Exploring data merging methods for a distributed processing system

Piotr Konopka and Barthélémy von Haller
The ALICE experiment, CERN, 1211 Geneva 23, Switzerland
E-mail: piotr.jan.konopka@cern.ch

Abstract.
The ALICE experiment at the CERN LHC (Large Hadron Collider) is undertaking a major upgrade during the LHC Long Shutdown 2 in 2019-2021, which includes a new computing system called O^2 (Online-Offline). The raw data input from the ALICE detectors will increase a hundredfold, up to 3.5 TB/s. By reconstructing the data online, it will be possible to compress the data stream down to 100 GB/s before storing it permanently.

The O^2 software is a message-passing system. It will run on approximately 500 computing nodes performing reconstruction, compression, calibration and quality control of the received data stream. As a direct consequence of having a distributed computing system, locally generated data might be incomplete and could require merging to obtain complete results.

This paper presents the O^2 Mergers, the software designed to match and combine partial data into complete objects synchronously to data taking. Based on a detailed study and results of extensive benchmarks, a qualitative and quantitative comparison of different merging strategies considered to reach the final design and implementation of the software is discussed.

1. Introduction

ALICE (A Large Ion Collider Experiment)[1] is one of the four major particle detectors at the CERN Large Hadron Collider (LHC)[2]. During the operational break in 2019-2022 the ALICE experiment is, among many activities, entirely replacing the key tracking sub-detectors to achieve higher spatial and time resolution. This will dramatically increase the amount of produced data, reaching a value of 3.5 TB/s [3]. To cope with the increased data throughput, a new Online-Offline computing system, called O^2, is being deployed [4]. The software relies on a message passing architecture, consisting of multiple processes exchanging data via message queues with a zero-copy approach. The computing farm consists of approximately 450 servers performing data acquisition, aggregation, processing, quality control and calibration.

Consequently, the data reaching the system will be distributed among many nodes. Many tasks will run in parallel on the main processing nodes. Each of their instances will compute only a fraction of the data and produce partial results. This requires a piece of software which efficiently merges widely used objects in the O^2 system, like histograms and tables, but also data structures designed and implemented for specific purposes.

The ALICE experiment has already performed data merging in its online reconstruction farm in previous years [5]. It was implemented as a message-passing process in its highly parallel architecture. This paper benefits from the experience and expertise of the developers who designed and implemented this system. The ATLAS experiment has reported using *Gatherers*

in their system [6][7]. The software can sum up or substitute objects coming from parallel sources and publish the result on demand or in specified intervals. The gathering might be performed in various stages (tiers) to split the load among multiple processes. The LHCb experiment takes advantage of *Adders* [8], which perform the same task. They also might be arranged as a tree - first to collect all data on the same node, then on sub-farm and farm levels, and finally aggregating the results of the complete processing system.

This paper explores different approaches to merging data in the High Energy Physics context. Performance, scalability and reliability of the presented solutions are evaluated.

2. Systematization

A review of the literature shows that the topic of this paper does not have a firmly established taxonomy. Even Mergers are referred by different names: Gatherers and Adders, and it is likely that other terms also exist. To avoid a confusion, this section defines the terminology, classification and assumptions used in this paper.

Merging or *merger* is an operation which combines two or more *input objects* into one. It is performed by *merging algorithms*. An object contains *data points* - sets of one or more measurements coming from one observation. These might be e.g. rows in table or singular entries in a histogram bin. If an object contains only a subset of all data points in the system, it is deemed *incomplete*, otherwise it is *complete*. *Data sources* send incomplete objects to *Mergers*, which are the applications, actors or processes responsible for merging them and publishing merged objects. Mergers can send complete objects to *data receivers*. An input object which contains only data points, which a Merger has not yet received is called a *delta*. An input object which contains all data points collected so far by a data source is an *entire* object. *Data types* or *data structures* define how data points are organised within an object. It is assumed that the order of merging objects does not influence the result size, which is true for histograms and tables (addition is commutative, row order does not change the table's size). The result has the same type as the incomplete objects. Also, there exists only one correct result for each merger, however it might be achieved with more than one algorithm.

Finally, the object types are discriminated according to the data size after merging:

- *Shrinking* - A result of the merger operation is smaller than the smallest input object.
- *Fixed-size* - A result of the merger operation has the same size. This category includes: scalar numbers, collections of scalars, histograms with identical bin ranges or combining multiple images into one with the same size.
- *Growing* - A result of the merger operation is larger than the largest input object: merging histograms with incoherent or sparse binning, joining tables (adding new columns or rows) or assembling any objects into collections.
- *Other* - Any other rules that does not fall into the preceding categories.

Merged data structures might be composites of simpler types. In the Quality Control system (QC) these are usually collections of different histograms and tables. They are merged one-by-one by matching their names. Alternatively, they are copied into the target collection if they do not have a counterpart there yet.

3. Merger design and investigation paths

As O^2 is a high-throughput message-passing system, the described software also follows this design choice. In order to distribute the load on several nodes, a multi-layer topology of Merger processes (Fig. 1) is assumed, where data Producers send input messages in regular time intervals. Mergers then receive objects via input channels, merge and pass them forward to a consecutive layer. The last one has only one Merger, which sends complete objects on a

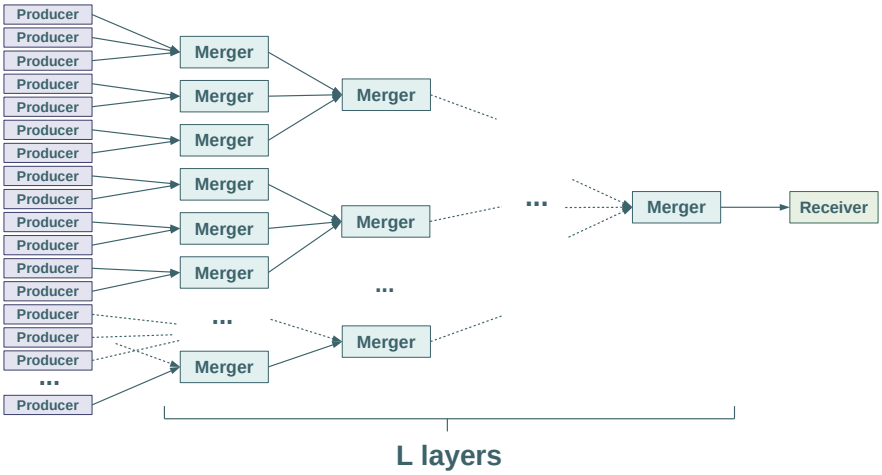

Figure 1: A multi-layer topology of Mergers.

topology output to the Receiver. In most of the cases only one layer of Mergers (i.e. just one Merger) is expected to be enough, with the exception of highly demanding data structures or large number of supported Producers.

If i is the layer number, 0 being the producer layer and Mergers start from layer 1, then the amount of Mergers per layer M_i can be calculated in the following ways:

- With L as the number of layers:
$$M_i = \lceil M_0^{\frac{L-i}{L}} \rceil \tag{1}$$
- With R as the global reduction factor - the maximum number of inputs handled by one Merger:
$$M_i = \lceil \frac{M_{i-1}}{R} \rceil \tag{2}$$

Then, the reduction per each layer R_i and the number of layers L become:

$$R_i = \frac{M_i}{M_{i-1}} \tag{3}$$

$$L = \lceil log_R M_0 \rceil \tag{4}$$

Inside the Merger logic, there is an optional cache, which is supposed to store incoming objects before merging them. The merging algorithm combines input data into the most recent complete object, using a defined method in the object's interface. The algorithm should expect either deltas (Fig. 2) or the most recent, entire versions on each node (Fig. 3). If there is more than one object stored in cache, merging a collection of them at one time might be beneficial. Objects can be merged periodically or when new object updates arrive. The publication of complete objects is performed periodically.

The presented design served as a base for investigating the merging methods and differences in performance and reliability, which follow in the next section.

4. Exploring data merging methods

In this section, the differences between merging deltas and entire objects are discussed in terms of efficiency and reliability. Additionally, the possibility to merge objects in cached collections is evaluated. Lastly, the influence of the number of layers on the overall performance is investigated.

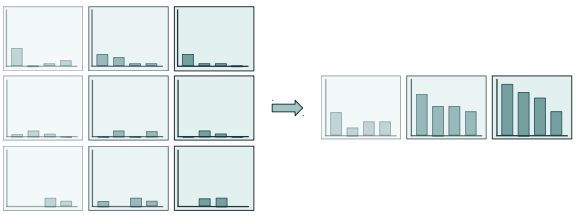

Figure 2: Merging deltas. Each received update (on the left) is integrated with the most recent complete version (on the right).

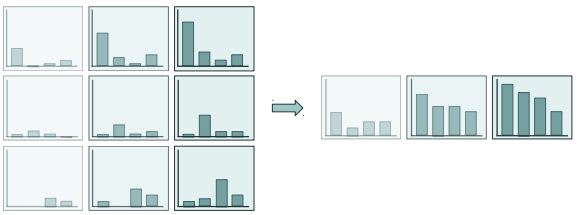

Figure 3: Merging entire objects. Each received object (on the left) replaces the previous one from the same data source.

4.1. Timespan of input objects

Objects which are meant to be merged might contain either all data points recorded in a data taking run (Fig. 3) or only new data points, which were not sent previously (Fig. 2).

Merging entire objects requires a few safety measures. First of all, taking into account certain randomness of the data transport in distributed processing systems, the merger operation should not be performed as soon as an object is received, since it might result in merging an object from the same data source twice. This implies having a cache with the latest versions of incomplete objects, which should come from distinguishable sources, so they are correctly replaced. Thanks to this cache, the Mergers can still make use of the last valid object if a data source stops sending new objects. However, the cache requires at least the amount of memory needed by an input object multiplied by the number of data sources, which might significantly increase the total memory usage.

An entire objects Merger is not required to be running as soon as any data source start sending objects. Because of that, if a Merger quits abruptly due to some kind of an unexpected event, it can fully recover and publish a valid result just after it receives incomplete objects from all data sources. Also, if the transport layer fails to deliver one of the input objects, the contained data is not lost irreversibly - it is present in succeeding objects. However, if a data source restarts and it does not have a recovery mechanism, it will lose all the data points it has gathered until that moment. As a consequence, that loss will be propagated to a Merger.

Merging *growing* objects is sub-optimal, as it requires also non-recent data to be transferred and merged repeatedly (e.g. tables).

Merging deltas has certain advantages in terms of performance and simplicity compared to the other alternative. Incoming objects might be stored in cache if there are performance benefits due to merging multiple objects at once (see the investigation in Sec. 4.2). Then, the merger algorithm might be triggered periodically or after having cached a certain amount of input objects. In the latter case, one should make sure that the last objects in a data acquisition run are merged as well. Tracking their provenance is not necessary as there is no risk of receiving the same data-points twice. Otherwise, the use of cache does not add any value and it can be omitted, keeping the memory requirements lower. Merging objects as soon as they are received makes the CPU usage more evenly distributed across time.

In order to have the complete set of data-points merged, a Merger should receive and process each message sent by data sources. In case of an unexpected failure, the accumulated data should be retrieved by an additional recovery mechanism. On the other hand, if a data source restarts, a Merger still contains all data-points received so far.

When handling *growing* objects, hardware requirements are lower compared to the other alternative. Transferring deltas consumes less network bandwidth, caching them requires less memory and merging might use less CPU time. The exact scale of this effect depends on the data structure being used and the amount of data points. According to benchmarks performed

Table 1: The major differences between merging entire objects and deltas.

Property	Entire objects	Deltas
Supports objects which cannot be reset	yes	no
Lower resource usage for growing objects	no	yes
Recoverability of Merger by design	yes	no
Recoverability of data sources by design	no	yes
No need for distinguishable data sources	no	yes
No need for cache	no	yes
Merging can be performed at object arrival	no	yes

on classes available in ROOT [9], *fixed-size* objects (such as TH1I, TH2I, TH3I) do not indicate any performance difference in relation to object timespan. Merging TTrees is more efficient when receiving only new table rows and so is merging sparse histograms with less data points. According to the performed benchmarks, merging 100 times less entries at a time translates to a 75 times less CPU usage for TTrees and 62 less for THnSparseI. However, one should remember that the size of sparse histograms and their merging performance depends significantly on patterns of the contained data.

It should also be noted that in multi-layer Merger topologies all the merging steps excluding the final layer (i.e. the one final Merger) should reset their objects upon sending them to the succeeding layer to avoid duplicating data points.

The summary of differences between the two discussed alternatives of object timespans is presented in Tab. 1. It was decided that deltas should be merged by default, unless it is required otherwise due to specific properties of data structures being used.

4.2. Merging collections

Some of the ROOT data types expect an object collection as an input for merging. In that case one can consider worthiness of caching the incoming objects (larger RAM usage) in order to decrease CPU consumption.

Fig. 4 contains benchmark results of the commonly used data structures in the O^2 QC. The measurements were obtained on a server with a dual Intel(R) Xeon(R) CPU E5-2640 v3 @ 2.60 GHz and 128 GB of RAM. The software was compiled with GCC v7.3.0 and optimisation settings set to *-O2*. The size of all the objects was tuned to 250 kB, except for `THnSparseI`, whose size depends mostly on the pseudo-random values used to fill it, thus cannot be determined a priori. The number of merged objects was always the same and they consisted of the same number of data points, but the objects were arranged in collections of different sizes. The standard histogram structures do not indicate any performance benefit, since the addition of corresponding bin counters is anyway the main bottleneck. Sparse histograms show a slight performance benefit - a collection of 1024 objects is merged 24% faster than 1024 individual ones. No improvement was observed in case of TTrees.

The presented results show that merging the popular ROOT types in collections does not provide any benefits in most cases. The observed speed-up for sparse histograms does not balance out the costs of required RAM for the object cache. Therefore, the O^2 Mergers do not support merging collections.

4.3. Multiple layers of Mergers

To estimate the number of Merger layers needed to sustain a required input, a model based on the queuing theory [10] is proposed. The M/D/1 queue can model the a Merger process assuming that the timing for incoming messages is determined by a Poisson process, the merging time is fixed and one incoming object is handled at a time. In the present context, the utilisation factor

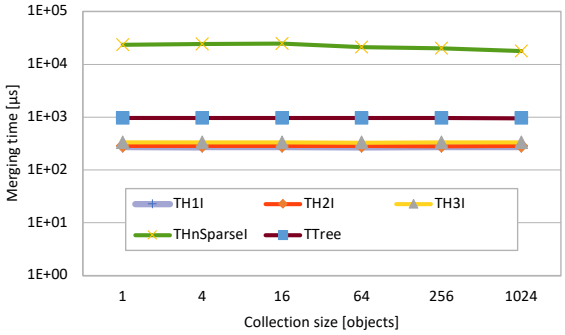

Figure 4: Merging time per object as a function of the number of objects in a collection.

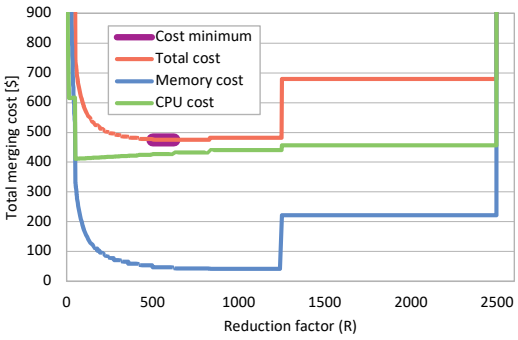

Figure 5: An example of cost functions in eq. 9. The total cost minimum lies within $R \in [500, 624]$.

takes the following forms:

$$\rho_i = \frac{\lambda}{\mu(R_i)} = \frac{R_i \lambda_s}{\mu(R_i)} \tag{5}$$

where: i - number of layer, R_i - reduction factor in i-th layer, λ - total object arrival rate, λ_s - object arrival rate from a single source, $\mu(R)$ - objects merging rate with respect to the reduction factor. If the condition $\rho < 1$ is not respected, then a message queue grows infinitely (Merger is too slow) and the proceeding formulas do not apply. The $\mu(R)$ component becomes a constant if a Merger performance is independent of the number of inputs.

The average number of messages in a queue, which corresponds to the average memory usage, can be found with eq. 6.

$$Q_i = \frac{\rho_i^2}{2(1-\rho_i)} \tag{6}$$

In total, one Merger and its queue contain approximately eq. 7 entire objects or eq. 8 deltas.

$$A_i = Q_i + \rho_i + R_i \tag{7}$$

$$A_i = Q_i + \rho_i + 1 \tag{8}$$

The ρ_i component in both formulas corresponds to an average processing time when an additional object is handled. In case of the merging of entire objects, the cache contains R_i entries, while the delta Merger stores only one, complete object. One should remember that estimating the used memory amount by multiplying A_i with object size will only apply if it is *fixed*. One can use the presented formulas to roughly estimate the amount of memory used by one Merger and to arbitrarily determine if one is actually enough.

To further investigate the performance dependence on the reduction factor and find an optimal value, approximate cost functions are formulated. If it is assumed that only *fixed-size* objects are merged, the total cost of CPU and memory usage with respect to R take the forms below:

$$C(R) = C_P(R) + C_M(R) = c_p \sum_{i=1}^{L} M_i \rho_i + c_m \cdot o_{size} \sum_{i=1}^{L} M_i A_i \tag{9}$$

where c_p - cost of a CPU core, c_m - cost of RAM per memory unit, o_{size} - object size. The eq. 9 should be minimised in order to find the optimal reduction factor.

An example with the following parameters is considered. There are 2500 data sources, each produces a 500 MB delta object each minute. The performance of a Merger when handling these objects follows a linear function $\mu(R_i) = -0.002R_i + 24$ in range [2, 2500]. A CPU core costs 118 \$, while the cost of RAM is 6.25 \$/GB. Fig. 5 presents the cost functions with these parameters. For small reduction factors, the memory usage is high due to the larger number

of processes. It again rises for $R > 1250$ when Mergers become highly occupied, resulting in message longer queues. CPU usage cost grows starting with $R = 14$ due to Merger performance becoming worse with larger numbers of supported inputs. Discontinuities and flat periods in the cost function come from the number of layers and Mergers being integer values. Therefore, the minimum resides in the range $[500, 624]$, because it implies having 2 layers with 5 and 1 Mergers, respectively. This example shows that sometimes having more processes in a topology might require less resources in total.

4.4. Other notes

In case of long data acquisition runs in the ALICE experiment, Mergers might need to operate for more than a dozen hours. Thus, even small memory leaks might eventually become very apparent and burdensome. Mergers are particularly prone to such problems, as they handle multiple objects with different structures and if they merge deltas, they cannot reset without any consequences. When using collection types available in the ROOT framework, one should pay attention to ownership of data they contain, which was also indicated in [11]. If a collection was serialised as non-owning, then sent as a message and deserialised in another process, it will not own the stored objects - they should be explicitly deleted after being merged. Browsing collections recursively will ensure that also nested containers are properly deleted.

5. Conclusion

The new distributed computing system in the ALICE experiment will have to cope with many data processing challenges, one of which is merging large amounts of incomplete data structures into complete objects. The version 1.0 of the O^2 Mergers has been reached after investigating a number of options, both by benchmarking and qualitative comparisons.

The paper contains an analysis of the merging aspects in distributed message-passing systems. Two alternatives: merging the entire objects or only deltas, were discussed. Their benefits and drawbacks in terms of CPU, memory and bandwidth usage, but also reliability and complexity were presented. A possibility to merge collections of objects was considered - it was shown to be slightly more efficient for sparse ROOT histograms. In case of particularly demanding objects which might require multi-layer Merger topologies, the optimal arrangement of merging processes was found by taking advantage of the presented estimated cost functions.

The authors hope that this research will prove useful when designing and implementing similar merger applications and help to avoid potential problems due to the described corner cases.

References

[1] ALICE Collaboration (ALICE) 2008 *JINST* **3** S08002
[2] Evans L and Bryant P 2008 *Journal of Instrumentation* **3** S08001–S08001 URL https://doi.org/10.1088%2F1748-0221%2F3%2F08%2Fs08001
[3] The ALICE Collaboration 2014 *Journal of Physics G: Nuclear and Particle Physics* **41** 087001 URL http://stacks.iop.org/0954-3899/41/i=8/a=087001
[4] The ALICE Collaboration 2015 Technical design report for the upgrade of the online–offline computing system Tech. rep. CERN
[5] The ALICE Collaboration 2019 *Computer Physics Communications* **242** 25 – 48 ISSN 0010-4655 URL http://www.sciencedirect.com/science/article/pii/S0010465519301250
[6] Conde Muino P 2005 *14th International Conference on Computing in High-Energy and Nuclear Physics* pp 111–114
[7] Renkel P and ATLAS 2010 *Journal of Physics: Conference Series* **219** 022043 URL https://doi.org/10.1088%2F1742-6596%2F219%2F2%2F022043
[8] van Herwijnen et al 2006 Control and monitoring of on-line trigger algorithms using a SCADA system Tech. rep. CERN Geneva cHEP-2006, Volume I and II, langid = URL https://cds.cern.ch/record/1442984
[9] 1996 *ROOT - An Object Oriented Data Analysis Framework* URL http://root.cern.ch/
[10] Cooper R B 1981 *Introduction to Queueing Theory* 2nd ed (North Holland) ISBN 0-444-00379-7
[11] Krzewicki M Real-time ROOT object merging in the HLT URL http://cern.ch/go/R9X9