ROOT : Outlook and Developments

WLCG Jamboree Amsterdam
16-18 June 2010
René Brun/CERN

Foreword

- Since 1995 the ROOT system is in continuous development in all areas: interpreters, 2D and 3D graphics, Mathlibs and statistics, and of course I/O.
- In this talk, I concentrate on recent developments to speed-up the I/O. In particular these developments will open new possibilities in client-server applications.
- Remote file access in WANs with efficient caching will be the main topic of this talk.

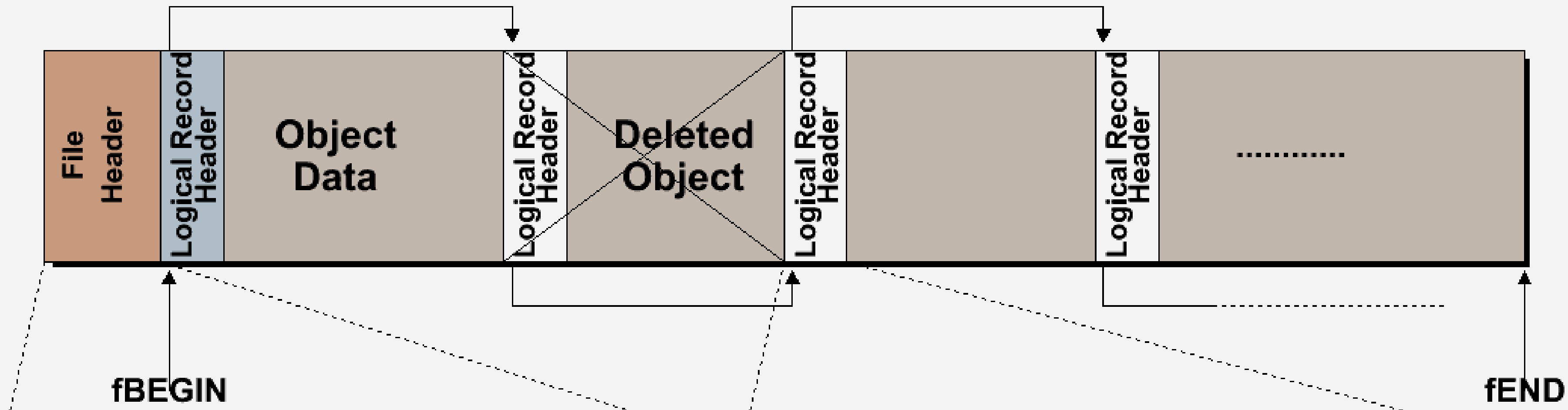
ROOT I/O

A short overview of the main features

Main features

- Designed for write once and read many times
  - but also support for object deletion and write in multiple jobs.
- Simple file format described in one slide
- Two types of objects: keys and trees
  - keys for objects like histograms, geometries (Unix-like)
  - trees for collections of similar objects (HEP events)
- self-describing portable and compact files
- client-server support

ROOT File description

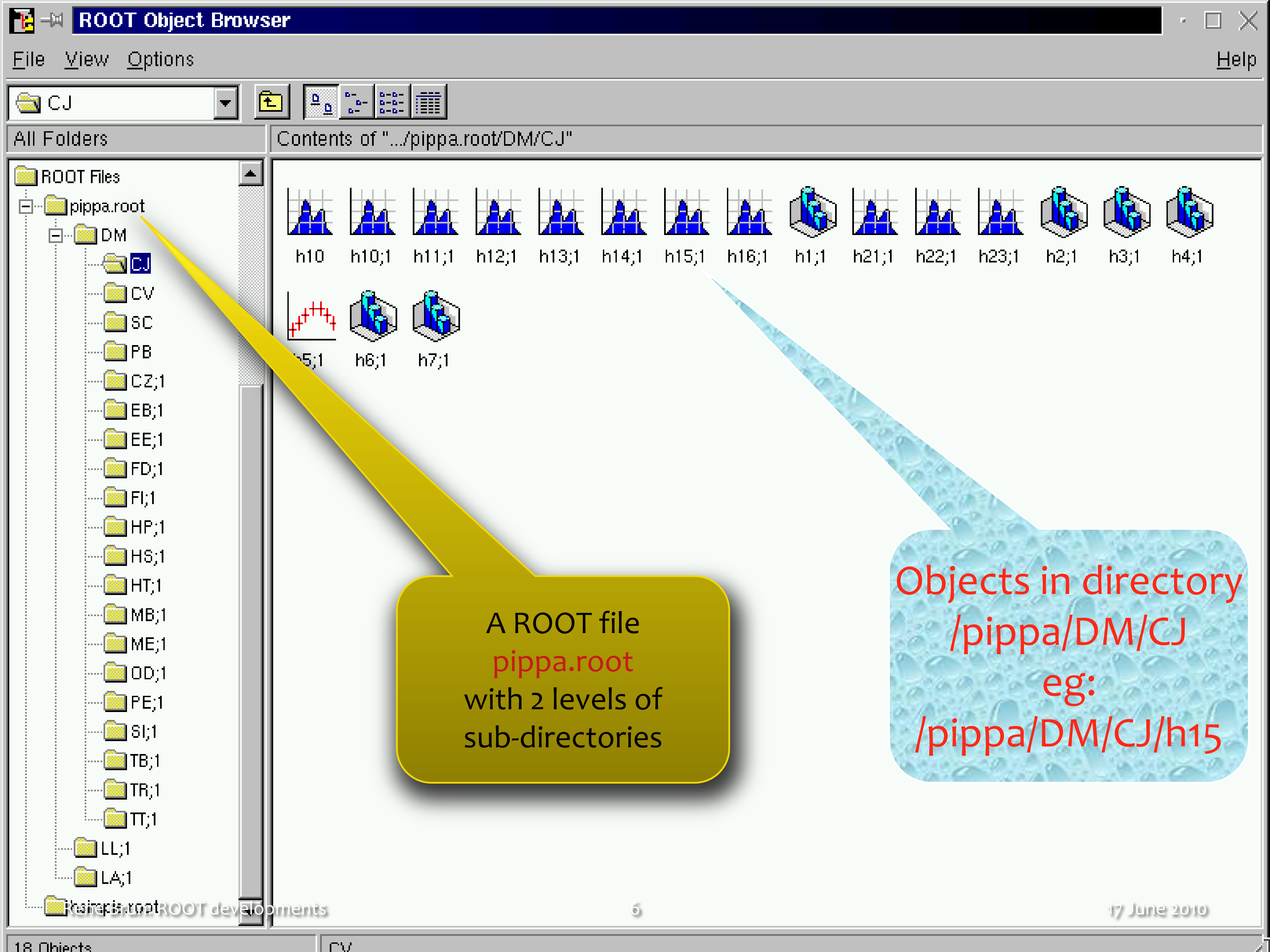


File Header

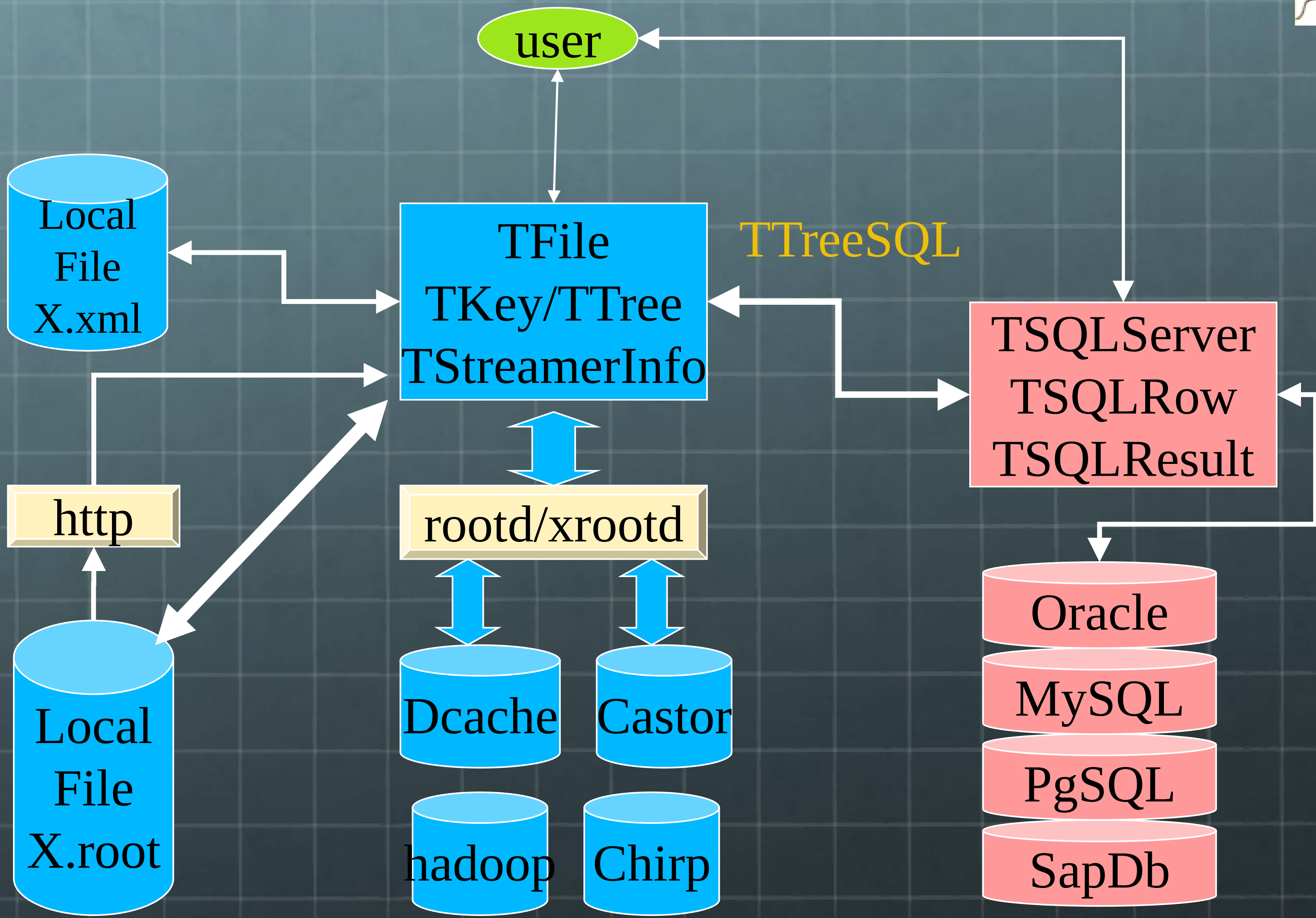
"root": Root File Identifier
fVersion: File version identifier
fBEGIN: Pointer to first data record
fEND: Pointer to first free word at EOF
fSeekFree: Pointer to FREE data record
fNbytesFree: Number of bytes in FREE
fNfree: Number of free data records
fNbytesName: Number of bytes in name/title
fUnits: Number of bytes for pointers
fCompress: Compression level

Logical Record Header (TKEY)

fNbytes: Length of compressed object
fVersion: Key version identifier
fObjLen: Length of uncompressed object
fDatime: Date/Time when written to store
fKeylen: Number of bytes for the key
fCycle : Cycle number
fSeekKey: Pointer to object on file
fSeekPdir: Pointer to directory on file
fClassName: class name of the object
fName: name of the object
fTitle: title of the object

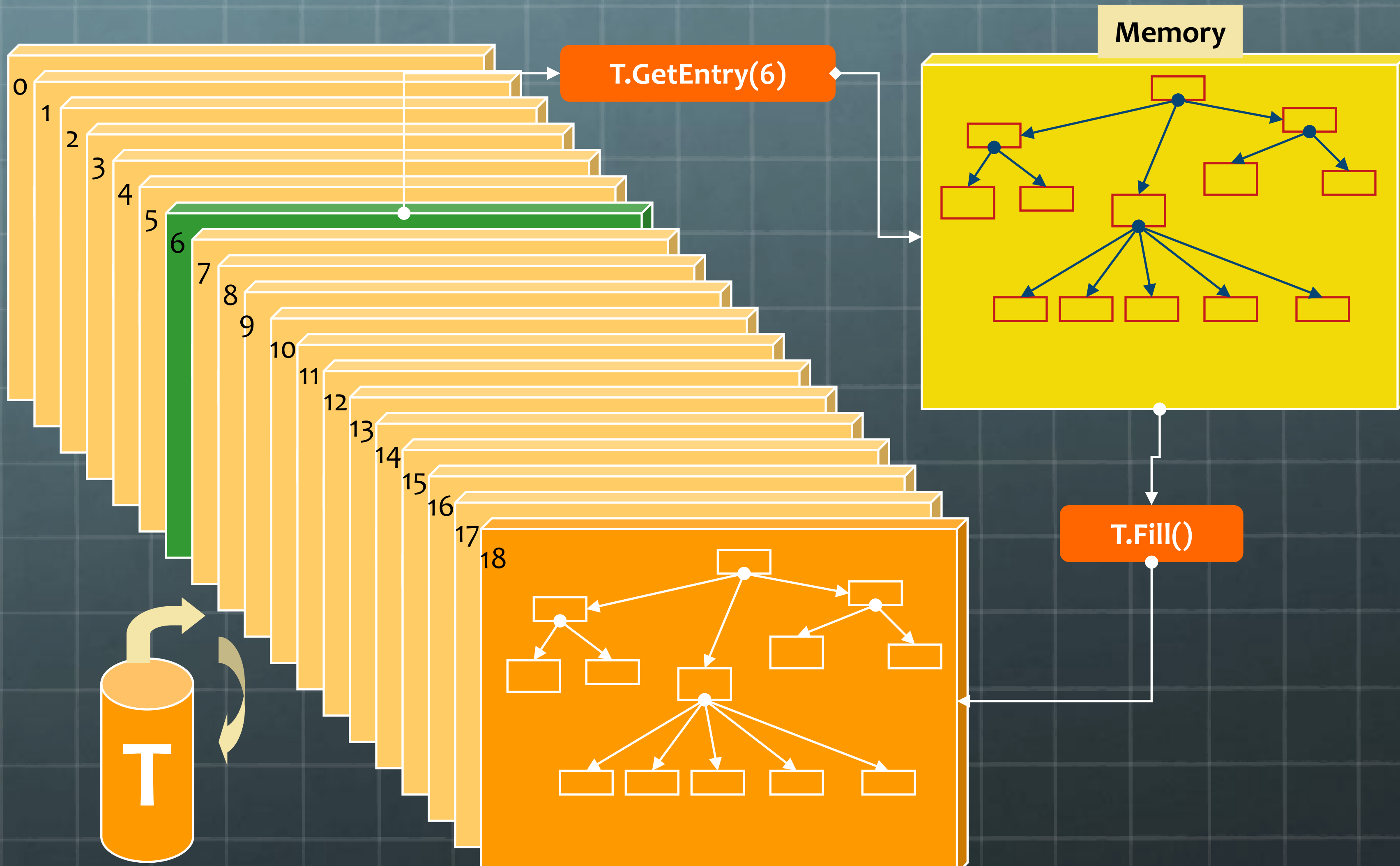

ROOT Object Browser
File View Options Help
CJ
All Folders
Contents of ".../pippa.root/DM/CJ"
ROOT Files
pippa.root
DM
CJ
CV
SC
PB
CZ;1
EB;1
EE;1
FD;1
FI;1
HP;1
HS;1
HT;1
MB;1
ME;1
OD;1
PE;1
SI;1
TB;1
TR;1
TT;1
LL;1
LA;1
h10 h10;1 h11;1 h12;1 h13;1 h14;1 h15;1 h16;1 h1;1 h21;1 h22;1 h23;1 h2;1 h3;1 h4;1
h6;1 h7;1
18 Objects
CV
A ROOT file
pippa.root
with 2 levels of
sub-directories
Objects in directory
/pippa/DM/CJ
eg:
/pippa/DM/CJ/h15

File types & Access

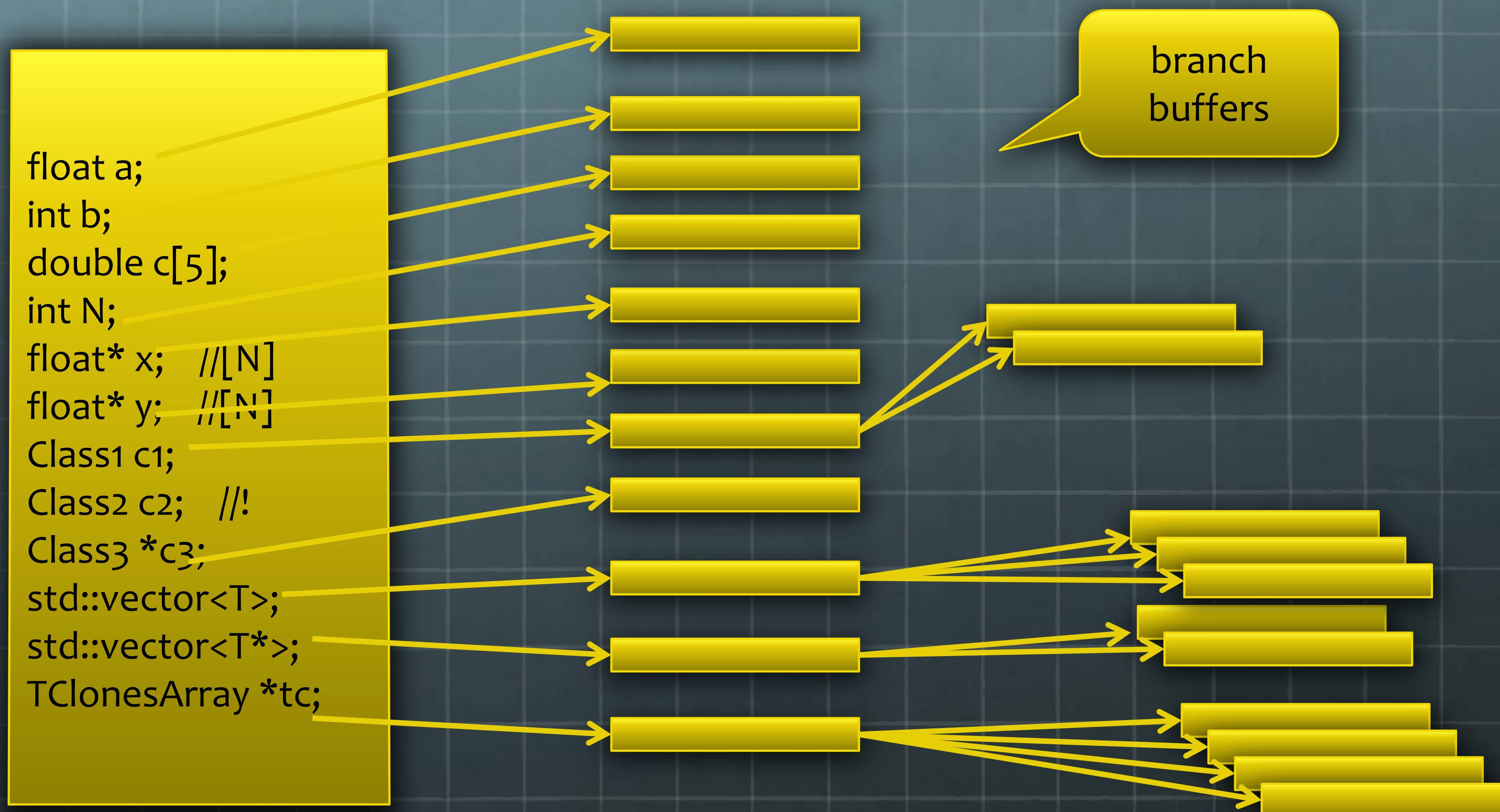


Memory <--> Tree

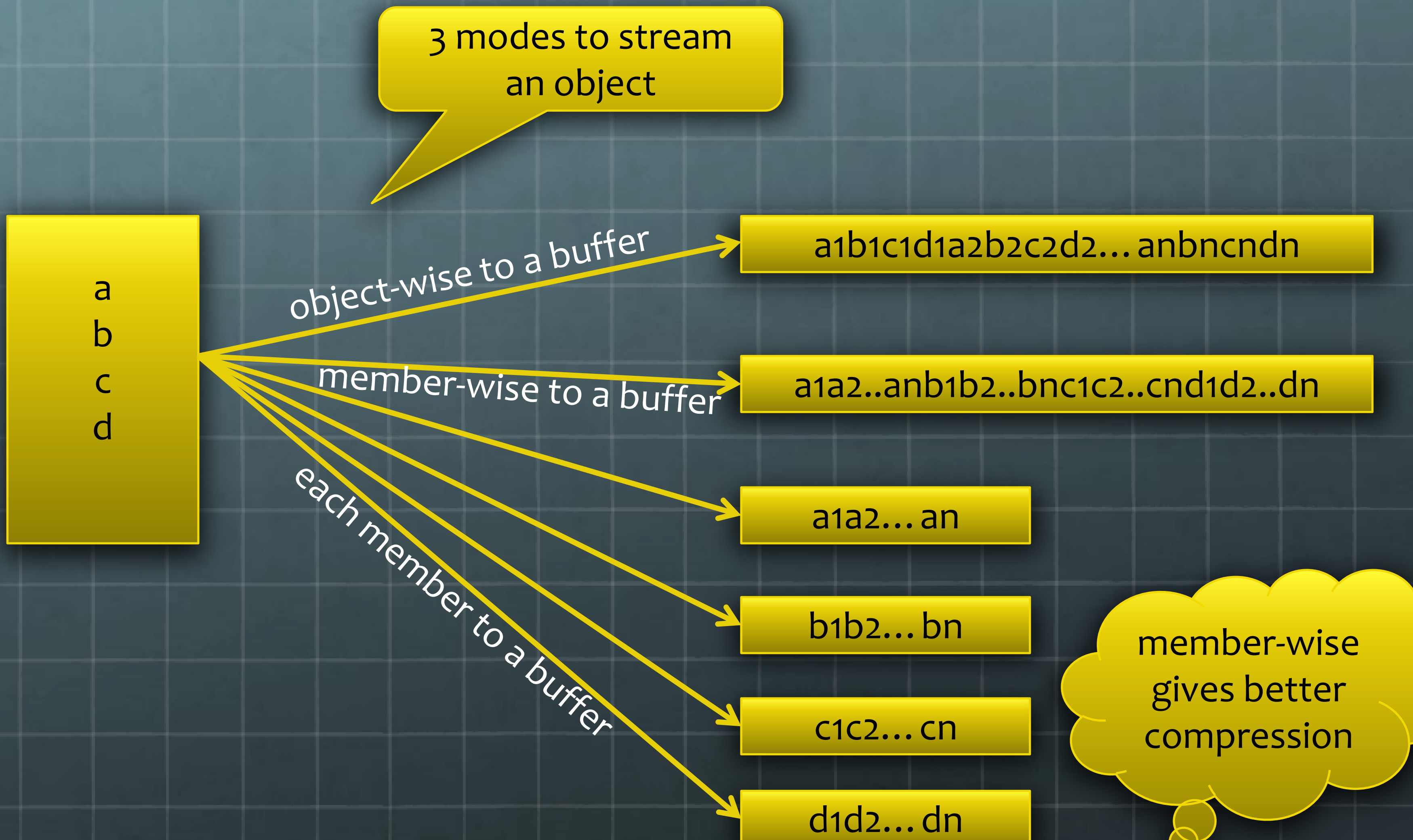
Each Node is a branch in the Tree



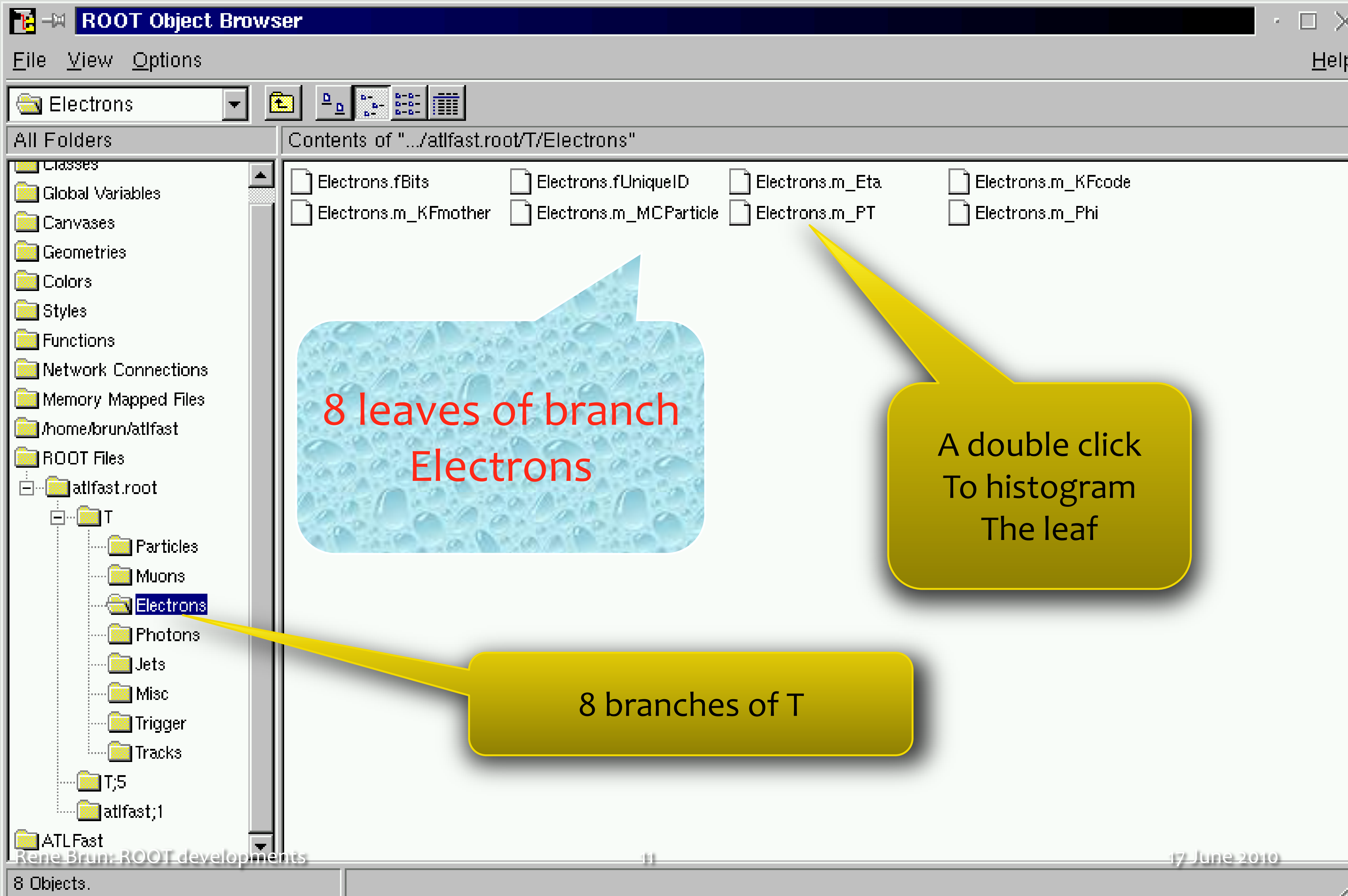
Automatic branch creation from object model



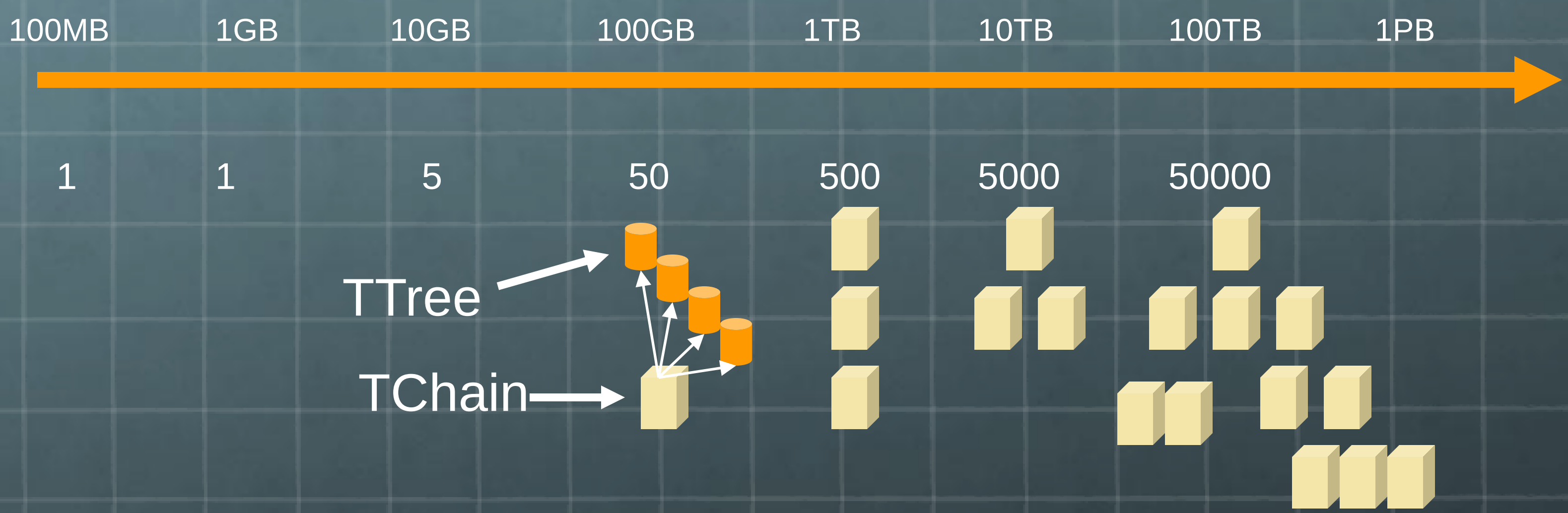
ObjectWise/MemberWise Streaming



Browsing a TTree with TBrowser



Data Volume & Organization



A TFile typically contains 1 TTree

A TChain is a collection of TTrees or/and TChains

A TChain is typically the result of a query to the file catalogue

Queries to the data base

- via the GUI (TBrowser of TTreeViewer)
- via a CINT command or a script
  - tree.Draw(“x”,”y<0 && sqrt(z)>6”);
  - tree.Process(“myscript.C”);
- via compiled code
  - chain.Process(“myselector.C+”);
- in parallel with PROOF

ROOT I/O

Current developments
Caches and Caches
Speed-up
Parallel Merge

Buffering effects

- Branch buffers are not full at the same time.
- A branch containing one integer/event and with a buffer size of 32Kbytes will be written to disk every 8000 events, while a branch containing a non-split collection may be written at each event.
- This may give serious problems when reading if the file is not read sequentially.

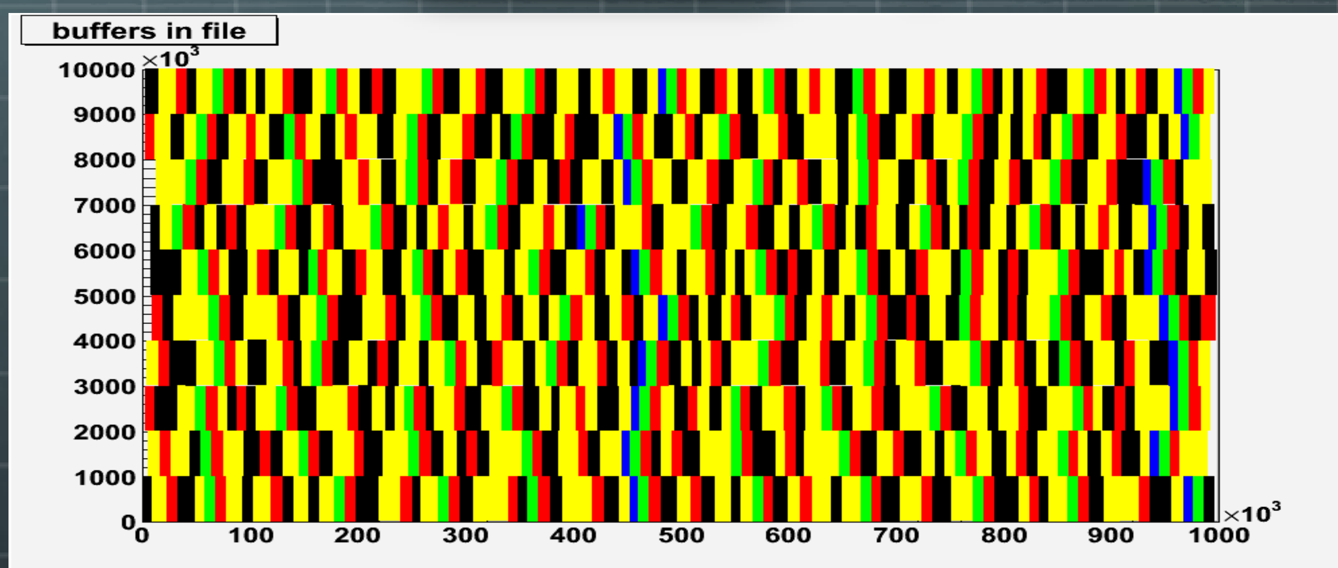
Tree Buffers layout

- Example of a Tree with 5 branches
  - b1 : 400 bytes/event
  - b2: 2500 ± 50 bytes/ev
  - b3: 5000 ± 500 bytes/ev
  - b4: 7500 ± 2500 bytes/ev
  - b5: 10000 ± 5000 bytes/ev

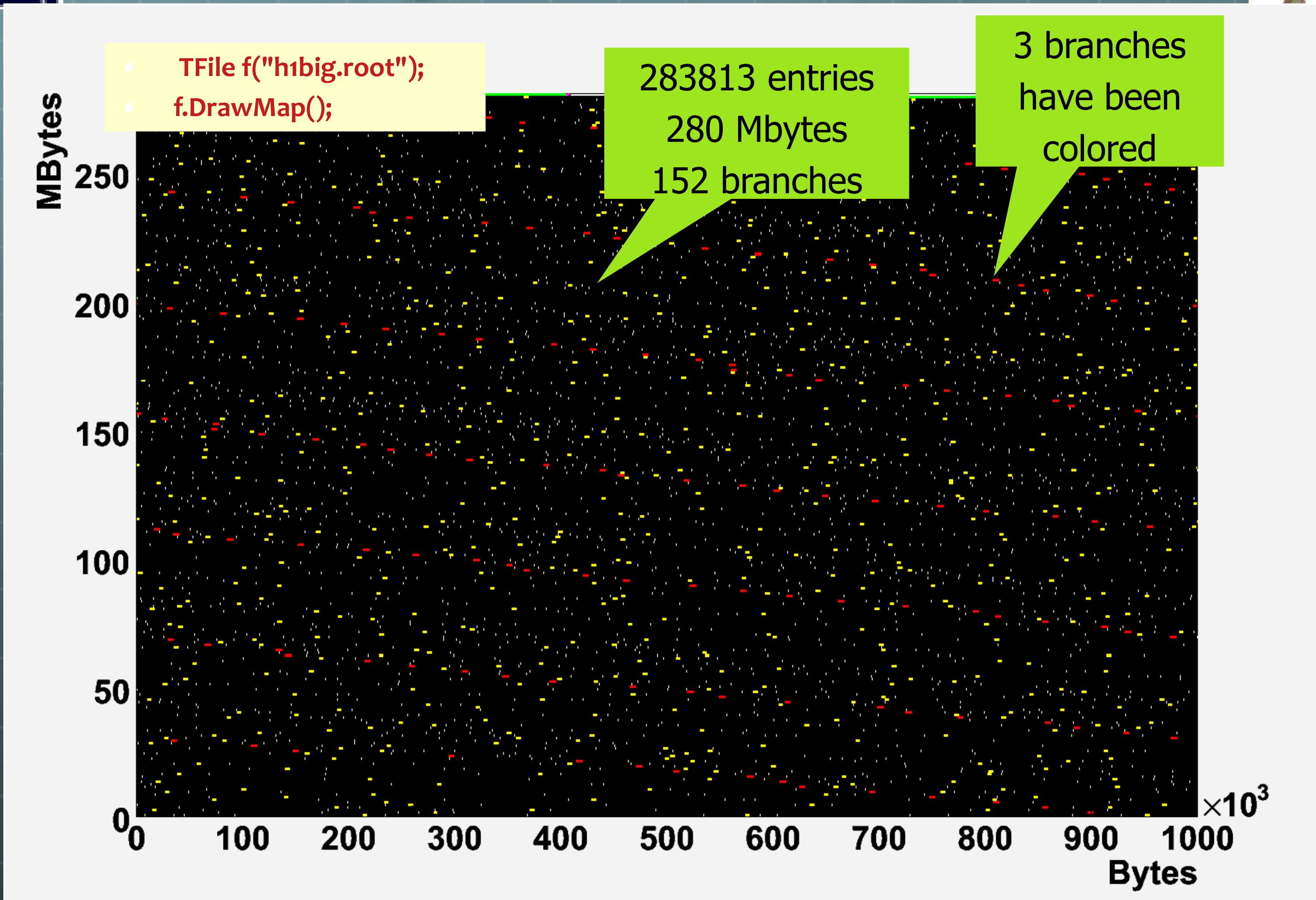
each branch has its own buffer (8000 bytes) (< 3000 zipped)

10 rows of 1 MByte in this 10 MBytes file

typical Trees have several hundred branches



Looking inside a ROOT Tree



I/O Performance Analysis

Monitor TTree reads with **TTreePerfStats**

```
TFile *f = TFile::Open("xyz.root");
T = (TTree*)f->Get("MyTree");

TTreePerfStats ps("ioperf",T);

Long64_t n = T->GetEntries();
for (Long64_t i = 0;i < n; ++i) {
   GetEntry(i);
   DoSomething();
}
ps.SaveAs("perfstat.root");
```

New in v5.25/04! (June 2009)

Study TTreePerfStats

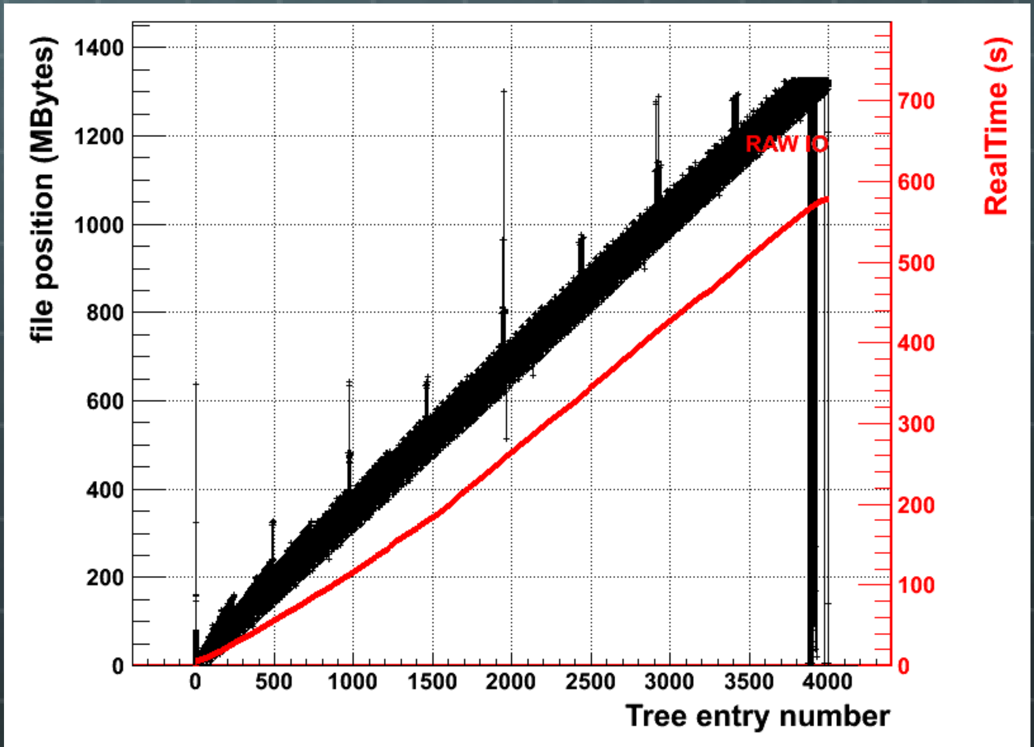
Visualizes read access:

x: tree entry

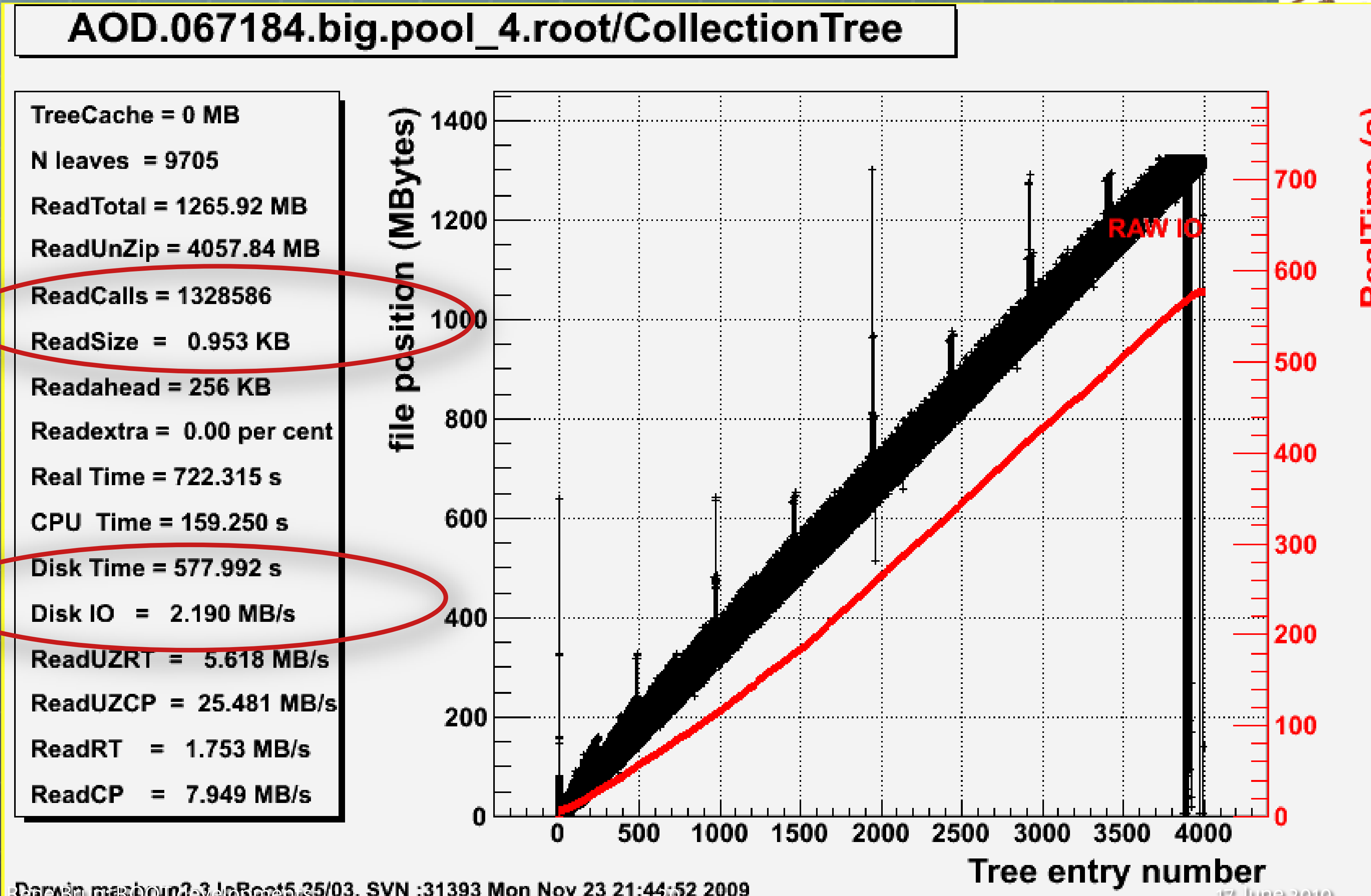
y: file offset

y: real time

```
TFile f("perfstat.root");
ioperf->Draw();
ioperf->Print();
```

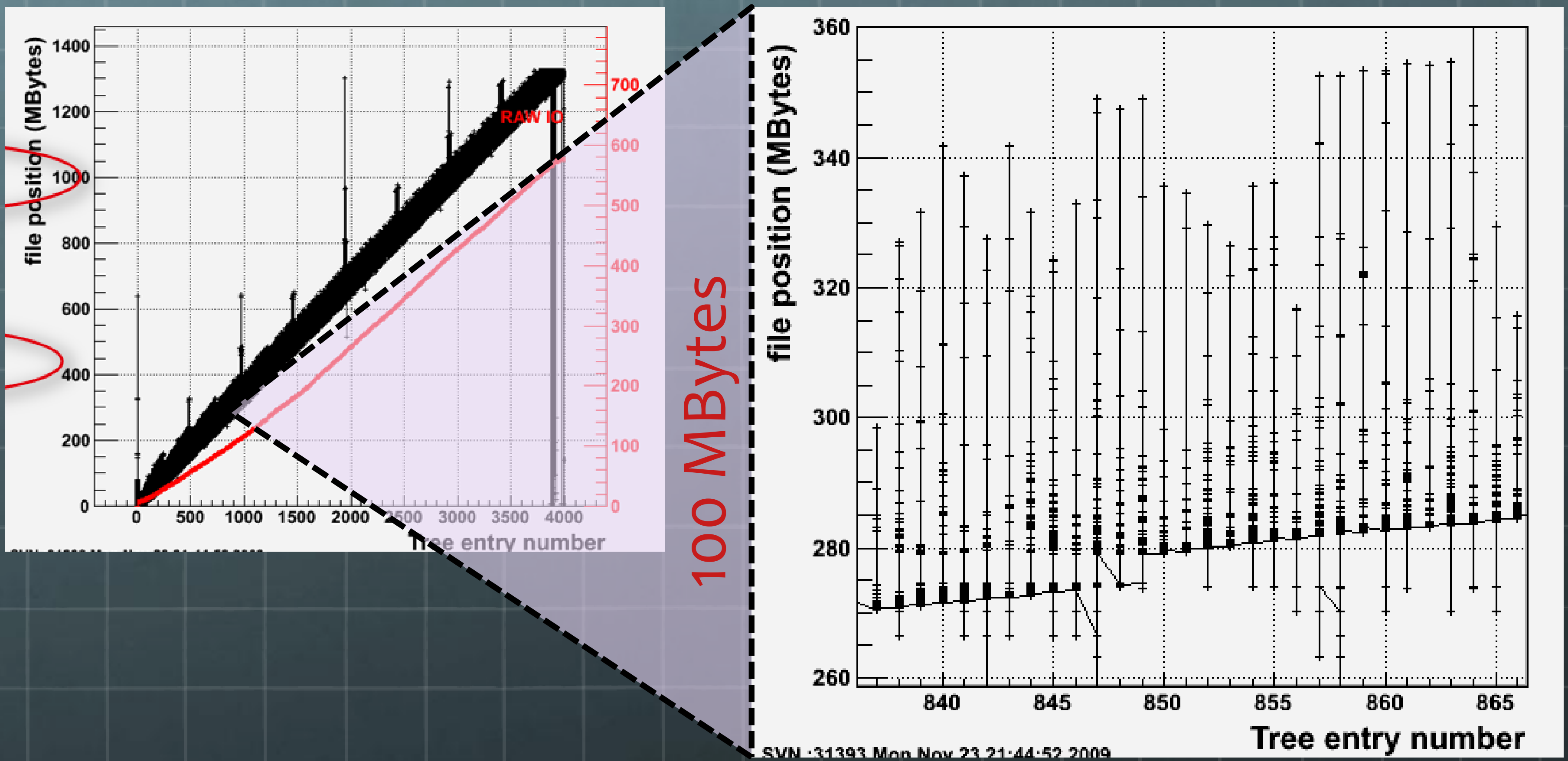


See Doctor



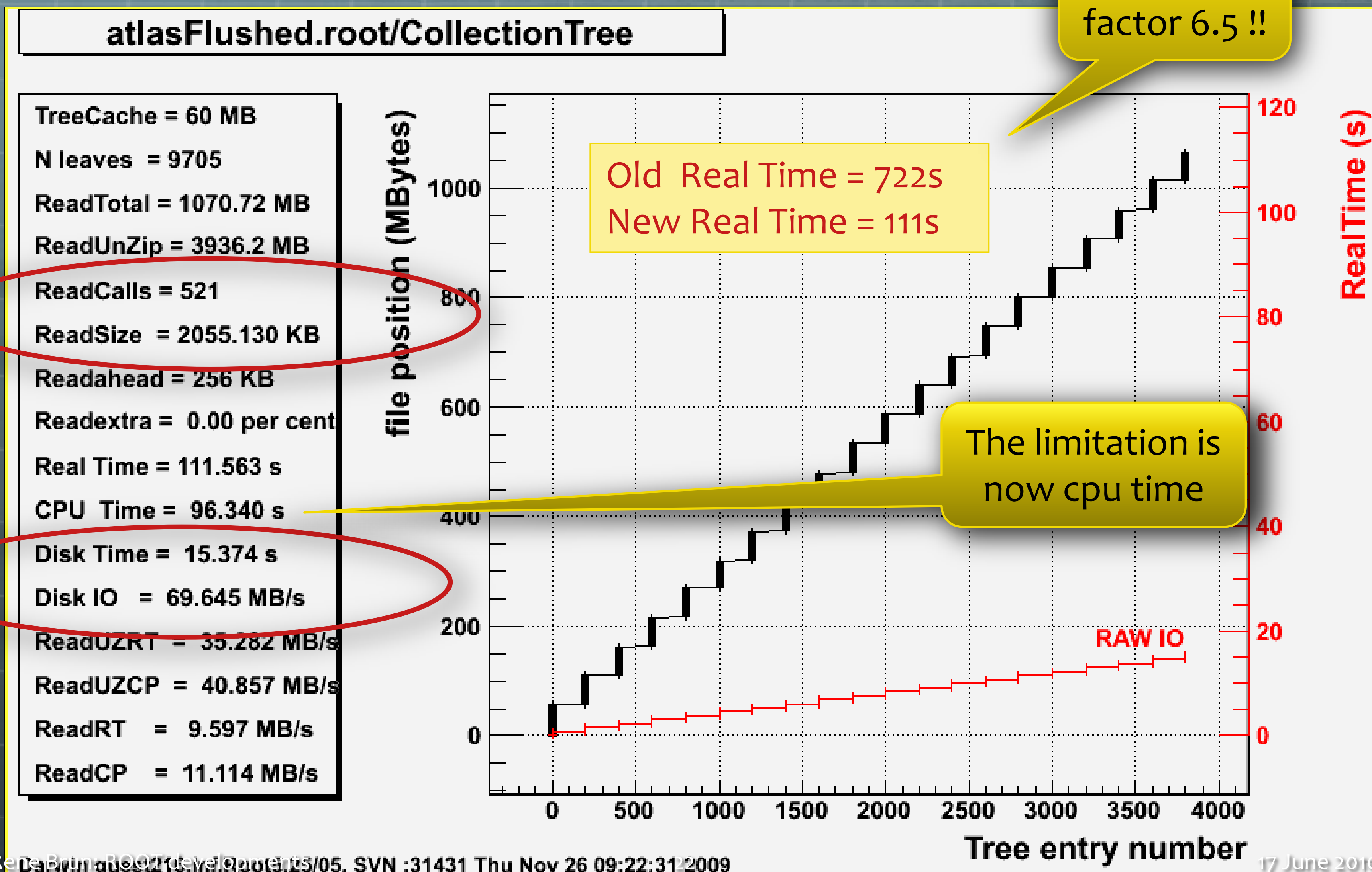
AOD.067184.big.pool_4.root/CollectionTree
TreeCache = 0 MB
N leaves = 9705
ReadTotal = 1265.92 MB
ReadUnZip = 4057.84 MB
ReadCalls = 1328586
ReadSize = 0.953 KB
Readahead = 256 KB
Readextra = 0.00 per cent
Real Time = 722.315 s
CPU Time = 159.250 s
Disk Time = 577.992 s
Disk IO = 2.190 MB/s
ReadUZRT = 5.618 MB/s
ReadUZCP = 25.481 MB/s
ReadRT = 1.753 MB/s
ReadCP = 7.949 MB/s
file position (MBytes)
0
200
400
600
800
1000
1200
1400
RAW IO
RealTime (s)
0
100
200
300
400
500
600
700
0 500 1000 1500 2000 2500 3000 3500 4000
Tree entry number
Darwin macbrun2.3 | Root5.25/03, SVN :31393 Mon Nov 23 21:44:52 2009

Overlapping reads



After doctor

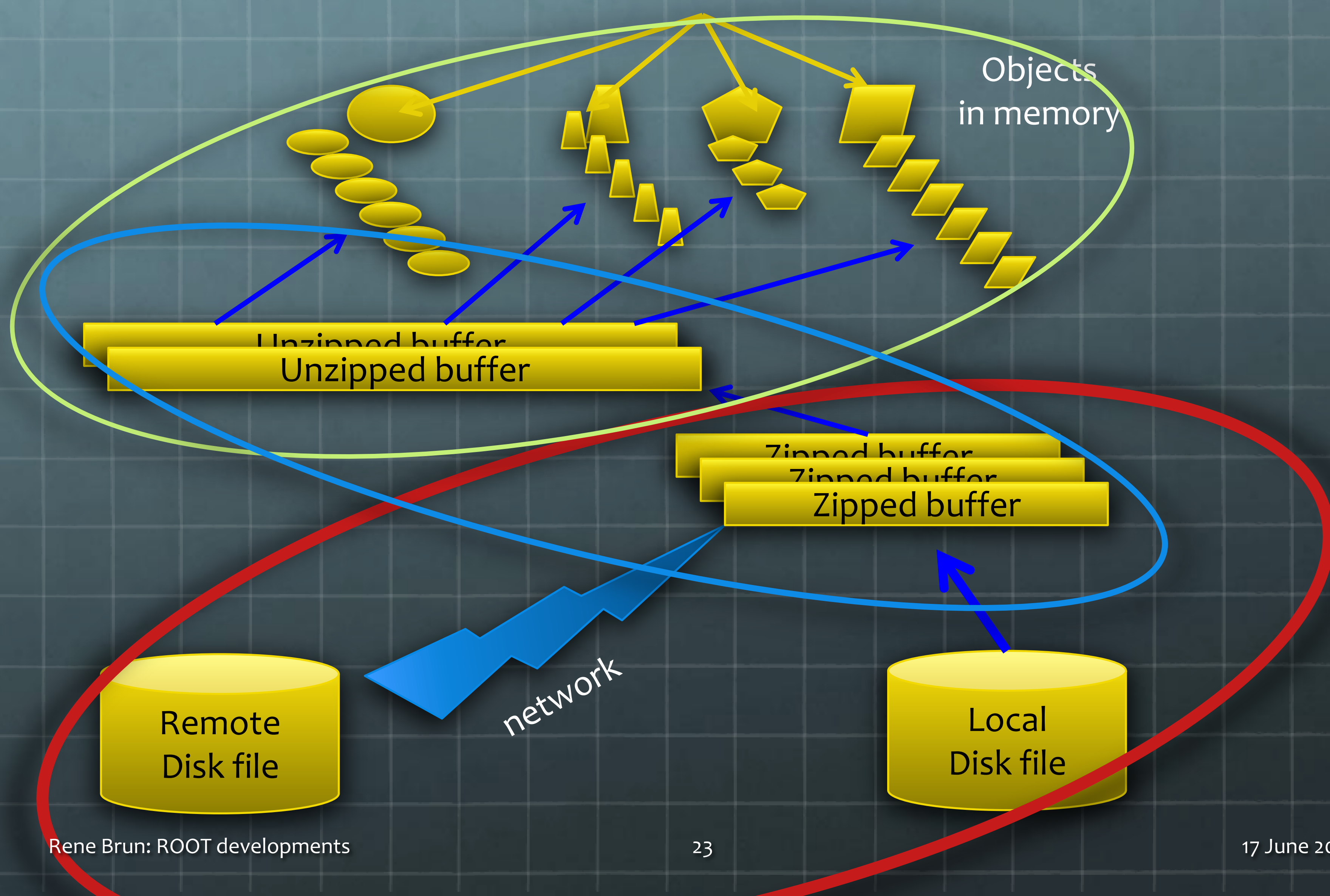
gain a factor 6.5 !!



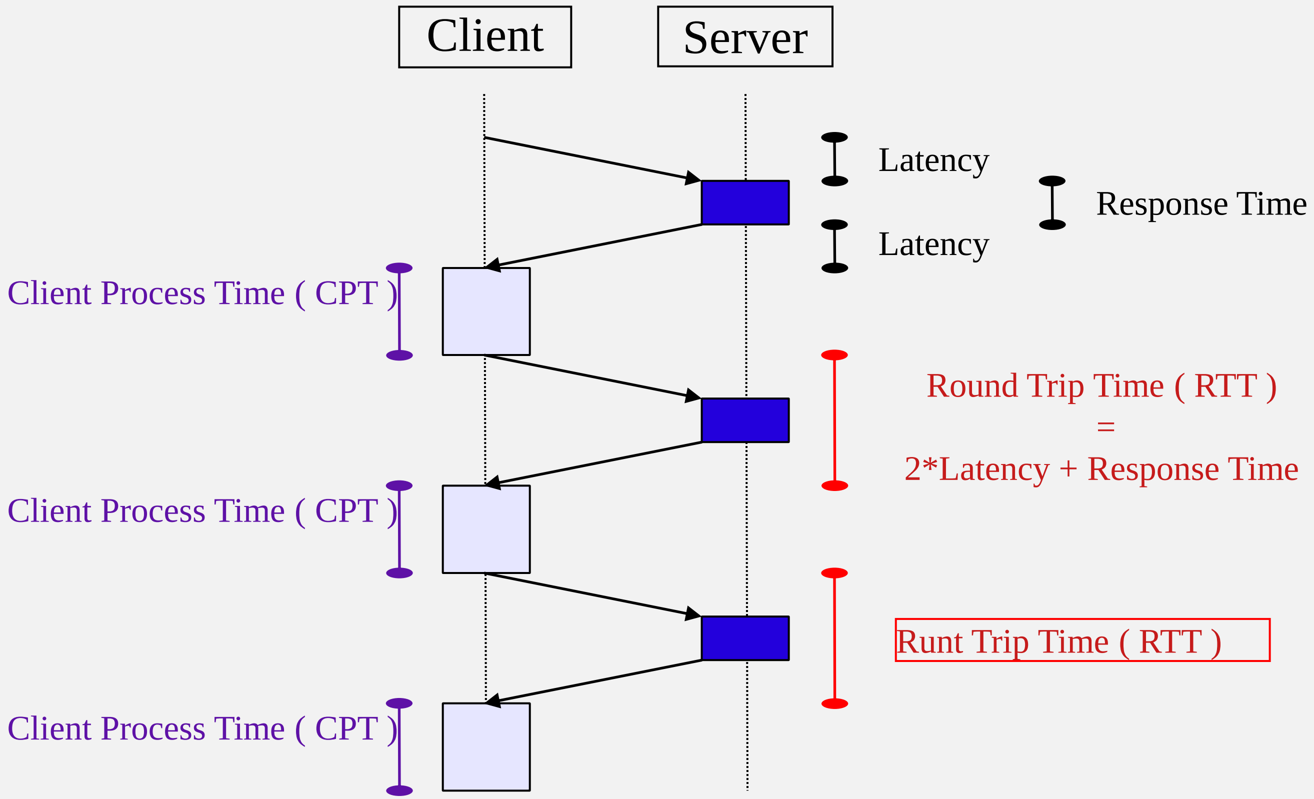
Old Real Time = 722s
New Real Time = 111s

The limitation is now cpu time

Important factors



A major problem: network latency

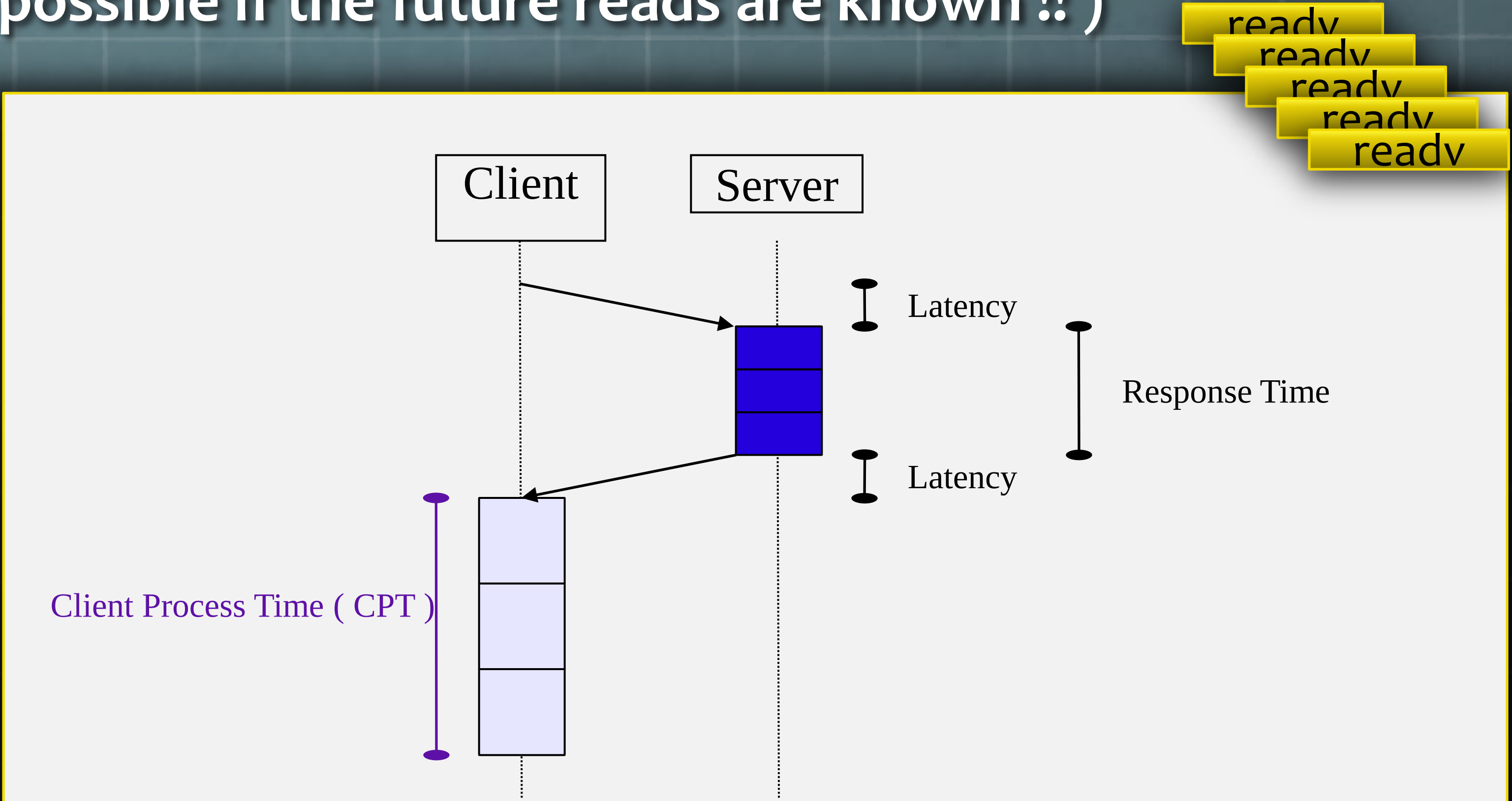


Total Time = 3 * [Client Process Time (CPT)] + 3*[Round Trip Time (RTT)]

Total Time = 3* (CPT) + 3 * (Response time) + 3 * (2 * Latency)

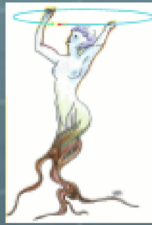
Idea (diagram)

- Perform a big request instead of many small requests (only possible if the future reads are known !!)



Total Time = 3* (CPT) + 3 * (Response time) + (2 * Latency)

What is the TreeCache



- It groups into one buffer all blocks from the used branches.
- The blocks are sorted in ascending order and consecutive blocks merged such that the file is read sequentially.
- It reduces typically by a factor 10000 the number of transactions with the disk and in particular the network with servers like httpd, xrootd or dCache.
- The typical size of the TreeCache is 30 Mbytes, but higher values will always give better results.

readv implementations

- xrootd
  - TFile f1("root://machine1.xx.yy/file1.root")
- dCache
  - TFile f2("dcap://machine2.uu.vv/file2.root")
- httpd
  - TFile f3(http://something.nikhef.nl/file3.root);
  - uses a standard (eg apache2) web server
  - performance winner (but not many people know !)

I like it

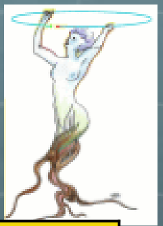
TTreeCache with LANs and WANs

client	latency (ms)	cachesize 0	cachesize 64k	cachesize 10 MB
A: local pcbrun.cern.ch	0	3.4 s	3.4	3.3
B: 100Mb.s CERN LAN	0.3	8.0 s	6.0	4.0
C: 10 Mb/s CERN wireless	2	11.6 s	5.6	4.9
D: 100 Mb/s Orsay	11	124.7 s	12.3	9.0
E: 100 Mb/s Amsterdam	22	230.9 s	11.7	8.4
F: 8 Mb/s ADSL home	72	743.7 s	48.3	28.0
G: 10 Gb/s Caltech	240	2800 s	125.4	4.6

old slide from 2005

One query to a 280 MB Tree I/O = 6.6 MB

TreeCache results table



Original Atlas file (1266 MB), 9705 branches split=99

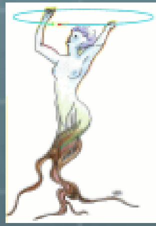
Cache size (MB)	readcalls	RT pcbrun4 (s)	CP pcbrun4 (s)	RT macbrun (s)	CP macbrun (s)
0	1328586	734.6	270.5	618.6	169.8
LAN 1ms 0	1328586	734.6+1300	270.5	618.6+1300	169.8
10	24842	298.5	228.5	229.7	130.1
30	13885	272.1	215.9	183.0	126.9
200	6211	217.2	191.5	149.8	125.4

Reclust: OptimizeBaskets 30 MB (1147 MB), 203 branches split=0

Cache size (MB)	readcalls	RT pcbrun4 (s)	CP pcbrun4 (s)	RT macbrun (s)	CP macbrun (s)
0	15869	148.1	141.4	81.6	80.7
LAN 1ms 0	15869	148.1 + 16	141.4	81.6 + 16	80.7
10	714	157.9	142.4	93.4	82.5
30	600	165.7	148.8	97.0	82.5
200	552	154.0	137.6	98.1	82.0

Reclust: OptimizeBaskets 30 MB (1086 MB), 9705 branches split=99

Cache size (MB)	readcalls	RT pcbrun4 (s)	CP pcbrun4 (s)	RT macbrun (s)	CP macbrun (s)
0	515350	381.8	216.3	326.2	127.0
LAN 1ms 0	515350	381.8 + 515	216.3	326.2 +515	127.0
10	15595	234.0	185.6	175.0	106.2
30	8717	216.5	182.6	144.4	104.5
200	2096	182.5	163.3	122.3	103.4



TreeCache: new interface

- Facts: Most users did not know if they were using or not the **TreeCache**.

- We decided to implement a simpler interface from **TTree** itself (no need to know about the class TTreeCache anymore).

- Because some users claimed to use the **TreeCache** and the results clearly showing the contrary, we decided to implement a new IO monitoring class **TTreePerfStats**.

TTreeCache

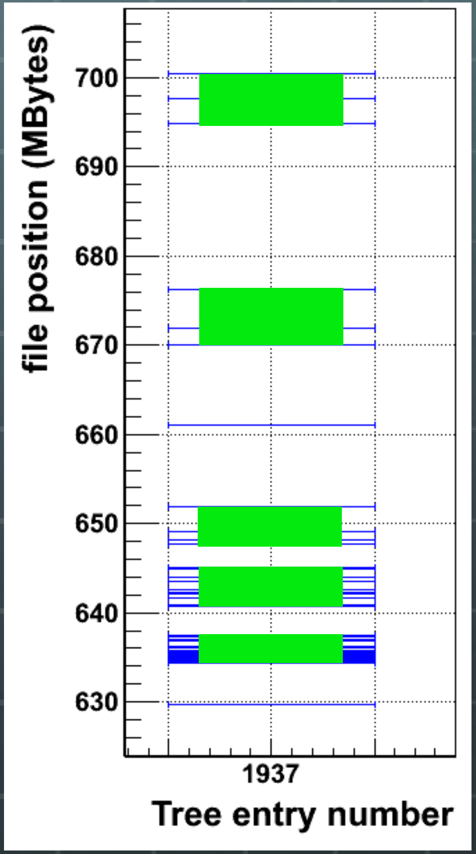
- Sends a collection of read requests *before* analysis needs the baskets
- Must predict baskets:
  - learns from previous entries
  - takes TEntryList into account
- Enabled per TTree

```
f = new TFile ("xyz.root");
T = (TTree*)f->Get("Events");
T->SetCacheSize(30000000);
T->AddBranchToCache("*");
```

Improved in v5.25/04!

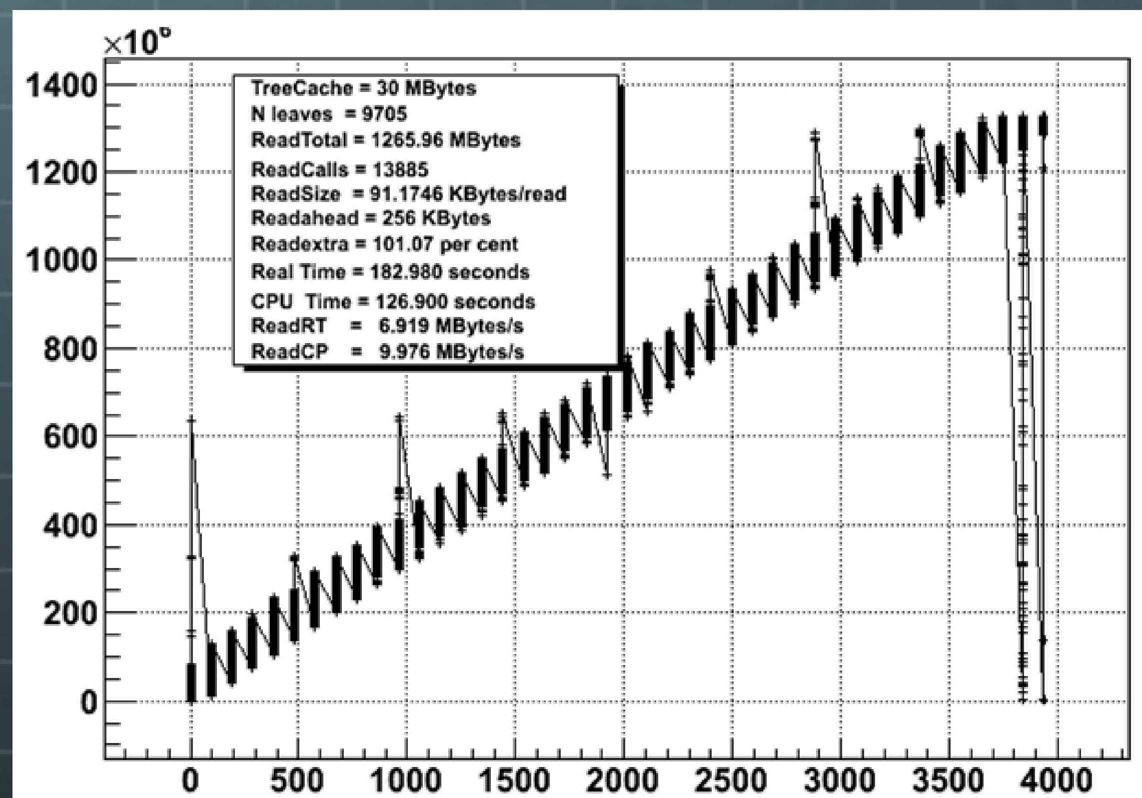
What is the readahead cache

- The readahead cache will read all non consecutive blocks that are in the range of the cache.
- It minimizes the number of disk access. This operation could in principle be done by the OS, but the fact is that the OS parameters are not tuned for many small reads, in particular when many jobs read concurrently from the same disk.
- When using large values for the TreeCache or when the baskets are well sorted by entry, the readahead cache is not necessary.
- Typical (default value) is 256 Kbytes, although 2 Mbytes seems to give better results on Atlas files, but not with CMS or Alice.



Half Way

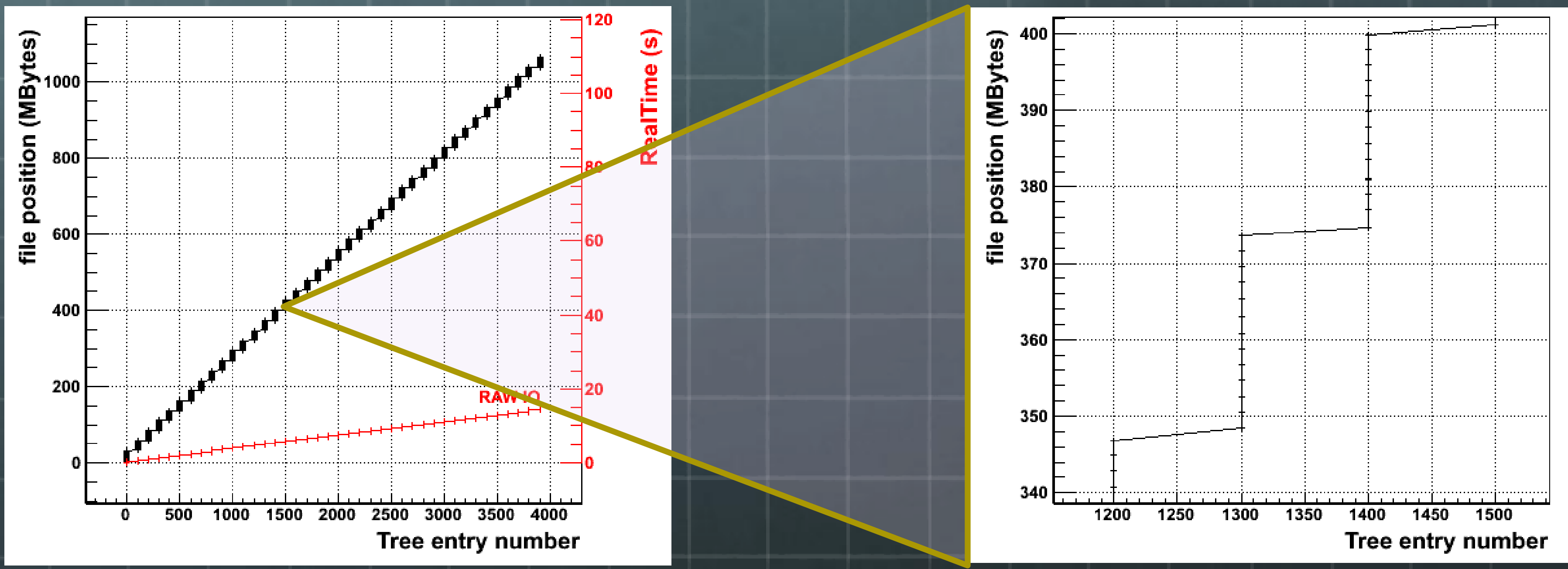
- Much more ordered reads
- Still lots of jumps because baskets spread across file



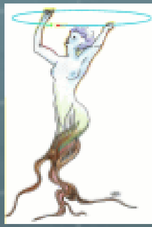
OptimizeBaskets, AutoFlush

New in v5.25/04!

- Solution, enabled by default:
  - Tweak basket size!
  - Flush baskets at regular intervals!



OptimizeBaskets



- **Facts**: Users do not tune the branch buffer size
- **Effect**: branches for the same event are scattered in the file.
- **TTree::OptimizeBaskets** is a new function in 5.25 that optimizes the buffer sizes taking into account the population in each branch.
- One can call this function on an existing read only Tree file to see the diagnostics.

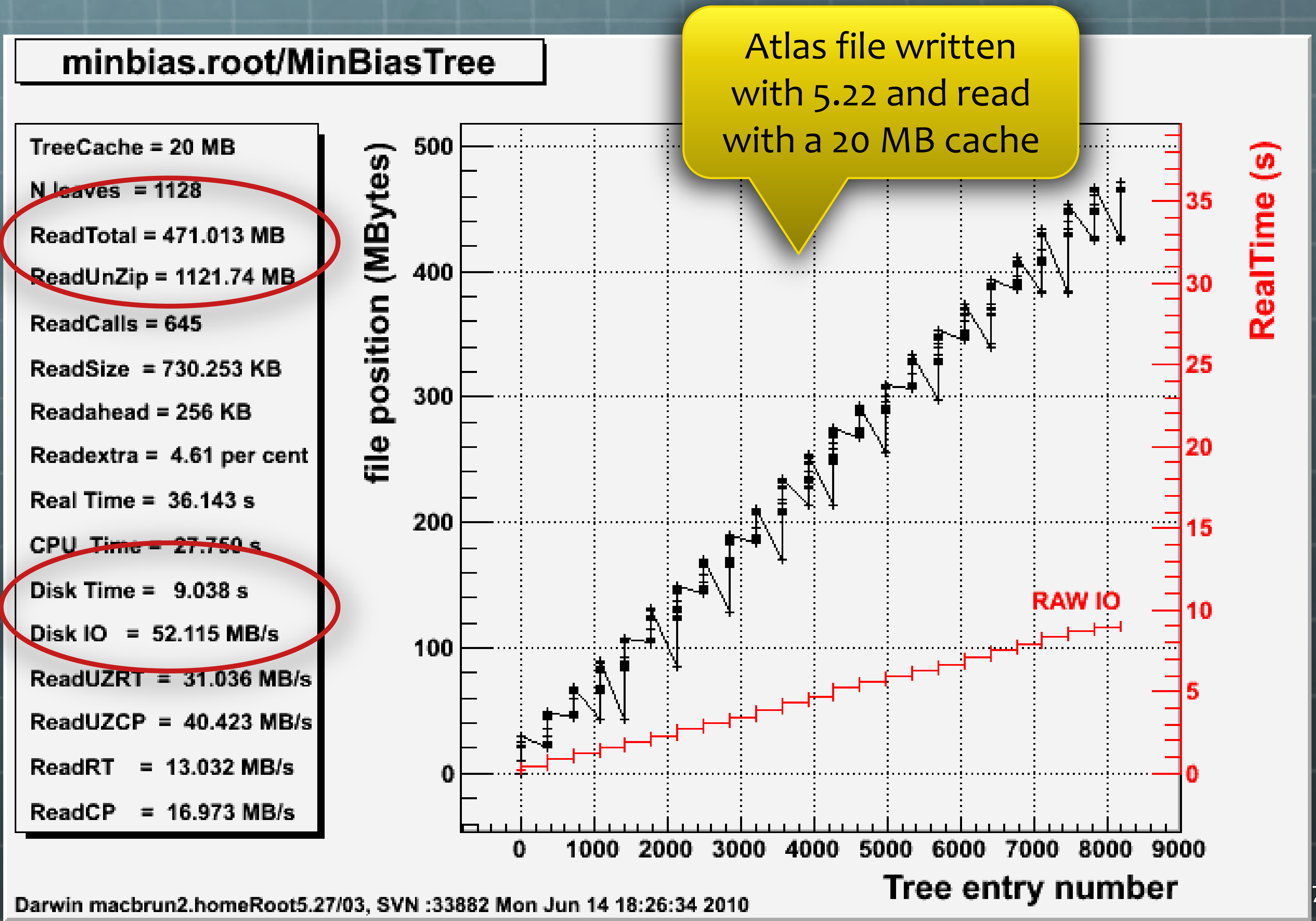
FlushBaskets

- TTree::FlushBaskets was introduced in 5.22 but called only once at the end of the filling process to disconnect the buffers from the tree header.
- In version 5.25/04 this function is called automatically when a reasonable amount of data (default is 30 Mbytes) has been written to the file.
- The frequency to call TTree::FlushBaskets can be changed by calling TTree::SetAutoFlush.
- The first time that FlushBaskets is called, we also call OptimizeBaskets.

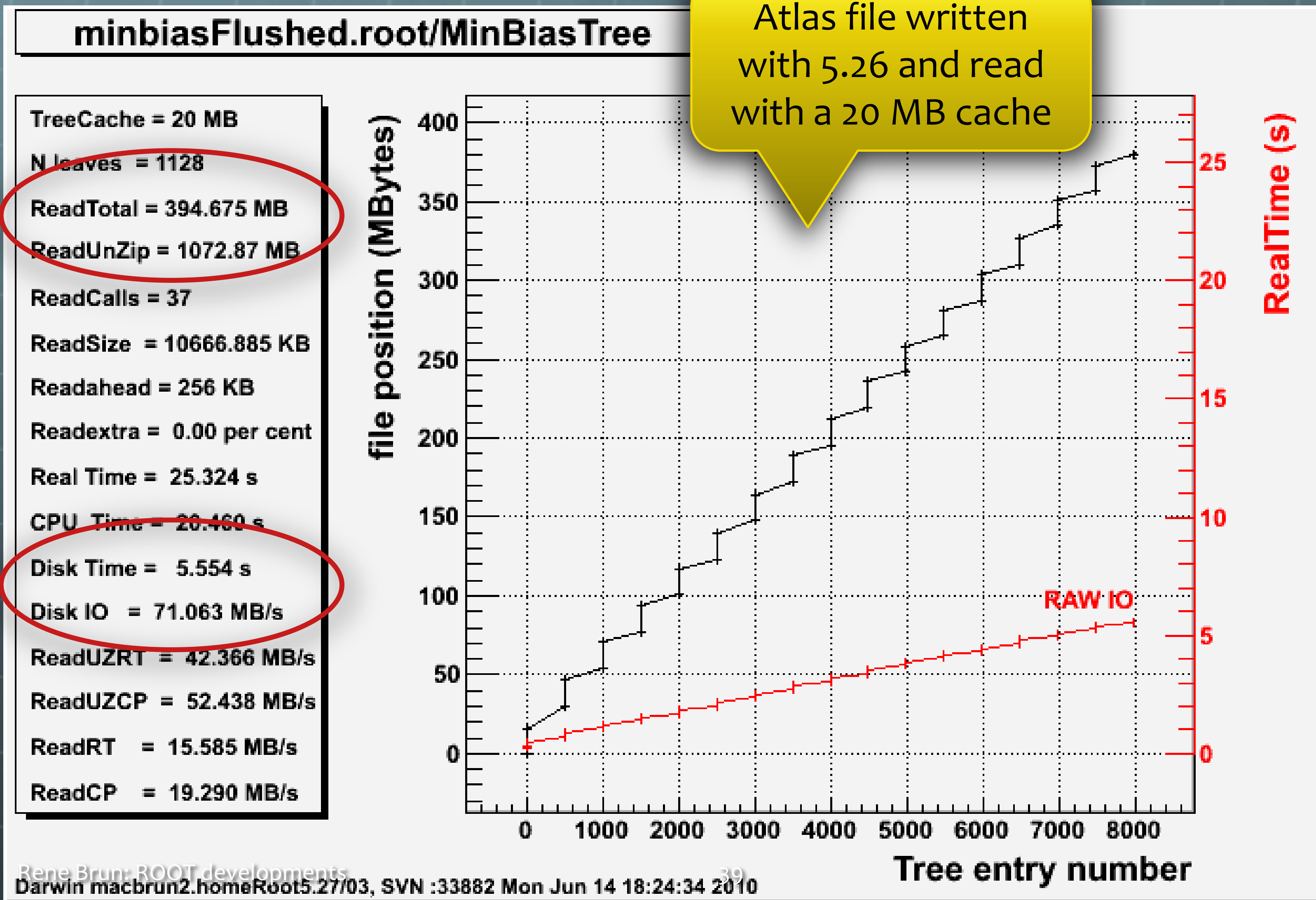
FlushBaskets 2

- The frequency at which FlushBaskets is called is saved in the Tree (new member fAutoFlush).
- This very important parameter is used when reading to compute the best value for the TreeCache.
- The TreeCache is set to a multiple of fAutoFlush.
- Thanks to FlushBaskets there is no backward seeks on the file for files written with 5.25/04. This makes a dramatic improvement in the raw disk IO speed.

without FlushBaskets

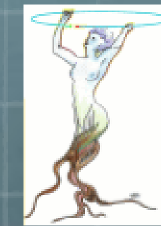


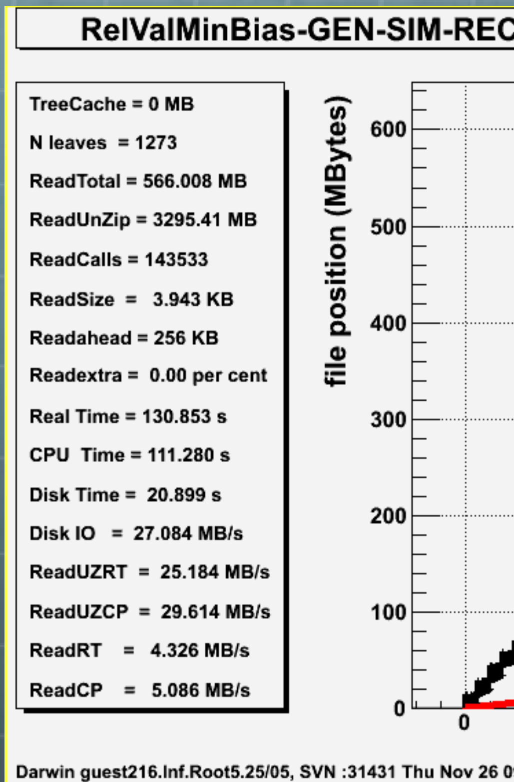
with FlushBaskets

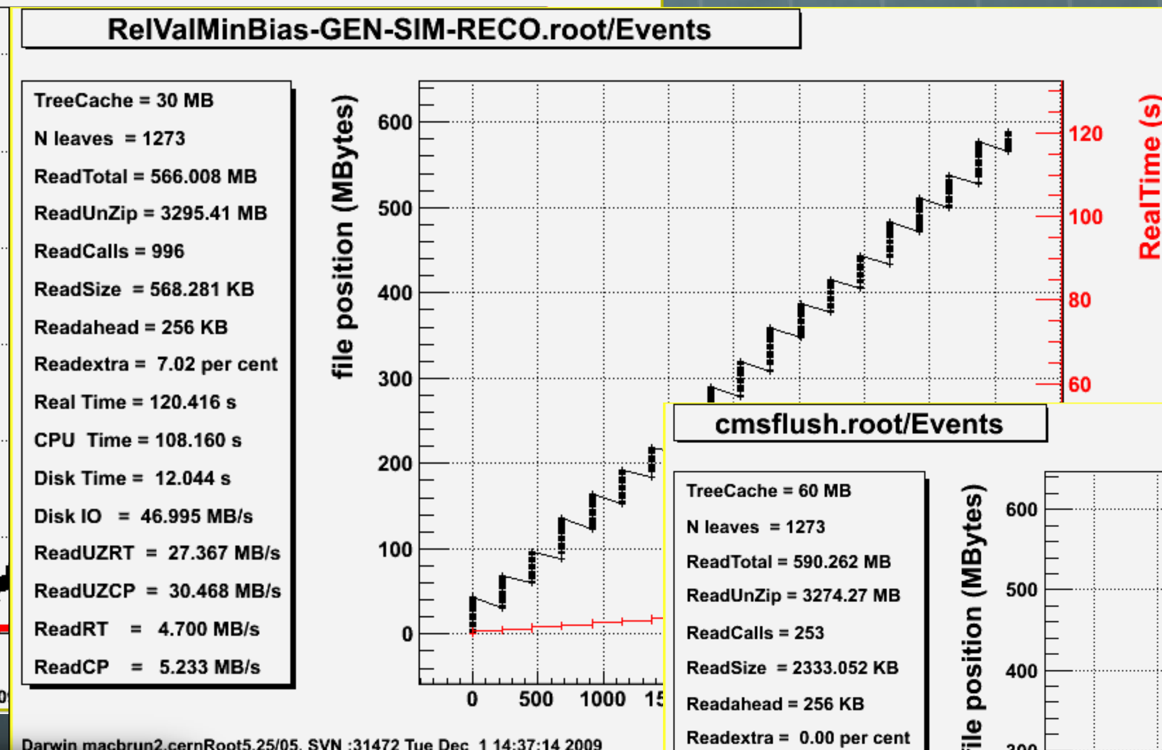


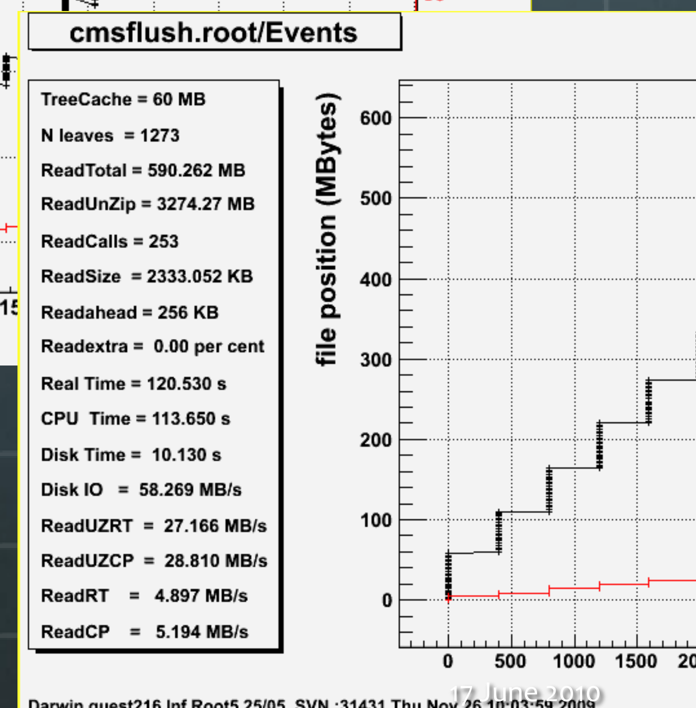
Atlas file written with 5.26 and read with a 20 MB cache

Similar pattern with CMS files









CMS : mainly CPU problem due to a complex object model

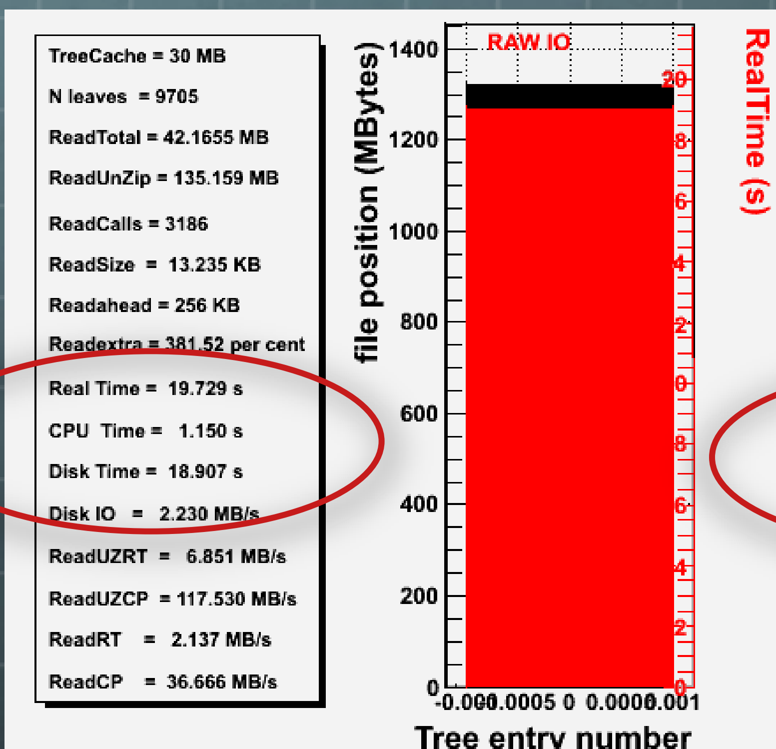
Use Case

reading 33 Mbytes out of 1100 MBytes

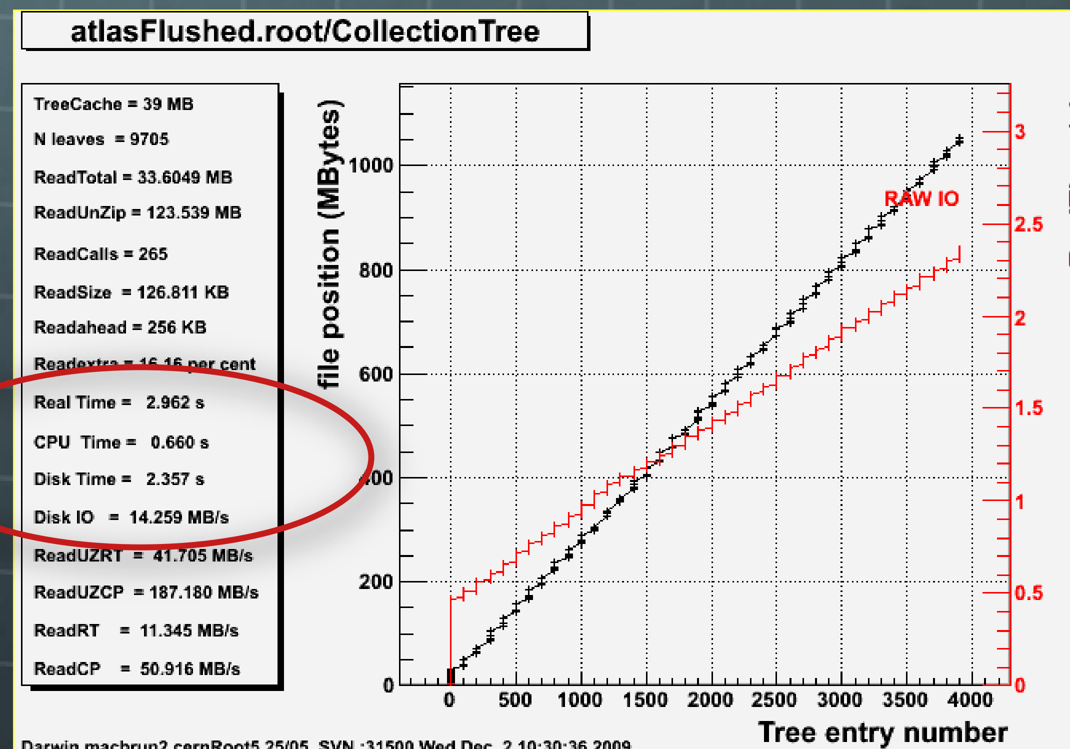
Seek time = 3186*5ms = 15.9s

Seek time = 265*5ms = 1.3s

Old ATLAS file



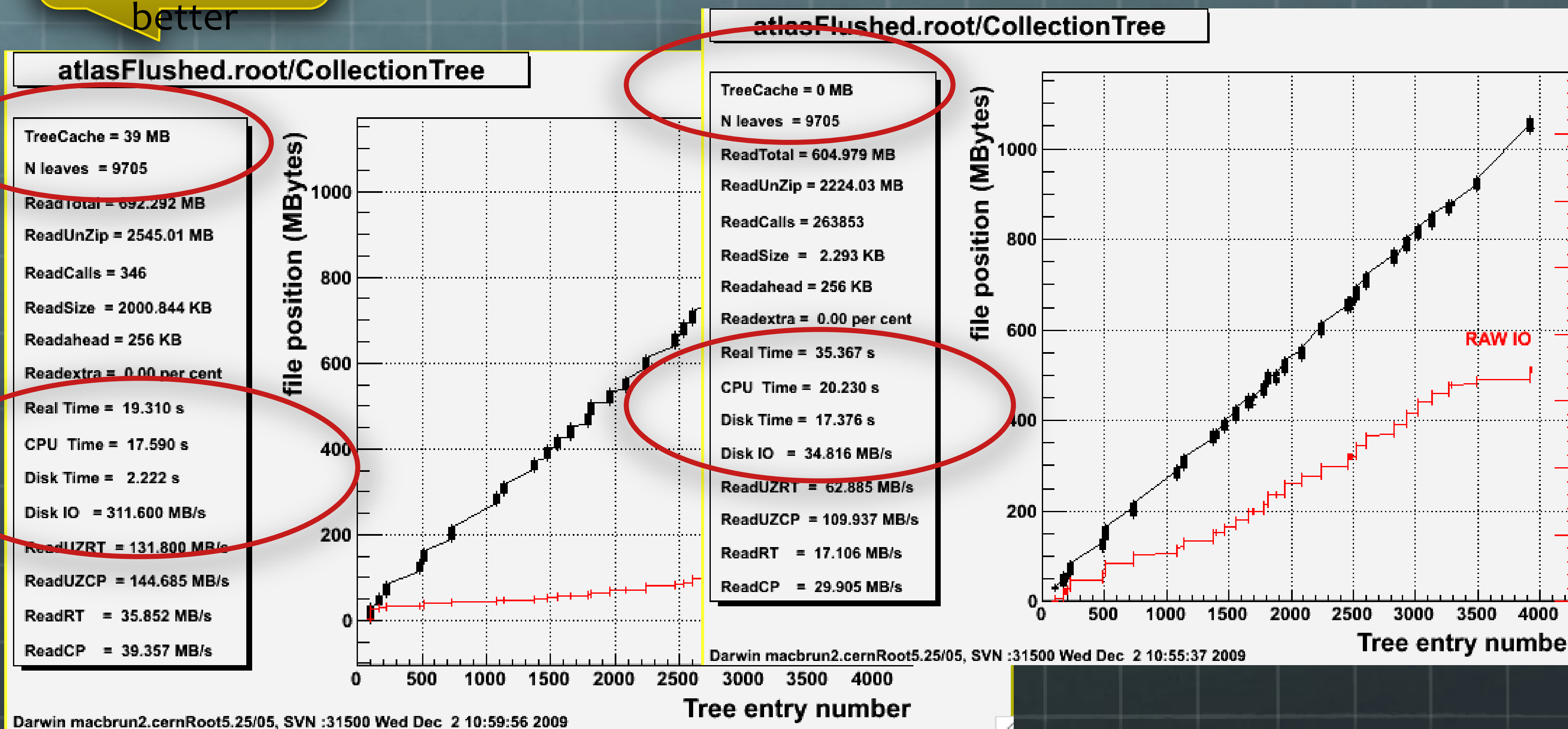
New ATLAS file



Use Case

reading 20% of the events

Even in this difficult case cache is better



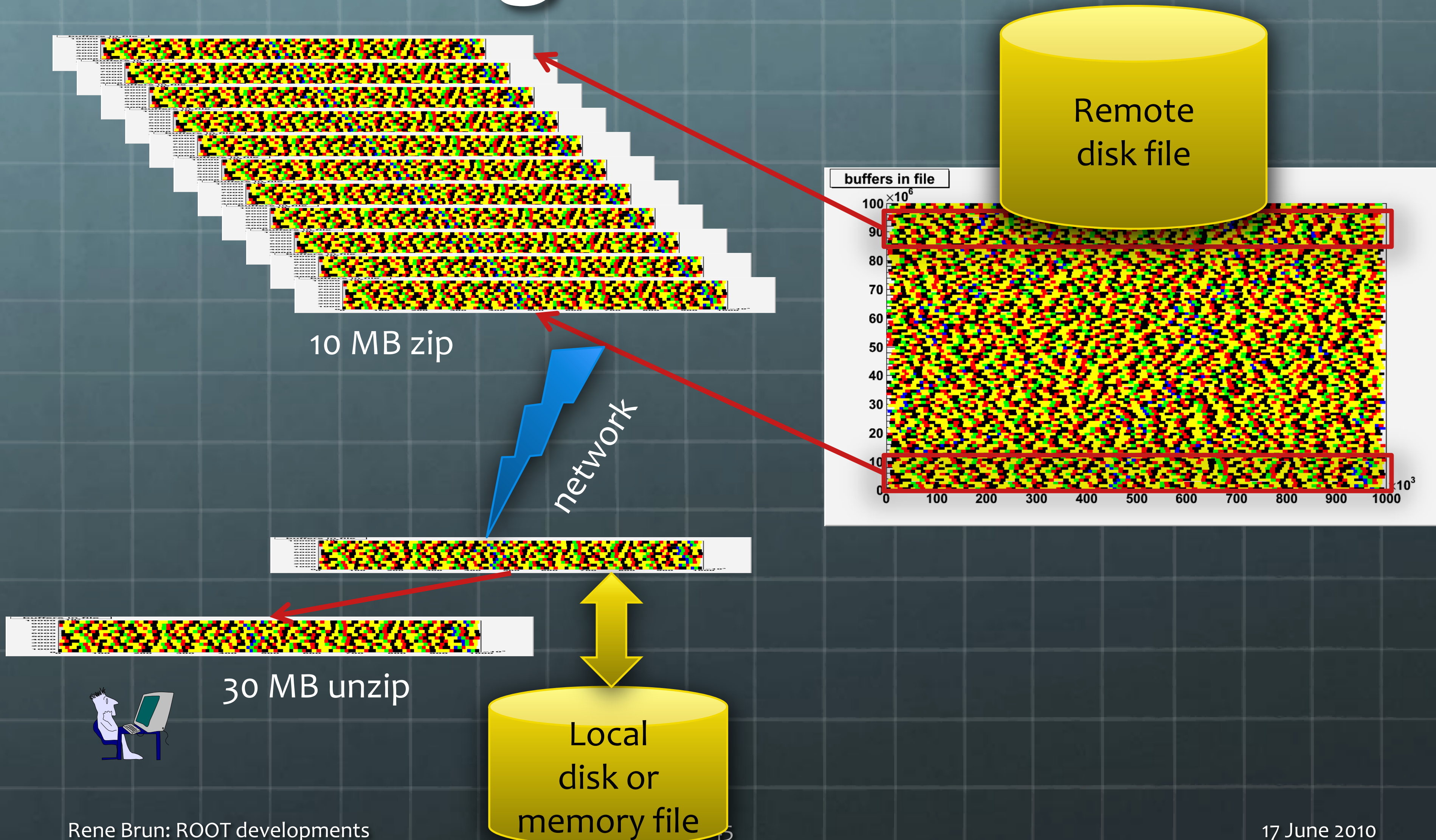
Caching a remote file

- ROOT can write a local cache on demand of a remote file. This feature is extensively used by the ROOT stress suite that read many files from root.cern.ch
  - TFile f(http://root.cern.ch/files/CMS.root","cacheread");
- The CACHEREAD option opens an existing file for reading through the file cache. If the download fails, it will be opened remotely. The file will be downloaded to the directory specified by SetCacheFileDir().

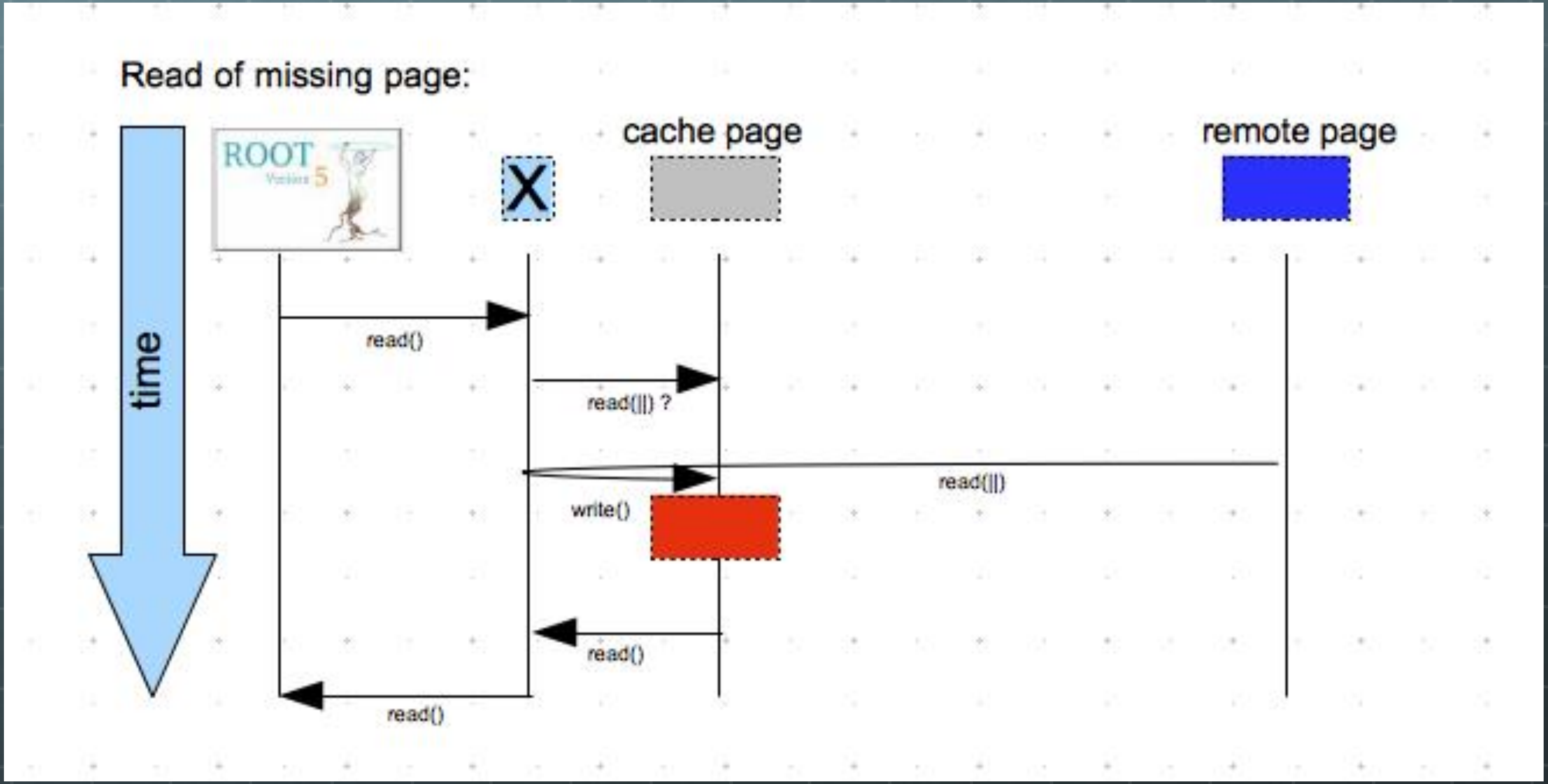
Caching the TreeCache

- The TreeCache is mandatory when reading files in a LAN and of course a WAN. It reduces by a factor 10000 the number of network transactions.
- One could think of a further optimization by keeping locally the TreeCache for reuse in a following session.
- A prototype implementation (by A.Peters) is currently being tested and looks very promising.
- A generalisation of this prototype to fetch treecache buffers on proxy servers would be a huge step forward.

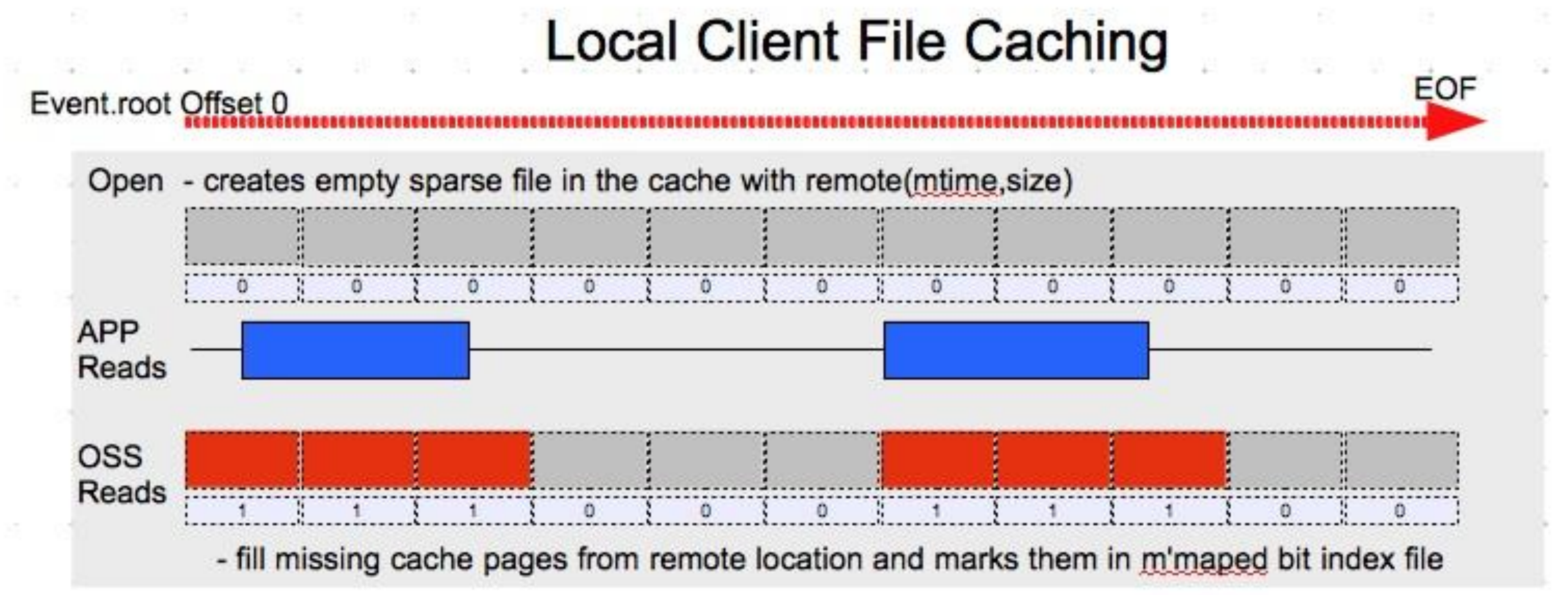
Caching the TreeCache



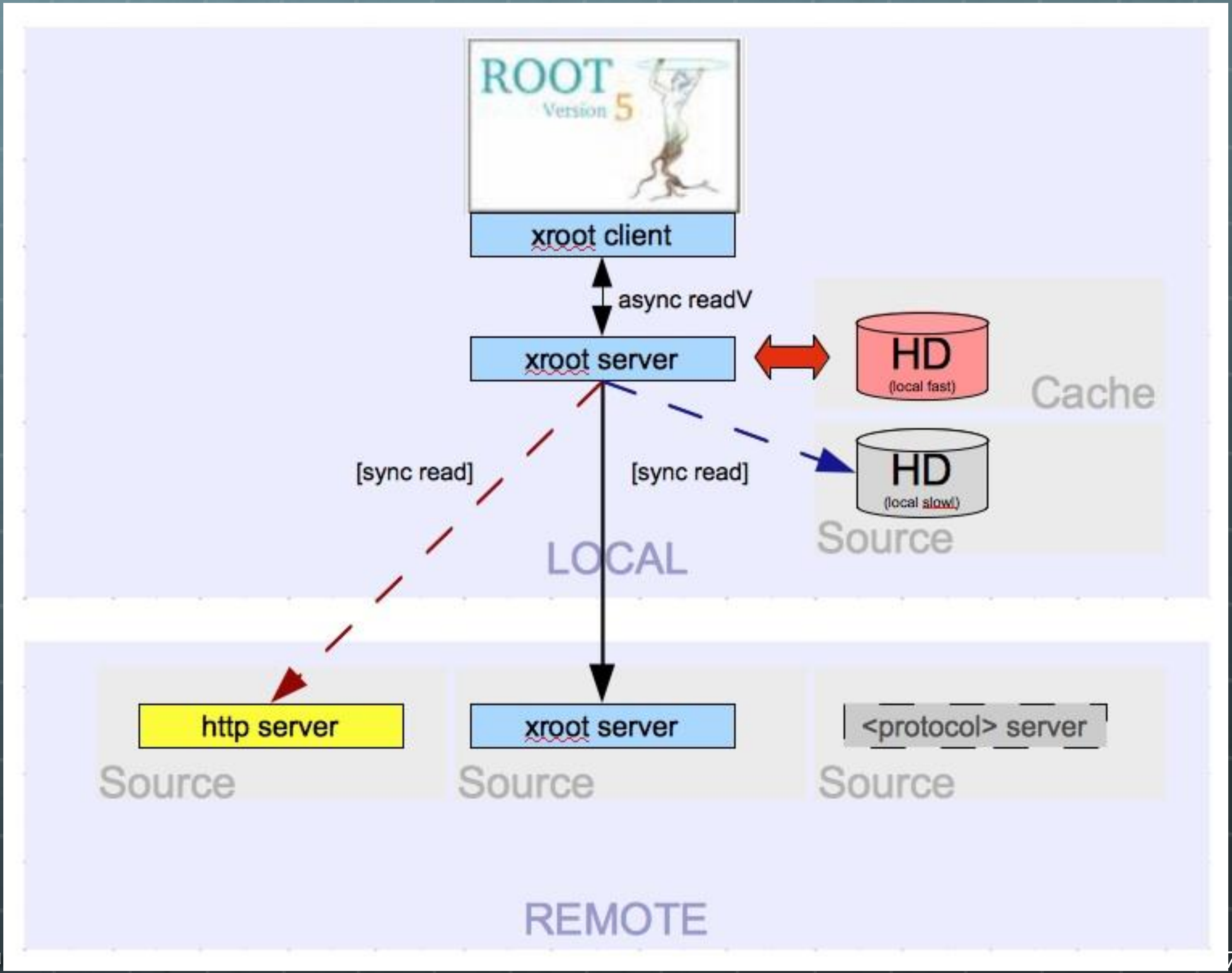
A.Peters cache prototype



A.Peters cache prototype



A.Peters cache prototype



caching the TreeCache Preliminary results

results on an Atlas AOD 1 GB file
with preliminary cache
from Andreas Peters

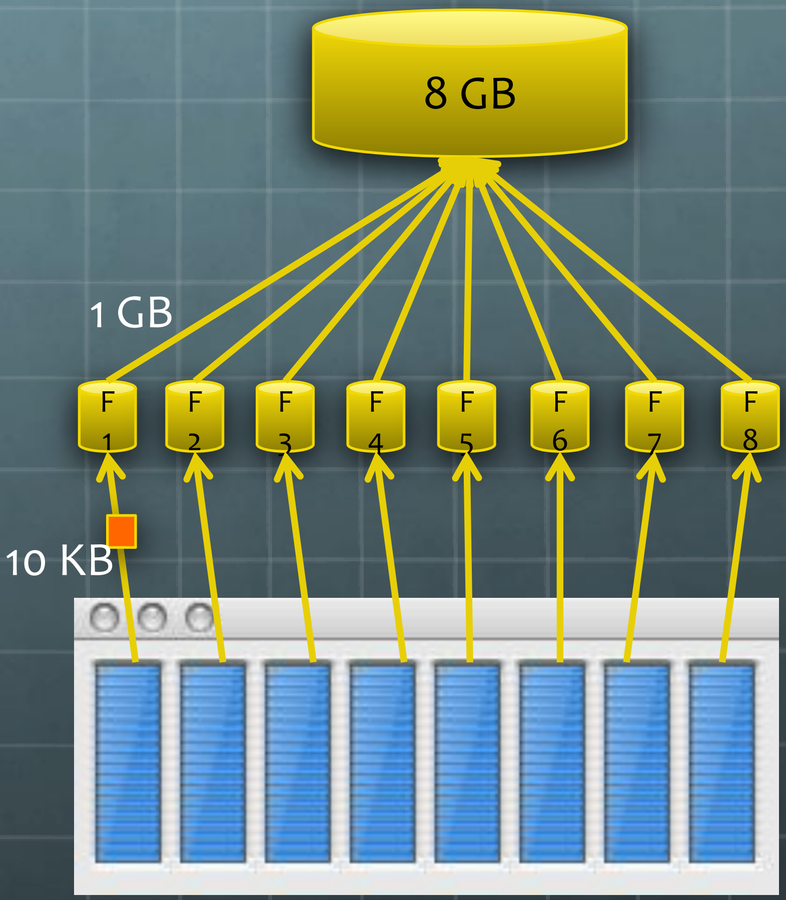
very
encouraging
results

session	Real Time(s)	Cpu Time (s)
local	116	110
remote xrootd	123.7	117.1
with cache (1st time)	142.4	120.1
with cache (2nd time)	118.7	117.9

other improvements

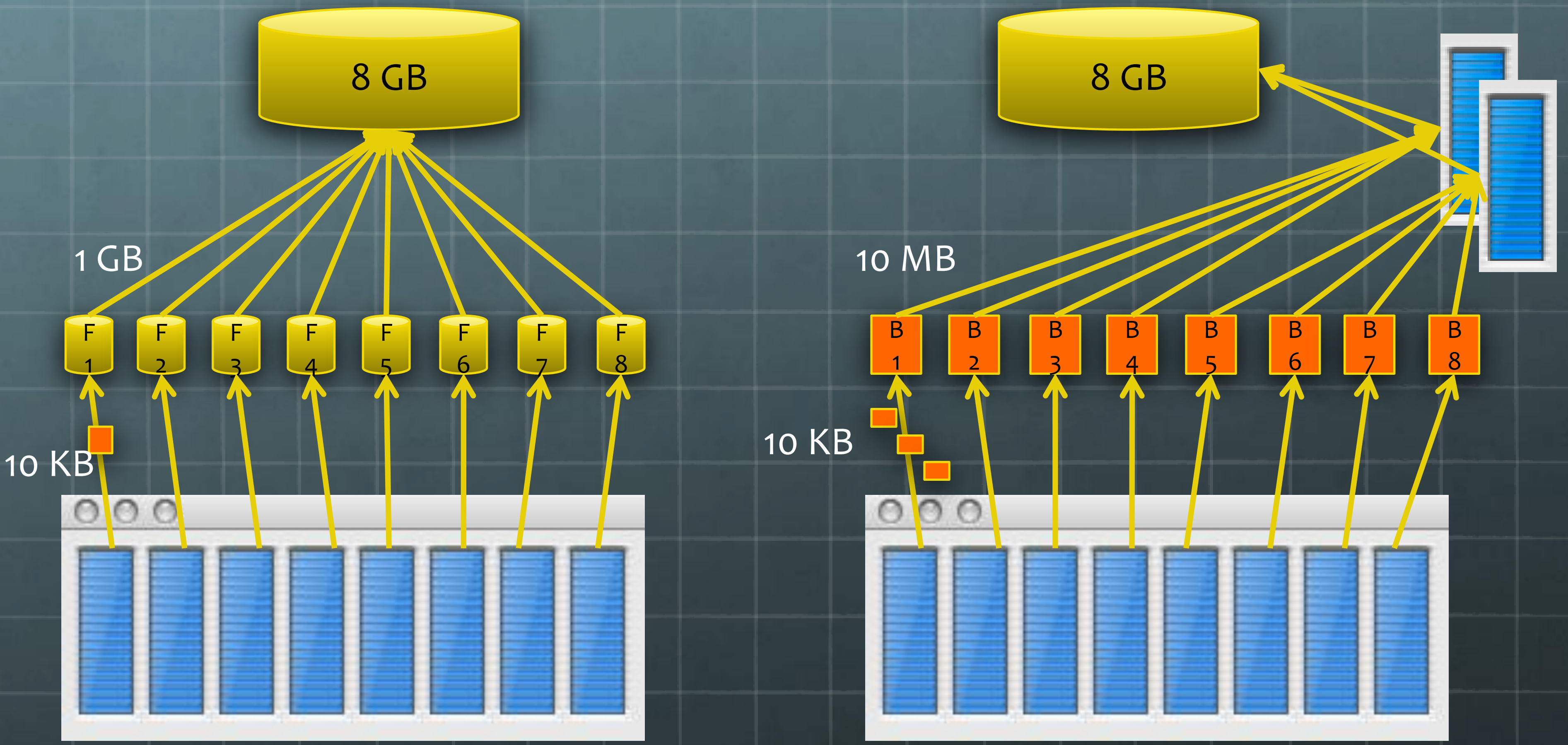
- Code optimization to reduce the CPU time for IO
- Use of memory pools to reduce malloc/free calls and in particular memory fragmentation. The use of memory pools could be extended automatically to include user data structures, the main cause for memory fragmentation.
- working on parallel buffers merge, a very important requirement for multi/many core systems

Parallel buffers merge



- parallel job with 8 cores
- each core produces a 1 GB file in 100 seconds.
- Then assuming that one can read each file at 50MB/s and write at 50 MB/s, merging will take 8*20+160 = 320s !!
- One can do the job in <160s

Parallel buffers merge



Summary

- After 15 years of developments, we are still making substantial improvements in the IO system thanks to the many use cases and a better understanding of the chaotic user analysis.
- We believe that file access in a WAN with local caches and proxys is the way to go. This will have many implications , including a big simplification of the data management.
- We are preparing the ground to make an efficient use of many-core systems.