Boosting RDataFrame performance with transparent bulk event processing

Enrico Guiraud[1,*], *Jakob* Blomer[1], *Philippe* Canal[2], and *Axel* Naumann[1]

[1]CERN
[2]FNAL

Abstract. RDataFrame is ROOT's high-level interface for Python and C++ data analysis. Since it first became available, RDataFrame adoption has grown steadily and it is now poised to be a major component of analysis software pipelines for LHC Run 3 and beyond. Thanks to its design inspired by declarative programming principles, RDataFrame enables the development of high-performance, highly parallel analyses without requiring expert knowledge of multi-threading and I/O: user logic is expressed in terms of self-contained, small computation kernels tied together by a high-level API. This design completely decouples analysis logic from its actual execution, and opens several interesting avenues for workflow optimization. In particular, in this work we explore the benefits of moving internal data processing from an event-by-event to a bulk-by-bulk loop. This refactoring dramatically reduces the framework's runtime overheads; in collaboration with the I/O layer it improves data access patterns; it exposes information that optimizing compilers might use to auto-vectorize the invocation of user-defined computations; finally, while existing user-facing interfaces remain unaffected, it becomes possible to additionally offer interfaces that explicitly expose bulks of events, useful e.g. for the injection of GPU kernels into the analysis workflow. In order to inform similar future R&D, design challenges will be presented, as well as an investigation of the relevant time-memory trade-off backed by novel performance benchmarks.

1 Introduction

The ROOT framework [1] is a fundamental component of today's high-energy physics (HEP) analysis software ecosystem. In version 6.14 (2018) ROOT introduced RDataFrame [2]: an ergonomic, modern interface to express HEP analysis workflows in C++ and Python. Similarly to what happens in commonly used data processing tools such as Dask [3] or Spark [4] (which inspired its design), RDataFrame decouples the definition of a computation graph that represents the analysis logic from the graph's actual execution (see Fig. 1); this enables transparent optimizations such as parallel and distributed processing [5], and it hides the complexity of the interaction with ROOT's I/O layer. At the same time, being explicitly designed with HEP use cases in mind, RDataFrame's API includes features that allow users to easily express concepts and operations that are typical of this field: notable examples are systematic variations or the filtering and manipulation of sub-collections.

*e-mail: enrico.guiraud@cern.ch

RDataFrame has seen widespread adoption across CERN experiments and beyond, also serving as the foundation for more specialized frameworks (e.g. [6], [7]). It was shown to provide a sweet spot in performance and ergonomics for HEP use cases when compared to more general-purpose query languages [8]. Nevertheless, RDataFrame keeps evolving [9], introducing features such as the aforementioned native support for expressing systematic variations or a tight integration with RNTuple [10], ROOT's new and experimental data format.

In the following sections we will present the benefits as well as the design challenges involved in moving RDataFrame's internal event loop from an event-by-event processing to bulk-by-bulk. This large refactoring makes memory access patterns friendlier to the CPU caches; it restructures the inner data loop so that compilers have better chances for optimizations; it makes it possible to optionally expose bulks of events to user code, enabling the development of highly optimized computation kernels for common HEP operations and, in the future, the injection of GPU kernels into the analysis workflow, e.g. for machine learning inference tasks. Such a large paradigm shift requires close collaboration with the I/O layer in order to remove per-event overheads related to data reading and preparation. The refactoring also poses some interesting design questions that are not specific to RDataFrame but rather relate to HEP's data format of choice and to the general tension between the often inherent per-event granularity of user (and physics) requirements and the needs of modern hardware, which is optimized for many-at-a-time I/O operations and computations.

After discussing the redesign and the related design questions in Sec. 2, in Sec. 3 we will present novel benchmarks that highlight the performance benefits as well as the current limitations of the R&D work presented.

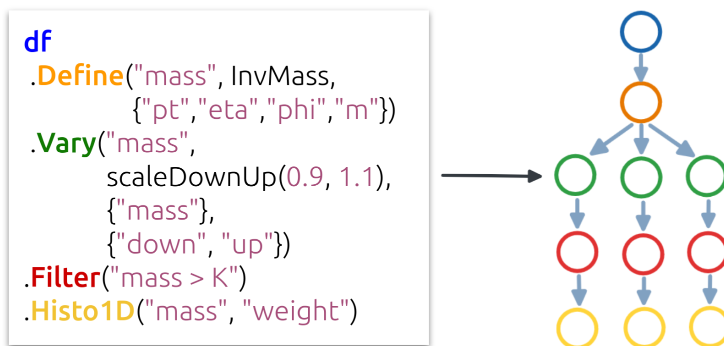

Figure 1. *Left:* A simple RDataFrame analysis that defines a new quantity "mass", attaches systematic variations to it, filters interesting events and finally fills a weighted histogram. *Right:* The corresponding computation graph: in this case the three branches represent the nominal case and the up and down variations.

2 Bulk event processing in RDataFrame

Fig. 1 shows the computation graph for a simple RDataFrame analysis. Users express their logic in terms of event-wise operations that often nicely correspond to the underlying physics ideas. However, with traditional event-by-event processing, RDataFrame performs one traversal of the computation graph for each entry: column values required by each node are prepared for reading, the node's operation is performed, and then execution moves on to

the next node in the graph. Therefore, for each execution thread, the CPU will do little work before having to switch to a different context: for sufficiently large computation graphs the CPU instruction and data caches will be continuously invalidated. In addition, in this processing mode RDataFrame communicates with ROOT's I/O layer once per entry to ensure that the required column values are correctly loaded. Most of the time ROOT I/O will have already loaded those values into system memory (it will not read data from disk or network for every event) but it will have to update internal pointers to correctly refer to the requested entry and perform a number of sanity checks. Furthermore, in the current RDataFrame API user-defined operations such as `InvMass` in Fig. 1 can only operate on one entry at a time. This makes it effectively impossible to write specialized computation kernels that take advantage of CPU vectorization or that dispatch computations to accelerators such as GPU.

In summary, high-energy physics analysis tasks are often naturally expressed in an event-oriented fashion, while hardware favors executing the same instructions on large chunks of contiguous memory. RDataFrame's current implementation incurs the per-event overheads described above because it does not take the hardware side of the equation into consideration.

However, it is in principle possible to load and prepare *bulks* of entry values at a time and then execute each node of the computation graph on many entries at a time. Per-event overheads would then become per-bulk overheads, and the CPU data and instruction caches would not be continuously invalidated. Not only, but if RDataFrame performed such a bulk-wise event loop it would also become possible to offer additional interfaces to users that expose bulks of column values: advanced users could then opt into this API in order to implement specialized computation kernels that leverage CPU vectorization or heterogeneous computing hardware, while other users can transparently make use of such operations with no changes to the interfaces they know (see Listing 1). In Sec. 3 we also present performance benchmarks that leverage this new RDataFrame "bulk API".

Several design questions arise when exploring a paradigm shift such as the one described above. The main ones are discussed in the rest of this section.

2.1 Bulk event filtering: event masks or value copies

A common analysis operation is the filtering of interesting events for further processing. Different branches in the RDataFrame computation graph can filter different sets of events. An event-wise event loop can simply quit the processing of a branch early if at some point a filter discards the event. A bulk-wise implementation has two main options: it can either have different nodes read data from shared memory locations, *masking out* values that are discarded by event filters, or alternatively it can prepare, for each node of the computation graph, a new bulk that only contains values for valid events. In real-world analyses, graphs can have many different branches that will read the same columns: masking only requires to keep track of one boolean mask per branch of the computation graph, while actual column values can be shared, rather than copying values in new bulks for each node of the graph. In our experience, in high-energy physics broad graphs are more common than deep graphs (especially when each systematic variation corresponds to a new branch, as it is in RDataFrame) so we preferred to avoid per-branch data duplication.

2.2 Bulk data loading: greedy and lazy loading of column values

ROOT's columnar data format and its implementation are named TTree. TTree can load and expose bulks of values at the granularity of a "basket" (i.e. a collection of column values that are compressed together); the boundaries of the baskets are misaligned across columns (see Fig. 2). RNTuple works similarly. As a consequence, in order to load a given range of events

```cpp
// scalar API: still available and recommended as default
float square(float x) { return x*x; }

// bulk API: for specialized performance-sensitive computations
void square(const REventMask &m,
            RVecF &results, // RVec is an ergonomic 1D array
            const RVecF &xs) {
        // operate on raw buffers to expose simpler code
        // to the optimizing compiler
        float *out = results.data();
        // `results` is guaranteed to have large enough size
        float *in = xs.data();

        // intentionally ignore the event mask
        // to better leverage the compiler's autovectorization
        const auto size = m.Size(); // the size of this bulk
        for (std::size_t i = 0; i < size; ++i)
                out[i] = in[i] * in[i];
}

// usage of the two versions of `square` is identical:
// complexity is encapsulated in the operation's implementation
df.Define("x2", square, {"x"});
```

Listing 1: Example usage of the new RDataFrame bulk API to perform a vectorized square calculation. The bulk operation must be implemented in C++ but can then also be used with a Python RDataFrame thanks to PyROOT's dynamic bindings.

across all columns, RDataFrame stores column values in its own cache, at the cost of a copy and extra memory usage. For each execution thread, all computation graph branches share the same column value cache.

The speed-up from bulk processing will have to offset the runtime cost of the extra copies to result in a net gain. Given that different branches of the computation graph might need to access different sets of events in a given range, RDataFrame can then either greedily load all column values in the range or lazily load the values as they are requested. In practice we have found that the best approach depends on the column. On one hand, columns with simple types (i.e. fundamental types or arrays thereof) coming from a TTree or RNTuple dataset are relatively cheap to load into the RDataFrame cache via a `memcpy` and, for TTree, the copy can be performed at the same time of a byteswap. On the other hand, columns with complex data types might require more complex deserialization logic and copying such values might be more expensive than a `memcpy`; also, column values produced by user-defined expressions might not be well-formed for filtered-out events.

RDataFrame therefore applies a hybrid strategy in which it greedily loads all values in a bulk for columns with simple types coming from a dataset; for all others, it instead loads only the column values requested. An additional per-column event mask is used to keep track of which column values are loaded.

2.3 Choosing an appropriate bulk size

How many events should be included in each bulk? While we would like the bulk size to be as large as possible for performance reasons, larger-than-memory datasets are common in HEP

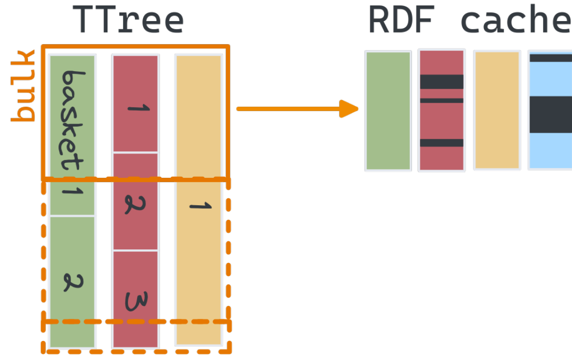

Figure 2. A TTree dataset (left) consists of multiple columns, or "branches". The new RNTuple format has a similar structure. For each column, groups of values are compressed together into "baskets". RDataFrame bulks are transversal to TTree baskets and might cross their boundaries for some of the branches. When preparing the input values for a given operation, column values are copied into dedicated memory buffers (the RDataFrame value cache, right) for each bulk. Column values pertaining to events that are filtered out by event selection might not be present in the cache, depending on how cheap they are to produce (see Sec. 2.1). Note that some columns are computed on the fly and do not come from the TTree, like the one in blue here.

so we need to somehow limit the bulk size accordingly. Furthermore, as mentioned above, RDataFrame's per-thread, per-column value cache represents an extra memory cost that is proportional to the bulk size. Finally, if we are to enable the implementation of specialized computation kernels that leverage CPU vectorization, the size should also be large enough to fill a modern CPU's registers dedicated to vector operations. For example, for the AVX-512 instruction set we would need at the very least 16 single-precision floating-point values in a bulk, or better yet multiple such blocks. For GPU processing buffers should be much larger.

Other constraints on the bulk size come from our data format of choice, i.e. TTree (again, similar arguments apply for RNTuple). As an optimization, TTree pre-fetches groups of column values in an internal cache that is shared by all columns. The range of entries that are pre-fetched together is called a "cluster". If RDataFrame moved back and forth across cluster boundaries when switching from the bulk loading of one column to the next, the pre-fetching cache would constantly be invalidated. Therefore RDataFrame bulks should not cross TTree cluster boundaries. For analogous reasons, when processing multiple trees (which can be vertically concatenated in what is commonly called a "chain" or horizontally concatenated as "friend trees"), a bulk of events should not cross over into any new tree: there are tree-wide, expensive operations involved in switching from one tree to the next. In practice, for each required column and each thread, RDataFrame pre-allocates memory for a certain *maximum* bulk size chosen ahead of time, and then for each new bulk the actual size is negotiated with the I/O layer so that it is not larger than the pre-allocated maximum as well as not larger than the number of entries still available in the TTree cluster being processed for each horizontally concatenated TTree.

The performance benchmarks in Sec. 3 have been instrumental in verifying that a "Goldilocks zone" of bulk sizes that satisfy these requirements actually exists: given our current benchmarks it is situated between 16 and 4096 events.

2.4 Flattening sub-collections for efficient access patterns

A final interesting problem is related to a common feature of HEP datasets, namely the presence of sub-collections (e.g. a collection of momenta of electrons for every entry). This gives the dataset a "jagged" or "ragged" shape (both terms are used in the literature). To allow efficient, possibly vectorized CPU access to the data, it is desirable to flatten these bulks of arrays (logically arrays of arrays) into a single contiguous buffer. For columns with dynamic size, then, the RDataFrame cache of Fig. 2 resizes itself to accommodate all elements of all arrays for every bulk of entries. Users are then served views over the flattened array of arrays which represent per-event data. Exposing the flattened buffers to users in a friendly manner is the subject of current R&D work.

3 Performance benchmarks

In this section we investigate whether RDataFrame's bulk processing actually provides the expected performance benefits. The implementation we evaluate is an advanced prototype that includes all mechanisms and optimizations described in the previous sections and that supports all but a few RDataFrame features. It is available at [11]. Source code for the benchmarks is available at [12]. Measurements are run on a Intel i7-10875H CPU with Turbo Boost, Speedstep and hyper-threading turned off (8 physical threads are present). The Linux CPU frequency governor is set to "performance". All code is built with GCC 12.2.1 and `O3` optimization level (note that using `O3` is recommended for RNTuple, which, in our tests, incurred significant slowdowns at `O2`). Runtimes reported are those provided by RDataFrame's logging facility itself and only include the time spent in the event loop, excluding any setup phase (e.g. the building of the computation graph). Maximum resident memory is measured with the `/usr/bin/time` Linux utility program.

All input files were compressed with ZSTD and read from warm filesystem cache: bulk processing only speeds up CPU processing and it does not impact I/O performance, so we factor out disk I/O. I/O-limited RDataFrame applications or applications for which decompression is the bottleneck will not benefit much from the transparent performance improvements but will still be able to leverage the new bulk API.

Two benchmark tasks are taken into consideration. The first is RDataFrame's "Dimuon tutorial": an example analysis based on CMS Open Data that matches muon pairs and produces a histogram of the dimuon mass spectrum. As such, it includes the main staples of HEP analyses: event selections, calculation of derived quantities, operations on collections, histogramming. From a purely computational standpoint, more complex analyses are often "more of the same". Around sixty million events are processed with data in the form of scalars or arrays thereof, similar to the CMS NanoAOD schema. The second benchmark ("ATLAS iotools") is inspired by an ATLAS Open Data tutorial that has previously been used to evaluate RNTuple performance [13]. It nicely complements the first benchmark as the prevalent column types in this schema are C++ standard library vectors, which are more complex to treat in terms of bulk I/O.

First of all we check what effect the bulk size parameter discussed in Sec. 2.3 has on runtimes and memory usage. Fig. 3 shows that, for the "Dimuon tutorial" benchmark, a bulk size of 16 is enough to reap the performance benefits, and memory usage is not significantly impacted until large bulk sizes over 10^5. The HEP analysis landscape is extremely varied and it can be dangerous to generalize results from a single application; nevertheless, our experience to date with several benchmarks suggests that these trends are general: there is a large range of viable bulk sizes that users can pick from depending on the needs of the hardware, and memory usage is not drastically impacted by RDataFrame's extra value caches.

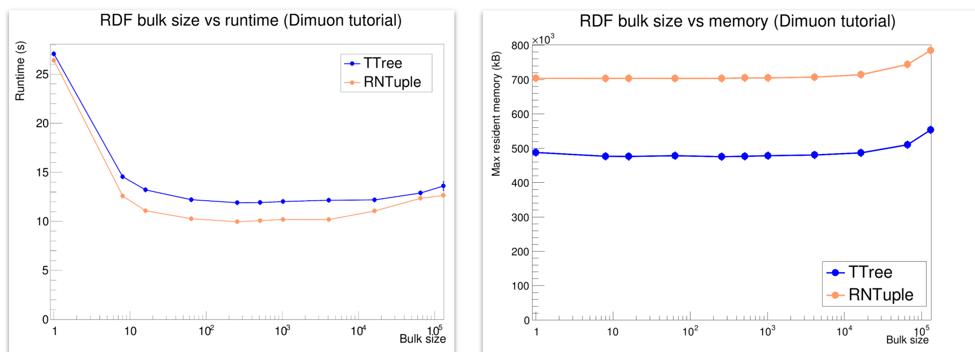

Figure 3. Runtime and maximum resident memory usage of the “Dimuon tutorial” benchmark for bulk RDataFrame, for different bulk sizes. Note that the x-axis is logarithmic. Error bars representing standard deviation across three measurements are present but almost invisible.

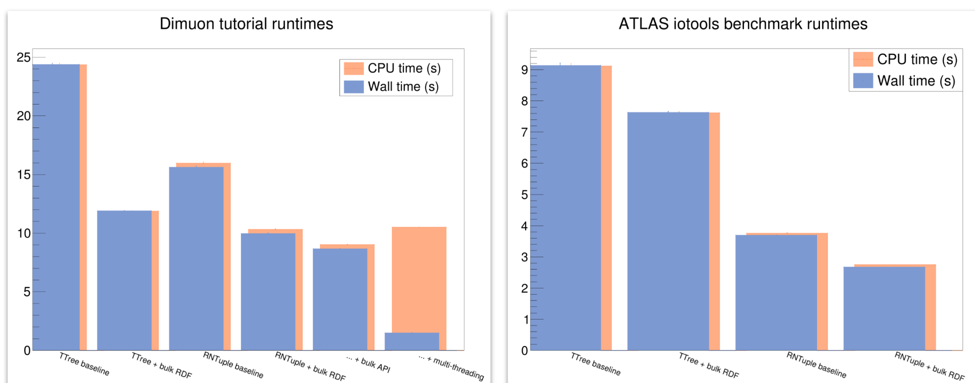

Figure 4. Runtimes for the “Dimuon tutorial” (left) and the “ATLAS iotools” (right) benchmarks for different set-ups. CPU time is in orange and wallclock time in blue. The “baseline” runtimes are without bulk processing, “+ bulk API” indicates that the invariant mass calculation is performed with an ad-hoc implementation that makes use of RDataFrame’s bulk API.

Fig. 4 reports CPU time and wallclock time for the two benchmarks taken in consideration, in several different set-ups. The overall speed-up for the simple data types of the “Dimuon tutorial” is around 2x for TTree and 1.6x for RNTuple. A more CPU-friendly version of the invariant mass calculation provides an extra 15% speed-up, showing that RDataFrame’s bulk API actually works and that vectorized computing kernels can in principle be helpful, although the effort is probably not worth it in this particular case: speed-ups will be larger when the calculation takes a larger fraction of the total runtime and/or when the vectorized computation kernel can dispatch them to a GPU. As expected, speed-ups from bulk processing compound nicely with those from multi-threading, as they are orthogonal: the design and implementation of RDataFrame’s bulk processing does not introduce any sort of thread contention. The “ATLAS iotools” benchmark, to the right of Fig. 4, shows that bulk processing provides smaller speed-ups with more complex column types: C++ standard library vectors of floating point values are more difficult to deserialize and process in bulks,

and C++ vectors of booleans introduce extra complications because of their bit-packing optimization. As a result, the speed-up is $1.2x$ for TTree and $1.4x$ for RNTuple: RNTuple's design allows a simpler treatment of C++ standard vectors.

4 Conclusions

We showed there is potential for a much faster RDataFrame for common analysis use cases, especially for simpler schemas made of scalars and simple C-style arrays thereof such as CMS NanoAODs. Bulk-wise computation kernels can further speed up expensive computations.

A few challenges remain before RDataFrame bulk processing can be released to the general public, which will be addressed in the near future. For example, the APIs and behaviors of some existing RDataFrame features such as user-defined callbacks do not map well to bulk-wise processing and will have to be adapted accordingly. In addition, as discussed, an ergonomic design to expose flattened versions of jagged arrays via the bulk API is also the subject of current research.

Our implementation of RDataFrame's bulk processing redesign is available at [14].

References

[1] R. Brun, F. Rademakers, Nuclear instruments and methods in physics research section A: accelerators, spectrometers, detectors and associated equipment **389**, 81 (1997)

[2] D. Piparo, P. Canal, E. Guiraud, X. Valls Pla, G. Ganis, G. Amadio, A. Naumann, E. Tejedor Saavedra, EPJ Web Conf. **214**, 06029 (2019)

[3] Dask Development Team, *Dask: Library for dynamic task scheduling* (2016), `https://dask.org`

[4] M. Zaharia, R.S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M.J. Franklin et al., Communications of the ACM **59**, 56 (2016)

[5] V.E. Padulano, I.D. Kabadzhov, E. Tejedor Saavedra, E. Guiraud, P. Alonso-Jordá, Journal of Grid Computing **21**, 9 (2023)

[6] P. David, EPJ Web Conf. **251**, 03052 (2021), `2103.01889`

[7] S. Brommer, S. Wunsch, nfaltermann, ralfschmieder, M. Oh, nshadskiy, E. Guiraud, A. Gottmann, M. Burkart, felix phy et al., *KIT-CMS/CROWN: v0.3* (2023), `https://doi.org/10.5281/zenodo.8325327`

[8] D. Graur, I. Müller, M. Proffitt, G. Fourny, G.T. Watts, G. Alonso, Journal of Physics: Conference Series **2438**, 012034 (2023)

[9] E. Guiraud, J. Blomer, S. Hageboeck, A. Naumann, V. Padulano, E. Tejedor, S. Wunsch, *RDataFrame enhancements for HEP analyses*, in *Journal of Physics: Conference Series* (IOP Publishing, 2023), Vol. 2438, p. 012116

[10] J. Blomer, P. Canal, A. Naumann, D. Piparo, *Evolution of the ROOT tree I/O*, in *EPJ Web of Conferences* (EDP Sciences, 2020), Vol. 245, p. 02030

[11] E. Guiraud, *RDataFrame bulk processing (commit used for benchmarks)*, `https://github.com/eguiraud/root/tree/df-bulk-chep2023` (2023)

[12] E. Guiraud, *RDataFrame benchmarks*, `https://github.com/eguiraud/rdf-benchmarks` (2023)

[13] J. Lopez-Gomez, J. Blomer, *RNTuple performance: status and outlook*, in *Journal of Physics: Conference Series* (IOP Publishing, 2023), Vol. 2438, p. 012118

[14] E. Guiraud, *RDataFrame bulk processing (latest version)*, `https://github.com/eguiraud/root/tree/df-bulk` (2023)