Performance Visualization of ROOT/IO on HPC Storage Systems

SEPTEMBER 2021

AUTHOR(S):
Rui Pedro Neto Reis

EP-SFT

SUPERVISOR(S):
Javier Lopez Gomez
Jakob Blomer

CERN openlab

PROJECT SPECIFICATION

HPC systems are becoming ever more important as a data processing resource for the LHC experiments. HPC sites typically use storage systems different from the well-understood HEP storage systems. Current HPC sites deploy high-performance cluster file systems (Lustre, Ceph-FS, GPFS), sometimes amended by burst buffers. Next-generation exascale systems are expected to gradually move from file systems to object stores (*e.g.,* Intel DAOS) to address storage scalability challenges. On the application side, the ROOT I/O library provides the predominant I/O base layer for LHC data handling. To optimally use so-far untapped storage systems, applications built on top of the ROOT I/O need to be tuned for the target storage system at hand (*e.g.*, in terms of I/O block sizes, level of I/O parallelism, networking link parameters, *etc*).

This project aims at providing visualizations of key performance indicators of the RNTuple I/O sub-system. The existing framework for collecting performance metrics should be extended to not only keep metric aggregates but also metric histograms. Metric aggregates and histograms should be presented as several performance overview plots, *e.g.,* request size distribution, distribution of the I/O queue depth, access pattern of the data. As a first application, the new visualizations can be used to better understand and improve the RNTuple storage backend for Intel DAOS.

ABSTRACT

Traditionally, the TTree columnar format has been used for HEP data storage. RNTuple, the new, experimental ROOT I/O subsystem, is a backwards-incompatible redesign that addresses several TTree shortcomings.

RNTuple currently provides a framework to collect performance metrics that help understanding where performance bottlenecks are. However, metrics are collected in the form of aggregates, *e.g.*, number of bytes read, time spent in reading/decompressing, *etc*.

To have a more accurate figure, the metrics framework could be improved by adding histograms, which would allow to gain insight into the distribution of the collected data. By tapping into the use of such histograms, allied with metric aggregates, RNTuple can now directly build, in a compact format, the data distribution associated with the metric being studied. This data can be later analysed by external tools as it’s exported in the CSV format.

TABLE OF CONTENTS

1. INTRODUCTION

As an addition to the ROOT I/O system, RNTuple provides a more efficient data-storage format in order to replace the previous TTree format. Besides its storage capabilities, RNTuple also provides a range of metrics regarding its I/O operations. However, it only stores an aggregate value for each performance counter, which limits the ability to further analyze collected data. In this project, we propose several histogram building techniques for the extraction of more useful information, such as the distribution of a performance counter.

The visual construction of a histogram could be done having access to the full history of all performance counters. However, when dealing with an HPC Storage System, this is not always possible. This is mainly because it would cause a severe performance impact on the system, as it would be necessary to be constantly dumping data onto a file. Besides that, it's also very dubious to assume one has the capacity to store any amount of information, and thus the system would eventually fail due to memory-shortage.

Thus, the problem becomes more complex, and techniques to build such histograms must be optimized. That means any data passed onto the metric histogram must be immediatly placed in a compact manner such that, after the job is finished, it is possible to directly extract the histogram data without further processing. However, for this approach to work, the histogram buiding technique must be as simple and efficient as possible, in order to imply minimal overhead in the overall project.

In this report, we introduce four histogram building techniques, which differ from each other in complexity and properties. The last of those techniques has been assigned to be the most complete and useful as it's the most intuitive. Nevertheless, the other techniques are presented as to demonstrante the action possibiliies brought into play by these techniques.

Initially we present the description and details of the inner-workings of each of the techniques, followed by the steps took to automatically test such techniques and how the collected data can be later ingested by external plotting utilities.

2. IMPLEMENTATION DETAILS

In order to develop a technique which minimizes the computational overhead, several models were developed. Each of the histogram building techniques require different arguments, and thus their functionality is limited accordingly. As such, different types of histograms provide a distinct number of bins and interval ranges. The following techniques have been established, which were designed by us, with the exception of the fixed width bin technique, that gains inspiration from external sources mentioned in the appopriate subsection.

- **User-Provided Set of Intervals** – Explicitly count values in specified intervals.
- **Log Scale** – Count values in the log-2 scale of the corresponding value.
- **Active Learning Phase** – Adaptative building technique which adjusts itself to the first N samples in order to create an equality separated number of bins.
- **Fixed Width Bins** – Take advantage of the binary representation of unsigned integer types, in order to determine the histogram bin for a given value with minimal overhead.

The above techniques are further portrayed in detail in the following subsections. The full implementation can be found on ROOT pull request #8880[1].

[1] https://github.com/root-project/root/pull/8880

a. User-Provided Set of Intervals

The utility for user-provided set of intervals is notorious. One may seek to only analyse a specific set of intervals to get more specific insights. In the current version, this technique is encapsulated by the class `RNTupleHistoInterval` and relies on binary search for the identification of the correct bin.

For each user-specified interval, an ordered vector by each interval's lower limit is created, using each of the intervals provided. For example, for the following intervals: (40,50), (10,20) and (70,80), a structure resembling Figure 1 is internally created.

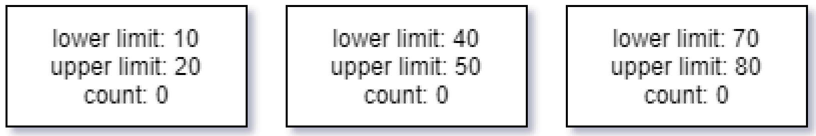


Figure 1 Internal structure for histogram intervals

Thus, when a new value is added, a binary search on the closest lower limit is performed. When a possible match is found, the corresponding counter is increased if the provided value is lower than the interval's upper limit. However, this approach suffers greatly from its lack of generalization.

This building technique also suffers from its inability to detect outliers, because the user has the responsibility to supply the intervals. Since the interval of occurrence of specific outliers is unlikely to be passed by the user, the building technique often can't detect them.

b. Log Scale

Due to the need for a type of building technique which doesn't require the need to explicitly define intervals, we've created the `RNTupleHistoCounterLog`, which implicitly counts the values by their corresponding log-2 scale exponent. For example, the value 9 would be accounted for in the exponent 3, as would all the values from 8 to 15.

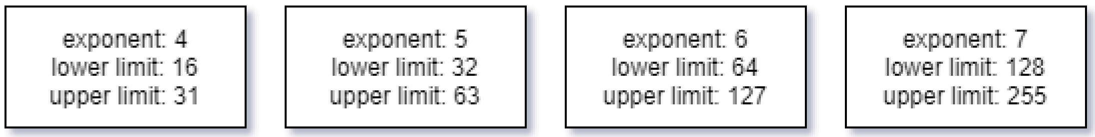


Figure 2 Internal structure for log scale histogram

Figure 2 represents the internal structure implicitly created by this histogram building technique. This kind of structure is very efficient, since the only complexity is calculating the log-2 of an integer, which can be easily performed via the use of the technique described in [1]. Despite its efficiency, the log-2 scale doesn't correctly account for values on a large scale, mainly because the interval amplitude of subsequent intervals, in this building technique, is exponentially large.

In one of our experiments, accounting for the size of read requests when loading a RNTuple page, we found the log scale building technique inappropriate and providing no useful information. This happened because, when accounting the number of bytes, a large scale is expected. So, in our case, only exponent 16 would be accounted for, as Figure 3 portrays, exponent 16 has a too great amplitude, and thus does not provide any useful information. A new technique, independent of scale, is necessary.

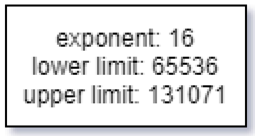


Figure 3 Interval for exponent 16

Regarding the ability to detect outliers, this technique is more robust since, depending on the scale, outliers are easily caught up on a drastically lower or greater exponent bin.

c. Active Learning Phase

As seen above, user-specified intervals and the assumption of a numeric scale are not typically desired. One solution is to first employ a learning phase, in which we sample a specific number of the numeric values from the incoming distribution, and, after having an *a-priori* measurement of the minimum and maximum bounds, we can create an arbitrary amount of equally separated bins to accommodate for the observed data amplitude, values viewed in the learning phase are then placed on the newly built bins. Figure 4 illustrates how this technique works internally.

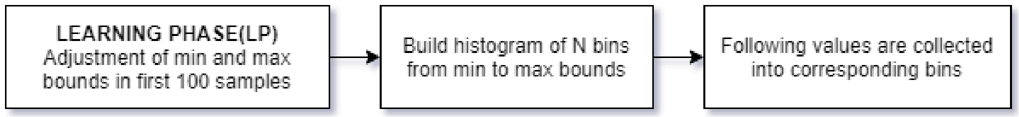

LEARNING PHASE(LP)
Adjustment of min and max bounds in first 100 samples
Build histogram of N bins from min to max bounds
Following values are collected into corresponding bins

Figure 4 Simplification of the learning phase technique

However, this technique also exhibits some drawbacks, mainly due to the assumption that the data distribution of the sampled values in the learning phase remains constant through the remaining entries. This is not always the case, since in many situations, later entries, outside of the learning phase, more strongly portray the full amplitude of the distribution.

This technique also suffers by the occurrence of outliers in the learning phase, since the bounds are adjusted to the extremes of the values, the bins generated will not be representative of the original distribution, since they are unaware of the outlier's existence.

d. Fixed Width Bins

The lack of outlier-detection capabilities is the biggest downside of the methods above. Neither provide a robust and reliable way to precisely indicate if a value is a true outlier or not. Besides that, the above methods also make a large amount of assumptions about the underlying data distribution, *i.e.*, log scale, that the first sample are representative of the real distribution, *etc*. Having too many assumptions hardens the generalization capabilities of our building technique, and thus are not adequate for a diverse range of scenarios.

There is a need for an unbiased building technique, which can easily generalize to whatever it is fed by an external source. Since we are analysing a constrained set of types, mainly unsigned long integers, we can take advantage of the numbers' binary representation.

Based upon an aggregation technique from [2] we derive an efficient solution for the matching of a value to its bin. This solution assumes the user has provided a width and offset; these values are then used to create a histogram with a fixed width starting from the offset. In the following paragraphs, we refer to β as being the number of bins which can fit below the provided offset. Figure 5 represents the fixed width bins technique, using width = 100, offset = 170 and β = 2.

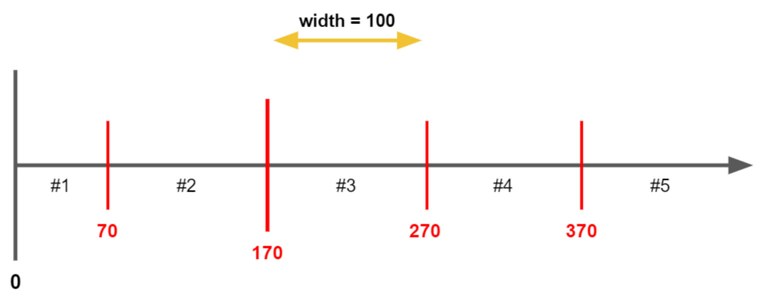


Figure 5 Representation of the fixed width bins technique

The technique to assign each value to a corresponding bin can be described as:

- If a new value, N, is **greater or equal** to the **offset**, then:
  - $\text{Key} \leftarrow \frac{(N - \text{offset})}{\text{width}} + \beta + 1$
- Else:
  - $\text{Key} \leftarrow \beta - \frac{(\text{offset} - N)}{\text{width}}$

The need for branching is to make sure the unsigned integer does not underflow when working with subtractions. With the above key-bin attribution technique, we can easily assign a value to its corresponding bin. A `std::unordered_map` is used to store all the created keys and corresponding atomic counters. This building technique is easily the most simple and complete of the solutions provided, besides not suffering from the tremendous downsides associated in the remaining techniques.

3. OUTPUT FORMAT

After the analysis, regardless of the chosen technique, the bin intervals and their associated value can be dumped into a CSV file for later analysis by external utilities. In the current format, each histogram, is dumped into a separate CSV file. Thus, a performance evaluation running with multiple histograms will generate various text files. This is something which can be improved in the future, namely finding a more efficient output format. The current format is presented as follows, the header line being the variable names, followed by each interval, in order, and their specific count.

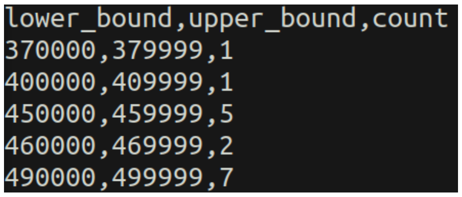

```
lower_bound,upper_bound,count
370000,379999,1
400000,409999,1
450000,459999,5
460000,469999,2
490000,499999,7
```

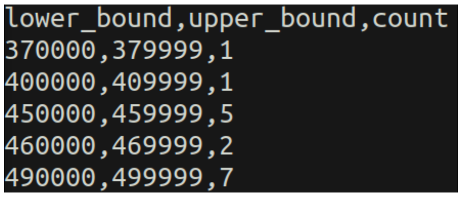
Figure 6 Current output CSV format

4. EXPERIMENTAL EVALUATION

The experimental evaluation was conducted on data readily available on the ROOT project repository, mainly referring to examples using the RNTuple and format conversion between the old format, TTree. More information on the data used can be found here[2].

a. RNTuple Tutorial Nº5

When analysing the RNTuple tutorial nº5 [3], we can use the fixed-width building technique to get a sense of the sparsity of the size of read requests when loading data pages. By enabling the default metrics, which now include a histogram of the size of read requests, we can obtain the following contents dumped in a CSV format.



```
lower_bound,upper_bound,count
370000,379999,1
400000,409999,1
450000,459999,5
460000,469999,2
490000,499999,7
```

Figure 7 Example of content dumped by histogram

In the example of Figure 7, we employed a fixed-width histogram with a width of 10000. This type of output can be easily fed onto external plotting utilities for a visual analysis of the data. Figure 8 presents the visual representation of the intervals portrayed above.

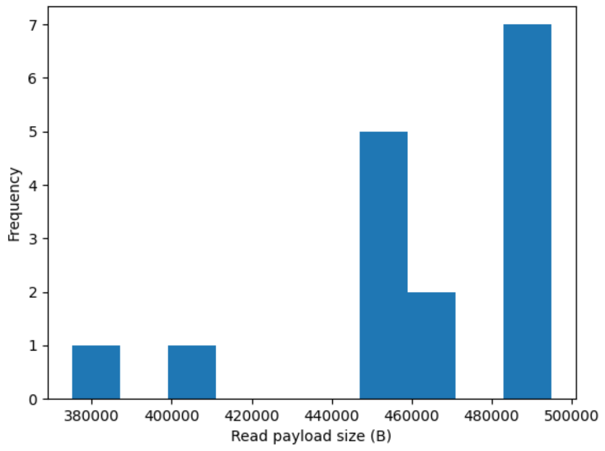


Figure 8 Visual representation of the 1st example

Using such a plot, we can understand that the size of read requests in this tutorial is unevenly distributed. With this kind of information, we could deploy new patches to further optimize access to data, *e.g.*, adapt buffer sizes to distribution, batching techniques, *etc*.

b. Convert LHCb run 1 open data from TTree to RNTuple format

Following the same logic as the previous experiment, we now try to get a sense for the size of read requests when loading a page, this time analysing the conversion between LHCb run 1 open data in the TTree format

2 https://github.com/root-project/root/tree/master/tutorials/v7/ntuple

to RNTuple [4]. Figure 9 is the output we get, here we can easily detect the existence of an outlier, most likely corresponding to the size of the last read request, using this knowledge we gain new insights into understanding the conversion impact on the RNTuple performance metrics, in this case we can understand that the size of read requests, when converting formats, is mostly constant.

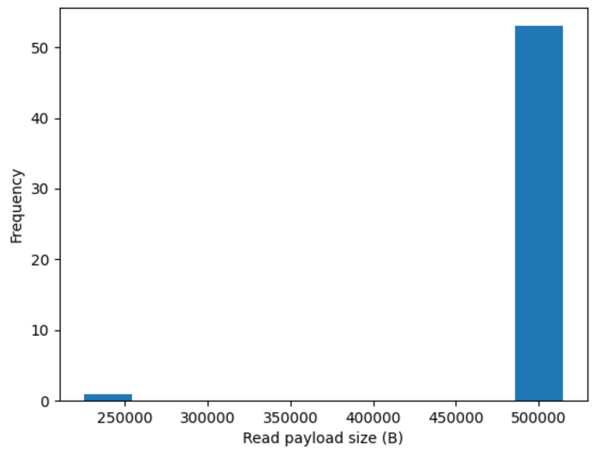


Figure 9 Histogram generated by LHCb run 1 open data conversion

5. CONCLUSIONS

As initially proposed, we can verify that the addition of more complex types of metrics into RNTuple, allows for a more concrete analysis of performance. With such metrics more in-depth analysis of RNTuple performance can be easily constructed, with minimal computational overhead. The fixed width bins building technique, chosen as the most complete and viable, provides a computational advantage over the remaining techniques, as it makes use of the underlying binary data representation.

Nonetheless, the remaining techniques can still allow for analysis of performance on other scenarios, when a more specialized approach is preferred over the general solution. Some of the building techniques portrayed could be also extended as to include non-integer interval bounds, *e.g.,* float, which was not the aim of this research. This data can also be easily exported into CSV format, such that it can be easily digested by external plotting utilities to visually observe the data.

As a future work it would be interesting to study more complex output format types, which can store various histogram representations into a single file. A deeper dive into visualization tools and ingestion pipelines for the histogram output file would also be an interesting path to follow.

6. BIBLIOGRAPHY

[1] S. E. Anderson, "Bit Twiddling Hacks," 5 May 2005. [Online]. Available: https://graphics.stanford.edu/~seander/bithacks.html#IntegerLogObvious. [Accessed 17 September 2021].

[2] Elasticsearch, "Histogram Aggregation," [Online]. Available: https://www.elastic.co/guide/en/elasticsearch/reference/current/search-aggregations-bucket-histogram-aggregation.html. [Accessed 17 September 2021].

[3] The ROOT Team, "Write and read an RNTuple from a user-defined class," [Online]. Available: https://github.com/root-project/root/blob/master/tutorials/v7/ntuple/ntpl005_introspection.C. [Accessed 25 September 2021].

[4] The ROOT Team, "Convert LHCb run 1 open data from a TTree to RNTuple," [Online]. Available: https://github.com/root-project/root/blob/master/tutorials/v7/ntuple/ntpl003_lhcbOpenData.C. [Accessed 25 September 2021].