ROOT Columnar Storage Evolution

J. Blomer (CERN, EP-SFT)

ROOT
Data Analysis Framework
https://root.cern

- The problem we are solving
- The contracts of the new interface
- Code examples

The Tree

- ROOT's column-wise collection format is empirically the best we can do
  - A solution designed by us for our very problem
- Only few other column-wise formats
  - Apache Parquet (Google Dremel): optimized for deep, sparse collections: our data is not sparse
  - Apache Arrow: in-memory only format
- ROOT's unique feature: seamless C++ integration
  - Users do not need to write/generate schema mapping (which is *lots of* boilerplate code)

Goals

- Speed
  - Design for vectorized and bulk I/O
  - Stay columnar even in deeply nested structures
- Robust interfaces
  - Compile-time safety if necessary
  - Separation of concerns to simply I/O extensions such as new storage systems
- Indicate sorted columns for indexing (e.g., timestamps)

```
class Hadron;
class Jet {
  std::vector<Hadron> hds;
};
class Event {
  std::vector<Jet> jets;
};
```

Challenging but opportunity to deprecate least used features

Note: RDataFrame covers many current TTree use cases

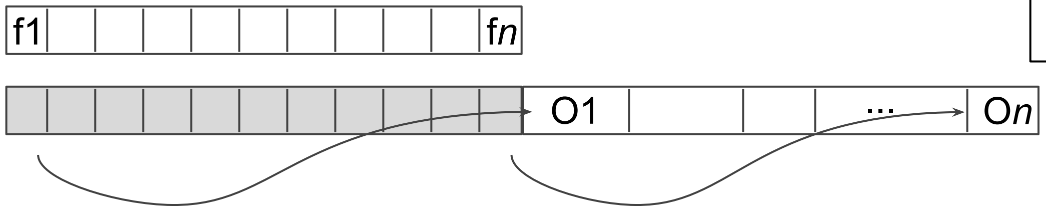
How we want to get there

- Separate high-level logical data layout (C++ classes) from low-level physical data layout (columns of simple types)
  - Mapping of data to storage devices only needs to know the low-level types
  - For simple classes (e.g. struct of float), in-memory representation should equal on-disk representation
  - Building block for vectorization and bulk I/O
- On the logical data layout: separate between static part (Schema / Tree Model) and dynamic part (entries that are being read and written)
  - A tree schema is composable, natural support for friend trees
  - Multiple entries for the same tree model can exist in parallel: building block for multi-threading
- Asymmetric interfaces for reading and writing
  - Saves locking logic when reading data

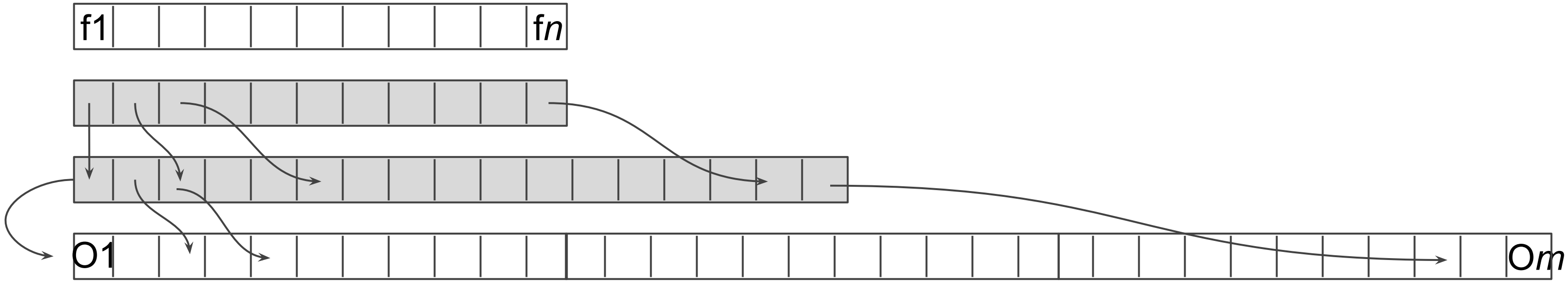
Sketch of the updated columnar layout

```
using Collection = std::vector<float>
class Event {
  std::vector<Collection> outer;
  float flat;
};
```

- TTree



- New design: “unfolded” nesting (same data volume)



The contracts of the new interface

- New classes
  - RTreeModel: branch names and types, composable
    - Can be shared by multiple trees
  - RTreeSource, RTreeSink: storage strategy
    - Either writing or reading
  - RTreeView
    - Lazy branch access, natural underpinning for TTreeReaderValue
- Interfaces
  - Explicit pointer ownership, entries are shared between the tree and the user code
  - Compile-time type-safety where possible (`Branch<Event>`)
  - Possibility to calculate values on `Fill()`
  - Vector interfaces: optimise reading/writing *n* entries at a time
  - Reading through hierarchical iterators (clusters, entries)

Milestones for 2018

- Prototyping phase
  - Define the user interface
  - Identify new classes
  - Demonstrator for reading and writing nested Ntuples of fundamental types of different size (say, float and long)

Sample Code: TreeModel

```
auto tree_model = std::make_shared<RTreeModel>();

auto event = /* shared pointer to Event */
   tree_model->Branch<Event>( "my_event" /*, { constr args }*/);
auto h1_px = tree_model->Branch< float>("h1_px ", 0.0);

auto track_model = std::make_shared<RTreeModel>();
auto track_energy = track_model->Branch< float>("energy");
auto tracks = tree_model->BranchCollection( "tracks", track_model);
// Resolves to branches "tracks" and "tracks.energy"
```

Ideas for more advanced writing

```
// calculate on fill from other values
tree_model->Branch<float>("is_exotic")->Bind(
   [event = event]() -> float { return (event->fEnergy < 0) ? 0.9 : 0.1; });

// Allow to capture a user provided shared pointer
auto calibration = std::make_shared<TCalibration>();
tree_model->Branch<TCalibration>("calibration")->Capture(calibration);

// Support decoupled writer modules that don't have the types available
// at compile time; type-checked at runtime using TClass.
auto branch_dynamic = tree_model->BranchDynamic("custom", "TUserClass");
// Can then be bound to a pointer + size
```

Filling

```
ROutputTree tree(tree_model, RTreeSink::MakeFileSink( "/a/b/c"));
// Possible to use other sinks, e.g. TTreeSink::MakeHDF5Sink
// We can reuse the tree model but not the tree medium

// Scalar filling as before
for (auto i = 0; i < 100; i++) {
  tree->Fill();
}

auto entry = tree->CreateEntry();
auto event = entry->Get("event");
...
tree->FillV(/* span (array) of entries */);
```

Reading

Two ways of reading:

1) High-level: for analysis - declarative approach to be adopted (a la TDF)
2) Low-level: for frameworks and power users

This is the low level approach

```
// TreeMedium provides file chaining functionality
// could also mix branches from different trees
RInputTree tree(tree_model, RTreeSource::MakeFileSource({ "/tree1",
"/tree2"}));

for (auto e : tree.GetEntryRange()) {
   // Populate shared storage locations given by tree model
   std::cout << px << std::endl;
   std::cout << event->fEnergy << std::endl;
}
```

Hierarchical iteration

```
// Hierarchical iteration
for (auto cluster : tree.GetClusterRange()) {
   for (auto entries : cluster.GetEntryRange()) {
   }
}

// Entry ranges could possibly be create more sophistically,
// for instance based on sorted branches such as timestamps
```

Lazy reading

```
// Open without model, allow for lazy views
RInputTree tree(RTreeSource::MakeFileSource({ "/tree1", "/tree2"}));

auto view_px = tree.GetView< float>("px");
auto view_chi2 = tree.GetView< float>("chi2");

for (auto e : tree.GetEntryRange(RRangeType::kLazy)) {
   if (view_px(e) > 1.0) {
      std::cout << view_chi2(e) << std::endl;
   }
}
```