How to make a petabyte ROOT file: proposal for managing data with columnar granularity

Jim Pivarski

Princeton University – DIANA

October 11, 2017

Motivation: start by stating the obvious

ROOT's selective reading is very important for analysis.

Datasets have about a thousand branches[1], so if you want to plot a quantity from a terabyte dataset with `TTree::Draw`, you only have to read a few gigabytes from disk.

[1] 3116 ATLAS MC, 1717 ATLAS data, 2151 CMS MiniAOD, 675+ CMS NanoAOD, 560 LHCb

Motivation: start by stating the obvious

ROOT's selective reading is very important for analysis.

Datasets have about a thousand branches[1], so if you want to plot a quantity from a terabyte dataset with `TTree::Draw`, you only have to read a few gigabytes from disk.

Same for reading over a network (XRootD).

```
auto file = TFile::Open("root://very.far.away/mydata.root");
```

[1] 3116 ATLAS MC, 1717 ATLAS data, 2151 CMS MiniAOD, 675+ CMS NanoAOD, 560 LHCb

Motivation: start by stating the obvious

ROOT's selective reading is very important for analysis.

Datasets have about a thousand branches[1], so if you want to plot a quantity from a terabyte dataset with `TTree::Draw`, you only have to read a few gigabytes from disk.

Same for reading over a network (XRootD).

```
auto file = TFile::Open("root://very.far.away/mydata.root");
```

This is GREAT.

[1] 3116 ATLAS MC, 1717 ATLAS data, 2151 CMS MiniAOD, 675+ CMS NanoAOD, 560 LHCb

Conversation with computer scientist

So it sounds like you already have a columnar database.

Conversation with computer scientist

So it sounds like you already have a columnar database.

Not exactly— we still have to manage data as files, rather than columns.

Conversation with computer scientist

So it sounds like you already have a columnar database.

Not exactly— we still have to manage data as files, rather than columns.

What? Why? Couldn't you just use XRootD to manage (move, backup, cache) columns directly? Why does it matter that they're inside of files?

Conversation with computer scientist

So it sounds like you already have a columnar database.

Not exactly— we still have to manage data as files, rather than columns.

What? Why? Couldn't you just use XRootD to manage (move, backup, cache) columns directly? Why does it matter that they're inside of files?

Because... because...

Evidence that it matters: the CMS NanoAOD project

Stated goal: to serve 30–50% of CMS analyses with a single selection of columns.

Need to make hard decisions about which columns to keep: reducing more makes data access easier for 50% of analyses while completely excluding the rest.

Evidence that it matters: the CMS NanoAOD project

Stated goal: to serve 30–50% of CMS analyses with a single selection of columns.

Need to make hard decisions about which columns to keep: reducing more makes data access easier for 50% of analyses while completely excluding the rest.

If we really had columnar data management, the problem would be moot: we'd just let the most frequently used 1–2 kB *of each event* migrate to warm storage while the rest cools.

Evidence that it matters: the CMS NanoAOD project

Stated goal: to serve 30–50% of CMS analyses with a single selection of columns.

Need to make hard decisions about which columns to keep: reducing more makes data access easier for 50% of analyses while completely excluding the rest.

If we really had columnar data management, the problem would be moot: we'd just let the most frequently used 1–2 kB *of each event* migrate to warm storage while the rest cools.

Instead, we'll probably put the whole small copy (NanoAOD) in warm storage and the whole large copy (MiniAOD) in colder storage.

Evidence that it matters: the CMS NanoAOD project

diana hep

Stated goal: to serve 30–50% of CMS analyses with a single selection of columns.

Need to make hard decisions about which columns to keep: reducing more makes data access easier for 50% of analyses while completely excluding the rest.

If we really had columnar data management, the problem would be moot: we'd just let the most frequently used 1–2 kB *of each event* migrate to warm storage while the rest cools.

Instead, we'll probably put the whole small copy (NanoAOD) in warm storage and the whole large copy (MiniAOD) in colder storage.

This is artificial.

Evidence that it matters: the CMS NanoAOD project

diana hep

Stated goal: to serve 30–50% of CMS analyses with a single selection of columns.

Need to make hard decisions about which columns to keep: reducing more makes data access easier for 50% of analyses while completely excluding the rest.

If we really had columnar data management, the problem would be moot: we'd just let the most frequently used 1–2 kB *of each event* migrate to warm storage while the rest cools.

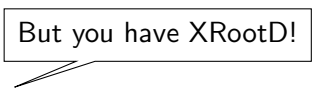
Instead, we'll probably put the whole small copy (NanoAOD) in warm storage and the whole large copy (MiniAOD) in colder storage.

This is artificial.

There's a steep popularity distribution across columns, but we cut it abruptly with file schemas (data tiers).

Except for the simplest TTree structures, we can't pull individual branches out of a file and manage them on their own.

Except for the simplest TTree structures, we can't pull individual branches out of a file and manage them on their own.



Except for the simplest TTree structures, we can't pull individual branches out of a file and manage them on their own.

But you have XRootD!

Yes, but only ROOT knows how to interpret a branch's relationship with other branches.

What would it look like if we could?

```
CREATE TABLE derived_data AS
    SELECT pt, eta, phi, deltaphi**2 + deltaeta**2 AS deltaR
    FROM original_data WHERE deltaR < 0.2;
```

creates a new `derived_data` table from `original_data`, but *links*, rather than *copying*, `pt`, `eta`, and `phi`.[2]

[2] Implementation dependent, but common. "`WHERE`" selection may be implemented with a stencil.

```
CREATE TABLE derived_data AS
    SELECT pt, eta, phi, deltaphi**2 + deltaeta**2 AS deltaR
    FROM original_data WHERE deltaR < 0.2;
```

creates a new `derived_data` table from `original_data`, but *links*, rather than *copying*, `pt`, `eta`, and `phi`.[2]

If `original_data` is deleted, the database would not delete `pt`, `eta`, and `phi`, as they're in use by `derived_data`.

[2] Implementation dependent, but common. "`WHERE`" selection may be implemented with a stencil.

```
CREATE TABLE derived_data AS
    SELECT pt, eta, phi, deltaphi**2 + deltaeta**2 AS deltaR
    FROM original_data WHERE deltaR < 0.2;
```

creates a new `derived_data` table from `original_data`, but *links*, rather than *copying*, `pt`, `eta`, and `phi`.[2]

If `original_data` is deleted, the database would not delete `pt`, `eta`, and `phi`, as they're in use by `derived_data`.

For data management, this is a very flexible system, as columns are a more granular unit for caching and replication.

[2] Implementation dependent, but common. "`WHERE`" selection may be implemented with a stencil.

What would it look like if we could?

```
CREATE TABLE derived_data AS
    SELECT pt, eta, phi, deltaphi**2 + deltaeta**2 AS deltaR
    FROM original_data WHERE deltaR < 0.2;
```

creates a new `derived_data` table from `original_data`, but *links*, rather than *copying*, `pt`, `eta`, and `phi`.[2]

If `original_data` is deleted, the database would not delete `pt`, `eta`, and `phi`, as they're in use by `derived_data`.

For data management, this is a very flexible system, as columns are a more granular unit for caching and replication.

For users, there is much less cost to creating derived datasets— many versions of corrections and cuts.

[2] Implementation dependent, but common. "`WHERE`" selection may be implemented with a stencil.

Idea #1. Cast data from ROOT files into a well-known standard for columnar, hierarchical data; manage those columns individually in an object store like Ceph.

Idea #1. Cast data from ROOT files into a well-known standard for columnar, hierarchical data; manage those columns individually in an object store like Ceph.

1. Apache Arrow is one such standard. It's similar to ROOT's splitting format but permits $\mathcal{O}(1)$ random access and splits down to all levels of depth.

Idea #1. Cast data from ROOT files into a well-known standard for columnar, hierarchical data; manage those columns individually in an object store like Ceph.

1. Apache Arrow is one such standard. It's similar to ROOT's splitting format but permits $\mathcal{O}(1)$ random access and splits down to all levels of depth.
2. PLUR or PLURP is my subset of the above with looser rules about how data may be referenced. Acronym for the minimum data model needed for physics: **Primitives**, **Lists**, **Unions**, **Records**, and maybe **Pointers** (beyond Arrow).

Idea #1. Cast data from ROOT files into a well-known standard for columnar, hierarchical data; manage those columns individually in an object store like Ceph.

1. Apache Arrow is one such standard. It's similar to ROOT's splitting format but permits $\mathcal{O}(1)$ random access and splits down to all levels of depth.
2. PLUR or PLURP is my subset of the above with looser rules about how data may be referenced. Acronym for the minimum data model needed for physics: **Primitives**, **Lists**, **Unions**, **Records**, and maybe **Pointers** (beyond Arrow).

(the "standard database" approach)

Idea #2 (this talk). Keep ROOT data as they are, but put individual *TBaskets* in the object store. TFile/TTree subclasses fetch data from the object store instead of seeking to file positions.

Idea #2 (this talk). Keep ROOT data as they are, but put individual *TBaskets* in the object store. TFile/TTree subclasses fetch data from the object store instead of seeking to file positions.

1. Presents the same TFile/TTree interface to users; **old scripts still work.**

Idea #2 (this talk). Keep ROOT data as they are, but put individual *TBaskets* in the object store. TFile/TTree subclasses fetch data from the object store instead of seeking to file positions.

1. Presents the same TFile/TTree interface to users; **old scripts still work.**
2. But data replication, storage class, and caching are handled by the object store with columnar granularity.

Idea #2 (this talk). Keep ROOT data as they are, but put individual *TBaskets* in the object store. TFile/TTree subclasses fetch data from the object store instead of seeking to file positions.

1. Presents the same TFile/TTree interface to users; **old scripts still work.**
2. But data replication, storage class, and caching are handled by the object store with columnar granularity.
3. Branches are shared transparently across derived datasets: all trees are friends.

Idea #2 (this talk). Keep ROOT data as they are, but put individual *TBaskets* in the object store. TFile/TTree subclasses fetch data from the object store instead of seeking to file positions.

1. Presents the same TFile/TTree interface to users; **old scripts still work.**
2. But data replication, storage class, and caching are handled by the object store with columnar granularity.
3. Branches are shared transparently across derived datasets: all trees are friends.
4. The logic of sharing, reference counting branches, managing datasets, etc. must all be implemented in ROOT; only ROOT understands how to combine branches.

Idea #2 (this talk). Keep ROOT data as they are, but put individual *TBaskets* in the object store. TFile/TTree subclasses fetch data from the object store instead of seeking to file positions.

1. Presents the same TFile/TTree interface to users; **old scripts still work.**
2. But data replication, storage class, and caching are handled by the object store with columnar granularity.
3. Branches are shared transparently across derived datasets: all trees are friends.
4. The logic of sharing, reference counting branches, managing datasets, etc. must all be implemented in ROOT; only ROOT understands how to combine branches.

(the "ROOT becomes the database" approach)

How it could be done

- Subclass of TFile initializes itself by getting data from a “controlling” database (document store like MongoDB might be best).

How it could be done

- Subclass of TFile initializes itself by getting data from a "controlling" database (document store like MongoDB might be best).
- Reference counts for objects referenced by TKeys (including TBaskets and user objects like histograms) are maintained by this controlling database.

How it could be done

- Subclass of TFile initializes itself by getting data from a "controlling" database (document store like MongoDB might be best).
- Reference counts for objects referenced by TKeys (including TBaskets and user objects like histograms) are maintained by this controlling database.
- Bulk data, the contents of TKeys, are in a "warehouse" database (object store—*might* be the same database). Optimal basket size may be big, like megabytes.

How it could be done

- Subclass of TFile initializes itself by getting data from a "controlling" database (document store like MongoDB might be best).
- Reference counts for objects referenced by TKeys (including TBaskets and user objects like histograms) are maintained by this controlling database.
- Bulk data, the contents of TKeys, are in a "warehouse" database (object store—*might* be the same database). Optimal basket size may be big, like megabytes.
- REST APIs for flexibility; TBaskets fetched by HTTP GET, may be web-cached. No new ROOT dependencies.

How it could be done

- Subclass of TFile initializes itself by getting data from a "controlling" database (document store like MongoDB might be best).
- Reference counts for objects referenced by TKeys (including TBaskets and user objects like histograms) are maintained by this controlling database.
- Bulk data, the contents of TKeys, are in a "warehouse" database (object store—*might* be the same database). Optimal basket size may be big, like megabytes.
- REST APIs for flexibility; TBaskets fetched by HTTP GET, may be web-cached. No new ROOT dependencies.
- Methods for deriving new TTrees from old TTrees:
  - share common TBranch data by default;

How it could be done

- Subclass of TFile initializes itself by getting data from a "controlling" database (document store like MongoDB might be best).
- Reference counts for objects referenced by TKeys (including TBaskets and user objects like histograms) are maintained by this controlling database.
- Bulk data, the contents of TKeys, are in a "warehouse" database (object store—*might* be the same database). Optimal basket size may be big, like megabytes.
- REST APIs for flexibility; TBaskets fetched by HTTP GET, may be web-cached. No new ROOT dependencies.
- Methods for deriving new TTrees from old TTrees:
  - share common TBranch data by default;
  - "soft skim" by stencil (event list/event bitmap), "hard skim" only if re-basketization is needed to compactify results (keeping fewer than $\sim$10% of original);

How it could be done

- Subclass of TFile initializes itself by getting data from a "controlling" database (document store like MongoDB might be best).
- Reference counts for objects referenced by TKeys (including TBaskets and user objects like histograms) are maintained by this controlling database.
- Bulk data, the contents of TKeys, are in a "warehouse" database (object store—*might* be the same database). Optimal basket size may be big, like megabytes.
- REST APIs for flexibility; TBaskets fetched by HTTP GET, may be web-cached. No new ROOT dependencies.
- Methods for deriving new TTrees from old TTrees:
  - share common TBranch data by default;
  - "soft skim" by stencil (event list/event bitmap), "hard skim" only if re-basketization is needed to compactify results (keeping fewer than $\sim$10% of original);
  - save all provenance and use git-like versioning to determine if two branches are related/may be combined (for a join by index position, rather than mutual column).

How it could be done

- Subclass of TFile initializes itself by getting data from a "controlling" database (document store like MongoDB might be best).
- Reference counts for objects referenced by TKeys (including TBaskets and user objects like histograms) are maintained by this controlling database.
- Bulk data, the contents of TKeys, are in a "warehouse" database (object store—*might* be the same database). Optimal basket size may be big, like megabytes.
- REST APIs for flexibility; TBaskets fetched by HTTP GET, may be web-cached. No new ROOT dependencies.
- Methods for deriving new TTrees from old TTrees:
  - share common TBranch data by default;
  - "soft skim" by stencil (event list/event bitmap), "hard skim" only if re-basketization is needed to compactify results (keeping fewer than $\sim$10% of original);
  - save all provenance and use git-like versioning to determine if two branches are related/may be combined (for a join by index position, rather than mutual column).
- No user-facing partition boundaries: huge dataset appears as one TTree.

How it could be done

- Subclass of TFile initializes itself by getting data from a "controlling" database (document store like MongoDB might be best).
- Reference counts for objects referenced by TKeys (including TBaskets and user objects like histograms) are maintained by this controlling database.
- Bulk data, the contents of TKeys, are in a "warehouse" database (object store—*might* be the same database). Optimal basket size may be big, like megabytes.
- REST APIs for flexibility; TBaskets fetched by HTTP GET, may be web-cached. No new ROOT dependencies.
- Methods for deriving new TTrees from old TTrees:
  - share common TBranch data by default;
  - "soft skim" by stencil (event list/event bitmap), "hard skim" only if re-basketization is needed to compactify results (keeping fewer than $\sim$10% of original);
  - save all provenance and use git-like versioning to determine if two branches are related/may be combined (for a join by index position, rather than mutual column).
- No user-facing partition boundaries: huge dataset appears as one TTree.
- Users work in shared TFile: home TDirectories; permissions managed by database.

Direct connection

User launches ROOT, does `TFile::Open("rootdb://data.cern/cms")`, and extracts objects for analysis: `Get("home/username/myhist")->Draw()`.

Two modes of use

Direct connection

User launches ROOT, does `TFile::Open("rootdb://data.cern/cms")`, and extracts objects for analysis: `Get("home/username/myhist")->Draw()`.

Job submission

User passes a macro, `TTree::Draw` request, or `TDataFrame` to a service that parallelizes it and puts results in the user's home TDirectory.

Direct connection

User launches ROOT, does `TFile::Open("rootdb://data.cern/cms")`, and extracts objects for analysis: `Get("home/username/myhist")->Draw()`.

Job submission

User passes a macro, `TTree::Draw` request, or `TDataFrame` to a service that parallelizes it and puts results in the user's home TDirectory.

- compute nodes use this same interface to communicate with storage;

Two modes of use

Direct connection

User launches ROOT, does `TFile::Open("rootdb://data.cern/cms")`, and extracts objects for analysis: `Get("home/username/myhist")->Draw()`.

Job submission

User passes a macro, `TTree::Draw` request, or `TDataFrame` to a service that parallelizes it and puts results in the user's home TDirectory.

- compute nodes use this same interface to communicate with storage;
- but a scheduler attempts to maximize shared cache locality on the compute nodes.

Two modes of use

Direct connection

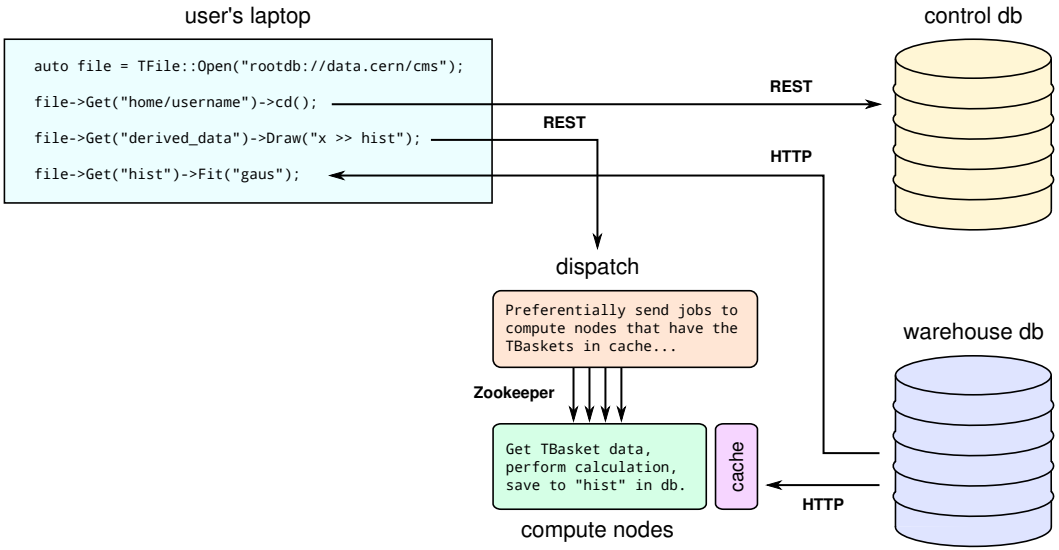
User launches ROOT, does `TFile::Open("rootdb://data.cern/cms")`, and extracts objects for analysis: `Get("home/username/myhist")->Draw()`.

Job submission

User passes a macro, `TTree::Draw` request, or `TDataFrame` to a service that parallelizes it and puts results in the user's home TDirectory.

- compute nodes use this same interface to communicate with storage;
- but a scheduler attempts to maximize shared cache locality on the compute nodes.

This is the "query server" idea I've been exploring for some time now, except that all of the interface is ROOT.


user's laptop
auto file = TFile::Open("rootdb://data.cern/cms");
file->Get("home/username")->cd();
file->Get("derived_data")->Draw("x >> hist");
file->Get("hist")->Fit("gaus");
control db
REST
REST
HTTP
dispatch
Preferentially send jobs to compute nodes that have the TBaskets in cache...
Zookeeper
warehouse db
Get TBasket data, perform calculation, save to "hist" in db.
cache
HTTP
compute nodes

Questions for you

Question: How would you feel if I developed this kind of service *within* ROOT (idea #2), rather than outside of ROOT (idea #1)?

Questions for you

diana hep

Question: How would you feel if I developed this kind of service *within* ROOT (idea #2), rather than outside of ROOT (idea #1)?

I'd want to sketch it out in Python (my uproot project) to figure out the architecture before committing to the ROOT codebase: ∼year timescale.

Questions for you

Question: How would you feel if I developed this kind of service *within* ROOT (idea #2), rather than outside of ROOT (idea #1)?

I'd want to sketch it out in Python (my uproot project) to figure out the architecture before committing to the ROOT codebase: ∼year timescale.

Question: Deeply nested columnar splitting, zero-copy structure manipulations, and many database indexing techniques are not possible with today's ROOT serialization.

Questions for you

Question: How would you feel if I developed this kind of service *within* ROOT (idea #2), rather than outside of ROOT (idea #1)?

I'd want to sketch it out in Python (my uproot project) to figure out the architecture before committing to the ROOT codebase: ∼year timescale.

Question: Deeply nested columnar splitting, zero-copy structure manipulations, and many database indexing techniques are not possible with today's ROOT serialization.

Are you interested in forward-incompatible changes to ROOT serialization that would make these things possible? I could propose them as a ROOT 7 serialization format.

Questions for you

Question: How would you feel if I developed this kind of service *within* ROOT (idea #2), rather than outside of ROOT (idea #1)?

I'd want to sketch it out in Python (my uproot project) to figure out the architecture before committing to the ROOT codebase: ∼year timescale.

Question: Deeply nested columnar splitting, zero-copy structure manipulations, and many database indexing techniques are not possible with today's ROOT serialization.

Are you interested in forward-incompatible changes to ROOT serialization that would make these things possible? I could propose them as a ROOT 7 serialization format.

(Subject of another talk.)