Parallel Interactive and Batch HEP-Data Analysis with PROOF

Maarten Ballintijn*, Marek Biskup**,
Rene Brun**, Philippe Canal***,
Derek Feichtinger****, Gerardo Ganis**,
Guenter Kickinger**, Andreas Peters**,
Fons Rademakers**

* - MIT ** - CERN *** - FNAL **** - PSI

Outline

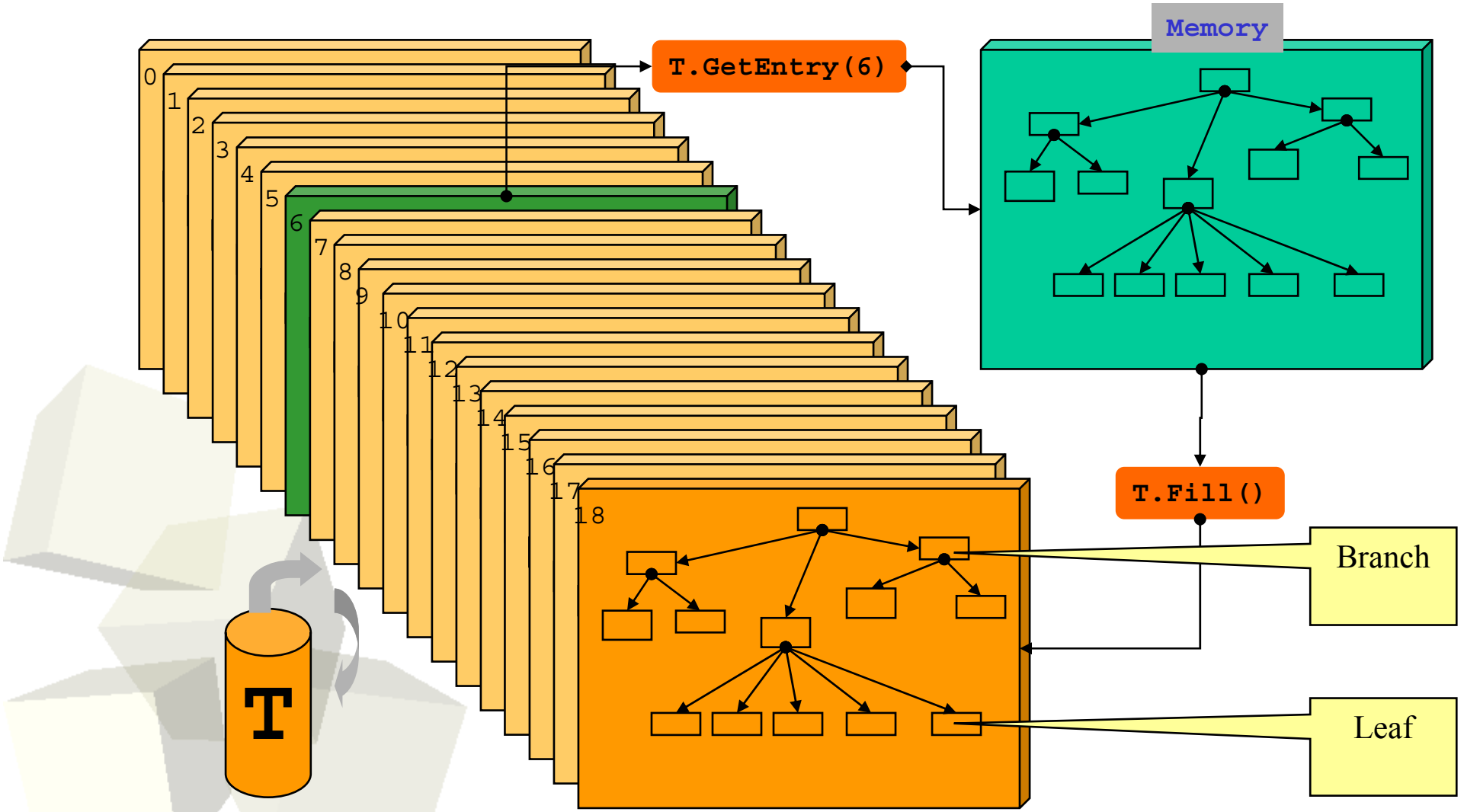
- Data analysis model of ROOT
- Overview of PROOF
- Recent developments
- Future plans - Interactive-Batch data analysis

ROOT Trees

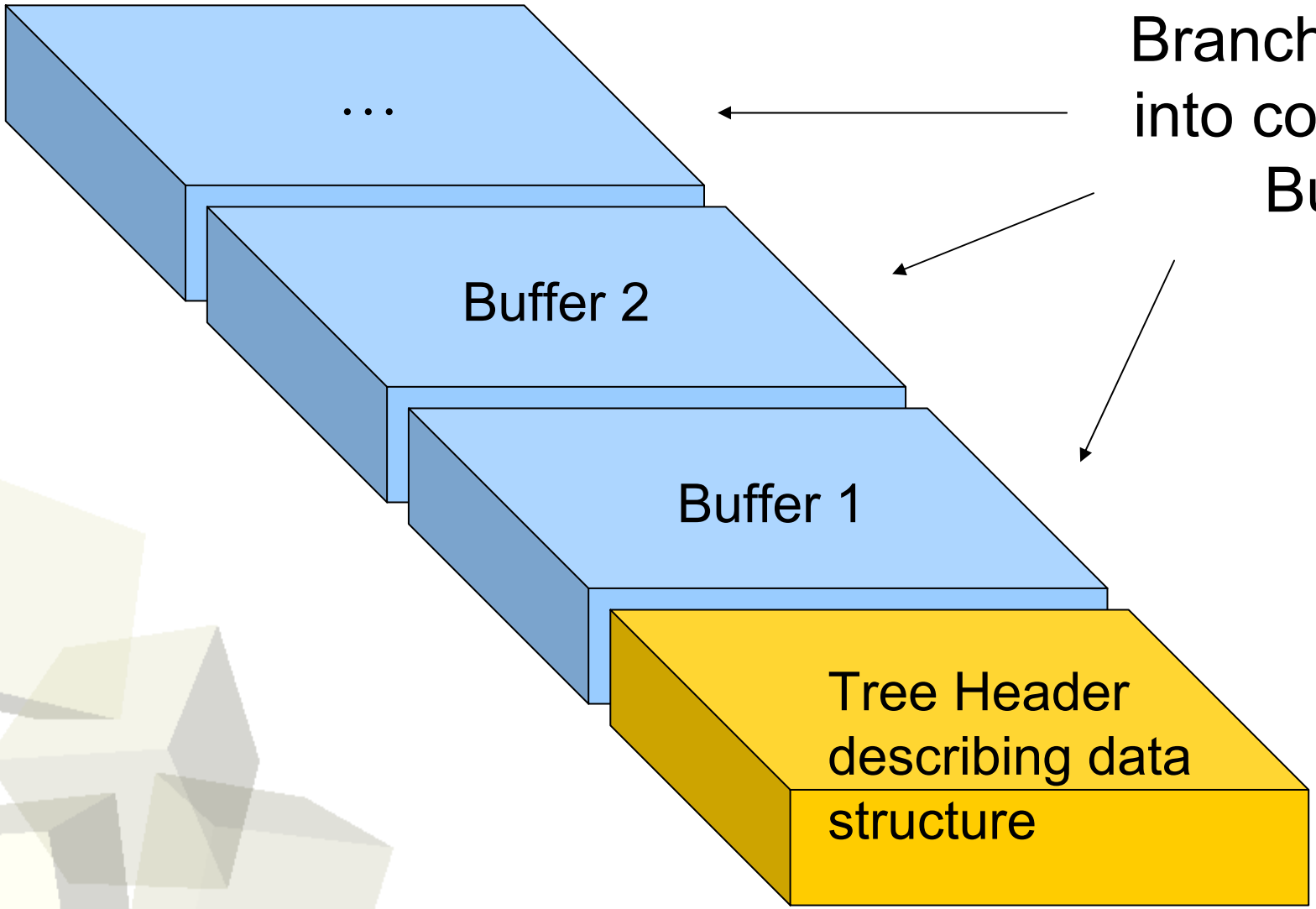
- Tree – main data structure of ROOT
- Set of records (entries)
- Record may contain basic C types (int, double, arrays) and any C++ object, polymorphic object, collection, stl collection, etc, e.g.:
  - stl::list<TrackClass> tracks;
  - Electrons
    - Int_t NoElectrons;
    - Double_t Momentum[NoElectrons][4];
    - Float_t Position[NoElectrons][4];
  - Muons
    - Int_t NoMuons;
    - …
- Provide efficient access to partial entry data
- Typical size < 2GB

Trees in memory and in files

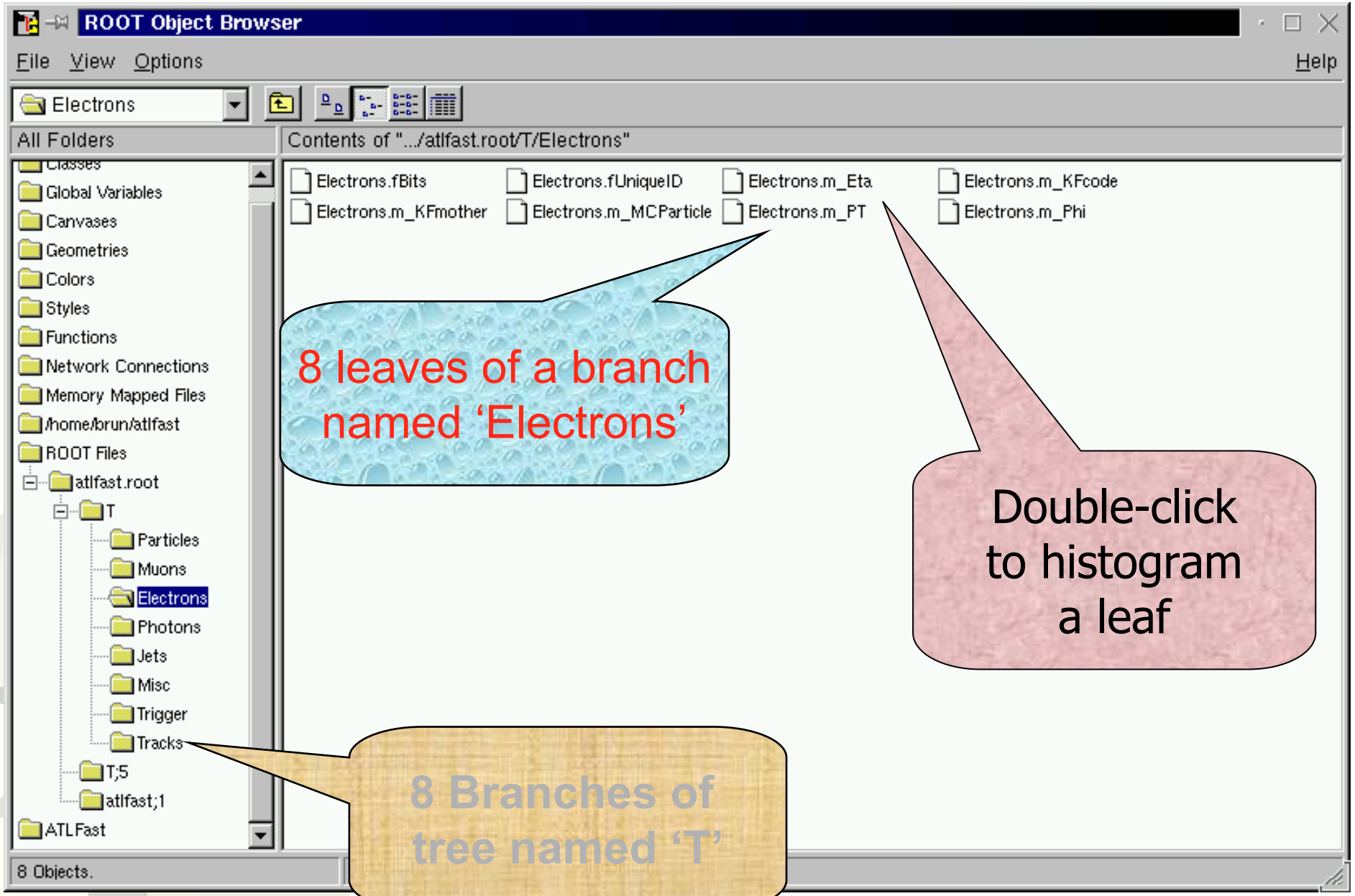
Each **Leaf** is an object (c++ object, array, basic type).
Each **Branch** groups several Leafs/Branches.



Tree data storage on disk

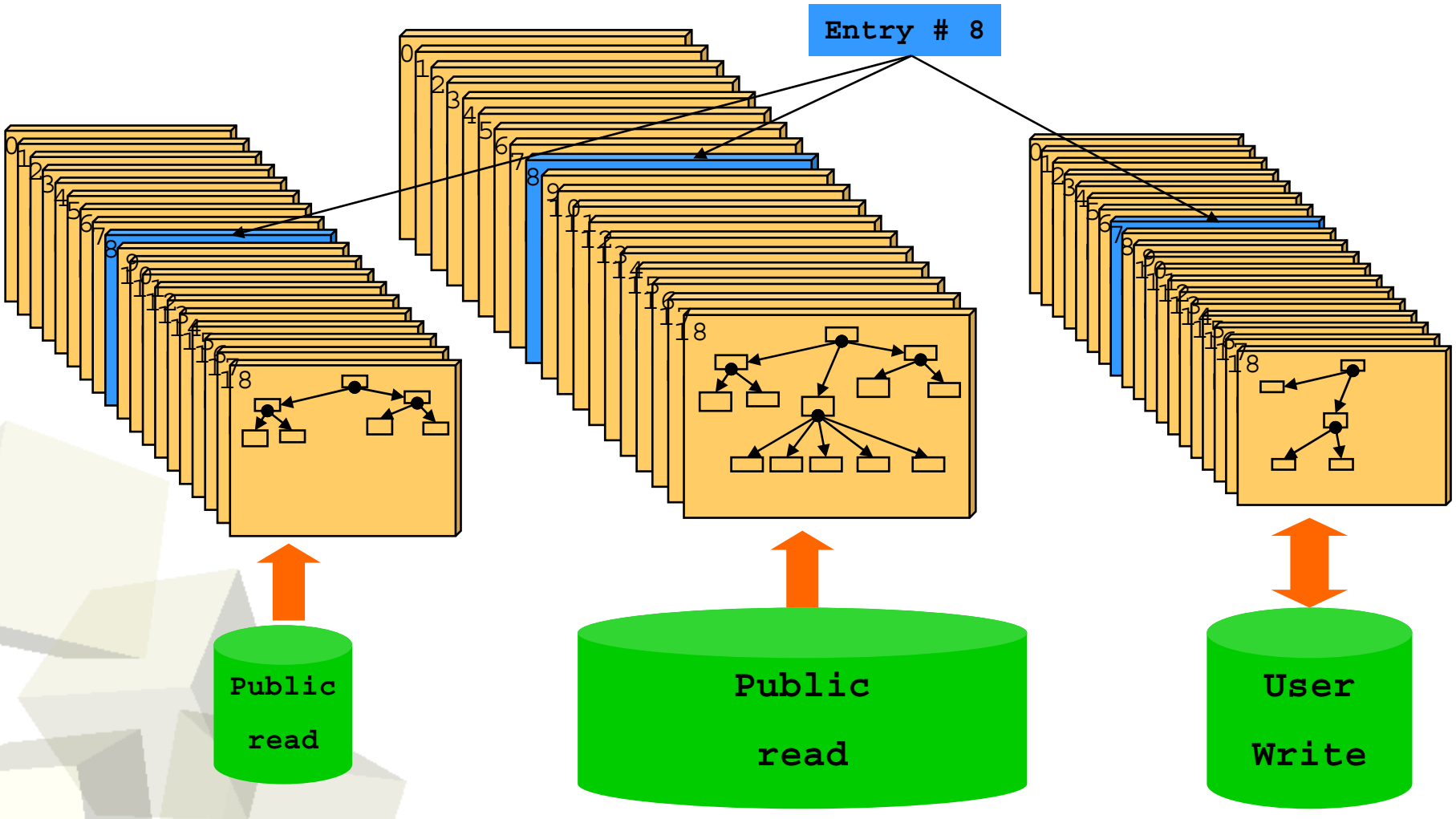


ROOT Trees - GUI



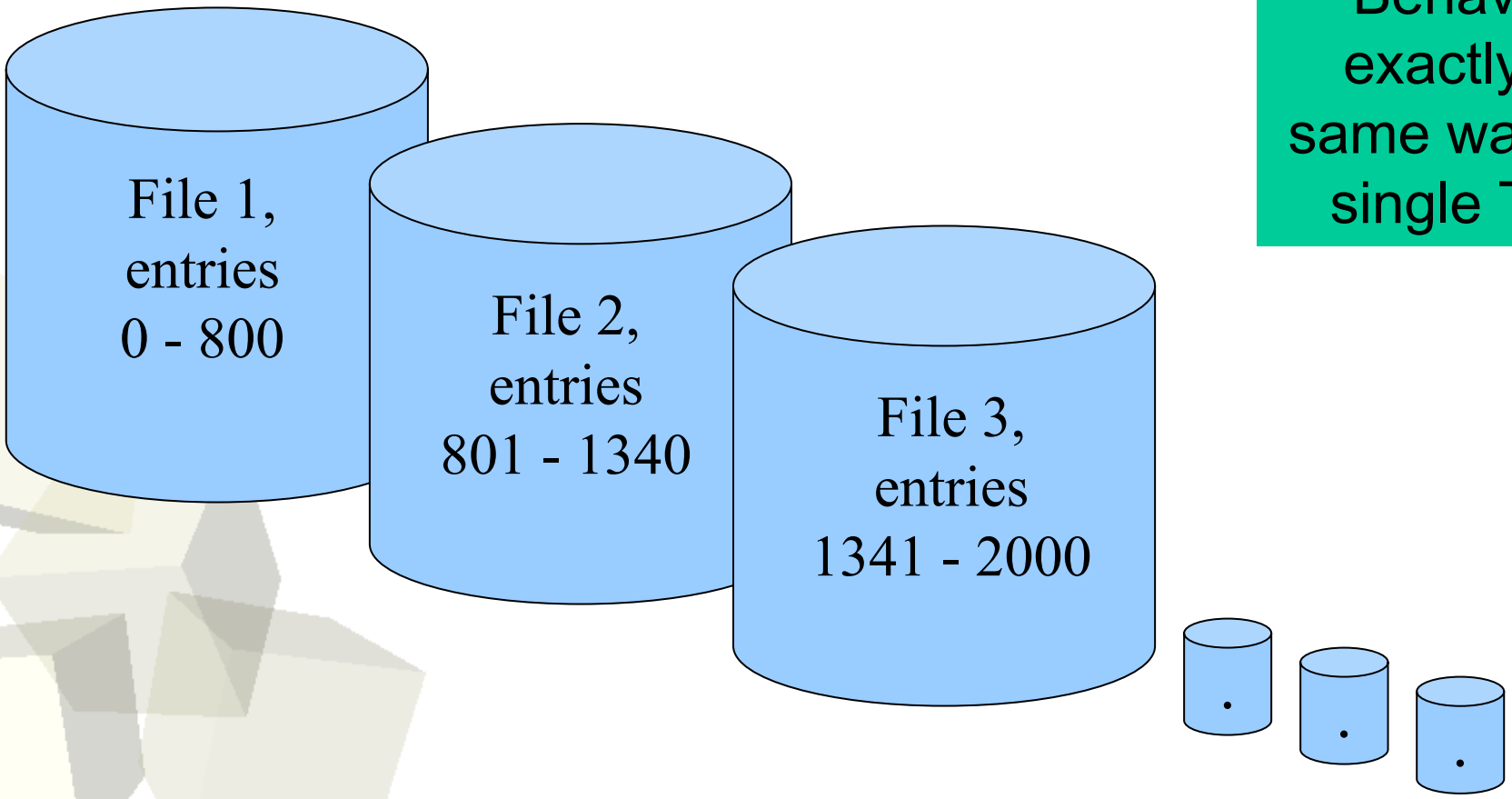
Tree Friends

Behave in exactly the same way as a single Tree!



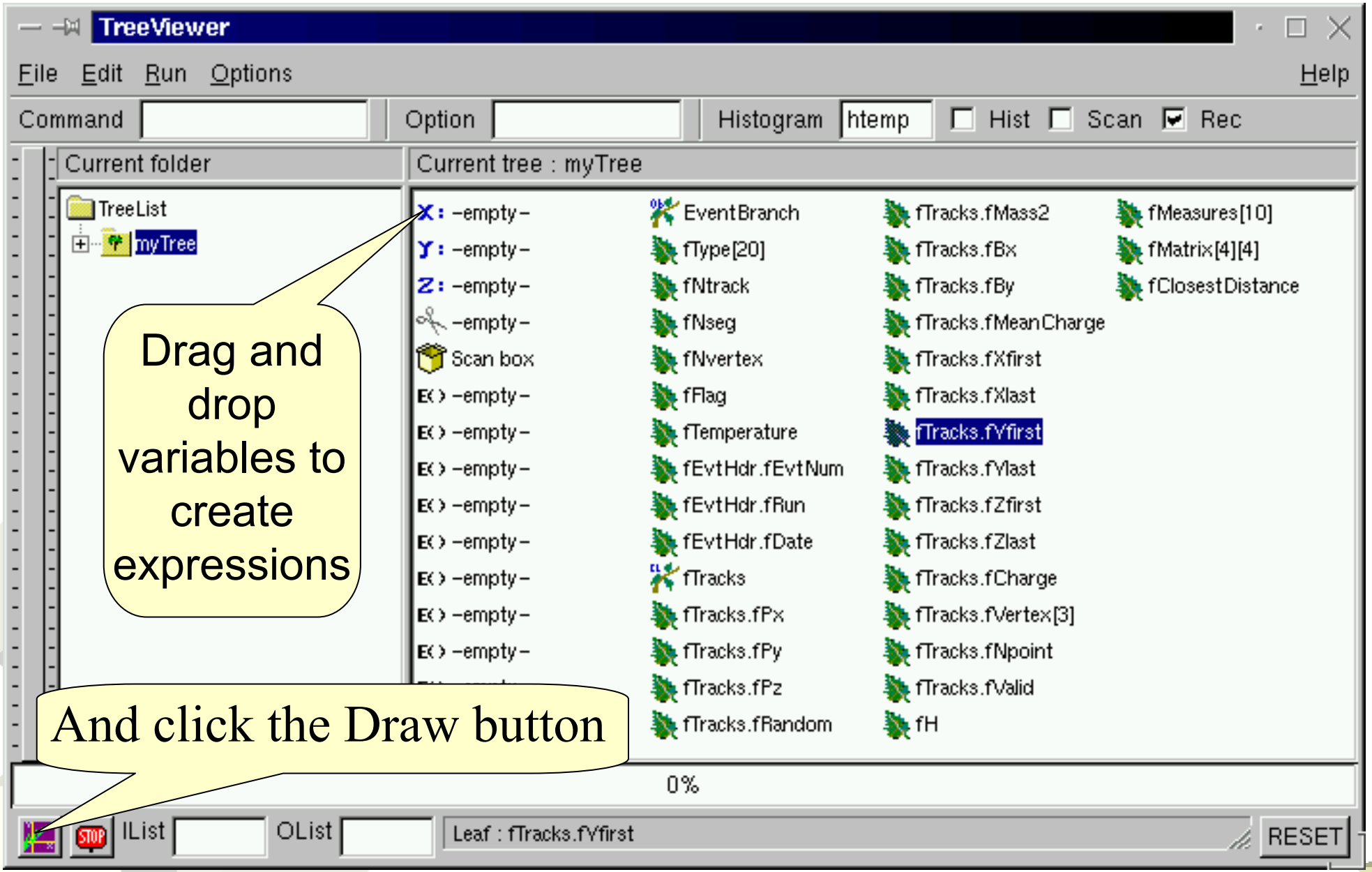
ROOT Chains

- A typical Tree: < 2GB – you can process it on your laptop
- Chain – list of trees
  - e.g. 1000 files – the processing takes long time!



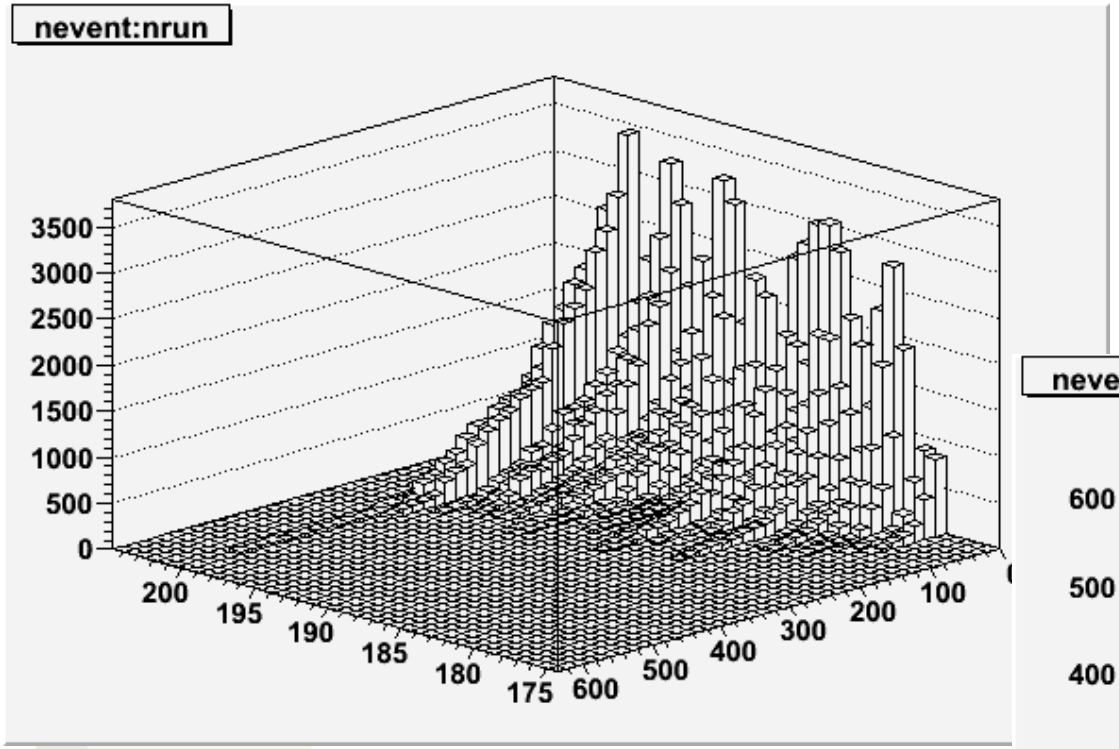
Behave in exactly the same way as a single Tree!

Tree Viewer

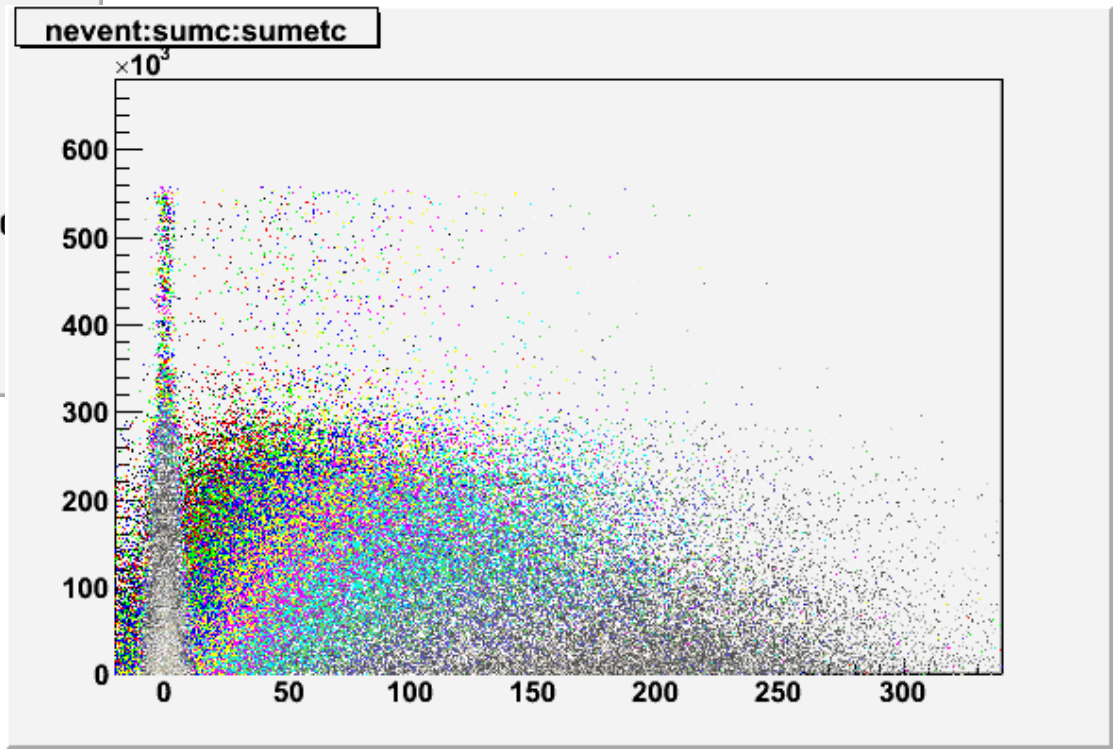

TreeViewer
File Edit Run Options Help
Command
Option
Histogram htemp
Hist Scan Rec
Current folder
Current tree : myTree
TreeList
myTree
X: -empty-
Y: -empty-
Z: -empty-
-empty-
Scan box
-empty-
-empty-
-empty-
-empty-
-empty-
-empty-
-empty-
-empty-
EventBranch
fType[20]
fNtrack
fNseg
fNvertex
fFlag
fTemperature
fEvtHdr.fEvtNum
fEvtHdr.fRun
fEvtHdr.fDate
fTracks
fTracks.fPx
fTracks.fPy
fTracks.fPz
fTracks.fRandom
fTracks.fMass2
fTracks.fBx
fTracks.fBy
fTracks.fMeanCharge
fTracks.fXfirst
fTracks.fXlast
fTracks.fYfirst
fTracks.fYlast
fTracks.fZfirst
fTracks.fZlast
fTracks.fCharge
fTracks.fVertex[3]
fTracks.fNpoint
fTracks.fValid
fH
fMeasures[10]
fMatrix[4][4]
fClosestDistance
Drag and drop variables to create expressions
And click the Draw button
0%
STOP
IList
OList
Leaf : fTracks.fYfirst
RESET

Chain.Draw()

chain.Draw() is a function called by the GUI for drawing



chain.Draw(“nevent:nrun”, “”, “lego”);



chain.Draw(“sumetc:nevent:nrun”, “”, “col”);

Advanced data processing

- Preprocessing and initialization
- Processing each entry (loop over all files and entries in each file)
- Post processing and clean-up

Do not assume anything about the order in which Process() is called for different entries!

Selectors contain only the functions important for processing

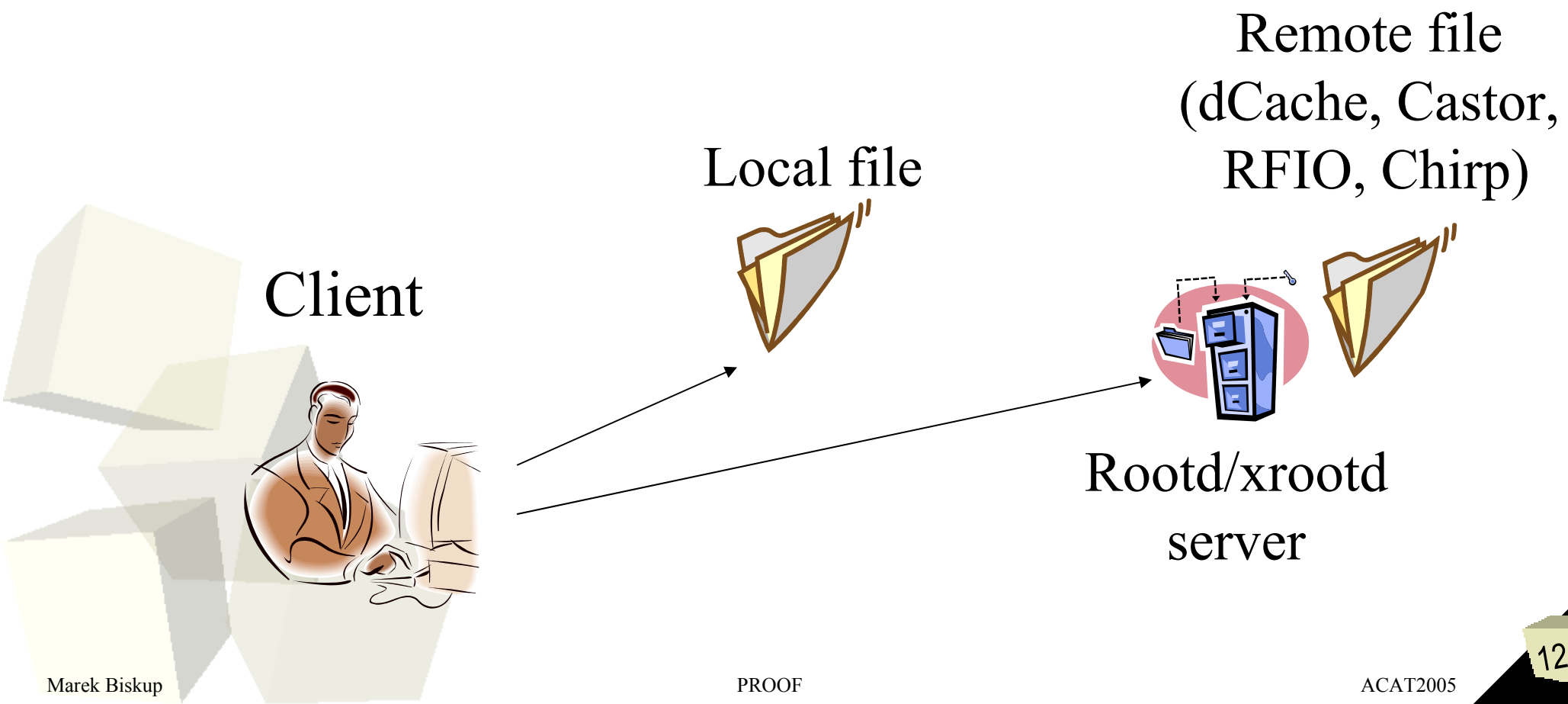
Terminal

```
void MySelector::Begin(TTree *tree)
{  // function called before starting the event loop
   fPtBranch = tree->GetBranch("Pt")
   fPtBranch->SetAddress(&fPt);
   fMyHist = new TH1("Pt", "Pt");
}
Bool_t MySelector::Process(Long64_t entry)
{   // entry is the entry number in the current Tree
   fPtBranch->GetEntry(entry);
   fMyHist->Fill(fPt);
}
void MySelector::Terminate()
{   // function called at the end of the event loop
   fMyHist->Draw();
}
```

We read only one branch

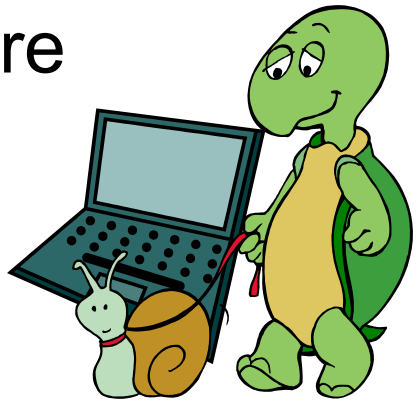
ROOT standard model

- Files analyzed on a local computer
- Remote data accessed via remote fileserver (rootd/xrootd)



Normal Laptop/PC can process up to 10MB/s.
Current experiments and LHC need much more

Data transfer takes time.



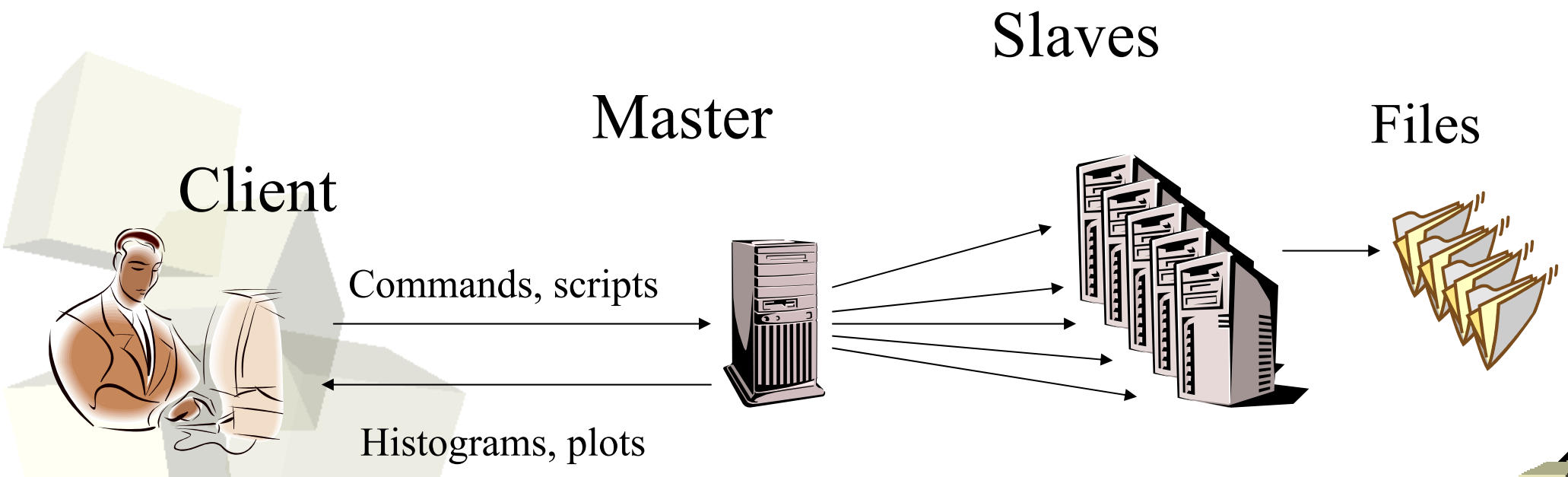
Bring the KiloBytes to the PetaBytes and not the PetaBytes to the KiloBytes

- Parallel interactive analysis of ROOT Data
- Using the same ROOT Selectors (transparency!)
- Execution on clusters of heterogeneous computers (scalability!)

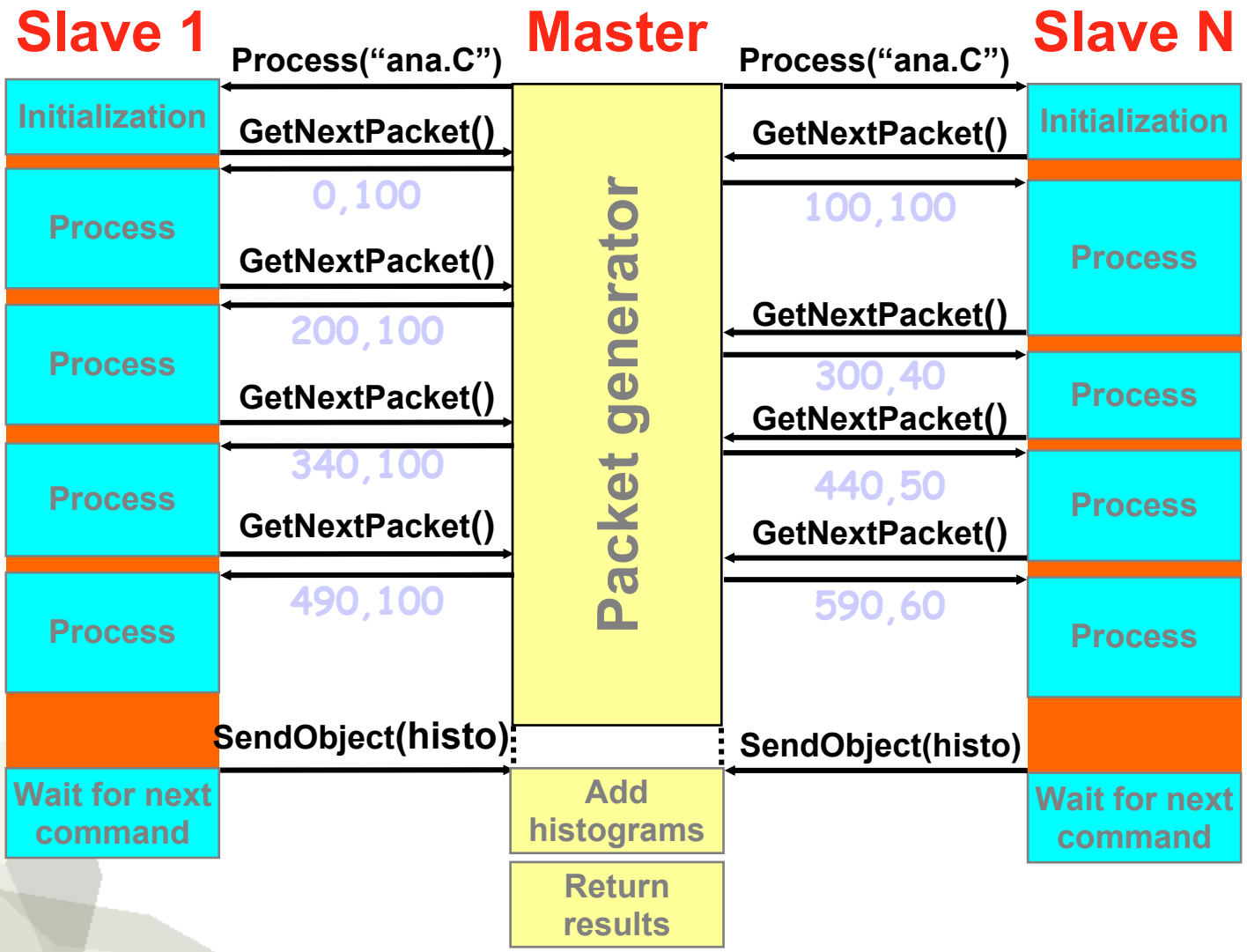
PROOF Basic Architecture

Single-Cluster mode

- The Master divides the work among the slaves
- After the processing finishes, merges the results (histograms, scatter plots)
- And returns the result to the Client



Workflow for tree analysis



PROOF and Selectors

Terminal

```
void MySelector::Begin(TTree *tree)
{  // called on the client before processing
}
void MySelector::SlaveBegin(TTree *tree)
{  // called on each slave before processing
   fMyHist = new TH1("Pt", "Pt");
   fOutput->Add(fMyHist);
}
void MySelector::Init(TTree *tree)
{   // called each time a tree is changed
   fPtBranch = tree->GetBranch("Pt")
   fPtBranch->SetAddress(&fPt);
}
Bool_t MySelector::Process(Long64_t entry)
{   // called on each slave for their entries
   fPtBranch->GetEntry(entry);
   fMyHist->Fill(fPt);
}
void MySelector::SlaveTerminate()
{   // called on each slave after processing
}
void MySelector::Terminate()
{   // called on the client after processing
   fMyHist->Draw();
}
```

The code is shipped to each slave and SlaveBegin(), Init(), Process(), SlaveTerminate() are executed there

Initialize each slave

Many Trees are being processed

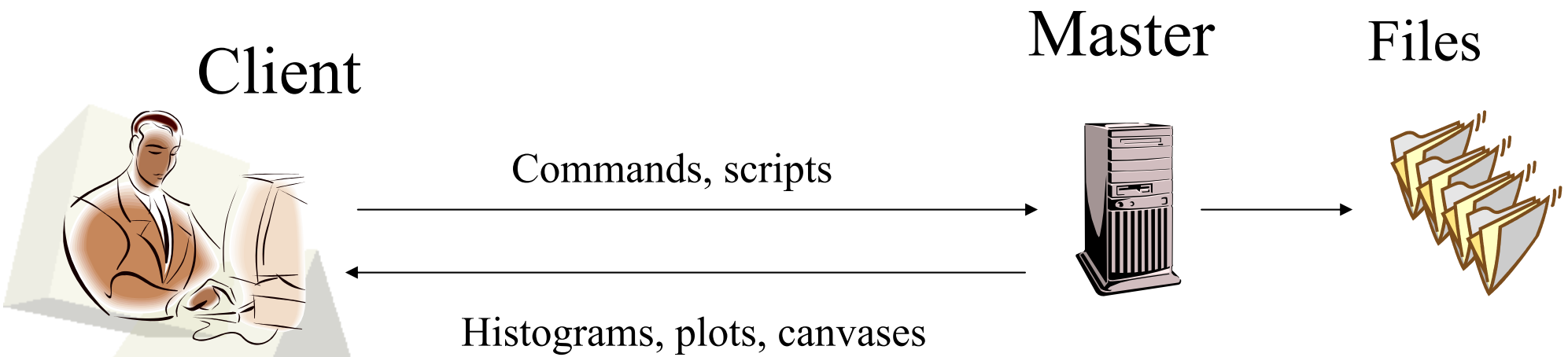
No user's control of the entries loop!

The same code works also without PROOF.

PROOF Sequential mode

- The Master executes scripts (**Selectors**) and returns results to the Client
- Canvases will be fetched from the Master automatically
- Pseudo-remote desktop (better than XWindow for WAN)

From the users point of view it works in the same way as the standard proof mode



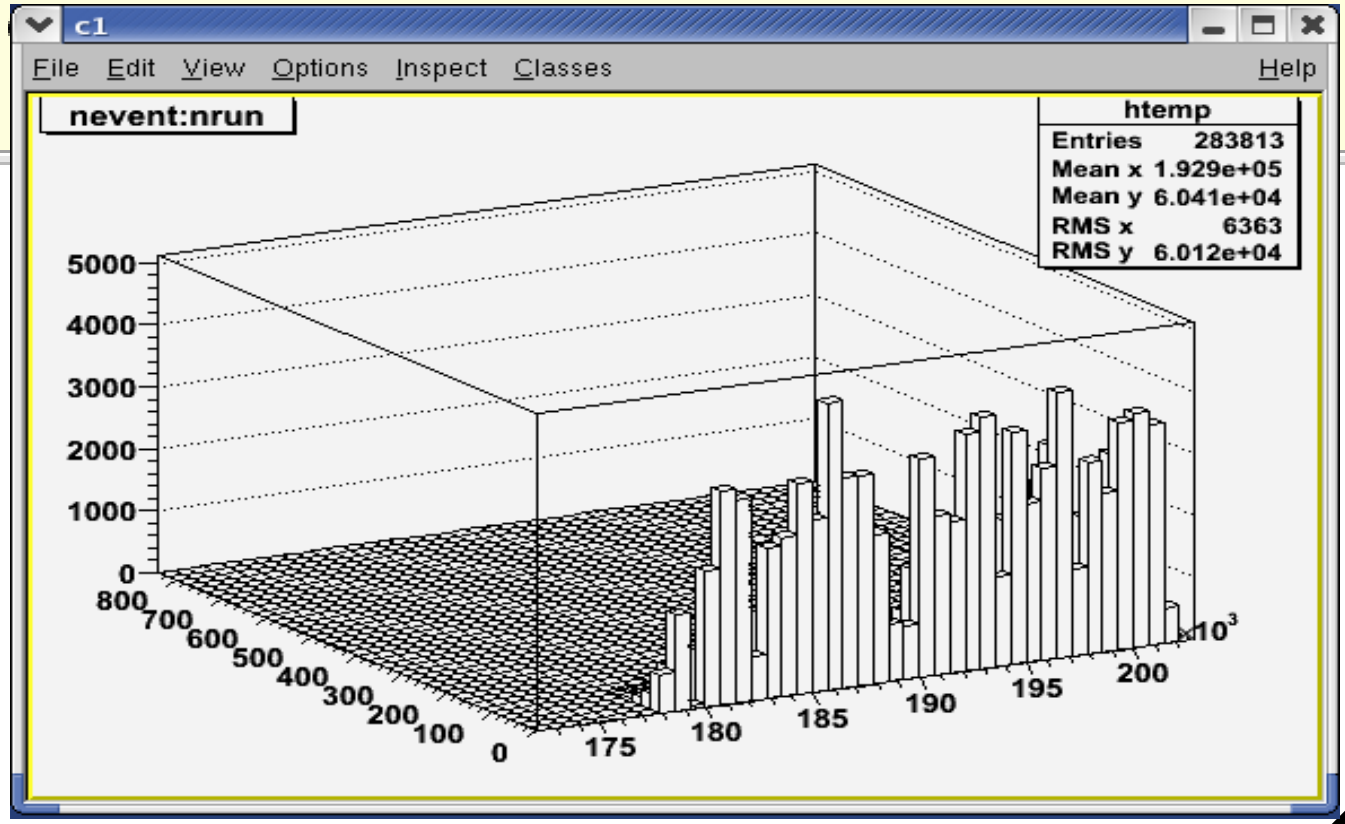
The canvas is automatically displayed after the processing has finished

Executes the selector and creates an off-screen canvas

PROOF – Drawing a histogram

Terminal

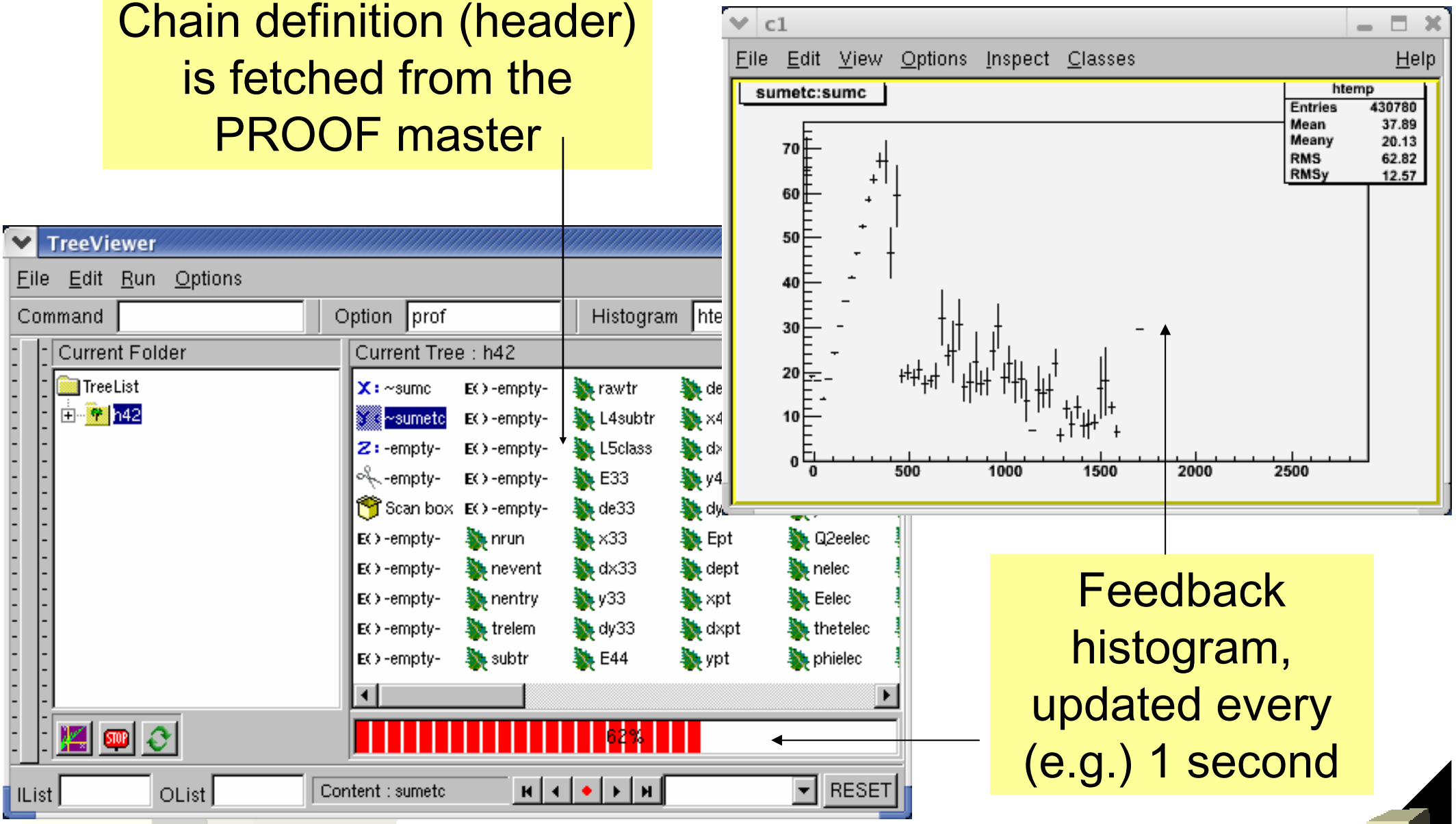
```
[mbiskup@pcphsft07 /home/mbiskup/root $ root
root [0] TChain chain("h42");
root [1] chain.Add("root://localhost//home/mbiskup/rootdata/dstarmb.root");
root [2] chain.Add("root://localhost//home/mbiskup/rootdata/dstarp1a.root");
root [3] chain.Add("root://localhost//home/mbiskup/rootdata/dstarp1b.root");
root [4] chain.Add("root://localhost//home/mbiskup/rootdata/dstarp2.root");
root [5] gROOT->Proof();
mbiskup@localhost.localdomain password:
PROOF set to parallel mode (3 slaves)
root [7] chain.Draw("nevent:nrun", "", "lego")
<TCanvas::MakeDefCanvas>:
(Long64_t)283813
root [8]
```



Chains may be also created automatically by a query to a grid catalog

GUI and real time feedback

Chain definition (header) is fetched from the PROOF master



Feedback histogram, updated every (e.g.) 1 second

Current Limitations of PROOF

Originally:

- Intended for interactive usage: Typical queries time – several minutes.

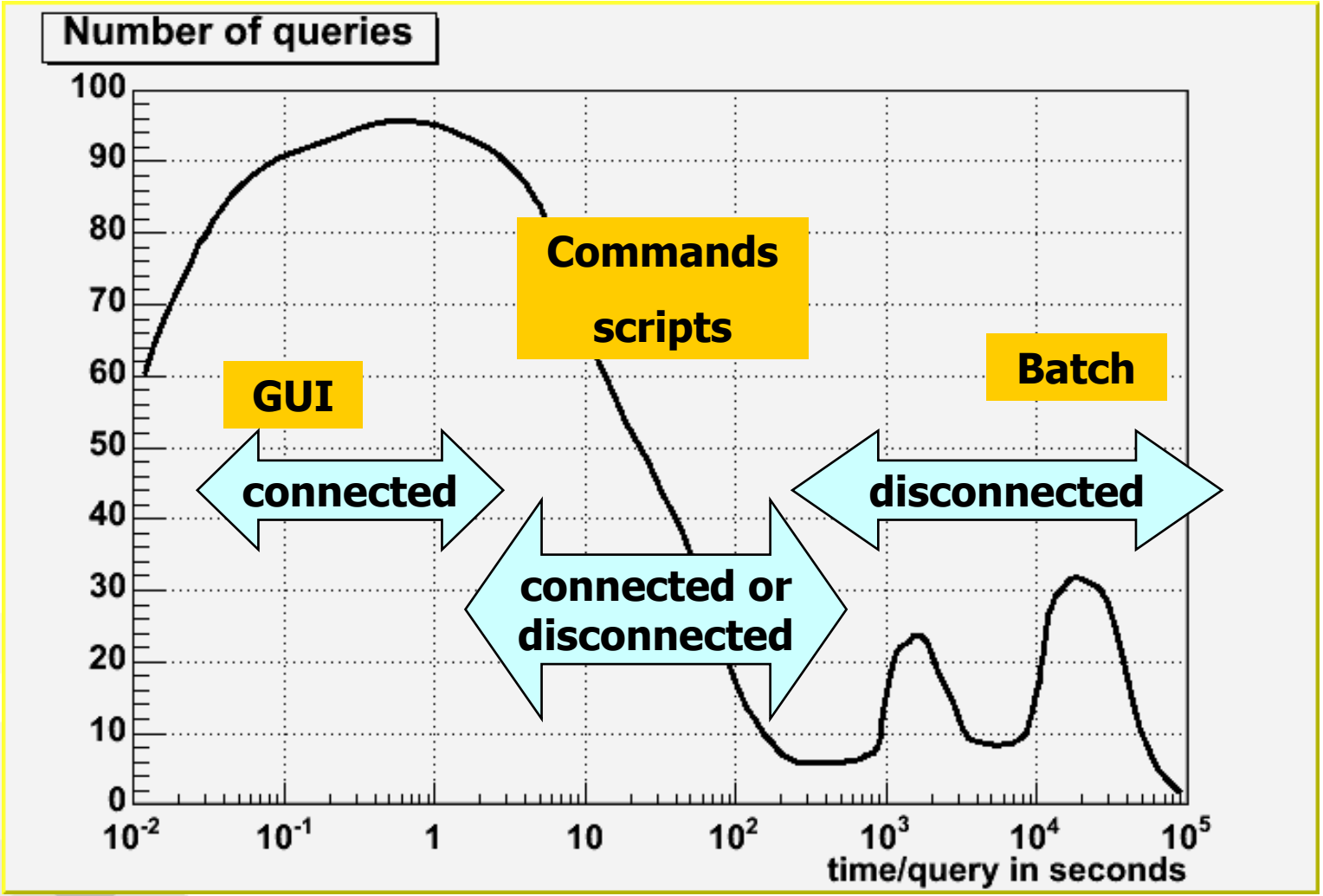
Processing blocks the client

- Designed to work on a local cluster with static configuration.

Permanent connection to the master.

No dynamic usage of the GRID.

Typical Queries



Analysis session snapshot

What are planning to implement:

AQ1: 1s query produces a local histogram

AQ2: a 10mn query submitted to PROOF1

AQ3->AQ7: short queries

AQ8: a 10h query submitted to PROOF2

Monday at 10h15 ROOT session On my laptop

BQ1: browse results of AQ2

BQ2: browse temporary results of AQ8

BQ3->BQ6: submit 4 10mn queries to PROOF1

Monday at 16h25 ROOT session On my laptop

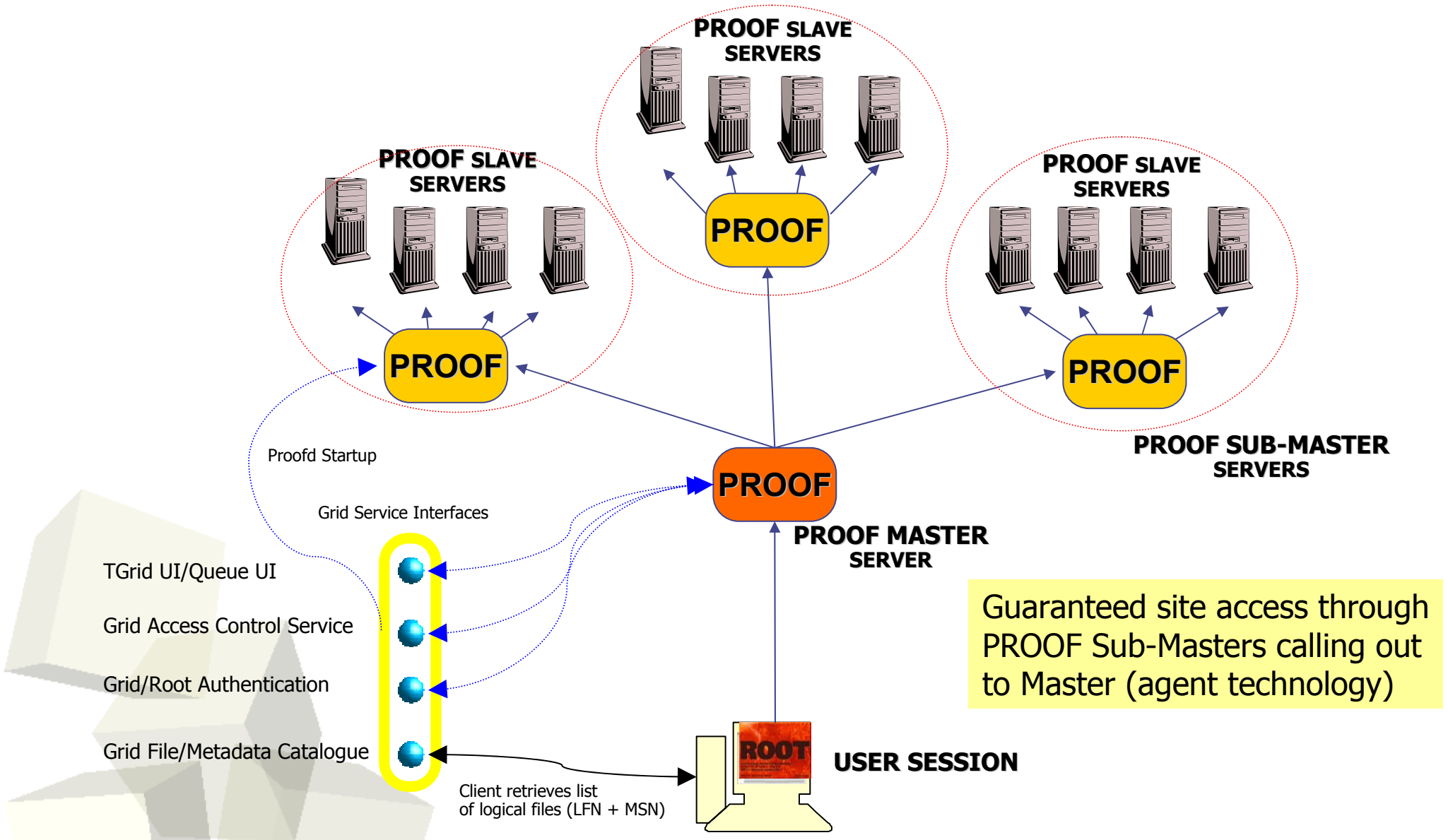
CQ1: Browse results of AQ8, BQ3->BQ6

Wednesday at 8h40 session on any web browser

Planned features

- Session disconnect and reconnect
- Asynchronous queries
- Start-up of slaves via Grid job scheduler
- Allow slaves to join/leave the computation
- Slaves calling out to master (firewalls)

PROOF on the Grid



Guaranteed site access through PROOF Sub-Masters calling out to Master (agent technology)

Summary

- ROOT is a powerful analysis framework with very efficient data storage mechanisms.
- PROOF works well for interactive parallel ROOT data analysis on a local cluster
  - Fully integrated with ROOT – you can use chains with PROOF in the same way as locally.
  - You can use the same Selectors you've written for local processing.
  - But it was designed for short-duration interactive queries.
- PROOF is evolving: we plan to accommodate longer running queries.
  - Disconnect from and reconnect to a running query.
  - Non-Blocking queries.
  - Dynamic configuration (using the GRID).

Questions