myclass;
root[4] myclass.Loop();
```

Example MyClass.h

```
class MyClass {
public :
   TTree          *fChain;   //!pointer to the analyzed TTree or TChain
   Int_t           fCurrent; //!current Tree number in a TChain

   // Declaration of leaf types
   vector<ROOT::Math::LorentzVector<ROOT::Math::PxPyPzE4D<double> > > *tracks;

   // List of branches
   TBranch        *b_tracks;   //!

   MyClass(TTree *tree=0);
   virtual ~MyClass();
   virtual Int_t    Cut(Long64_t entry);
   virtual Int_t    GetEntry(Long64_t entry);
   virtual Long64_t LoadTree(Long64_t entry);
   virtual void     Init(TTree *tree);
   virtual void     Loop();
   virtual Bool_t   Notify();
   virtual void     Show(Long64_t entry = -1);
};
```

NOTE: To have correct branch top level definition, branches must be not splitted

Example MyClass.h

This is what you get with split branch

```
class MyClass {
public :
   TTree          *fChain;   //!pointer to the analyzed TTree or TChain
   Int_t           fCurrent; //!current Tree number in a TChain

   // Declaration of leaf types
   Int_t           tracks_;
   Double_t        tracks_fCoordinates_fX[kMaxtracks];   //[tracks_]
   Double_t        tracks_fCoordinates_fY[kMaxtracks];   //[tracks_]
   Double_t        tracks_fCoordinates_fZ[kMaxtracks];   //[tracks_]
   Double_t        tracks_fCoordinates_fT[kMaxtracks];   //[tracks_]

   // List of branches
   TBranch        *b_tracks_;   //!
   TBranch        *b_tracks_fCoordinates_fX;   //!
   TBranch        *b_tracks_fCoordinates_fY;   //!
   TBranch        *b_tracks_fCoordinates_fZ;   //!
   TBranch        *b_tracks_fCoordinates_fT;   //!

   MyClass(TTree *tree=0);
   virtual ~MyClass();
   virtual Int_t    Cut(Long64_t entry);
   virtual Int_t    GetEntry(Long64_t entry);
   virtual Long64_t LoadTree(Long64_t entry);
   virtual void     Init(TTree *tree);
   virtual void     Loop();
   virtual Bool_t   Notify();
   virtual void     Show(Long64_t entry = -1);
};
```

Example MyClass.C

- Fill in Loop() the user code for analysis
  - e.g. plot invariant mass of tracks

```
void MyClass::Loop()
{
  Long64_t nentries = fChain->GetEntriesFast();

   TH1D * h1 = new TH1D("h1","Invariant Mass of all tracks", 100, 0,100);

   Long64_t nbytes = 0, nb = 0;
   for (Long64_t jentry=0; jentry<nentries;jentry++) {
      Long64_t ientry = LoadTree(jentry);
      if (ientry < 0) break;
      //fChain->GetEntry(jentry);
      b_tracks->GetEntry(jentry);    // faster to read only the branch
      // if (Cut(ientry) < 0) continue;

      for (unsigned int i = 0; i < (*tracks).size() ; ++i)
         for (unsigned int j = i+1; j < (*tracks).size() ; ++j)
            h1->Fill( ( (*tracks)[i]+(*tracks)[j] ).M() );
   }
   h1->Draw();
}
```

Time for Exercises!

Put in practice the concepts to which you were just exposed: read the instructions and solve the exercises on reading and analyzing the Tree

Exercise: Analyze a ROOT Tree

TSelector

- Another way to analyze Tree is using the TSelector class
  - the user creates a new class `MySelector` deriving from **`TSelector`**

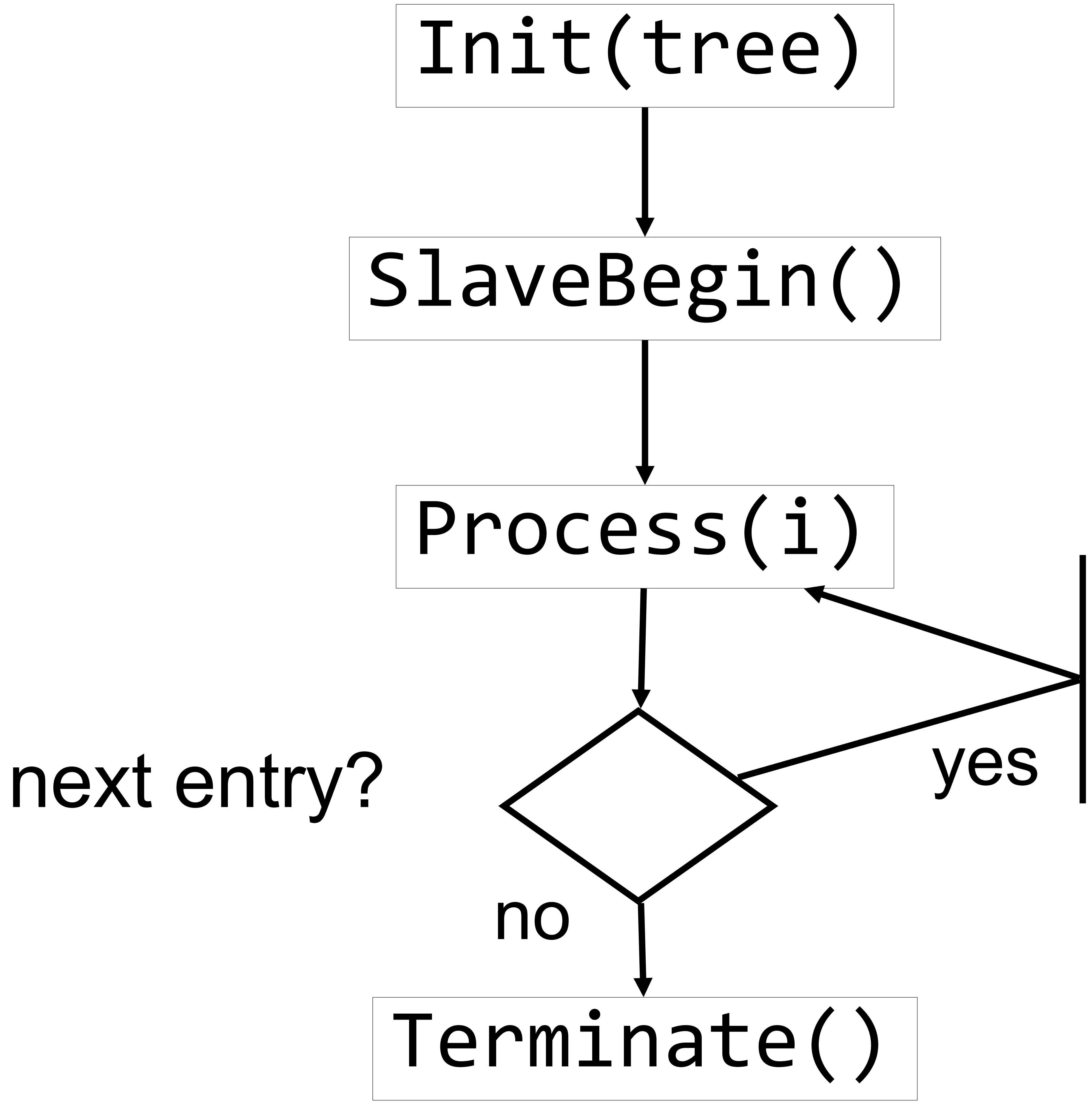
```
root[1] tree->MakeSelector("MySelector");
```

    - generates file `MySelector.h` and `MySelector.C`
  - the `MySelector` object is used in `TTree::Process(TSelector*,...)`
  - ROOT invokes the `TSelector's` functions which are virtuals, so the user provided function implemented in `MySelector` will be called.

Tree Data Access

E.g.

```
tree->Process("MySelector.C+")
```



TSelector

Steps of ROOT using a `TSelector`:

1. ***setup*** **`TMySelector::Init(TTree *tree)`**
   `fChain = tree; fChain->SetBranchAddress()`
   initialize branches

2. ***start*** **`TMySelector::SlaveBegin()`**
   create histograms

3. ***run*** **`TMySelector::Process(Long64_t)`**
   `fChain->GetTree()->GetEntry(entry);`
   analyze data, fill histograms,…

4. ***end*** **`TMySelector::Terminate()`**
   fit histograms, write them to files,…

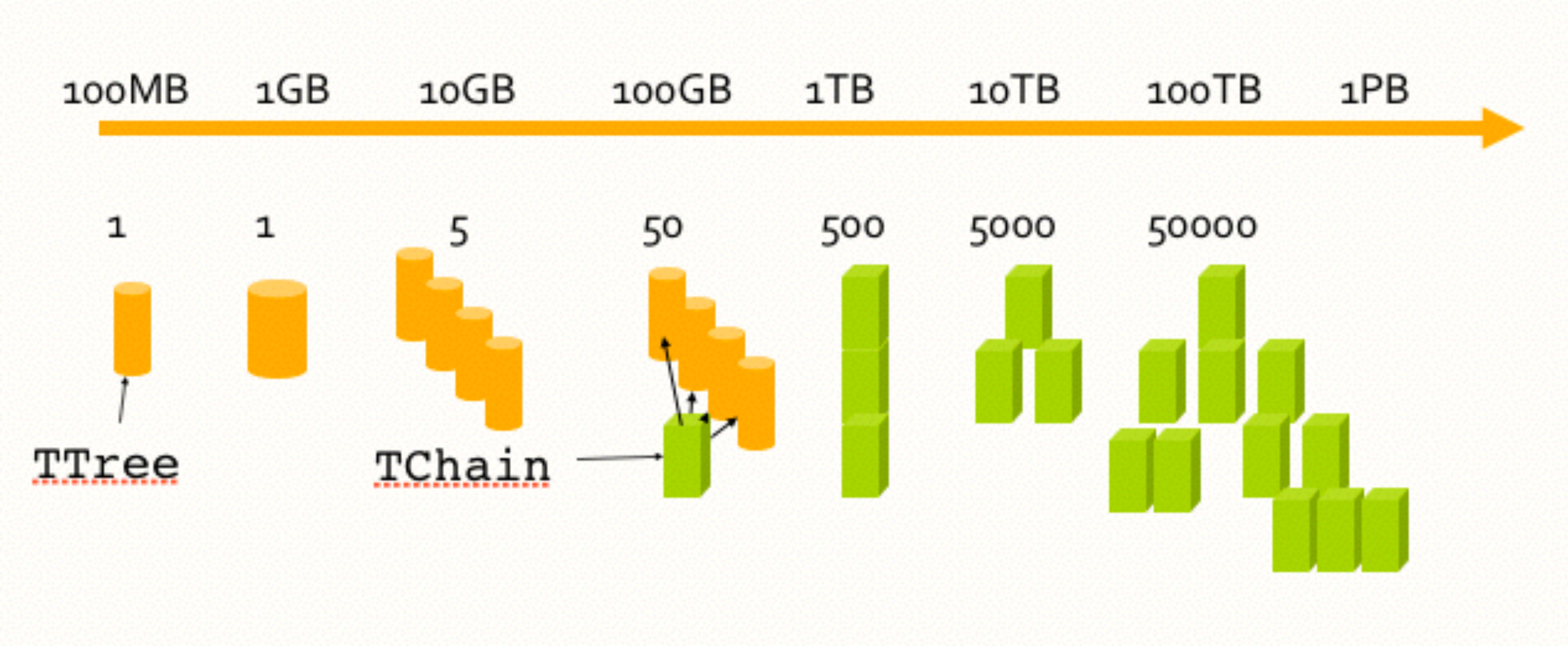
Time for Exercises!

Put in practice the concepts to which you were just exposed: read the instructions and solve the exercises on reading and analyzing the Tree

Exercise: Analyze a ROOT Tree using a TSelector

Data Volume and Organization

- A `TFile` typically contains 1 `TTree`
- A `TChain` is a collection of `TTrees` or/and `TChains`



TChain

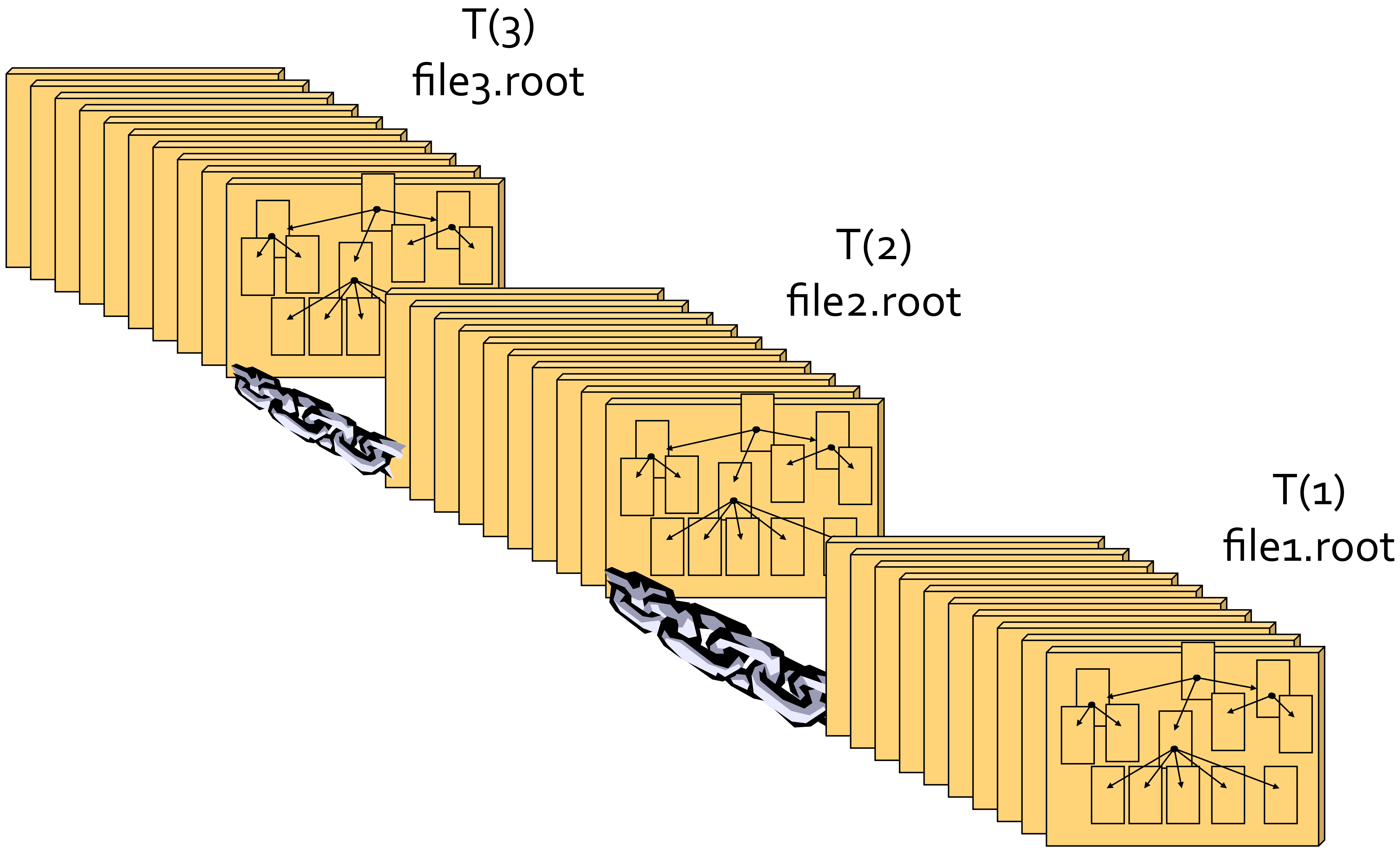
- Collection of ROOT files containing the same tree
- Same semantics as `TTree`.
  - As an example, assume we have three files called file1.root, file2.root, file3.root. Each contains tree called "T". Create a chain:

```
TChain chain("T"); // argument: tree name
chain.Add("file1.root");
chain.Add("file2.root");
chain.Add("file3.root");
```

  - Now we can use the `TChain` like a `TTree`!

TChain

- Chain Files together



Time for Exercises!

Put in practice the concepts to which you were just exposed: read the instructions and solve the exercises on creating a TChain

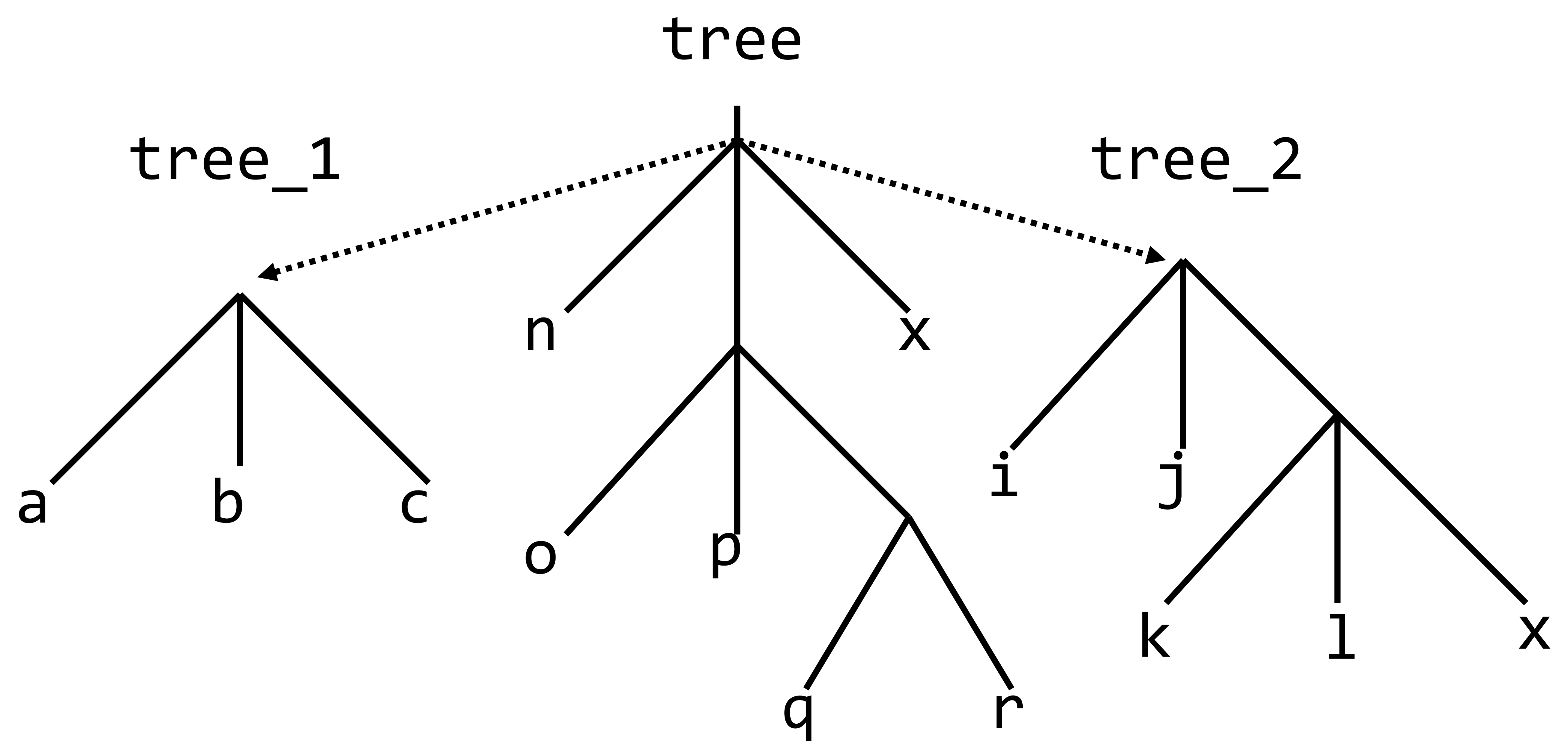
Exercise: Chaining ROOT Files

Tree Friends

- Trees are designed to be read only
- Often, people want to add branches to existing trees and write their data into it
- Using tree friends is the solution:
  - Create a new file holding the new tree
  - Create a new Tree holding the branches for the user data
  - Fill the tree/branches with user data
  - Add this new file/tree as friend of the original tree

Tree Friends

- Using Tree Friends




```
TFile f1("tree.root");
tree.AddFriend("tree_1", "tree1.root")
tree.AddFriend("tree_2", "tree2.root");
tree.Draw("x:a", "k<c");
tree.Draw("x:tree_2.x");
```

Performance Considerations

A split branch is:

- Faster to read – if you only want a subset of data members
- Slower to write due to the large number of branches

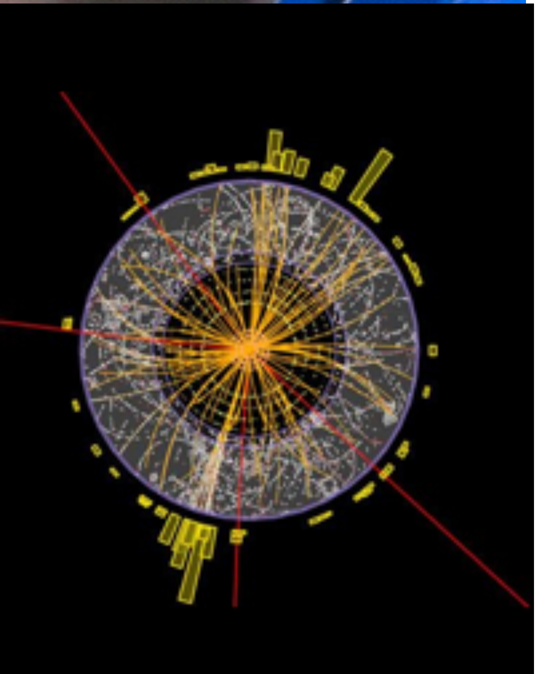
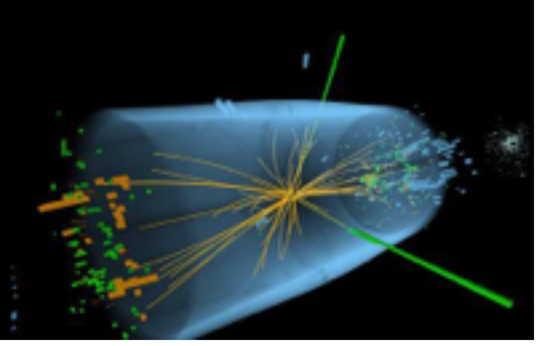
- For reading a subset of data recommend to use
  - `branch->GetEntry(ientry)`
  - will read only the required branch data (big difference in case of trees with many branches)
- Alternatively can use also
  - `tree->SetBranchStatus("*", 0);`
  - `tree->SetBranchStatus("myBranch",1);`

Analyzing Trees: Summary

- Tree is an efficient storage and access for huge amounts of structured data
- Allows selective access of data
- It is used to analyze and select data.
- A convenient way to analyze data store in a Tree is with the **`TSelector`** class
  - the user creates a new class `MySelector` deriving from **`TSelector`**
  - the `MySelector` object is used in `TTree::Process(TSelector*,...)`
  - ROOT invokes the `TSelector's` functions which are virtuals, so the user provided function implemented in `MySelector` will be called.

Summary

- The ROOT Tree is one of the most powerful collections available for HEP
- Extremely efficient for huge number of data sets with identical layout
- Very easy to look at `TTree` - use `TBrowser`!
- Write once, read many: ideal for experiments' data; use friends to extend
- Branches allow granular access; use splitting to create branch for each member, even through collections
- `TSelector` class provides a powerful way of processing the Tree data using compiled code

Interactive Data Analysis with PROOF

Bleeding Edge Physics
with Bleeding Edge Computing

Dr Lorenzo Moneta
CERN PH-SFT
CH-1211 Geneva 23
sftweb.cern.ch
root.cern.ch

Parallel Analysis: PROOF

Some numbers (from Alice experiment)

- 1.5 PB ($1.5 * 10^{15}$) of raw data per year
- 360 TB of ESD+AOD* per year (20% of raw)
- One pass at 15 MB/s will take 9 months!

Parallelism is the only way out!

* ESD: Event Summary Data   AOD: Analysis Object Data

PROOF

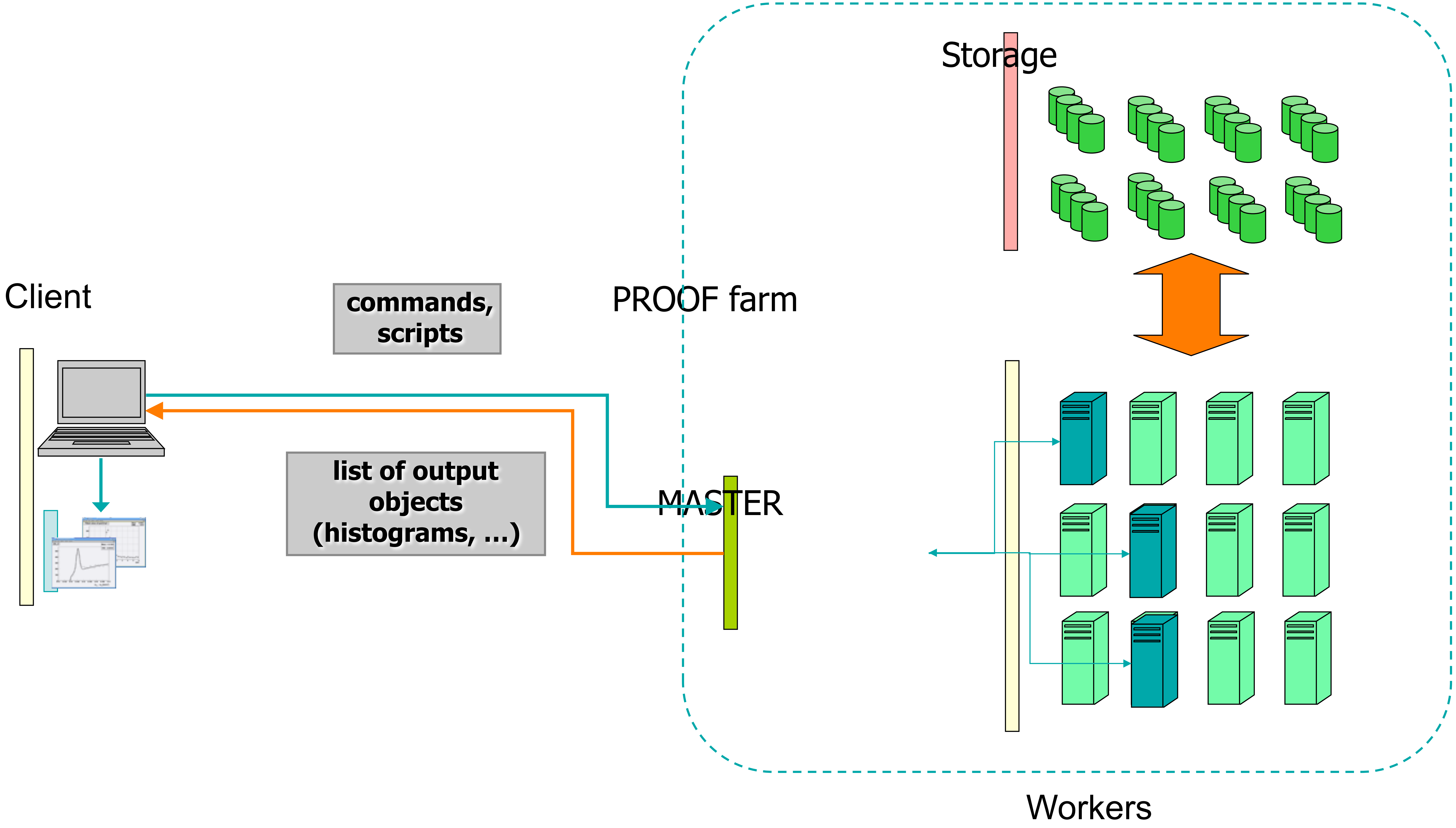
Huge amounts of events, hundreds of CPUs

Split the job into N events / CPU!

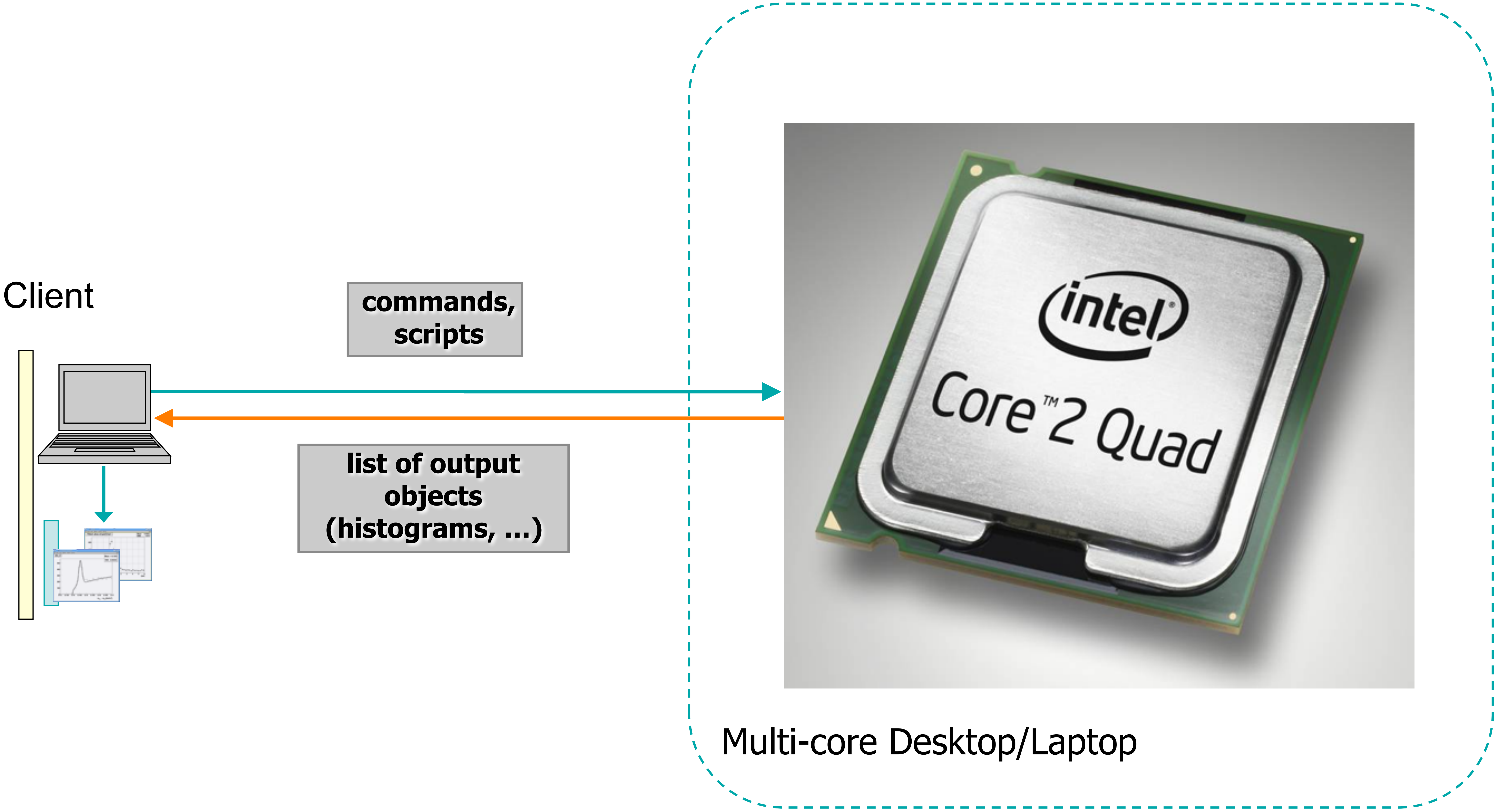
PROOF for TSelector based analysis:

- start analysis locally ("client"),
- PROOF distributes data and code,
- lets CPUs ("workers") run the analysis,
- collects and combines (merges) data,
- shows analysis results locally

PROOF


Storage
Client
commands, scripts
PROOF farm
list of output objects (histograms, …)
MASTER
Workers

PROOF Lite


Client
commands, scripts
list of output objects (histograms, …)
intel
Core™2 Quad
Multi-core Desktop/Laptop

What is PROOF Lite?

- PROOF optimized for single many-core machines
- Zero configuration setup
  - No config files and no daemons
- Like PROOF it can exploit fast disks, SSD's, lots of RAM, fast networks and fast CPU's
- If your code works on PROOF, then it works on PROOF Lite and vice versa

Creating a session

To create a PROOF Lite session from the ROOT prompt, just type:

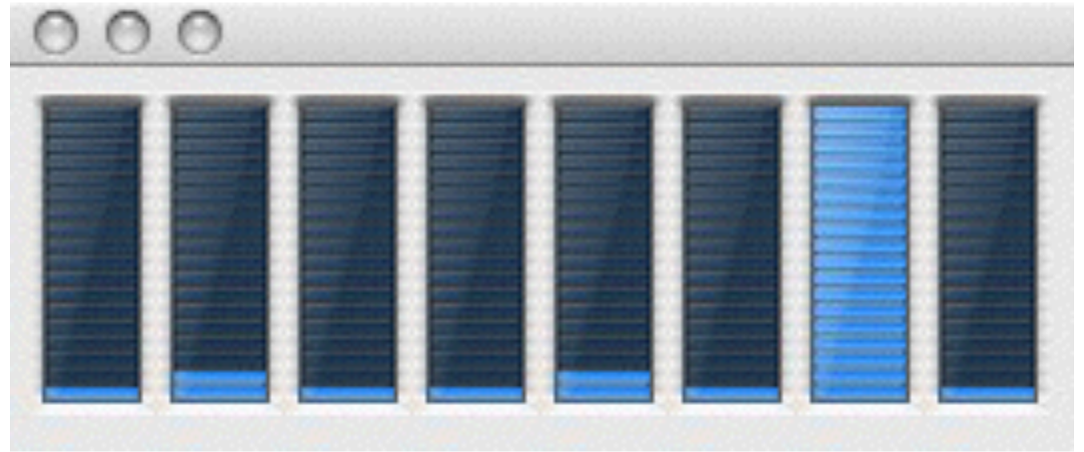
```
TProof::Open("")
```

Then you can use your multicore computer as a PROOF cluster!

PROOF Analysis

- Example of local TChain analysis

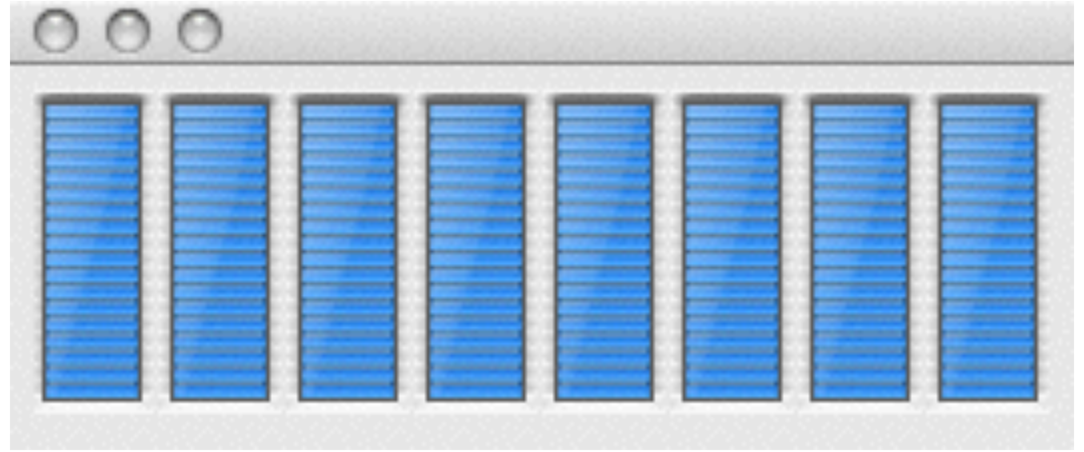
```
// Create a chain of trees
root[0] TChain *c = new TChain("myTree");
root[1] c->Add("http://www.any.where/file1.root");
root[2] c->Add("http://www.any.where/file2.root");

// MySelector is a TSelector
root[3] c->Process("MySelector.C+");
```



PROOF Analysis

- Same example with PROOF

```
// Create a chain of trees
root[0] TChain *c = new TChain("myTree");
root[1] c->Add("http://www.any.where/file1.root");
root[2] c->Add("http://www.any.where/file2.root");

// Start PROOF and tell the chain to use it
root[3] TProof::Open("");
root[4] c->SetProof();

// Process goes via PROOF
root[5] c->Process("MySelector.C+");
```



TSelector & PROOF

- Begin() called on the client only
- SlaveBegin() called on each worker: create histograms
- SlaveTerminate() rarely used; post processing of partial results before they are sent to master and merged
- Terminate() runs on the client: save results, display histograms, …

Output List (result of the query)

- Each worker has a partial output list
- Objects have to be added to the list in TSelector::SlaveBegin() e.g.:

```
fHist = new TH1F("h1", "h1", 100, -3., 3.);
fOutput->Add(fHist);
```

- At the end of processing the output list gets sent to the master
- The Master merges objects and returns them to the client. Merging is e.g. "Add()" for histograms, appending for lists and trees

Example

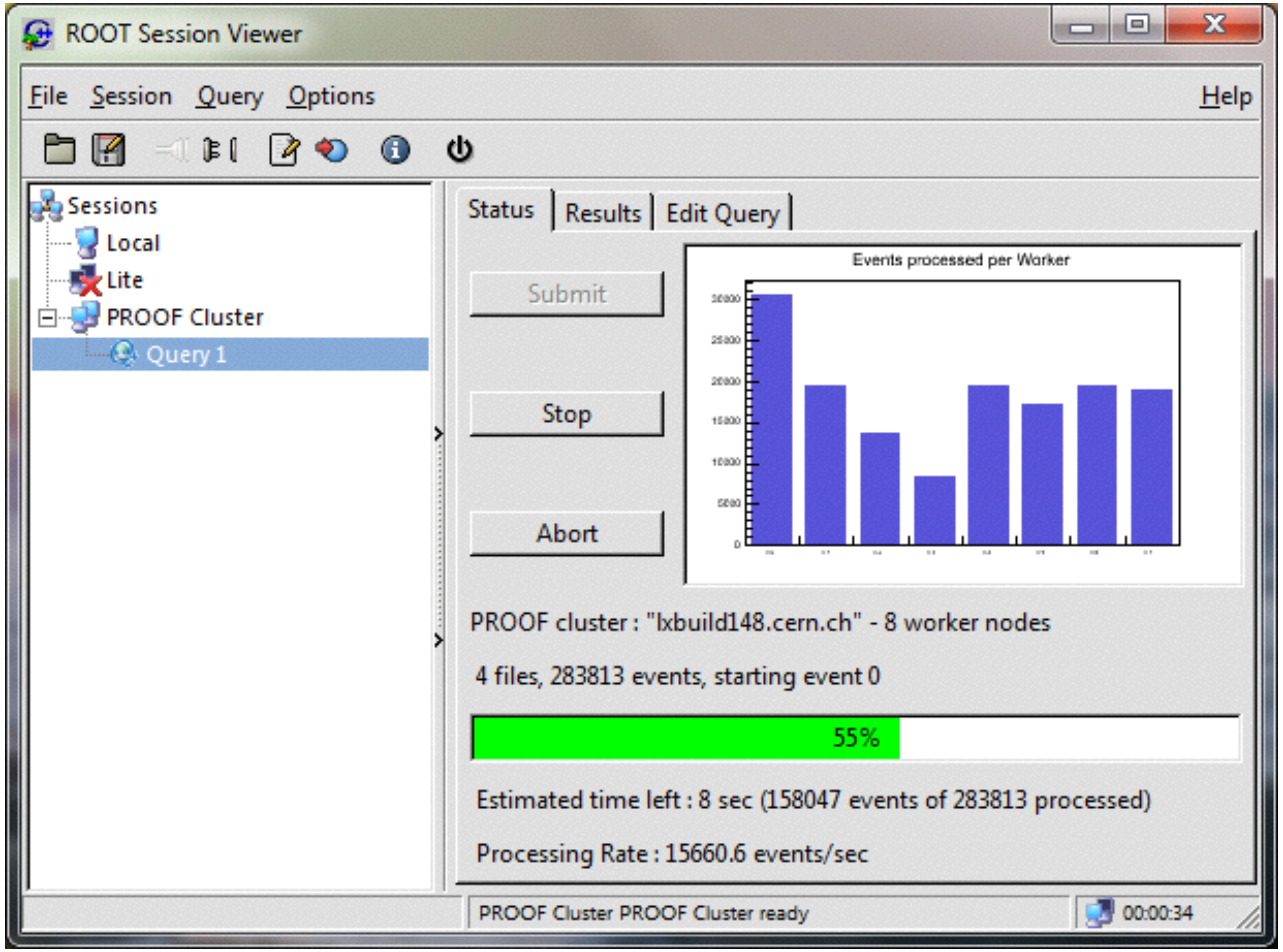
```
void MySelector::SlaveBegin(TTree *tree) {
   // create histogram and add it to the output list
   fHist = new TH1F("MyHist","MyHist",40,0.13,0.17);
   GetOutputList()->Add(fHist);
}

Bool_t MySelector::Process(Long64_t entry) {
   my_branch->GetEntry(entry); // read branch
   fHist->Fill(my_data);       // fill the histogram
   return kTRUE;
}

void MySelector::Terminate() {
   fHist->Draw();              // display histogram
}
```

PROOF GUI Session

Starting a PROOF GUI session is trivial:

```
TProof::Open()
```

Opens GUI:



Summary

- We are at the end
  - still to complete the exercise on PROOF
- We have learned about ROOT
  - general overview of ROOT
  - some basics of ROOT I/O
  - tree and their use for data analysis
  - PROOF for parallel analysis of Trees

- Next we will cover fitting, advanced statistical analysis with RooFit/ RooStats and machine learning with TMVA

Time for Exercises!

Put in practice the concepts to which you were just exposed: read the instructions and solve the exercises on data analysis using PROOF

[Exercise: Use PROOF to analyze a TTree]