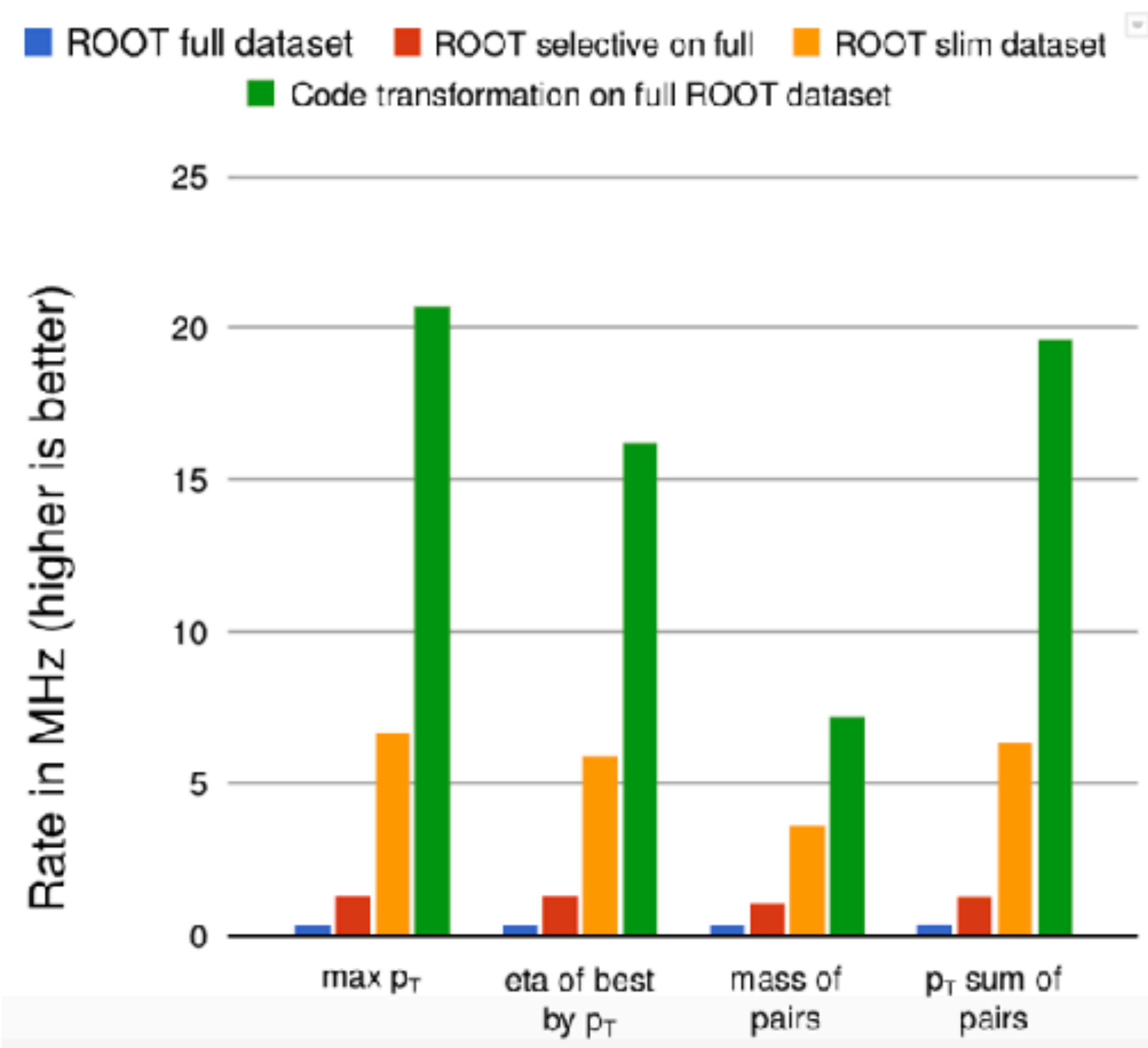
Bulk IO Update

Brian Bockelman

Recap: Sales Pitch

- Experiment event data models are complex — and slow to read!
  - Experiments don't care because input I/O time is minimal compared to reconstruction.
  - Experiments care about volume because they have lots of expensive disk.
- Analysis is different: data model is often simple.
  - Much smaller data volume. Often on SSD (now NVMe).
  - Minimal CPU costs: iterate over events many times, quickly.
  - **I/O Speed is king!**
- Bulk I/O is an approach to deliver a cluster of events at once

What’s possible?



See details here: https://arxiv.org/pdf/1708.08319.pdf

Here, we are apply four simple kernels across a 5.4M event dataset. Iterating over muon objects in an event; each muon has 42 attributes.

- “**full dataset**” - ROOT reads all 42 attributes from each muon in the event.
- “**ROOT selective on full**” - ROOT reads 3 attributes out of 42.
- “**ROOT slim dataset**” - Separately prepare a derived dataset with only the 3 relevant attributes, use ROOT to iterate on that.
- “**Code transformation on full ROOT dataset**” - Python implementation of the kernels, analyzing the full dataset. Data is delivered via bulk IO.

N.B.: Same compute kernels applied to in-memory arrays (no I/O) are ~3x faster. Exception is “mass of pairs”, which is more compute-bound.

ROOT Bulk IO API

- Add a new public method to TBranch:

```
ROOT::Experimental::Internal::TBulkBranchRead &
        GetBulkRead();
```

- GetBulkRead returns a dummy object inside the `ROOT::Experimental::Internal` namespace.
    - All bulk I/O operations occur with `TBulkBranchRead`.
    - Awkward separation is on purpose: clearly telegraph to users that this is experimental / internal.
    - N.B.: Philippe has suggested this could switch from being a dummy object to use of inheritance…

TBulkBranchRead

- Has 4 basic public functions; we'll walk through the precise arguments in a second:

```
Int_t  GetEntriesFast(Long64_t evt, TBuffer& user_buf, bool checkDeserializeType=true);
Int_t  GetEntriesSerialized(Long64_t evt, TBuffer& user_buf, bool checkDeserializeType=true);
Int_t  GetEntriesSerialized(Long64_t evt, TBuffer& user_buf, TBuffer* count_buf, bool
                            checkDeserializeType=true);
Bool_t SupportsBulkRead() const;
```

- The `GetEntries*` calls is essentially the "bulk IO API".

- Meant to allow highly optimized usage for experts. *Not* meant for users.

    - Example: initial PR includes C code to directly export ROOT branches into numpy arrays.

    - We don't expect every grad student to write such a thing…

GetEntriesFast

```
Int_t  GetEntriesFast(Long64_t evt, TBuffer& user_buf, bool checkDeserializeType=true);
```

- Given an event at the beginning of an basket, return the deserialized objects in a user-provided buffer.
    - Caller is expected to keep track of basket boundaries.
    - On success (return code >=0), buffer points to deserialized event data. Return code is the number of events in the buffer.
    - If the branch holds a single double, then the double from `evt+idx` is at:

```
reinterpret_cast<double*>(user_buf.GetCurrent())[idx]
```

`GetEntriesFast` - Continued

- `checkDeserializeType` provides a way to bypass (sometimes costly) internal type checks. Can be set to `false` after first successful bulk IO read.
- Caller does not own the memory in `user_buf` on successful return - it's managed by the TBranch.
  - **Suggestion from Philippe**: instead of sharing ownership of user buffer, swap contents of buffer and internal basket.
- Caller must track: basket beginning and end boundaries; branch type.
- Caller must know how to iterate correctly through data in buffer (most useful for fixed-size branches).
- Overall, quite a bit for the caller to do! (Recall, this is meant to be the internal interface)

GetEntriesSerialized

```
Int_t  GetEntriesSerialized(Long64_t evt, TBuffer& user_buf, bool checkDeserializeType=true);
Int_t  GetEntriesSerialized(Long64_t evt, TBuffer& user_buf, TBuffer* count_buf, bool
                            checkDeserializeType=true);
```

- Similar to `GetEntriesFast`, but the resulting buffer contains the raw serialized data.
  - Second overload also optionally returns a 'counts buffer' in the case of arrays. Returns the number of events per entry in the `user_buf`.
- For `int`-typed branches, the returned data is in big endian ordering.
- Why such a raw interface?
  - Some consumers (`numpy`) can work with the big-endian data directly.
  - If we deserialize (e.g., byteswap) in the ROOT I/O libraries, we may iterate through a large array, flushing the processor cache in the process.
    - `TTreeReaderFast` will inline the byteswap when the values are accessed by the user code.

When does bulk IO work?

- Bulk IO can only work with a limited number of cases; we can only manipulate the data in-place, meaning the deserialized object in memory must be smaller-than-or-equal-to the serialized byte stream.
  - Primitive types.
  - C-style structs. (Nothing with virtual pointers!).
  - arrays or std::vector of basic types.
  - No references or pointers.
- Quite limiting compared to ROOT's full capabilities: likely not that limiting for analysis users!
  - Current version has some additional caveats (very limited ability to handle `TLeafElement`).

TTreeReaderFast

- Consider sample TTreeReader code:

```
TTreeReader myReader("T", hfile);
TTreeReaderValue<float> myF(myReader, "myFloat");
Long64_t idx = 0;
Float_t sum = 1;
while (myReader.Next()) {
  sum += *myF;
}
```

  - **Observation**: here, `TTreeReaderValue<float>` provides compile-time guarantees about the object type.

- **Idea**: write a `TTreeReaderFast` class that manages the `TBuffer` and basket boundary management in `GetEntriesFast`.

  - **myReader.Next()** could be inlined by compiler, avoiding function calls unless a new basket is needed.

  - Since the compiler knows the branch type, ***myF** would invoke the appropriate deserialization code via template specialization.

TTreeReaderFast

- `TTreeReaderFast` provides a user-friendly interface on top of the bulk IO API.
  - Unfortunately, since we use inlining techniques, compiler must be told branch is “bulk I/O friendly”. (Not clear if fallback to TTreeReader is possible.)

```
ROOT::Experimental::TTreeReaderFast reader("DecayTree", hfile);
ROOT::Experimental::TTreeReaderValueFast<int> val_h1_is_muon(reader, "H1_isMuon");

reader.SetEntry(0);
Long64_t idx = 0;
for (auto it : reader) {
  idx++;
}
printf("There were %lld events read.\n", idx);
```

Next Steps

- Cross the finish line. Technical work has been stalled for a few months.
- Zero-copy interface: If `TFile` was extended with mmap-compatible interfaces, we could avoid memory copies.
- Continue to expand object types and branches that can use the bulk IO API.
- Merge testing suite into `roottest`; add to `rootbench`.
- Figure out how to make this usable by `TDataFrame` (and whether TDF is fast enough for this to be relevant).