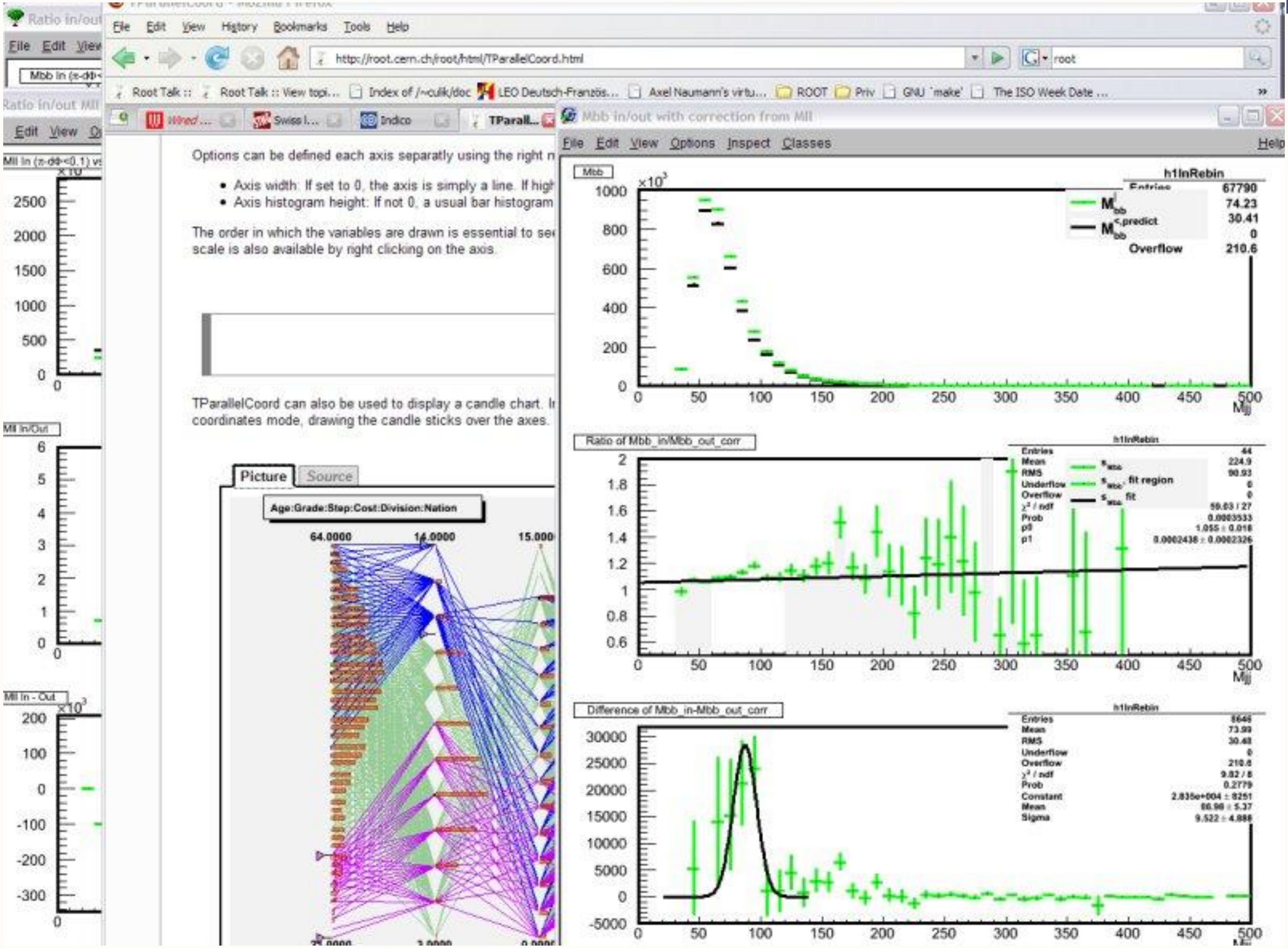
ROOT

Bertrand Bellenot, Axel Naumann
CERN

WHAT IS ROOT?

What's ROOT?



File Edit View History Bookmarks Tools Help
http://root.cern.ch/root/html/TParallelCoord.html
Root Talk :: Root Talk :: View topi... Index of /~culk/doc LEO Deutsch-Französ... Axel Naumann's virtu... ROOT Priv GNU `make' The ISO Week Date ...
Wired ... Swiss I... Indico TParall...
Options can be defined each axis separatly using the right m
Axis width: If set to 0, the axis is simply a line. If high
Axis histogram height: If not 0, a usual bar histogram
The order in which the variables are drawn is essential to see
scale is also available by right clicking on the axis.
TParallelCoord can also be used to display a candle chart. In
coordinates mode, drawing the candle sticks over the axes.
Picture
Source
Age:Grade:Step:Cost:Division:Nation
Mbb in/out with correction from Mll
File Edit View Options Inspect Classes Help
Mbb
h1InRebin
Overflow 210.6
Ratio of Mbb_in/Mbb_out_corr
Difference of Mbb_in-Mbb_out_corr
Mjj

ROOT: An Open Source Project

- Started in 1995
- 7 full time developers at CERN, plus Fermilab
- Large number of part-time developers: let users participate
- Available (incl. source) under GNU LGPL

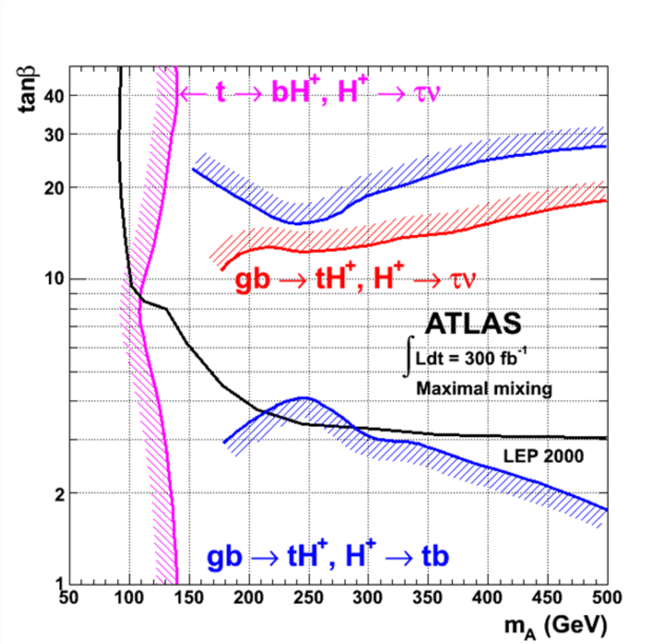
ROOT in a Nutshell

Framework for large scale data handling

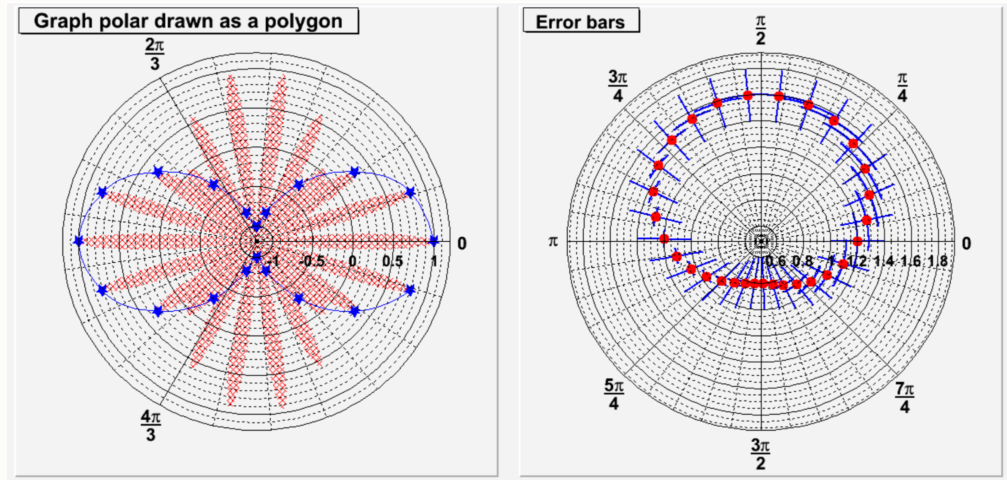
Provides, among others,

- an efficient data storage, access and query system (PetaBytes)
- advanced statistical analysis algorithms (multi dimensional histogramming, fitting, minimization and cluster finding)
- scientific visualization: 2D and 3D graphics, Postscript, PDF, LateX
- geometrical modeller
- PROOF parallel query engine

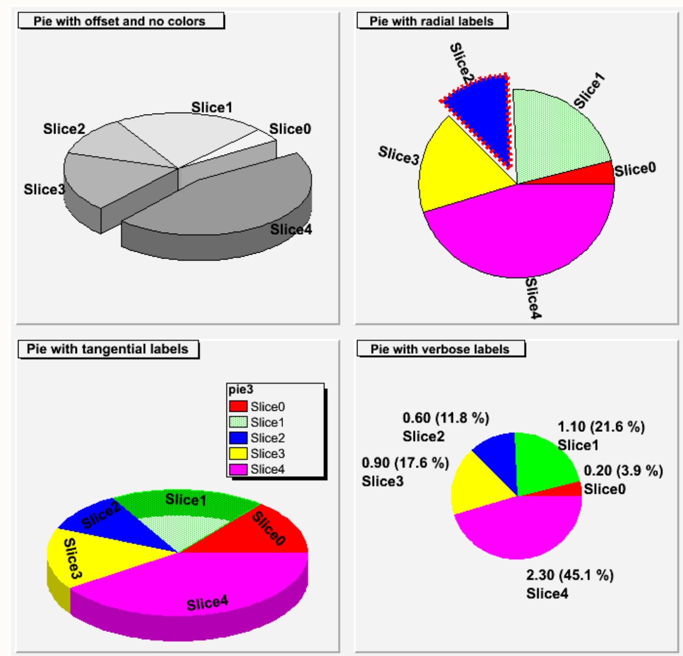
Graphics



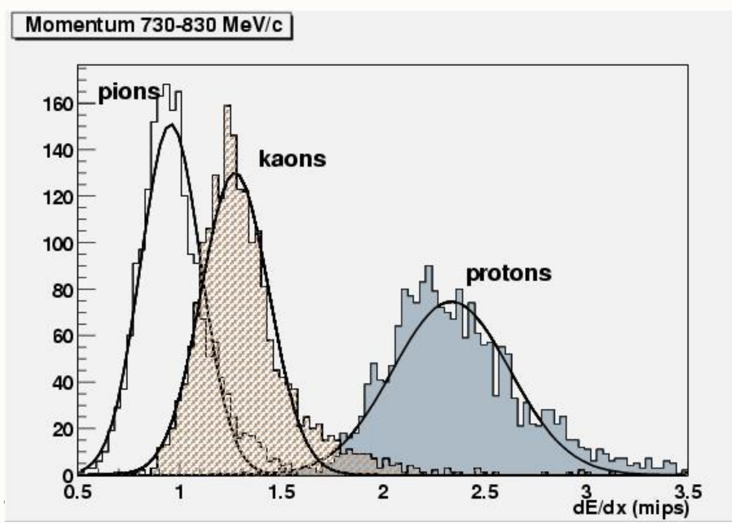
tanβ
t → bH+, H+ → τν
gb → tH+, H+ → τν
ATLAS
∫Ldt = 300 fb-1
Maximal mixing
LEP 2000
gb → tH+, H+ → tb
mA (GeV)



Graph polar drawn as a polygon
Error bars



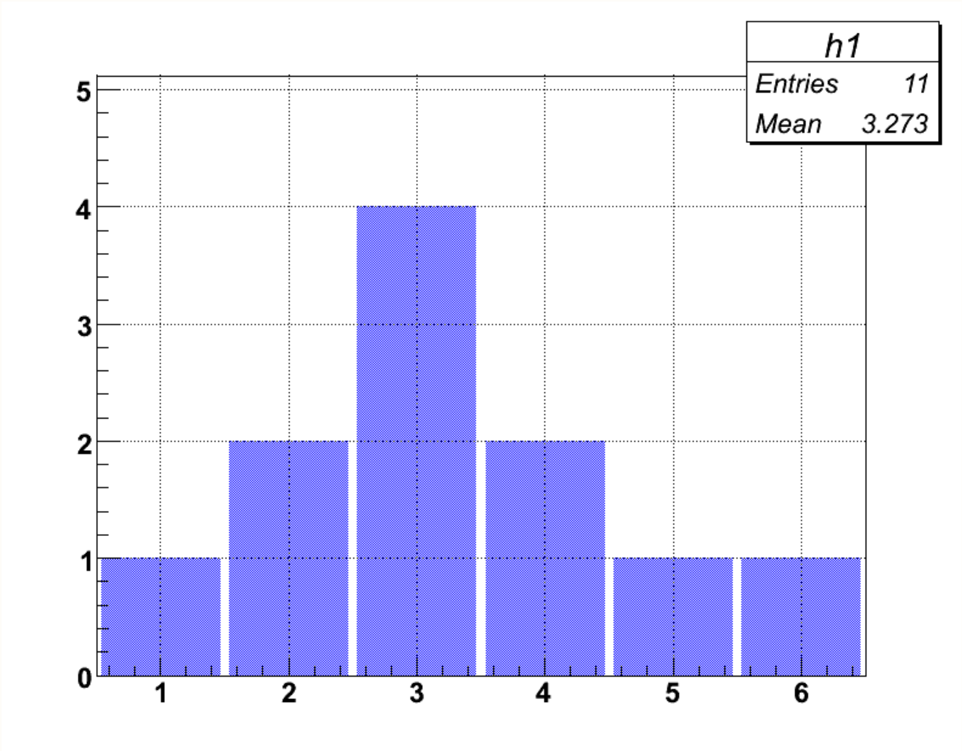
Pie with offset and no colors
Pie with radial labels
Pie with tangential labels
Pie with verbose labels
Slice0
Slice1
Slice2
Slice3
Slice4
pie3
0.60 (11.8 %)
1.10 (21.6 %)
0.90 (17.6 %)
0.20 (3.9 %)
2.30 (45.1 %)



Momentum 730-830 MeV/c
pions
kaons
protons
dE/dx (mips)

Histogramming

- Histogram is just occurrence counting, i.e. how often they appear
- Example: {1,3,2,6,2,3,4,3,4,3,5}



Histogramming

- **How is a Real Histogram Made?**
  Lets consider the age distribution of the CSC participants in 2008:

Ages
26.93
23.03
29.66
33.74
32.19
31.40
24.88
27.56
26.99
26.65
[illegible]

Age	Number
20-22	1
22-24	5
24-26	14
26-28	14
28-30	10
30-32	5
32-34	5
34-36	1
36-38	1
38-40	1

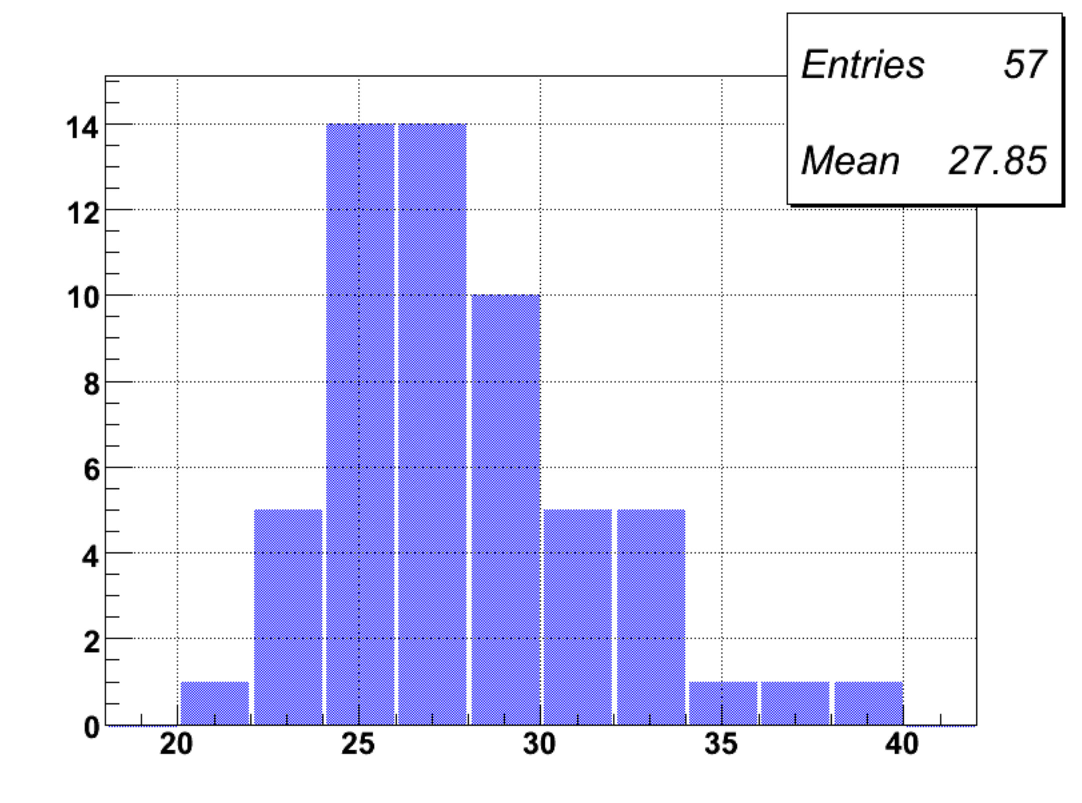
Binning:

Grouping ages of participants in several categories (bins)

Histogramming

Table of Ages (binned)

Age	Number
20-22	1
22-24	5
24-26	14
26-28	14
28-30	10
30-32	5
32-34	5
34-36	1
36-38	1
38-40	1



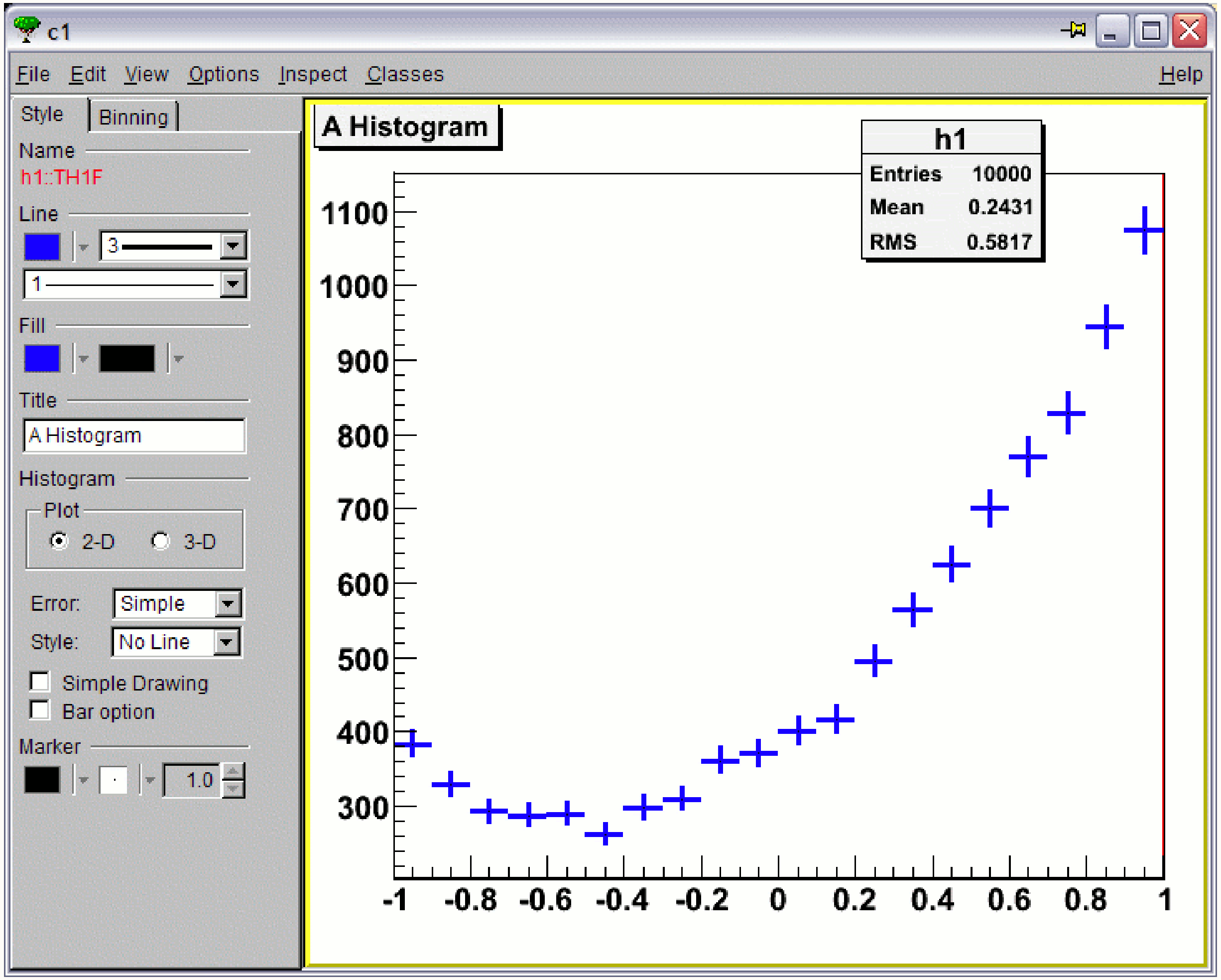
Shows distribution of ages, total number of entries (57 participants) and average: 27 years 10 months 6 days…

Histograms

Analysis result: often a histogram

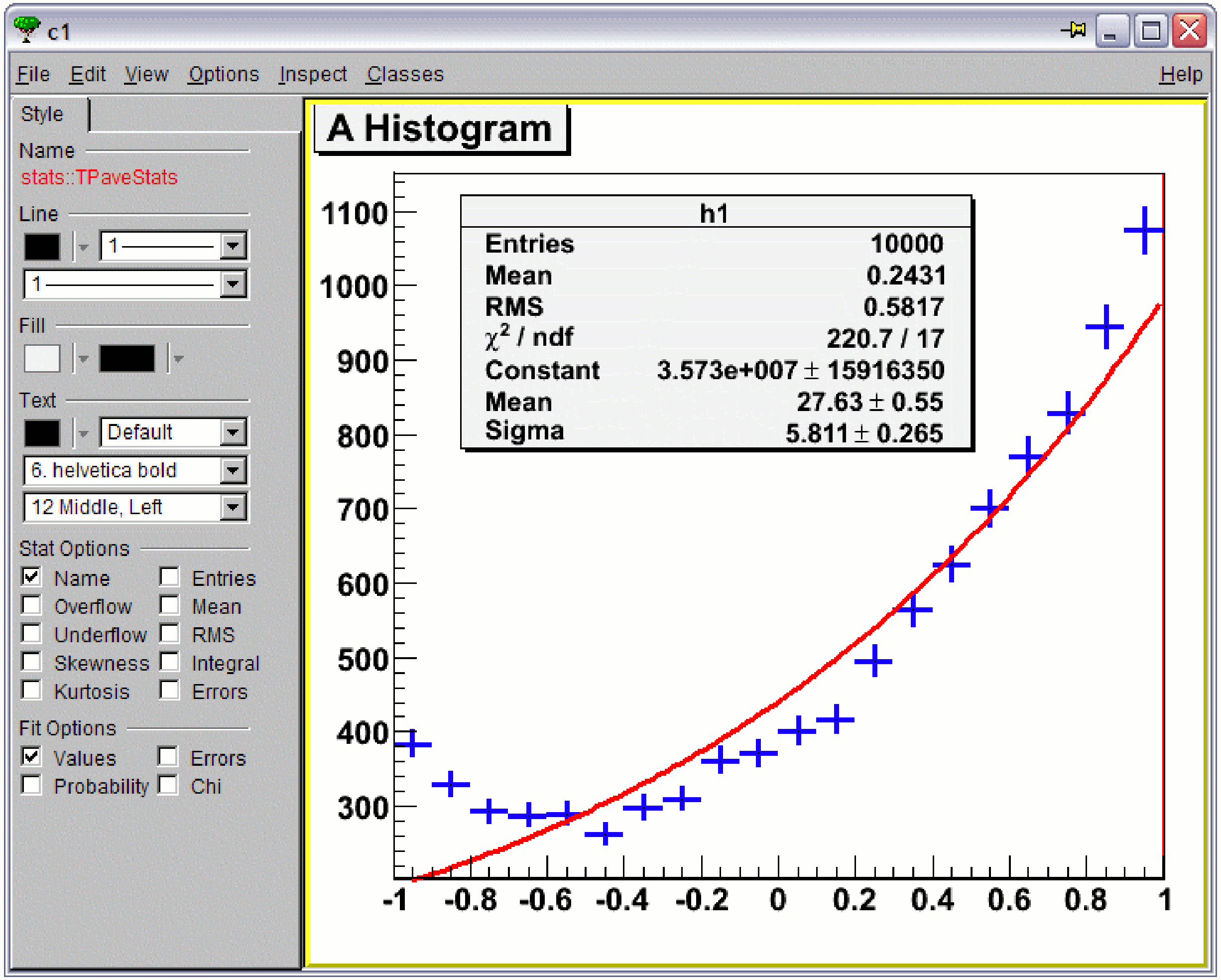
Menu:

View / Editor



Fitting

Analysis result: often a ***fit*** of a histogram



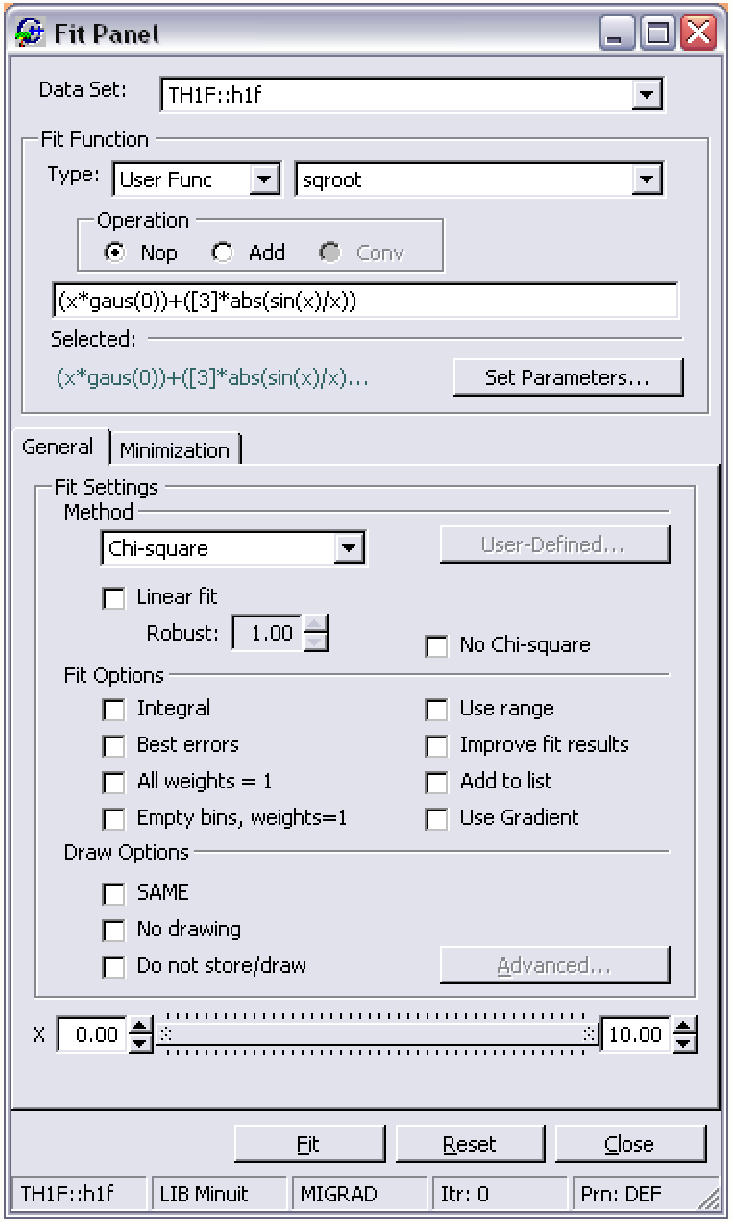
Fit Panel

To fit a histogram:

right click histogram,

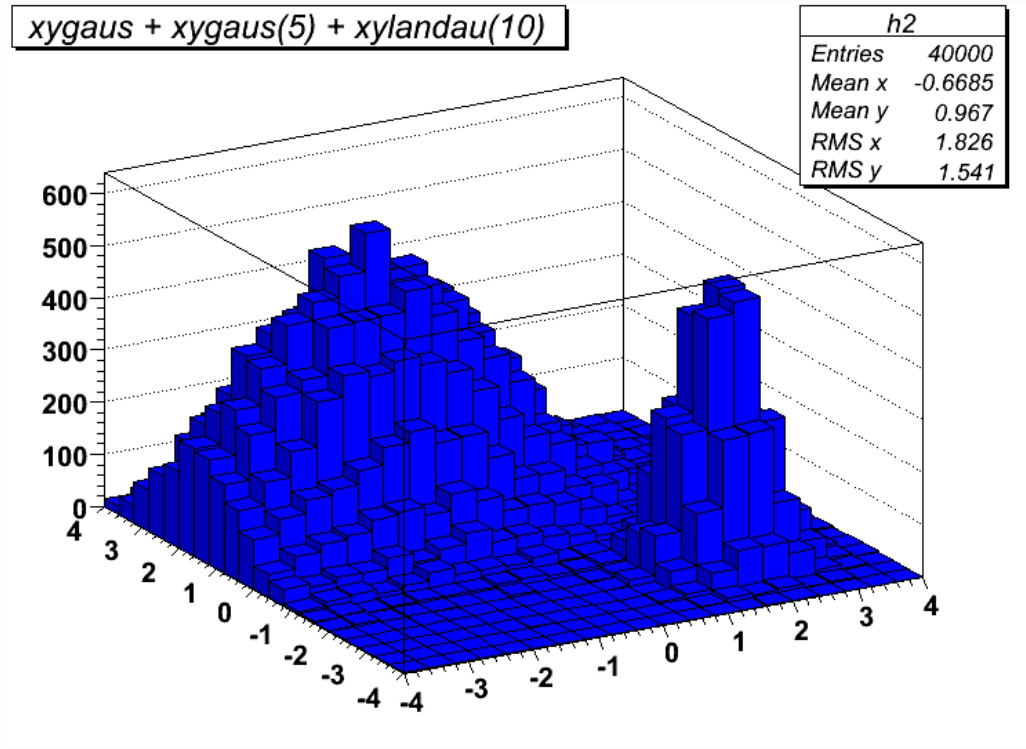
"Fit Panel"

Straightforward interface for fitting!

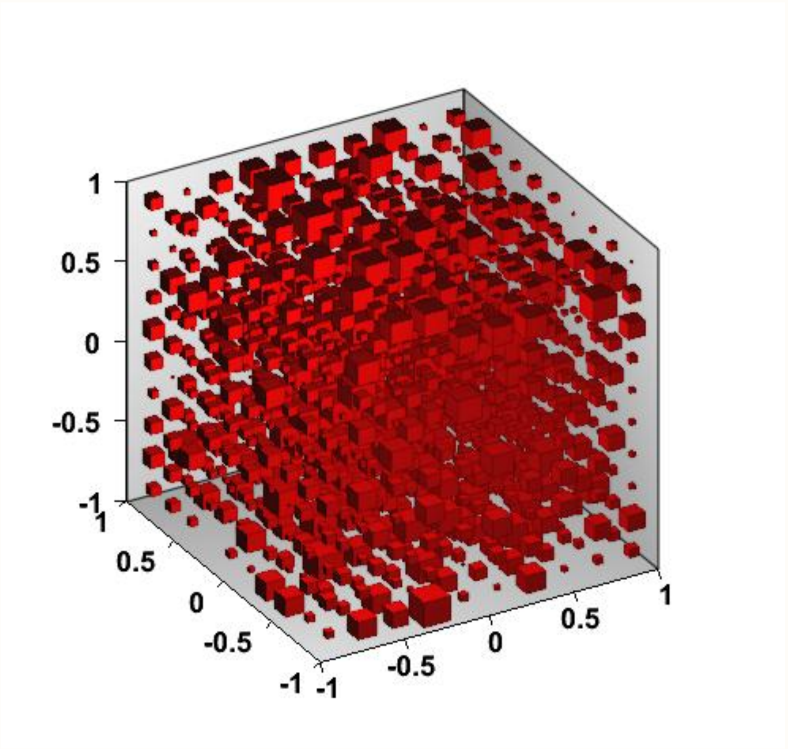


2D/3D

We have seen 1D histograms, but there are also histograms in more dimensions.



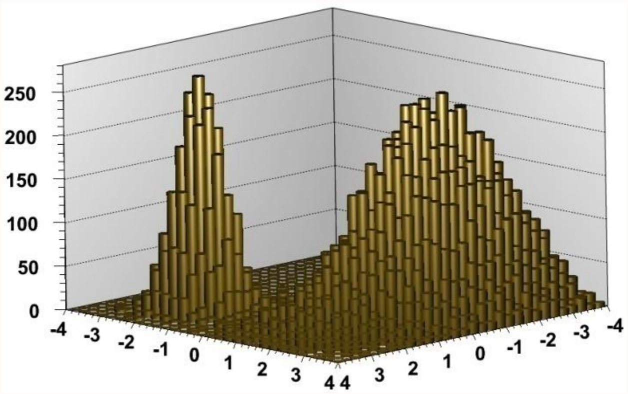
2D Histogram

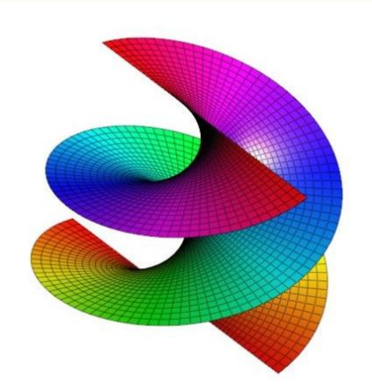


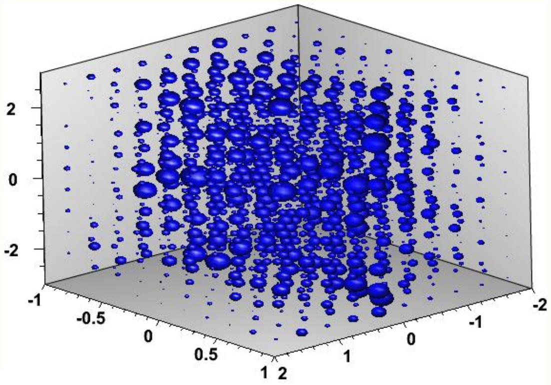
3D Histogram

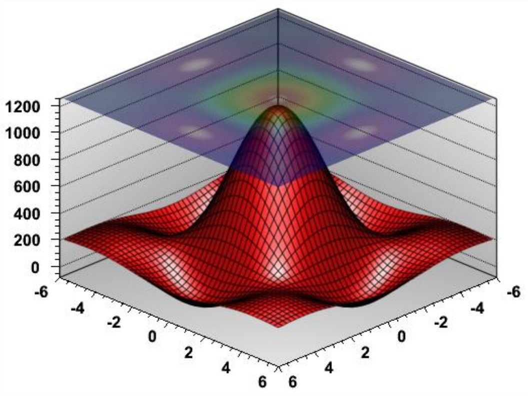
OpenGL

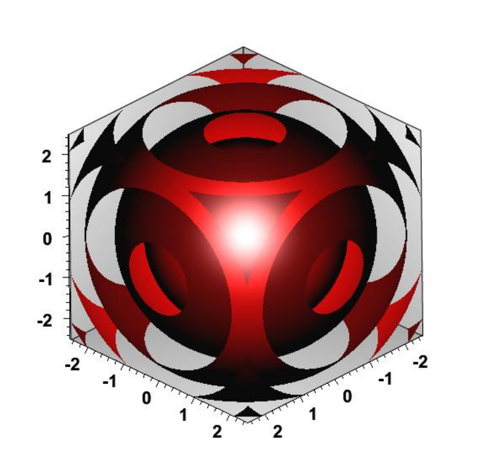
OpenGL can be used to render 2D & 3D histograms, functions, parametric equations, and to visualize 3D objects (geometry)







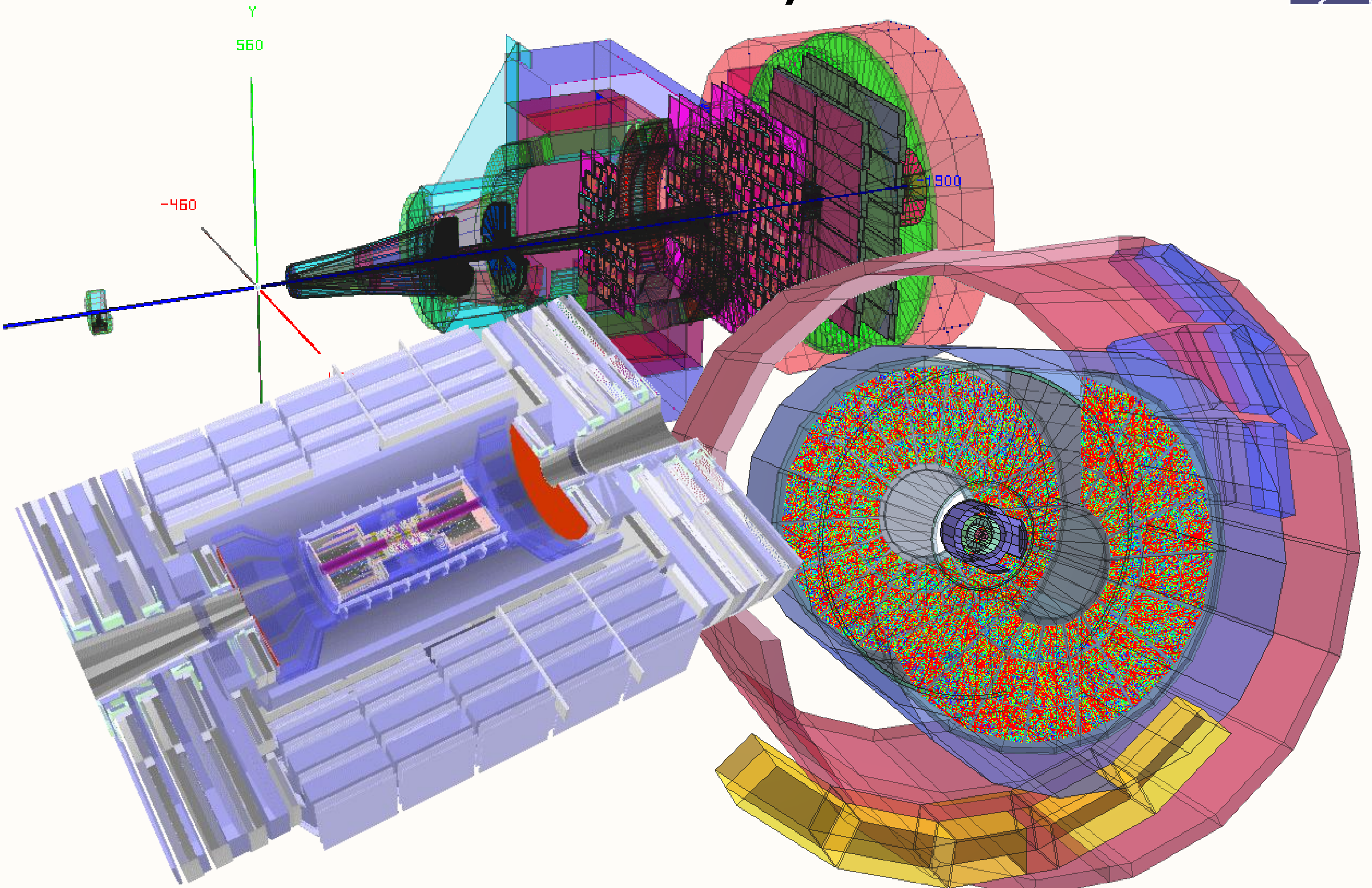




Geometry

- Describes complex detector geometries
- Allows visualization of these detector geometries with e.g. OpenGL
- Optimized particle transport in complex geometries
- Working in correlation with simulation packages such as GEANT3, GEANT4 and FLUKA

Geometry



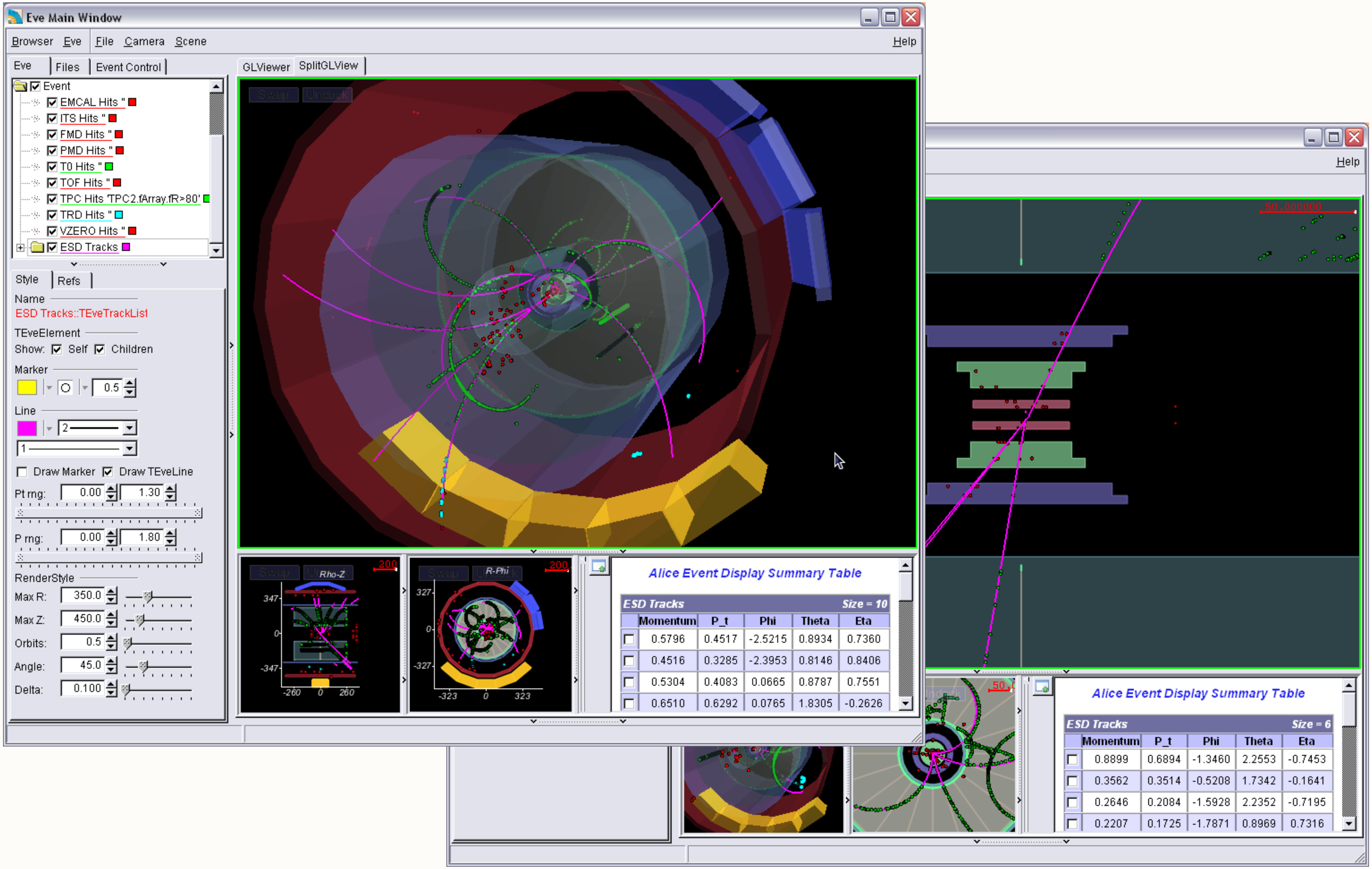
EVE (Event Visualization Environment)

- Event: Collection of data from a detector (hits, tracks, ...)

Use EVE to:

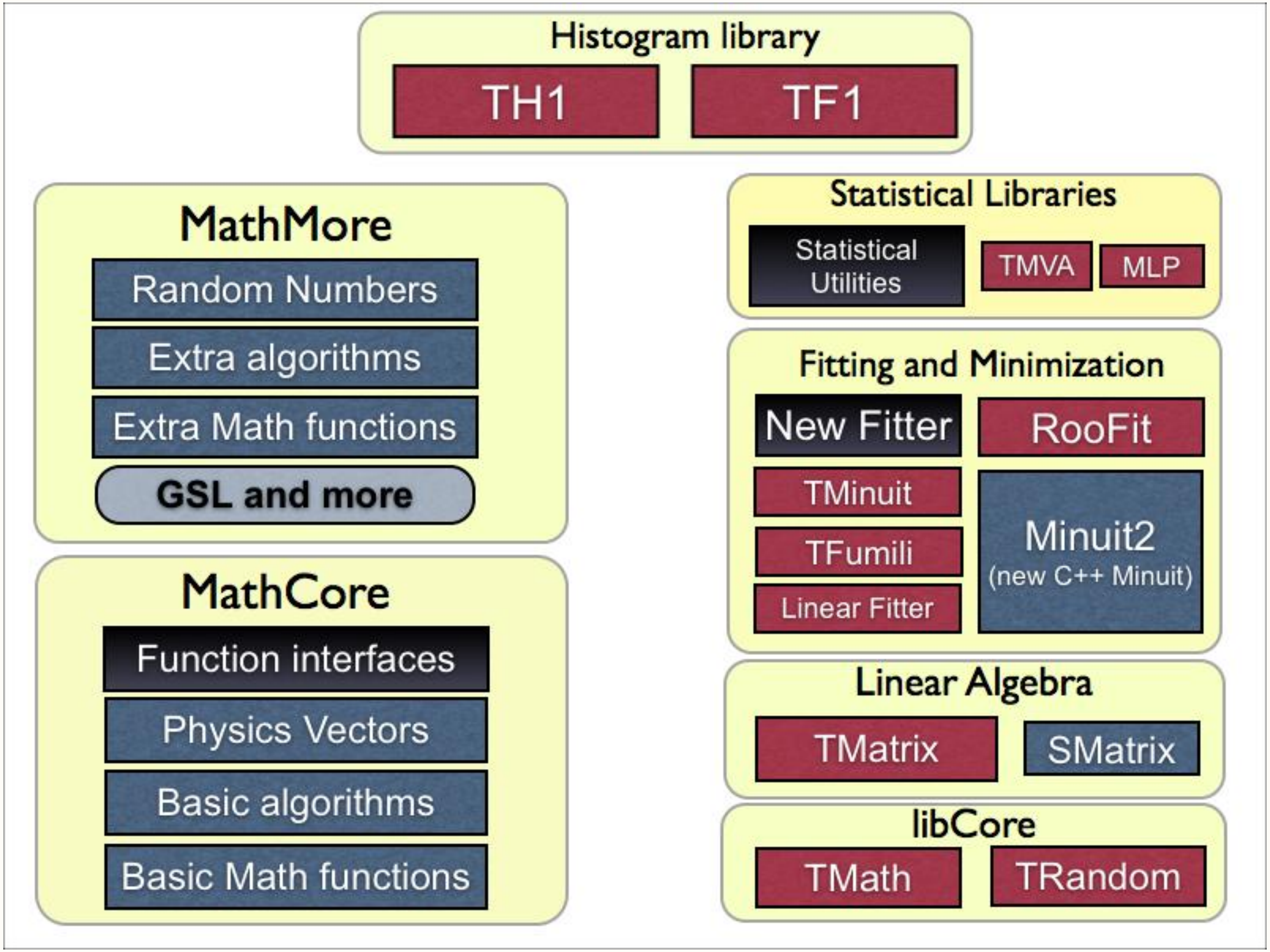
- Visualize these physics objects together with detector geometry (OpenGL)
- Visually interact with the data, e.g. select a particular track and retrieve its physical properties

EVE



Eve Main Window
Browser Eve File Camera Scene Help
Eve Files Event Control
GLViewer SplitGLView
Event
EMCAL Hits "
ITS Hits "
FMD Hits "
PMD Hits "
T0 Hits "
TOF Hits "
TPC Hits 'TPC2.fArray.fR>80'
TRD Hits "
VZERO Hits "
ESD Tracks
Style Refs
Name
ESD Tracks::TEveTrackList
TEveElement
Show: Self Children
Marker
0.5
Line
2
1
Draw Marker Draw TEveLine
Pt rng: 0.00 1.30
P rng: 0.00 1.80
RenderStyle
Max R: 350.0
Max Z: 450.0
Orbits: 0.5
Angle: 45.0
Delta: 0.100
Rho-Z
R-Phi
Alice Event Display Summary Table
ESD Tracks Size = 10
Momentum P_t Phi Theta Eta
0.5796 0.4517 -2.5215 0.8934 0.7360
0.4516 0.3285 -2.3953 0.8146 0.8406
0.5304 0.4083 0.0665 0.8787 0.7551
0.6510 0.6292 0.0765 1.8305 -0.2626
Help
Alice Event Display Summary Table
ESD Tracks Size = 6
Momentum P_t Phi Theta Eta
0.8899 0.6894 -1.3460 2.2553 -0.7453
0.3562 0.3514 -0.5208 1.7342 -0.1641
0.2646 0.2084 -1.5928 2.2352 -0.7195
0.2207 0.1725 -1.7871 0.8969 0.7316

Math


Histogram library
TH1
TF1
MathMore
Random Numbers
Extra algorithms
Extra Math functions
GSL and more
MathCore
Function interfaces
Physics Vectors
Basic algorithms
Basic Math functions
Statistical Libraries
Statistical Utilities
TMVA
MLP
Fitting and Minimization
New Fitter
RooFit
TMinuit
TFumili
Linear Fitter
Minuit2
(new C++ Minuit)
Linear Algebra
TMatrix
SMatrix
libCore
TMath
TRandom

Math Example:

RANDOM NUMBERS

Quasi-Random Numbers

- Needed e.g. to simulate nature: will particle interact?
- Trivial example for random number generator function:
  last digit of n = n + 7,
  say start with 0:
  0, 7, 4, 1, 8, 5, 2, 9, 6, 3, 0, 7, 4, 1, 8, 5, 2, 9, 6, 3, 0, 7, 4, 1, 8,
- Properties:
  - identical frequency of all numbers 0..9
  - "looks" random, but short period:
    0, 7, 4, 1, 8, 5, 2, 9, 6, 3, 0, 7, 4, 1, 8, 5, 2, 9, 6, 3, 0, 7, 4, 1
  - numbers not independent!

Random Number Generator

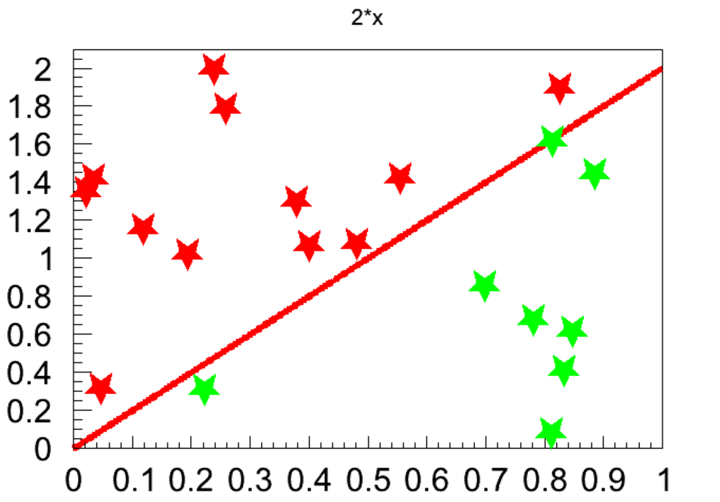
- Solution: more complex function
  - Mersenne Twister (TRandom3) is recommended

```
TRandom3 myRnd; myRnd.Uniform();
```

    generates random number between >0 and <= 1
  - period 10^{6000}, fast!
- Flat probability distribution function good for dice, usually not for physics:
  - measurement uncertainty: gaussian
  - particle lifetime: $N(t) = N_0 \exp(-t/\tau)$ i.e. exponential
  - energy loss of particles in matter: landau

Naïve Random Distribution

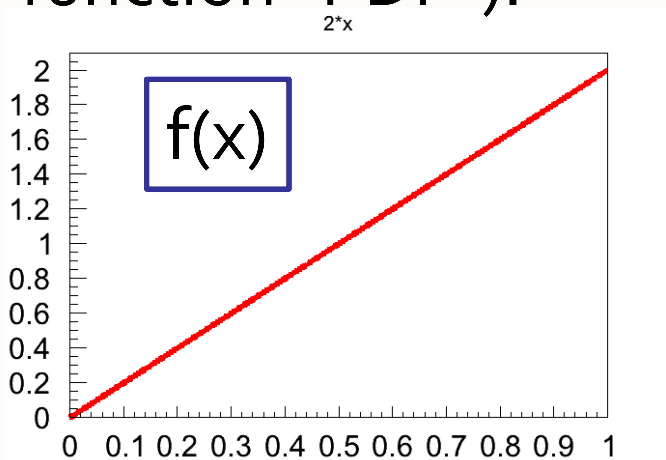
- Want to "sample" distribution
  y = 2x for 0 < x < 1
- Acceptance-rejection method
- Generate random x^* and y^* point: if $y^* <= 1 - x^2$, return as random number, else generate new x^*, y^*.

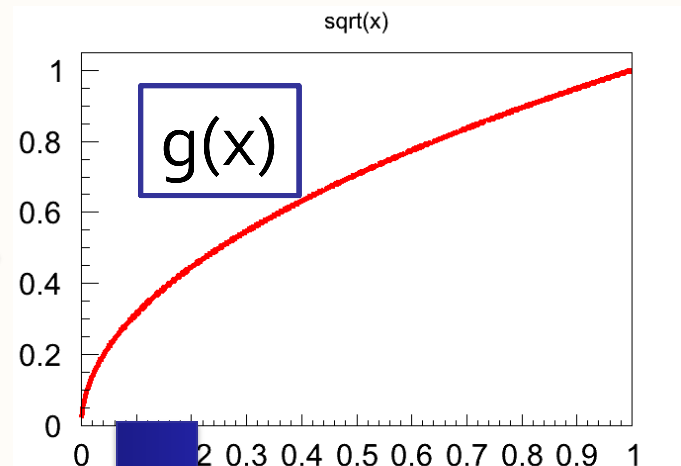


- Problem: waste of CPU, especially if function evaluation costs a lot of CPU time

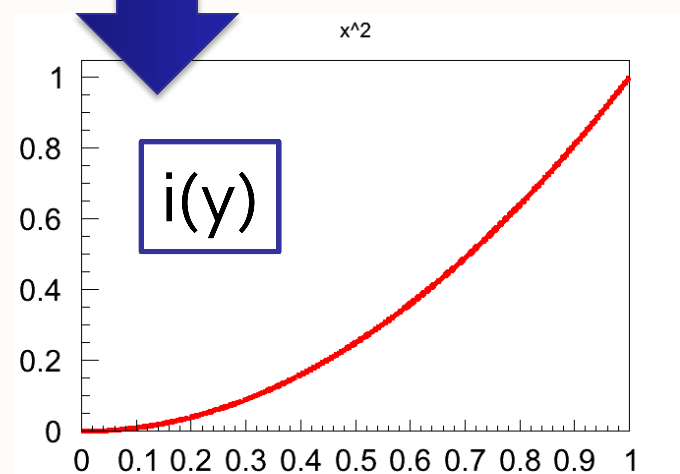
Random Distribution From Inverse

- Integral g(x) of distribution f(x) (probability density function “PDF”):





- and inverse i(y) of g(x)
- random number $0 < y^* < 1$
- return $i(y^*)$
- $i(y^*)$ is distributed like f(x)



Smart Random Distribution

- Problem with inverse: must be known!
- Can combine rejection on f() and inverse of a() and b() with a(x) <= f(x) <= b(x) to reduce sampling overhead

- ROOT implements *fast* generators for random numbers distributed like Gauss, exp, Landau, Poisson…

Interlude: HELP!

ROOT is a framework – only as good as its documentation.

http://root.cern.ch

- User's Guide (it has your answers!)
- Reference Guide

  What is TRandom?
  What functions does it have?

LET'S FIRE UP ROOT!

Setting Up ROOT

Before starting ROOT:

setup environment variables $PATH, $LD_LIBRARY_PATH

(ba)sh:

```
$ source /PathToRoot/bin/thisroot.sh
```

(t)csh:

```
$ source /PathToRoot/bin/thisroot.csh
```

Starting Up ROOT

ROOT is prompt-based

```
$ root
root [0] _
```

Prompt speaks C++

```
root [0] gROOT->GetVersion();↵
(const char* 0x5ef7e8)"5.27/04"
```

ROOT As Pocket Calculator

Calculations:

```
root [0] sqrt(42)
(const double)6.48074069840786038e+00
root [1] double val = 0.17;
root [2] sin(val)
(const double)1.69182349066996029e-01
```

Uses C++ Interpreter CINT

Running Code

To run function mycode() in file mycode.C:

```
root [0] .x mycode.C
```

Equivalent: load file and run function:

```
root [0] .L mycode.C
root [1] mycode()
```

Quit:

```
root [0] .q
```

All of CINT's commands (help):

```
root [0] .h
```

ROOT Prompt

- ? Why C++ and not a scripting language?!
- ! You'll write your code in C++, too. Support for python, ruby,… exists.

- ? Why a prompt instead of a GUI?
- ! ROOT is a programming framework, not an office suite. Use GUIs where needed.

Running Code

Macro: file that is interpreted by CINT (**`.x`**)

```
int mymacro(int value)
{
  int ret = 42;
  ret += value;
  return ret;
}
```

Execute with **`.x mymacro.C(42)`**

Compiling Code: ACLiC

Load code as shared lib, much faster:

```
.x mymacro.C+(42)
```

Uses the system's compiler, takes seconds

Subsequent **`.x mymacro.C+(42)`** check for changes, only rebuild if needed

Exactly as fast as e.g. Makefile based stand-alone binary!

CINT knows types, functions in the file, e.g. call

```
mymacro(43)
```

Compiled versus Interpreted

- ? Why compile?
- ! Faster execution, CINT has limitations, validate code.

- ? Why interpret?
- ! Faster Edit → Run → Check result → Edit cycles ("rapid prototyping").
  Scripting is sometimes just easier.

- ? Are Makefiles dead?
- ! Yes! ACLiC is even platform independent!

A LITTLE C++

A Little C++

Hopefully many of you know – but some don't.

- Object, constructor, assignment
- Pointers
- Scope, destructor
- Stack vs. heap
- Inheritance, virtual functions

If you use C++ you *have* to understand these concepts!

Objects, Constructors, =

Look at this code:

```
TNamed myObject("name", "title");
TNamed mySecond;
mySecond = myObject;
cout << mySecond.GetName() << endl;
```

Objects, Constructors, =

Look at this code:

```
TNamed myObject("name", "title");
TNamed mySecond;
mySecond = myObject;
cout << mySecond.GetName() << endl;
```

Creating objects:

1. Constructor **TNamed::TNamed(const char*, const char*)**
2. Default constructor **TNamed::TNamed()**

Objects, Constructors, =

Look at this code:

```
TNamed myObject("name", "title");
TNamed mySecond;
mySecond = myObject;
cout << mySecond.GetName() << endl;
```

3. Assignment:

mySecond

```
TNamed:
fName ""
fTitle ""
```

myObject

```
TNamed:
fName "name"
fTitle "title"
```

Objects, Constructors, =

Look at this code:

```
TNamed myObject("name", "title");
TNamed mySecond;
mySecond = myObject;
cout << mySecond.GetName() << endl;
```

Assignment: creating a twin

Objects, Constructors, =

Look at this code:

```
TNamed myObject("name", "title");
TNamed mySecond;
mySecond = myObject;
cout << mySecond.GetName() << endl;
```

4. New content

mySecond

```
TNamed:
fName "name"
fTitle "title"
```

output: `"name"`

Pointers

Modified code:

```
TNamed myObject("name", "title");
TNamed* pMySecond = 0;
pMySecond = &myObject;
cout << pMySecond->GetName() << endl;
```

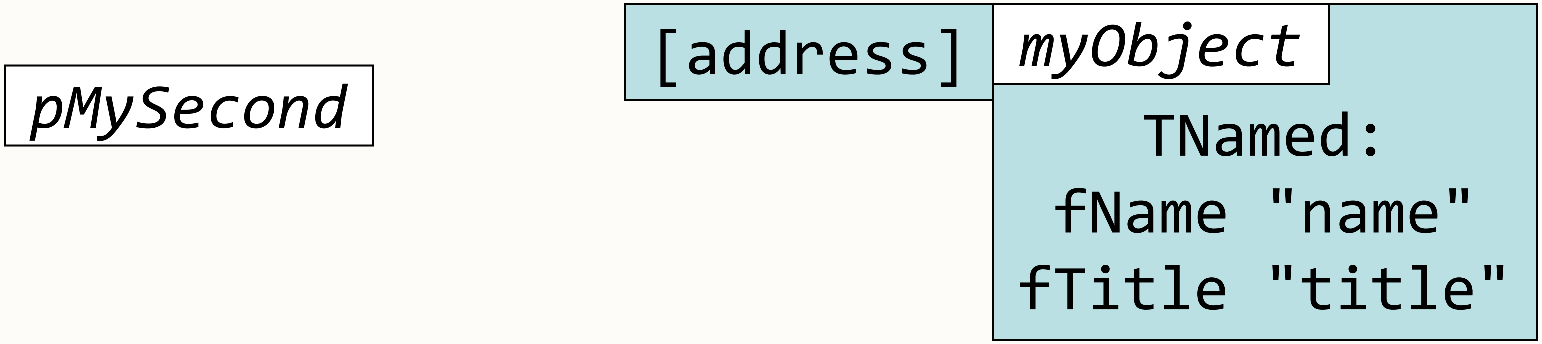
Pointer declared with "*", initialize to 0

Pointers

Modified code:

```
TNamed myObject("name", "title");
TNamed* pMySecond = 0;
pMySecond = &myObject;
cout << pMySecond->GetName() << endl;
```

"&" gets address:



Pointers

Modified code:

```
TNamed myObject("name", "title");
TNamed* pMySecond = 0;
pMySecond = &myObject;
cout << pMySecond->GetName() << endl;
```

Assignment: point to myObject; no copy

Pointers

Modified code:

```
TNamed myObject("name", "title");
TNamed* pMySecond = 0;
pMySecond = &myObject;
cout << pMySecond->GetName() << endl;
```

Access members of value pointed to by "->"

Pointers

Changes propagated:

```
TNamed myObject("name", "title");
TNamed* pMySecond = 0;
pMySecond = &myObject;
pMySecond->SetName("newname");
cout << myObject.GetName() << endl;
```

Pointer forwards to object

Name of object changed – prints "newname"!

Object vs. Pointer

Compare object:

```
TNamed myObject("name", "title");
TNamed mySecond = myObject;
cout << mySecond.GetName() << endl;
```

to pointer:

```
TNamed myObject("name", "title");
TNamed* pMySecond = &myObject;
cout << pMySecond->GetName() << endl;
```

Object vs. Pointer: Parameters

Calling functions: object parameter obj gets copied for function call!

```
void funcO(TNamed obj);

TNamed myObject;
funcO(myObject);
```

Pointer parameter: only address passed, no copy

```
void funcP(TNamed* ptr);

TNamed myObject;
funcP(&myObject);
```

Object vs. Pointer: Parameters

Functions changing parameter: funcO can only access copy! **`caller`** not changed!

```
void funcO(TNamed obj){
  obj.SetName("nope");
}
funcO(caller);
```

Using pointers (or references) funcP can change **`caller`**

```
void funcP(TNamed* ptr){
  ptr->SetName("yes");
}
funcP(&caller);
```

Scope

Scope: range of visibility and C++ "life".

Birth: constructor, death: destructor

```
{ // birth: TNamed() called
  TNamed n;
} // death: ~TNamed() called
```

Variables are valid / visible only in scopes:

```
int a = 42;
{ int a = 0; }
cout << a << endl;
```

Scope

Functions are scopes:

```
void func(){ TNamed obj; }

func();
cout << obj << end; // obj UNKNOWN!
```

must not return pointers to local variables!

```
TNamed* func(){
  TNamed obj;
  return &obj; // BAD!
}
```

Stack vs. Heap

So far only stack:

```
TNamed myObj("n","t");
```

Fast, but often < 10MB. Only survive in scope.

Heap: slower, GBs (RAM + swap), creation and destruction managed by user:

```
TNamed* pMyObj = new TNamed("n","t");
delete pMyObj; // or memory leak!
```

Stack vs. Heap: Functions

Can return heap objects without copying:

```
TNamed* CreateNamed(){
  // user must delete returned obj!
  TNamed* ptr = new TNamed("n","t");
  return ptr; }
```

ptr gone – but TNamed object still on the heap, address returned!

```
TNamed* pMyObj = CreateNamed();
cout << pMyObj->GetName() << endl;
delete pMyObj; // or memory leak!
```

Inheritance

Classes "of same kind" can re-use functionality

E.g. plate and bowl are both dishes:

```
class TPlate: public TDish {...};
class TBowl: public TDish {...};
```

Can implement common functions in TDish:

```
class TDish {
public:
  void Wash();
};
```

```
TPlate *a = new TPlate();
a->Wash();
```

Inheritance: The Base

Use TPlate, TBowl as dishes:
- assign pointer of derived to pointer of base "every plate is a dish"

```
TDish *a = new TPlate();
TDish *b = new TBowl();
```

But not every dish is a plate, i.e. the inverse doesn't work.
- And a bowl is totally not a plate!

```
TPlate* p = new TDish(); // NO!
TPlate* q = new TBowl(); // NO!
```

Virtual Functions

Often derived classes behave differently:

```
class TDish { ...
  virtual bool ForSoup() const;
};
class TPlate: public TDish { ...
  bool ForSoup() const { return false; }
};
class TBowl: public TDish { ...
  bool ForSoup() const { return true; }
};
```

Pure Virtual Functions

But TDish cannot know! Mark as "not implemented"

```
class TDish { ...
  virtual bool ForSoup() const = 0;
};
```

Only for virtual functions.

Cannot create object of TDish anymore (one function is missing!)

Calling Virtual Functions

Call to virtual functions evaluated at runtime:

```
void FillWithSoup(TDish* dish) {
  if (dish->ForSoup())
    dish->SetFull();
}
```

Works for any type as expected:

```
TDish* a = new TPlate();
TDish* b = new TBowl();
FillWithSoup(a); // will not be full
FillWithSoup(b); // is now full
```

Virtual vs. Non-Virtual

So what happens if non-virtual?

```
class TDish { ...
  bool ForSoup() const {return false;}
};
```

Will now always call TDish::ForSoup(), i.e. false

```
void FillWithSoup(TDish* dish) {
  if (dish->ForSoup())
    dish->SetFull();
}
```

Congrats!

CERN School of Computing

You have earned yourself the CSC ROOT C++ Diploma.

From now on you may use C++ without feeling lost!

Summary

We know:

- why and how to start ROOT
- C++ basics
- that you run your code with ".x"
- can call functions in libraries
- can (mis-) use ROOT as a pocket calculator!

Lots for you to discover during next two lectures and especially the exercises!

Streaming, Reflection, TFile,

Schema Evolution

SAVING DATA

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);
```

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

Serialization

Store *data members* of TNamed: serialization

1) type of object: runtime-type-information RTTI

2) data members for the type: reflection

3) where data members are in memory: introspection

4) read their values from memory, write to disk: raw I/O

Complex task, and C++ is not your friend.

Reflection

Need type description (aka *reflection*)

1. types, sizes, members

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

TMyClass is a class.

Members:

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

Platform Data Types

Fundamental data types (int, long,…):
  size is platform dependent

Store "long" on 64bit platform, writing 8 bytes:
  00, 00, 00, 00, 00, 00, 00, 42

Read on 32bit platform, "long" only 4 bytes:
  00, 00, 00, 00

Data loss, data corruption!

ROOT Basic Data Types

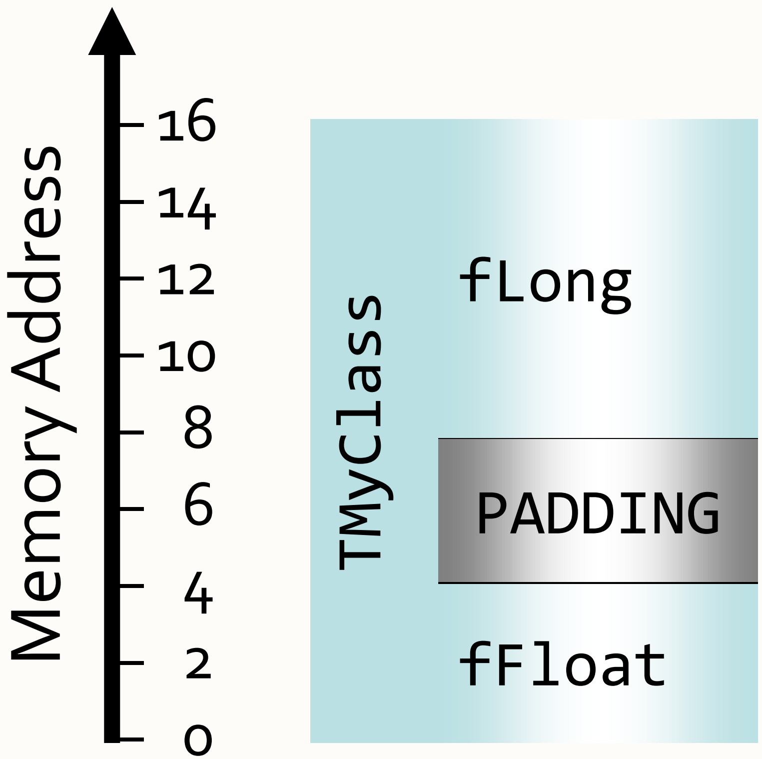
Solution: ROOT typedefs

Signed	Unsigned	sizeof [bytes]
`Char_t`	`UChar_t`	1
`Short_t`	`UShort_t`	2
`Int_t`	`UInt_t`	4
`Long64_t`	`ULong64_t`	8
`Double32_t`		float on disk, double in RAM

Reflection

Need type description (platform dependent)

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



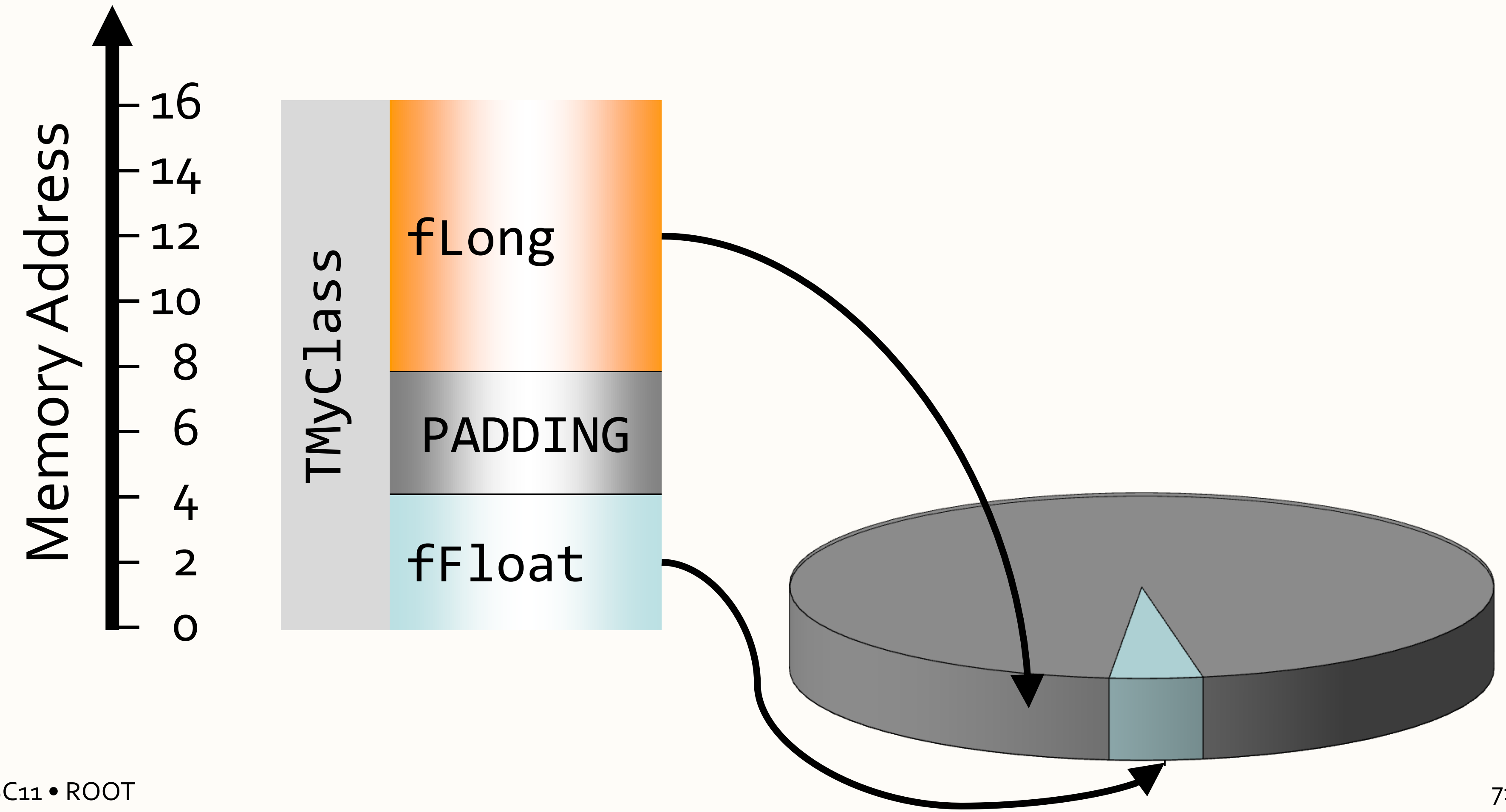

```
class TMyClass {
  float fFloat;
  Long64_t fLong;
};
```

"fFloat" is at offset 0

"fLong" is at offset 8

I/O Using Reflection

members → memory → disk



C++ Is Not Java

Lesson: need reflection!

Where from?

Java: get data members with

```
Class.forName("MyClass").getFields()
```

C++: get data members with

– oops. Not part of C++.

CAREFUL

THIS LANGUAGE HAS NO BRAIN. USE YOUR OWN

ROOT And Reflection

Simply use ACLiC:

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

Creates library with reflection data ("dictionary") of all types in MyCode.cxx!

Dictionary needed for interpreter, too

ROOT has dictionary for all its types

Back To Saving Objects

Given a TFile:

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

Write an object deriving from TObject:

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

"optionalName" or `TObject::GetName()`

Write any object (with dictionary):

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

TFile

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");
```

TFile compresses data ("zip"):

```
f->GetCompressionFactor()
2.6
```

"Where Is My Histogram?"

TFile owns histograms, graphs, trees
(due to historical reasons):

```
TFile* f = TFile::Open("myfile.root");
TH1F* h = new TH1F("h","h",10,0.,1.);
TNamed* o = new TNamed("name", "title");
o->Write();
delete f;
```

h automatically deleted: owned by file.

o still there

even if saving o to file!

unique names!

TFile acts like a scope for hists, graphs, trees!

Risks With I/O

Physicists can loop a lot:

For each particle collision

For each particle created

For each detector module

Do something.

Physicists can loose a lot:

Run for hours…

Crash.

Everything lost.

Name Cycles

Create snapshots regularly:

MyObject;1

MyObject;2

...

MyObject;5427

MyObject

Write() does not replace but append!

but see documentation TObject::Write()

The "I" Of I/O

Reading is simple:

```
TFile* f = TFile::Open("myfile.root");
TH1F* h = 0;
f->GetObject("h", h);
h->Draw();
delete f;
```

Remember:

- TFile owns histograms!
- file gone, histogram gone!

Ownership And TFiles

Separate TFile and histograms:

```
TFile* f = TFile::Open("myfile.root");
TH1F* h = 0;
TH1::AddDirectory(kFALSE);
f->GetObject("h", h);
h->Draw();
delete f;
```

… and h will stay around.

Put in root_logon.C in current directory to be executed when root starts

Changing Class – The Problem

Things change:

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

Changing Class – The Problem

Things change:

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

Inconsistent reflection data, mismatch in memory, on disk

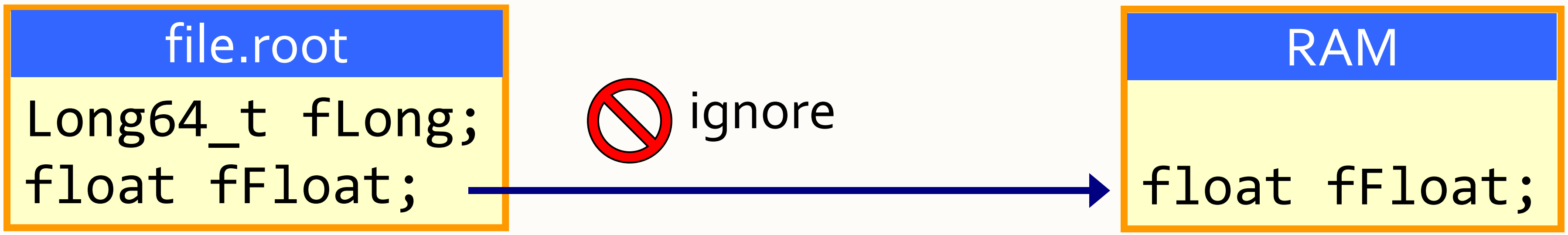
Objects written with old version cannot be read

Need to store reflection with data to detect!

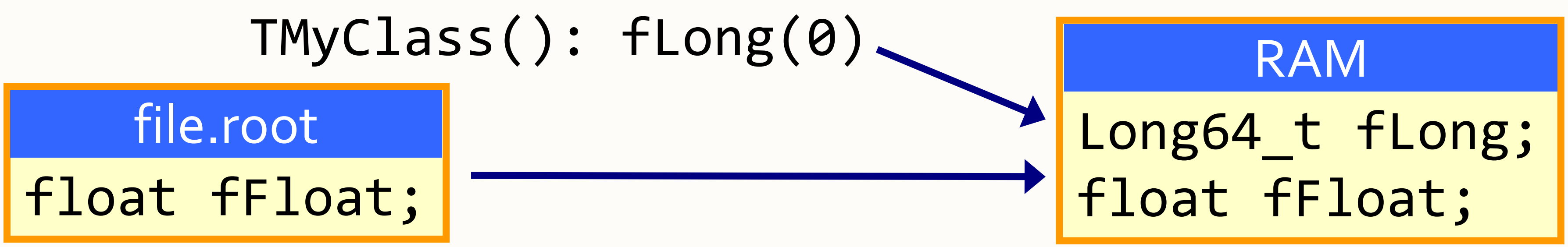
Schema Evolution

Simple rules to convert disk to memory layout

1. skip removed members



2. default-initialize added members



3. convert members where possible

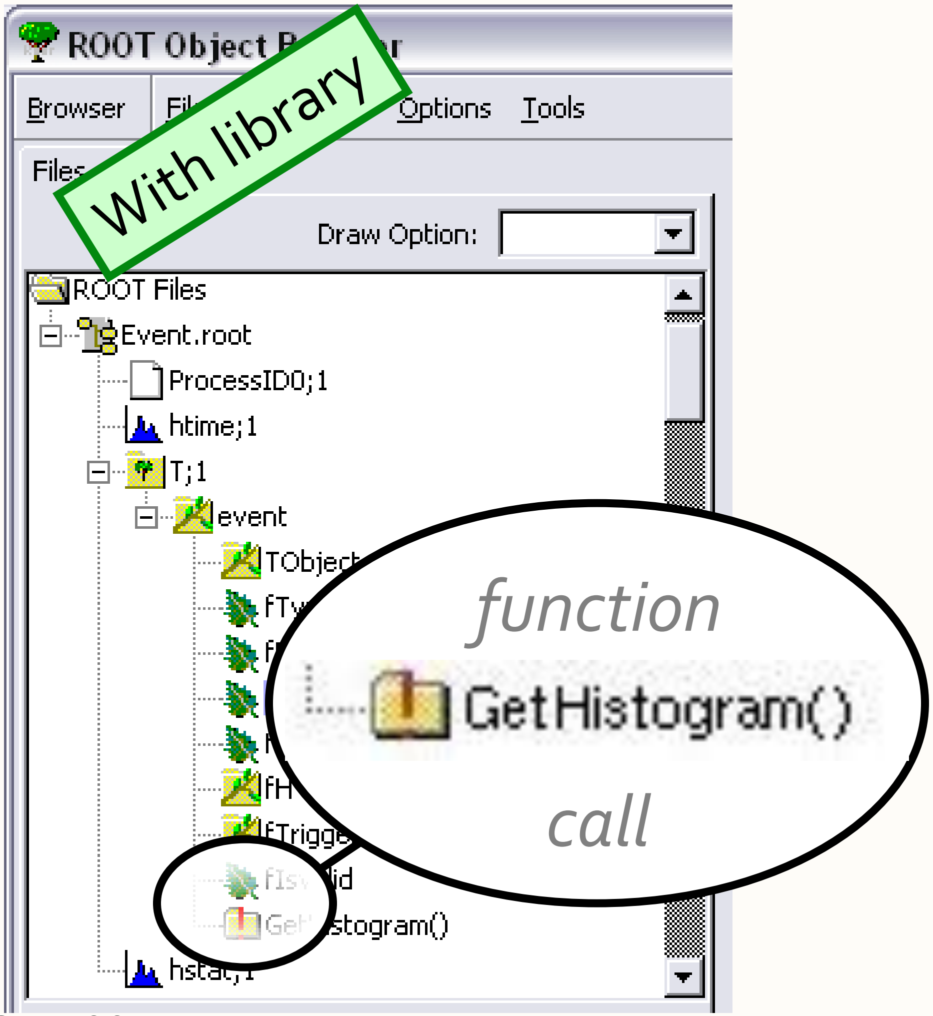
Class Version

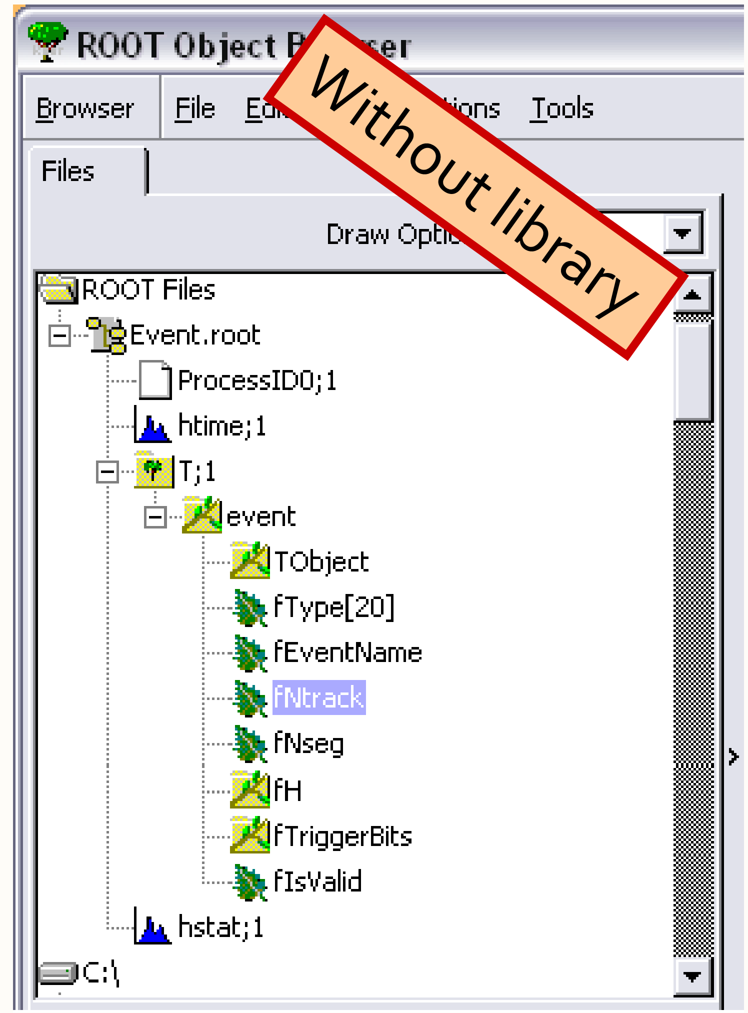
- ClassDef() macro makes I/O faster, needed when deriving from TObject
- Can have multiple class versions in same file
- Use version number to identify layout:

```
class TMyClass: public TObject {
public:
  TMyClass(): fLong(0), fFloat(0.) {}
  virtual ~TMyClass() {}
  ...
  ClassDef(TMyClass,1); // example class
};
```

Reading Files

Files store reflection and data: need no library!





ROOT I/O Candy

- Nice viewer for TFile: `new TBrowser`
- Can even open `TFile::Open("http://cern.ch/file.root")` including read-what-you-need!
- Combine contents of TFiles with `$ROOTSYS/bin/hadd`

Summary

Big picture:

- you know ROOT files – for petabytes of data
- you learned that reflection is key for I/O
- you learned what schema evolution is

Small picture:

- you can write your own data to files
- you can read it back
- you can change the definition of your classes

ROOT COLLECTION CLASSES

Collection Classes

ROOT collections polymorphic containers: hold pointers to `TObject`, so:

- Can only hold objects that inherit from `TObject`
- Return pointers to `TObject`, that have to be cast back to the correct subclass

```
void DrawHist(TObjArray *vect, int at)
{
   TH1F *hist = (TH1F*)vect->At(at);
   if (hist) hist->Draw();
}
```

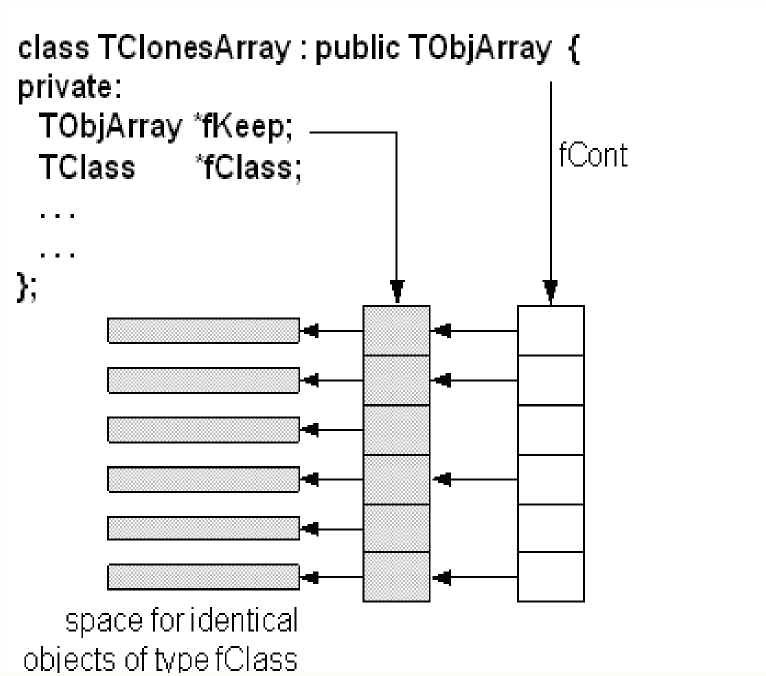
TClonesArray

Array of objects of the same class ("clones")

Designed for repetitive data analysis tasks:

- same type of objects
- created and deleted
- many times.

No comparable class in STL!



The internal data structure of a TClonesArray

TClonesArray

Standard array:

```
while (next_event()) {

   for (int i=0;i<N;++i)
      a[i] = new TTrack(x,y,z);
   do_something(a);

   a.clear();
};
```

TClonesArray:

```
while (next_event()) {

   for (int i=0;i<N;++i)
      new(a[i]) TTrack(x,y,z);
   do_something(a);

   a.Delete();
};
```

Traditional Arrays

Very large number of new and delete calls in large loops like this (N_{events} x N_{tracks} times new/delete):

```
TObjArray a(10000);
while (TEvent *ev = (TEvent *)next()) {
   for (int i = 0; i < ev->Ntracks; ++i) {
      a[i] = new TTrack(x,y,z,...);
      ...
   }
   a.Delete();
}
```

N_{events} = 100000

N_{tracks} = 10000

Use of TClonesArray

You better use a `TClonesArray` which reduces the number of new/delete calls to only N_{tracks}:

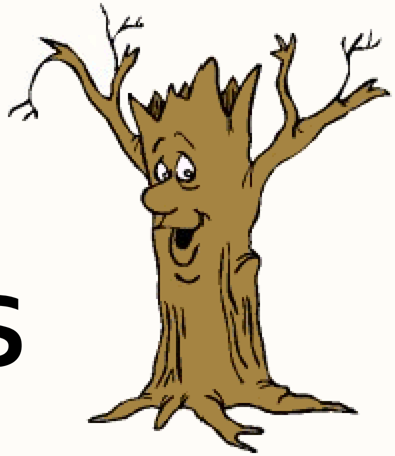
```
TClonesArray a("TTrack", 10000);
while (TEvent *ev = (TEvent *)next()) {
   for (int i = 0; i < ev->Ntracks; ++i) {
      new(a[i]) TTrack(x,y,z,...);
      ...
   }
   a.Delete();
}
```

N_{events} = 100000

N_{tracks} = 10000

- Pair of new / delete calls cost about 4 μs
- Allocating / freeing memory N_{events}*N_{tracks} = 10^9 times costs about 1 hour!

ROOT TREES

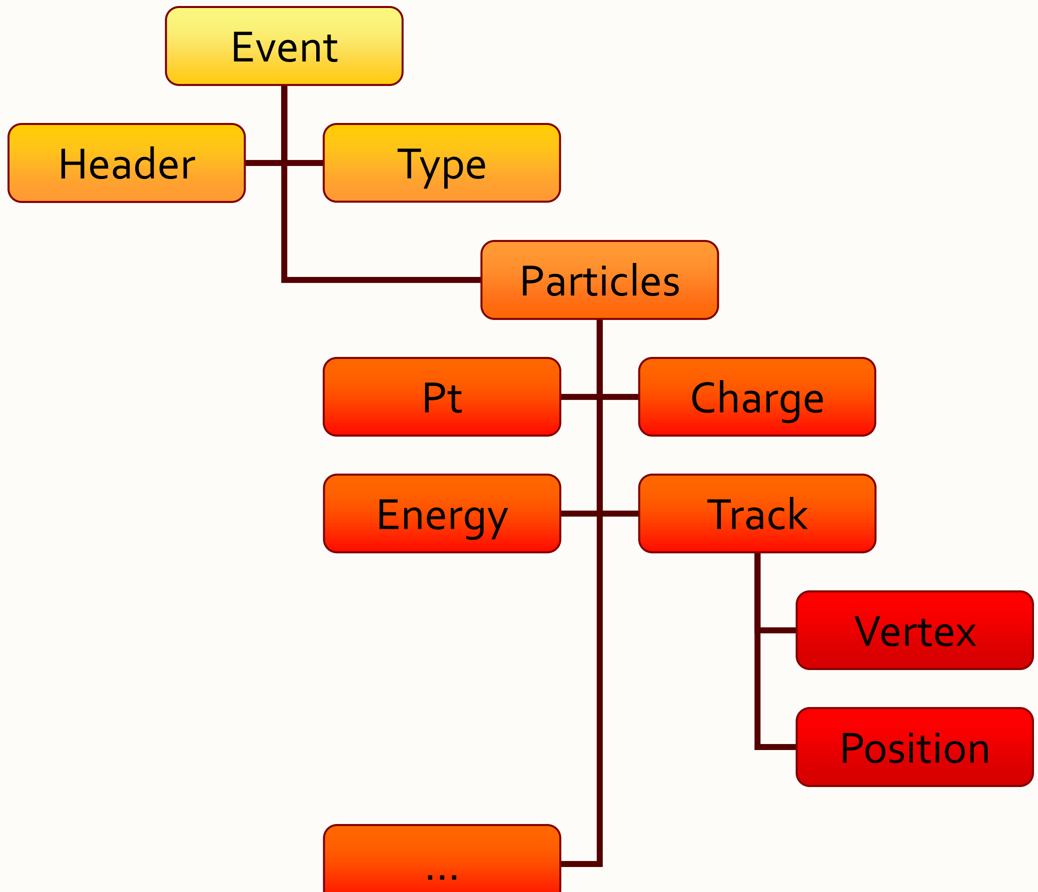


Trees

From:
Simple data types
(e.g. Excel tables)

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
0.552454	-0.21231	0.350281
-0.18495	1.187305	1.443902
0.205643	-0.77015	0.635417
1.079222	-0.32739	1.271904
-0.27492	-1.72143	3.038899
2.047779	-0.06268	4.197329
-0.45868	-1.44322	2.293266
0.304731	-0.88464	0.875442
-0.71234	-0.22239	0.556881
-0.27187	1.181767	1.470484
0.886202	-0.65411	1.213209
-2.03555	0.527648	4.421883
-1.45905	-0.464	2.344113
1.230661	-0.00565	1.514559
[illegible]	[illegible]	3.562347

To:
Complex data types
(e.g. Database tables)



Why Trees ?

- Extremely efficient write once, read many ("WORM")
- Designed to store $>10^9$ (HEP events) with same data structure
- Trees allow fast direct and random access to any entry (sequential access is the best)
- Optimized for network access (read-ahead)

Why Trees ?

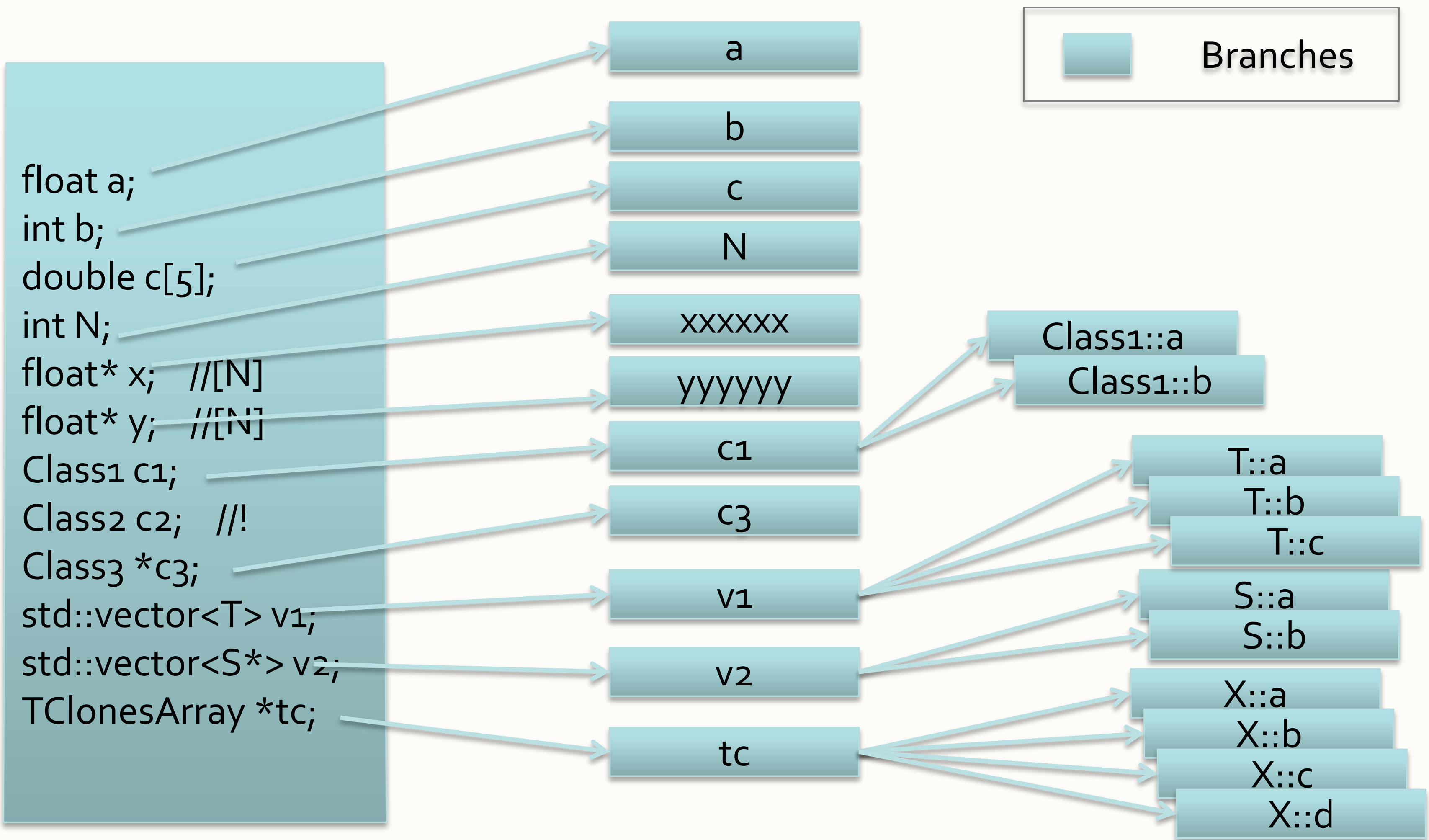
object.Write() convenient for simple objects like histograms, inappropriate for saving collections of events containing complex objects

- Reading a collection: read all elements (all events)
- With trees: only one element in memory, or even only a part of it (less I/O)
- Trees buffered to disk (TFile); I/O is integral part of TTree concept

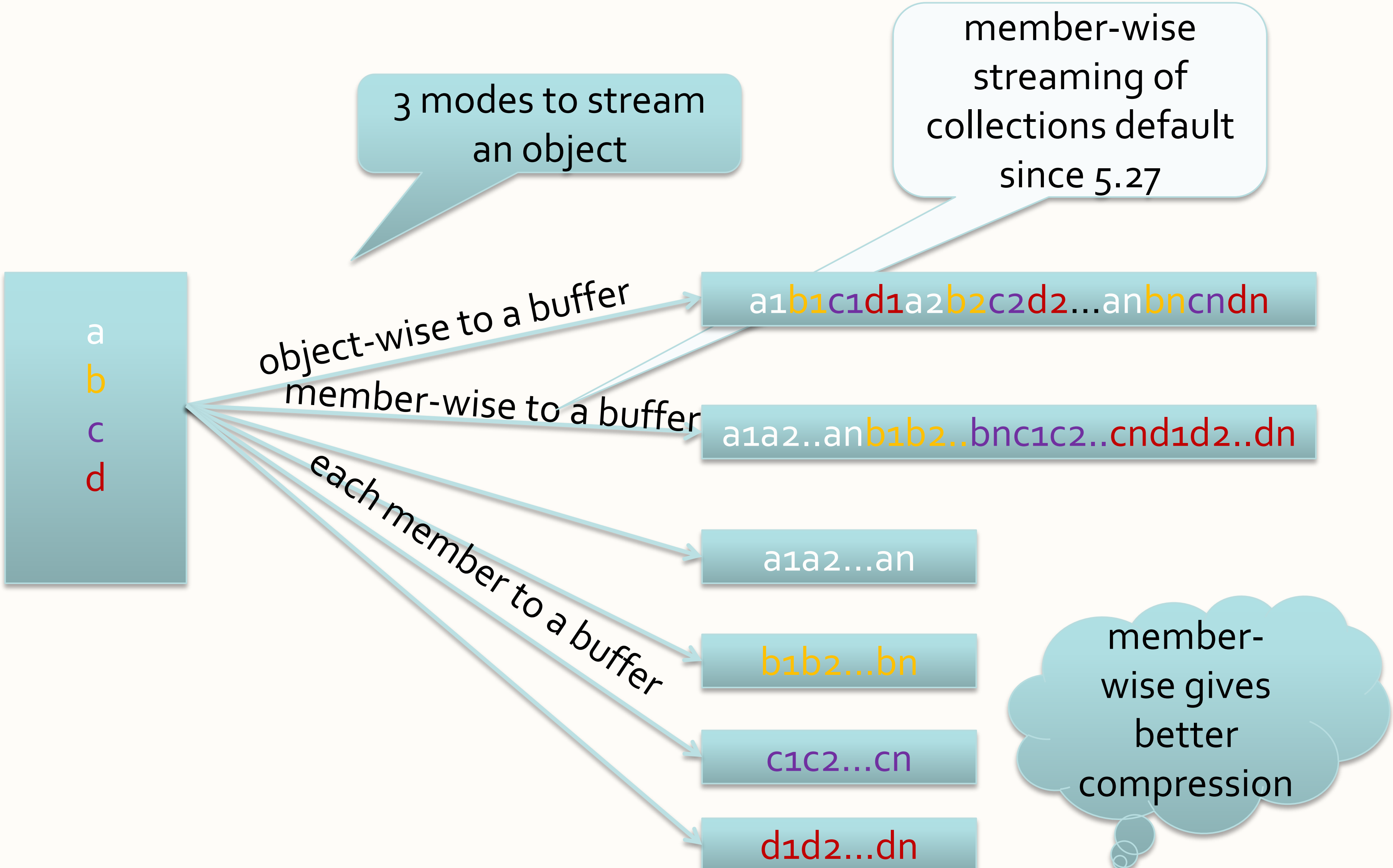
Tree Access

- Databases have row wise access
  - Can only access the full object (e.g. full event)
- ROOT trees have column wise access
  - Direct access to any event, any branch or any leaf even in the case of variable length structures
  - Designed to access only a subset of the object attributes (e.g. only particles' energy)
  - Makes same members consecutive, e.g. for object with position in X, Y, Z, and energy E, all X are consecutive, then come Y, then Z, then E. A lot higher zip efficiency!

Branch Creation from Class



ObjectWise/MemberWise Streaming



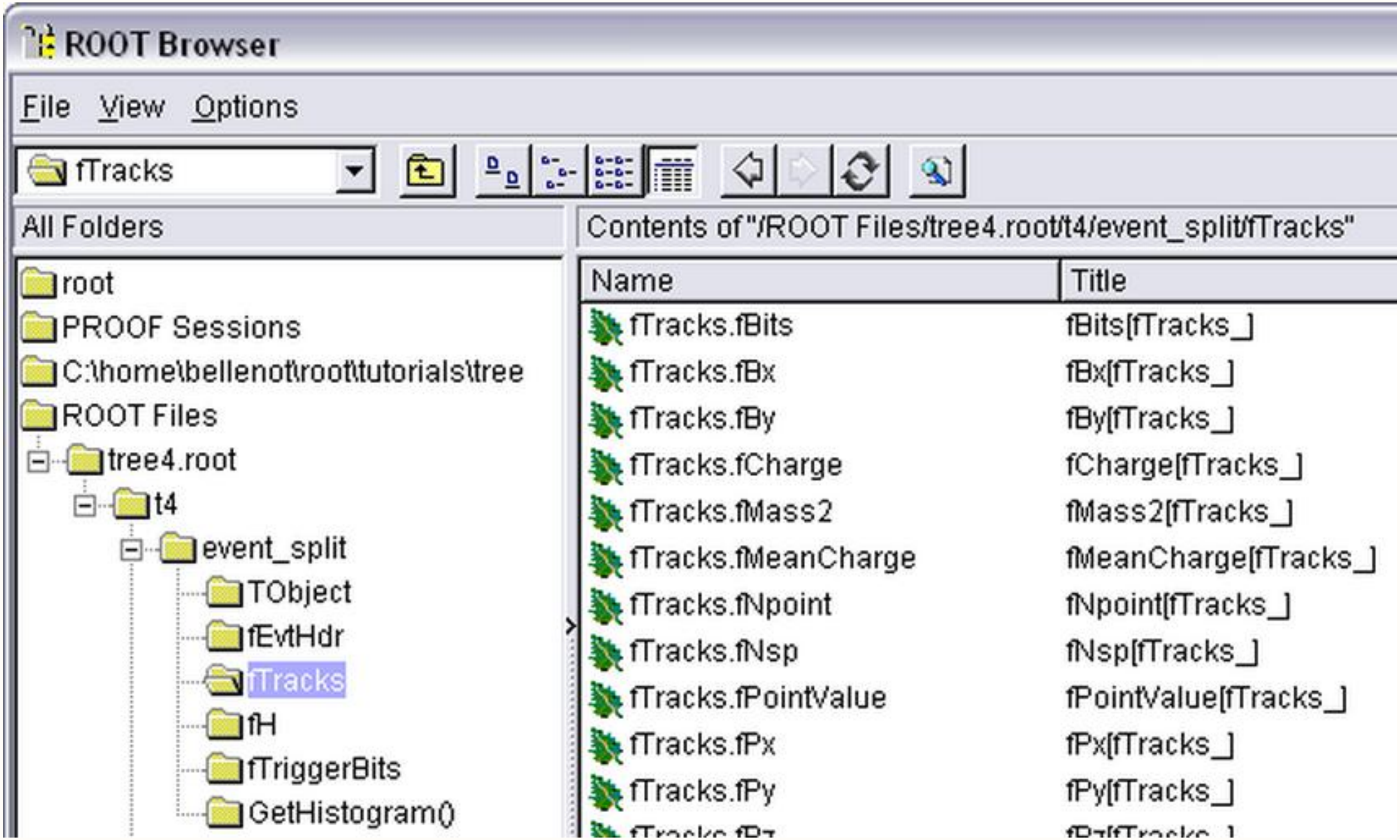
Building ROOT Trees

Overview of

- Trees
- Branches

5 steps to build a TTree

Tree structure

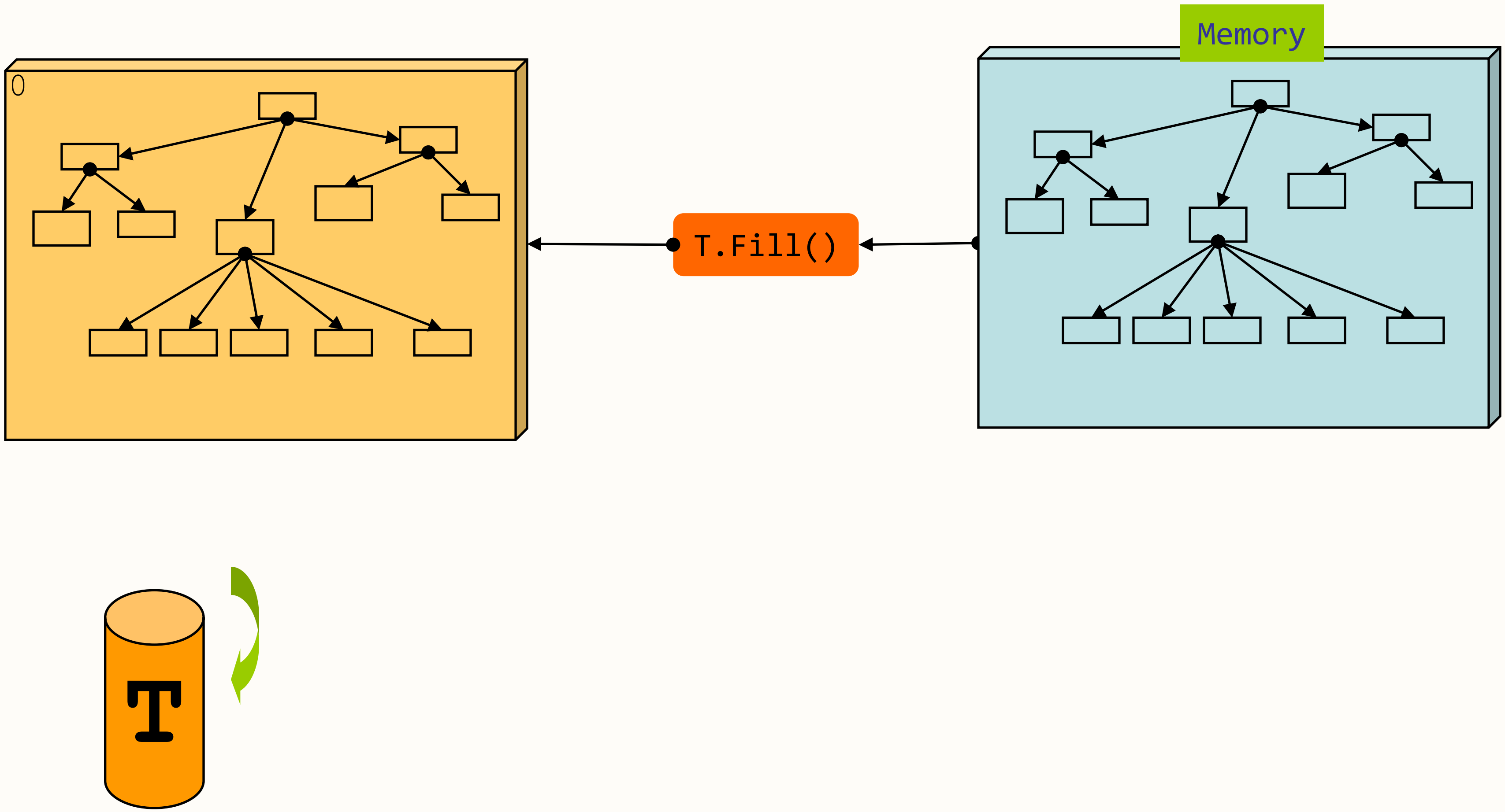


Tree structure

- Branches: directories
- Leaves: data containers
- Can read a subset of all branches – speeds up considerably the data analysis processes
- Branches of the same **TTree** can be written to separate files

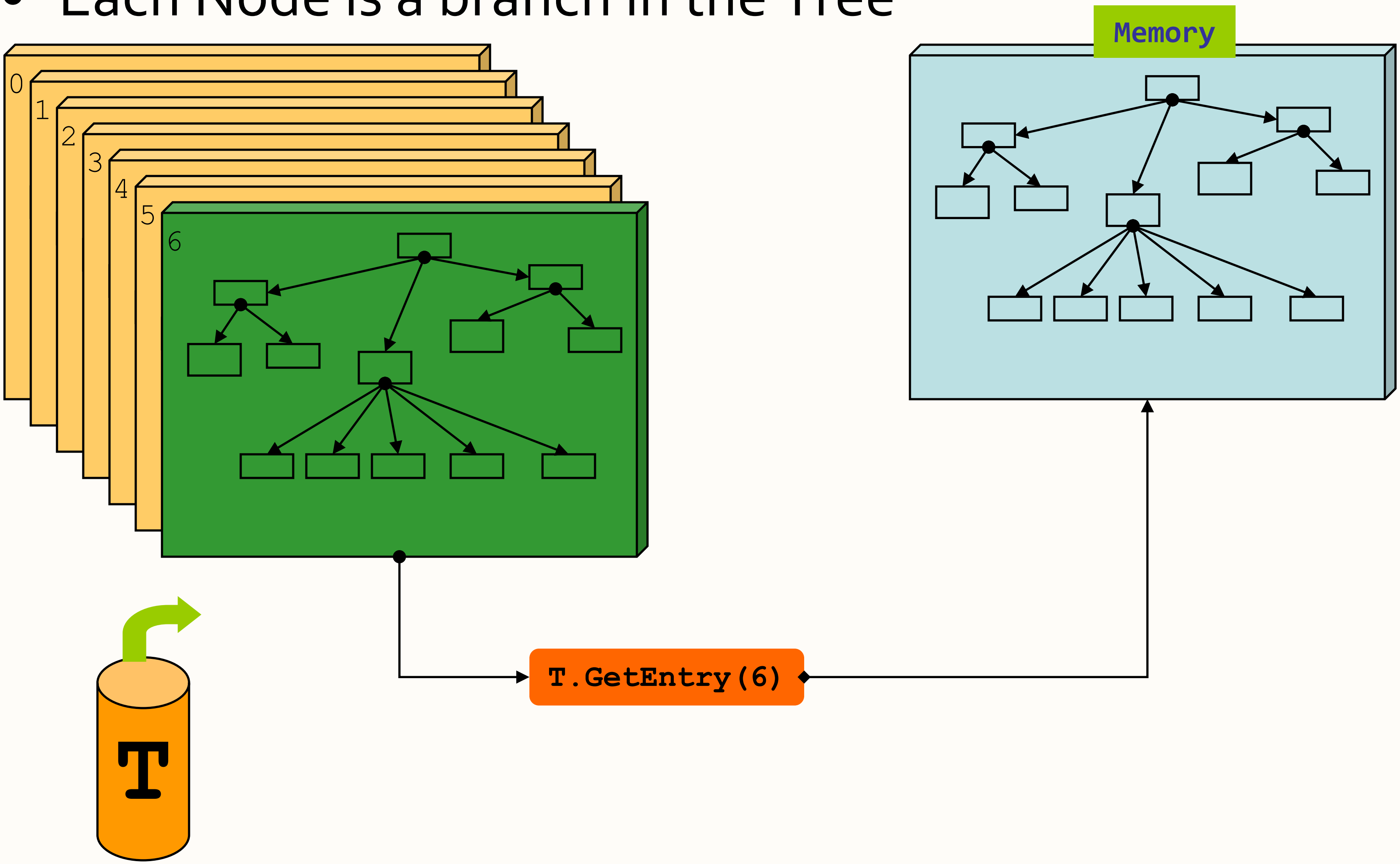
Memory ↔ Tree

- Each Node is a branch in the Tree



Memory ↔ Tree

- Each Node is a branch in the Tree



Five Steps to Build a Tree

Steps:

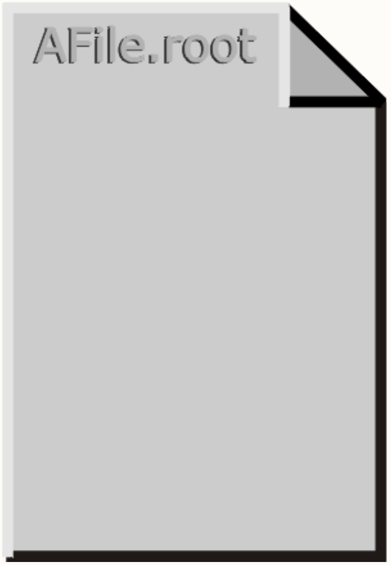
1. Create a TFile
2. Create a TTree
3. Add TBranch to the TTree
4. Fill the tree
5. Write the file

Example macro

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

Step 1: Create a TFile Object

Trees can be huge → need file for swapping filled entries

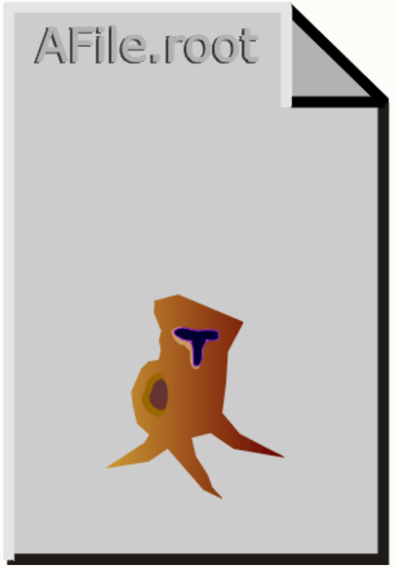



```
TFile *hfile = TFile::Open("AFile.root",
                           "RECREATE");
```

Step 2: Create a TTree Object

The TTree constructor:

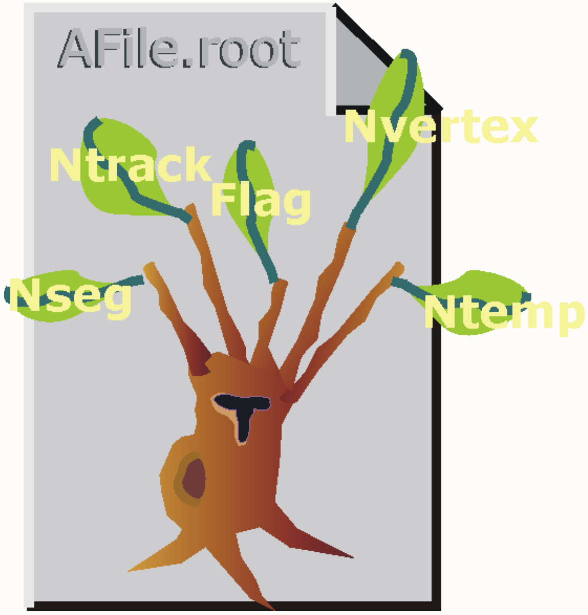
- Tree name (e.g. "myTree")
- Tree title




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

Step 3: Adding a Branch

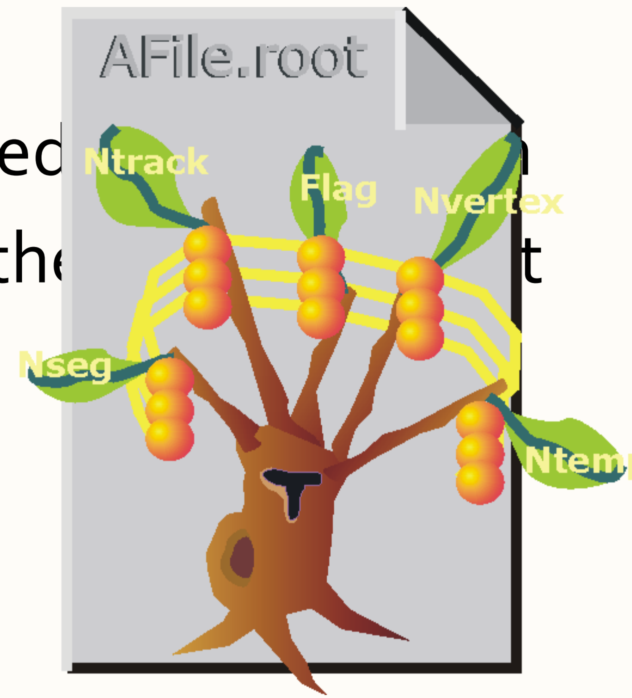
- Branch name
- Address of pointer
  to the object




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

Step 4: Fill the Tree

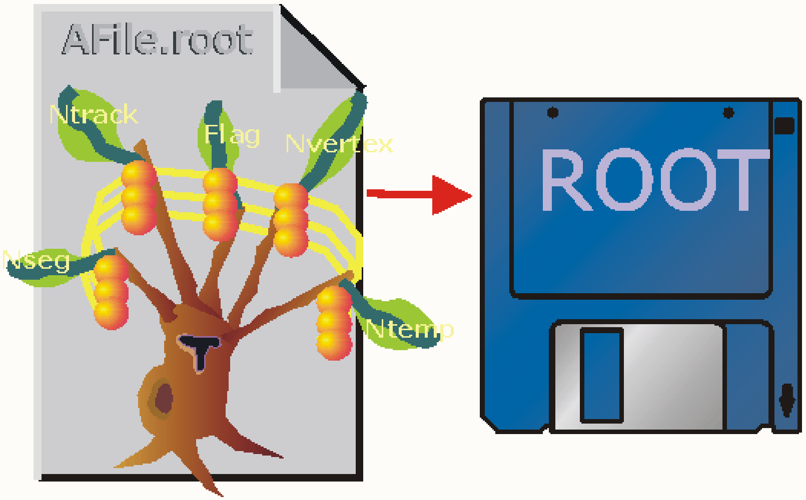
- Create a for loop
- Assign values to the object contained n
- TTree::Fill() creates a new entry in the t
  of values of branches' objects




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

Step 5: Write Tree To File

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



Reading a TTree

- Looking at a tree
- How to read a tree
- Friends and chains

Example macro

```
void ReadTree() {
  TFile f("AFile.root");
  TTree *T = (TTree*)f->Get("T");
  Event *myE = 0; TBranch* brE = 0;
  T->SetBranchAddress("EvBranch", &myE, brE);
  T->SetCacheSize(10000000);
  T->AddBranchToCache("EvBranch");
  Long64_t nbent = T->GetEntries();
  for (Long64_t e = 0;e < nbent; ++e) {
     brE->GetEntry(e);
     myE->Analyze();
  }
}
```

Data pointers (e.g. myE) MUST be set to 0

How to Read a TTree

Example:

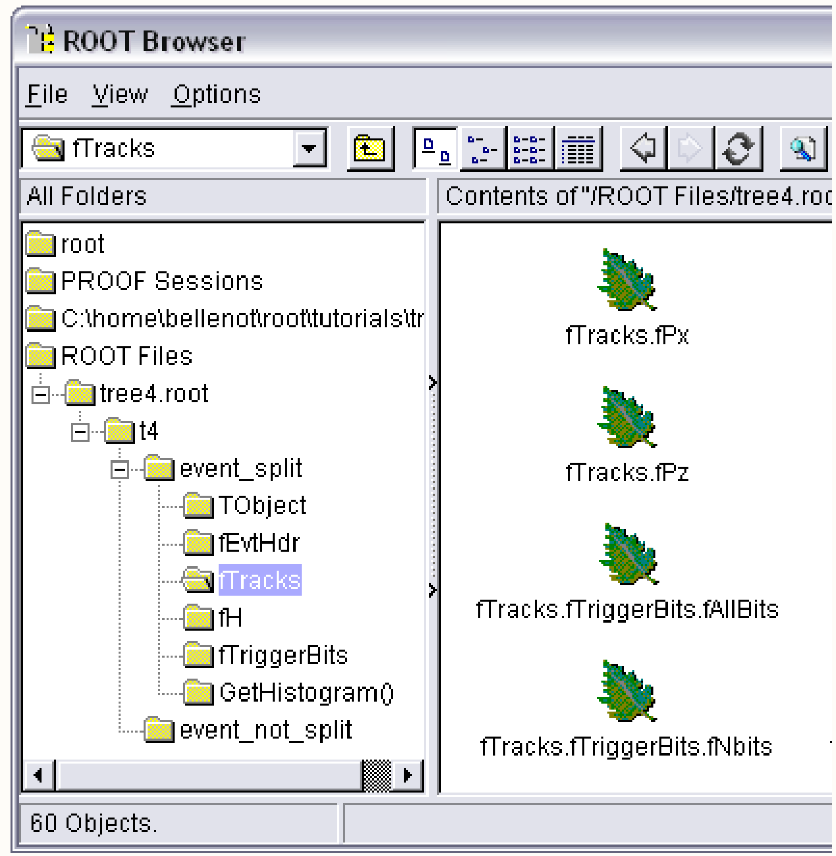
1. Open the Tfile

```
TFile f("AFile.root")
```

2. Get the TTree

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

or



How to Read a TTree

3. Create a variable pointing to the data

```
root [] Event *myEvent = 0;
```

4. Associate a branch with the variable:

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

5. Read one entry in the TTree

```
root [] myTree->GetEntry(0)
```

```
root [] 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
[...]
```

Branch Access Selection

- Use TTree::SetBranchStatus() or TBranch::GetEntry() to select branches to be read
- Speed up considerably the reading phase

```
TClonesArray* myMuons = 0;
// disable all branches
myTree->SetBranchStatus("*", 0);
// re-enable the "muon" branches
myTree->SetBranchStatus("muon*", 1);
myTree->SetBranchAddress("muon", &myMuons);
// now read (access) only the "muon" branches
myTree->GetEntry(0);
```

Looking at the Tree

TTree::Print() shows the data layout

```
root [] TFile f("AFile.root")
root [] myTree->Print();
******************************************************************************
*Tree    :myTree    : A ROOT tree                                            *
*Entries :       10 : Total =          867935 bytes  File  Size =     390138 *
*        :          : Tree compression factor =   2.72                       *
******************************************************************************
*Branch  :eBranch                                                        *
*Entries :       10 : BranchElement (see below)                              *
*............................................................................*
*Br    0 :fUniqueID :                                                        *
*Entries :       10 : Total  Size=        698 bytes  One basket in memory    *
*Baskets :        0 : Basket Size=      64000 bytes  Compression=   1.00     *
*............................................................................*
…
…
```

Looking at the Tree

TTree::Scan("leaf:leaf:….") shows the values

```
root [] myTree->Scan("fNseg:fNtrack"); > scan.txt

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 *
```

TTree Selection Syntax

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:

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

Prints the values of var1, var2 and var3 for the entries where var1 is greater than 0

Use the same syntax for TTree::Draw()

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,…
```

TChain: the Forest

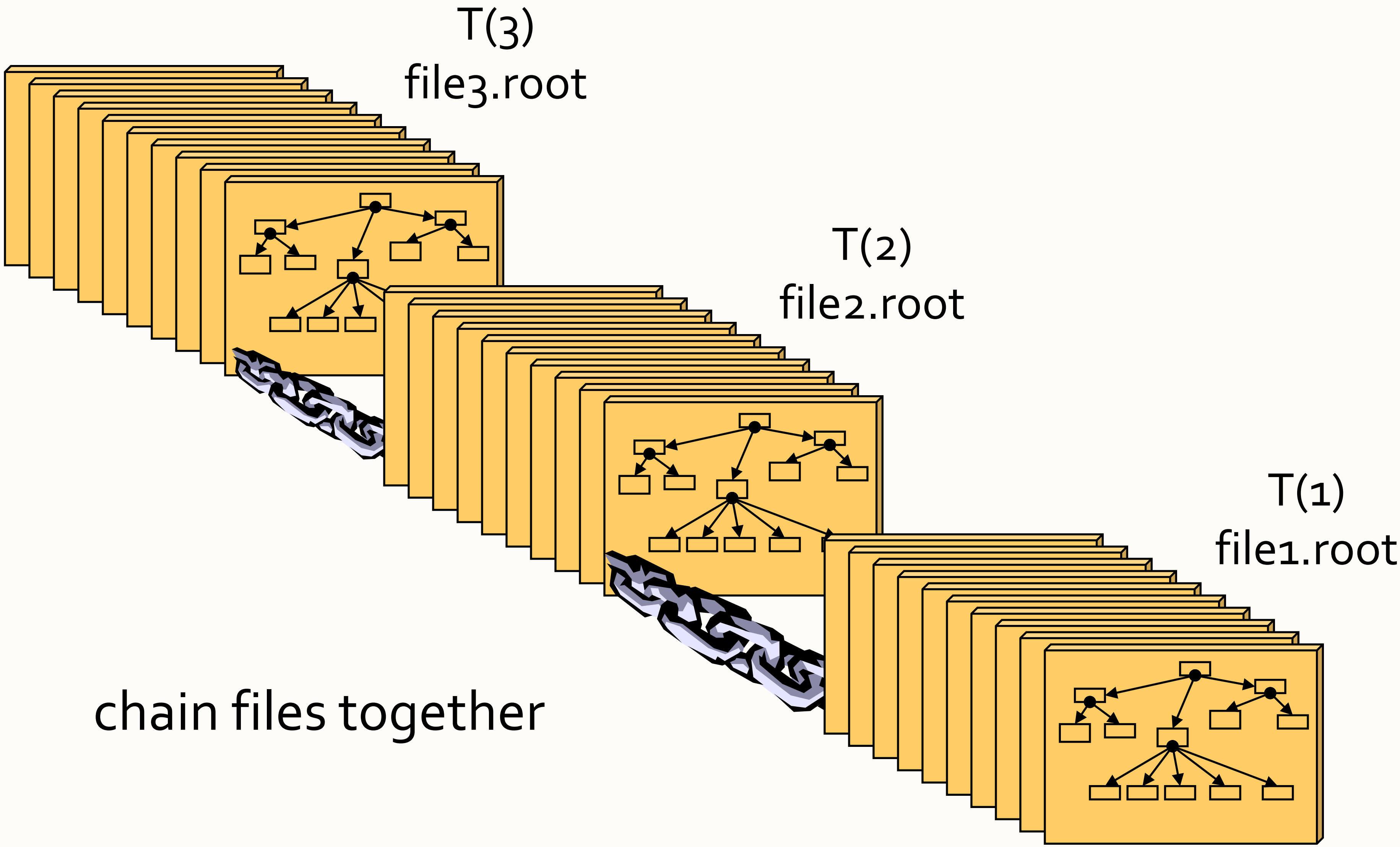
- Collection of TTrees: list of ROOT files containing the same tree
- Same semantics as TTree

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!

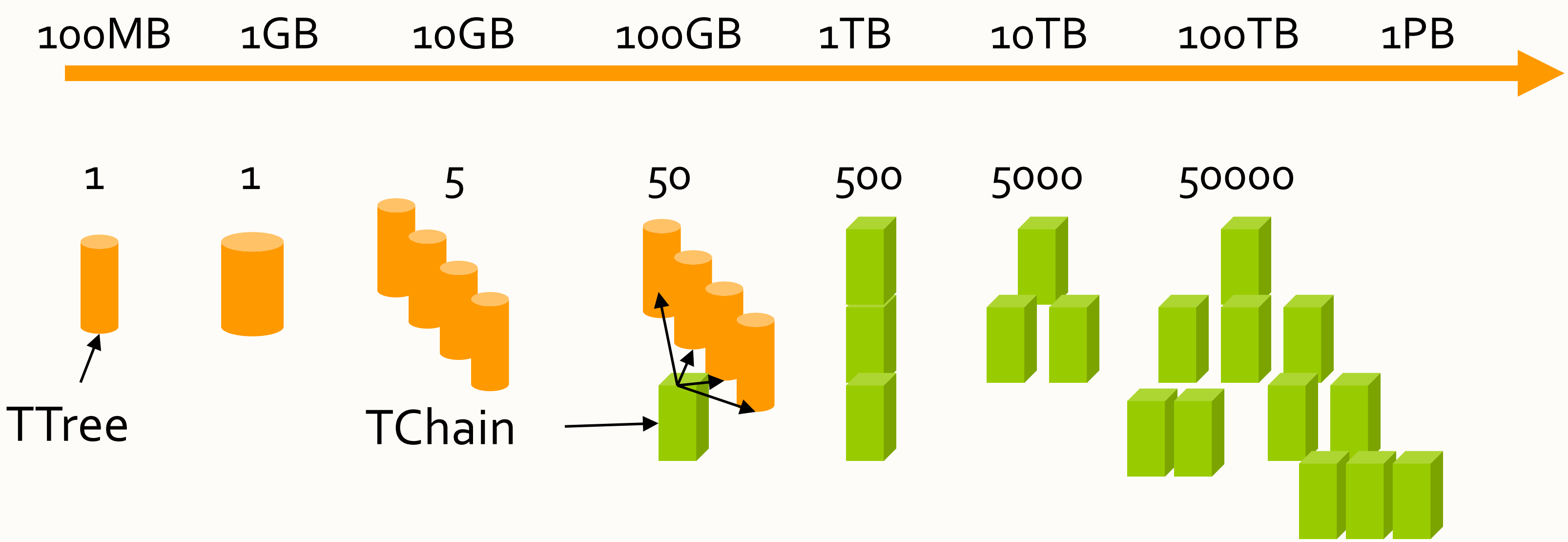
TChain



chain files together

Data Volume & Organisation

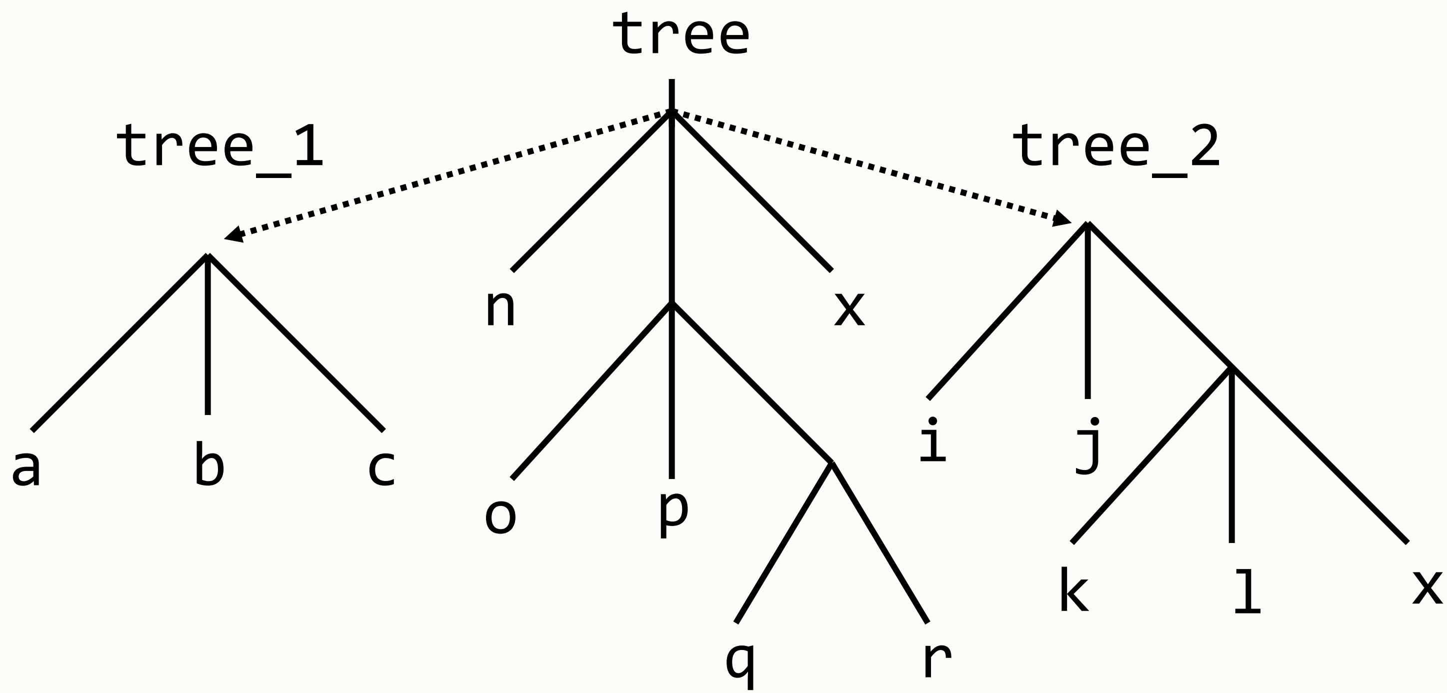
- A TFile typically contains 1 TTree
- A TChain is a collection of TTrees or/and TChains



Tree Friends

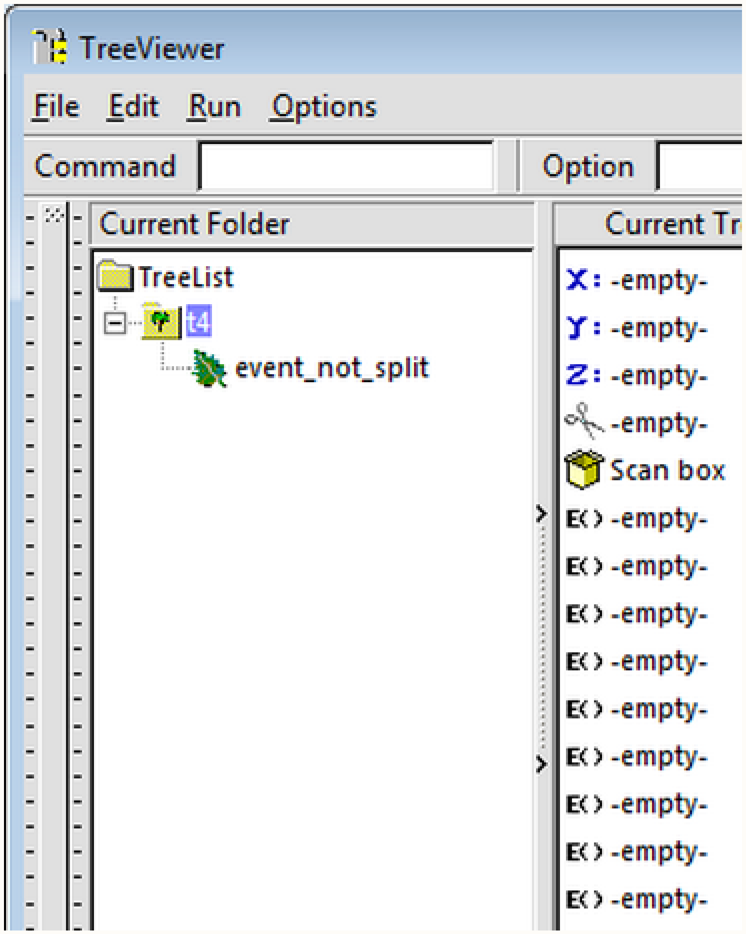
- Trees are designed to be read only
- Often, people want to add branches to existing trees and write their data into it
- Using tree friends is the solution:
  - Create a new file holding the new tree
  - Create a new Tree holding the branches for the user data
  - Fill the tree/branches with user data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

Tree Friends

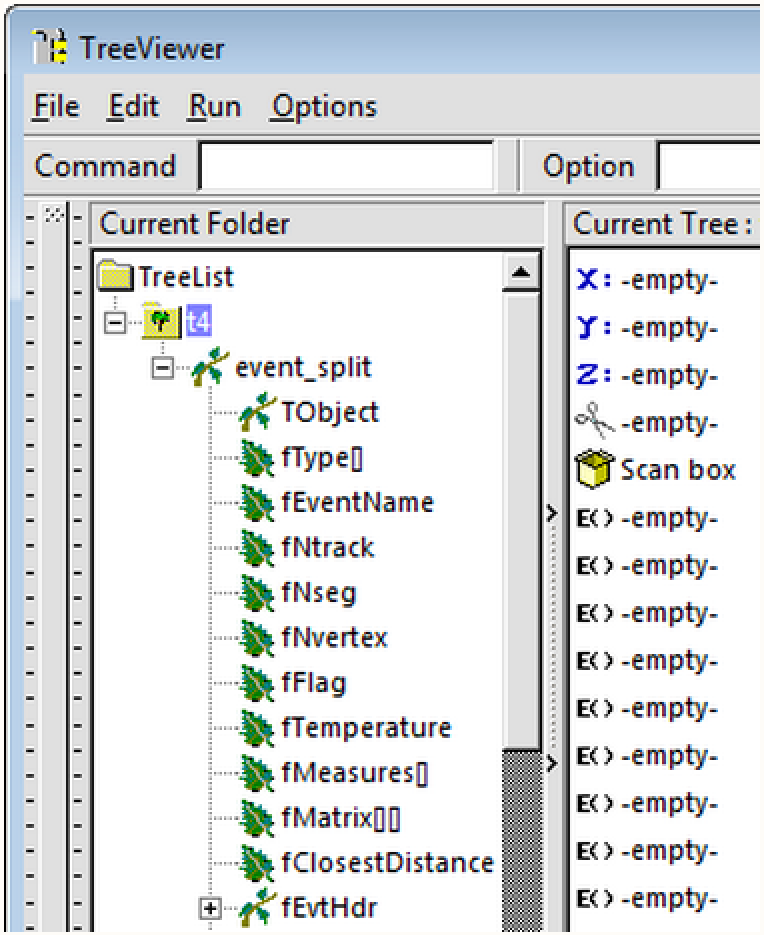



```
TFile f1("tree.root");
tree.AddFriend("tree_1", "tree1.root")
tree.AddFriend("tree_2", "tree2.root");
tree.Draw("x:a", "k<c");
tree.Draw("x:tree_2.x");
```

Splitting



Split level = 0



Split level = 99

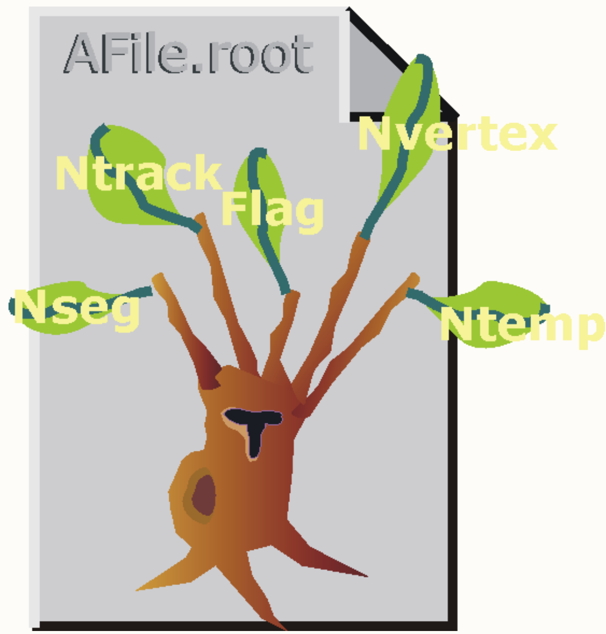
Splitting

- Creates one branch per member – recursively
- Allows to browse objects that are stored in trees, even without their library
- Fine grained branches allow fine-grained I/O - read only members that are needed
- Supports STL containers too, even vector<T*>!

Splitting

Setting the split level (default = 99)

Split level = 0



Split level = 99

```
tree->Branch("EvBr", &event, 64000, 0 );
```

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

Summary: Trees

- TTree 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 (WORM) ideal for experiments' data; use friends to extend
- Branches allow granular access; use splitting to create branch for each member, even through collections

Selectors, Analysis, PROOF

ANALYZING TREES

Recap

TTree efficient storage and access
  for huge amounts of structured data

Allows selective access of data

TTree knows its layout

Almost all HEP analyses based on TTree

TTree Data Access

TSelector: generic "TTree based analysis"

Derive from it ("TMySelector")

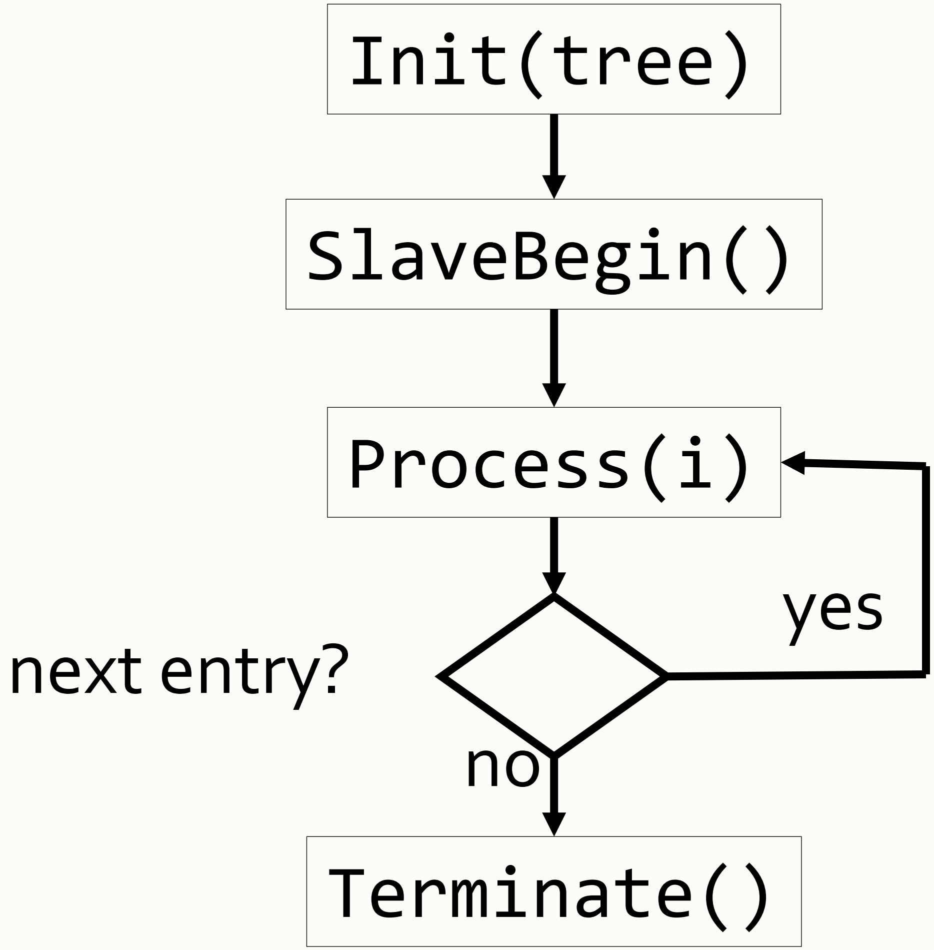
ROOT invokes TSelector's functions,

Used e.g. by tree->**Process**(TSelector*,...), PROOF

Functions called are virtual, thus TMySelector's functions called.

TTree 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()`
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,…

Analysis

TSelector gives the structure of analyses

Content of data analysis:

- science by itself
- covered by Ivica Puljak

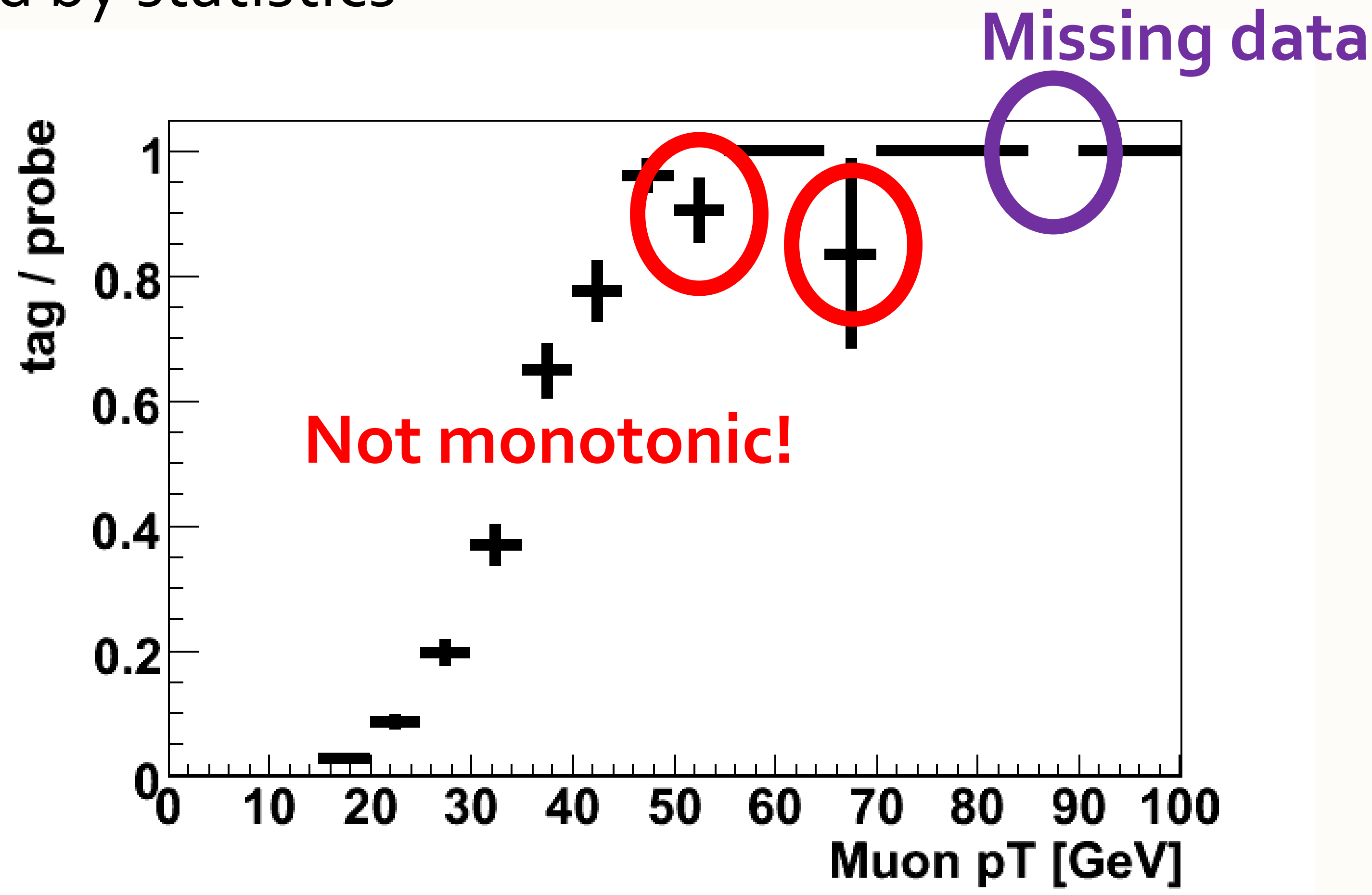
Example for a common ROOT analysis ingredient

FITTING

Fitting

Sampling "known" distribution

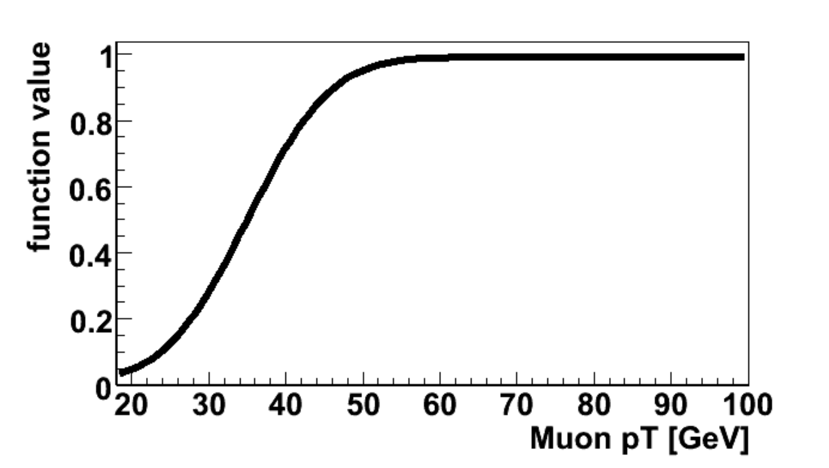
Influenced by statistics



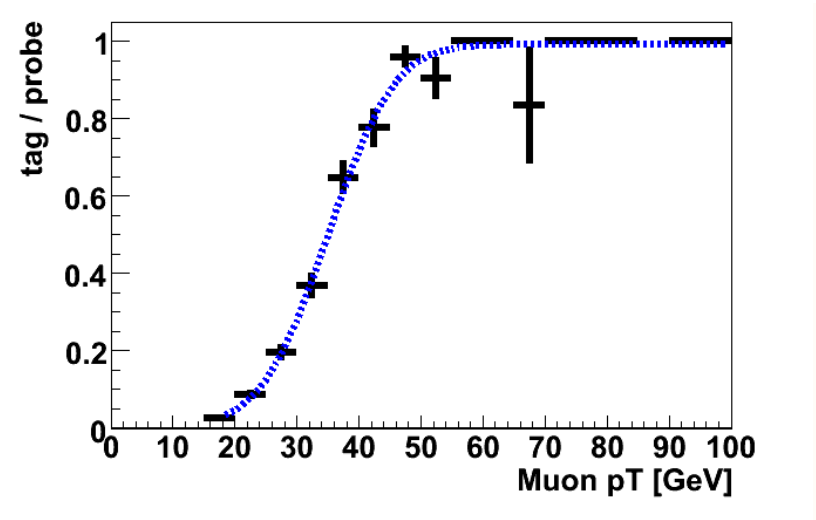
Fit

Combine our knowledge with statistics / data by fitting a distribution:

1. Find appropriate function with parameters



1. Fit function to distribution



Fitting: The Math

Fitting = finding parameters such that

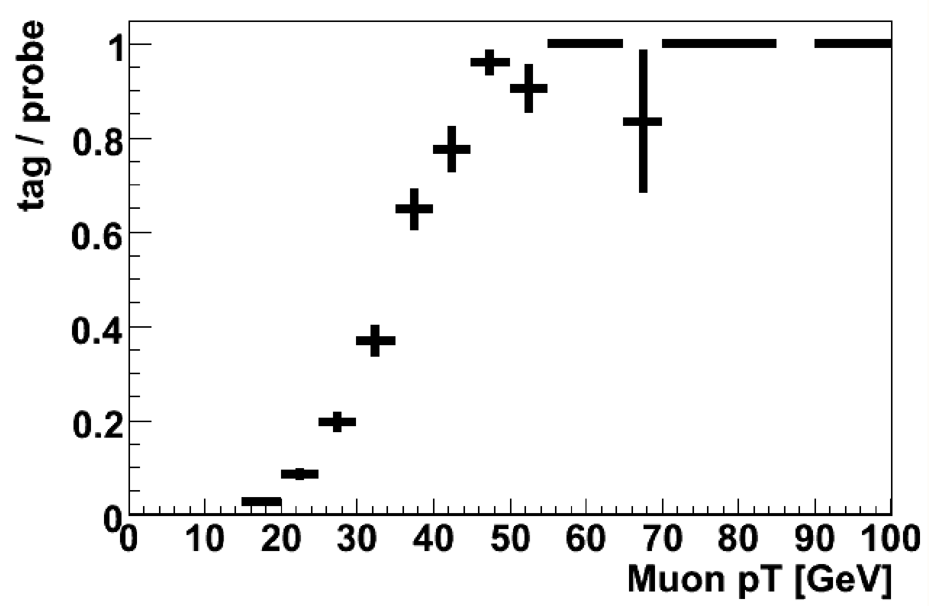
$|f(x) – hist(x)|$

minimal for all points x [or any similar measure]

Histogram with errors:

$|f(x) – hist(x)| / err(x)$

or similar



Fitting: The Function

Finding the proper function involves:

- behavioral analysis:
  starts at 0, goes to constant, monotonic,...
- physics interpretation:
  "E proportional to sin^2(phi)"
- having a good knowledge of typical functions (see TMath)
- finding a good compromise between generalization ("constant") and precision ("polynomial 900th degree")

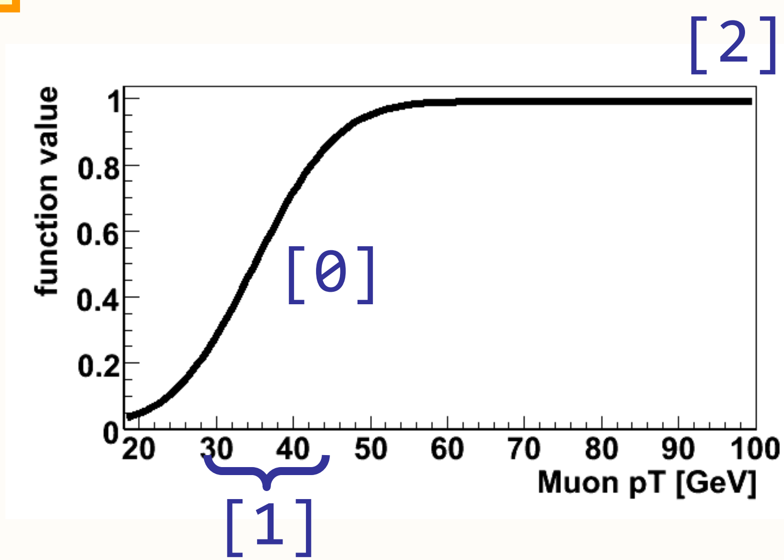
Fitting: Parameters

Let's take "erf"

```
erf(x)/2.+0.5
```

Free parameters:

- [0]: x @ center of the slope
- [1]: ½ width of the slope
- [2]: maximum efficiency

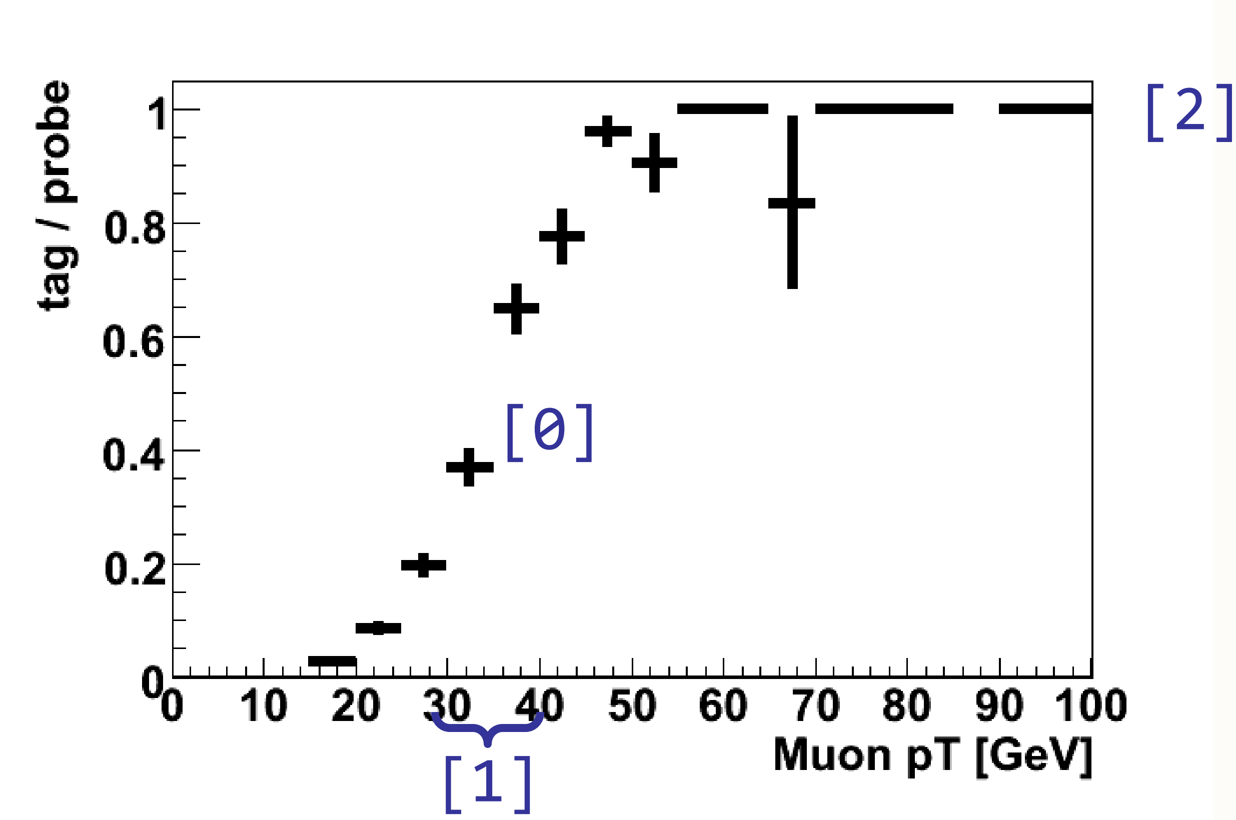


Define fit function:

```
TF1* f = new TF1("myfit",
  "(TMath::Erf((x-[0])/[1])/2.+0.5)*[2]"
  0., 100.);
```

Fitting: Parameter Init

A must!



Sensible values:

```
f->SetParameter(0, 35.);
f->SetParameter(1, 10.);
f->SetParameter(2,  1.);
```

Fitting Result

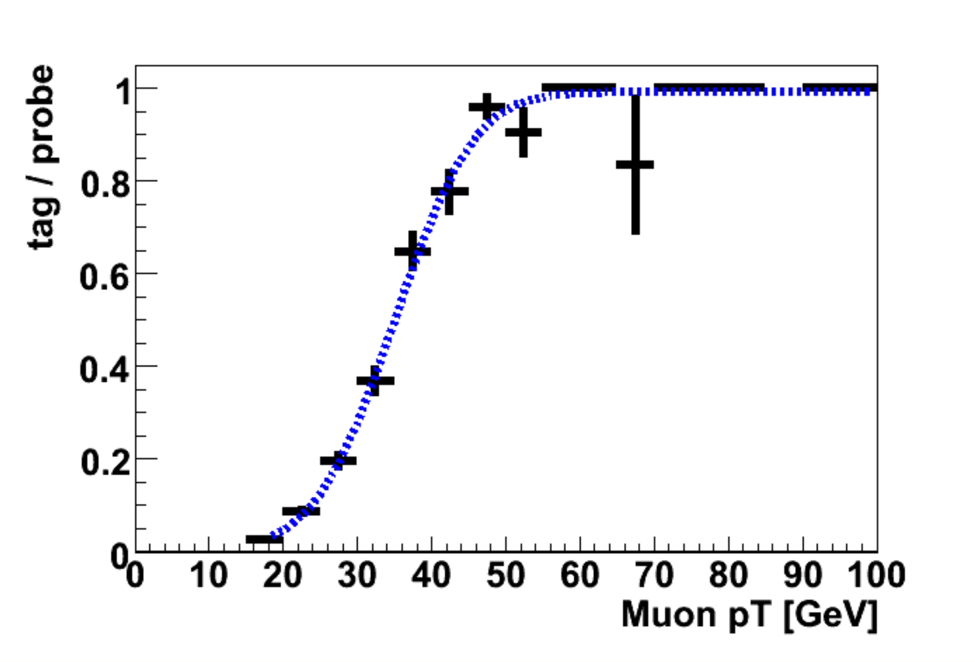
Result of `hist->Fit(f);` is printed, or use

`f->GetParameter(0)`

[0]: 34.9

[1]: 12.1

[2]: 0.98



which means:

`(TMath::Erf((x-34.9)/12.1)/2.+0.5)*0.98`

Get efficiency at pT=42GeV: `f->Eval(42.)`

Fitting: Recap

You now know

- why large samples are relevant
- what fitting is, how it works, when to do it, and how it's done with ROOT.

Bleeding Edge Physics
with
Bleeding Edge Computing

INTERACTIVE DATA ANALYSIS WITH PROOF

Parallel Analysis: PROOF

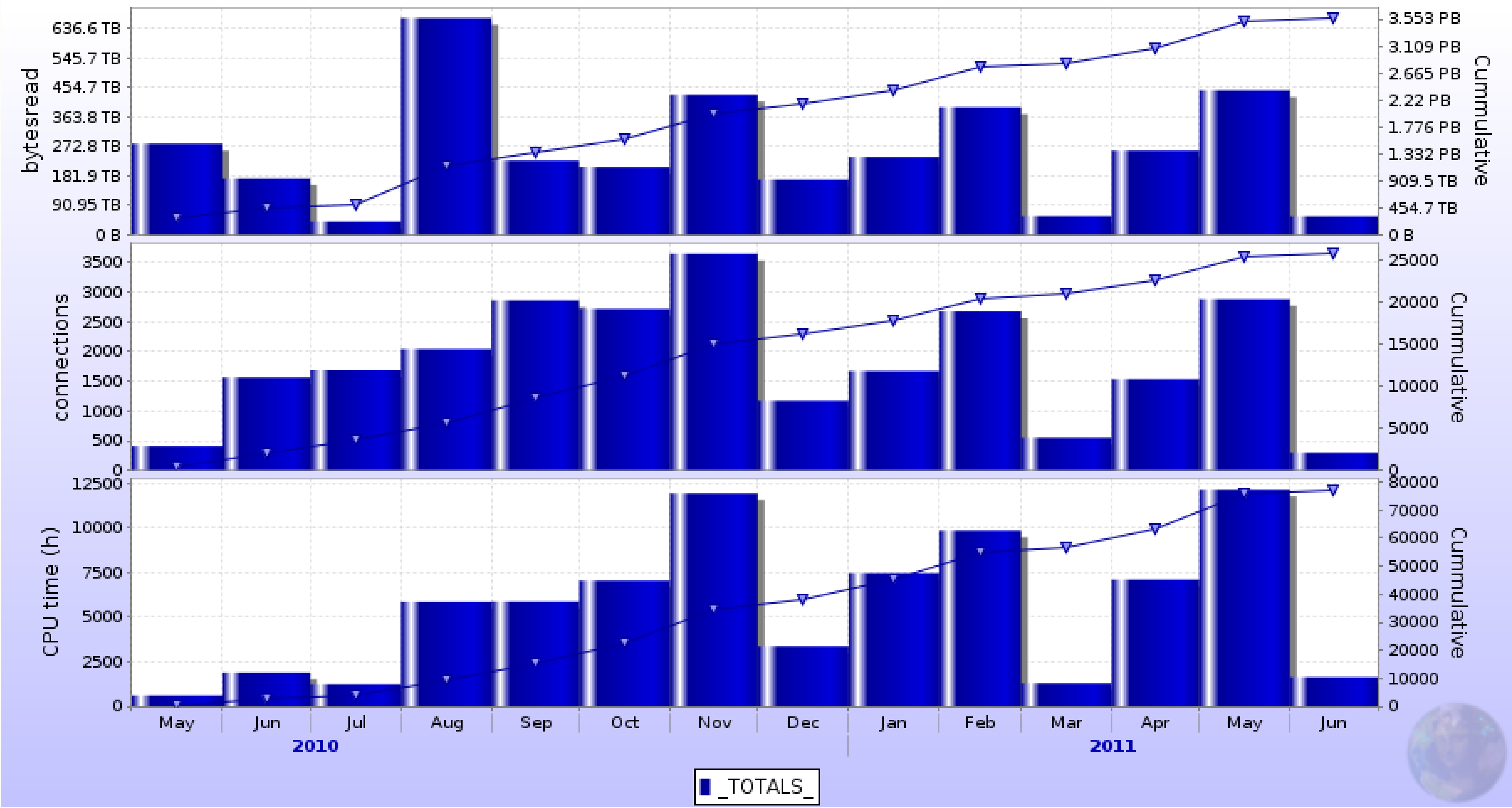
Some numbers (from Alice experiment)

- 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

CAF Usage Statistics



PROOF

Huge amounts of events, hundreds of CPUs

Split the job into N events / CPU!

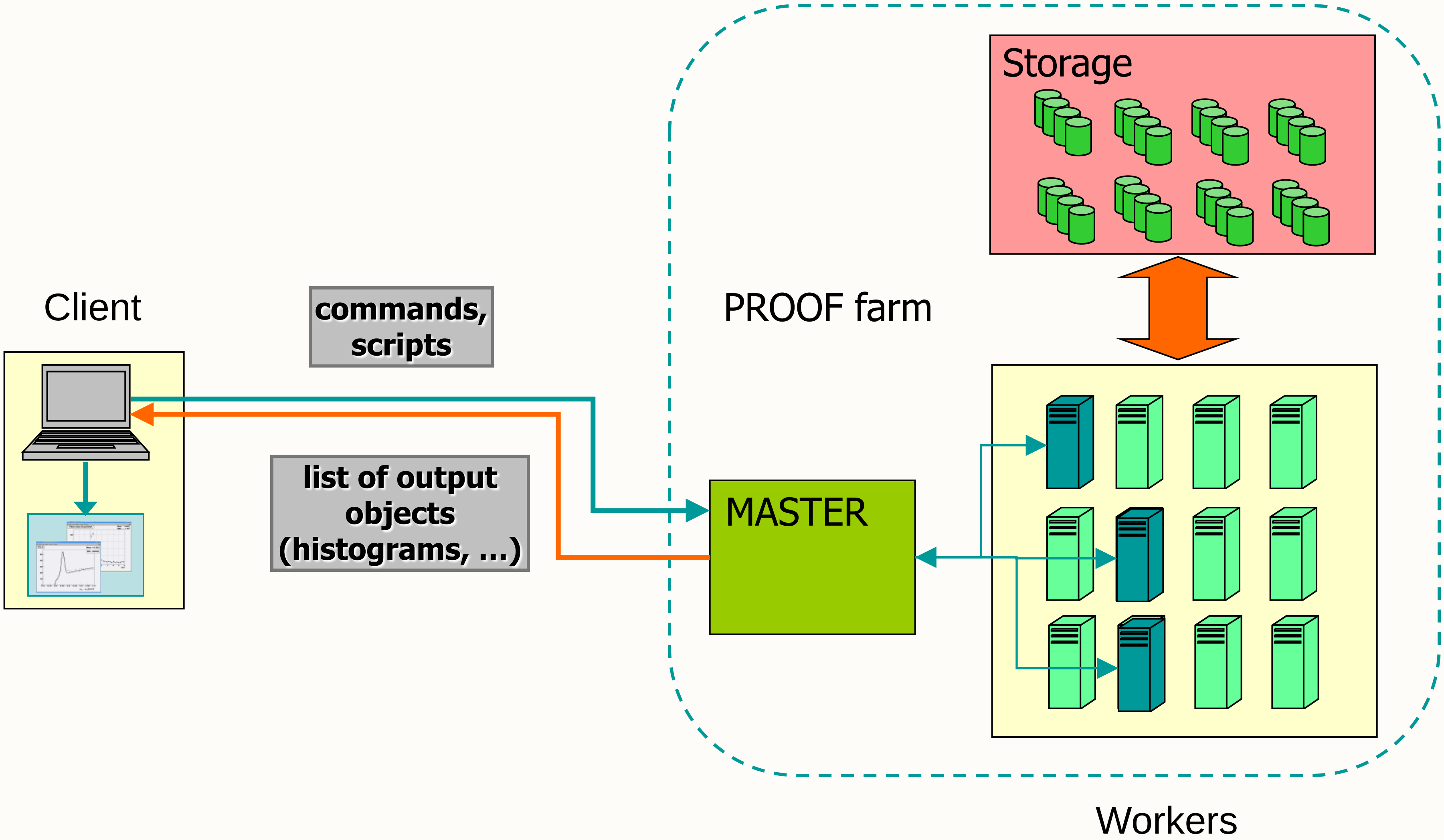
PROOF for TSelector based analysis:

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

Interactive!

- Start analysis
- Watch status while running
- Forgot to create a histogram?
  - Interrupt the process
  - Modify the selector
  - Re-start the analysis
- More dynamic than a batch system

PROOF


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

Scheduling

- Decides where to run which (part of) the jobs
- E.g. simple batch system
- Can autonomously split jobs into parts (“packets”)
- Involves
  - resource management (CPU, I/O, memory)
  - data locality
  - priorities (jobs / users)
  - and whatever other criteria are deemed relevant
- Often optimizing jobs’ distribution towards overall goal: maximum CPU utilization (Grid), minimum time to result (PROOF)

Packetizer Role and Goals

- Distributes units of work ("packets") to workers
- Grid's packet: >=1 file
- Result arrives when last resource has processed last file:

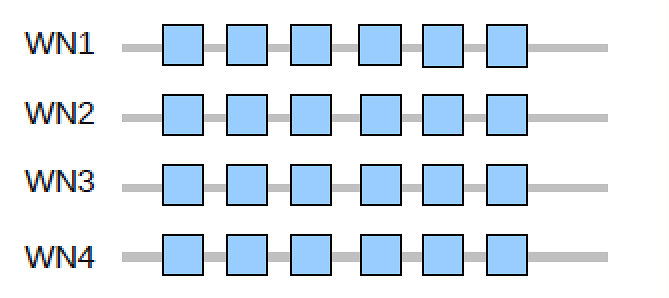
$$t = t_{init} + \max_{jobs}(R_i \cdot N_i^{files}) + t_{final}$$

t_{init}, t_{final}: time to initialize / finalize the jobs
R_i: processing rate of job i
N_i^{files}: number of files for job i

- Result:
  - slowest job defines running time
  - large tail in CPU utilization versus time

Static…

- Example: 24 files on 4 worker nodes, one under-performing



The slowest worker node sets the processing time

PROOF's Dynamic Packetizer

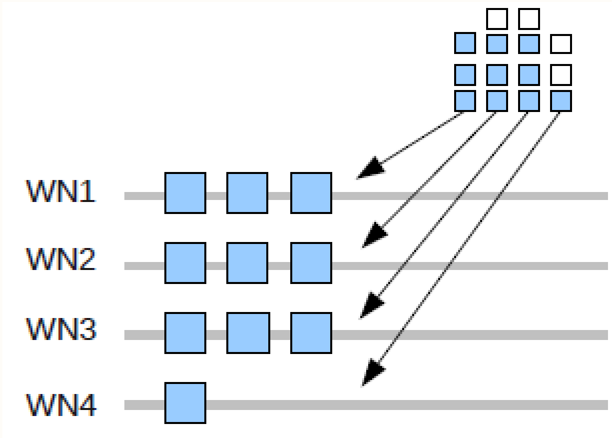
- PROOF packetizer's goal: results as early as possible
- All workers should finish at the same time:

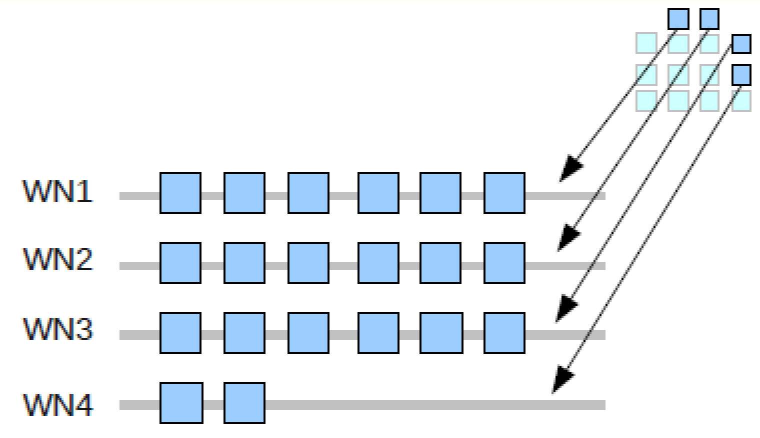
$$t = t_{init} + max_{jobs}(R_i \cdot N_i^{files}) + t_{final}$$

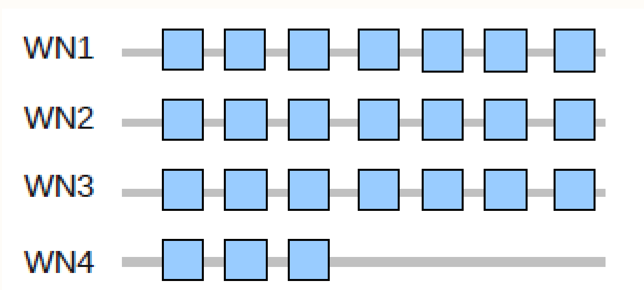
ideally, $R_i \cdot N_i^{files}$ equal for all jobs

- Cannot reliably predict performance (R_i) of workers
  - job interaction, e.g. number of jobs accessing the same disk
  - CPU versus I/O duty of jobs
- Instead: update prediction based on real-time past performance while running
- Pull architecture: workers ask for new packets

Dynamic Packet Distribution

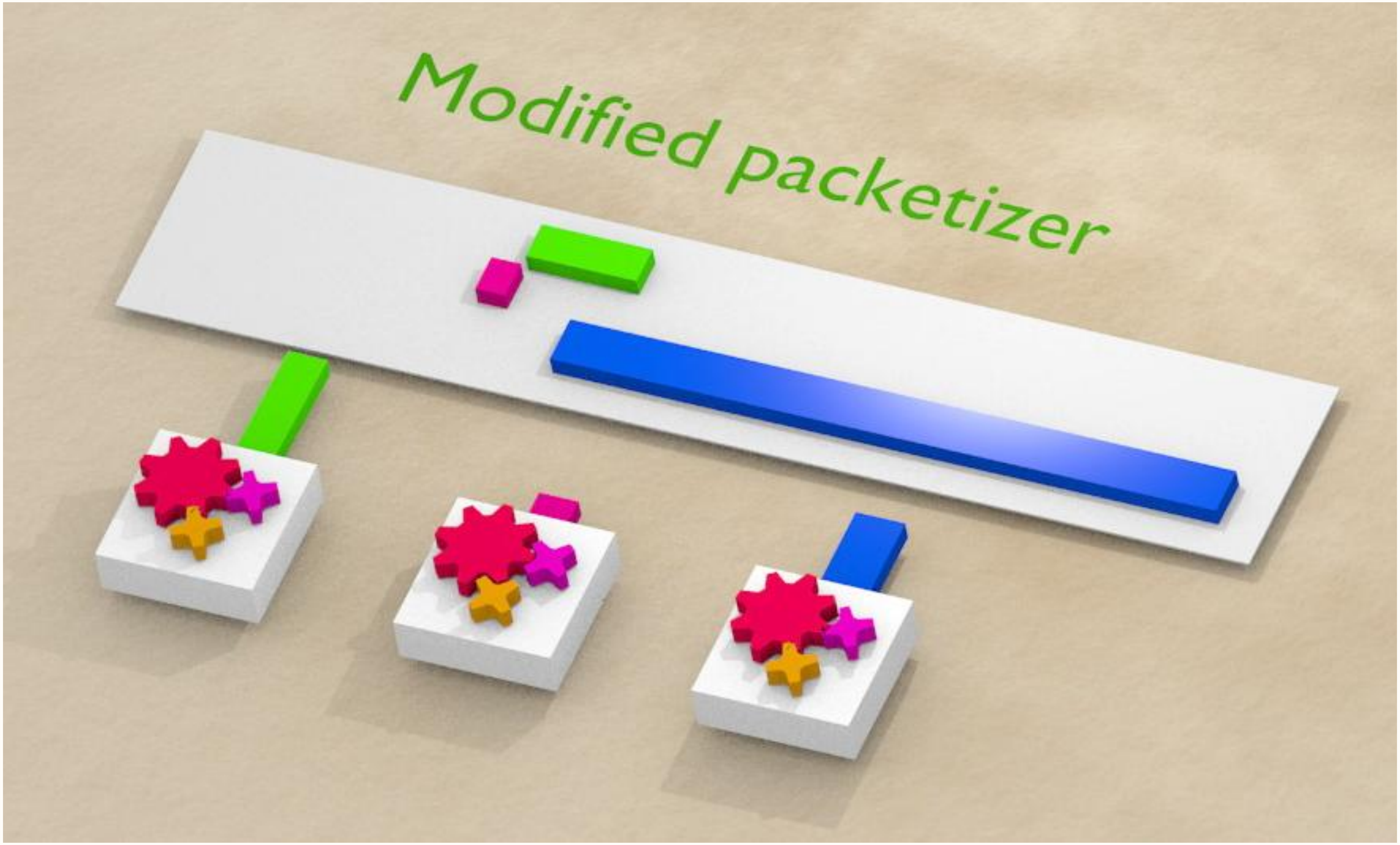






The slowest worker node gets less work to do: the processing time is less affected by its under performance

PROOF Packetizer Live



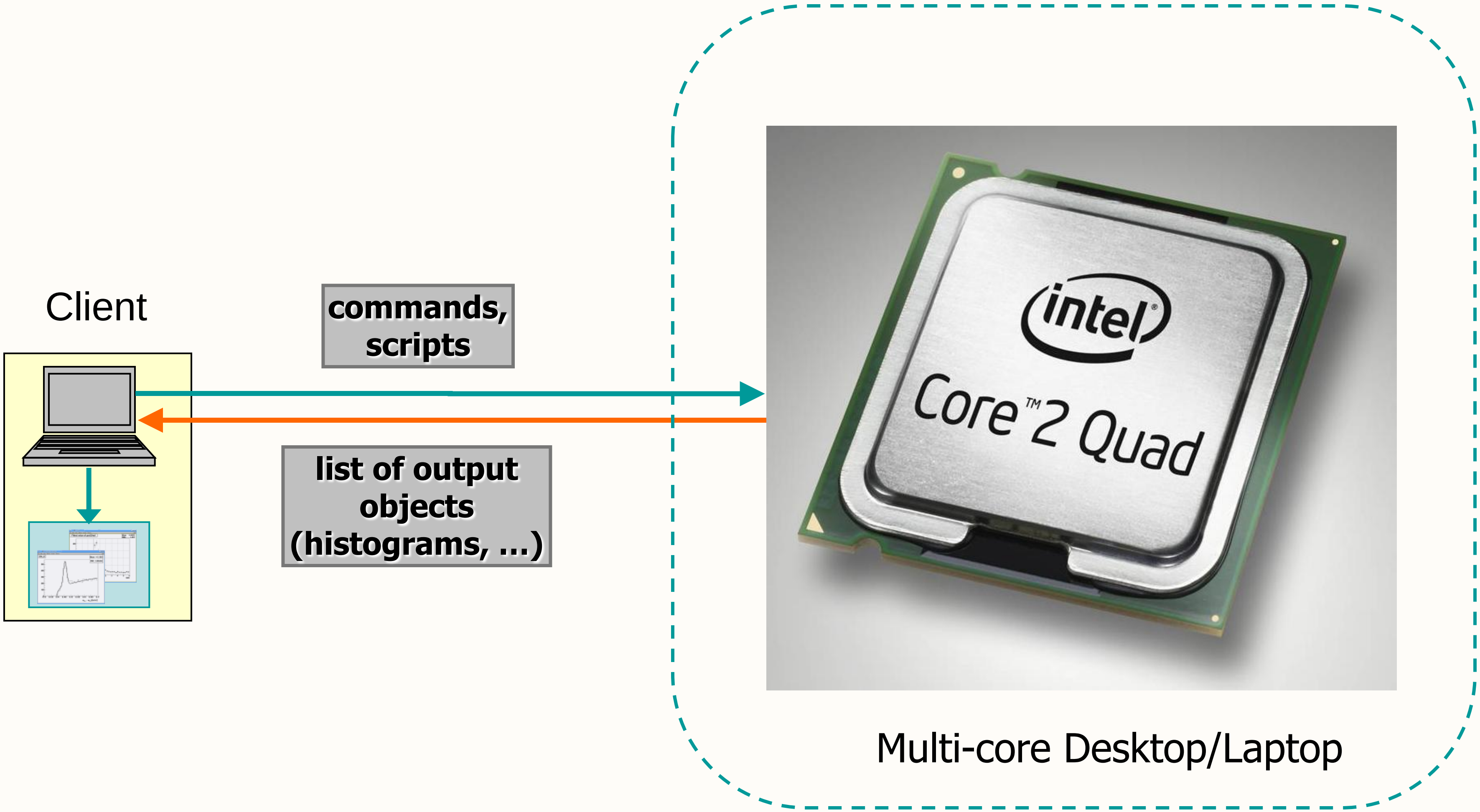
Creating a session

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

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

where "master" is the hostname of the master machine on the PROOF cluster

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

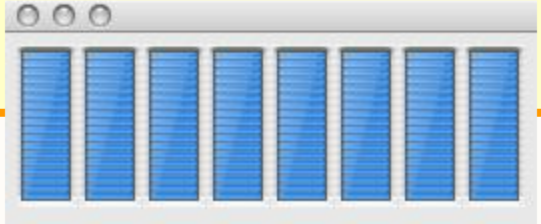
- Example of local TChain analysis

```
// 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

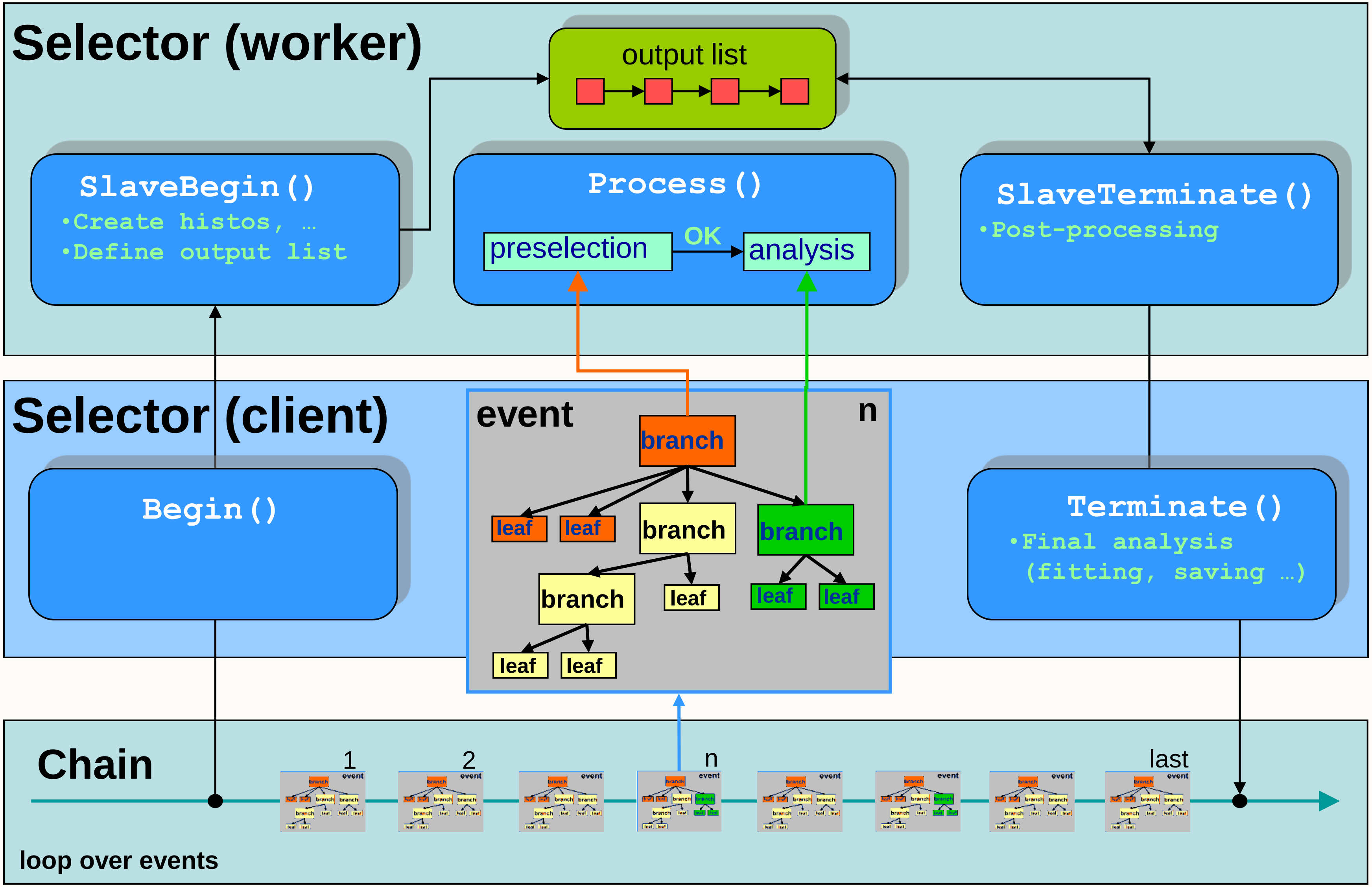
```
// Create a chain of trees
root[0] TChain *c = new TChain("myTree");
root[1] c->Add("http://www.any.where/file1.root");
root[2] c->Add("http://www.any.where/file2.root");

// 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, ...

PROOF Analysis



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
}
```

Results

At the end of Process(), the output list is accessible via gProof->GetOutputList()

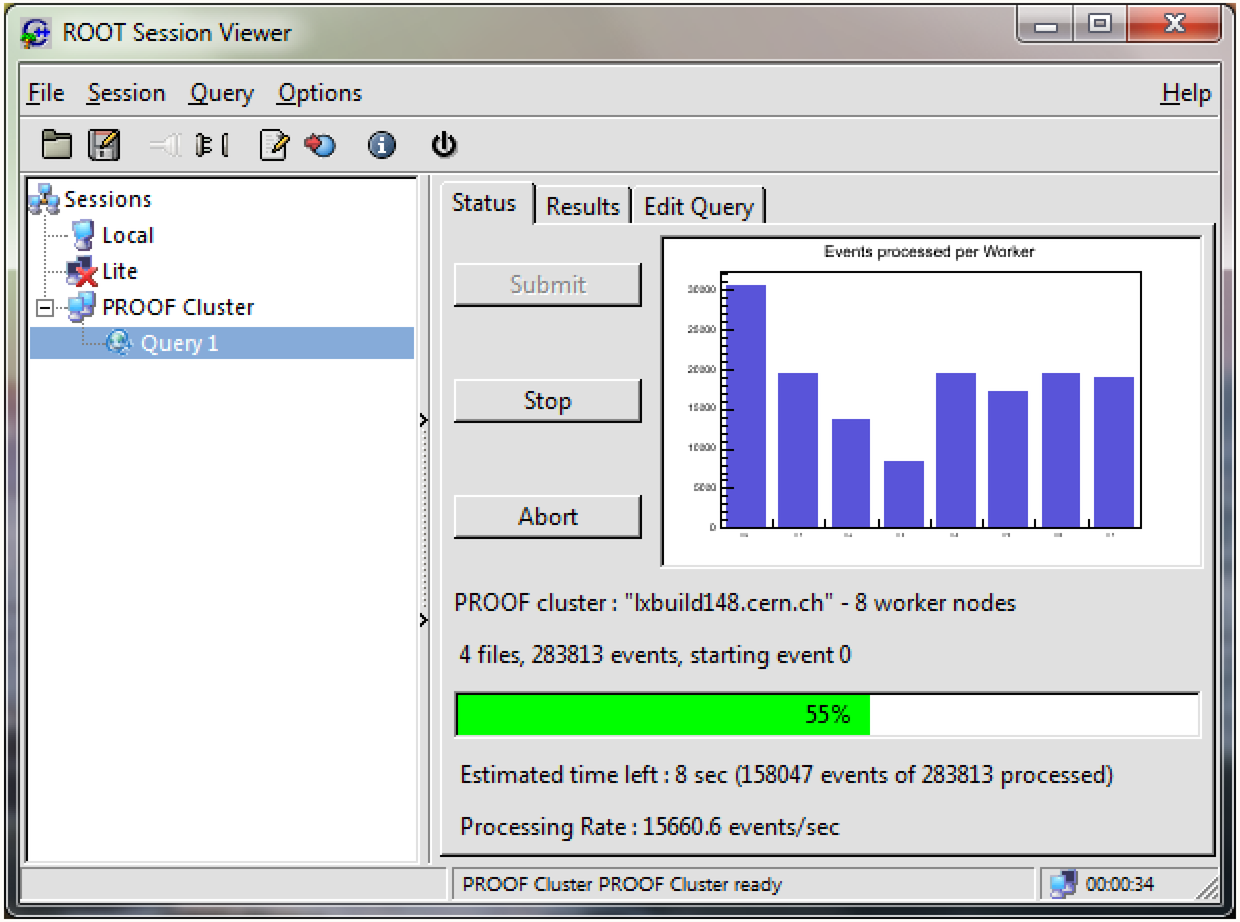
```
// Get the output list
root[0] TList *output = gProof->GetOutputList();
// Retrieve 2D histogram "h2"
root[1] TH2F *h2 = (TH2F*)output->FindObject("h2");
// Display the histogram
root[2] h2->Draw();
```

PROOF GUI Session

Starting a PROOF GUI session is trivial:

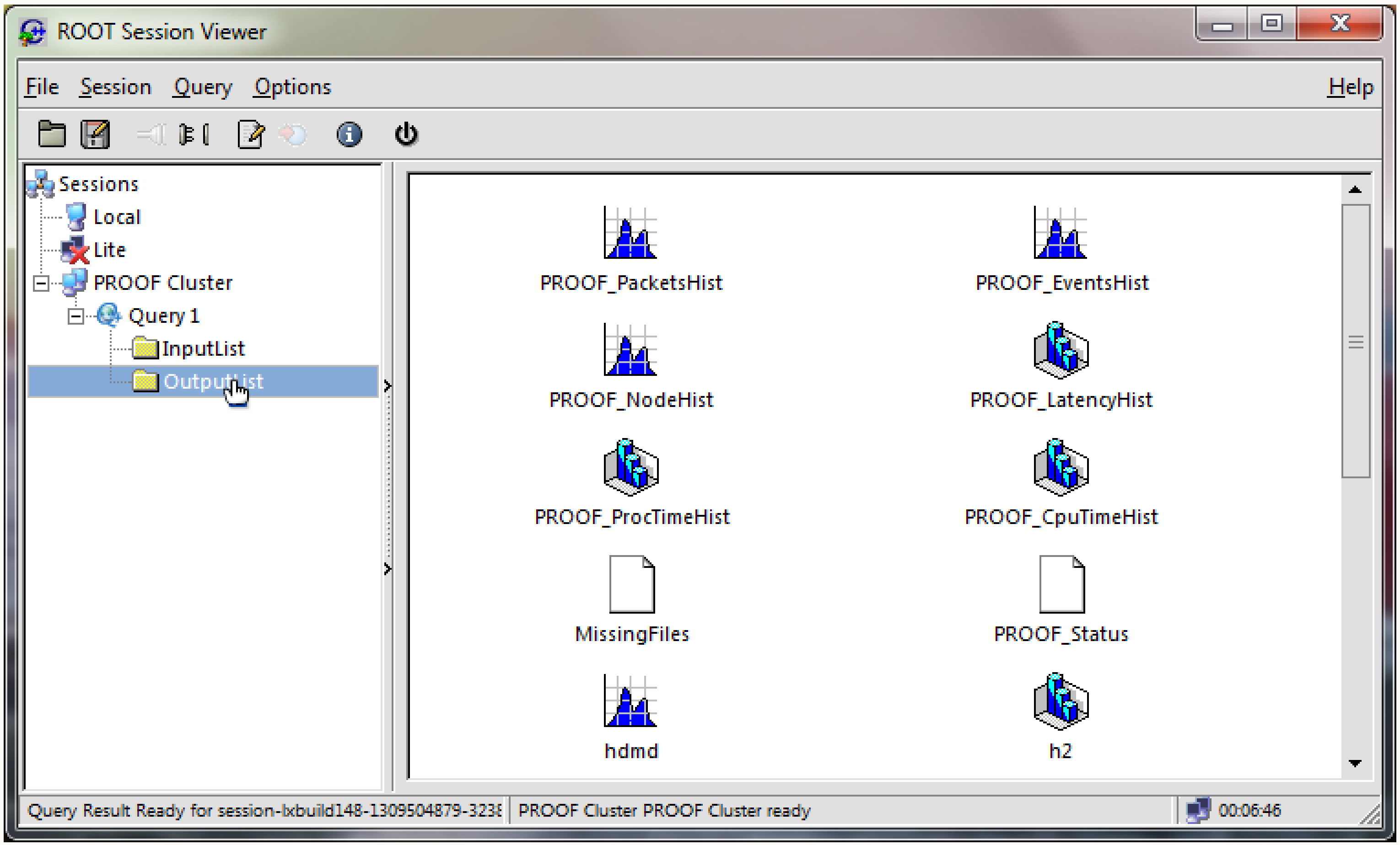
```
TProof::Open()
```

Opens GUI:



PROOF GUI Session – Results

Results accessible via TSessionViewer, too:



PROOF Documentation

Documentation available online at

http://root.cern.ch/drupal/content/proof

But of course you need a little cluster of CPUs

Like your multi-core game console!

Summary

You've learned:

- analyzing a TTree can be easy and efficient
- integral part of physics is counting
- ROOT provides histogramming and fitting
- > 1 CPU: use PROOF!

Looking forward to hearing from you:

- as a user (help! bug! suggestion!)
- and as a developer!