ROOT I/O

Philippe Canal and Jakob Blomer for the ROOT Team

ROOT
Data Analysis Framework
https://root.cern

Resources

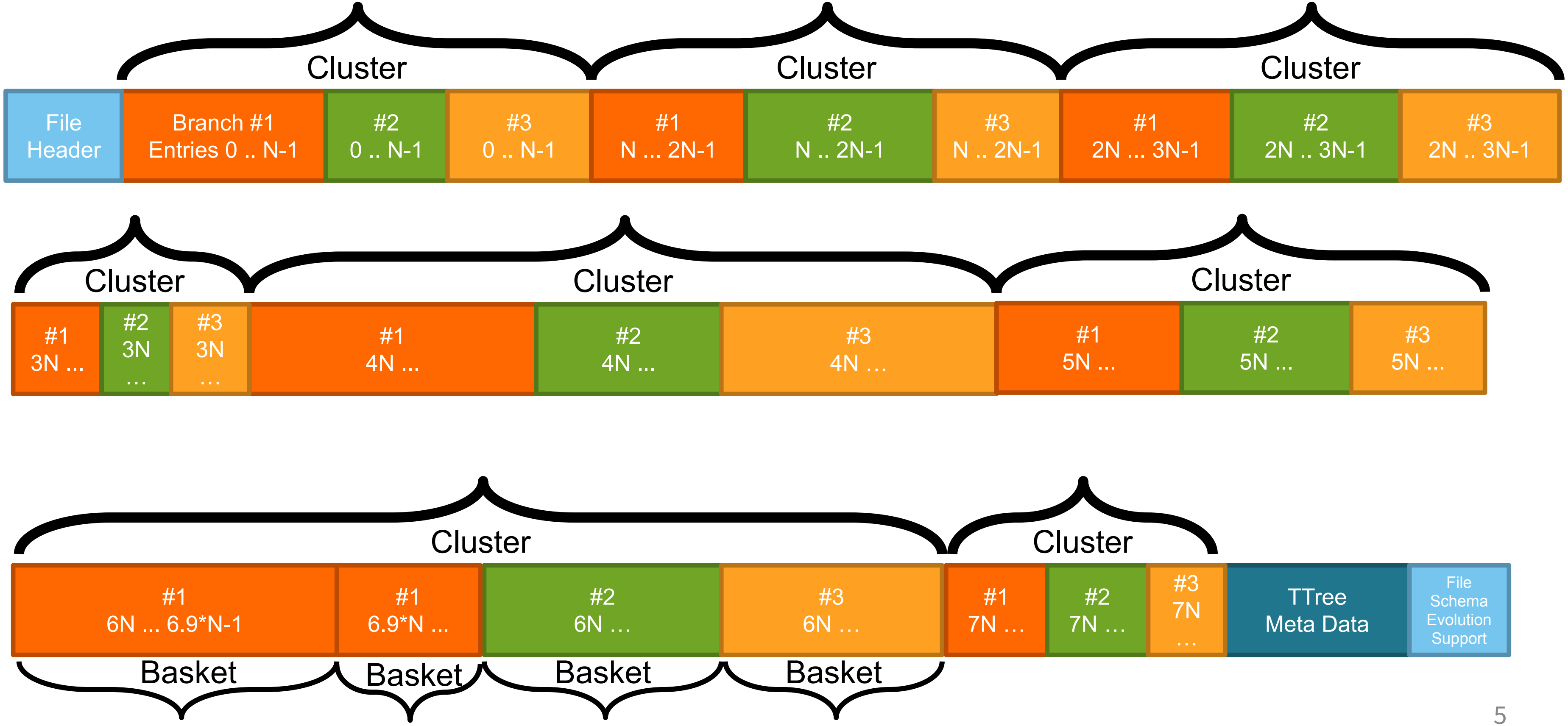
- ROOT Website: https://root.cern
- Introduction material: https://root.cern/getting-started
    - Includes a booklet for beginners: **the “ROOT Primer”**
- Reference Guide: https://root.cern/doc/master/index.html
- Training material: https://github.com/root-project/training
- Forum: https://root-forum.cern.ch

Parallelism

- Ongoing efforts to provide means for parallelisation in ROOT
- Explicit parallelism
  - **TThreadExecutor** and **TProcessExecutor**
  - Protection of resources
- Implicit parallelism
  - **RDataFrame**: Declarative Parallel analysis
  - TTreeProcessor: process tree events in parallel
  - TTree::GetEntry: process of tree branches in parallel
- Parallelism is a prerequisite element for tackling data analysis during LHC Run III and HL-LHC

The ROOT Columnar Format

Anatomy of a File



Cluster
Cluster
Cluster

File Header
Branch #1 Entries 0 .. N-1
#2 0 .. N-1
#3 0 .. N-1
#1 N ... 2N-1
#2 N .. 2N-1
#3 N .. 2N-1
#1 2N ... 3N-1
#2 2N .. 3N-1
#3 2N .. 3N-1

Cluster
Cluster
Cluster

#1 3N ...
#2 3N …
#3 3N …
#1 4N ...
#2 4N ...
#3 4N …
#1 5N ...
#2 5N ...
#3 5N ...

Cluster
Cluster

#1 6N ... 6.9*N-1
#1 6.9*N ...
#2 6N …
#3 6N …
#1 7N …
#2 7N …
#3 7N …
TTree Meta Data
File Schema Evolution Support

Basket
Basket
Basket
Basket

Many readers?

TFile, TTree and parallelism

- One Thread, One File (and it's TTrees)
  - Most flexible at the cost of memory
- Operations that ***can*** be run in parallel for a given TTree.
  - Prefetching of the raw bytes (thread)
    - Often this means that the “raw I/O” latency can be completely eliminated
  - Unzipping of baskets (task based - TBB for now)
    - Not enough feedback to evaluate how much it (can) help
  - Processing of branches’ content (task based - TBB for now)
    - Compression/Decompression and Streaming/Unstreaming done in parallel
    - Call to I/O system calls are ‘serialized’ (but likely // by the OS to some extent)

CMSSW Writting Bottleneck

- Output module fill each branch ‘independently’ but
  - Does not (yet?) turn on the feature to allow this operation to run concurrently.
  - In addition even with this feature, to preserve the onfile layout (contiguity of the cluster), there is a barrier to be respected at each cluster boundaries
    - The file is still fine/readable just not optimal without this.
  - So currently, lock/serialize all writes

CMSSW Reading Bottleneck

- Prefetch cache is part of each TTree’s and TFile’s state
- CMSSW needs 2 prefetching caches
    - One with few branches, One will all branches
    - Requires explicit synchronization.

CMSSW Reading Bottleneck

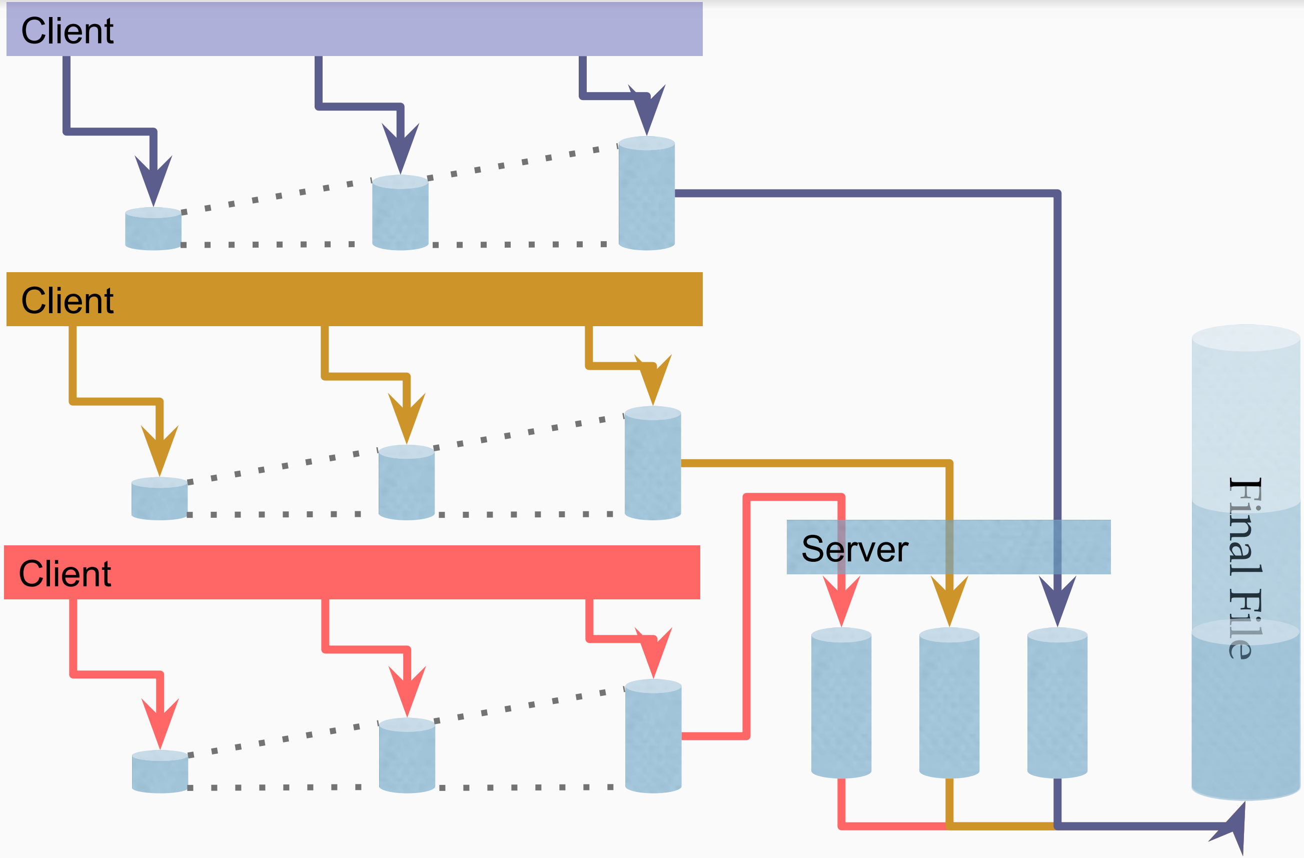
- Output module fill each branch ‘independently’ but
  - Does not (yet?) turn on the feature to allow this operation to run concurrently.
  - Even with that, one need to explicit stay within the confine of the content of the TTreeCache (to avoid cache thrashing)
  - So currently, lock/serialize all (most) reads.

CMSSW requests

- Interface to call TTree::GetEntry (or similar) with a set of branches
    - Would benefit from parallelism without changing the TTree states
- An interface to query a TTreeCache to see if a given branch+entry# is currently in the cache.
- Thread-safe interface to get an entry and use a given cache
- Thread-safe asynchronous interface for branch decompression
    - An asynchronous decompression of the clusters in the cache exist but might not be the right interface for CMSSW

Many writers?

Old Fashion Arrangement

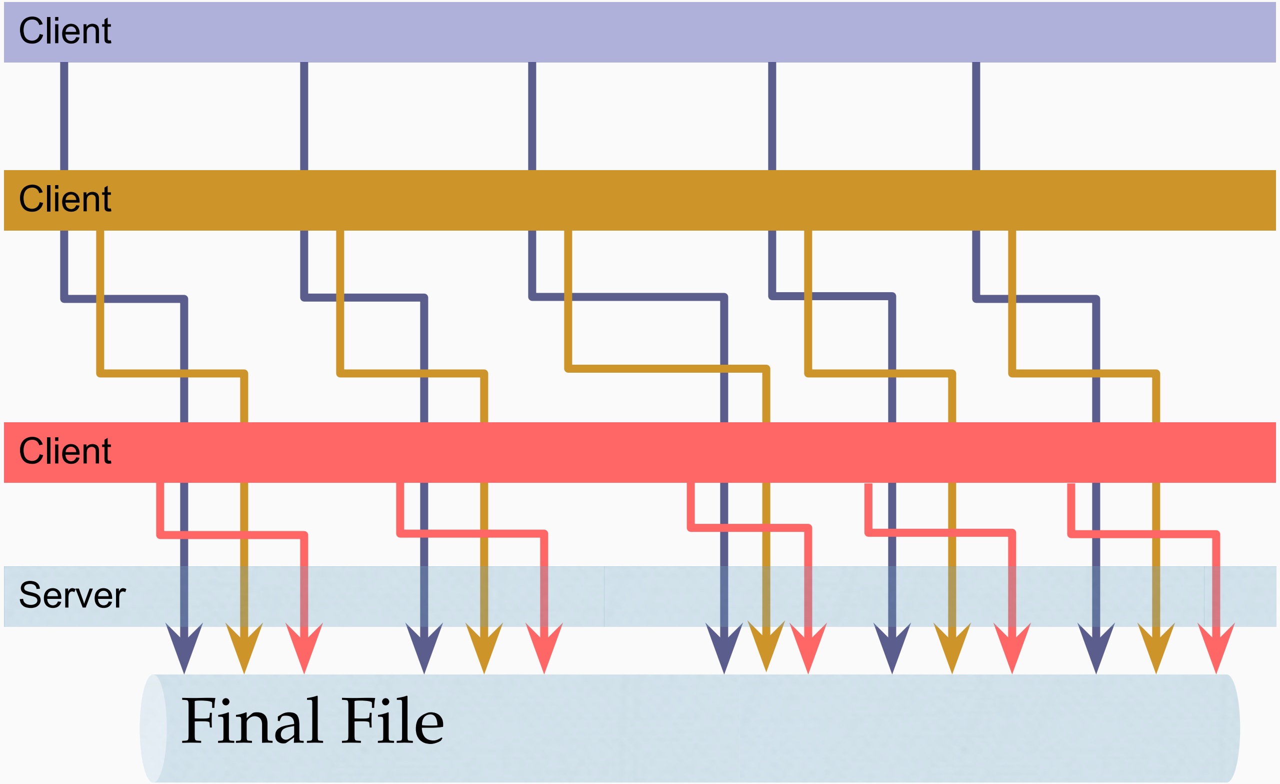


Fast Merging

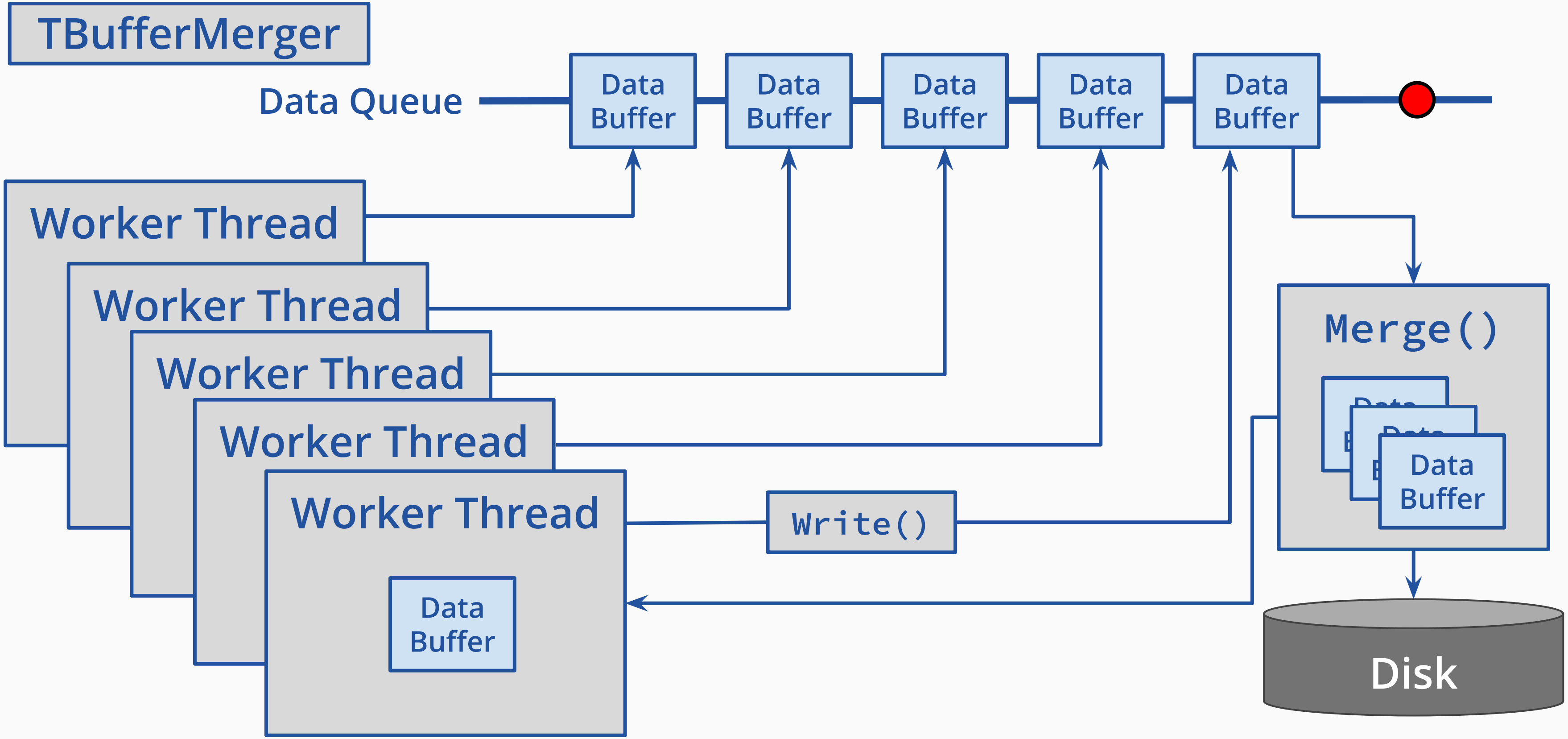
- ROOT Files can be ‘fast’ merged by ‘only’
  - Copying/appending the compressed data (baskets)
  - Updating the meta data (TTree object)
  - In first approximation we reach disk bandwith
    - Actually … half … since we read then write.

- Leverage this capability and use in-memory file to add support for multiple writers to the same file
  - Multi-thread in production
  - MPI prototype

With Parallel Merging



TBufferMerger

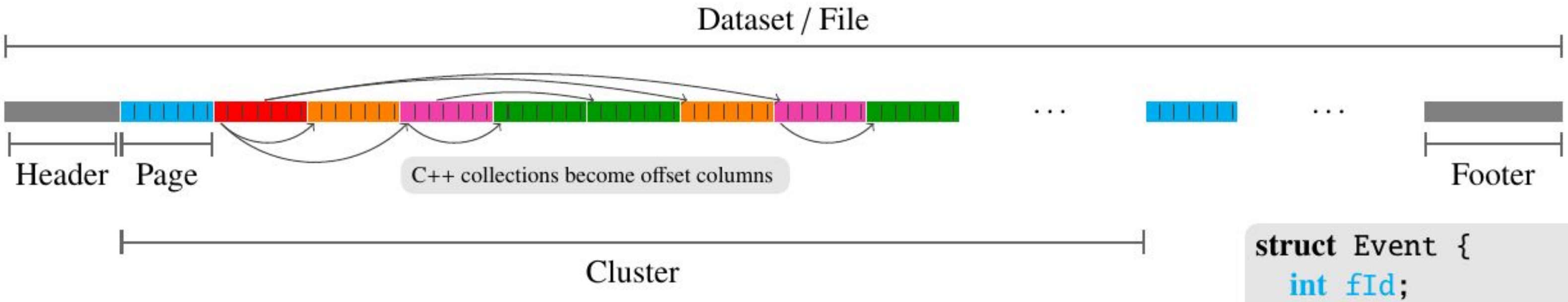


Additional Note

- TFile WriteCache
  - Allow delaying and coalescing the write at the cost of more memory
  - Not often used as gain is minimal on a single disk and memory often tight
- FastMerge mechanism can
  - Collect and reorganize how the baskets are layout on the file
- And could
  - Delay, coalesce or even distribute the actual writing

RNTuple: Evolution of the TTree I/O

RNTuple Format Breakdown



Approximate translation between TTree and RNTuple concepts:

Basket ≈ Page
Leaf ≈ Column
Cluster ≈ Cluster

```
struct Event {
  int fId;
  vector<Particle> fPtcls;
};
struct Particle {
  float fE;
  vector<int> fIds;
};
```

Cluster:

- Block of consecutive complete events
- Unit of thread parallelization (read & write)
- Typically tens of megabytes

Page:

- Unit of memory mapping or (de)compression
- Typically tens of kilobytes
- Naturally representable by an object, e.g. in the DAOS object store (under investigation)

RNTuple Concurrency Considerations

Current Status

- RNTuple is thread-friendly: no global state
- RNTuple reader and writer objects need to be used serialized, but can be used from multiple threads

- Support for multiple concurrent RNTuple readers on same file

- Asynchronous data preloading by default, 1 I/O thread per RNTuple reader (PR)

- Vectorization: through templates and inlining compiler sees the uncompressed page buffers (little endian), which is a precondition for vectorizing loops (to be confirmed)

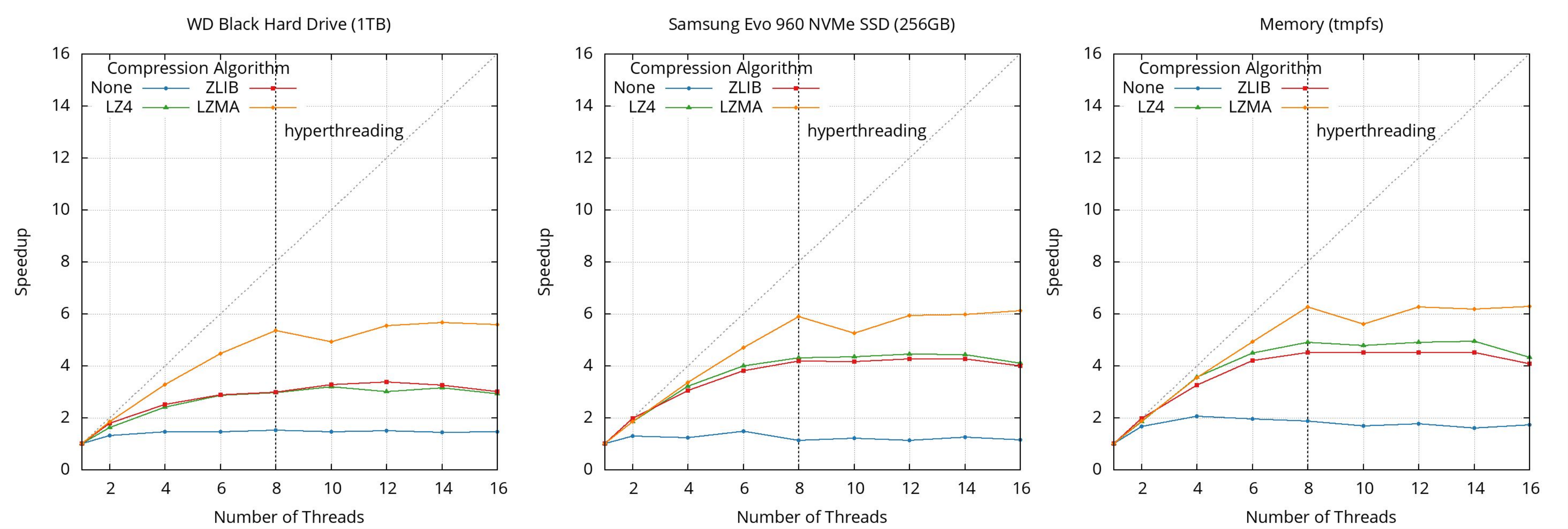
Ongoing and planned development

- Parallel page decompression offloaded to (experiment) task scheduler (TBB)
- MT access to single RNTuple reader provided that set of active clusters stays fixed
- Parallel writing: one cluster per thread, cluster data structure allows for append-only merging

Backup slides

Multi Branch Benchmark: Speedup

Test creates 10 branches, each with a vector of 10 Event



All figures using ROOT master branch

RDataFrame Basics

Can we do Better?

simple yet powerful way to analyse data with modern C++

---

provide high-level features, e.g.
less typing, better expressivity, abstraction of complex operations

---

allow transparent optimisations, e.g.
multi-thread parallelisation and caching

Improved Interfaces

what we write

```
TTreeReader reader(data);
TTreeReaderValue<A> x(reader,"x");
TTreeReaderValue<B> y(reader,"y");
TTreeReaderValue<C> z(reader,"z");
while (reader.Next()) {
   if (IsGoodEntry(*x, *y, *z))
      h->Fill(*x);
}
```

what we *mean*

- full control over the event loop
- requires some boilerplate
- users implement common tasks again and again
- parallelisation is not trivial

RDataFrame: declarative analyses

```
RDataFrame d(data);
auto h = d.Filter(IsGoodEntry, {"x","y","z"})
          .Histo1D("x");
```

- full control over *the analysis*
- no boilerplate
- common tasks are already implemented

? parallelization is not trivial?

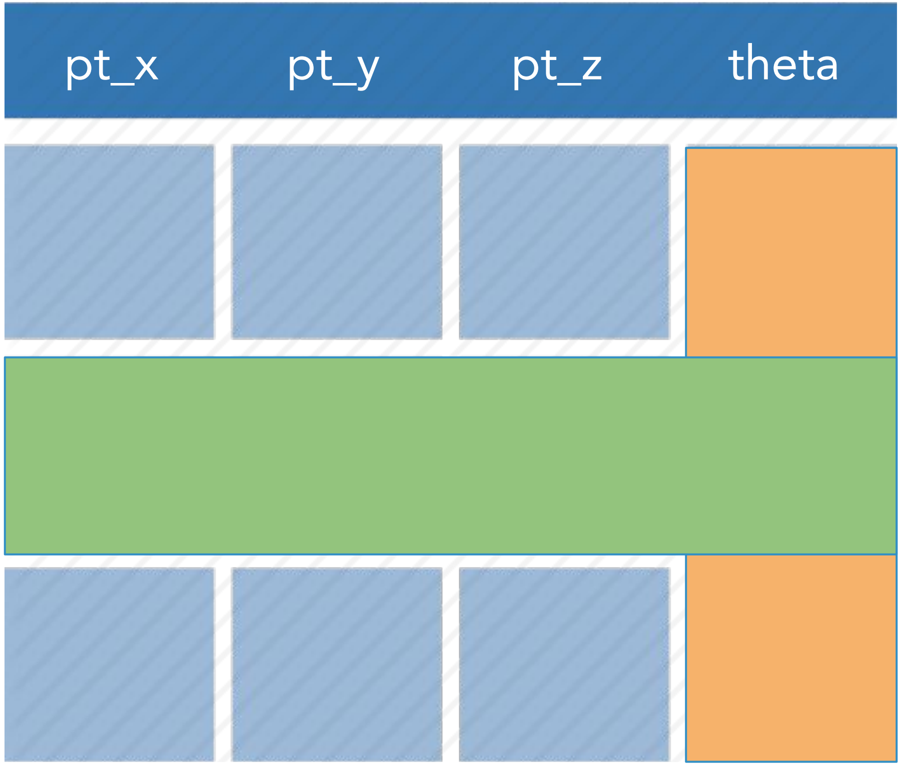
RDataFrame: declarative analyses

```
ROOT::EnableImplicitMT();
RDataFrame d(data);
auto h = d.Filter(IsGoodEntry, {"x","y","z"})
          .Histo1D("x");
```

- full control over *the analysis*
- no boilerplate
- common tasks are already implemented

? parallelization is not trivial?

Columnar Representation



entries
or events
or rows

columns
or “branches”

can contain any kind
of c++ object

RDataFrame: quick how-to

1. build a data-frame object by specifying your data-set
2. apply a series of transformations to your data
   - filter (e.g. apply some cuts) or
   - define new columns
3. apply actions to the transformed data to produce results (e.g. fill a histogram)

Creating a RDataFrame - 1 file

```
RDataFrame d1("treename", "file.root");

auto filePtr = TFile::Open("file.root");
RDataFrame d2("treename", filePtr);

TTree *treePtr = nullptr;
filePtr->GetObject("treename", treePtr);
RDataFrame d3(*treePtr); // by reference!
```

Three ways to create a RDataFrame that reads tree "treename" from file "file.root"

Creating a RDataFrame - more files

```
RDataFrame d1("treename", "file*.root");
RDataFrame d2("treename", {"file1.root","file2.root"});

std::vector<std::string> files = {"file1.root","file2.root"};
RDataFrame d3("treename", files);

TChain chain("treename");
chain.Add("file1.root); chain.Add("file2.root);
RDataFrame d4(chain); // passed by reference, not pointer!
```

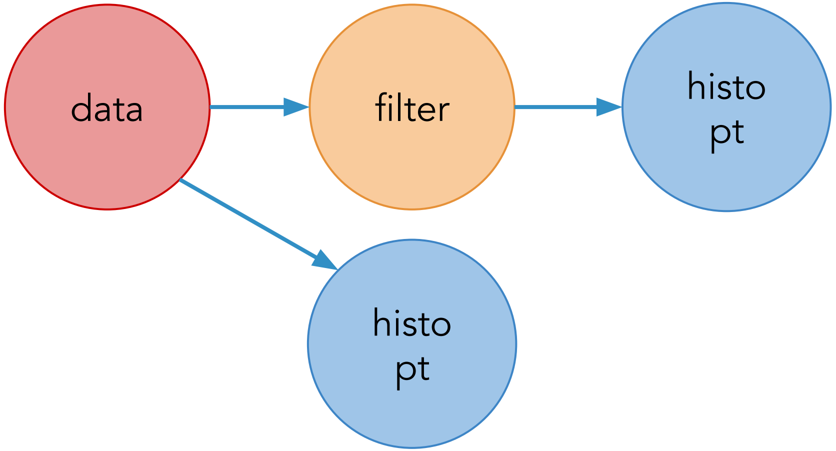
Here RDataFrame reads tree “treename” from files “file1.root” and “file2.root”

Cut on theta, fill histogram with pt

```
RDataFrame d("t", "f.root");
auto h = d.Filter("theta > 0").Histo1D("pt");
h->Draw(); // event loop is run here, when you access a result
          // for the first time
```

event-loop is run *lazily*, upon first access to the results

Think of your analysis as data-flow



```
auto h2 = d.Filter("theta > 0").Histo1D("pt");
auto h1 = d.Histo1D("pt");
```

Using callables instead of strings

```
// define a c++11 lambda - an inline function - that checks “x>0”
auto IsPos = [](double x) { return x > 0.; };
// pass it to the filter together with a list of branch names
auto h = d.Filter(IsPos, {"theta"}).Histo1D("pt");
h->Draw();
```

any callable (function, lambda, functor class) can be used as a filter, as long as it returns a boolean

Filling multiple histograms

```
auto h1 = d.Filter("theta > 0").Histo1D("pt");
auto h2 = d.Filter("theta < 0").Histo1D("pt");
h1->Draw();         // event loop is run once here
h2->Draw("SAME"); // no need to run loop again here
```

Book all your actions upfront. The first time a result is accessed, RDataFrame will fill all booked results.

Define a new column

```
double m = d.Filter("x > y")
            .Define("z", "sqrt(x*x + y*y)")
            .Mean("z");
```

`Define` takes the name of the new column and its expression. Later you can use the new column as if it was present in your data.

Define a new column

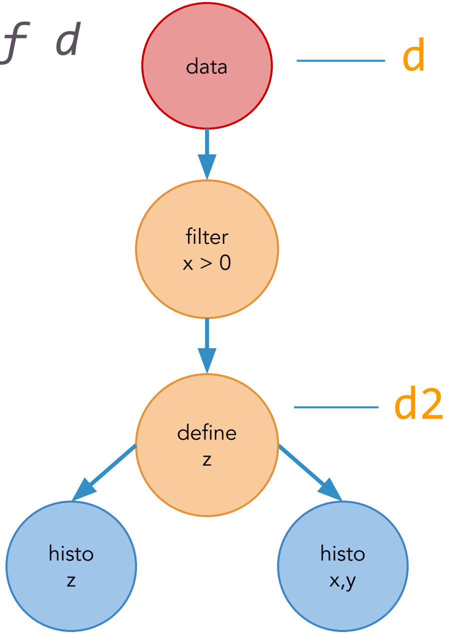
```
double SqrtSumSq(double, double) { return … ; }
double m = d.Filter("x > y")
            .Define("z", SqrtSumSq, {"x","y"})
            .Mean("z");
```

Just like `Filter`, `Define` accepts any callable object (function, lambda, functor class…)

Think of your analysis as data-flow

```
// d2 is a new data-frame, a transformed version of d
auto d2 = d.Filter("x > 0")
           .Define("z", "x*x + y*y");

// make multiple histograms out of it
auto hz = d2.Histo1D("z");
auto hxy = d2.Histo2D("x","y");
```



You can store transformed data-frames in variables, then use them as you would use a RDataFrame.

Cutflow reports

```
d.Filter("x > 0", "xcut")
 .Filter("y < 2", "ycut");
d.Report();
```

```
// output
xcut        : pass=49          all=100         --   49.000 %
ycut        : pass=22          all=49          --   44.898 %
```

When called on the main TDF object, `Report` prints statistics for all filters *with a name*

Running on a range of entries #1

```
// stop after 100 entries have been processed
auto hz = d.Range(100).Histo1D("x");

// skip the first 10 entries, then process one every two until the end
auto hz = d.Range(10, 0, 2).Histo1D("x");
```

Ranges are only available in single-thread executions.
They are useful for quick initial data explorations.

Running on a range of entries #2

```
// ranges can be concatenated with other transformations
auto c = d.Filter("x > 0")
          .Range(100)
          .Count();
```

This \`Range\` will process the first 100 entries
that pass the filter

Saving data to file

```
auto new_df = df.Filter("x > 0")
                .Define("z", "sqrt(x*x + y*y)")
                .Snapshot("tree", "newfile.root");
```

We filter the data, add a new column, and then save everything to file. No boilerplate code at all.

Creating a new data-set

```
RDataFrame d(100);
auto new_d = d.Define("x", []() { return double(rand()) / RAND_MAX; })
               .Define("y", []() { return rand() % 10; })
               .Snapshot("tree", "newfile.root");
```

We create a special TDF with 100 (empty) entries, define some columns, save it to file

N.B. `rand()` is generally not a good way to produce uniformly distributed random numbers

Not Only ROOT Datasets

- TDataSource: Plug *any columnar* format in RDataFrame
- Keep the programming model identical!
- ROOT provides CSV data source
- More to come
  - TDataSource is a programmable interface!
  - E.g. https://github.com/bluehood/mdfds LHCb raw format - not in the ROOT repo

Not Only ROOT Datasets

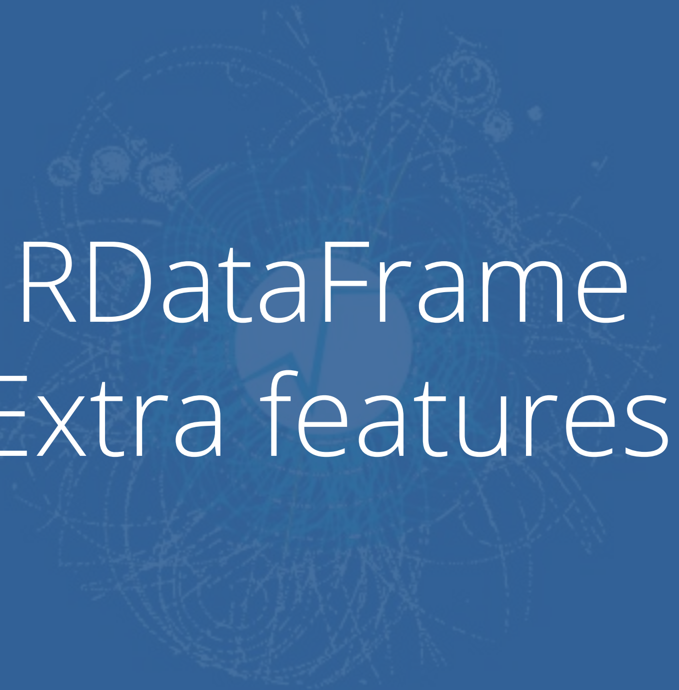
```
auto fileName = "tdf014_CsvDataSource_MuRun2010B.csv";
auto tdf = ROOT::Experimental::TDF::MakeCsvDataFrame(fileName);

auto filteredEvents =
tdf.Filter("Q1 * Q2 == -1")
.Define("m", "sqrt(pow(E1 + E2, 2) - (pow(px1 + px2, 2) + pow(py1 + py2, 2) + pow(pz1 + pz2, 2)))");

auto invMass =
filteredEvents.Histo1D({"invMass", "CMS Opendata: #mu#mu mass;mass [GeV];Events", 512, 2, 110}, "m");
```

tdf014_CsvDataSource_MuRun2010B.csv:

```
Run,Event,Type1,E1,px1,py1,pz1,pt1,eta1,phi1,Q1,Type2,E2,px2,py2,pz2,pt2,eta2,phi2,Q2,M
146436,90830792,G,19.1712,3.81713,9.04323,-16.4673,9.81583,-1.28942,1.17139,1,T,5.43984,-0.362592,2.62699,-
4.74849,2.65189,-1.34587,1.70796,1,2.73205
146436,90862225,G,12.9435,5.12579,-3.98369,-11.1973,6.4918,-1.31335,-0.660674,-1,G,11.8636,4.78984,-6.26222,
-8.86434,7.88403,-0.966622,-0.917841,1,3.10256
```



RDataFrame
Extra features

Caching

```
RDataFrame d("mytree", "myFile.root");
auto cached_d = d.Cache();
```

All the content of the TDF is now in (contiguous) memory.
Analysis as fast as it can be (vectorisation possible too).

N.B. It is always possible to selectively cache columns to save some memory!

Creating a new data-set - parallel

```
ROOT::EnableImplicitMT();
RDataFrame d(100);
auto new_d = d.Define("x", []() { return double(rand()) / RAND_MAX; })
               .Define("y", []() { return rand() % 10; })
               .Snapshot("tree", "newfile.root");
```

We create a special TDF with 100 (empty) entries, define some columns, save it to file -- in parallel

N.B. `rand()` is generally not a good way to produce uniformly distributed random numbers

More on histograms #1

```
auto h = d.Histo1D("x","w");
```

TDF can produce *weighted* TH1D, TH2D and TH3D.
Just pass the extra column name.

More on histograms #2

```
auto h = d.Histo1D({"h","h",10,0.,1.},"x", "w");
```

You can specify a model histogram with a set axis range, a name and a title (optional for TH1D, mandatory for TH2D and TH3D)

Filling histograms with arrays

```
auto h = d.Histo1D("pt_array", "x_array");
```

If \`pt_array\` and \`x_array\` are an array or an STL container (e.g. std::vector), TDF fills histograms with all of their elements. \`pt_array\` and \`x_array\` are required to have equal size for each event.