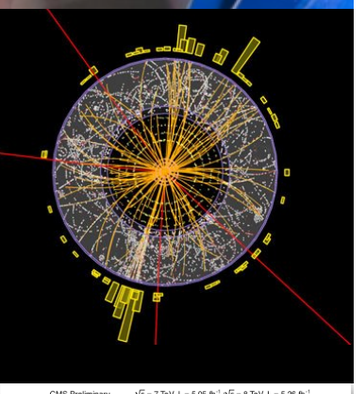
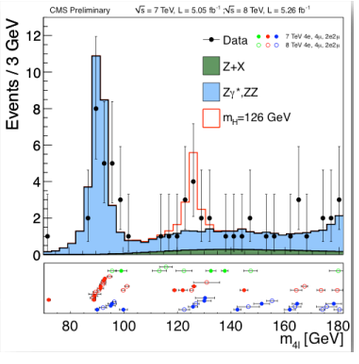
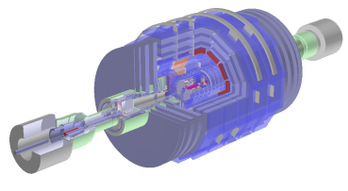






ROOT I/O Review and Future Plans

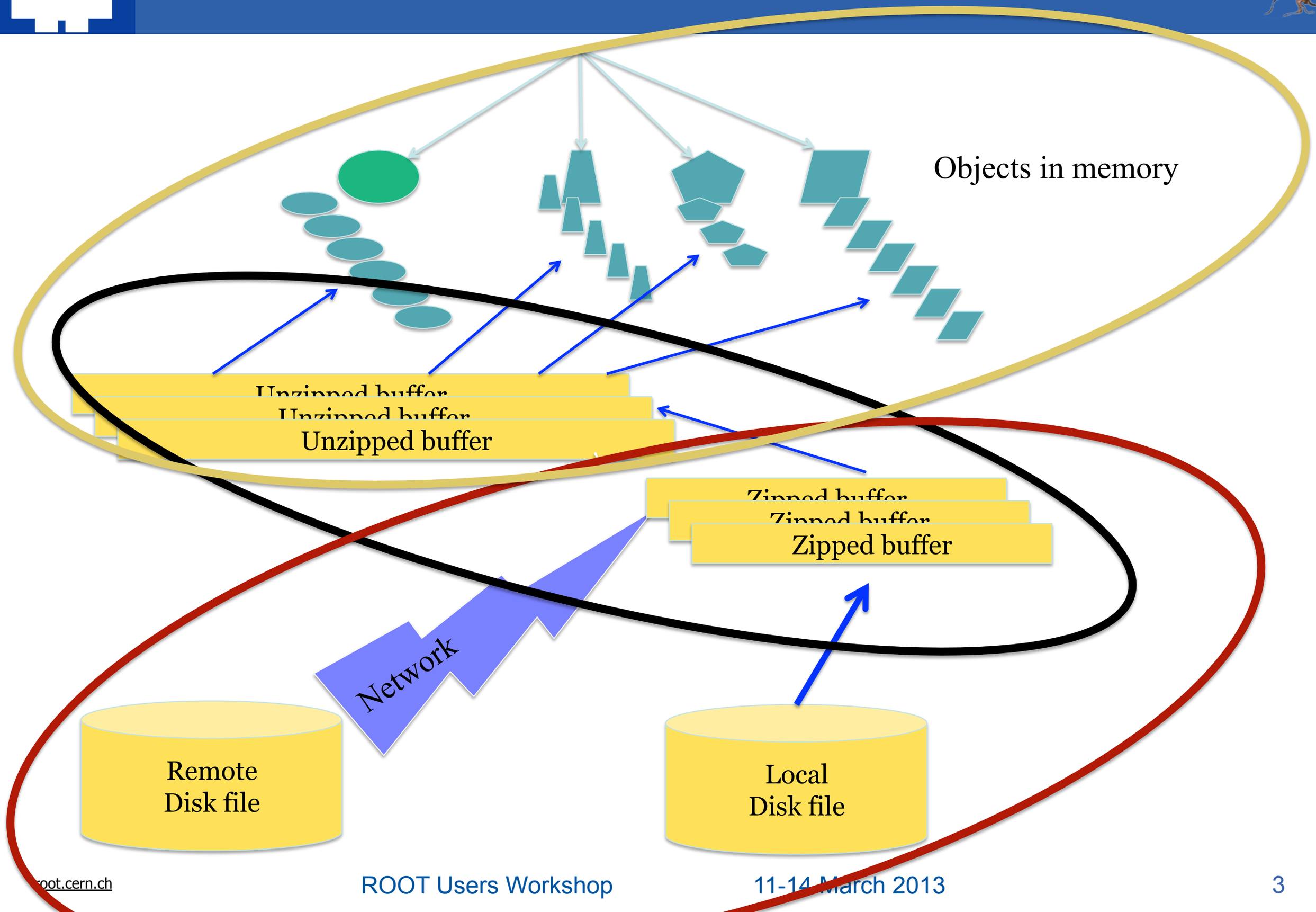
Philippe Canal
Fermilab

Philippe CANAL
root.cern.ch

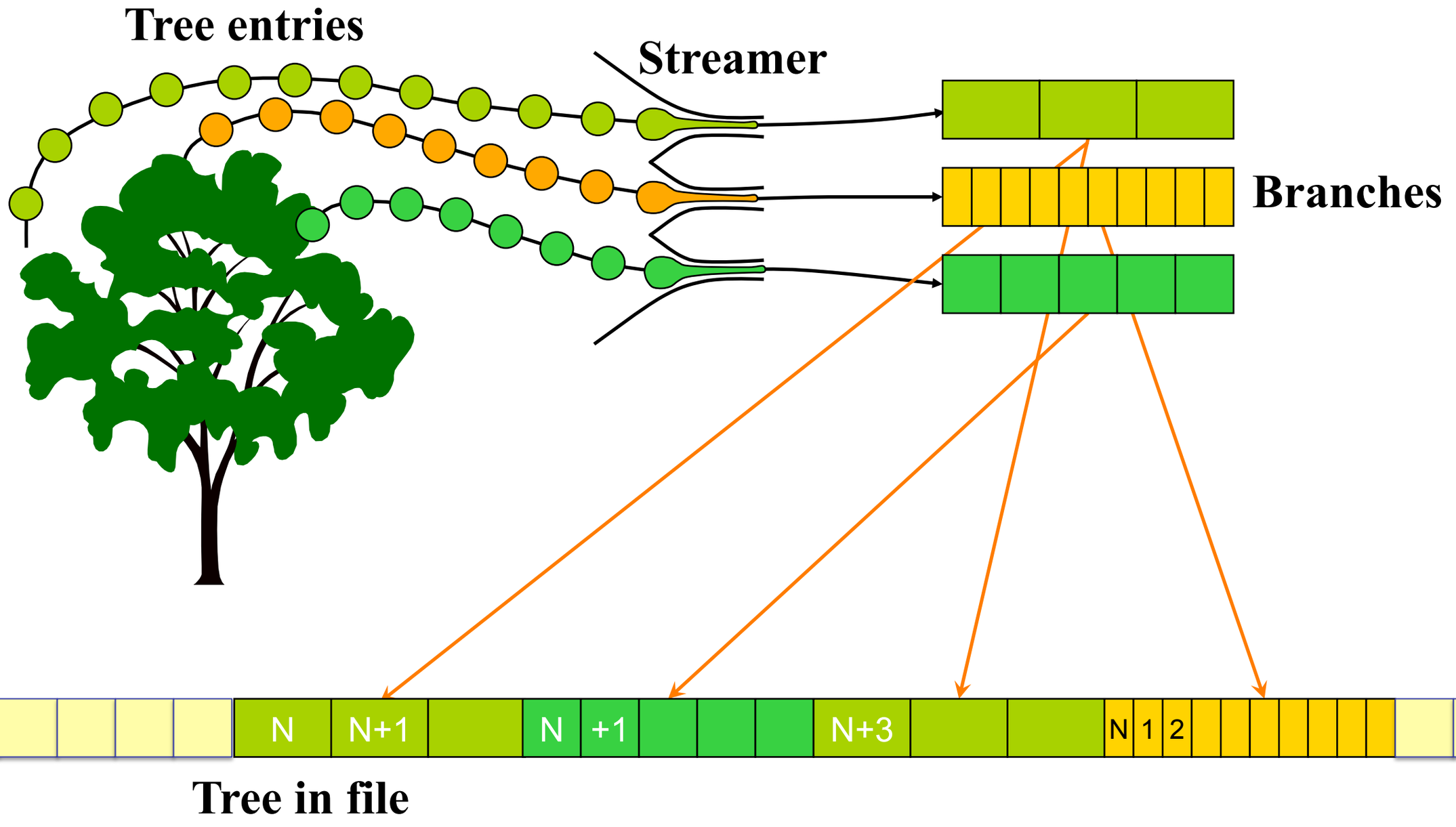
Overview

- TTreeCache
- I/O Customization Rules.
- Here comes cling
- ROOT I/O and Parallel Processing
  - Parallel Merge
  - Multi-threading

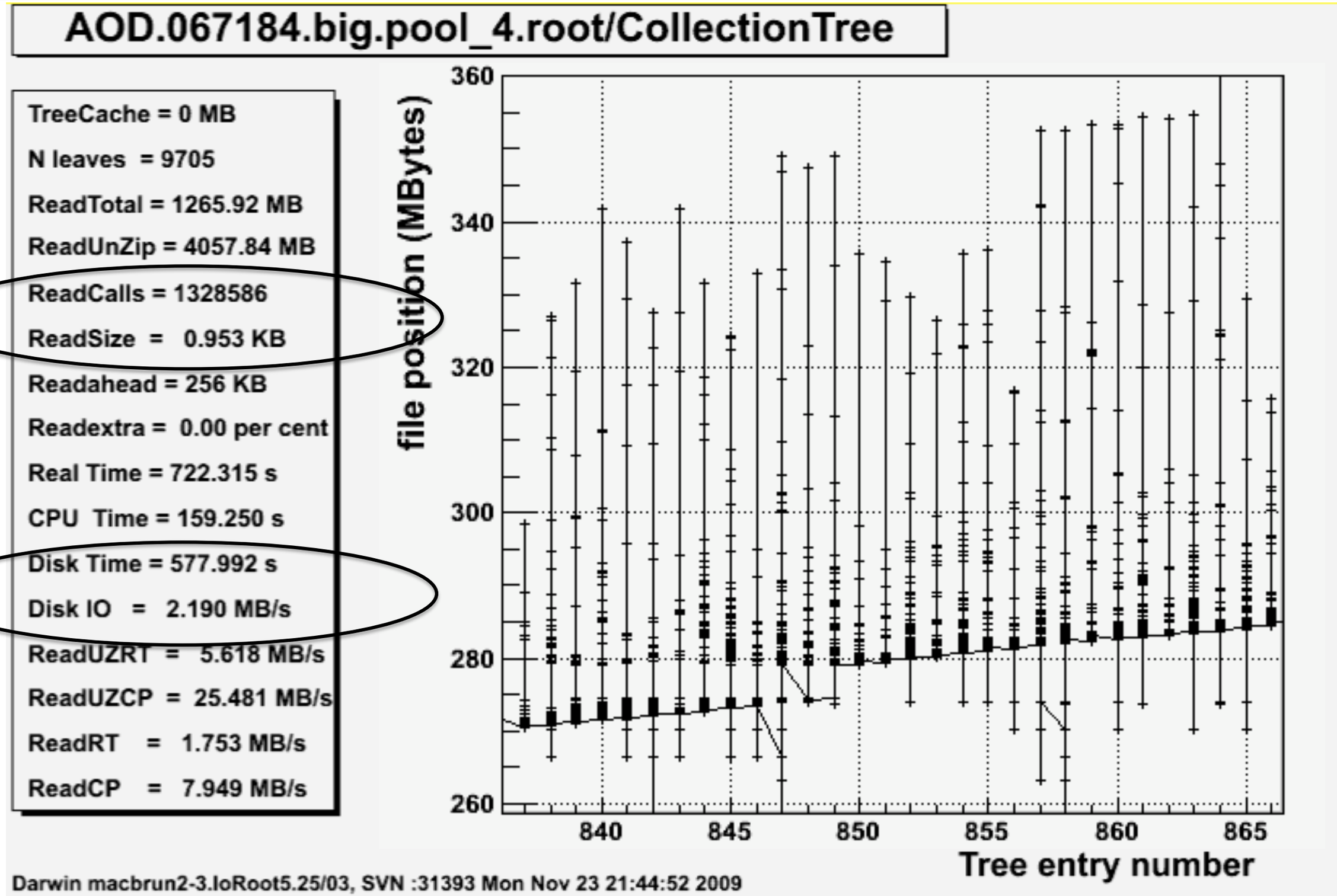
ROOT I/O Landscape



ROOT I/O – Branches And Baskets



Without Prefetching Baskets



Without Prefetching Baskets

- Default was for all buffers to have the same size.
- 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.
- Without ***TTreeCache***:
  - Many small reads.
- When reading with prefetching there were still inefficiencies:
  - Backward seeks.
  - Gap in reads.
- Hand tuning the baskets which was feasible with a dozens branches became completely impracticable for ***TTree*** with more than 10000 branches.

Solution: *TTreeCache*

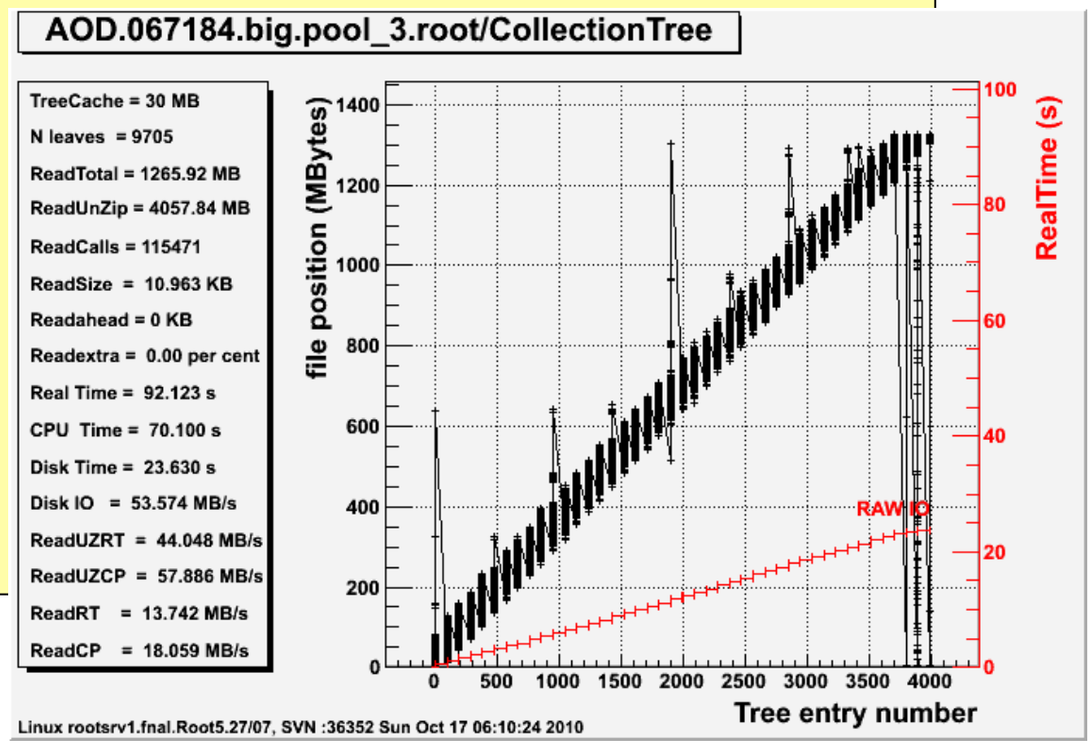
- Prefetches and caches a set of baskets (from several branches).
- Designed to reduce the number of file reads (or network messages) when reading a ***TTree*** (by a factor of 10000).

```
T->SetCacheSize(cachesize);
if (cachesize != 0) {
   T->SetCacheEntryRange(efirst,elast);
   T->AddBranchToCache(data_branch,kTRUE); // Request all the sub branches too
   T->AddBranchToCache(cut_branch,kFALSE);
   T->StopCacheLearningPhase();
}
```

- Configuration
  - Size of the reserved memory area
  - Set of branches to be read or range of entry to learn
  - Range of entry to read

TTreePerfStats

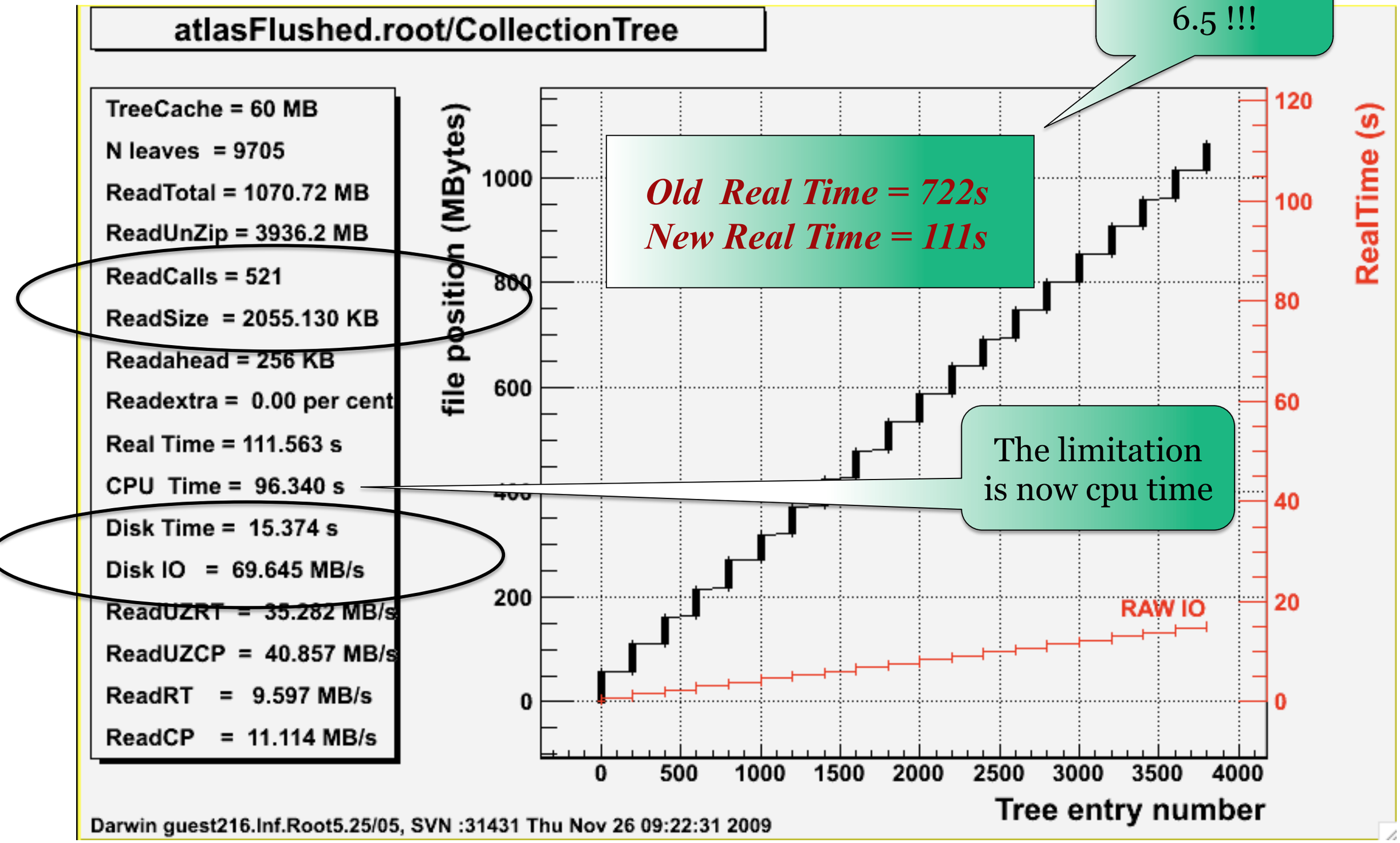
```
void taodr(Int_t cachesize=10000000) {
   gSystem->Load("aod/aod");  //shared lib generated with TFile::MakeProject
   TFile *f = TFile::Open("AOD.067184.big.pool.root");
   TTree *T = (TTree*)f->Get("CollectionTree");
   Long64_t nentries = T->GetEntries();
   T->SetCacheSize(cachesize);
   T->AddBranchToCache("*",kTRUE);

   TTreePerfStats ps("ioperf",T);

   for (Long64_t i=0;i<nentries;i++) {
      T->GetEntry(i);
   }
   T->PrintCacheStats();
   ps.SaveAs("aodperf.root");
}
```




```
******TreeCache statistics for file: AOD.067184.big.pool_3.root ******
Number of branches in the cache ...: 9705
Cache Efficiency ..................: 0.973247
Cache Efficiency Rel...............: 1.000000
Learn entries......................: 100
Reading............................: 1265967732 bytes in 115472 transactions
Readahead..........................: 0 bytes with overhead = 0 bytes
Average transaction................: 10.963417 Kbytes
Number of blocks in current cache..: 3111, total size: 2782963
```

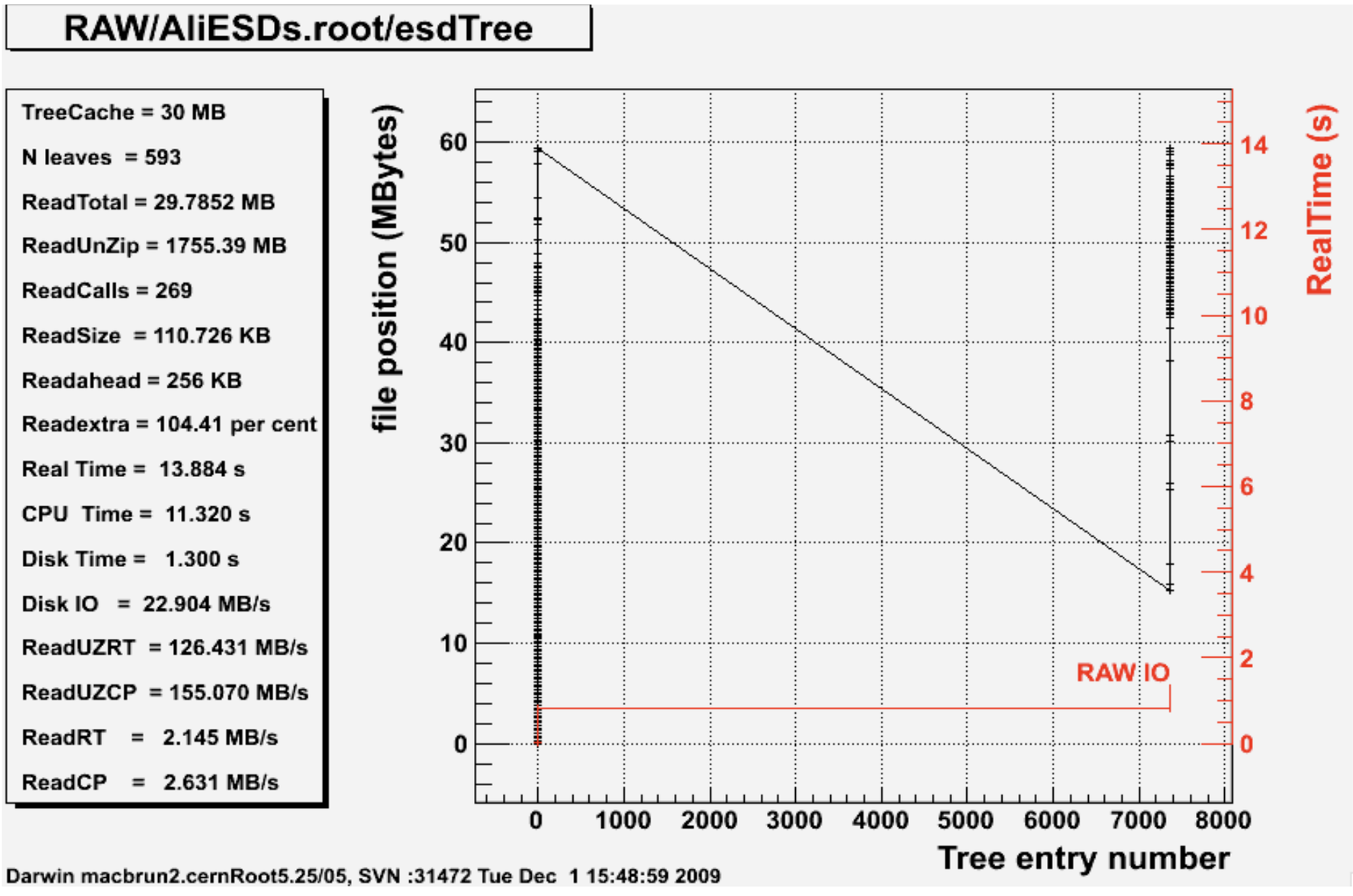
Root > TFile f("aodperf.root")
Root > ioperf.Draw()

With *TTreeCache*

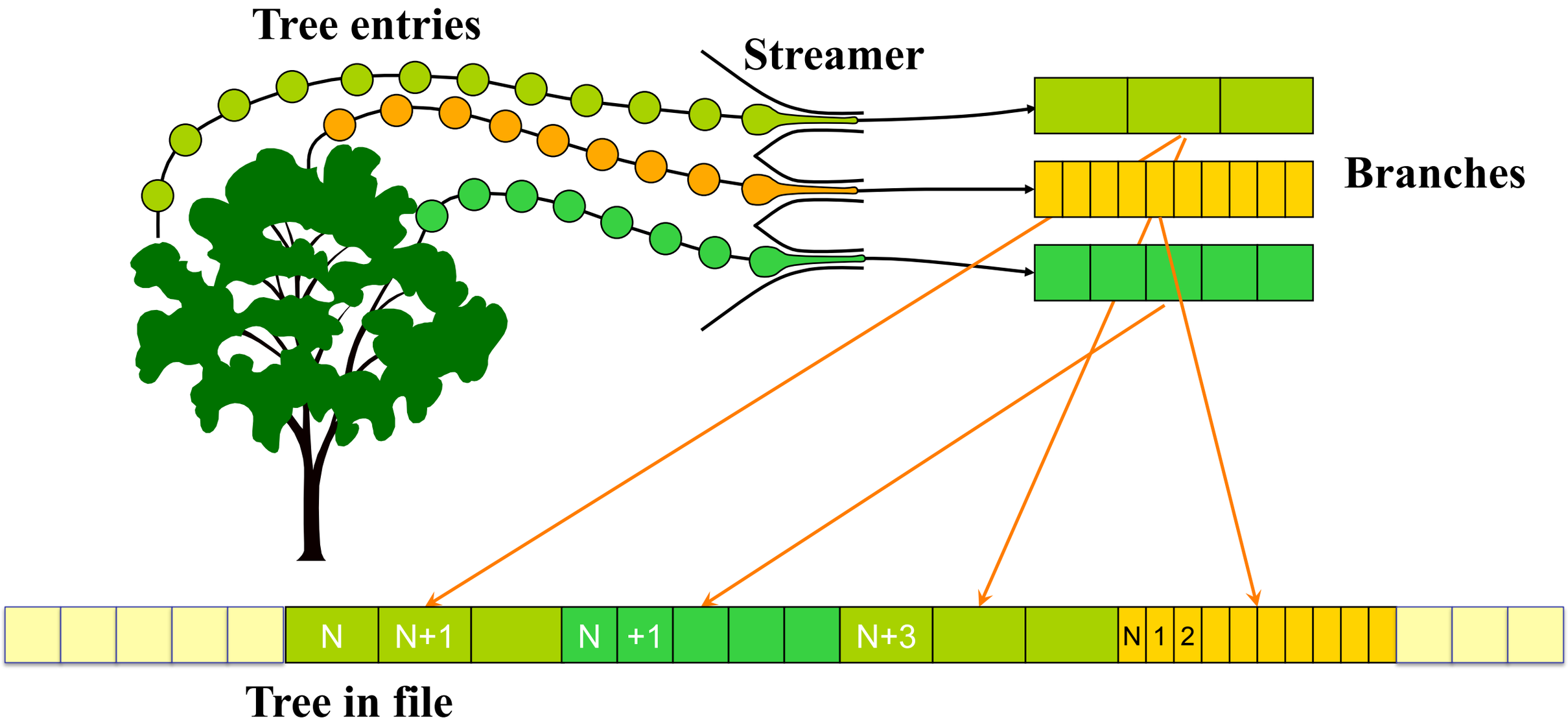


Better But ....

- Sequential read still requires some backward file seeks.
- Still a few interleaved reads.



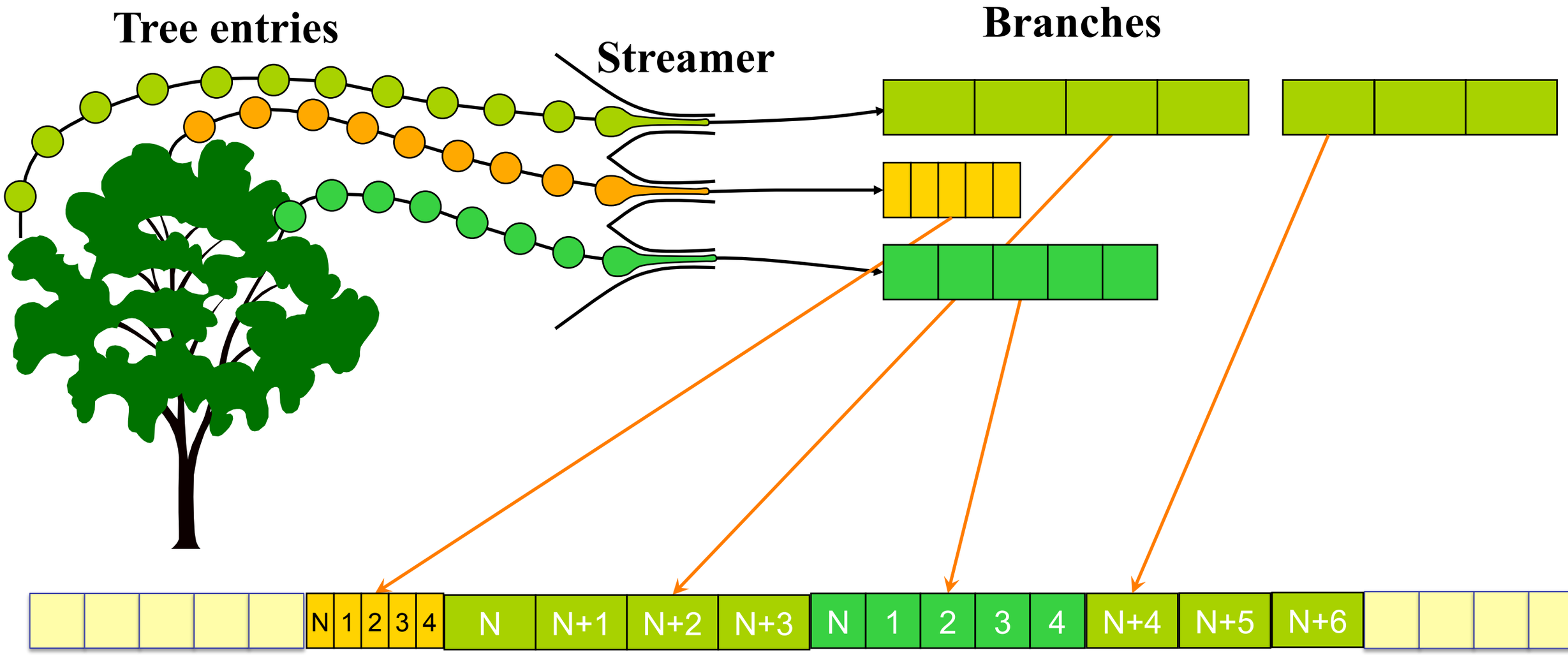
ROOT I/O – Default Basket sizes



OptimizeBaskets

- Improve Basket Size:
  - Default basket size was the same for all branches and tuning the size by hand is very time consuming.
  - ***TTree::OptimizeBaskets*** resizes the basket to even out the number of entries per basket in all the branches and reduce the total memory use.

ROOT I/O -- *OptimizeBaskets*



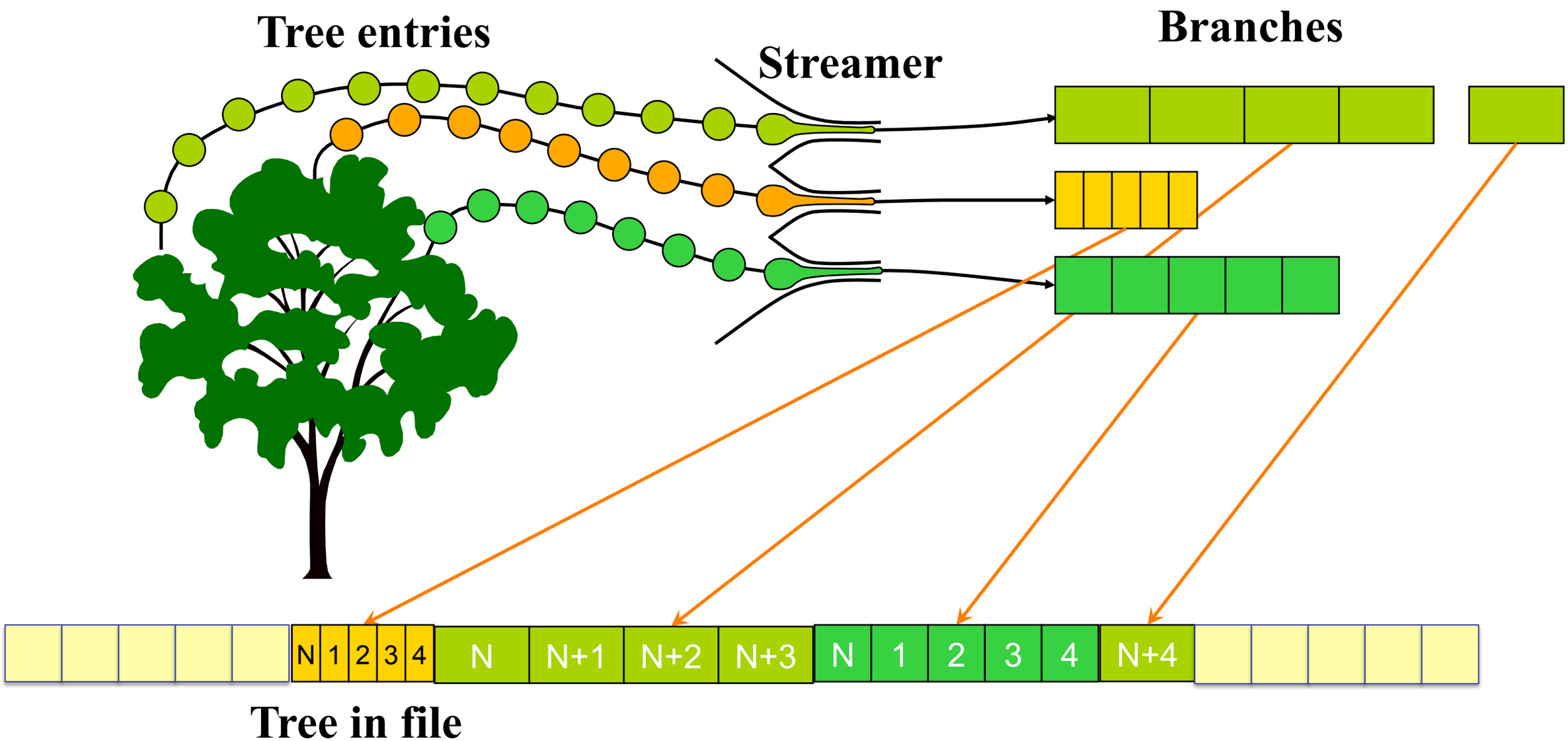
Clustering (*AutoFlush*)

Enforce "clustering":

- Once a reasonable amount of data (default is 30 Mbytes) has been written to the file, all baskets are flushed out and the number of entries written so far is recorded in ***fAutoFlush***.
- From then on for every multiple of this number of entries all the baskets are flushed.
- This insures that the range of entries between 2 flushes can be read in one single file read.
- The first time that ***FlushBaskets*** is called, we also call ***OptimizeBaskets***.
- The ***TTreeCache*** is always set to read a number of entries that is a multiple of ***fAutoFlush*** entries.

No backward seeks needed to read file.
Dramatic improvement in the raw disk IO speed.

ROOT I/O -- Split/Cluster

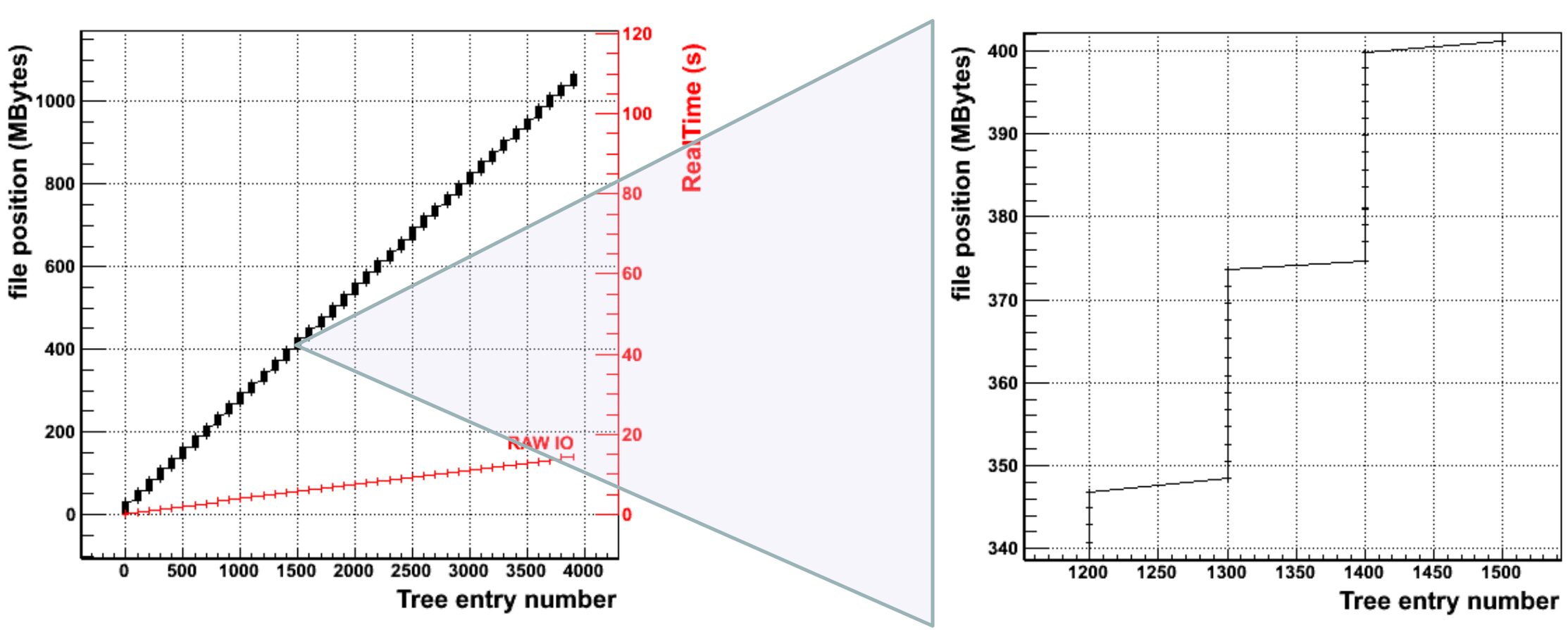


OptimizeBaskets, *AutoFlush*

These solutions are available since ***v5.26***.

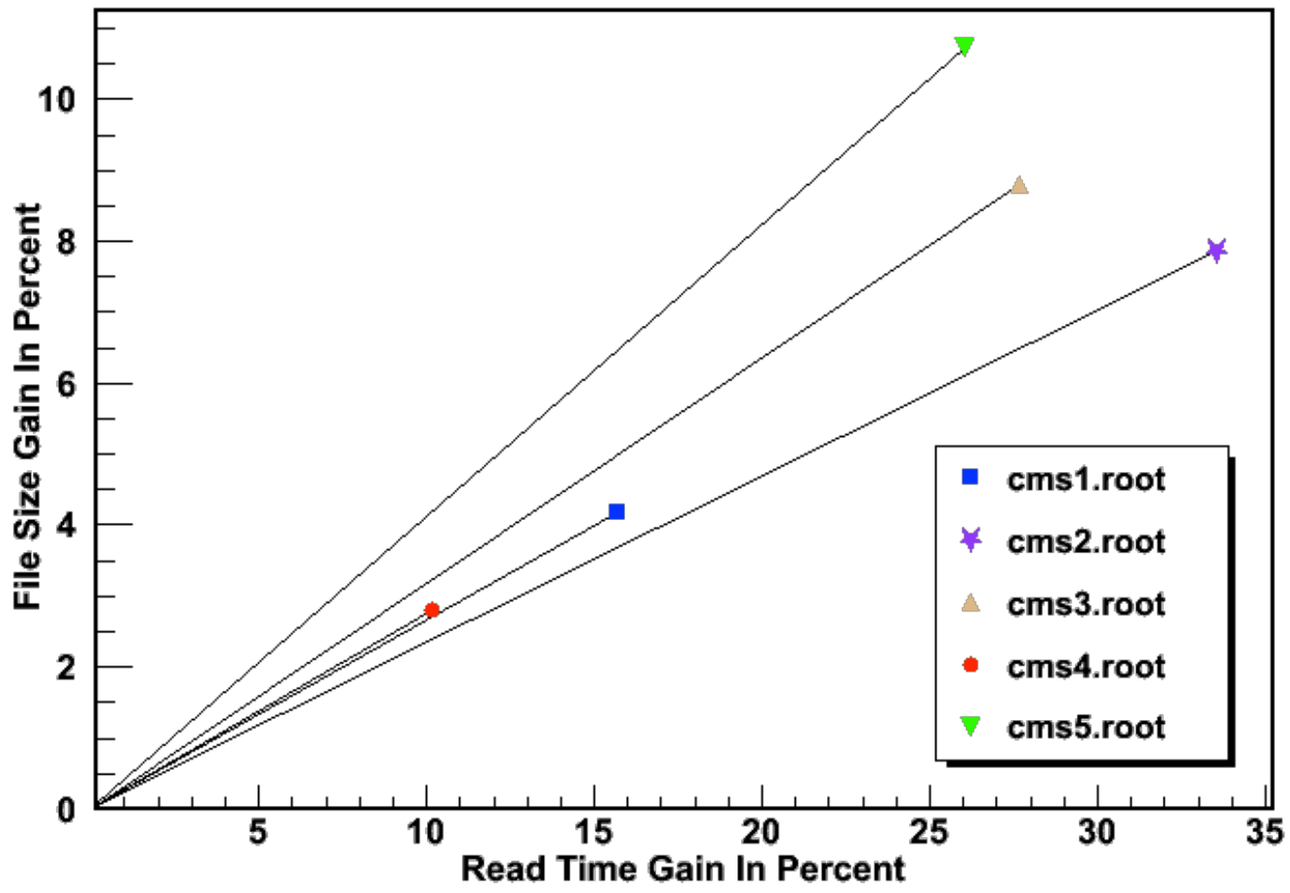
- Automatically tweak basket size.
- Flush baskets at regular intervals.

→ ***Greater performance!***



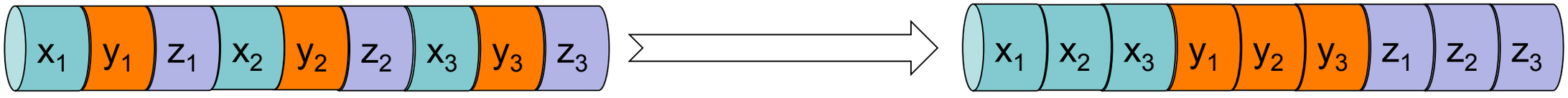
Memberwise Streaming

- Used for split collection inside a ***TTree***.
- The ***default*** for streaming collections even when not split.
- Better compression, faster read time.



Results for CMS files,

- some fully split
- some unsplit



Apples And Oranges

Simple Automatic Schema Evolution.

- Easily lets you transform  into 

Hand Coded Schema Evolution

- Allows to transform  into 

- Requires specific coding for each type of apple and orange.

NEW Complex Automatic Schema Evolution

- Allow almost any kind of transformation

- even  to 

Simple Automatic Schema Evolution

Support

- Changing the order of the members
- Changing simple data type (float to int)
- Adding or removing data members, base classes
- Migrating a member to base class

Limitations

- Handle only removal, addition of members and change in simple type
- Does not support complex change in type, change in semantic (like units)
- Further customization requires using a Streamer function
  - Allow complete flexibility including setting transient members

However they can ***NOT*** be used for member-wise streaming (TTrees)

Complex Automatic Schema Evolution

I/O customization rules solves many existing limitations

- Assign values to transient data members
- Rename classes
- Rename data members
- Change the shape of the data structures or convert one class structure to another
- Change the meaning of data members
- Ability to access the ***TBuffer*** directly when needed
- Ensure that the objects in collections are handled in the same way as the ones stored separately
- Transform data before writing

☞ ***Make things operational also in bare ROOT mode***

☞ ***Supported in object-wise, member-wise and split modes.***

Complex Automatic Schema Evolution

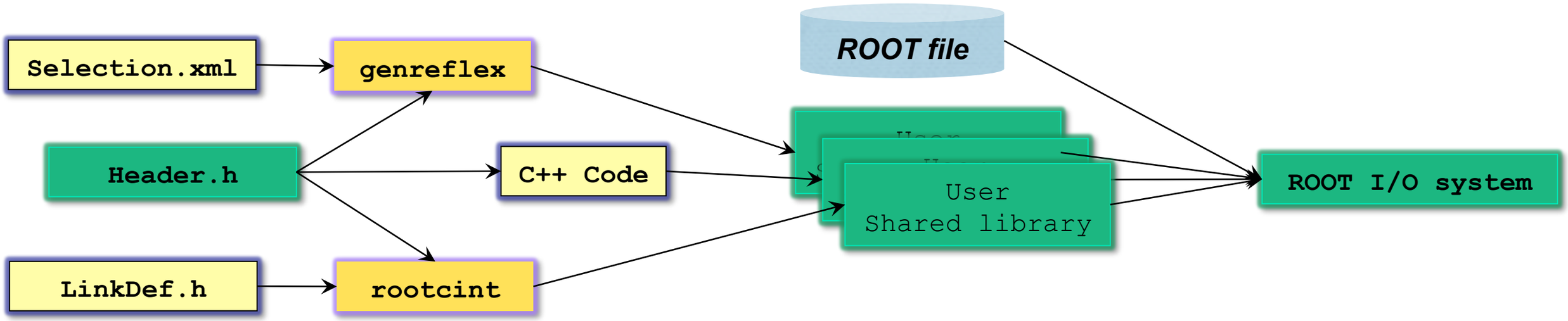
User can now supply a function to convert individual data members from disk to memory and rule defining when to apply the rules

A schema evolution rule is composed of:

- ***sourceClass***; version, checksum: identifier of the on disk class
- ***targetClass***: name of the class in memory
- ***source***: list of type and name of the on disk data member needed for the rule
- ***target:*** list of in memory data member modified by the rules.
- ***include***: list header files needed to compile the conversion function
- ***code***: function or code snippet to be executed for the rule

Rules can be registered via:

- LinkDef.h, Selection.xml, C++ API (via TClass), ROOT files



Dictionary Generation Syntax

Example of registering a rule from a LinkDef file:

```
#pragma read sourceClass=”oldname” version="[1-]" checksum=“[12345,23456]” \
   source=”type1 var; type2 var2;" \
   targetClass=“newname” target=”var3" \
   include=“<cmath> <myhelper>” \
   code=”{ … ‘code calculating var3 from var1 and var2’ … }"
```

Example of registering a rule from a Selection.xml file:

```
<read sourceClass=”oldname” version="[4-5,7,9,12-]” checksum="[12345,123456]”
      source=”type1 var; type2 var2;”
      targetClass=”newname” target=”var3”
      include=“<cmath> <myhelper>”
<![CDATA[
   … ‘code calculating var3 from var1 and var2’ …
]]>
</read>
```

Setting A Transient Member

MyClass.h

```
class MyClass {
private:
  Type fComplexData;
  Double_t fValue; //! Calculated from fComplexData
  Bool_t fCached;  //! True if fValue has been calculated
public:
  double GetValue() { if (!fCached) { fValue = … ; }; return fValue; }
```

MyClassLinkDef.h

```
#pragma read sourceClass="MyClass" version="[1-]" source=""
   targetClass="MyClass" \
   target="fCached" \
   code="{ fCached = false; }"
```

This example shows how to initialize a transient member

`source=""` indicates that no input is needed

`version="[1-]` indicates that the rule applies to all versions of the class

`target="fCached"` indicates which member will be modified by the rule

This resolves the outstanding issues where transient members are currently not updated when (re-)reading an object from a split branch

Merging Several Data Members

MyClass.h

```
class MyClass {
private:
  int fX;
  int fY; // Values between 0 and 999
  int fZ; // Values between 0 and 9
public:
  int GetX() { return fX; }
  int GetY() { return fY; }
  ClassDef(MyClass,8);
}
```

MyClass.h

```
class MyClass {
private:
  long fValues; // Merging of fX, fY and fZ
public:
  int GetX() { return fValues / 1000; }
  int GetY() { return (fValues%1000)-GetZ(); }
  int GetZ() { return fValues % 10; }
  ClassDef(MyClass,9);
}
```

MyClassLinkDef.h

```
#pragma read sourceClass="MyClass" version="[8]" targetClass="MyClass " \
   source="int fX; int fY; int fZ" target="fValues" \
   code="{ fValues = onfile.fX*1000 + onfile.fY*10 + onfile.fZ; }"
```

In ***MyClass*** version 9, to save memory space, 3 data members were *merged*.

`source="int fX; … "` indicates the types and name of the original members.

`onfile.fX` gives access to the value of ***fX*** read from the buffer.

Renaming A Class

MyClass.h

```
class MyClass {
private:
  int fX;
  int fY; // Values between 0 and 999
  int fZ; // Values between 0 and 9
public:
  int GetX() { return fX; }
  int GetY() { return fY; }
  ClassDef(MyClass,8);
}
```

Properties.h

```
class Properties {
private:
  long fValues; // Merging of fX, fY and fZ
public:
  int GetX() { return fValues / 1000; }
  int GetY() { return (fValues%1000)-GetZ(); }
  int GetZ() { return fValues % 10;
  ClassDef(Properties,2);
}
```

PropertiesLinkDef.h

```
#pragma read sourceClass="MyClass" version="[9]" targetClass="Properties"
#pragma read sourceClass="MyClass" version="[8]" targetClass="Properties" \
  source="int fX; int fY; int fZ" target="fValues" \
  code="{ fValues = onfile.fX*1000 + onfile.fY*10 + onfile.fZ; }"
```

To clarify its purpose the class needed to be renamed.

- ***sourceClass*** and ***targetClass*** are respectively ***MyClass*** and ***Properties***
- 1st rule indicates that version 9 of ***MyClass*** can be read directly into a ***Properties*** object using only the simple automatic schema evolution rules.
- 2nd rule indicates that in addition to the simple rules, a complex conversion needs to be applied when reading version 8 of ***MyClass*** into a ***Properties*** object.

Here comes cling

- Cling introduces binary compatible Just In Time compilation of script and code snippets.

- Will allows:
  - I/O for ‘interpreted’ classes
  - Runtime generation of CollectionProxy
    - Dictionary ***no longer*** needed for collections!
  - Run-time compilation of I/O Customization rules
    - including those carried in ROOT file.
  - Derivation of ‘interpreted’ class from compiled class
    - In particular ***TObject***
  - Faster, smarter TTreeFormula
  - Potential performance enhancement of I/O
    - Optimize hotspot by generating/compiling new code on demand
  - Interface simplification via the use complex C++ features
    - New, simpler TTree interface (***TTreeReader***)

TTreeReader

- New experimental interfaces to simplify and consolidate simple use cases.

```
void tread_obj() {
   // Reading object branches:
   TFile* f = TFile::Open("tr.root");
   TTreeReader tr("T");
   TTreeReaderValuePtr< MyParticle > p(tr, "p");
   TTreeReaderArray<double> e(tr, "v.fPos.fY");
   while (tr.GetNextEntry()) {
     printf("Particle momentum: %g\n", p->GetP());
     if (!e.IsEmpty())
       printf("lead muon energy: %g\n", e.At(0));
   }
   delete f;
}
```

- Automatically turns on all relevant optimizations
  - ***TTreeCache***, Partial reading. Etc.

Thread Safety

- Cling enables support for sturdy multi-thread I/O
  - Cling has clear separation of database engine and execution engine allowing to lock them independently

- Currently multi-threaded I/O supported as long as
  - All the ***TClass*** and ***TStreamerInfo*** are (explicitly) created serially.
  - ***TFile*** and ***TTree*** objects are accessed by only one thread (or the user code is explicitly locking the access to them).

- Cling will allow to remove the first limitation.

Why One TTree per thread/schedule

- When reading TTree holds:
  - Static State:
    - List of branches, their types their data location on file.
  - Dynamic State:
    - Current entry number, ***TTreeCache*** buffer (per ***TTree***), User object ptr (one per (top level) branch), Decompressed basket (one per branch)
  - Separating both would decrease efficiency

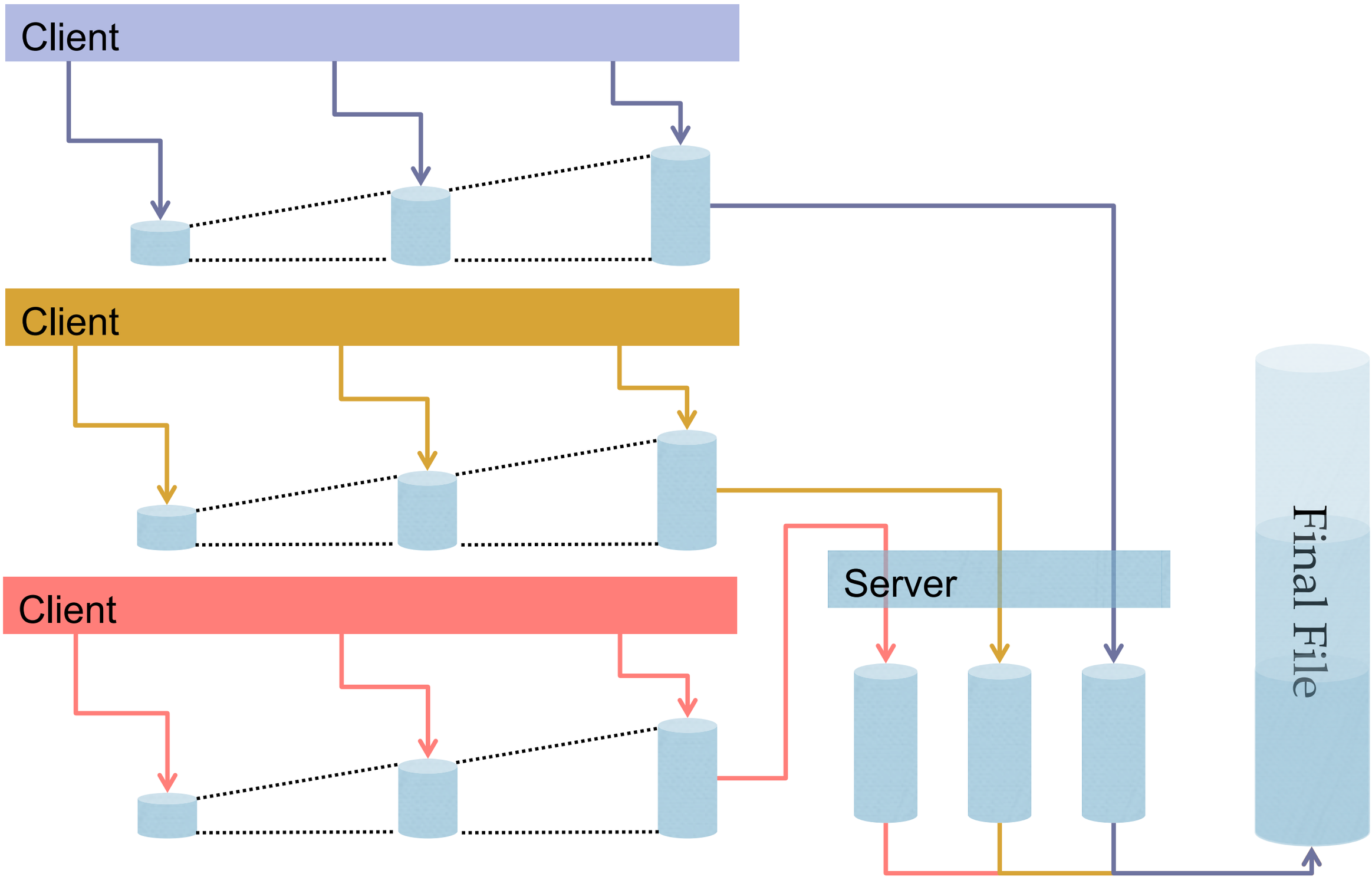
- Advantages
  - Works now!
  - No need for locks or synchronization
  - Decoupling of the access patterns
- Disadvantages
  - Duplication of some data and some buffers.
    - However this is usually small compare to the dynamic state.
  - Duplication of work if access overlap

Multi Processing Bottleneck

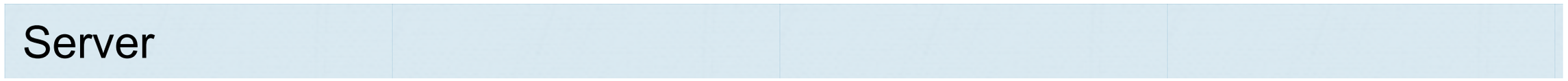
- Number of cores and nodes increasing dramatically
- Managing very large number of files is both hard and somewhat wasteful.
- Usual solution is to merge the files.

- In addition, the number of disks is not increasing as fast
  - Hidden serialization, for example when using whole node allocation and fork on write.

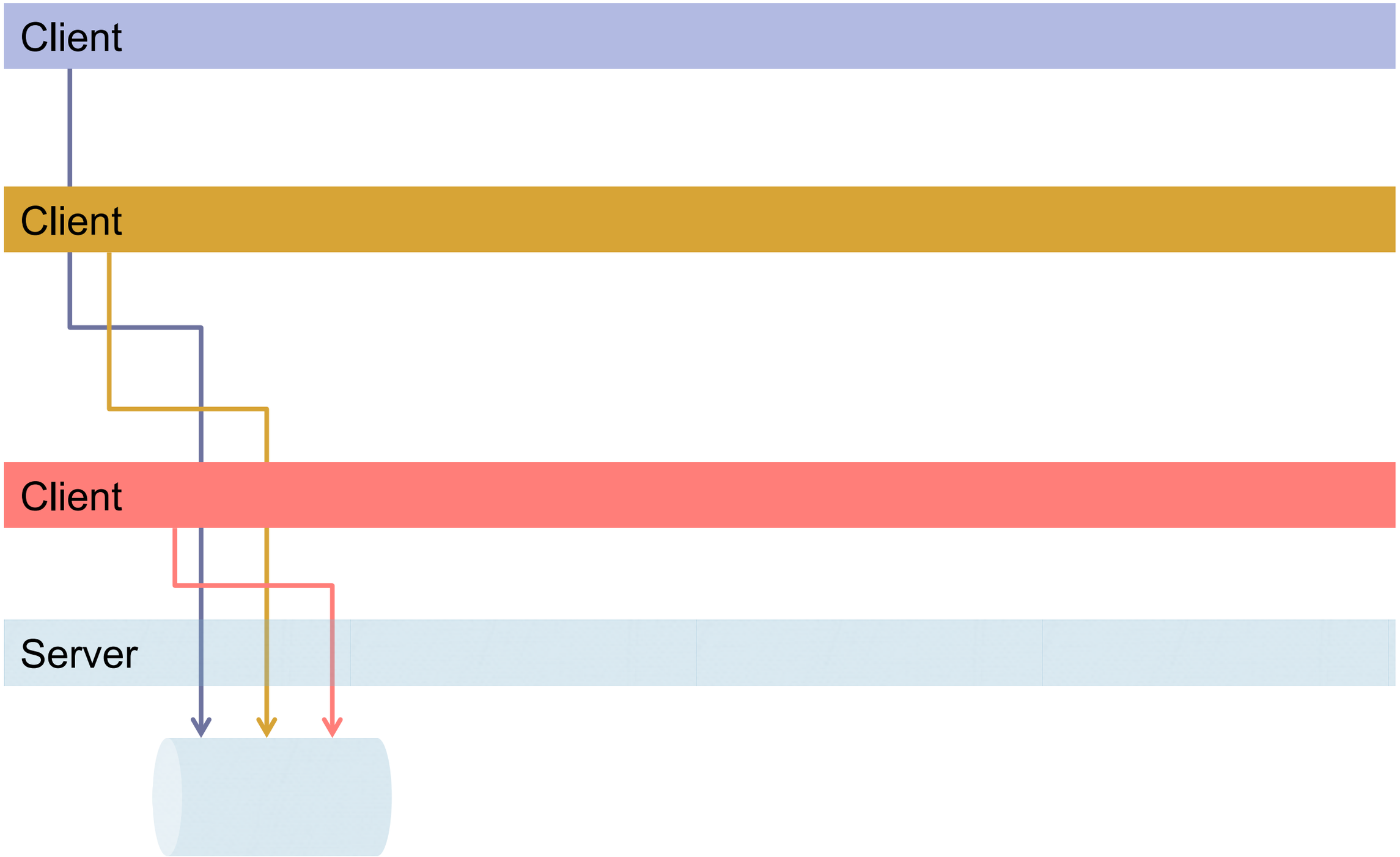
Typical Arrangement



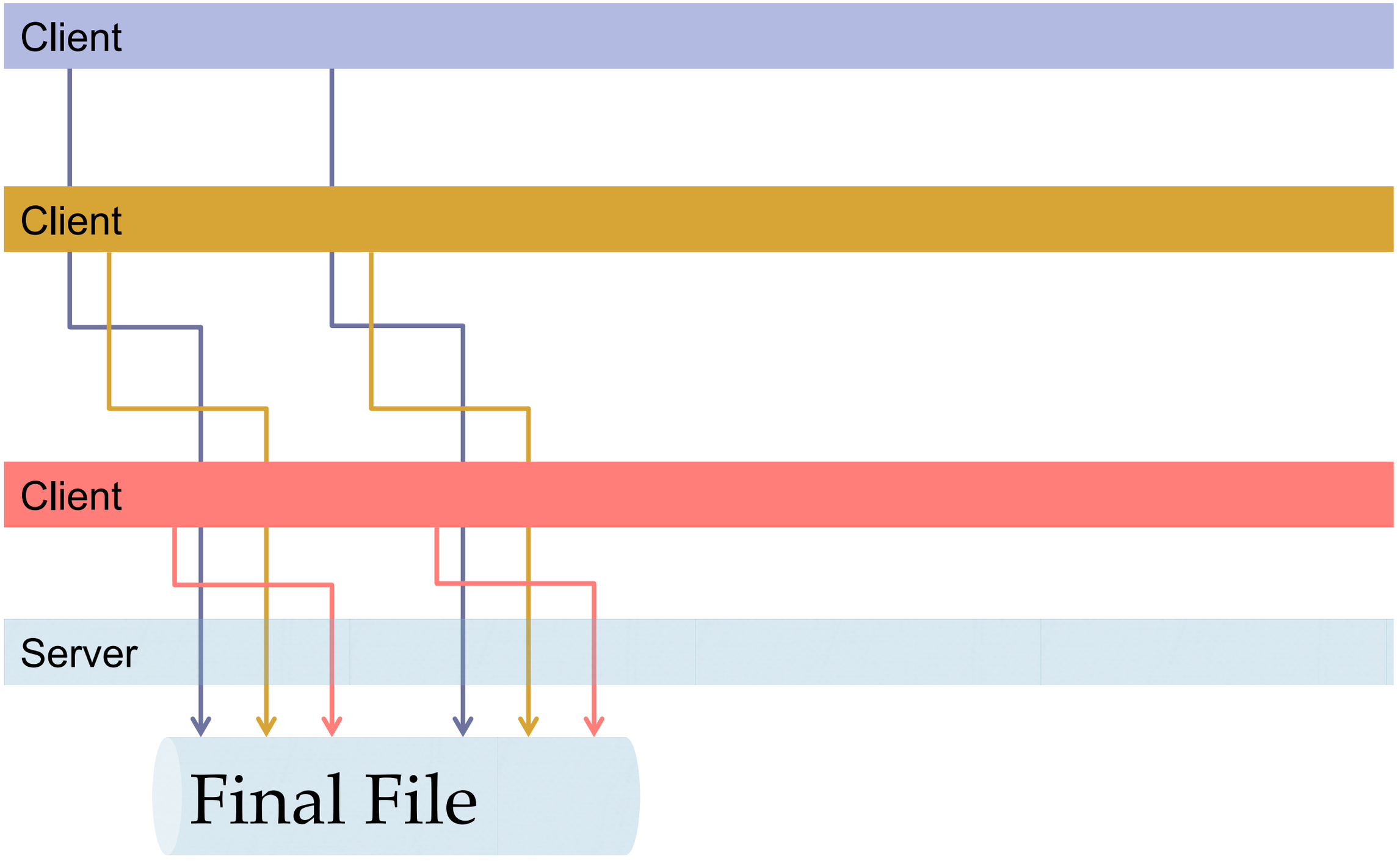
With Parallel Merging



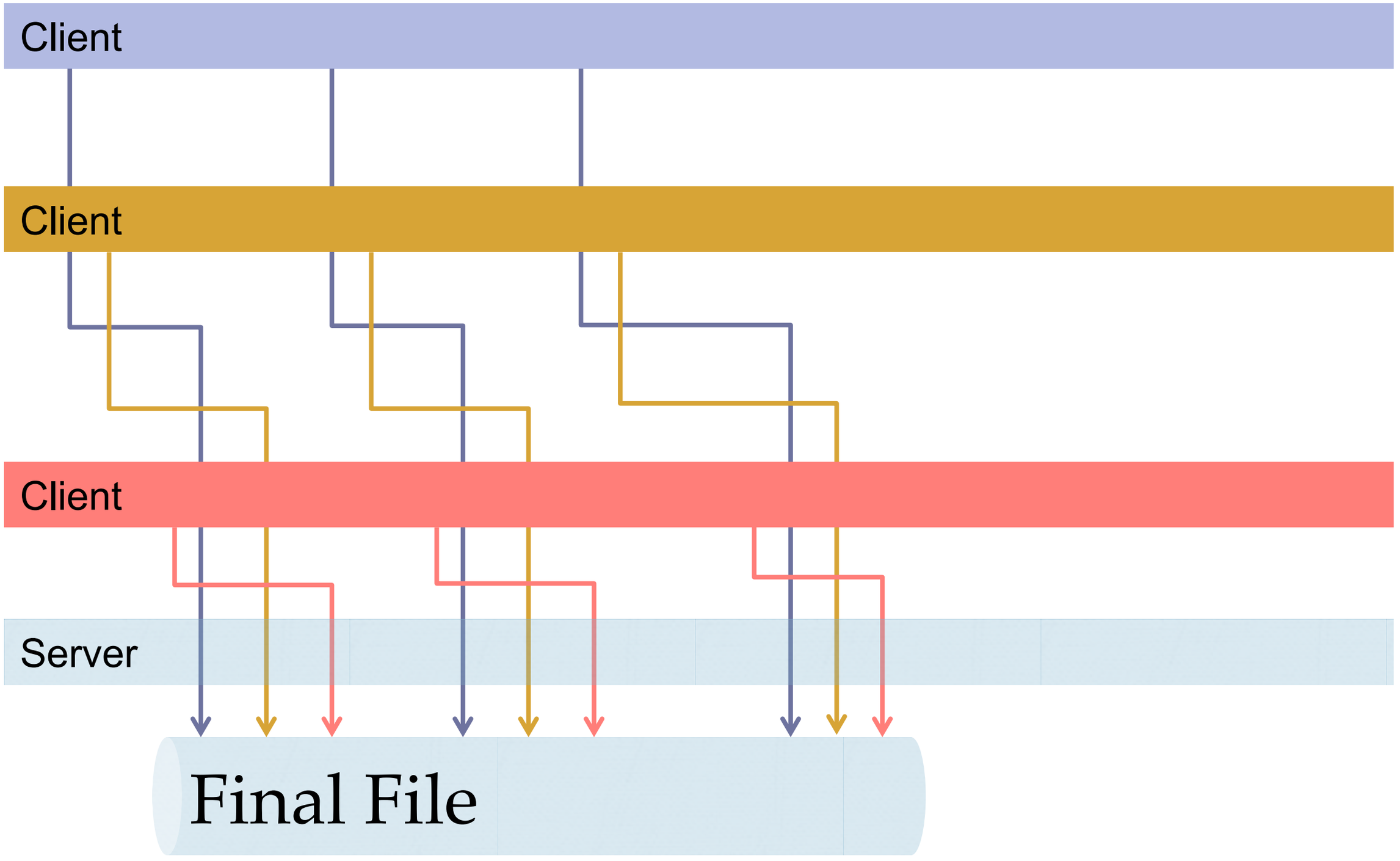
With Parallel Merging



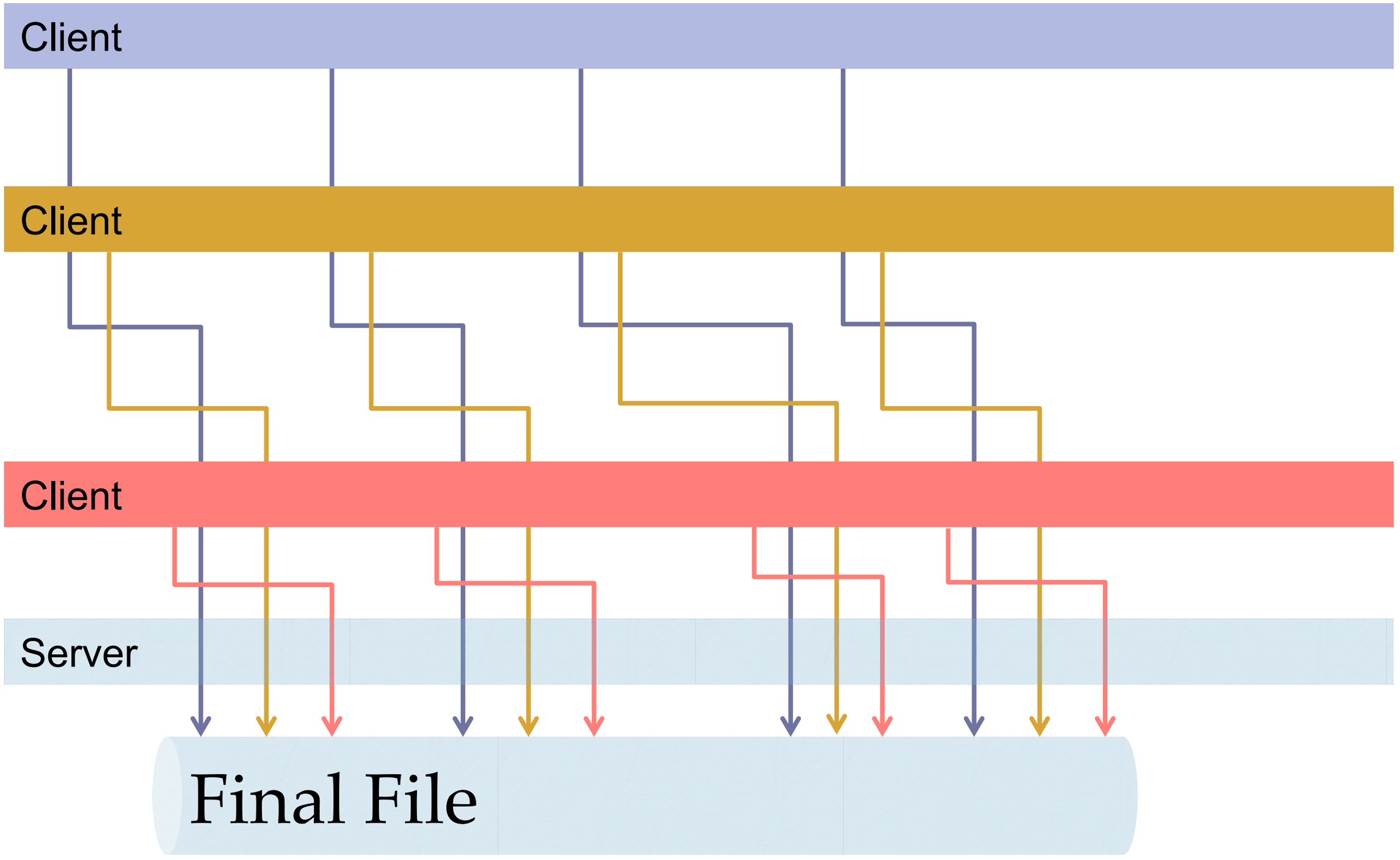
With Parallel Merging



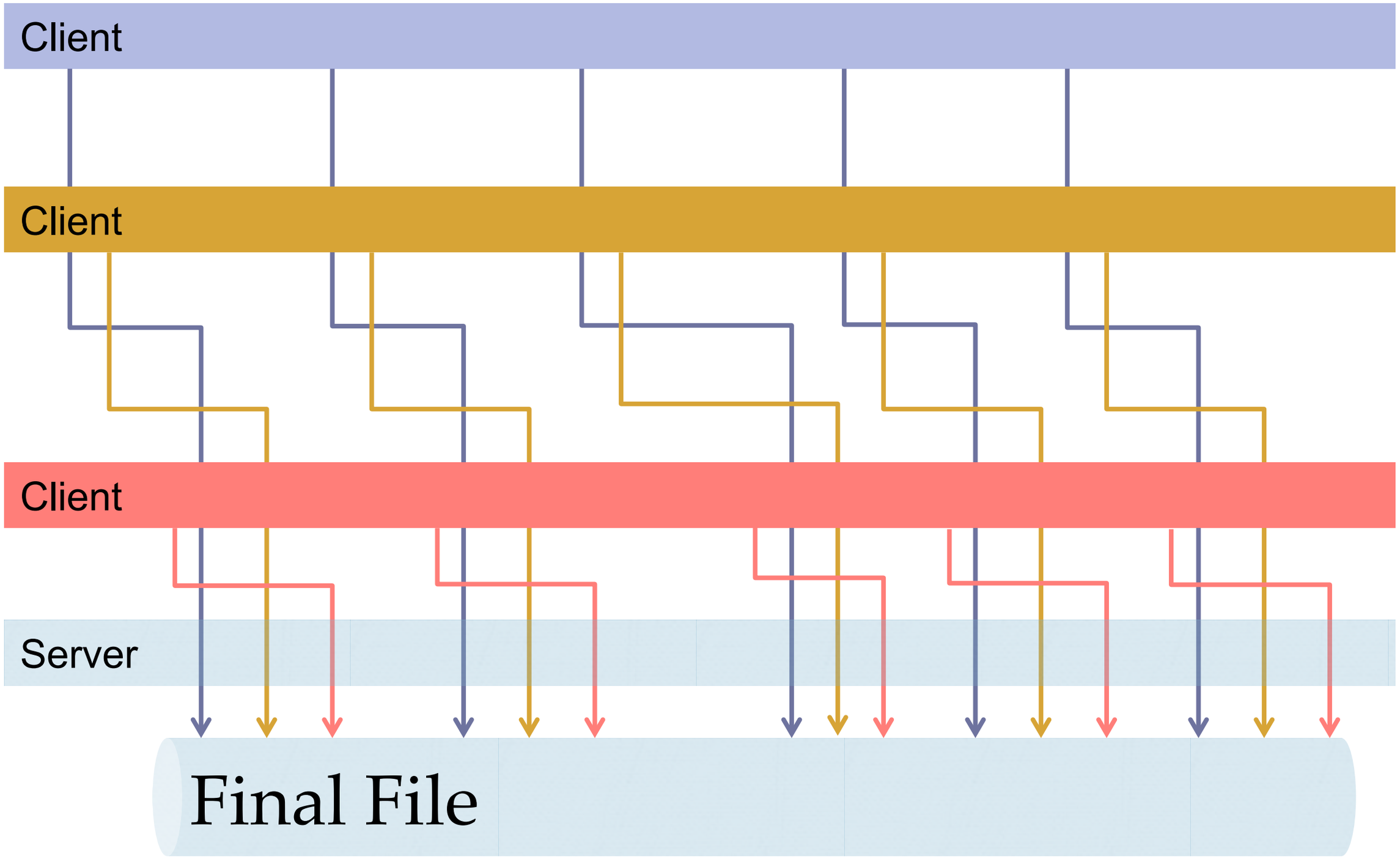
With Parallel Merging



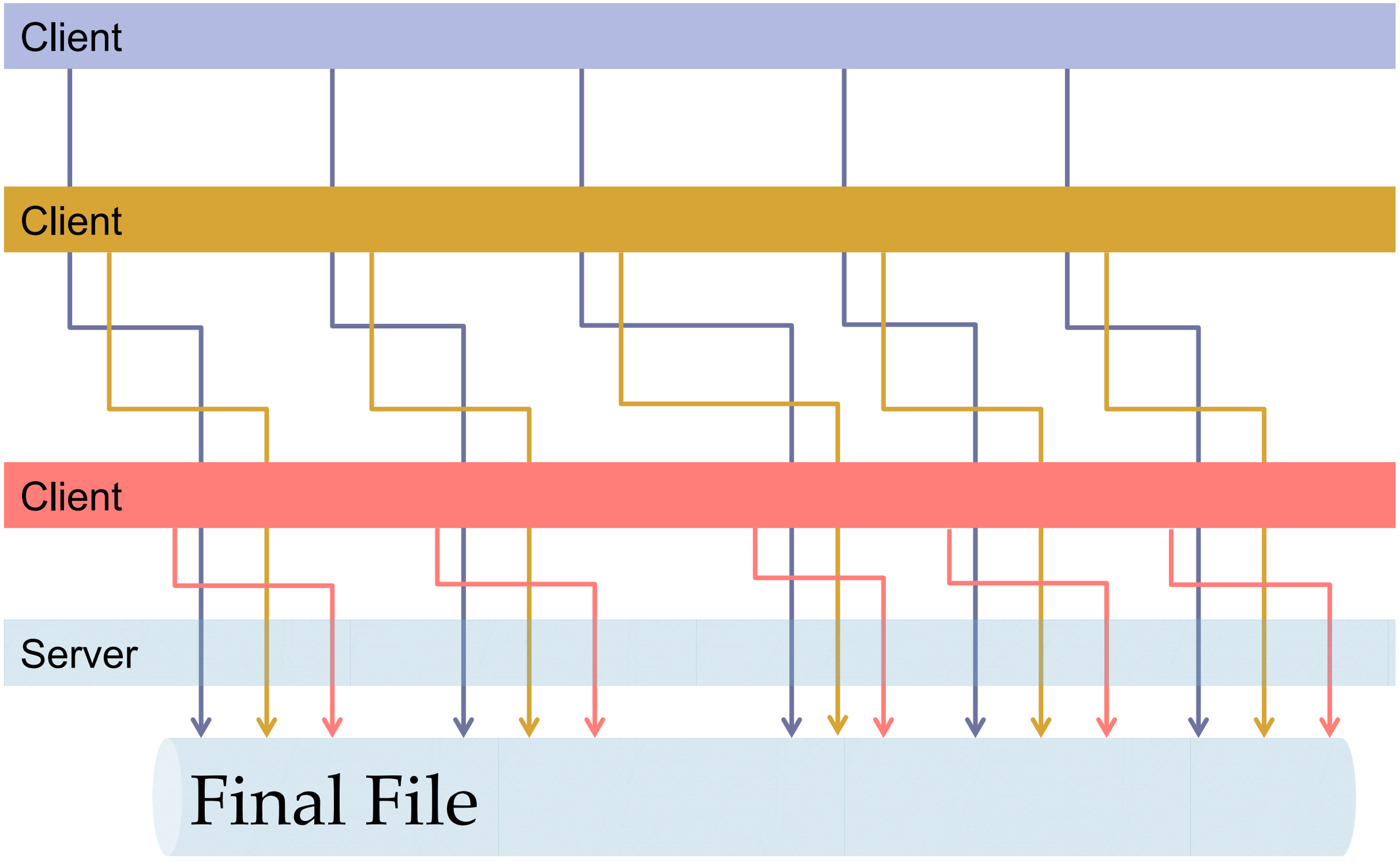
With Parallel Merging



With Parallel Merging



With Parallel Merging



Parallel Merge

- New class ***TMemFile***
  - A completely in memory version of ***TFile***.
- New class ***TParallelMergingFile***
  - A ***TMemFile*** that on a call to Write will
    - Upload its current content to a parallelMergerServer
    - Reset the ***TTree*** objects to facilitate the new merge.

```
TFile::Open("mergedClient.root?pmerge=localhost:1095","RECREATE");
```

- New daemon parallelMergeServer
  - Receive input from local or remote client and merges into requested file (which can be local or remote).
  - Fast merge ***TTree***. Re-merge all histogram at regular interval.

Parallel Merge Challenges

- Efficiently deal with many histograms
  - Each of them still need to be merged at the end

- Lack of ordering of the output of the workers
  - No enforcing of luminosity block boundaries for example
  - Introducing support for the ordering would lead to increased the interdependency between the worker and the server
  - Advance space reservation is challenging due to the variable size of the entries.

Conclusion

- Changes bring in new opportunities and challenges
  - Cling
  - Many cores
  - Remote processing
  - Evolving experiments data and algorithms

- ROOT I/O continues to evolve to meet those challenges and leverages those opportunities
  - TTreeCache
  - Parallel Merges
  - Multi-thread support
  - I/O Customization rules.