RDataFrame

Riley Xu

April 29, 2022

Introduction

- New ROOT interface to read/write TTrees[1]
- Available in v6.14+ (June 2018) and actively updated
- Much faster than `TTree::Draw()`, `TTree::GetEntry()`, or `TTreeReader`
  - Multi-threading
  - Parallel actions per event loop
  - Optimized filtering and I/O
- My use case: plotting filtered TH2s from 56M events in 20 seconds vs 1 hour with `GetEntry()`.
- These slides: a lot of code examples, hopefully a useful future reference

[1] Documentation

Basic Use

```cpp
#include "ROOT/RDataFrame.hxx"

// Create the dataframe from a root file
ROOT::RDataFrame df("tree_name", file_path);

// Fill a histogram using branch "jet_m" (lazy)
auto h = df.Histo1D("jet_m");

// Access a result which triggers the event loop
h->Draw();
```

- The `Histo1D()` function is a lazy action
  - Actions are registered, not run
  - Returns a smart pointer (`RResultPtr`)
  - Dereferencing the pointer triggers the action
- All registered actions run in parallel when the event loop is triggered

Multiple Actions

Correct:

```
auto h_m  = df.Histo1D("jet_m");
auto h_pT = df.Histo1D("jet_pT");

h_m->Draw(); // this triggers the event loop
h_pT->Draw(); // this result is cached

cout << df.GetNRuns(); // "1"
```

Wrong:

```
auto h_m  = df.Histo1D("jet_m");
h_m->Draw(); // this triggers the event loop

auto h_pT = df.Histo1D("jet_pT");
h_pT->Draw(); // this triggers the event loop again

cout << df.GetNRuns(); // "2"
```

Transformations

```
// Register transformations
auto d2  = df.
    Range(0, 1000000).      // only process a range of events
    Filter("jet_pT > 200"); // cut on value of branch "jet_pT"

// Register actions using the filtered dataframe
auto h1 = d2.Histo1D("jet_m");
auto h2 = d2.Histo1D("X_m");
```

- Transformations return a new database[2]
- Like actions, they are only registered
- Run when the event loop is triggered

[2] Actually, a reference to the node in the computation graph

Define and Snapshot

```cpp
// Filter the dataframe and create a "pT" column
auto df2 = df.Define("pT", "sqrt(px*px + py*py)").
              Filter("MET > 200").
              Filter("pT > 100");

// Save some columns of the filtered dataframe
df2.Snapshot("myTree", "myFile.root", {"MET", "pT"});
```

- `Define()` is a transformation that creates a new column
- `Snapshot()` is an instant that writes a new TTree
- Instants trigger the event loop and evaluate immediately

Lazy Evaluation

```cpp
// Define before filter
auto h1 = df.Define("pT", "sqrt(px*px + py*py)").
             Filter("MET > 200").
             Histo1D("pT");

// Define after filter
auto h2 = df.Filter("MET > 200").
             Define("pT", "sqrt(px*px + py*py)").
             Histo1D("pT");
```

- These two calls are equivalent
- `Define` is only run when all `Filter` pass
- Generally can put all `Define` at the beginning

Useful Functions

Transformations: Manipulates data, returns a new dataframe

```
Define() // create a new column
Filter() // filter entries based on column values
Range()  // select a subset of entries (single-thread only)
```

Actions: Retrieves a result, returns a lazy pointer

```
Aggregate() // accumulate column values with a custom operation
Book()      // register a custom action
Fill()      // register a custom fill function
Graph()     // create a TGraph from columns
Histo1/2D() // create a TH1/2 from columns
Take()      // get a column as a std::vector
```

Instants: Operations that happen immediately

```
Foreach()  // custom operation on each event
Snapshot() // write the dataframe to a new TTree
```

Intermediate Usage

Fully Compiled Expressions

```cpp
// String version; requires JIT
df.Filter("MET > 200");

// Using lambda function, argument must have matching type
auto cut = [](double MET) { return MET > 200; }
df.Filter(cut, {"MET"});

// Using normal function
bool cut(double MET) { return MET > 200; }
df.Filter(cut, {"MET"});
```

- String expressions require just-in-time compilation (JIT)
- Use a function/functor for better performance and flexibility

Fully Compiled Typing

```cpp
// Implicit types; requires JIT
auto h = df.Histo1D("pT", "weight");

// Explicit types; fully compiled
auto h = df.Histo1D<float, double>("pT", "weight");
```

- RDataFrame will implicitly determine branch types (JIT)
- Specify template parameters to improve performance

Multithreading

```cpp
// Enable multithreading
ROOT::EnableImplicitMT();

// Make sure any RDataFrame is created AFTER the call above
ROOT::RDataFrame df("tree_name", file_path);
```

- RDataFrame can parallelize the event loop
- Machine with 8 cores ⟹ 16x speedup
- Gotcha: User-defined expressions must be thread-safe
- Side-effect free functions are thread-safe by default

Thread Safety

- Use alternate functions such as `DefineSlot()` or `ForeachSlot()`
- Takes the thread id as an extra argument
- Example: Implement `Sum()` with `ForEach()`

```cpp
// Unsafe version of Sum
double sum = 0;
auto adder = [&sum](double x) { sum += x; };
df.Foreach(adder, {"branch"});

// Safe version of Sum
vector<double> sums(df.GetNSlots(), 0);
auto adder = [&sums](unsigned slot, double x) { sums[slot] += x; };
df.ForeachSlot(adder, {"branch"});
double sum = accumulate(sums.begin(), sums.end(), 0);
```

Advanced Usage

Callbacks

- Callback every N events with `OnPartialResult`
- Function doesn't need to be thread safe
- Example: print a progress message

```
auto h = df.Histo1D("pT");
auto callback = [](TH1D &h_) { cout << h_.GetEntries() << endl; }
h.OnPartialResult(1e5, callback);
```

Custom Actions

- Define custom actions with `Book`
- Can do arbitrary actions on any number of columns
- See manual and `Book` docstring

```
class MyAction : public ROOT::Detail::RDF::RActionImpl<MyAction> {
public:
   // Advertise the type of the result
   using Result_t = int;

   // Address of the result that will be filled
   std::shared_ptr<Result_t> GetResultPtr() const;

   // Called at every entry
   void Exec(unsigned slot, double pT);

   // Called at the end of the event loop
   void Finalize();
};

auto result = df.Book<double>(MyAction(), {"pT"});
```

Conclusion

- RDataFrame is the new and superior way to read/write TTrees
- Much much faster than `TTreeReader`, implicit multithreading
- Workflow consists of transformations and actions that are lazy evaluated
- Works in pyRoot too!

Backup